ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ

ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ

ΑΛΓΟΡΙΘΜΟΙ

ΔΙΔΑΣΚΩΝ

ΚΩΣΤΑΣ ΚΟΛΟΜΒΑΤΣΟΣ

Στόχοι του Μαθήματος

- Ανάπτυξη αναλυτικής σκέψης, συνθετικής και κριτικής ικανότητας
- Ανάπτυξη και καλλιέργεια της δημιουργικότητας στο σχεδιασμό λύσεων και στην αναζήτηση εναλλακτικών λύσεων
- Εισαγωγή στις θεμελιώδεις αλγοριθμικές έννοιες και τεχνικές
- Παρουσίαση σημαντικών αλγορίθμων και ανάλυσή τους

Περιεχόμενο Μαθήματος

Εισαγωγή στους Αλγορίθμους

- Αλγοριθμική Επίλυση Προβλημάτων, Σημαντικά Προβλήματα

Ανάλυση Αλγορίθμων

- Πλαίσιο Ανάλυσης Αλγορίθμων, Συμβολισμοί και Πολυπλοκότητα, Εμπειρική Ανάλυση Αλγορίθμων
- Μαθηματική Ανάλυση μη Αναδρομικών Αλγορίθμων, Μαθηματική Ανάλυση Αναδρομικών Αλγορίθμων

Μεθοδολογίες

- Δυναμικός Προγραμματισμός, Άπληστες Μέθοδοι
- Brute Force and Exhaustive Search
- Decrease and Conquer, Divide and Conquer, Transform and Conquer
- Travelling Salesman Problem, Knapsack Problem, Assignment Problem

Αλγόριθμοι Ταξινόμησης

- Selection Sort, Bubble Sort, Insertion Sort, Median Sort, Quicksort, Heap Sort, Counting Sort, Bucket Sort

Αλγόριθμοι Αναζήτησης

- Σειριακή Αναζήτηση, Δυαδική Αναζήτηση, Hash-based Αναζήτηση

Αλγόριθμοι Γράφων

- Shortest Paths, Minimum Spanning Tree, Maximum Flow

Αλγόριθμοι Δένδρων

- Depth-first Search, Breadth-first Search, A* Search

Χωρο-Χρονική Ανάλυση

- Ταξινόμηση με Καταμέτρηση, Ταίριασμα Συμβολοσειρών, Hashing, B-Δένδρα

Επαναληπτική Βελτιστοποίηση

- Γραμμικός Προγραμματισμός, Μέθοδος Simplex, Πρόβλημα Μέγιστης Ροής

Περιορισμοί Αλγορίθμων

- P, NP και NP-Complete Προβλήματα, Προσεγγιστικοί Αλγόριθμοι

Βιβλιογραφία

- Cormen, T., Leiserson, C., Rivest, R., Stein, C., Introduction to Algorithms', 3rd Edition, The MIT Press, 2009
  - Το βιβλίο έχει εκδοθεί στα Ελληνικά από το ΙΤΕ-Πανεπιστημιακές Εκδόσεις Κρήτης (2012)
- Μποζάνης, Π., 'Αλγόριθμοι', Εκδόσεις Τζιόλα, 2006
- Levitin, A., 'Introduction to the Design and Analysis of Algorithms', 3rd Edition, Pearson Education Inc. publishing, 2012

ΕΙΣΑΓΩΓΗ

Αλγόριθμοι

- Τι είναι αλγόριθμοι;
- Σε τι μας χρησιμεύει η μελέτη τους;
- Ποιος είναι ο ρόλος τους σε σχέση με άλλες τεχνολογίες που χρησιμοποιούνται στους υπολογιστές;

Ορισμός Αλγορίθμου (1/2)

Knuth

- Ένας αλγόριθμος είναι μια πεπερασμένη, συγκεκριμένη, αποτελεσματική διαδικασία, με μια είσοδο και κάποια έξοδο

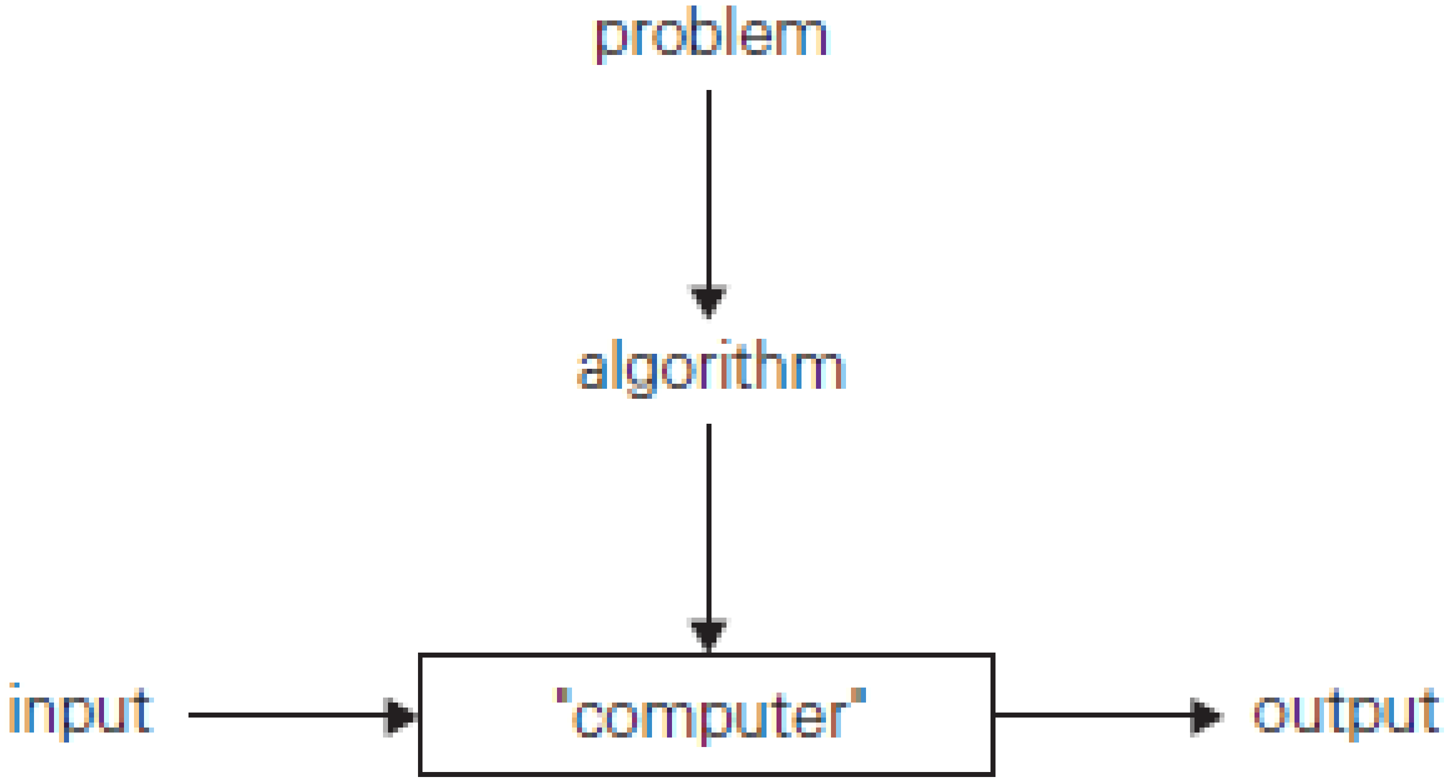
Schneider, M. and J. Gersting (1995), *An Invitation to Computer Science*, West Publishing Company, New York, NY, p. 9.

- Ένας αλγόριθμος είναι μια καλά ορισμένη ακολουθία σαφών και υπολογιστικά αποδοτικών λειτουργιών / βημάτων οι οποίες όταν εκτελεστούν παράγουν ένα αποτέλεσμα και τερματίζουν σε πεπερασμένο χρονικό διάστημα

Ο όρος ***αλγόριθμος*** αναφέρεται σε οποιαδήποτε καλά ορισμένη υπολογιστική διαδικασία που δέχεται κάποια τιμή ή κάποιο σύνολο τιμών ως ***είσοδο*** και δίνει κάποια τιμή ή κάποιο σύνολο τιμών ως ***έξοδο***.

Ένας αλγόριθμος είναι μια ακολουθία υπολογιστικών βημάτων που μετασχηματίζει την είσοδο στην έξοδο.

Ορισμός Αλγορίθμου (2/2)



Παράδειγμα Αλγορίθμου (1/2)

Ταξινόμηση μιας ακολουθίας αριθμών

- Είσοδος: η ακολουθία των αριθμών ($α_1$, $α_2$, $α_3$, …, $α_N$)

- Έξοδος: μια αναδιάταξη των αριθμών της εισόδου ($α'_1$, $α'_2$, $α'_3$, …, $α'_N$) τέτοια ώστε ($α'_1 <= α'_2 <= α'_3 <= … <= α'_N$) για αύξουσα σειρά ή ($α'_1 >= α'_2 >= α'_3 >= … >= α'_N$) για φθίνουσα σειρά

- Για την ακολουθία (12, 3, 24, 35, 6, 32) θα πάρουμε (3, 6, 12, 24, 32, 35)

Παράδειγμα Αλγορίθμου (2/2)

Αλγόριθμος του Ευκλείδη

- Εύρεση μέγιστου κοινού διαιρέτη
- Είσοδος: δύο θετικοί ακέραιοι αριθμοί
- Έξοδος: ο μεγαλύτερος ακέραιος που διαιρεί ακριβώς τους δύο αριθμούς που παίρνουμε σαν είσοδο
- Βασίζεται στην εφαρμογή πολλαπλών εκτελέσεων της ισότητας

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

- Μέχρι να έχουμε m mod n = 0
- Παράδειγμα

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

ALGORITHM *Euclid*(m, n)

//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return m

Ορθοί Αλγόριθμοι

Στιγμιότυπο του προβλήματος δηλώνει την είσοδο η οποία απαιτείται για να υπολογιστεί μια λύση του προβλήματος (εννοείται ότι η είσοδος αυτή θα ικανοποιεί όλους τους περιορισμούς που επιβάλλει η διατύπωση του προβλήματος)

Ένας αλγόριθμος θεωρείται **σωστός / ορθός** να για κάθε στιγμιότυπο εισόδου, ο αλγόριθμος τερματίζει δίνοντας σωστή έξοδο

Ένας **μη ορθός** αλγόριθμος μπορεί να μην τερματίζει καν σε ορισμένα στιγμιότυπα εισόδου, ή μπορεί να τερματίζει δίνοντας αποτέλεσμα διαφορετικό από το ζητούμενο

Διάσταση Προβλήματος

Παραδείγματα

- Πλήθος αριθμών n
- Πλήθος αντικειμένων n

Επηρεάζει την επίδοση ενός αλγορίθμου

Εφαρμογές Αλγορίθμων

Οι αλγόριθμοι βρίσκονται στην καρδιά κάθε υπολογιστικής διαδικασίας

- Δίκτυα επικοινωνιών
- Διαδίκτυο και Παγκόσμιος Ιστός
- Τεχνητή Νοημοσύνη
- Μεταγλωττιστές
- Βάσεις Δεδομένων
- Ανάλυση Δεδομένων
- Γραφικά Υπολογιστών
- Επεξεργασία Σημάτων
- Επιστημονικός Υπολογισμός
- Ηλεκτρονικό Εμπόριο
- ……..

Συγγραφή Αλγορίθμων

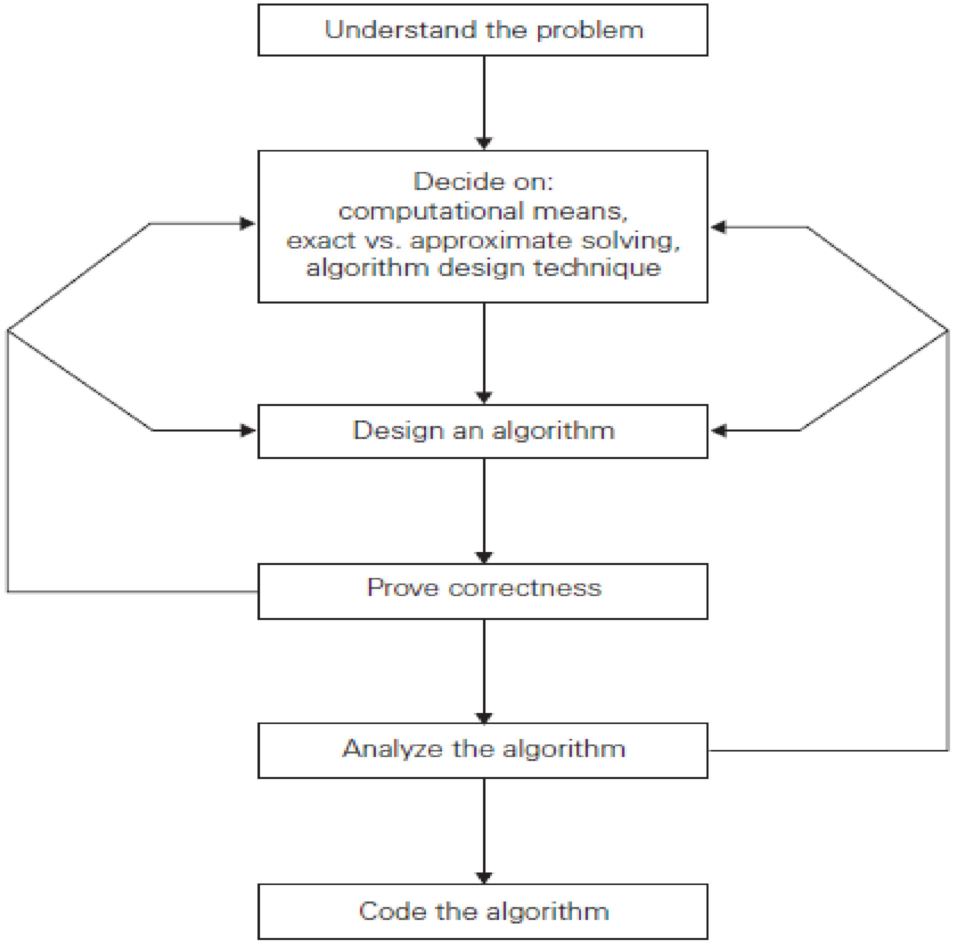
Πολλαπλές επιλογές

- Χρήση φυσικής γλώσσας (Ελληνικά – Αγγλικά - …)
  - Μειονέκτημα: χρήση πολλών λέξεων
- Χρήση γλωσσών προγραμματισμού
  - Μειονέκτημα: απαιτείται η γνώση της γλώσσας προγραμματισμού στην οποία έχει γραφτεί ο αλγόριθμος
- Ψευδοκώδικας
- Διαγράμματα ροής δεδομένων

Βασικές Έννοιες Επίλυσης Προβλημάτων

Βήματα επίλυσης:

- Πλήρης κατανόηση του προβλήματος
- Επιλογή μεθόδου επίλυσης
- Σχεδιασμός αλγορίθμου
- Απόδειξη ορθότητας / εγκυρότητας
- Ανάλυση του Αλγορίθμου
- Κωδικοποίηση



Υπολογιστικές Μηχανές - Υλικό

- Μετά την κατανόηση του προβλήματος θα πρέπει να διαπιστώσουμε τις δυνατότητες του Υλικού
- Οι περισσότεροι αλγόριθμοι έχουν σχεδιαστεί για μηχανές αρχιτεκτονικής von Neumann
- Βασική θέση στις αρχιτεκτονικές αυτές έχει η λεγόμενη **Random Access Machine (RAM)**
- Βασική υπόθεση: οι εντολές εκτελούνται **ακολουθιακά**
- Οι αντίστοιχοι αλγόριθμοι ονομάζονται **ακολουθιακοί**
- Οι σύγχρονες εξελίξεις στο Υλικό βοηθούν τη δημιουργία παράλληλων αλγορίθμων (οι λειτουργίες εκτελούνται παράλληλα)

Επιλογή Μεθόδου Επίλυσης

Επακριβής επίλυση του προβλήματος
- exact algorithms

Προσεγγιστική επίλυση
- approximation algorithms

Υπάρχει πλήθος προβλημάτων που δεν λύνονται επακριβώς (π.χ. τετραγωνικές ρίζες, μη γραμμικά συστήματα, ολοκληρώματα)

Οι διαθέσιμοι αλγόριθμοι για την επίλυση προβλημάτων μπορεί να είναι πολύ αργοί λόγω **υψηλής πολυπλοκότητας**

Τεχνικές Σχεδίασης Αλγορίθμων

Κρίσιμο ερώτημα: Πως σχεδιάζουμε ένα αλγόριθμο για ένα δοσμένο πρόβλημα;

Τεχνική σχεδίασης αλγορίθμων

- *μια γενική προσέγγιση για την αλγοριθμική επίλυση προβλημάτων που είναι εφαρμόσιμη σε μια ποικιλία προβλημάτων σε διάφορες περιοχές της Επιστήμης των Υπολογιστών*

Ένα σύνολο τεχνικών θα παρουσιαστούν στα πλαίσια του μαθήματος

Κάποιες τεχνικές μπορεί να είναι ανεφάρμοστες

Συνδυασμός τεχνικών

Αλγόριθμοι και Δομές Δεδομένων

- Δίνουμε έμφαση στην επιλογή των απαραίτητων δομών δεδομένων
- Παράδειγμα: ο αλγόριθμος του Ερατοσθένη για την εξαγωγή των πρώτων αριθμών που δεν ξεπερνούν ένα δοσμένο n > 1 (***sieve of Eratosthenes***) διαρκεί περισσότερο αν υιοθετηθούν συνδεδεμένες λίστες αντί για απλούς πίνακες
- Μεγαλύτερη σημασία λόγω των αντικειμενοστραφών γλωσσών προγραμματισμού

Sieve of Eratosthenes (1/2)

- Αρχικοποιεί μια λίστα από υποψήφιους πρώτους αριθμούς με συνεχόμενους αριθμούς από 2 μέχρι n
- Στο πρώτο iteration εξαλείφει όλα τα πολλαπλάσια του 2, 4, 6, …
- Προχωρά στο επόμενο στοιχείο 3 και εξαλείφει τα πολλαπλάσιά του
- Δεν χρειάζεται εξέταση του 4
- Το επόμενο στοιχείο είναι το 5
- Συνεχίζει μέχρι να μην υπάρχει αριθμός που να μπορεί να εξαλειφθεί

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	3		5		7				11		13				17		19				23		25
2	3		5		7				11		13				17		19				23		

Sieve of Eratosthenes (2/2)

```
ALGORITHM  Sieve(n)
    //Implements the sieve of Eratosthenes
    //Input: A positive integer n > 1
    //Output: Array L of all prime numbers less than or equal to n
    for p ← 2 to n do A[p] ← p
    for p ← 2 to ⌊√n⌋ do      //see note before pseudocode
        if A[p] ≠ 0            //p hasn't been eliminated on previous passes
            j ← p ∗ p
            while j ≤ n do
                A[j] ← 0       //mark element as eliminated
                j ← j + p
    //copy the remaining elements of A to array L of the primes
    i ← 0
    for p ← 2 to n do
        if A[p] ≠ 0
            L[i] ← A[p]
            i ← i + 1
    return L
```

Απόδειξη Ορθότητας

- Όταν οριστεί ένας αλγόριθμος θα πρέπει να αποδείξουμε την ***ορθότητα του*** (***correctness***)
- Ο αλγόριθμος επιστρέφει αποτέλεσμα για κάθε είσοδο σε πεπερασμένο χρονικό διάστημα
- Μπορεί να είναι πολύπλοκη διαδικασία
- Τυπική προσέγγιση: **μαθηματική επαγωγή**
- Οι επαναλήψεις ενός αλγορίθμου προσφέρουν μια ‘φυσική’ αλληλουχία βημάτων για την υιοθέτηση της επαγωγής
- Στους προσεγγιστικούς αλγορίθμους αποδεικνύουμε ότι το σφάλμα είναι κάτω από ένα συγκεκριμένο όριο

Ανάλυση Αλγορίθμων

Αποδοτικότητα (efficiency)
- Μπορεί να οριστεί με μαθηματικό τρόπο

Time efficiency
- Πόσο γρήγορα ‘τρέχει’ ένας αλγόριθμος

Space efficiency
- Πόση επιπλέον μνήμη χρειάζεται

Απλότητα (simplicity)
- Είναι υποκειμενική (απλοί αλγόριθμοι μπορούν να γίνουν αντιληπτοί πιο εύκολα)

Γενικότητα (Generality)
- Γενικότητα του προβλήματος που επιλύει ο αλγόριθμος
- Το σύνολο των δεδομένων που δέχεται

Κωδικοποίηση

Μεγάλη προσοχή στο πέρασμα από τον αλγόριθμο στο πρόγραμμα

- Κίνδυνος λανθασμένης κωδικοποίησης
- Κίνδυνος μη αποδοτικής κωδικοποίησης

Επαλήθευση (verification)

Εγκυρότητα (validity)

- Πολλαπλά και εξαντλητικά τεστ
- Τεχνικές testing και debugging

Σημαντικά Προβλήματα

- Ταξινόμηση
- Αναζήτηση
- Επεξεργασία Αλφαριθμητικών
- Προβλήματα Γράφων
- Συνδυαστικά Προβλήματα
- Γεωμετρικά Προβλήματα
- Αριθμητικά Προβλήματα

Αλγόριθμοι Ταξινόμησης

Αναδιάταξη αντικειμένων μιας δοσμένης λίστας

Τα αντικείμενα πρέπει να επιδέχονται μιας διάταξης

Για εγγραφές πρέπει να επιλέξουμε ένα πεδίο ως προς το οποίο θα γίνεται η ταξινόμηση: **κλειδί (key)**

Stability

- Ο αλγόριθμος διατηρεί τη σχετική σειρά οποιονδήποτε δύο ίσων αντικειμένων στην είσοδο του

In place

- Ο αλγόριθμος δεν απαιτεί επιπλέον μνήμη

Αλγόριθμοι Αναζήτησης

- Εύρεση μιας τιμής (**search key**) σε ένα σύνολο αντικειμένων
- Σειριακή, Δυαδική αναζήτηση, αλγόριθμοι αναπαράστασης των δεδομένων σε διαφορετική μορφή εύκολα ‘αναζητήσιμη’
- Δεν υπάρχει το πρόβλημα της σταθερότητας
- Σε δυναμικά περιβάλλοντα με συνεχείς αλλαγές των δεδομένων οι αλγόριθμοι αναζήτησης πρέπει να συνδυάζονται με ενέργειες προσθαφαίρεσης δεδομένων
- Προσεκτική επιλογή δομών δεδομένων

Επεξεργασία Αλφαριθμητικών

- Ένα αλφαριθμητικό (string) είναι μια ακολουθία από χαρακτήρες ενός αλφαβήτου
- Ακολουθίες χαρακτήρων, δυαδικών ψηφίων, ακολουθίες γονιδίων, κ.λπ.
- **String matching**
  - Παράδειγμα: Αναζήτηση μιας λέξης μέσα σε ένα κείμενο

Αλγόριθμοι Γράφων

- Ένας **γράφος (graph)** είναι ένα σύνολο **κόμβων (vertices)** από τους οποίους κάποιοι είναι συνδεδεμένοι μεταξύ τους μέσω **ακμών (edges)**
- **Κατευθυνόμενοι – Μη κατευθυνόμενοι**
- Μοντελοποιούν πλήθος προβλημάτων
- Βασικοί αλγόριθμοι
  - Διάσχιση γράφων
  - Συντομότερο μονοπάτι
  - Τοπολογική ταξινόμηση
- Κάποια προβλήματα είναι υπολογιστικά πολύπλοκα
  - Travelling salesman problem (shortest tour through n cities that visits every city exactly once)
  - Graph coloring problem (assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices have the same color)

Συνδυαστικά – Γεωμετρικά Προβλήματα

Συνδυαστικά προβλήματα

- Αναζητούν ένα συνδυαστικό αντικείμενο (π.χ., αναδιάταξη, συνδυασμό) το οποίο ικανοποιεί ένα σύνολο περιορισμών
- Πολύ δύσκολα προβλήματα
  - Επηρεάζονται από το μέγεθος του προβλήματος
  - Απαιτούν υψηλό χρόνο εκτέλεσης

Γεωμετρικά προβλήματα

- Σχετίζονται με σχήματα (γραμμές, σημεία, πολύγωνα)
- The closest-pair problem is self-explanatory: given n points in the plane, find the closest pair among them
- The convex-hull problem asks to find the smallest convex polygon that would include all the points of a given set

Αριθμητικά Προβλήματα

- Επίλυση εξισώσεων, υπολογισμός ολοκληρωμάτων, επίλυση συστημάτων εξισώσεων, κ.λπ.
- Πολλά προβλήματα αυτής της κατηγορίας μπορούν να επιλυθούν μόνο προσεγγιστικά
- Απαιτούν χειρισμό πραγματικών αριθμών οι οποίοι τυχαίνουν προσεγγιστικής επεξεργασίας από τους υπολογιστές

ΑΝΑΛΥΣΗ ΑΛΓΟΡΙΘΜΩΝ

Μέγεθος Εισόδου

Λογική παρατήρηση: η συντριπτική πλειοψηφία αλγορίθμων ‘τρέχει’ περισσότερο για μεγαλύτερο μέγεθος εισόδου

Μελετούμε την απόδοση των αλγορίθμων ως συνάρτηση μιας παραμέτρου **n** που υποδηλώνει το μέγεθος εισόδου

Παραδείγματα

- Μέγεθος λίστας στοιχείων για:
  - Ταξινόμηση
  - Αναζήτηση
  - Εύρεση μικρότερου – μεγαλύτερου
  - …

Μετρικές Μεγέθους Εισόδου

Χρήση μιας τυπικής χρονικής μονάδας (π.χ., second, millisecond)

- Εξάρτηση από το υλικό
- Εξάρτηση από την ποιότητα του αλγορίθμου / προγράμματος

Καλύτερη λύση: Μέτρηση του πλήθους των εκτελέσεων των λειτουργιών των αλγορίθμων

- Αναγνώριση των **βασικών λειτουργιών (basic operations)** – λειτουργίες που συνεισφέρουν περισσότερο στο χρόνο εκτέλεσης
- Μέθοδος: αναγνώριση της λειτουργίας που απαιτεί τον περισσότερο χρόνο στον πιο εσωτερικό βρόχο (loop)

Παράδειγμα – Ταξινόμηση - Βασική λειτουργία: Σύγκριση

Παράδειγμα

Έστω c_{op} είναι ο χρόνος εκτέλεσης μιας βασικής λειτουργίας

Έστω **C(n)** είναι το πλήθος των βασικών λειτουργιών που θα εκτελέσει ο αλγόριθμος

Η εκτίμηση του χρόνου εκτέλεσης είναι $T(n) \sim c_{op}\, C(n)$

Προσοχή: θα πρέπει να λάβουμε υπόψιν μας όλες τις λειτουργίες ενός αλγορίθμου

Υπόθεση: $C(n) = \frac{1}{2}n(n-1)$

Αν διπλασιάσουμε το μέγεθος n της εισόδου πόσο περισσότερο θα αυξηθεί ο χρόνος εκτέλεσης?

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2 \qquad \frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Ρυθμός Αύξησης

Ρυθμός αύξησης (rate of growth, order of growth) υποδηλώνει το ρυθμό αύξησης του χρόνου εκτέλεσης

Στον ακόλουθο πίνακα απεικονίζονται παραδείγματα ρυθμού αύξησης

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Περιπτώσεις Ανάλυσης

Χειρότερη περίπτωση (worst case efficiency)

- Η αποδοτικότητα για την είσοδο της χειρότερης περίπτωσης που είναι μια είσοδος μεγέθους n για την οποία ο αλγόριθμος απαιτεί τον **περισσότερο** χρόνο εκτέλεσης για όλες τις πιθανές εισόδους μεγέθους n
- Μας δίνει ένα άνω όριο εκτέλεσης
- Αναγνώριση της εισόδου που απαιτεί τον περισσότερο χρόνο εκτέλεσης

Μέση περίπτωση (average case efficiency)

- Υιοθετούμε υποθέσεις για το μέγεθος εισόδου n
- Πιθανοτικοθεωρητική ανάλυση

Καλύτερη περίπτωση (best case efficiency)

- Η αποδοτικότητα για την είσοδο της καλύτερης περίπτωσης που είναι μια είσοδος μεγέθους n για την οποία ο αλγόριθμος απαιτεί το **λιγότερο** χρόνο εκτέλεσης για όλες τις πιθανές εισόδους μεγέθους n

Σειριακή Αναζήτηση (1/4)

- Αναζητά το κλειδί K σε μια λίστα n στοιχείων
- Ελέγχει σειριακά την ύπαρξη του στοιχείου σε κάθε θέση της λίστας
- Επιστρέφει τη θέση στην οποία έχει βρεθεί το κλειδί αλλιώς επιστρέφει το -1
- Ο χρόνος εκτέλεσης μπορεί να είναι διαφορετικός ακόμα και για το ίδιο n

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element in A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Σειριακή Αναζήτηση (2/4)

- Χειρότερη περίπτωση:
  - **$C_{worst}(n) = n$**
- Καλύτερη περίπτωση:
  - **$C_{best}(n) = 1$**

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//          or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Σειριακή Αναζήτηση (3/4)

- Μέση περίπτωση:
  - Έστω p η πιθανότητα μιας επιτυχούς αναζήτησης (0 <= p <= 1)
  - Η πιθανότητα επιτυχούς αναζήτησης στη θέση i είναι ίδια για όλα τα i ίση με **p/n** και το πλήθος των αναζητήσεων είναι i
  - Σε περίπτωση ανεπιτυχούς αναζήτησης οι συγκρίσεις θα είναι n με πιθανότητα **1 – p**
  - Άρα

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

Σειριακή Αναζήτηση (4/4)

- Αν p = 1:
  - Συγκρίσεις: **(n+1)/2**
- Αν p = 0:
  - Συγκρίσεις: **n**

$$
\begin{aligned}
C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p) \\
&= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p) \\
&= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).
\end{aligned}
$$

Ασυμπτωτικός Συμβολισμός (1/2)

Συμβολισμός ασυμπτωτικού χρόνου εκτέλεσης

Χρήση: σύγκριση ρυθμού αύξησης

Τρεις συμβολισμοί:

- **Big oh (O)**
- **Big omega (Ω)**
- **Big theta (Θ)**

Ασυμπτωτικός Συμβολισμός (2/2)

$O(g(n))$ είναι το **σύνολο όλων των συναρτήσεων** με μικρότερο ή ίσο ρυθμό αύξησης του $g(n)$

Παραδείγματα

- $n \in O(n^2)$
- $100n + 5 \in O(n^2)$
- $\frac{1}{2}\, n(n-1) \in O(n^2)$

O-notation

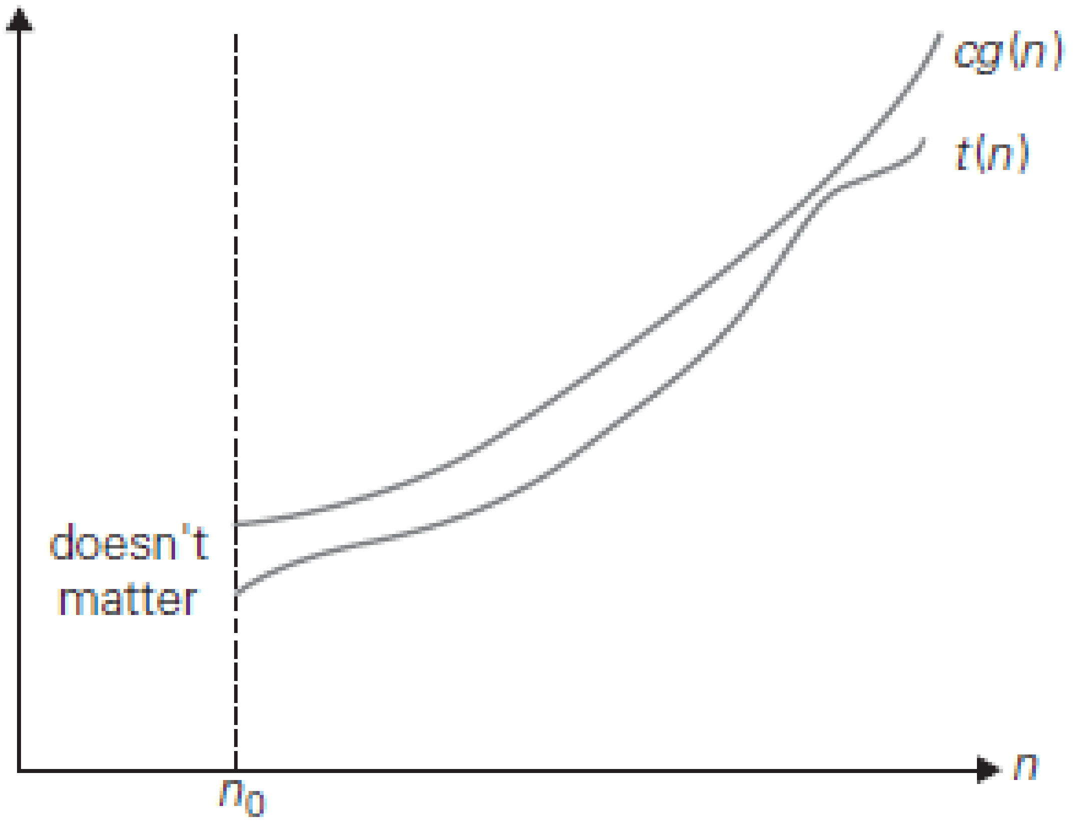
Μια συνάρτηση T(n) λέγεται ότι είναι O(g(n)) – T(n) є O(g(n)) – αν η T(n) έχει **άνω φράγμα** την g(n) επί ένα σταθερό αριθμό c.

$T(n) <= c\ g(n)$ για κάθε $n >= n_0$

Παραδείγματα:

$$100n + 5 \in O(n^2)$$

$$100n + 5 \le 100n + n \text{ (for all } n \ge 5) = 101n \le 101n^2$$



Ω-notation

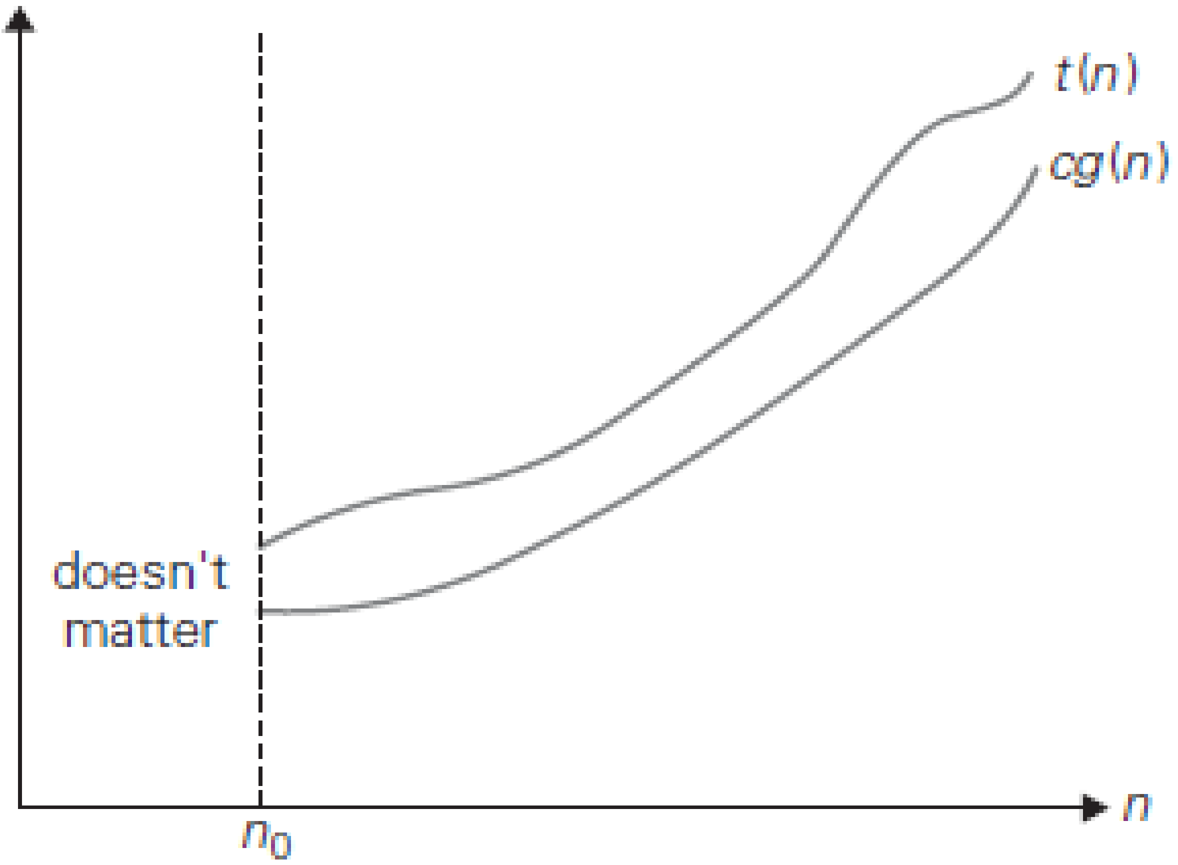
Μια συνάρτηση T(n) λέγεται ότι είναι Ω(g(n)) – T(n) ϵ Ω(g(n)) – αν η T(n) έχει **κάτω φράγμα** την g(n) επί ένα σταθερό αριθμό c.

T(n) >= c g(n) για κάθε n >= n_0

Παραδείγματα:

$n^3 \in \Omega(n^2)$

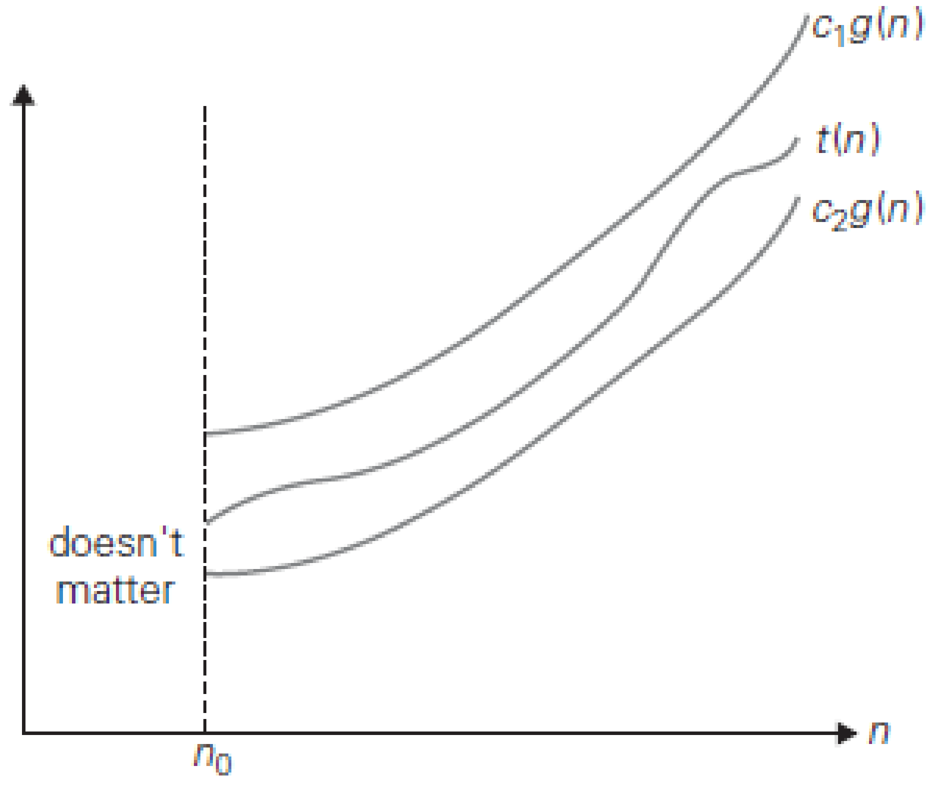
$n^3 \geq n^2 \quad \text{for all } n \geq 0$



Θ-notation (1/2)

Μια συνάρτηση T(n) λέγεται ότι είναι Θ(g(n)) – T(n) ϵ Θ(g(n)) – αν η T(n) έχει **κάτω και άνω φράγμα** πολλαπλάσια της g(n) για σταθερές c_1 και c_2.

c_2 g(n) <= T(n) <= c_1 g(n) για κάθε n >= n_0



Θ-notation (2/2)

Παραδείγματα:

$$\tfrac{1}{2}n(n-1) \in \Theta(n^2)$$

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \ge \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \ge 2) = \frac{1}{4}n^2.$$

o-notation

Μια συνάρτηση T(n) λέγεται ότι είναι o(g(n)) – T(n) ϵ o(g(n)) – αν η T(n) έχει **άνω φράγμα** την g(n) επί ένα σταθερό αριθμό c.

$T(n) < c\ g(n)$ για κάθε $n >= n_0$

Διαφορά με τον big oh notation

- Στο O(g(n)) η ανίσωση ισχύει για κάποιες σταθερές ενώ στο o(g(n)) η ανίσωση ισχύει για όλες τις σταθερές

Παράδειγμα

- $2n = o(n^2)$ αλλά $2n^2 \neq o(n^2)$

ω-notation

Μια συνάρτηση T(n) λέγεται ότι είναι o(g(n)) – T(n) ϵ o(g(n)) – αν η T(n) έχει **κάτω φράγμα** την g(n) επί ένα σταθερό αριθμό c.

$T(n) > c\, g(n)$ για κάθε $n >= n_0$

Διαφορά με τον big omega notation

- Στο Ω(g(n)) η ανίσωση ισχύει για κάποιες σταθερές ενώ στο ω(g(n)) η ανίσωση ισχύει για όλες τις σταθερές

Παράδειγμα

- $n^2/2 = \omega(n)$ αλλά $2n^2 \neq \omega(n^2)$

Θεωρήματα

Για οποιεσδήποτε δύο συναρτήσεις f(n) και g(n), έχουμε f(n) = Θ(g(n)) όταν και μόνο όταν f(n) = O(g(n)) και f(n) = Ω(g(n))

- Η απόδειξη εξάγεται εύκολα από τους ορισμούς

Χρήσιμο θεώρημα για ανάλυση αλγορίθμων με συνεχόμενα τμήματα

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Πολυπλοκότητα κατά Μέρη - Απόδειξη

Αφού $t_1(n) \in O(g_1(n))$ και $t_2(n) \in O(g_2(n))$

τότε

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1$$

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2$$

Θέτω $c_3 = \max\{c_1, c_2\}$ και $n \ge \max\{n_1, n_2\}$

Προσθέτω κατά μέλη

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\} \end{aligned}$$

Συνεπώς $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

για $2c_3 = 2\max\{c_1, c_2\}$ και $\max\{n_1, n_2\}$.

Χαρακτηριστικές Πολυπλοκότητες

1	Σταθερή πολυπλοκότητα
$\log(n)$	Λογαριθμική πολυπλοκότητα
$\log^k(n)$	Πολυλογαριθμική πολυπλοκότητα όταν k μία σταθερά
n	Γραμμική πολυπλοκότητα (πχ, $f(n) = an + b$, με a, b σταθερές)
$n \log n$	
n^2	Τετραγωνική πολυπλοκότητα
n^3	Κυβική πολυπλοκότητα
n^k	Πολυωνυμικη πολυπλοκότητα, με k μία σταθερά, δηλ. $f(x) = a_k n^k + \ldots + a_0$
a^n	Εκθετική πολυπλοκότητα $1 < a < 2$
2^n	Εκθετική πολυπλοκότητα
a^n	Εκθετική πολυπλοκότητα $a > 2$
$n!$	Παραγοντική πολυπλοκότητα
n^n	Υπερεκθετική πολυπλοκότητα

Ασυμπτωτική Συμπεριφορά

Αν

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = a \neq 0$$

τότε $f(n) = \Theta(g(n))$

Αν

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

τότε $f(n) = O(g(n))$ και $g(n) = \Omega(f(n))$ (με $f(n) \neq \Theta(g(n))$)

Αν

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \infty$$

τότε $g(n) = O(f(n))$ και $f(n) = \Omega(g(n))$ (με $g(n) \neq \Theta(f(n))$)

Συναρτήσεις Σύγκρισης (1/3)

Μεταβατικότητα (transitivity)

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)),$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)),$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)),$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)),$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)).$$

Συναρτήσεις Σύγκρισης (2/3)

Ανακλαστικότητα (reflexivity)

$$
\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}
$$

Συναρτήσεις Σύγκρισης (3/3)

Συμμετρία (symmetry)

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) \, .$$

Ανάστροφη συμμετρία (transpose symmetry)

$$\begin{aligned} f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) \, , \\ f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) \, . \end{aligned}$$

Πράξεις

$$cO(f(n)) = O(f(n)) \text{ με } c > 0$$

$$O(g(n)) + O(g(n)) = O(g(n))$$

$$O(g_1(n)) + O(g_2(n)) = O(\max\{g_1(n), g_2(n)\})$$

$$O(g_1(n)) * O(g_2(n)) = O(g_1(n) * g_2(n))$$

Ασκήσεις

Για τις ακόλουθες συναρτήσεις να βρείτε τον **επικρατέστερο όρο (dominant term)** καθώς και τον big oh notation συμβολισμό.

		$O(\ldots)$
$5 + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n\log_{10} n$		
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$		
$n^2 \log_2 n + n(\log_2 n)^2$		
$n\log_3 n + n\log_2 n$		
$3\log_8 n + \log_2 \log_2 \log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n\log_2 n + n(\log_2 n)^2$		
$100n\log_3 n + n^3 + 100n$		
$0.003\log_4 n + \log_2 \log_2 n$		

Λύσεις

		$O(\ldots)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10} n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log n)$
$n\log_3 n + n\log_2 n$	$n\log_3 n,\ n\log_2 n$	$O(n\log n)$
$3\log_8 n + \log_2 \log_2 \log_2 n$	$3\log_8 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n\log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	$O(n(\log n)^2)$
$100n\log_3 n + n^3 + 100n$	n^3	$O(n^3)$
$0.003\log_4 n + \log_2 \log_2 n$	$0.003\log_4 n$	$O(\log n)$

Υπολογισμός Πολυπλοκότητας (1/4)

Εξαρτάται από τις εντολές που υπάρχουν στον αλγόριθμο και το ‘είδος’ τους

Παραδείγματα:

A. Sequence of statements

Statement 1

Statement 2

…

Statement k

Ο συνολικός χρόνος προκύπτει ως το άθροισμα των χρόνων για κάθε εντολή

total time = time(statement 1) + time(statement 2) + ... + time(statement k)

Λύνεται με το θεώρημα που δείξαμε πιο πριν

Υπολογισμός Πολυπλοκότητας (2/4)

B. If – Then – Else

```
if (cond) then
    block 1 (sequence of statements)
else
    block 2 (sequence of statements)
end if;
```

Εδώ θα εκτελεστεί / εκτελείται είτε το block 1 ή το block 2

Η χειρότερη περίπτωση του αλγορίθμου εξαρτάται από τη χειρότερη περίπτωση από τα δύο blocks

Υπολογισμός Πολυπλοκότητας (3/4)

Γ. Loops

```
for I in 1 .. N
    sequence of statements
end;
```

Ελέγχουμε το πλήθος των εκτελέσεων του loop καθώς και την πολυπλοκότητα των εντολών μέσα στο loop

Παράδειγμα: αν το loop είναι O(N) και οι εντολές μέσα στο loop είναι O(1), τότε η πολυπλοκότητα είναι O(N)

Υπολογισμός Πολυπλοκότητας (4/4)

Γ. Nested Loops

```
for I in 1 .. N
  for J in 1 .. M
    sequence of statements
  end;
end;
```

Εδώ ελέγχουμε την πολυπλοκότητα του καθενός και υπολογίζουμε το γινόμενο

Προσοχή χρειάζεται όταν το πλήθος επαναλήψεων ενός loop εξαρτάται από το πλήθος ενός άλλου

Παράδειγμα: αν το εξωτερικό loop είναι O(N) και το εσωτερικό loop είναι O(M), τότε η πολυπλοκότητα είναι O(M * N)

Ασκήσεις

Υπολογίστε την πολυπλοκότητα του επόμενου αλγορίθμου

```
List<Integer> getBiggersList( int[] intArray )
{
   List<Integer> biggers = new LinkedList<Integer>();
   int sum = 0;
   for ( int item : intArray )
   {
          if ( item > sum )
             biggers.add( item );
       sum += item;
   }
   return biggers;
}
```

Λύσεις

O(n) διότι εξαρτάται μόνο από την επαναληπτική διαδικασία

Ασκήσεις

Υπολογίστε την πολυπλοκότητα των ακόλουθων αλγορίθμων

a.
```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        sequence of statements of O(1)
    }
}
```

b.
```
for (i = 0; i < N; i++) {
    for (j = i+1; j < N; j++) {
        sequence of statements of O(1)
    }
}
```

Λύσεις

(a) $O(N^2)$

(b) $O(N^2)$, διότι το πλήθος των επαναλήψεων είναι $(N-1)+(N-2)+....+0$

Χρήσιμοι Μαθηματικοί Τύποι (1/3)

Λογάριθμοι

$$\log_a 1 = 0$$

$$\log_a a = 1$$

$$\log_a x^y = y \log_a x$$

$$\log_a xy = \log_a x + \log_a y$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_a x = \frac{\log_b x}{\log_b a} = \log_a b \log_b x$$

Χρήσιμοι Μαθηματικοί Τύποι (2/3)

Αθροίσματα

$$\sum_{i=l}^{u} 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ times}} = u - l + 1 \ (l, u \text{ are integer limits, } l \le u); \quad \sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$$

$$\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \ (a \ne 1); \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma, \text{ where } \gamma \approx 0.5772\ldots \text{ (Euler's constant)}$$

$$\sum_{i=1}^{n} \lg i \approx n \lg n$$

Χρήσιμοι Μαθηματικοί Τύποι (3/3)

Ιδιότητες

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$$

$$\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

$$\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i, \text{ where } l \le m < u$$

$$\sum_{i=l}^{u}(a_i - a_{i-1}) = a_u - a_{l-1}$$

$$\int_{l-1}^{u} f(x)dx \le \sum_{i=l}^{u} f(i) \le \int_{l}^{u+1} f(x)dx \quad \text{for a nondecreasing } f(x)$$

$$\int_{l}^{u+1} f(x)dx \le \sum_{i=l}^{u} f(i) \le \int_{l-1}^{u} f(x)dx \quad \text{for a nonincreasing } f(x)$$

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$$

$$\lfloor x + n \rfloor = \lfloor x \rfloor + n \text{ and } \lceil x + n \rceil = \lceil x \rceil + n \quad \text{for real } x \text{ and integer } n$$

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

$$\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$$

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{as } n \to \infty \text{ (Stirling's formula)}$$

Ασκήσεις (1/7)

Να αποδειχθεί ο κανόνας της συμμετρίας: f(n) = Θ(g(n)) είναι ισοδύναμο με g(n) = Θ(f(n))

Απόδειξη:

Από τον ορισμό έχουμε $\exists c_1, c_2, n_0 > 0$ ώστε $0 \le c_1.g(n) \le f(n) \le c_2.g(n)$

Αφού c1, c2 > 0, διαιρούμε κατά μέλη. Συνεπώς:

$$0 \le \frac{1}{c_2}f(n) \le g(n) \text{ and } g(n) \le \frac{1}{c_1}f(n)$$

και

$$0 \le \frac{1}{c_2}f(n) \le g(n) \le \frac{1}{c_1}f(n)$$

με

$$k_1 = \frac{1}{c_2} \quad k_2 = \frac{1}{c_1}$$

Συνεπώς $0 \le k_2.f(n) \le g(n) \le k_1.f(n)$ for all $n \ge n_0$

Ασκήσεις (2/7)

Έστω $f(n) = ½\, n^2 – 3n$. Να αποδειχθεί ότι $f(n) = Θ(n^2)$

Απόδειξη:

Αρκεί να αποδείξουμε ότι υπάρχουν c_1, $c_2 > 0$ τέτοια ώστε

$$0 \le c_1.n^2 \le \frac{1}{2}n^2 - 3n \le c_2.n^2$$

Διαιρούμε με n^2 και έχουμε $$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

Παίρνουμε τρεις ανισώσεις

$$0 \le c_1$$

$$c_1 \le \frac{1}{2} - \frac{3}{n}$$

$$\frac{1}{2} - \frac{3}{n} \le c_2$$

Ασκήσεις (3/7)

Η 1η ανίσωση είναι προφανής για οποιοδήποτε c_1

Η δεύτερη ικανοποιείται για το μικρότερο n που την κάνει θετική. Αυτό είναι το 7. Το δεξιό μέρος γίνεται 1/14 για $c_1 = 1/14$ και $n_0' = 7$

Η τρίτη ικανοποιείται για $n_0'' = 1$ και $c_2 = 1$

Συνεπώς, $c_1 = 1/14$, $c_2 = 1$ και $n_0 = \max\{n_0', n_0''\} = 7$

$$0 \leq c_1$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n}$$

$$\frac{1}{2} - \frac{3}{n} \leq c_2$$

Ασκήσεις (4/7)

Να διαπιστώσετε αν $\frac{1}{1000}n^3 = O(1000n^2)$

Απόδειξη

Έστω ότι ισχύει. Σε αυτή την περίπτωση έχουμε:

$$\frac{1}{1000}n^3 \leq c.1000n^2$$

Οπότε $\forall n \geq n_0$:

είναι $\frac{1}{1000}n \leq 1000.c \Leftrightarrow n \leq 1000000.c$

Το οποίο είναι ψευδές

Ασκήσεις (5/7)

Έστω πολυώνυμο p(n) βαθμού k τέτοιο ώστε ο συντελεστής του μεγιστοβάθμιου όρου να είναι θετικός. Να δείξετε ότι $p(n) = \Theta(n^k)$

Απόδειξη

Αρκεί να αποδείξουμε ότι για κάθε $n \geq n_0,\ 0 \leq c_1 n^k \leq p(n) \leq c_2 n^k$

Δηλαδή αρκεί να αποδείξουμε ότι

$$c_1 n^k \leq a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} \ldots + a_1 n + a_0 \leq c_2 n^k$$

Διαιρώντας με n^k παίρνουμε

$c_1 \leq a_k + \underbrace{\frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \ldots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}}_{T} \leq c_2$ με $\lim_{n \to \infty} T = 0$. Άρα $0 < c_1 < a_k \quad c_2 > a_k$

Ασκήσεις (6/7)

Αν $a \in \mathbb{R}$ και $b \in \mathbb{R}^+$. Να αποδείξετε ότι $(n+a)^b \approx n^b$

Παρατηρούμε ότι

$n + a \leq n + |a| \leq 2n,$ με $n \geq |a|$

$n + a \geq n - |a| \geq \frac{n}{2}$ με $\frac{n}{2} \geq |a|,\ n \geq 2|a|$

Έχουμε

$$\frac{1}{2}n \leq n + a \leq 2n, \text{ if } n \geq 2|a|$$

Υψώνοντας στη b^{th} παίρνουμε

$$\left(\frac{1}{2}\right)^b n^b \leq (n+a)^b \leq 2^b n^b$$

Έτσι

$$c_1 = \left(\tfrac{1}{2}\right)^b,\ c_2 = 2^b,\ n_0 = \lceil 2|a| \rceil$$

Ασκήσεις (7/7)

Να αποδειχθεί ότι $\sqrt[n]{n} = \Theta(1)$

Απόδειξη

Ισχύει

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

και $\sqrt[n]{n} = e^{\ln \sqrt[n]{n}} = e^{\left(\frac{\ln n}{n}\right)}$

Οπότε $e^{\left(\frac{\ln n}{n}\right)} = 1 + \underbrace{\frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \ldots}_{T(n)}$

Όμως $S(x) = \frac{\ln x}{x} + \frac{\ln^2 x}{x^2} + \frac{\ln^3 x}{x^3} + \ldots, \qquad \lim_{x\to\infty} S(x) = 0$

και συνεπώς $\lim_{n\to\infty} T(n) = 0$

Άρα $\lim_{n\to\infty} \sqrt[n]{n} = 1$

Μη Αναδρομικά Προβλήματα

Εύρεση Μέγιστης Τιμής (1/2)

Προσπαθούμε να βρούμε το μέγιστο ανάμεσα σε n αριθμούς

Η προφανής μετρική είναι το μέγεθος / πλήθος των αριθμών n

Δύο λειτουργίες είναι αυτές που θα εκτελεστούν μέσα στο for:

- Η σύγκριση
- Η ανάθεση τιμής στο μέγιστο

Η βασική λειτουργία είναι η **σύγκριση**

Αφού η σύγκριση θα εκτελεστεί n-1 φορές δεν υπάρχει ανάγκη για διάκριση καλύτερης, μέσης, χειρότερης περίπτωσης

ALGORITHM *MaxElement*$(A[0..n-1])$

 //Determines the value of the largest element in a given array
 //Input: An array $A[0..n-1]$ of real numbers
 //Output: The value of the largest element in A
 $maxval \leftarrow A[0]$
 for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
   $maxval \leftarrow A[i]$
 return $maxval$

Εύρεση Μέγιστης Τιμής (2/2)

Έστω C(n) το πλήθος των συγκρίσεων ως συνάρτηση του n

Έχουμε:

$$C(n) = \sum_{i=1}^{n-1} 1$$

Πολύ εύκολη παράσταση με την οποία καταλήγουμε ότι:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Μεθοδολογία

Η γενική μεθοδολογία για την ανάλυση μη αναδρομικών αλγορίθμων έχει ως εξής:

- **Αποφασίζουμε** την παράμετρο του μεγέθους της εισόδου
- **Αναγνωρίζουμε** τη βασική λειτουργία του αλγορίθμου (π.χ. σύγκριση, ανάθεση τιμής)
- **Ελέγχουμε** αν το πλήθος εκτέλεσης της βασικής λειτουργίας εξαρτάται μόνο από το μέγεθος εισόδου. Αν εξαρτάται από κάποια επιπλέον παράμετρο τότε μελετούμε ξεχωριστά την καλύτερη, τη χειρότερη και τη μέση περίπτωση.
- **Δημιουργούμε** ένα άθροισμα που απεικονίζει το πλήθος των εκτελέσεων των βασικών λειτουργιών.
- **Χρησιμοποιούμε** μαθηματικές μεθόδους και τύπους / θεωρήματα ώστε να καταλήξουμε στον τελικό ρυθμό αύξησης του αλγορίθμου.

Εύρεση Μοναδικότητας Στοιχείων (1/2)

Μέγεθος εισόδου: n (πλήθος στοιχείων του πίνακα)

Το loop περιέχει μόνο μια σύγκριση στοιχείων που είναι και η βασική λειτουργία

Το πλήθος συγκρίσεων εξαρτάται επίσης από το αν υπάρχουν ίσα στοιχεία

Μελετάμε τη χειρότερη περίπτωση

- **Πίνακες με όλα τα στοιχεία διαφορετικά**
- **Πίνακες με τα δύο τελευταία στοιχεία ίσα**

$C_{worst} = n$

```
ALGORITHM  UniqueElements(A[0..n − 1])
   //Determines whether all the elements in a given array are distinct
   //Input: An array A[0..n − 1]
   //Output: Returns "true" if all the elements in A are distinct
   //          and "false" otherwise
   for i ← 0 to n − 2 do
       for j ← i + 1 to n − 1 do
           if A[i] = A[j] return false
   return true
```

Έχουμε:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Εύρεση Μοναδικότητας Στοιχείων (2/2)

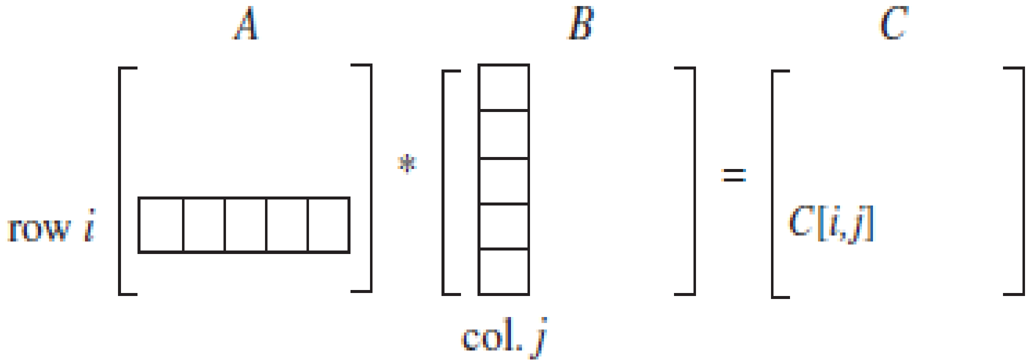
Εναλλακτικά:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Πολ/μος Δισδιάστατων Πινάκων (1/3)



$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

$$C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$$

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Πολ/μος Δισδιάστατων Πινάκων (2/3)

Στο εσωτερικό loop υπάρχουν πολ/μοι και προσθέσεις

Παραδοσιακά ο πολ/μος θα θεωρηθεί ως η βασική πράξη του αλγορίθμου

Υπολογίζουμε το συνολικό αριθμό των πολ/μων

Η μεταβλητή k πάει από το 0 στο n-1 μέσα στο εσωτερικό loop

Συνεπώς το πλήθος των πολ/μων στο εσωτερικό loop είναι

$$\sum_{k=0}^{n-1} 1$$

και το πλήθος όλων των πολ/μων $$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Πολ/μος Δισδιάστατων Πινάκων (3/3)

Οι υπολογισμοί έχουν ως εξής:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

Εύρεση Ψηφίων

Η πιο συχνή λειτουργία είναι η σύγκριση στο while αφού θα εκτελείται πάντα 1 φορά παραπάνω από το πλήθος επαναλήψεων

Οι τιμές του n δεν είναι πάντα οι ίδιες

Οι τιμή της n είναι περίπου μισή σε σχέση με την προηγούμενη

Συνεπώς, το πλήθος των επαναλήψεων θα είναι $\log_2 n + 1$

ALGORITHM *Binary*(n)
  //Input: A positive decimal integer n
  //Output: The number of binary digits in n's binary representation
  $count \leftarrow 1$
  while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
  return $count$

Ασκήσεις (1/5)

Βρείτε την πολυπλοκότητα του ακόλουθου αλγορίθμου:

$s \leftarrow 0$
for $i \leftarrow 1$ to $n-4$
  for $j \leftarrow i$ to $i+4$
    for $k \leftarrow i$ to j
      $s \leftarrow s + a_i$

Λύση

Το εξωτερικό loop εκτελείται n-4 φορές.

Το μεσαίο loop εκτελείται 5 φορές.

Το εσωτερικό loop εκτελείται 1, 2, 3, 4 ή 5 φορές για τις τιμές του j ίσες με I, i+1, i+2, i+3 και i+4.

Έτσι η πολυπλοκότητα είναι ίση με Θ(n)

Ασκήσεις (2/5)

Βρείτε την πολυπλοκότητα του ακόλουθου αλγορίθμου:

```
a ← 0
for i ← 1 to n
    for j ← i to n
        a ← a + 1
return a
```

Λύση

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}\Big(\underbrace{\sum_{j=1}^{n} 1}_{n} - \underbrace{\sum_{j=1}^{i-1} 1}_{i-1}\Big) = \sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n}(n+1) - \sum_{i=1}^{n} i =$$

$$n(n+1) - \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$$

Ασκήσεις (3/5)

Βρείτε την πολυπλοκότητα του ακόλουθου αλγορίθμου:

$$
\begin{array}{l}
a \leftarrow 0 \\
\textbf{for } i \leftarrow 1 \textbf{ to } n \\
\quad \textbf{for } j \leftarrow i+1 \textbf{ to } n \\
\qquad a \leftarrow a+1 \\
\textbf{return } a
\end{array}
$$

Λύση

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}\Big(\underbrace{\sum_{j=1}^{n} 1}_{n} - \underbrace{\sum_{j=1}^{i} 1}_{i}\Big) = \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n}(n) - \sum_{i=1}^{n} i =$$

$$n^2 - \frac{n(n+1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)$$

Ασκήσεις (4/5)

Βρείτε την πολυπλοκότητα του ακόλουθου αλγορίθμου:

```
a ← 0
for i ← 1 to n−1
    for j ← i+1 to n
        for k ← 1 to j
            a ← a+1
return a
```

Λύση

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j}1=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}j=\sum_{i=1}^{n-1}\left(\sum_{j=1}^{n}j-\sum_{j=1}^{i}j\right)=$$

$$\sum_{i=1}^{n-1}\left(\frac{1}{2}n(n+1)-\frac{1}{2}i(i+1)\right)=$$

$$\sum_{i=1}^{n-1}\left(\frac{1}{2}n(n+1)\right)-\frac{1}{2}\sum_{i=1}^{n-1}(i^2+i)=$$

$$\frac{1}{2}n(n+1)(n-1)-\frac{1}{2}\sum_{i=1}^{n-1}i^2-\frac{1}{2}\sum_{i=1}^{n-1}i$$

Όμως $\sum_{i=1}^{n}i^2=\frac{1}{6}n(n+1)(2n+1)$ και $\sum_{i=1}^{n-1}i^2=\frac{1}{6}(n-1)n(2n-1)$

Ασκήσεις (5/5)

Βρείτε την πολυπλοκότητα του ακόλουθου αλγορίθμου:

$a \leftarrow 0$
for $i \leftarrow 1$ to $n-1$
    for $j \leftarrow i+1$ to n
        for $k \leftarrow 1$ to j
            $a \leftarrow a+1$
return a

Λύση (συνέχεια)

$$\frac{1}{2}n(n-1)(n+1) - \frac{1}{12}n(n-1)(2n-1) - \frac{1}{4}n(n-1) =$$
$$\frac{1}{2}n(n-1)\left(n+1-\frac{1}{6}(2n-1)-\frac{1}{2}\right) =$$
$$\frac{1}{12}n(n-1)(6n+3-2n+1) = \frac{1}{12}n(n-1)(4n+4) =$$
$$\frac{1}{3}n(n-1)(n+1)$$

Άρα ο αλγόριθμος είναι $\Theta(n^3)$

Αναδρομικά Προβλήματα

Διαίρει και Βασίλευε

Τρία βήματα για την επίλυση:

- **Διαίρει**: διαιρούμε το πρόβλημα σε ένα πλήθος υπο-προβλημάτων
- **Βασίλευε**: επιλύουμε τα προβλήματα
- **Συνδύασε**: συνδυάζουμε τις λύσεις σε μια γενική λύση για το αρχικό πρόβλημα

Βασιζόμαστε σε αναδρομικές σχέσεις

Παράδειγμα:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Μέθοδοι Επίλυσης Αναδρομών

Τρείς μέθοδοι επίλυσης

- **Backward substitution**
- **Δένδρα αναδρομής**
- **Η master μέθοδος**

Δυαδική Αναζήτηση (1/6)

Αναζήτησε σε ταξινομημένο πίνακα μια τιμή x

```
int binarySearch (int A[], int x, int left, int right){
   int mid;
   if (left>right) return -1;
   mid=(left+right)/2;
   if (A[mid]==x) return mid;
   if (A[mid]<x) return binarySearch(A,x,mid+1,right);
   else return binarySearch(A,x,left,mid-1);
}

void main(){
   int A[10]={0,1,2,3,4,5,6,7,8,9};
   cout << binarySearch(A,8,0,9) << endl;
   cout << binarySearch(A,10,0,9) << endl;
}
```

Find: 9

Array 3 5 7 8 9 12 15

Δυαδική Αναζήτηση (2/6)

Χειρότερη περίπτωση: Το στοιχείο δεν βρίσκεται στον πίνακα

Μας ενδιαφέρει το πλήθος συγκρίσεων που γίνονται.

Σε κάθε κλήση της αναδρομής ο αλγόριθμος κάνει μια σύγκριση και κατόπιν συνεχίζει για το μισό πίνακα που απομένει.

Έστω T(N) η πολυπλοκότητα για μέγεθος πίνακα N.

Τότε έχω $T(N) <= T(\lfloor N/2 \rfloor) + 1$ με αρχική τιμή $T(1)=1$

Για τη λύση της αναδρομής διακρίνω 2 περιπτώσεις:

-Το N είναι δύναμη του 2

-Το N δεν είναι

Δυαδική Αναζήτηση (3/6)

Αν το Ν είναι δύναμη του 2 τότε έχω:

$T(N) <= T(N/2) +1$

$T(N/2) <= T(N/4) +1$

$T(N/4) <= T(N/8) + 1$

…

$T(N/2^{logN-1}) <= T(N/2^{logn}) + 1$

$T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = 1 + 1 + 1 + T(n/8)$ …. $T(n) = k + T(n/2^{k})$

με $k = \log n$, έχω $n/(2^{\log n})=n/n = 1$ και παίρνω $T(N) <= \log N + 1$

Δυαδική Αναζήτηση (4/6)

Αν το N δεν είναι δύναμη του 2 αποδεικνύω επαγωγικά ότι T(N)<= logN + 1

- Για N=3 ισχύει αφού χρειάζομαι 2 συγκρίσεις (T(N)=2)
- Έστω ότι ισχύει για k<N, Θα δείξω για k=N

Έχω: T(N) <= T($\lfloor N/2 \rfloor$) + 1

<= (από επαγωγική υπόθεση) $\lfloor \log \lfloor N/2 \rfloor \rfloor$ + 1 + 1

<= $\lfloor \log((N-1)/2) \rfloor$ +1 +1

<= $\lfloor \log(N-1) \rfloor$ + 1

<= $\lfloor \log N \rfloor$ + 1

Επομένως ο αλγόριθμος στη χειρότερη περίπτωση (όπου δε βρίσκουμε αυτό που αναζητούμε) έχει πολυπλοκότητα **O(logN)**.

Δυαδική Αναζήτηση (5/6)

Μέση Περίπτωση:

Στη μέση περίπτωση το στοιχείο μπορεί να βρίσκεται στον πίνακα σε οποιαδήποτε θέση του ή και να μη βρίσκεται.

-Υπάρχουν 2N + 1 καταστάσεις στις οποίες μπορεί να τερματίσει η αναζήτηση.

-Οι N αντιστοιχούν σε επιτυχή αναζήτηση (μία για κάθε θέση του πίνακα που μπορεί να βρεθεί το στοιχείο)

-Οι υπόλοιπες N+1 αντιστοιχούν στις θέσεις που μπορεί να τερματίσει η αναζήτηση ανεπιτυχώς (N-2 ενδιάμεσα των στοιχείων και 2 στις άκρες του πίνακα)

Αν θεωρήσω ισοπίθανα τα 2N+1 ενδεχόμενα αθροίζοντας προκύπτει πάλι O(logN)

Δυαδική Αναζήτηση (6/6)

Μη αναδρομικός αλγόριθμος:

```
int iterbin2(T[n], x) {
        if (n=1 || x<T[0])
                return 0;
        i=0; j=n-1;
        while (i<j) {
                //αν T[i]≤x<T[j+1]
                int k=(i+j+1) / 2;
                if (x<T[k])
                        j=k-1;
                else if (x≥T[k+1])
                        i=k+1;
                else {i=k; j=k;}
        }
        return i;
}
```

Υπολογισμός Παραγοντικού (1/3)

Μαθηματικός τύπος:

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

Βασική λειτουργία: πολ/μος

Το πλήθος των πολ/μων έχει ως εξής:

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0$$

Κάθε φορά το πλήθος εξαρτάται από το πλήθος στο n-1

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Υπολογισμός Παραγοντικού (2/3)

Η αρχική συνθήκη βρίσκεται στο if

Όταν ν=0 ο αλγόριθμος δεν κάνει πολ/μους

$$M(0) = 0.$$

the calls stop when $n = 0$ — no multiplications when $n = 0$

Εφαρμόζουμε τη μέθοδο των ***backward substitutions***

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \end{aligned}$$

Υπολογισμός Παραγοντικού (3/3)

Γενικός τύπος

$$M(n) = M(n-i) + i$$

Αφού η αρχική συνθήκη είναι n=0, αντικαθιστούμε με i=n και έχουμε:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n$$

Complexity → O(n)

Μεθοδολογία

Η γενική μεθοδολογία για την ανάλυση αναδρομικών αλγορίθμων έχει ως εξής:

- **Αποφασίζουμε** την παράμετρο του μεγέθους της εισόδου
- **Αναγνωρίζουμε** τη βασική λειτουργία του αλγορίθμου (π.χ. σύγκριση, ανάθεση τιμής)
- **Ελέγχουμε** αν το πλήθος εκτέλεσης της βασικής λειτουργίας εξαρτάται μόνο από το μέγεθος εισόδου. Αν εξαρτάται από κάποια επιπλέον παράμετρο τότε μελετούμε ξεχωριστά την καλύτερη, τη χειρότερη και τη μέση περίπτωση.
- **Δημιουργούμε** μια αναδρομική σχέση με μια κατάλληλη αρχική συνθήκη που απεικονίζει το πλήθος των εκτελέσεων της βασικής λειτουργίας.
- **Επιλύουμε** την αναδρομική σχέση.

Backward Substitution

Δύο βασικά βήματα:

- Περιγράφουμε τη μορφή της λύσης
- Χρησιμοποιούμε επαγωγή για να βρούμε σταθερές που αποδεικνύουν ότι η λύση ισχύει

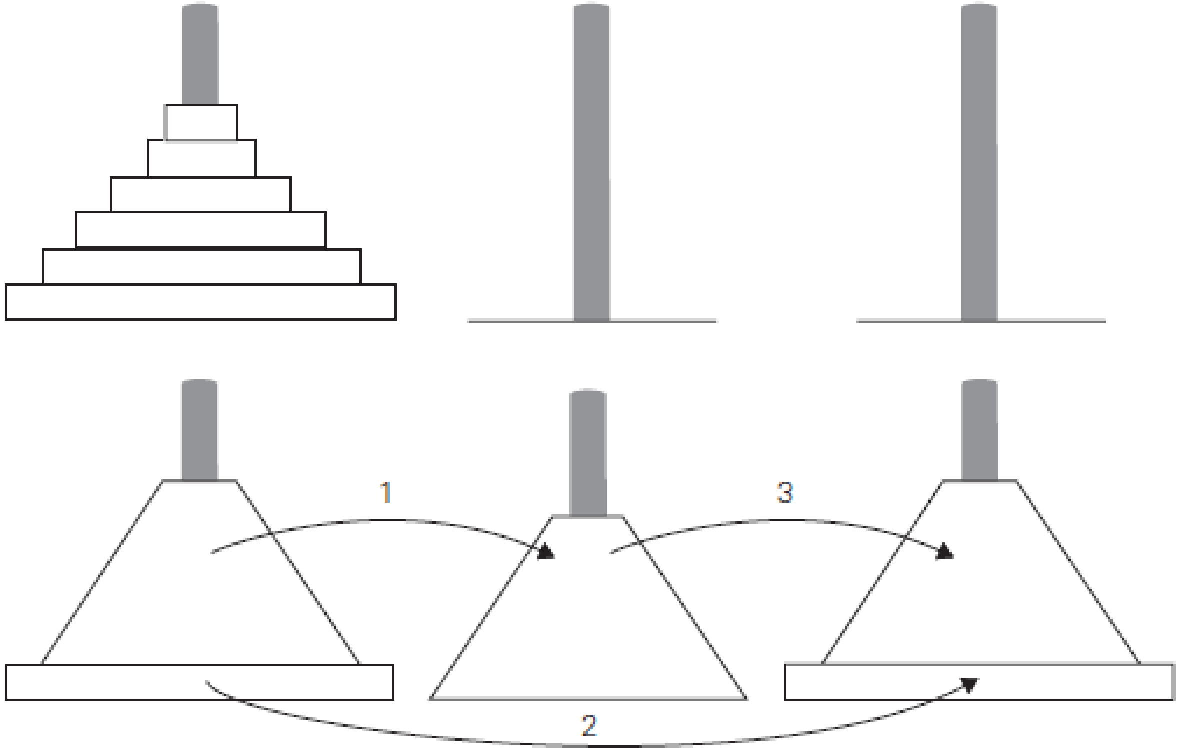
Η λύση που προτείνουμε είναι μια συνάρτηση όπου εφαρμόζουμε την επαγωγή

Οι Πύργοι του Ανόϊ (1/5)

Έχουμε n δίσκους διαφορετικού μεγέθους

Οι δίσκοι πρέπει να μεταφερθούν από τον 1ο στύλο στον 3ο αλλά:

- Ένας δίσκος κάθε φορά μπορεί να μεταφερθεί
- Μεγαλύτερος δίσκος δεν πρέπει να τοποθετηθεί πάνω από μικρότερο



Οι Πύργοι του Ανόϊ (2/5)

Για την μετακίνηση n δίσκων πρώτα μεταφέρουμε αναδρομικά n-1 δίσκους από τον 1ο στύλο στον 2ο (με τον 3ο στύλο βοηθητικό) και έπειτα μεταφέρουμε το μεγαλύτερο δίσκο στον 3ο στύλο

Στη συνέχεια μεταφέρουμε αναδρομικά n-1 δίσκους από τον 2ο δίσκο στον 3ο (με τον 1Ο δίσκο βοηθητικό)

Όταν n=1, απλά μετακινούμε το δίσκο από τον στύλο ‘πηγή’ στο στύλο ‘προορισμό’

Το μέγεθος εισόδου είναι n

Βασική λειτουργία: μετακίνηση / μεταφορά δίσκου

Οι Πύργοι του Ανόϊ (3/5)

Η αναδρομική σχέση είναι

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

Έχουμε:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1$$

$$M(1) = 1$$

Εφαρμόζοντας backward substitution παίρνουμε:

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$

$$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$

$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1$$

Οι Πύργοι του Ανόϊ (4/5)

Η σχέση που φαίνεται μέσα από την αναδρομική σχέση είναι:

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = \boxed{2^i M(n-i) + 2^i - 1}$$

n-i = 1 → i=n-1

Η αρχική συνθήκη ισχύει για n=1 και την έχουμε για i=n-1 έχουμε:

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

Ένα δένδρο που δείχνει τα βήματα της αναδρομής μπορεί να βοηθήσει στην επίλυση

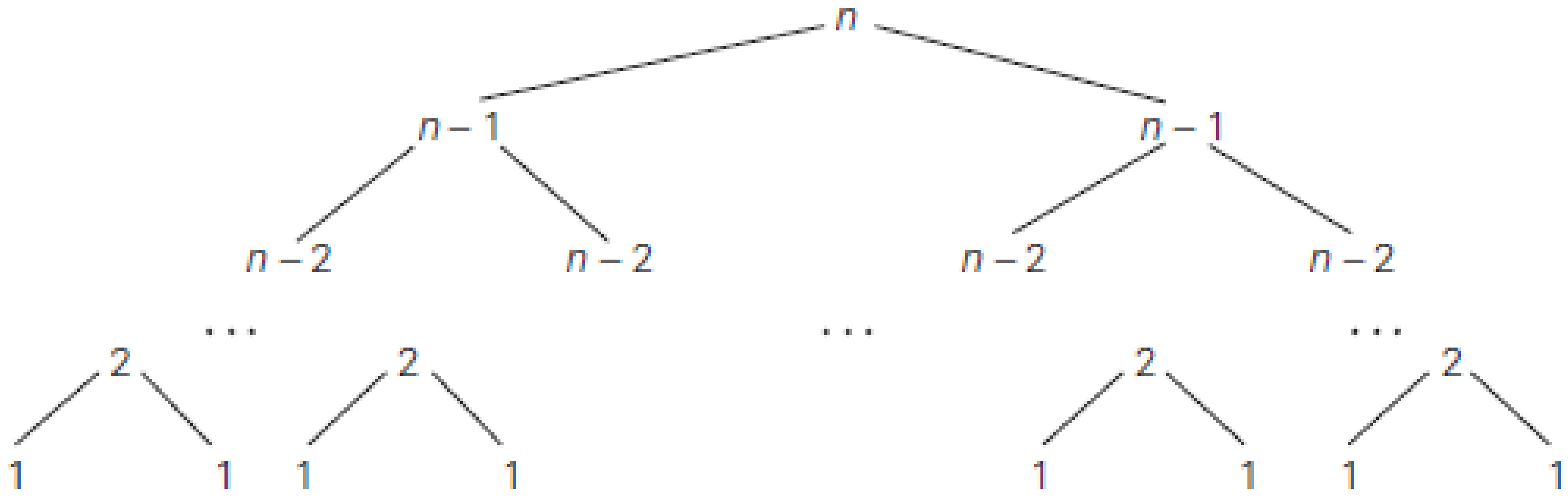
Complexity $O(2^n)$

Οι Πύργοι του Ανόϊ (5/5)

Πλήθος κόμβων στο δένδρο:

$$C(n) = \sum_{l=0}^{n-1} 2^l$$

Όπου l είναι τα επίπεδα του δένδρου



Εύρεση Ψηφίων Αναδρομικά (1/3)

Στόχος: εύρεση αναδρομικής σχέσης

Πλήθος προσθέσεων: $A(\lfloor n/2 \rfloor) + 1$

Αναδρομική σχέση:

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Πρόβλημα:

- Η μέθοδος backward substitution είναι δύσκολο να εφαρμοστεί για άλλους αριθμούς εκτός από τα πολλαπλάσια του 2

ALGORITHM $BinRec(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

Εύρεση Ψηφίων Αναδρομικά (2/3)

Επιλύουμε το πρόβλημα για τα πολλαπλάσια του 2 και εφαρμόζουμε τον **κανόνα εξομάλυνσης (smoothness rule)**

Έχουμε:

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0$$
$$A(2^0) = 0.$$

και με backward substitution παίρνουμε:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots\\
&\ldots\\
&= A(2^{k-i}) + i\\
&\ldots\\
&= A(2^{k-k}) + k
\end{aligned}$$

Εύρεση Ψηφίων Αναδρομικά (3/3)

Τελικά καταλήγουμε στο:

$$A(2^k) = A(1) + k = k$$

και επιστρέφοντας στη συλλογιστική μας για οποιοδήποτε n έχουμε:

$$A(n) = \log_2 n \in \Theta(\log n)$$

αφού $2^k = n$

Ουσιαστικά

$$A(n) = \lfloor \log_2 n \rfloor$$

The Master Theorem (1/7)

Μας προσφέρει μια μεθοδολογία για να επιλύουμε αναδρομικές εξισώσεις της μορφής:

$$T(n) = aT(n/b) + f(n)$$

με α>=1 και b>1 να είναι σταθερές και f(n) είναι μια ασυμπτωτικά θετική συνάρτηση

Η παραπάνω συνάρτηση χωρίζει το πρόβλημα μεγέθους n σε α υπο-προβλήματα μεγέθους n/b

Τα α υπο-προβλήματα λύνονται πάλι αναδρομικά σε χρόνο n/b

Η f(n) απεικονίζει το κόστος της διαίρεσης του προβλήματος και της συγχώνευσης των λύσεων

The Master Theorem (2/7)

Το n/b μπορεί να μην είναι ακέραιος

Παίρνουμε το floor ή το ceiling του n/b χωρίς να επηρεάζεται η ασυμπτωτική συμπεριφορά

Θα πρέπει να θυμόμαστε τρεις περιπτώσεις

Υπάρχουν κενά στη μέθοδο αφού η f(n) πρέπει να είναι πολυωνυμικά μικρότερη / μεγαλύτερη από το $n^{\log_b a}$

Αν είναι τότε δεν μπορούμε να χρησιμοποιήσουμε το master theorem για την επίλυση αναδρομικών σχέσεων

The Master Theorem (3/7)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

The Master Theorem (4/7)

Ας δούμε ένα παράδειγμα:

Για την $T(n) = 9T(n/3) + n$

έχουμε $a = 9,\ b = 3,\ f(n) = n$,

και έτσι $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

Αφού $f(n) = O(n^{\log_3 9 - \epsilon})$

με $\epsilon = 1$

Εφαρμόζουμε την 1η περίπτωση και καταλήγουμε στο

$$T(n) = \Theta(n^2)$$

The Master Theorem (5/7)

Παράδειγμα

Για την $T(n) = T(2n/3) + 1$

έχουμε $a = 1, b = 3/2, f(n) = 1$

και έτσι $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Αφού $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$

Εφαρμόζουμε τη $2^η$ περίπτωση και καταλήγουμε στο

$$T(n) = \Theta(\lg n)$$

The Master Theorem (6/7)

Παράδειγμα

Για την $T(n) = 3T(n/4) + n \lg n$

έχουμε $a = 3,\ b = 4,\ f(n) = n \lg n$

και έτσι $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$

Αφού $f(n) = \Omega(n^{\log_4 3+\epsilon})$

με $\epsilon \approx 0.2$

για μεγάλο n έχουμε $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$

Εφαρμόζουμε τη 3η περίπτωση και καταλήγουμε στο $T(n) = \Theta(n \lg n)$

The Master Theorem (7/7)

Παράδειγμα

Για την $T(n) = 2T(n/2) + n \lg n$

η master μέθοδος δεν μπορεί να εφαρμοστεί αφού για

$a = 2, b = 2, f(n) = n \lg n$ και $n^{\log_b a} = n$

Δεν μπορεί να εφαρμοστεί η 3η περίπτωση αφού η f(n) είναι ασυμπτωτικά μεγαλύτερη από την $n^{\log_b a} = n$ αλλά όχι πολυωνυμικά μεγαλύτερη. Ο λόγος $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$

Είναι ασυμπτωτικά μικρότερος από το n^{ϵ}

για οποιοδήποτε ε

Ασκήσεις (1/2)

1. $T(n) = 3T(n/2) + n^2$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = T(n/2) + 2^n$
4. $T(n) = 2^n T(n/2) + n^n$
5. $T(n) = 16T(n/4) + n$
6. $T(n) = 2T(n/2) + n \log n$

---

1. $T(n) = 3T(n/2) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = T(n/2) + 2^n \implies \Theta(2^n)$ (Case 3)
4. $T(n) = 2^n T(n/2) + n^n \implies$ Does not apply (a is not constant)
5. $T(n) = 16T(n/4) + n \implies T(n) = \Theta(n^2)$ (Case 1)
6. $T(n) = 2T(n/2) + n \log n \implies T(n) = n \log^2 n$ (Case 2)

Ασκήσεις (2/2)

7. $T(n) = 2T(n/2) + n/\log n$

8. $T(n) = 2T(n/4) + n^{0.51}$

9. $T(n) = 0.5T(n/2) + 1/n$

10. $T(n) = 16T(n/4) + n!$

11. $T(n) = \sqrt{2}T(n/2) + \log n$

12. $T(n) = 3T(n/2) + n$

7. $T(n) = 2T(n/2) + n/\log n \Longrightarrow$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)

8. $T(n) = 2T(n/4) + n^{0.51} \Longrightarrow T(n) = \Theta(n^{0.51})$ (Case 3)

9. $T(n) = 0.5T(n/2) + 1/n \Longrightarrow$ Does not apply ($a < 1$)

10. $T(n) = 16T(n/4) + n! \Longrightarrow T(n) = \Theta(n!)$ (Case 3)

11. $T(n) = \sqrt{2}T(n/2) + \log n \Longrightarrow T(n) = \Theta(\sqrt{n})$ (Case 1)

12. $T(n) = 3T(n/2) + n \Longrightarrow T(n) = \Theta(n^{\lg 3})$ (Case 1)

The Master Theorem

Εναλλακτικά:

$$\text{If } f(n) \in \Theta(n^d) \quad d \geq 0$$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Ασκήσεις (1/12)

Να λύσετε την επόμενη αναδρομική σχέση:

$$T(n)=3T(n/2)+n^2$$

Ασκήσεις (2/12)

Να λύσετε την επόμενη αναδρομική σχέση: T(n)=3T(n/2)+n^2

Λύση

$n^{\log_2 3} = n^{1.58}$

Άρα f(n) = Ω($n^{1.58}$)

Κοιτάζουμε την ανίσωση: af(n/b)<=cf(n)

Έχουμε: 3/4 n^2<=c n^2, ισχύει άρα 3η περίπτωση του Master, συνεπώς T(n)=Θ(n^2)

Ασκήσεις (3/12)

Να λύσετε την επόμενη αναδρομική σχέση:

$$T(n)=4T(n/2)+n^2$$

Ασκήσεις (4/12)

Να λύσετε την επόμενη αναδρομική σχέση: T(n)=4T(n/2)+n^2

Λύση

$n^{\log_2 4} = n^2$

Άρα f(n) = Θ(n^2)

2η περίπτωση του Master, συνεπώς T(n)=Θ($n^2 logn$)

Ασκήσεις (5/12)

Να λύσετε την επόμενη αναδρομική σχέση:

$$T(n)=T(n/2)+2^{n}$$

Ασκήσεις (6/12)

Να λύσετε την επόμενη αναδρομική σχέση: T(n)=T(n/2)+2^n

Λύση

$n^{\log_2 1} = n^0$=1

Άρα f(n) = Ω(n^0)

Κοιτάζουμε την ανίσωση: af(n/b)<=cf(n)

Έχουμε: $2^{n/2}$<=c 2^n, ισχύει άρα 3η περίπτωση του Master, συνεπώς T(n)=Θ(2^n)

Ασκήσεις (7/12)

Να λύσετε την επόμενη αναδρομική σχέση:

$$T(n)=16T(n/4)+n$$

Ασκήσεις (8/12)

Να λύσετε την επόμενη αναδρομική σχέση: T(n)=16T(n/4)+n

Λύση

$n^{\log_4 16} = n^2$

Άρα f(n) = O(n^2)

1η περίπτωση του Master, συνεπώς T(n)=Θ(n^2)

Ασκήσεις (9/12)

Να λύσετε την επόμενη αναδρομική σχέση:

$$T(n)=\sqrt{2}T(n/2)+logn$$

Ασκήσεις (10/12)

Να λύσετε την επόμενη αναδρομική σχέση: T(n)=$\sqrt{2}$T(n/2)+$logn$

Λύση

$n^{\log_2 \sqrt{2}} = n^{1/2}$

Άρα f(n) = O($n^{1/2}$)

1η περίπτωση του Master, συνεπώς T(n)=Θ($n^{1/2}$)

Ασκήσεις (11/12)

Να λύσετε την επόμενη αναδρομική σχέση:

$$T(n)=3T(n/2)+n$$

Ασκήσεις (12/12)

Να λύσετε την επόμενη αναδρομική σχέση: T(n)=3T(n/2)+n

Λύση

$n^{\log_2 3} = n^{1.58}$

Άρα f(n) = O($n^{1.58}$)

1η περίπτωση του Master, συνεπώς T(n)=Θ($n^{1.58}$)

Διαίρει και βασίλευε

(συνέχεια)

Πολ/μός Μεγάλων Αριθμών (1/9)

Κάποιες εφαρμογές (π.χ. κρυπτογραφία) απαιτούν το χειρισμό ακεραίων ακόμη και με 100 ψηφία

Αυτού του είδους οι αριθμοί δεν μπορούν να χωρέσουν σε μια λέξη ενός υπολογιστή

Απαιτούν ειδική μεταχείριση

Αν προσπαθήσουμε να πολλαπλασιάσουμε αριθμούς n ψηφίων χρειαζόμαστε n^2 πολλαπλασιασμούς (κάθε ένα ψηφίο του πρώτου αριθμού πολλαπλασιάζεται με όλα τα ψηφία του δεύτερου)

Πολ/μός Μεγάλων Αριθμών (2/9)

Βασική ιδέα αλγορίθμου

- Παράδειγμα: πολλαπλασιασμός του 23 και 14
- Οι αριθμοί γράφονται ως εξής

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \quad \text{and} \quad 14 = 1 \cdot 10^1 + 4 \cdot 10^0$$

- Οπότε

$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0 \end{aligned}$$

Πολ/μός Μεγάλων Αριθμών (3/9)

Ο τελευταίο τύπος εξάγει το σωστό αποτέλεσμα αλλά απαιτεί 4 πολλαπλασιασμούς

Μπορεί να γίνει μια βελτίωση ως εξής:

$$2*4+3*1=(2+3)*(1+4)-2*1-3*4$$

Για δύο αριθμούς $a=a_1a_0$ και $b=b_1b_0$

Το γινόμενό τους μπορεί να δοθεί ως εξής:

$$c=a*b=c_2 10^2+c_1 10^1+c_0,$$ με

$$c_2=a_1*b_1$$

$$c_0=a_0*b_0$$

$$c_1=(a_1+a_0)*(b_1+b_0)-(c_2+c_0)$$

Πολ/μός Μεγάλων Αριθμών (4/9)

Εφαρμόζουμε τον τύπο για τον πολλαπλασιασμό αριθμών με n ψηφία

Διαιρούμε τους αριθμούς στη μέση

α_1 είναι τα πρώτα ψηφία του α και α_0 είναι τα υπόλοιπα

b_1 είναι τα πρώτα ψηφία του b και b_0 είναι τα υπόλοιπα

Συνεπώς:

$$a = a_1 10^{n/2} + a_0$$

$$b = b_1 10^{n/2} + b_0.$$

Πολ/μός Μεγάλων Αριθμών (5/9)

Οπότε

$$
\begin{aligned}
c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0) \\
&= c_2 10^n + c_1 10^{n/2} + c_0
\end{aligned}
$$

με

$$
\begin{aligned}
c_2 &= a_1 * b_1 \\
c_0 &= a_0 * b_0 \\
c_1 &= (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)
\end{aligned}
$$

Πολ/μός Μεγάλων Αριθμών (6/9)

Αν το n/2 είναι άρτιος μπορούμε να συνεχίσουμε με την ίδια μέθοδο για τα c_2, c_1, c_0

Συνεπώς αν το n είναι δύναμη του 2 μπορούμε εύκολα να καταλήξουμε στην αναδρομική σχέση

Η αναδρομική σχέση καταλήγει όταν $n = 1$

Μπορεί να καταλήξει όταν το n είναι πολύ μικρό

Απαιτούνται 3 πολλαπλασιασμοί αριθμών με n/2 ψηφία

Πολ/μός Μεγάλων Αριθμών (7/9)

Υπολογισμοί

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

$$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2})$$

$$= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k.$$

$$k = \log_2 n$$

$$a^{\log_b c} = c^{\log_b a}$$

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$$

Πολ/μός Μεγάλων Αριθμών (8/9)

Προσθέσεις και αφαιρέσεις

- Απαιτούνται πέντε προσθέσεις και μια αφαίρεση
- Παίρνουμε την ακόλουθη αναδρομική σχέση

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \quad A(1) = 1.$$

- Και καταλήγουμε ότι

$$A(n) \in \Theta(n^{\log_2 3}),$$

Πολ/μός Μεγάλων Αριθμών (9/9)

Η διαίρει και βασίλευε είναι γρηγορότερη ακόμα και για αριθμούς 8 ψηφίων σε σχέση με τις συμβατικές μεθόδους

Για αριθμούς με 300 ψηφία είναι δύο φορές πιο γρήγορη

Στις γλώσσες Java, C++ υπάρχουν ειδικές κλάσεις για χειρισμό μεγάλων αριθμών

Ερωτήματα:

- **Τι γίνεται αν οι δύο αριθμοί δεν είναι του ιδίου μεγέθους?**
- **Τι γίνεται αν το πλήθος των ψηφίων n δεν είναι πολλαπλάσιο του 2?**

Matrix Multiplication (1/6)

Αν $A = (a_{ij})$ και $B = (b_{ij})$ είναι nxn πίνακες τότε το γινόμενό τους είναι

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Αλγόριθμος

SQUARE-MATRIX-MULTIPLY(A, B)

```
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Matrix Multiplication (2/6)

Προσέγγιση με διαίρει και βασίλευε

- Υποθέτουμε ότι σπάμε τους πίνακες σε 4 n/2 x n/2 πίνακες

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Matrix Multiplication (3/6)

Για κάθε μια από τις προηγούμενες 4 εξισώσεις απαιτούνται 2 πολ/μοι και η πρόσθεση των n/2 x n/2 γινομένων

Νέος αλγόριθμος

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

$n = A.rows$
let C be a new $n \times n$ matrix
if $n == 1$
    $c_{11} = a_{11} \cdot b_{11}$
else partition A, B, and C as in equations (4.9)
    $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
    $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
    $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
    $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
return C

Matrix Multiplication (4/6)

Χωρίζουμε τους πίνακες χωρίς να αντιγράφουμε στοιχεία

Χρησιμοποιούμε index calculations

Αναγνωρίζουμε ένα υπο-πίνακα από ένα εύρος γραμμών και στηλών στον αρχικό πίνακα

Έστω T(n) είναι ο χρόνος για τον πολλαπλασιασμό των πινάκων

Όταν n=1 εκτελούμε ένα πολλαπλασιασμό, οπότε T(1)=Θ(1)

Η αναδρομική σχέση ισχύει για n>1

Matrix Multiplication (5/6)

Το κόστος της γραμμής 5 του αλγορίθμου είναι Θ(1)

Στις γραμμές 6-9 εκτελούμε 8 κλήσεις της αναδρομής

Κάθε κλήση συνεισφέρει με T(n/2) στο συνολικό χρόνο

Άρα το συνολικό κόστος των κλήσεων είναι 8T(n/2)

Στις γραμμές 6-9 εκτελούμε 4 προσθέσεις πινάκων

Κάθε υπο-πίνακας περιέχει $n^2/4$ στοιχεία

Οι 4 προσθέσεις έχουν πολυπλοκότητα $\Theta(n^2)$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4      c11 = a11 · b11
5  else partition A, B, and C as in equations (4.9)
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

Matrix Multiplication (6/6)

Ο συνολικός χρόνος για τις αναδρομικές κλήσεις και τους πολλαπλασιασμούς – προσθέσεις είναι:

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) \end{aligned}$$

Αν στη διάσπαση των πινάκων επιλέξουμε την αντιγραφή (κόστος $\Theta(n^2)$) η αναδρομική σχέση δεν θα αλλάξει

Συνεπώς
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Πολυπλοκότητα: $O(n^3)$ από την 1η περίπτωση του Θ.Κ.

Strassen's Method(1/7)

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices . This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

Strassen’s Method (2/7)

Αναδρομική σχέση

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Με τη χρήση του master θεωρήματος παίρνουμε ότι $T(n) = \Theta(n^{\lg 7})$.

Strassen’s Method (3/7)

Δημιουργία 10 υπο-πινάκων στο 2ο βήμα

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

Κόστος $\Theta(n^2)$

Strassen’s Method (4/7)

Πολλαπλασιασμός στο 3ο βήμα

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

Strassen's Method (5/7)

Προσθέσεις στο 4ο βήμα

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$
\begin{array}{rrrrrrrr}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & & & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} & & \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} & \\
\hline
A_{11} \cdot B_{11} & & & & & & + A_{12} \cdot B_{21} &
\end{array}
$$

Strassen's Method (6/7)

$$C_{12} = P_1 + P_2$$

$$\begin{array}{l} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \qquad\qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \qquad\qquad\quad + A_{12} \cdot B_{22} \end{array}$$

$$C_{21} = P_3 + P_4$$

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad\qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad\quad + A_{22} \cdot B_{21} \end{array}$$

Strassen’s Method (7/7)

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

$$
\begin{array}{rrrrrrr}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12}
\end{array}
$$

Brute Force and Exhaustive Search

Εισαγωγή

Πρόκειται για μια άμεση μέθοδο για την επίλυση προβλημάτων συχνά βασισμένη στη διατύπωση του προβλήματος και στους ορισμούς των εννοιών που εμπλέκονται

Το όνομα της μεθόδου σημαίνει: Just do it!

Πολλές φορές είναι η πιο απλή μέθοδος

Παράδειγμα:

- Υπολογισμός της δύναμης $α^ν$

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}}$$

Χαρακτηριστικά

Εφαρμόζεται σε ένα μεγάλο εύρος προβλημάτων

- Μάλλον είναι η μέθοδος που δύσκολα κάποιος βρίσκει προβλήματα που να μην μπορεί να τα χειριστεί

Οδηγεί σε ‘λογικούς’ αλγορίθμους με πρακτική σημασία

Παρά το γεγονός ότι γενικά δεν είναι αποδοτική μέθοδος, μπορεί να χρησιμοποιηθεί για μικρού μεγέθους προβλήματα

Μπορεί να αποτελέσει μια θεωρητική βάση για τη σύγκριση με άλλες μεθοδολογίες

Εφαρμογές

Αριθμητικά προβλήματα

- Αποδεκτή επίδοση
- Μπορεί να χρησιμοποιηθεί για μεγάλου μεγέθους προβλήματα

Συνδυαστικά Προβλήματα

Εύρεση Διαιρετών

Δοσμένου ενός αριθμού n να βρεθούν οι διαιρέτες του

Απαριθμούμε όλους τους ακέραιους από το 1 μέχρι το n (υποψήφιοι διαιρέτες)

Ελέγχουμε αν ο καθένας από αυτούς είναι διαιρέτης του n

```
d ← 1
while ( d < n + 1 ) do
        if ( n mod d == 0 )
        then output (d)
        d ← d + 1
```

String Matching (1/6)

Δεδομένου ενός αλφαριθμητικού n χαρακτήρων (text) και ενός αλφαριθμητικού m χαρακτηρων (pattern) με m <= n να βρεθεί ένα υπο-αλφαριθμητικό του text που ταιριάζει με το pattern

Θέλουμε να βρούμε το i-index του πιο αριστερά χαρακτήρα τέτοιο ώστε

$$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}:$$

$$\begin{array}{ccccccccccc}
t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & & \ldots & t_{n-1} & \text{text } T \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\
 & & p_0 & \ldots & p_j & \ldots & p_{m-1} & \text{pattern } P & & &
\end{array}$$

Μπορεί ο αλγόριθμος να συνεχίζει για πολλαπλές εμφανίσεις

String Matching (2/6)

Η προσέγγιση brute force είναι προφανής:

- ‘Ευθυγραμίζουμε’ το pattern με τους πρώτους m χαρακτήρες του text και συγκρίνουμε τα αντίστοιχα ζεύγη
- Συνεχίζουμε μέχρι να ταιριάξουν όλοι οι χαρακτήρες ή να βρούμε μια διαφορά
- Αν βρούμε διαφορά, τότε μετακινούμε μια θέση το pattern προς τα δεξιά και ξεκινούμε από την αρχή
- Η τελευταία θέση στην οποία μπορεί να υπάρχει το pattern είναι η n – m
- Πέρα από αυτή τη θέση δεν υπάρχουν αρκετοί χαρακτήρες

String Matching (3/6)

Αλγόριθμος

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//          an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//          matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

String Matching (4/6)

Παράδειγμα εκτέλεσης:

N	O	B	O	D	Y	_	N	O	T	I	C	E	D	_	H	I	M
N	**O**	**T**															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	**O**	**T**								

String Matching (5/6)

Η χειρότερη περίπτωση είναι να γίνουν m συγκρίσεις προτού μετακινηθεί το pattern προς τα δεξιά

Ο μέγιστος αριθμός προσπαθειών είναι $n – m + 1$

Στη χειρότερη περίπτωση γίνονται $m (n – m + 1)$ συγκρίσεις

Πολυπλοκότητα $O(m \cdot n)$

Η μέση περίπτωση έχει πολυπλοκότητα $\Theta(n)$

Προχωρημένοι αλγόριθμοι για string matching θα συζητηθούν σε επόμενα μαθήματα

String Matching (6/6)

Άσκηση

Καθορίστε το πλήθος των συγκρίσεων (χαρακτήρες) που γίνονται από τον brute force αλγόριθμο για string matching και για το pattern **GANDHI** ως προς το text

THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED

Closest Pair Problem (1/4)

Αναζητούμε τα κοντινότερα σημεία μέσα σε ένα πλήθος n σημείων

Ανήκει στη κατηγορία της υπολογιστικής γεωμετρίας

Παραδείγματα:

- θέσεις αεροπλάνων
- θέσεις γραφείων
- εγγραφές σε βάσεις δεδομένων
- ανάλυση συστάδων (cluster)
- ....

Closest Pair Problem (2/4)

Υποθέτουμε σημεία στο επίπεδο

Απόσταση σημείων

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Υπολογίζουμε την απόσταση μεταξύ κάθε ενός ζεύγους σημείων και παίρνουμε το ζεύγος με τη μικρότερη απόσταση

Για να μην μετρήσουμε την απόσταση του ίδιου ζεύγους δυο φορές εστιάζουμε σε ζεύγη (p_i, p_j) με $i < j$

Closest Pair Problem (3/4)

Αλγόριθμος

ALGORITHM *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points

$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

Closest Pair Problem (4/4)

Η βασική λειτουργία του αλγορίθμου είναι ο υπολογισμός της τετραγωνικής ρίζας

Ο υπολογισμός της ρίζας είναι προσεγγιστικός πολλές φορές

Αποφεύγουμε τη σύγκριση της ρίζας αν συγκρίνουμε τα $(x_i - x_j)^2 + (y_i - y_j)^2$

Το πλήθος των εκτελέσεων της βασικής λειτουργίας είναι:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i)$$

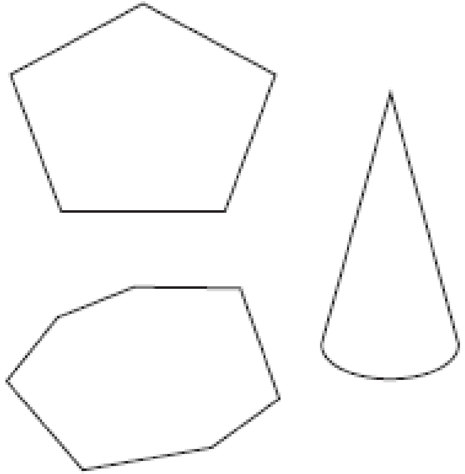
$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2)$$

Convex Hull Problem (1/10)

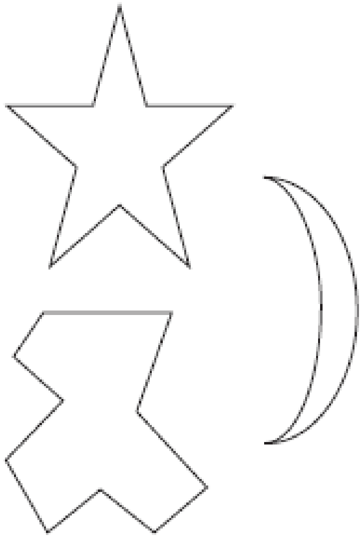
Ορισμός

Ένα σύνολο σημείων καλείται κυρτό όταν για δύο οποιαδήποτε σημεία p και q του συνόλου, ολόκληρη το ευθύγραμμο τμήμα με τελικά σημεία p και q ανήκει στο σύνολο.

Κυρτά σύνολα



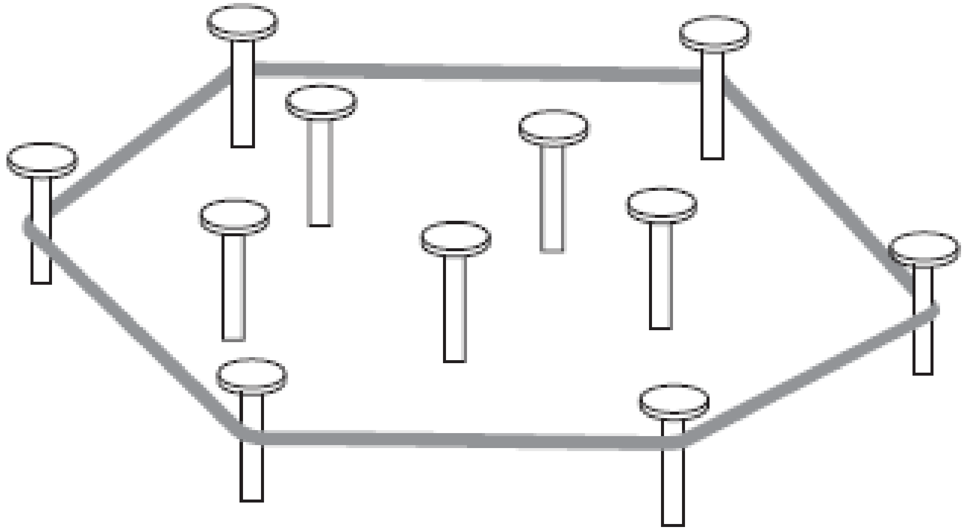
Μη κυρτά σύνολα



Convex Hull Problem (2/10)

Ορισμός

Το convex hull ενός συνόλου σημείων S είναι το μικρότερο κυρτό σύνολο που περιέχει το S



Convex Hull Problem (3/10)

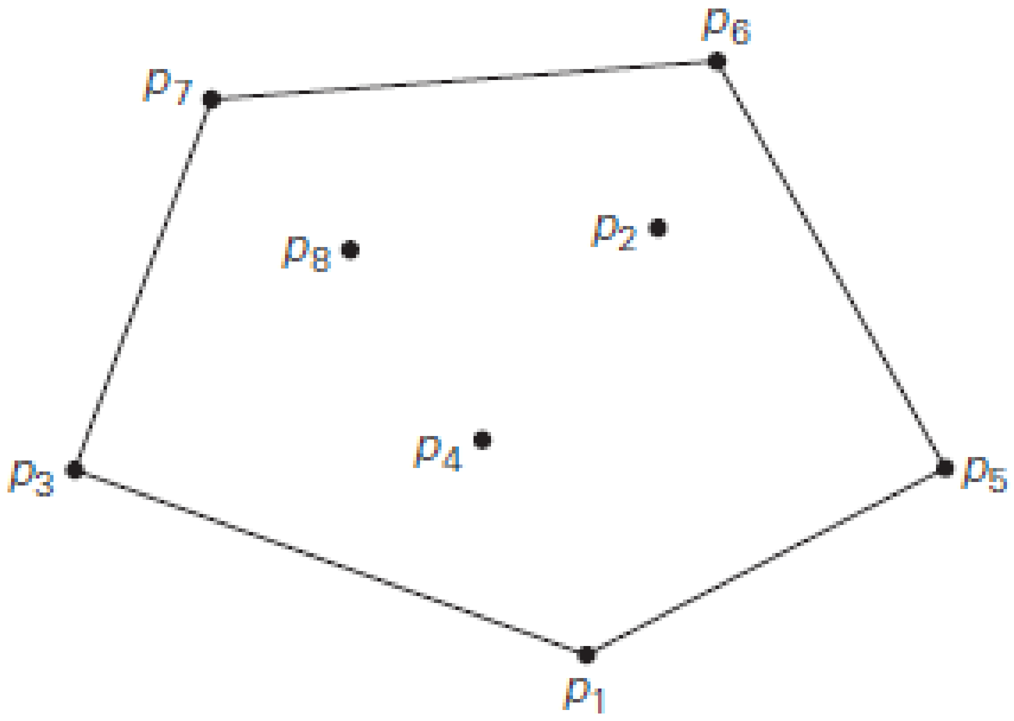
Αν το S είναι κυρτό τότε το convex hull είναι το ίδιο το S

Αν το S αποτελείται από δύο σημεία τότε το convex hull είναι η ευθεία που ενώνει τα δύο αυτά σημεία

Αν το S αποτελείται από τρία σημεία που δεν ανήκουν στην ίδια ευθεία, τότε το convex hull είναι το τρίγωνο με κορυφές στα τρία σημεία του S; Αν τα τρία σημεία ανήκουν στην ίδια ευθεία, τότε το convex hull είναι η ευθεία που έχει ως άκρες τα πιο απομακρυσμένα σημεία

Convex Hull Problem (4/10)

Το convex hull για περισσότερα σημεία φαίνεται στην ακόλουθη εικόνα



Convex Hull Problem (5/10)

Θεώρημα

Το convex hull ενός συνόλου σημείων S με πλήθος σημείων n > 2 τα οποία δεν ανήκουν όλα στην ίδια ευθεία γραμμή είναι ένα κυρτό πολύγωνο με κορυφές σε κάποια από τα σημεία του S. Αν όλα τα σημεία είναι στην ίδια γραμμή, το πολύγωνο εκφυλίζεται σε ένα ευθύγραμμο τμήμα αλλά με άκρες πάλι δύο σημεία του S.

Convex Hull Problem (6/10)

Το πρόβλημα είναι η κατασκευή του convex hull για ένα σύνολο σημείων S

Εύρεση των σημείων που θα αποτελέσουν τις κορυφές του πολυγώνου (extreme points)

Τα extreme points είναι σημεία τα οποία δεν βρίσκονται ενδιάμεσα σε ένα ευθύγραμμο τμήμα με τελικά σημεία στο S

Προσπαθούμε επίσης να βρούμε ποια σημεία στα άκρα του συνόλου S θα είναι αυτά που θα συνδεθούν μεταξύ τους

Convex Hull Problem (7/10)

Απλός αλγόριθμος εύρεσης convex hull (μη αποδοτικός όμως)

- Ένα ευθύγραμμο τμήμα που συνδέει δύο σημεία p_i και p_j ενός συνόλου n σημείων είναι τμήμα του convex hull όταν και μόνο όταν όλα τα υπόλοιπα σημεία του συνόλου βρίσκονται στην ίδια πλευρά της ευθείας των δύο σημείων
- Επαναλαμβάνουμε αυτό το σενάριο για όλα τα ζεύγη σημείων και βρίσκουμε τα ευθύγραμμα τμήματα που αποτελούν το όριο του convex hull
- Εύρεση ευθύγραμμου τμήματος

$$ax + by = c \qquad a = y_2 - y_1,\ b = x_1 - x_2,\ c = x_1 y_2 - y_1 x_2.$$

Convex Hull Problem (8/10)

- Ένα ευθύγραμμο τμήμα που συνδέει δύο σημεία χωρίζει το επίπεδο σε δύο τμήματα
- Στο ένα τμήμα ισχύει

$$ax + by > c$$

- Ενώ στο άλλο

$$ax + by < c.$$

- Για τα σημεία πάνω στο ευθύγραμμο τμήμα ισχύει

$$ax + by = c$$

- Για να ελέγξουμε αν δυο σημεία βρίσκονται στο ίδιο τμήμα ελέγχουμε το πρόσημο της $ax + by - c$

Convex Hull Problem (9/10)

Επίδοση του αλγορίθμου: $O(n^3)$

Για κάθε ένα από τα $n(n-1)/2$ ζεύγη, πρέπει να βρούμε το πρόσημο της προηγούμενης παράστασης για κάθε ένα από τα υπόλοιπα $n-2$ σημεία

Αποδοτικότεροι αλγόριθμοι για την επίλυση του προβλήματος θα συζητηθούν στην πορεία του μαθήματος

Convex Hull Problem (10/10)

Πρόβλημα υπολογιστικής γεωμετρίας

Δεδομένης μιας λίστας n σημείων να βρεθεί ο μικρότερος κυρτός χώρος που περικλείει όλα τα σημεία (convex hull)

Παράδειγμα:

- Computer animation: η αντικατάσταση αντικειμένων από το κυρτό τους χώρο οδηγεί στην ταχύτατη αναγνώριση πιθανών συγκρούσεων
- Γεωγραφικά πληροφοριακά συστήματα
- Outliers detection σε στατιστικές μεθόδους

Exhaustive Search (1/4)

Πρόκειται για μια brute force τεχνική για συνδυαστικά προβλήματα

Οδηγεί στη δημιουργία όλων των στοιχείων ενός προβλήματος

Επιλέγει τα στοιχεία που ικανοποιούν κάποια κριτήρια

Τελικά, βρίσκει το επιθυμητό στοιχείο

Κλασικά προβλήματα:

- the Traveling Salesman Problem
- the Knapsack Problem
- the Assignment Problem

Exhaustive Search (2/4)

Στρατηγική:

- Απαριθμούμε όλες τις πιθανές υποψήφιες λύσεις
- Ελέγχουμε αν μια λύση ικανοποιεί τις συνθήκες του προβλήματος
- Αν απαιτείται, επιλέγουμε μια λύση από το σύνολο των εφικτών λύσεων

Πως θα ελέγξουμε όλες τις πιθανές λύσεις?

Exhaustive Search (3/4)

Βασικός αλγόριθμος

```
c ← generate a first candidate solution
while ( c is a candidate ) do
   if ( c is a valid solution )
   then output (c)
   c ← generate the next candidate solution, if any
```

Μπορεί να τερματιστεί όταν:

- Βρει την πρώτη εφικτή λύση
- Βρει ένα αριθμό αποδεκτών λύσεων
- Ελέγξει ένα αριθμό υποψήφιων λύσεων
- ‘Ξοδέψει’ μια ποσότητα χρόνου ή resources

Exhaustive Search (4/4)

Χαρακτηριστικά:

- Απλή τεχνική
- Πάντα βρίσκει μια λύση εφόσον υπάρχει
- Το κόστος είναι ανάλογο με το πλήθος των υποψήφιων λύσεων
- Εκτοξεύεται το πλήθος και ο χρόνος εκτέλεσης
- Είναι πρακτική μόνο για μικρού μεγέθους προβλήματα

Επιτάχυνση Υπολογισμών

Μείωση του χώρου των πιθανών λύσεων
- Heuristics / Analysis

Ανακατανομή του χώρου των λύσεων
- Χρήσιμη όταν ψάχνουμε μια λύση
- Ο αναμενόμενος χρόνος εξαρτάται από τη σειρά των υποψήφιων λύσεων
- Ελέγχουμε τις πιο πολλά υποσχόμενες λύσεις πρώτα

The Travelling Salesman Problem (1/4)

Πρέπει να βρεθεί το συντομότερο μονοπάτι ανάμεσα σε n πόλεις με μια επίσκεψη σε κάθε πόλη και με τελικό προορισμό την πόλη - αφετηρία

Μοντελοποιείται σαν γράφος με βάρη

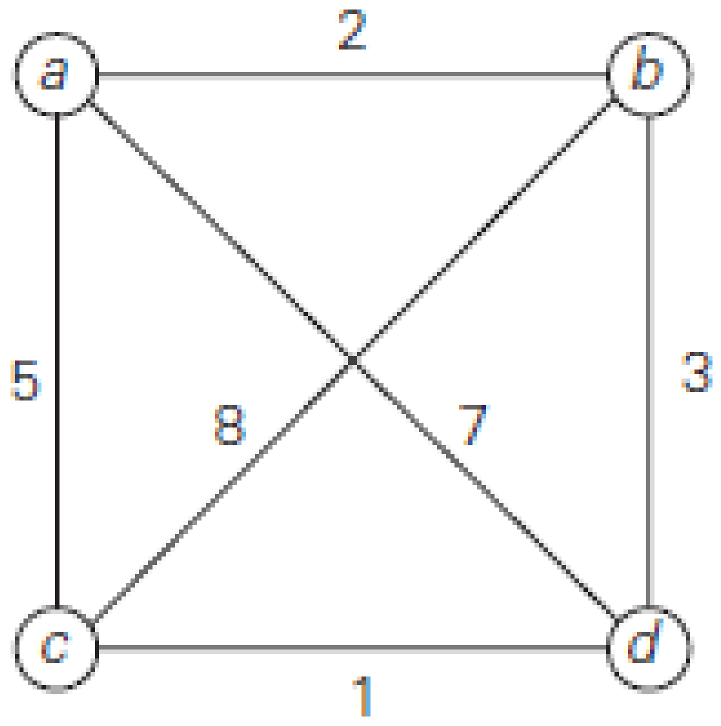
Οι κόμβοι είναι οι πόλεις και οι ακμές είναι οι διαδρομές

Τα βάρη είναι οι αποστάσεις των πόλεων

Μπορούμε να πάρουμε όλες τις διαδρομές των n-1 πόλεων, να υπολογίσουμε το μήκος των συνολικών διαδρομών και να βρούμε τη συντομότερη

The Travelling Salesman Problem (2/4)

Παράδειγμα



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

The Travelling Salesman Problem (3/4)

Εύρεση του συντομότερου Hamiltonian circuit του γράφου

- Ακολουθία από n + 1 γειτονικών κορυφών
- Η πρώτη είναι ίδια με την τελευταία

Απλός αλγόριθμος:

- Επέλεξε μια κορυφή ως αφετηρία
- Δημιούργησε (n-1)! permutations για τις ενδιάμεσες κορυφές
- Για κάθε κύκλο, υπολόγισε το συνολικό κόστος
- Υπολόγισε το μικρότερο κόστος

The Travelling Salesman Problem (4/4)

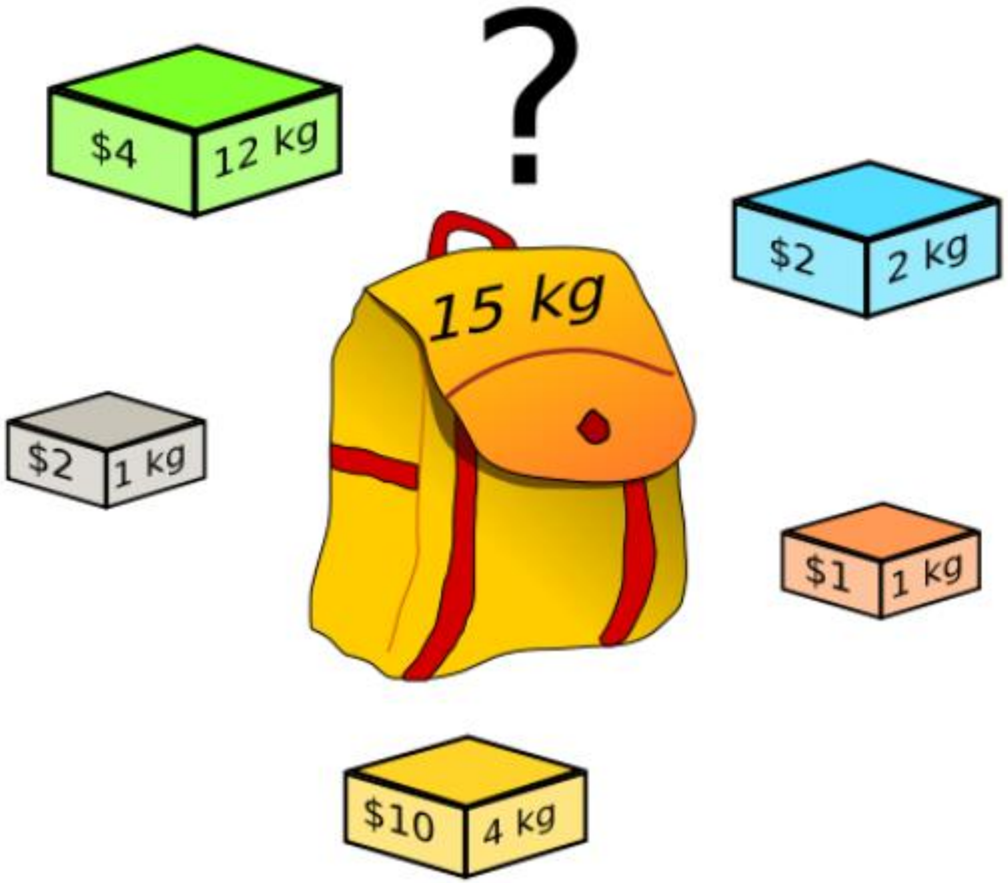
Επίδοση

- O(n!)
- Μπορεί να εφαρμοστεί μόνο για μικρού μεγέθους προβλήματα

The Knapsack Problem (1/4)

Δεδομένων n αντικειμένων με βάρη w_i , αξία $υ_i$ και ένα σακούλι χωρητικότητας W, να βρεθούν τα μεγαλύτερης αξίας αντικείμενα που χωράνε στο σακούλι.

Δημιουργούμε όλα τα υπο-σύνολα αντικειμένων, υπολογίζουμε το συνολικό βάρος τους, αναγνωρίζουμε τα υπο-σύνολα που χωράνε στο σακούλι και επιλέγουμε το υπο-σύνολο με τη μεγαλύτερη αξία



The Knapsack Problem (2/4)

Το πλήθος των υπο-συνόλων είναι 2^n

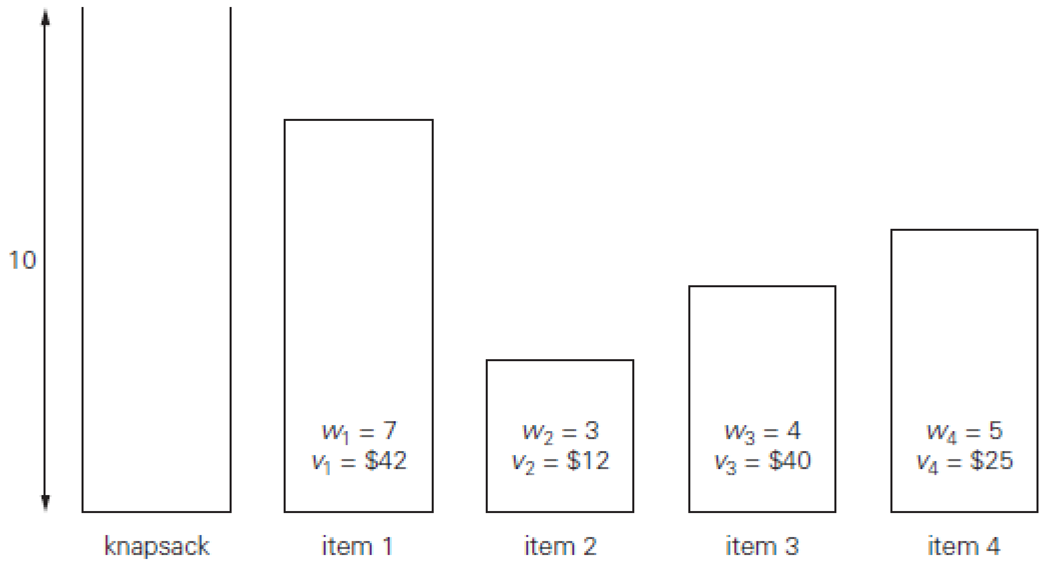
Πολυπλοκότητα: $\Omega(2^n)$

Μαζί με το traveling salesman problem ανήκουν στην κατηγορία των NP-hard προβλημάτων

Τα NP-hard προβλήματα δεν μπορούν να περιγραφούν από πολυωνυμικούς αλγορίθμους και θα συζητηθούν σε επόμενα μαθήματα

The Knapsack Problem (3/4)

Παράδειγμα:



Subset	Total weight	Total value
∅	0	\$ 0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1, 2}	10	\$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	\$52
{2, 4}	8	\$37
{3, 4}	**9**	**\$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

The Knapsack Problem (4/4)

Φορμαλισμός

	$\max \sum x_i v_i$
subject to	$\sum x_i w_i \leq W$
with	x_i in $\{0, 1\}$

The Assignment Problem (1/5)

Υπάρχουν n άνθρωποι στους οποίους πρέπει να ανατεθούν n εργασίες. Το κόστος ανάθεσης μιας εργασίας σε κάποιον είναι C[i,j] για κάθε ζεύγος. Να βρεθεί η λύση με το μικρότερο συνολικό κόστος.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

The Assignment Problem (2/5)

Ο πίνακας κόστους καθορίζει την τελική λύση

Οι πιθανές λύσεις περιγράφονται με tuples $\langle j_1, \ldots, j_n \rangle$

Στην i θέση περιγράφεται η στήλη του στοιχείου που έχει επιλεγεί στην i γραμμή

Το tuple <2, 3, 4, 1> υποδηλώνει ότι στο άτομο 1 θα ανατεθεί η εργασία 2, στο άτομο 2 η εργασία 3, στο άτομο 3 η εργασία 4, κ.ο.κ.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

The Assignment Problem (3/5)

Παράδειγμα

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

etc.

The Assignment Problem (4/5)

Φορμαλισμός

$$\min \sum \sum x[i,j]\ c[i,j]$$

subject to $\sum x[i,j] = 1$, for every i

$\sum x[i,j] = 1$, for every j

with $x[i,j]$ in $\{0, 1\}$

The Assignment Problem (5/5)

Απλός αλγόριθμος:

- Generate all permutations
- Compute the total cost for each permutation
- Find the smallest cost

Μια και το πλήθος των αντιμεταθέσεων (permutations) είναι n!, η μέθοδος exhaustive search δεν είναι καθόλου πρακτική

Πολυπλοκότητα $\Omega(n!)$

Υπάρχει πιο αποδοτικός αλγόριθμος: Hungarian method

The Hungarian Method (1/2)

[1]. Identify least element in row, subtract value of element from all elements in row. (This creates at least one zero). Repeat for all rows.

[2]. If a column contains more than one zero (this implies that two objects can be assigned at equal weight, and one assignment has no determined lowest value), repeat [1] for all columns.

[3]. Select elements in columns for which a distinct minimum weight has been determined and add to solution.

[4]. If full solution has not been achieved (implying indeterminate assignments still present), flag rows without solutions. Flag all columns in flagged rows that contain a zero. Flag all rows with a previously determined solution in previously flagged columns.

[5]. From elements remaining in UNFLAGGED columns and FLAGGED rows, determine least element. Subtract from every unflagged element, and add to every element that has been flagged twice.

[6]. Repeat [3] – [5] until full solution has been achieved.

The Hungarian Method (2/2)

Επίδοση

- Αρχικά $O(n^4)$
- Μετά από βελτιώσεις $O(n^3)$

Αλγόριθμοι Ταξινόμησης

Εισαγωγή (1/3)

Η έρευνα για αποδοτικούς αλγόριθμους ταξινόμησης έχει τις ρίζες της από τις αρχές της εμφάνισης της Επιστήμης Η/Υ

Μια συλλογή αντικειμένων για τα οποία μπορεί να υλοποιηθεί σύγκριση πρόκειται να ταξινομηθούν

Η πρόθεση είναι να βάλουμε τα αντικείμενα σε τέτοια σειρά ώστε A[i] <= A[j] για i<j

Αν υπάρχουν διπλά αντικείμενα τότε αυτά πρέπει να είναι συνεχόμενα

Η ταξινομημένη λίστα πρέπει να είναι μια αναδιάταξη (permutation) των αντικειμένων

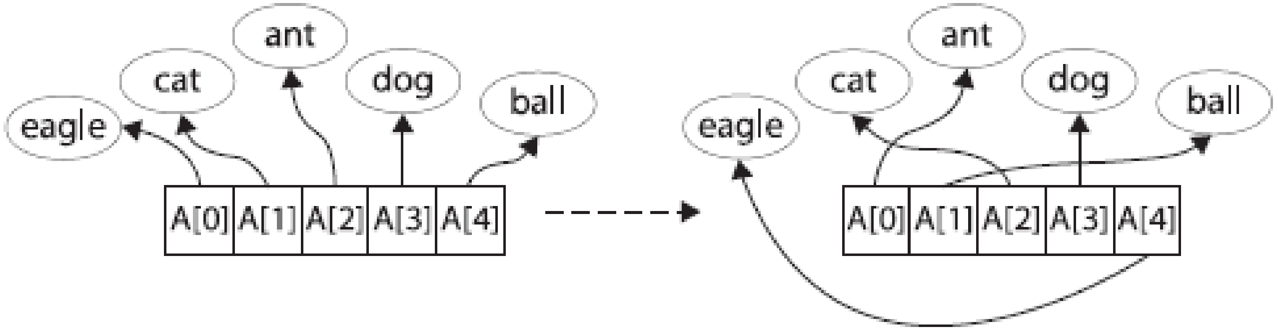
Εισαγωγή (2/3)

Τα στοιχεία / αντικείμενα μπορεί να είναι στη μνήμη ή στο σύστημα αρχείων

Η αποθηκευμένες πληροφορίες στη μνήμη είναι είτε pointer-based ή value-based

Pointer based

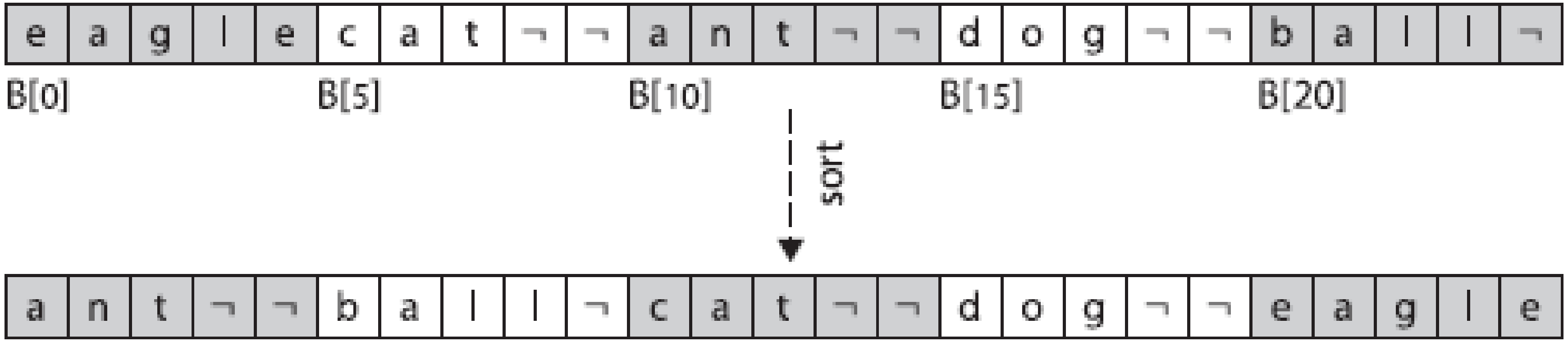
- Αποθηκεύουμε τους δείκτες προς την πληροφορία



Εισαγωγή (3/3)

Value based

- Αποθηκεύουμε τα στοιχεία σε record blocks



Insertion Sort

Εισαγωγή

Αποδοτικός αλγόριθμος για ταξινόμηση μικρού πλήθους στοιχείων

Ανήκει στην κατηγορία των αλγορίθμων **Decrease and Conquer**

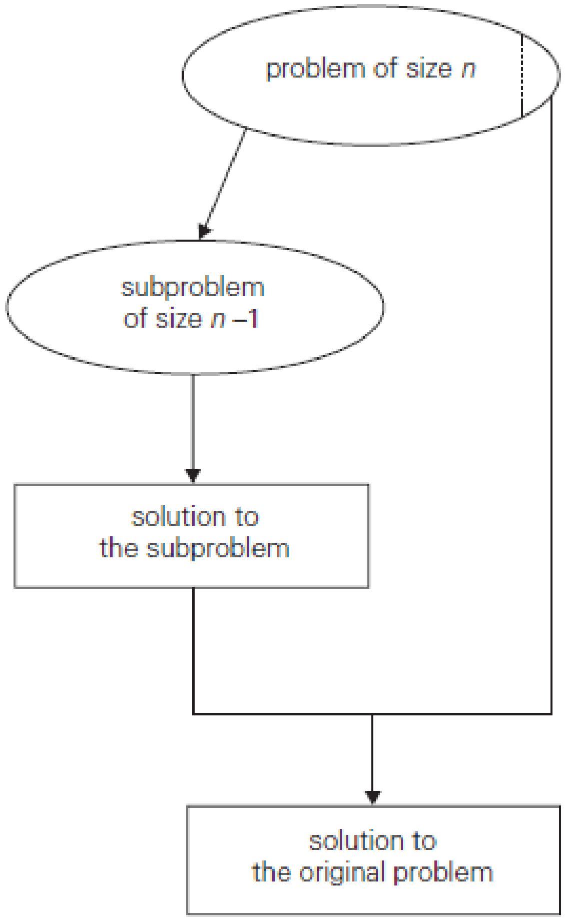
- Αποτιμά τη σχέση μιας λύσης με τη λύση του προβλήματος σε μικρότερο μέγεθος του προβλήματος
- Αν η σχέση οριστεί, τότε η λύση έρχεται είτε ακολουθώντας μια bottom-up ή μια top-down προσέγγιση
- Η top-down οδηγεί σε μια αναδρομική σχέση (υπάρχουν περιπτώσεις που λύνονται με μη αναδρομικές σχέσεις)
- Η bottom-up υλοποιείται επαναληπτικά ξεκινώντας από τη λύση στο μικρότερο μέγεθος και ακολουθώντας μια **incremental** προσέγγιση

Decrease and Conquer (1/3)

Τρεις παραλλαγές:

- Μείωση κατά μια σταθερά
  - Σε κάθε επανάληψη μειώνουμε κατά τη σταθερά
  - Υπολογισμός του $α^ν$
  - $α^ν = α^{ν-1}\ α$
  - Οπότε

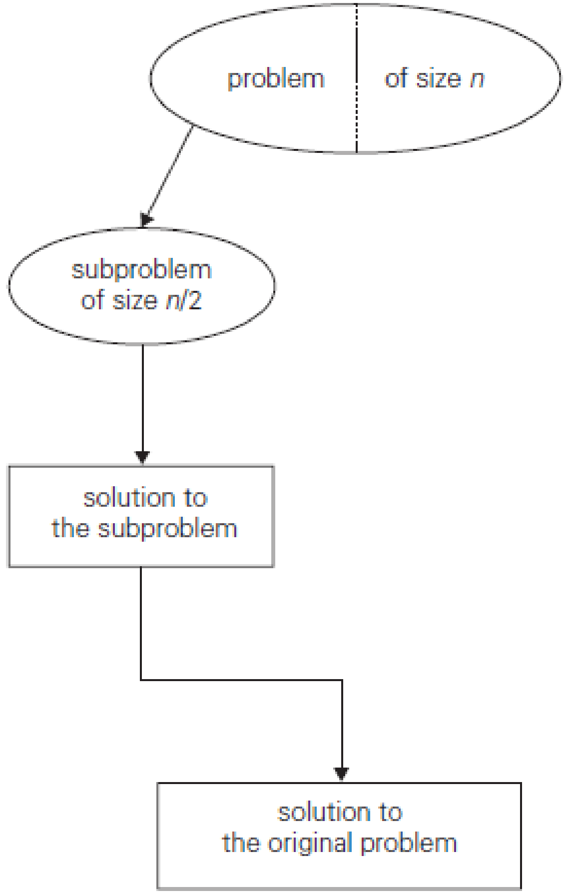
$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$



Decrease and Conquer (2/3)

- Μείωση κατά ένα σταθερό παράγοντα
  - Στις περισσότερες εφαρμογές ο παράγοντας είναι ίσος με 2
  - Υπολογισμός του α^{ν}
  - $\alpha^{\nu} = (\alpha^{\nu/2})^2$
  - Αν το ν είναι περιττός, πρέπει να υπολογίσουμε το $\alpha^{\nu-1}$ και μετά να πολλαπλασιάσουμε με το α
  - Οπότε

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$



Decrease and Conquer (3/3)

- Μεταβλητό μέγεθος μείωσης
  - Το pattern μείωσης είναι μεταβλητό σε κάθε επανάληψη
  - Παράδειγμα: Μέγιστος κοινός διαιρέτης

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Insertion Sort (1/9)

Εφαρμόζει μείωση κατά ένα

Υποθέτουμε ότι το μικρότερο πρόβλημα της ταξινόμησης των n-2 στοιχείων έχει ήδη επιλυθεί για την ταξινόμηση των n-1 στοιχείων

Βρίσκουμε την κατάλληλη θέση για ένα στοιχείο μέσα στα n-1 και το τοποθετούμε εκεί

Το στοιχείο A[i] μπαίνει στην κατάλληλη θέση

Η προσπέλαση στα στοιχεία γίνεται ξεκινώντας από αριστερά προς τα δεξιά

Insertion Sort (2/9)

ALGORITHM *InsertionSort*$(A[0..n-1])$

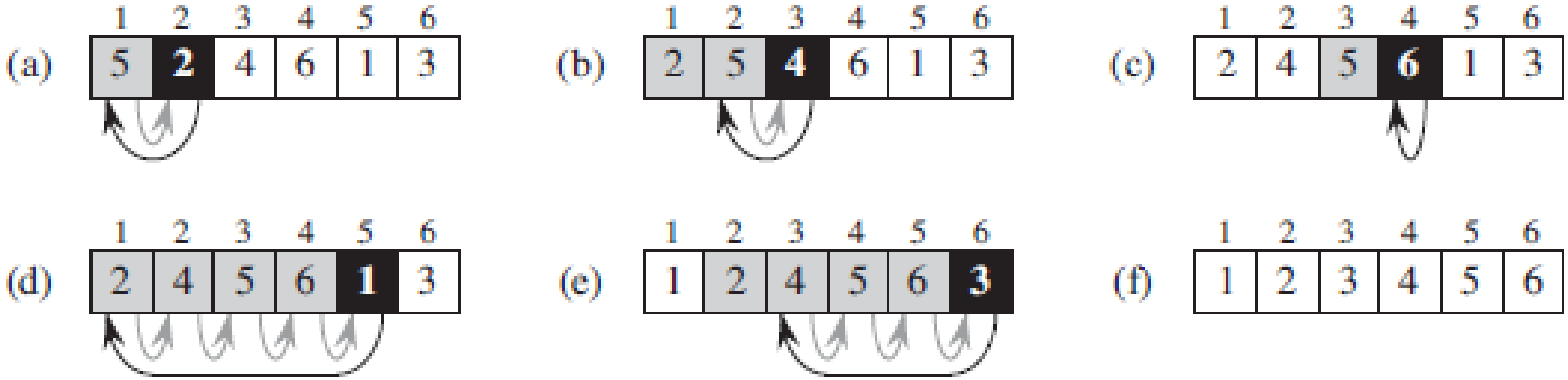
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

$$A[0] \leq \cdots \leq A[j] < A[j+1] \leq \cdots \leq A[i-1] \mid A[i] \cdots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

Insertion Sort (3/9)

$$A = \langle 5, 2, 4, 6, 1, 3 \rangle$$



Insertion Sort (4/9)

Βασική λειτουργία

- Σύγκριση A[j] > υ

Στη χειρότερη περίπτωση η σύγκριση εκτελείται για κάθε τιμή του j

Η χειρότερη περίπτωση είναι ένας πίνακας με στοιχεία σε φθίνουσα σειρά

Το πλήθος των συγκρίσεων στη χειρότερη περίπτωση είναι:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Insertion Sort (5/9)

Στην καλύτερη περίπτωση, η σύγκριση θα εκτελεστεί μόνο μια φορά για κάθε επανάληψη του εξωτερικού loop

Στην καλύτερη περίπτωση ο αρχικός πίνακας είναι ήδη ταξινομημένος

Πλήθος συγκρίσεων

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Στην ουσία, δεν είναι χρήσιμη προσέγγιση

Insertion Sort (6/9)

Στη μέση περίπτωση ο αλγόριθμος εκτελεί περίπου τις μισές συγκρίσεις από ότι στη χειρότερη περίπτωση

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

Insertion Sort (7/9)

Εναλλακτική προσέγγιση στην ανάλυση του αλγορίθμου:

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j - 1]$.	0	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i + 1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i + 1] = key$	c_8	$n - 1$

Insertion Sort (8/9)

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \end{aligned}$$

Καλύτερη περίπτωση (γραμμική συνάρτηση)

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Insertion Sort (9/9)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Χειρότερη περίπτωση

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Insertion Sort (10/10)

Demo

http://visualgo.net/sorting.html#

http://www.sorting-algorithms.com/insertion-sort

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Selection Sort

Selection Sort (1/6)

Αναζητά στη λίστα των αντικειμένων το μικρότερο στοιχείο

Τοποθετεί το μικρότερο στοιχείο στη θέση του

Αρχικά ξεκινά από την πρώτη θέση της λίστας

Έπειτα προχωρά στη δεύτερη, την τρίτη, κ.ο.κ.

Κάθε φορά το μικρότερο στοιχείο τοποθετείται στη θέση του ($1^η$, $2^η$, $3^η$, …)

Ανήκει στην κατηγορία των Brute Force αλγορίθμων

Selection Sort (2/6)

Γενικά ο αλγόριθμος αναζητά το i^th μικρότερο στοιχείο στα n-1 στοιχεία και το τοποθετεί στην i θέση

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \mid \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

Μετά από n-1 περάσματα η λίστα έχει ταξινομηθεί

Selection Sort (3/6)

Αλγόριθμος

ALGORITHM *SelectionSort*$(A[0..n-1])$

//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

Selection Sort (4/6)

Παράδειγμα

- Ταξινόμηση των 89, 45, 68, 90, 29, 34, 17

| 89 45 68 90 29 34 **17**

17 | 45 68 90 **29** 34 89

17 29 | 68 90 45 **34** 89

17 29 34 | 90 **45** 68 89

17 29 34 45 | 90 **68** 89

17 29 34 45 68 | 90 **89**

17 29 34 45 68 89 | 90

Selection Sort (5/6)

Ανάλυση πολυπλοκότητας

- Μέγεθος εισόδου: n
- Βασική λειτουργία: A[j] < A[min]
- Πλήθος εκτελέσεων

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

- Πολυπλοκότητα: $\Theta(n^2)$
- Πολυπλοκότητα των αντιμεταθέσεων: $\Theta(n)$

Selection Sort (6/6)

Demo

http://visualgo.net/sorting.html#

http://www.sorting-algorithms.com/selection-sort

Bubble Sort

Bubble Sort (1/7)

Ανήκει στην κατηγορία των Brute Force αλγορίθμων

Συγκρίνει γειτονικά στοιχεία και τα ανταλλάσσει αν είναι εκτός σειράς

Οι συνεχόμενες αντιμεταθέσεις προκαλούν τη μετακίνηση και τοποθέτηση του κάθε στοιχείου στη θέση του (προσομοίωση φυσαλίδας)

Στο 1^{o} πέρασμα, το 1^{o} στοιχείο μπαίνει στη θέση του, στο 2^{o} πέρασμα, το 2^{o} στοιχείο μπαίνει στη θέση του, κ.ο.κ.

Bubble Sort (2/7)

Στο πέρασμα i (0<= i <= n-2) η εκτέλεση του αλγορίθμου έχει ως εξής:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \leq \cdots \leq A_{n-1}}_{\text{in their final positions}}$$

Bubble Sort (3/7)

Αλγόριθμος

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Bubble Sort (4/7)

Παράδειγμα

- Ταξινόμηση των 89, 45, 68, 90, 29, 34, 17

89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17	|	90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17	|	90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17	|	90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17	|	90
45		68		29		34		17	|	89		90

etc.

Bubble Sort (5/7)

Ανάλυση πολυπλοκότητας

- Μέγεθος εισόδου: n
- Βασική λειτουργία: A[j+1] < A[j]
- Πλήθος εκτελέσεων

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

- Πολυπλοκότητα: $\Theta(n^2)$
- Πολυπλοκότητα των αντιμεταθέσεων: $\Theta(n^2)$

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Bubble Sort (6/7)

Βελτιωμένη έκδοση

```
for(int i = 0 ; i &lt; a.length ; i++){
        boolean swap = false;
        for(int j = 0 ; j&lt; a.length - i-1 ; j++){
                if(a[j] &gt; a[j+1]){
                        temp = a[j];
                        a[j] = a[j+1];
                        a[j+1]=temp;
                        swap=true;
                }
        }
        if(!swap)
                break;
}
```

Bubble Sort (7/7)

Demo

http://visualgo.net/sorting.html#

http://www.sorting-algorithms.com/bubble-sort

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Merge Sort

Merge Sort (1/8)

Χαρακτηριστικό παράδειγμα της μεθόδου διαίρει και βασίλευε

Διαιρεί τη λίστα στη μέση και ταξινομεί τα τμήματα αναδρομικά

ALGORITHM *Mergesort*($A[0..n-1]$)

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$
  copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$
  Mergesort($B[0..\lfloor n/2\rfloor - 1]$)
  Mergesort($C[0..\lceil n/2\rceil - 1]$)
  Merge(B, C, A)

Merge Sort (2/8)

Διαδικασία συγχώνευσης

- Δύο δείκτες υιοθετούνται ώστε να δείχνουν στο πρώτο στοιχείο των λιστών που πρόκειται να συγχωνευτούν
- Τα δύο στοιχεία ελέγχονται και το μικρότερο μπαίνει στη νέα λίστα που δημιουργείται
- Ο δείκτης του μικρότερου στοιχείου ‘προχωρά’ στο επόμενο στοιχείο
- Η διαδικασία επαναλαμβάνεται μέχρι να εξαντληθούν οι πίνακες
- Αν εξαντληθεί ο ένας πίνακας τότε τα στοιχεία του άλλου αντιγράφονται στη νέα δομή

Merge Sort (3/8)

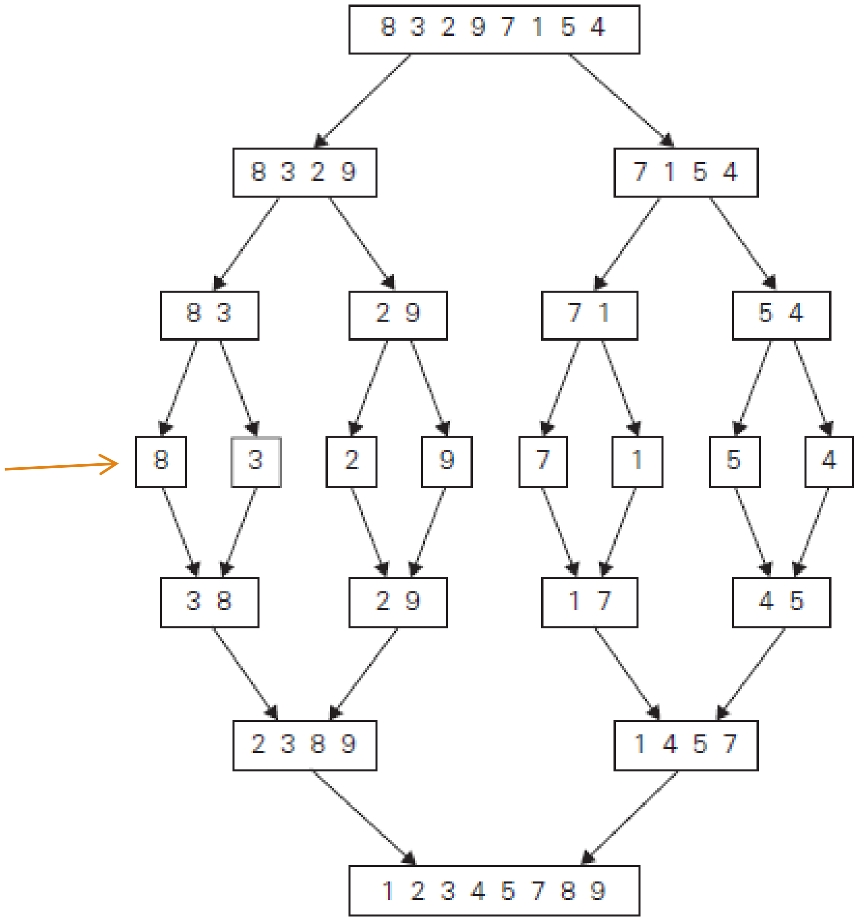
Αλγόριθμος συγχώνευσης

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
    else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Merge Sort (4/8)

Παράδειγμα: ταξινόμηση των 8, 3, 2, 9, 7, 1, 5, 4



Merge Sort (5/8)

Ανάλυση

- Πλήθος εκτελέσεων

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0$$

- Συγχώνευση
  - Σε κάθε βήμα, μια σύγκριση γίνεται και έπειτα ο συνολικός αριθμός των στοιχείων προς επεξεργασία μειώνεται κατά 1
  - Στη χειρότερη περίπτωση καμία από τις δύο υπο-λίστες δεν ‘αδειάζει’ πριν από την άλλη
  - Άρα $C_{merge}(n) = n - 1$

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0$$

Merge Sort (6/8)

Εφαρμόζουμε το master θεώρημα και έχουμε:

$$C_{worst}(n) = n \log_2 n - n + 1$$

$n = 2^k$:

$$C_{worst}(n) \in \Theta(n \log n)$$

Merge Sort (7/8)

Το πλήθος των συγκρίσεων στη χειρότερη περίπτωση είναι πολύ κοντά στο θεωρητικό ελάχιστο

Για μεγάλο n το πλήθος των συγκρίσεων στη μέση περίπτωση είναι 0.25n

Το πλεονέκτημα του merge sort σε σύγκριση με τους heap and quick sort είναι η σταθερότητα που επιδεικνύει

Το μειονέκτημα του αλγορίθμου είναι η γραμμική ποσότητα επιπλέον χώρου που απαιτεί

Merge Sort (8/8)

Demo

https://visualgo.net/en/sorting?slide=1

http://www.sorting-algorithms.com/merge-sort

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Quick Sort

Quick Sort (1/30)

Ανήκει στην κατηγορία των διαίρει και βασίλευε αλγορίθμων

Παρά το γεγονός ότι στη χειρότερη περίπτωση έχει πολυπλοκότητα $\Theta(n^2)$ υιοθετείται πρακτικά αφού έχει πολύ καλή επίδοση στη μέση περίπτωση

Έχει επίσης πολύπλοκη λογική εκτέλεσης

Quick Sort (2/30)

Ταξινόμηση ενός πίνακα A[p,r]

- Χωρισμός του πίνακα σε δύο (πιθανώς άδειους) υπο-πίνακες A[p,q-1], A[q+1,r] τέτοιοι ώστε κάθε στοιχείο του A[p,q-1] να είναι μικρότερο ή ίσο του A[q] και κάθε στοιχείο του A[q+1,r] να είναι μεγαλύτερο ή ίσο του A[q] – Υπολογισμός / εύρεση του A[q]
- Ταξινόμηση των δύο υπο-πινάκων A[p,q-1], A[q+1,r] μέσω αναδρομικών κλήσεων του αλγορίθμου quicksort
- Μια και οι δύο υπο-πίνακες είναι ταξινομημένοι, δεν απαιτείται προσπάθεια για συγχώνευση τους – Ο πίνακας A είναι ήδη ταξινομημένος

Quick Sort (3/30)

Η αρχική κλήση του αλγορίθμου είναι: QUICKSORT(A, 1, A.length)

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

Quick Sort (4/30)

Το σημαντικότερο τμήμα του αλγορίθμου είναι η εύρεση του στοιχείου A[q] και ο διαχωρισμός του αρχικού πίνακα

Ο αλγόριθμος διαχωρισμού έχει ως εξής:

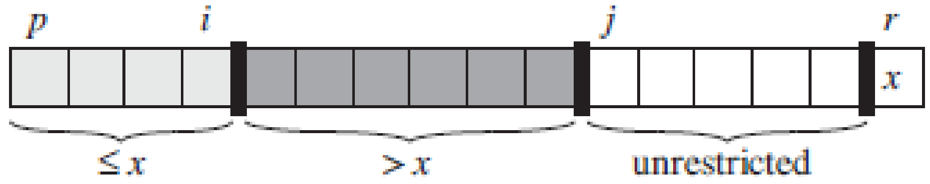
```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quick Sort (5/30)

Επιλογή του στοιχείου A[r] ως pivot γύρω από το οποίο θα γίνει ο διαχωρισμός του A[p,r]

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Ο πίνακας χωρίζεται σε 4 περιοχές (πιθανώς άδειες)



Στην αρχή κάθε επανάληψης (γραμμές 3-6), οι περιοχές ικανοποιούν τις ακόλουθες ιδιότητες (για κάθε k)

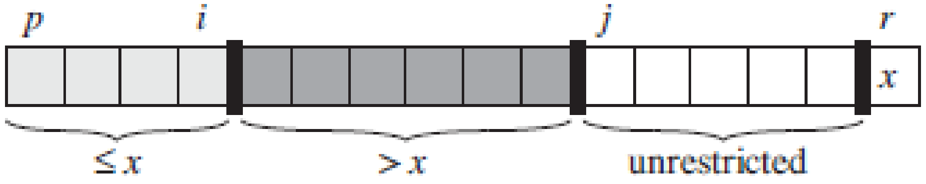
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Quick Sort (6/30)

Οι δείκτες j και r-1 ορίζουν μια περιοχή στην οποία τα στοιχεία δεν έχουν κάποια ιδιαίτερη σχέση με το pivot x

Πριν από την 1^{η} επανάληψη, θέτουμε i=p-1, j=p. Αφού δεν υπάρχουν τιμές στην περιοχή που ορίζεται από τα i+1, j-1, οι πρώτες δύο συνθήκες ικανοποιούνται

```
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



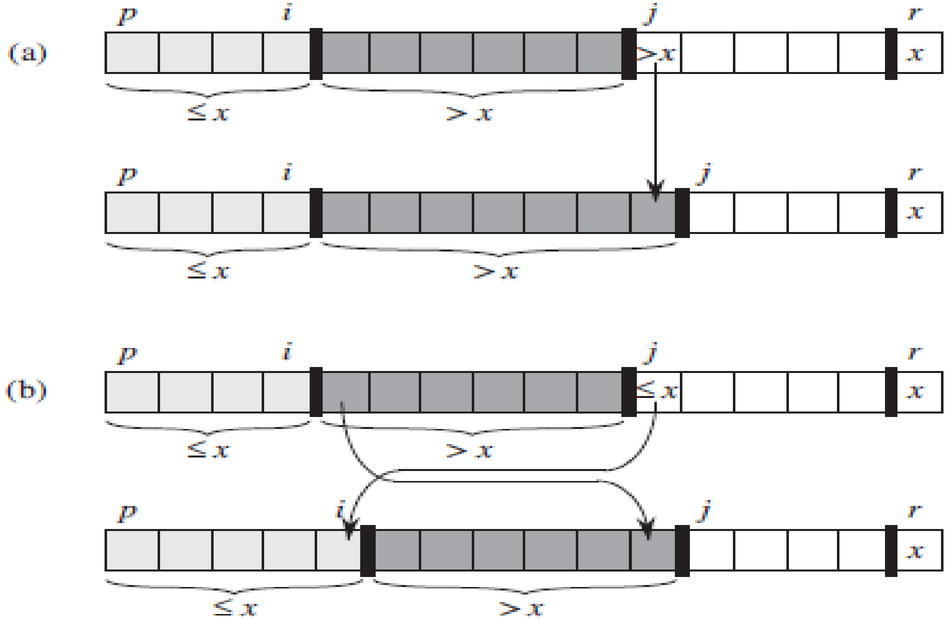
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Quick Sort (7/30)

Όπως φαίνεται στην εικόνα, με βάση τη συνθήκη στη γραμμή 4, αν A[j]>x τότε αυξάνεται η τιμή του j κατά 1. Μόνο το condition 2 ικανοποιείται και όλα τα άλλα στοιχεία παραμένουν όπως έχουν

Όταν A[j]<=x, το i αυξάνει κατά 1 και γίνεται ανταλλαγή των A[i], A[j] και έπειτα αυξάνει το j κατά 1 - Με την ανταλλαγή έχουμε A[i]<=x και ικανοποιούμε την $1^{η}$ συνθήκη – Επίσης έχουμε ότι A[j-1]>x

```
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



Quick Sort (8/30)

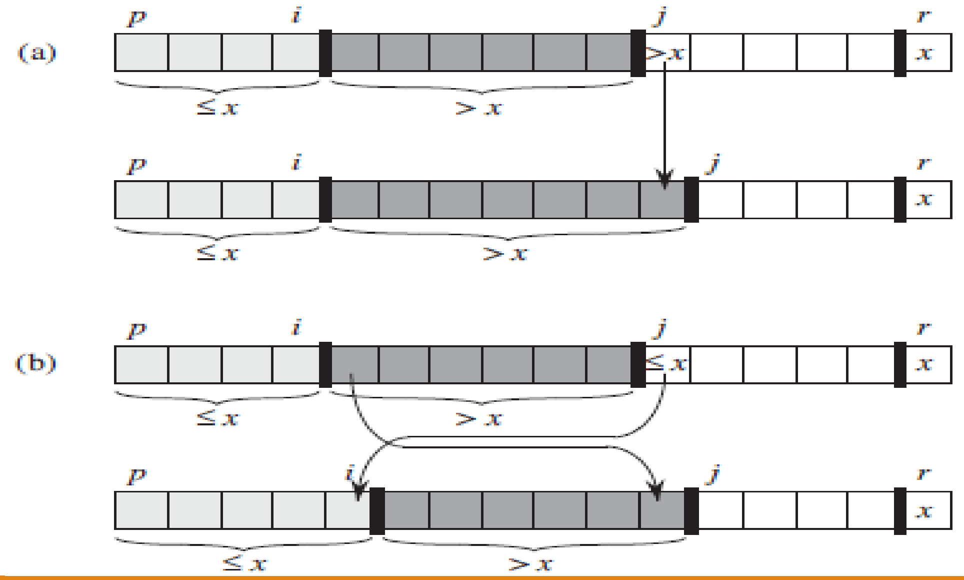
Ο τερματισμός του αλγορίθμου

Με βάση τα προηγούμενα κάθε στοιχείο του πίνακα έχει μπει σε ένα από τρία σύνολα: μικρότερα ή ίσα του x, μεγαλύτερα του x και ένα μονοσύνολο, το x

Οι τελευταίες δύο γραμμές του αλγορίθμου ανταλλάσσουν το x με το πιο αριστερά στοιχείο της περιοχής με τα στοιχεία που είναι μεγαλύτερα του x

Στη συνέχεια ο αλγόριθμος επιστρέφει το νέο δείκτη του pivot

```
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



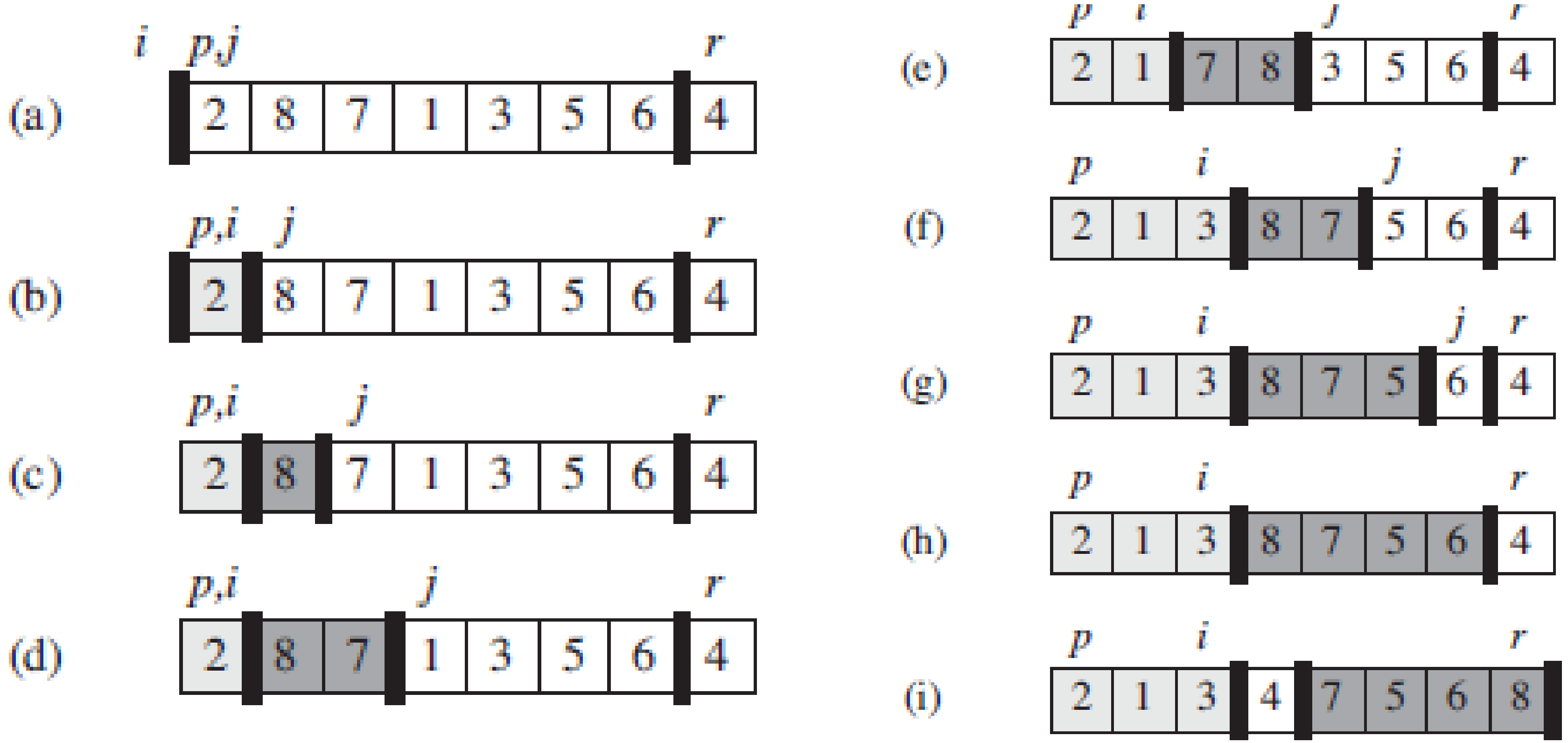
Quick Sort (9/30)

Ο χρόνος εκτέλεσης του τμήματος του διαχωρισμού έχει πολυπλοκότητα Θ(n)

Για τον πίνακα A[p,r] έχουμε ότι n=r-p+1

Quick Sort (10/30)

Παράδειγμα εκτέλεσης




```
for j = p to r – 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quick Sort (11/30)

Ο χρόνος εκτέλεσης εξαρτάται από το αν το πλήθος των στοιχείων στους υπο-πίνακες είναι ισορροπημένο

Εξαρτάται από τα στοιχεία που χρησιμοποιούνται για το διαχωρισμό

Αν τα στοιχεία είναι ισοκατανεμημένα ο αλγόριθμος είναι ασυμπτωτικά τόσο γρήγορος όσο το merge sort

Αν τα στοιχεία δεν είναι ισοκατανεμημένα, ο αλγόριθμος είναι ασυμπτωτικά τόσο αργός όσο το insertion sort

Quick Sort (12/30)

Η χειρότερη περίπτωση είναι όταν ο αλγόριθμος παράγει ένα υπο-πρόβλημα που περιλαμβάνει δύο υπο-πίνακες, ένα με n-1 στοιχεία και ένα με 0 στοιχεία (άδειος υπο-πίνακας)

Για τη χειρότερη περίπτωση, υποθέτουμε ότι ο παραπάνω διαχωρισμός ισχύει για όλες τις αναδρομικές κλήσεις

Ο διαχωρισμός κοστίζει Θ(n) και η αναδρομική κλήση σε ένα πίνακα μεγέθους 0 έχει κόστος Τ(0) = Θ(1) (μόνο το return εκτελείται)

Συνεπώς:
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Quick Sort (13/30)

Με τη μέθοδο της αντικατάστασης μπορούμε να αποδείξουμε ότι $T(n)=\Theta(n^2)$

Στη χειρότερη περίπτωση, ο αλγόριθμος έχει ίδια πολυπλοκότητα με τον insertion sort

Η χειρότερη πολυπλοκότητα συναντάται επίσης όταν ο πίνακας είναι ήδη ταξινομημένος – ο insertion sort σε αυτή την περίπτωση έχει πολυπλοκότητα $O(n)$

Quick Sort (14/30)

Στην καλύτερη περίπτωση, το τμήμα διαχωρισμού παράγει δύο υπο-πίνακες μεγέθους περίπου n/2 (floor(n/2), ceiling(n/2)-1)

Η αναδρομική σχέση είναι: **$T(n)=2T(n/2)+\Theta(n)$**

Με βάση το master θεώρημα (περίπτωση 2) έχουμε: **$T(n) = \Theta(n \log n)$**

Quick Sort (15/30)

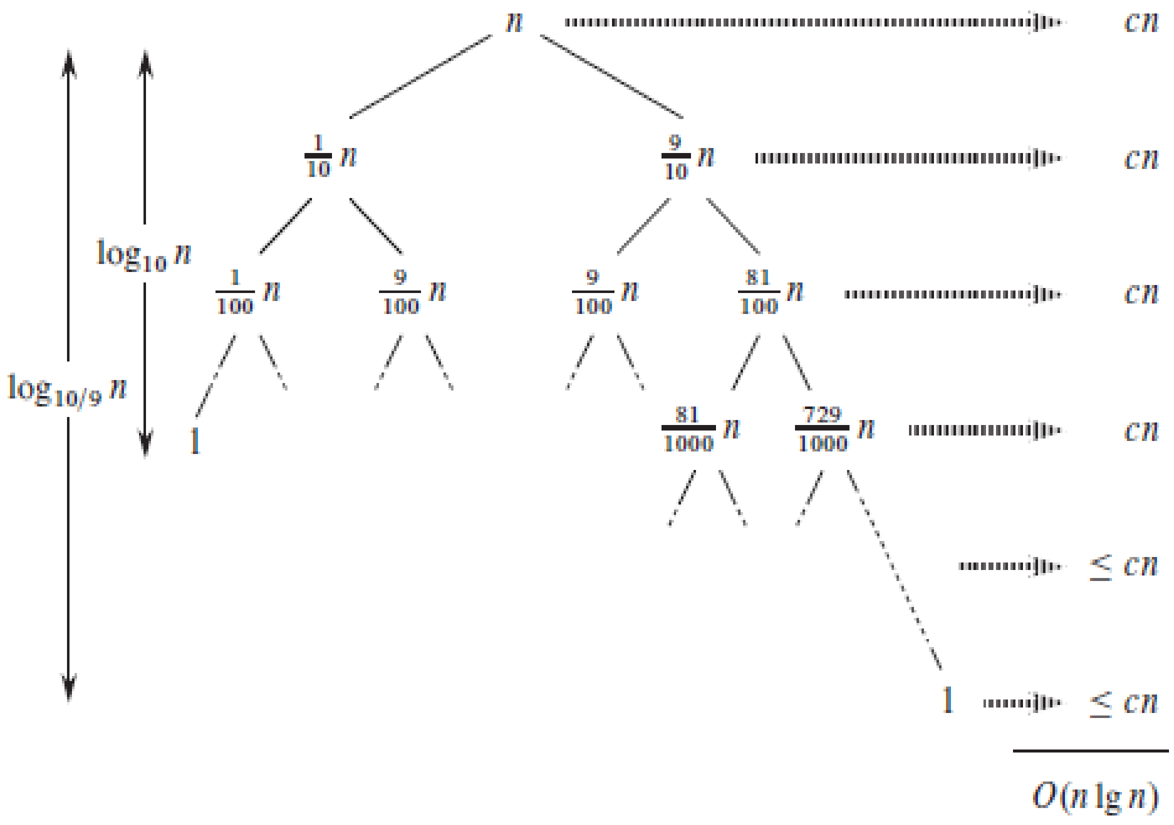
Η μέση περίπτωση είναι πιο κοντά στην καλύτερη περίπτωση παρά στη χειρότερη

Ας υποθέσουμε ότι ο διαχωρισμός παράγει 9-προς-1 αναλογία

Παίρνουμε την εξής αναδρομική σχέση

T(n)=T(9n/10)+T(n/10)+cn

Το βάθος του δένδρου είναι $\log_{10}n=\Theta(\log n)$

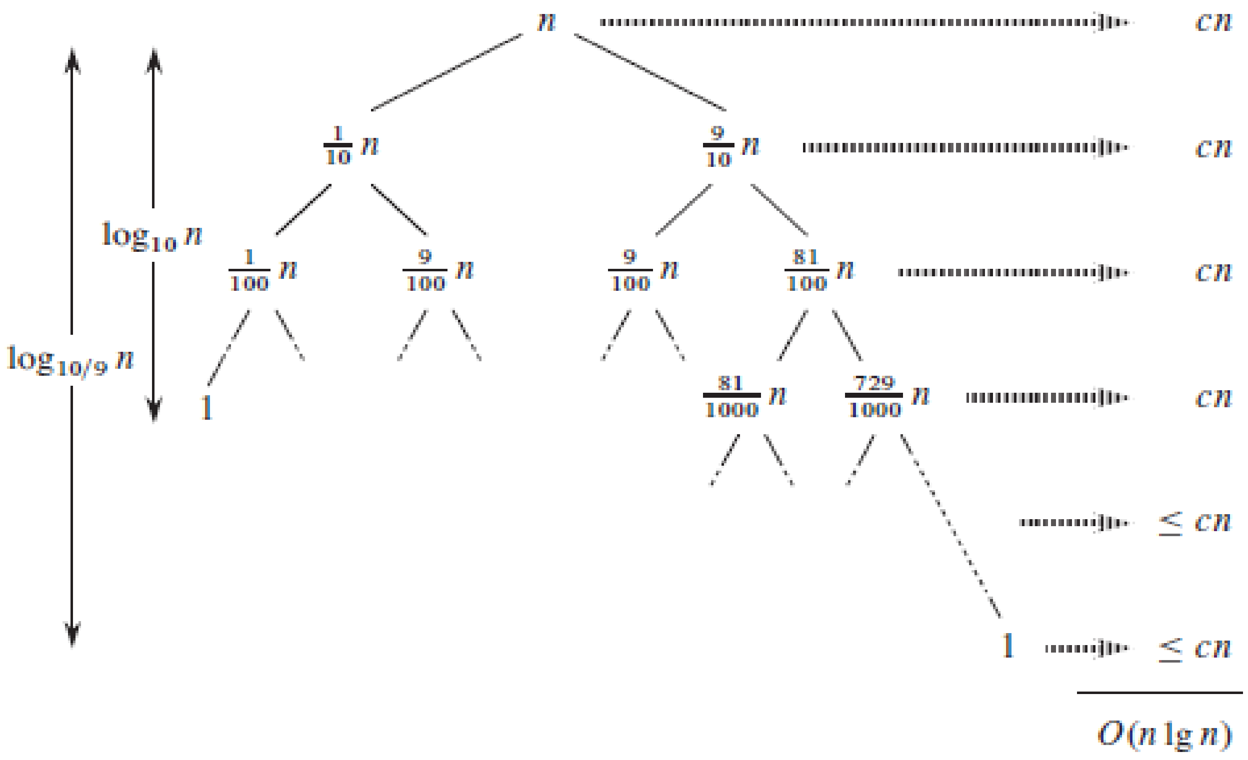


Quick Sort (16/30)

Η αναδρομή τερματίζει στο $\log_{10/9}n=\Theta(\log n)$

Συνολικό κόστος: $O(n \log n)$

Βλέπουμε πως παρά το γεγονός ότι ο αρχικός διαχωρισμός δεν είναι ισοκατανεμημένος, ο αλγόριθμος έχει πολυπλοκότητα $O(n \log n)$



Quick Sort (17/30)

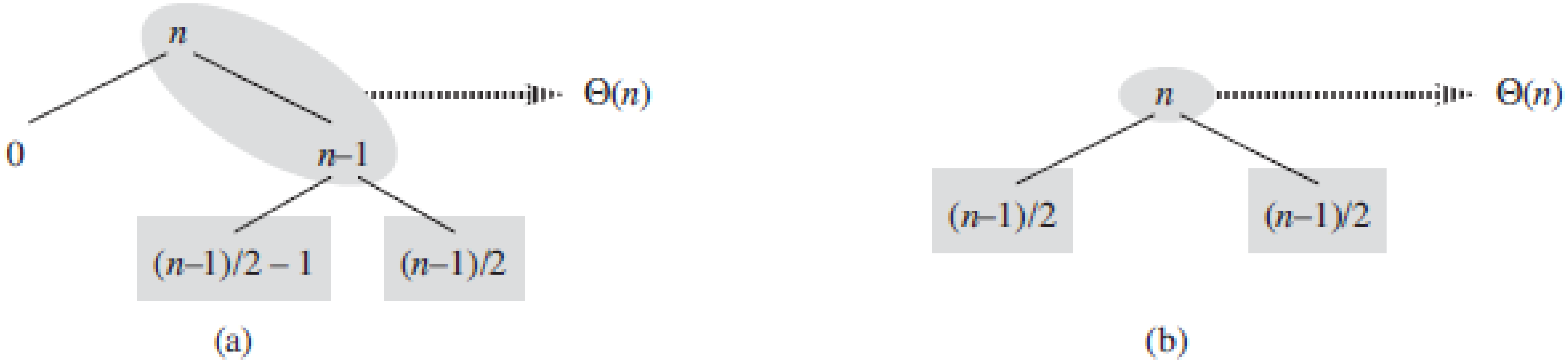
Ο αλγόριθμος εξαρτάται από την κατανομή των στοιχείων στον πίνακα και όχι από το ποια στοιχεία είναι αυτά

Αν υποθέσουμε ότι οι αναδιατάξεις είναι ισοπίθανες και τρέξουμε τον αλγόριθμο για τυχαίους πίνακες, τότε θα πάρουμε περιπτώσεις όπου οι δύο υπο-πίνακες θα είναι ισοκατανεμημένοι και άλλες όπου δεν θα έχουμε ισοκατανομή

Στη μέση περίπτωση, η κατανομή των ‘καλών’ και ‘κακών’ διαχωρισμών θα είναι τυχαία ‘τοποθετημένες’ στο δένδρο διαχωρισμού

Quick Sort (18/30)

Υποθέτουμε ότι οι ‘καλοί’ διαχωρισμοί ανήκουν στην καλύτερη περίπτωση και οι ‘κακοί’ στην χειρότερη περίπτωση



Quick Sort (19/30)

Αν έχουμε συνδυασμό ‘καλών’ και ‘κακών’ διαχωρισμών π.χ. ένας κακός διαχωρισμός ακολουθείται από ένα καλό τότε θα έχουμε για τα δύο βήματα, υπο-πίνακες μεγέθους **0, (n-1)/2-1, (n-1)/2**

Σε αυτή την περίπτωση, το κόστος του διαχωρισμού θα είναι: **Θ(n) + Θ(n-1) = Θ(n) – περίπτωση (a)**

Στην περίπτωση (b), για τους δύο υπο-πίνακες μεγέθους (n-1)/2 το κόστος διαχωρισμού είναι πάλι **Θ(n)**

Quick Sort (20/30)

Randomized version

- Αντί να επιλέξουμε για pivot το A[r], επιλέγουμε ένα τυχαίο στοιχείο του A[p,r]
- Ανταλλάσσουμε το A[r] με ένα τυχαίο στοιχείο
- Με αυτό τον τρόπο θεωρούμε ισοπίθανα το A[r] με τα υπόλοιπα r-p+1 στοιχεία
- Οι αλλαγές στους αλγορίθμους έχουν ως εξής:

```
RANDOMIZED-PARTITION(A, p, r)

i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)

if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Quick Sort (21/30)

Γενική Ανάλυση του αλγορίθμου:

- Χειρότερη περίπτωση
  - Αναδρομική σχέση

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

  - Όπου το q κινείται από το 0 μέχρι το n-1
  - Υποθέτουμε ότι T(n)<= cn^2 για κάποιο c
  - Με αντικατάσταση παίρνουμε ότι

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

Quick Sort (22/30)

Η παράσταση $q^2 + (n - q - 1)^2$ έχει δεύτερη παράγωγο θετική και έτσι μεγιστοποιείται στο [0, n-1] και παίρνουμε ότι

$$\max_{0 \le q \le n-1}(q^2 + (n - q - 1)^2) \le (n - 1)^2 = n^2 - 2n + 1$$

και

$$\begin{aligned} T(n) &\le cn^2 - c(2n - 1) + \Theta(n) \\ &\le cn^2 \end{aligned}$$

Αν επιλέξουμε ένα c αρκετά μεγάλο τέτοιο ώστε ο όρος c(2n-1) υπερισχύει του Θ(n).

Έτσι: $T(n)=O(n^2)$ – κάποιες περιπτώσεις έχουν $\Omega(n^2)$ και άρα $T(n)=\Theta(n^2)$

Quick Sort (23/30)

Ο quicksort και ο randomized quicksort διαφέρουν μόνο στην επιλογή του pivot

Ο χρόνος εκτέλεσης εξαρτάται από το χρόνο που δαπανάται στο διαχωρισμό

Το πολύ n κλήσεις μπορεί να γίνουν στο τμήμα του διαχωρισμού

Κάθε κλήση απαιτεί O(1) για την ανάθεση συν το χρόνο για το loop

Σε κάθε iteration γίνεται μια σύγκριση

Πρέπει να μετρήσουμε το πλήθος των εκτελέσεων της γραμμής 4

```
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Quick Sort (24/30)

Αν X είναι το πλήθος των συγκρίσεων (γραμμή 4) σε ολόκληρη την εκτέλεση του quicksort ο χρόνος εκτέλεσης θα είναι **O(n+X)**

Πρέπει να υπολογίσουμε το X βγάζοντας ένα όριο για το συνολικό αριθμό των συγκρίσεων

Έστω ότι τα στοιχεία του πίνακα είναι τα $z_1, z_2, \ldots, z_n$

και z_i είναι το i μικρότερο στοιχείο

Παίρνουμε το σύνολο $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

των στοιχείων ανάμεσα στα z_i and z_j

Quick Sort (25/30)

Τα στοιχεία συγκρίνονται μόνο με το pivot και μετά το πέρας του τμήματος διαχωρισμού το pivot δεν ξαναελέγχεται

Υιοθετούμε την indicator function

$$X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\}$$

που δείχνει αν μια σύγκριση λαμβάνει χώρα στον αλγόριθμο (όχι μόνο στο τμήμα διαχωρισμού)

Αφού κάθε ζεύγος συγκρίνεται μια φορά το πλήθος των συγκρίσεων θα είναι:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Quick Sort (26/30)

Παίρνοντας την αναμενόμενη τιμή έχουμε:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}
\end{aligned}$$

Μένει να υπολογίσουμε την πιθανότητα σύγκρισης ενός ζεύγους

Quick Sort (27/30)

Αν έχουμε ως pivot το x με $z_i < x < z_j$

γνωρίζουμε ότι τα δύο στοιχεία $z_{i,}$ z_j δεν πρόκειται να συγκριθούν σε επόμενες φάσεις του αλγορίθμου

Αν το z_i επιλεγεί ως pivot πριν από τα υπόλοιπα στοιχεία του Z_{ij} τότε το z_i θα συγκριθεί με τα στοιχεία του Z_{ij} εκτός από τον εαυτό του

Αν το z_j επιλεγεί ως pivot πριν από τα υπόλοιπα στοιχεία του Z_{ij} τότε το z_j θα συγκριθεί με τα στοιχεία του Z_{ij} εκτός από τον εαυτό του

Οπότε, αφού το Z_{ij} έχει j-i+1 στοιχεία και κάθε ένα από αυτά έχει την ίδια πιθανότητα να επιλεγεί ως pivot, η πιθανότητα επιλογής είναι **1/(j-i+1)**

Quick Sort (28/30)

Έτσι έχουμε:

$$
\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}
$$

Quick Sort (29/30)

Άρα:

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n)
\end{aligned}
$$

$$k = j - i$$

Quick Sort (30/30)

Demo

https://visualgo.net/en/sorting

http://www.sorting-algorithms.com/quick-sort

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Median Sort

Median Sort (1/9)

Ανήκει στην κατηγορία Διαίρει και Βασίλευε

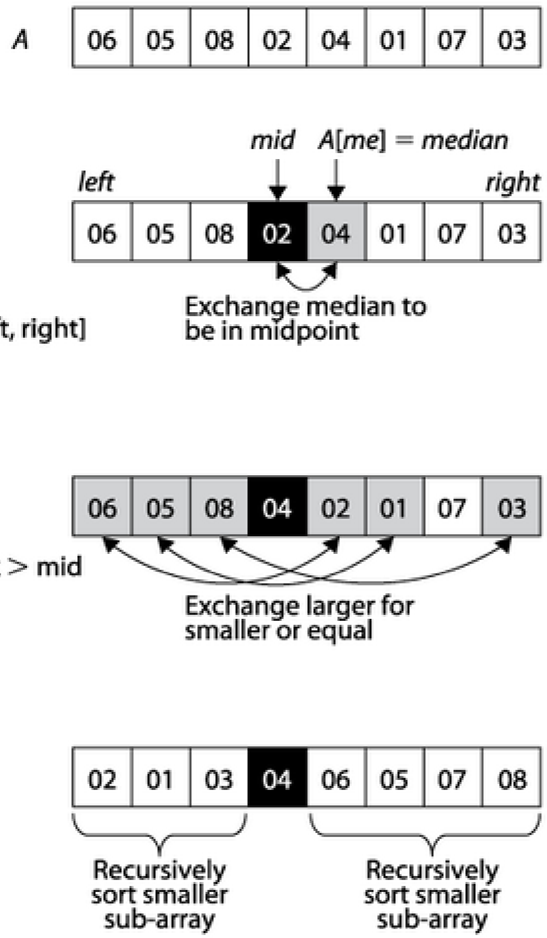
Ανταλλάσσει το μεσαίο (median) στοιχείο με το στοιχείο στο μέσο του πίνακα (middle element)

Ο αλγόριθμος ανταλλάσσει τα στοιχεία που είναι στο αριστερό μέρος και είναι μεγαλύτερα από το στοιχείο στη μέση με στοιχεία που είναι στη δεξιά πλευρά και είναι μικρότερα από το στοιχείο στη μέση

Αυτό χωρίζει τον πίνακα σε δύο υπο-πίνακες που περιλαμβάνουν περίπου τα μισά στοιχεία

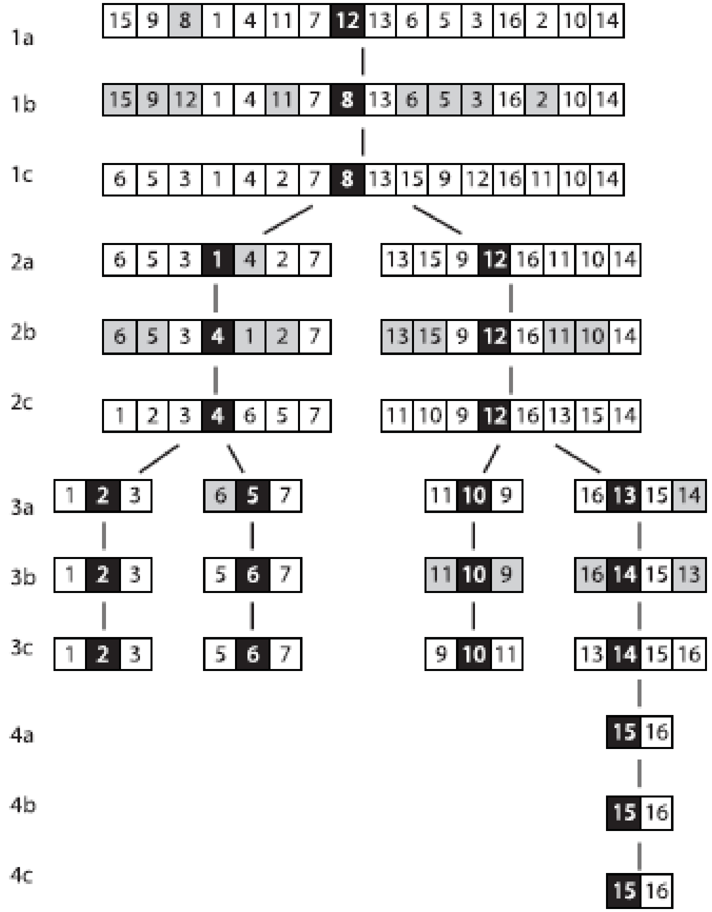
Median Sort (2/9)

```
sort (A)
1.    medianSort (A, 0, n – 1)
end

medianSort (A, left, right)
1.    if (left < right) then
2.        find median value A[me] in A[left, right]
3.        mid = ⌊(right+left)/2⌋
4.        swap A[mid] and A[me]
5.        for left=0 to mid – 1 do
6.            if (A[i] > A[mid]) then
7.                find A[k] ≤ A[mid] where k > mid
8.                swap A[i] and A[k]
9.        medianSort (A, left, mid – 1)
10.       medianSort (A, mid + 1, right)
end
```



Median Sort (3/9)

Παράδειγμα



Median Sort (4/9)

Η επίδοση του αλγορίθμου εξαρτάται από την αποδοτική επιλογή του median σε ένα μη ταξινομημένο πίνακα

Ας υποθέσουμε ότι έχουμε στη διάθεσή μας μια συνάρτηση partition(left, right, pivotIndex)

Η συνάρτηση επιλέγει το στοιχείο A[pivotIndex] να είναι το στοιχείο που χωρίζει τον πίνακα A σε δύο τμήματα: το πρώτο περιλαμβάνει στοιχεία μικρότερα ή ίσα του pivot και το δεύτερο που περιέχει στοιχεία που είναι μεγαλύτερα ή ίσα του pivot

Ισχύει πως left <= pivotIndex <= right

Η υλοποίηση μπορεί να αναζητηθεί στο Algorithms in a Nutshell

Median Sort (5/9)

Η συνάρτηση διαχωρισμού δεν ταξινομεί τα στοιχεία, απλά επιστρέφει το index του pivot

Το pivot υιοθετείται για την εύρεση του k^{th} στοιχείου αναδρομικά στο A[left, right] για οποιοδήποτε 1<=k<=right-left+1

Αν k=pivotIndex+1

- Το pivot είναι το k στοιχείο

Αν k<pivotIndex+1

- Το k στοιχείο είναι το k στοιχείο του A[left,pivotIndex]

Αν k>pivotIndex+1

- Το k στοιχείο είναι το k-pivotIndex στοιχείο του A[pivotIndex+1, right]

Median Sort (6/9)

Για την επιλογή του k υιοθετούνται διάφορες στρατηγικές:

- Επιλογή της 1ης ή της τελευταίας θέσης
- Επιλογή μιας τυχαίας θέσης στον A[left, right]

Αν η επιλογή του pivot δεν είναι αποδοτική, η επιλογή του k θα έχει επίδοση $O(n^2)$

Η επίδοση της καλύτερης και της μέσης περίπτωσης είναι O(n)

Median Sort (7/9)

Ανάλυση

- Στη μέση περίπτωση, ο αλγόριθμος έχει πολυπλοκότητα $O(n \log n)$
- Στη χειρότερη περίπτωση, ο αλγόριθμος έχει πολυπλοκότητα $O(n^2)$
- Το τμήμα διαχωρισμού είναι αυτό που επιβαρύνει περισσότερο

Median Sort (8/9)

Αλγόριθμος Blum-Floyd-Pratt-Rivest-Rarjan (BFPRT)

- Επιλογή του pivot
- Ομαδοποιεί τα στοιχεία σε n/4 ομάδες των 4 στοιχείων (αγνοεί μέχρι 3 στοιχεία που δεν ταιριάζουν με τις ομάδες των τεσσάρων)
- Εντοπίζει το median σε κάθε μια από τις ομάδες – απαιτούνται πέντε συγκρίσεις
- Πολυπλοκότητα (n/4)5=1.25n ~ O(n)
- Το median είναι το τρίτο στοιχείο σε κάθε ομάδα
- Όλα τα median στοιχεία αποτελούν ένα νέο σύνολο M
- Εύρεση του median στο M
- Αναδρομική κλήση στο M

Median Sort (9/9)

Παραδείγματα εκτέλεσης

Χειρότερη Περίπτωση

n	Randomized pivot selection	Leftmost pivot selection	Blum-Floyd-Pratt-Rivest-Tarjan pivot selection
256	0.000088	0.000444	0.00017
512	0.000213	0.0024	0.000436
1,024	0.000543	0.0105	0.0011
2,048	0.0012	0.0414	0.0029
4,096	0.0032	0.19	0.0072
8,192	0.0065	0.716	0.0156
16,384	0.0069	1.882	0.0354
32,768	0.0187	9.0479	0.0388
65,536	0.0743	47.3768	0.1065
131,072	0.0981	236.629	0.361

Καλύτερη Περίπτωση

n	Randomized pivot selection	Leftmost pivot selection	Blum-Floyd-Pratt-Rivest-Tarjan pivot selection
256	0.00009	0.000116	0.000245
512	0.000197	0.000299	0.000557
1,024	0.000445	0.0012	0.0019
2,048	0.0013	0.0035	0.0041
4,096	0.0031	0.0103	0.0128
8,192	0.0082	0.0294	0.0256
16,384	0.018	0.0744	0.0547
32,768	0.0439	0.2213	0.4084
65,536	0.071	0.459	0.5186
131,072	0.149	1.8131	3.9691

Counting Sort

Counting Sort (1/6)

Ο αλγόριθμος υποθέτει ότι το κάθε ένα στοιχείο από τα n είναι ένας ακέραιος στο διάστημα [0,k] για κάποιο k

Όταν k=O(n), ο αλγόριθμος έχει πολυπλοκότητα Θ(n)

Για κάθε είσοδο x, καθορίζει το πλήθος των στοιχείων που είναι μικρότερα του x

Χρησιμοποιεί αυτή την πληροφορία για να τοποθετήσει το x απ’ ευθείας στη θέση του

Για παράδειγμα, αν 15 στοιχεία είναι μικρότερα από το x, τότε το x πρέπει να μπει στη $16^{η}$ θέση

Για ένα πίνακα A[1,n], ο αλγόριθμος απαιτεί ένα πίνακα B[1,n] για το τελικό αποτέλεσμα και ένα πίνακα C[0,k] για προσωρινή αποθήκευση

Counting Sort (2/6)

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Counting Sort (3/6)

Ο πίνακας C περιέχει το πλήθος των στοιχείων που είναι ίσα με το δείκτη της κάθε θέσης

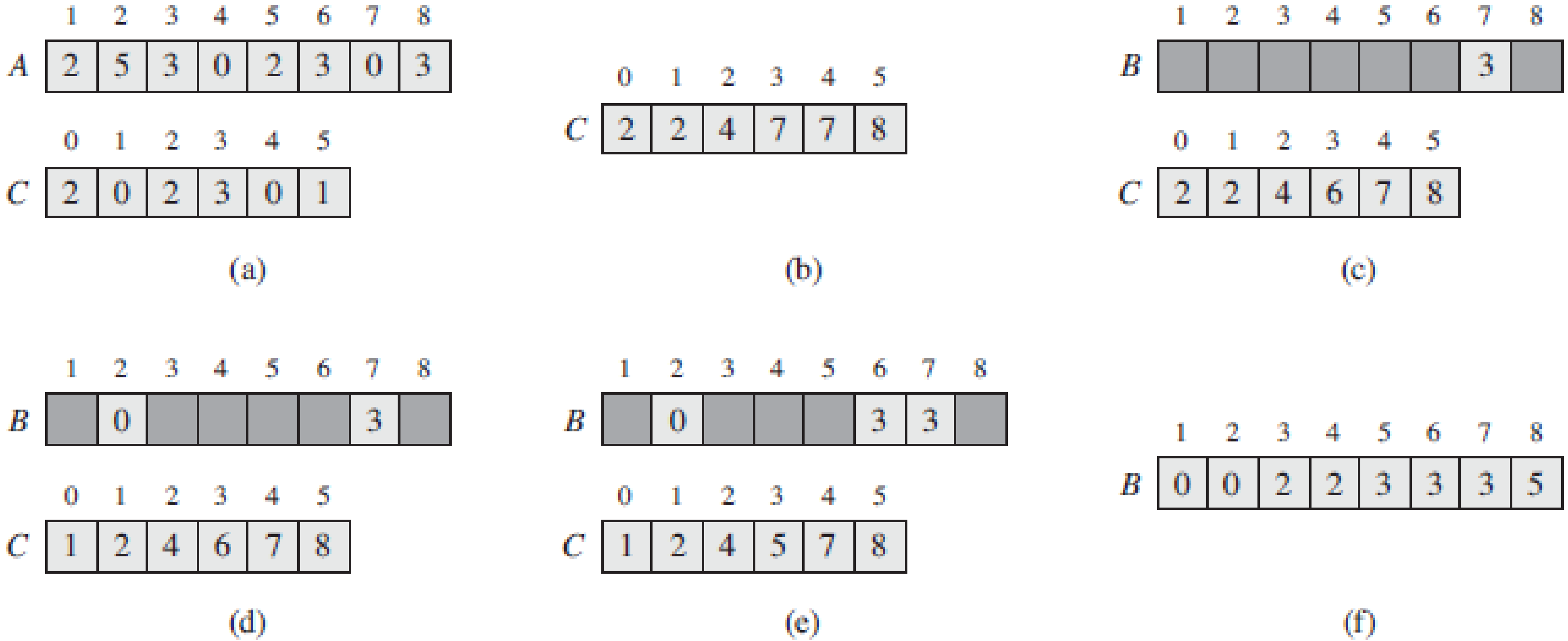
Παράδειγμα: το C[i] έχει το πλήθος των στοιχείων που είναι ίσα με i

Στις γραμμές 7-8, ο αλγόριθμος μετράει πόσα στοιχεία είναι μικρότερα ή ίσα από το i

Οι τελευταίες γραμμές του αλγορίθμου βάζουν το κάθε στοιχείο στη σωστή θέση

Counting Sort (4/6)

Παράδειγμα



Counting Sort (5/6)

Ανάλυση

- Οι γραμμές 2-3 έχουν πολυπλοκότητα Θ(k)
- Οι γραμμές 4-5 έχουν πολυπλοκότητα Θ(n)
- Οι γραμμές 7-8 έχουν πολυπλοκότητα Θ(k)
- Οι γραμμές 10-12 έχουν πολυπλοκότητα Θ(n)
- Συνολική πολυπλοκότητα: Θ(k+n)
- Αν k=O(n), τότε ο αλγόριθμος έχει πολυπλοκότητα Θ(n)
- Το μικρότερο όριο είναι το Ω(n logn)
- Ο αλγόριθμος δεν κάνει συγκρίσεις!
- Πρόκειται για ένα σταθερό αλγόριθμο αφού οι είσοδοι που είναι ίδιες εμφανίζονται με την ίδια σειρά στο τελικό αποτέλεσμα

```
COUNTING-SORT(A, B, k)
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i
7  for i = 1 to k
8      C[i] = C[i] + C[i - 1]
9  // C[i] now contains the number of elements less than or equal to i
10 for j = A.length downto 1
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] - 1
```

Counting Sort (6/6)

Demo

https://visualgo.net/en/sorting?slide=1

Radix Sort

Radix Sort (1/7)

Ο αλγόριθμος θεωρεί ότι κάθε στοιχείο είναι ένας αριθμός d ψηφίων

Κάθε ψηφίο παίρνει k πιθανές τιμές

Αρχικά ο αλγόριθμος ξεκινά από το λιγότερο σημαντικό ψηφίο πρώτα προς το πιο σημαντικό

Χρησιμοποιεί ένα σύνολο κάδων (bins)

Για αριθμούς στο δεκαδικό σύστημα κάθε στήλη θα έχει 10 θέσεις

Απαιτούνται d περάσματα για την ταξινόμηση

Radix Sort (2/7)

Παράδειγμα εκτέλεσης

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

Radix Sort (3/7)

```
RADIX-SORT(A, d)
  for i = 1 to d
      use a stable sort to sort array A on digit i
```

Radix Sort (4/7)

Lemma 1

- Δοσμένων n αριθμών με d ψηφία όπου κάθε ψηφίο έχει τιμές μέχρι k, ο Radix sort ταξινομεί αυτούς τους αριθμούς σε Θ(d(n+k)) εφόσον ο επιλεγμένος σταθερός αλγόριθμος ταξινόμησης ολοκληρώνει σε Θ(n+k)
  - Ο counting sort είναι μια καλή επιλογή όταν το k δεν είναι πολύ μεγάλο

Lemma 2

- Δοσμένων n b-bit αριθμών και οποιουδήποτε θετικού ακεραίου r<=b, ο radix sort ταξινομεί τους αριθμούς σε $\Theta((b/r)(n+2^r))$ εφόσον ο επιλεγμένος σταθερός αλγόριθμος ταξινόμησης ολοκληρώνει σε Θ(n+k)

Radix Sort (5/7)

Proof of Lemma 2

Για r<=b έχουμε ότι d=ceiling(b/r) ψηφία των r bits.

Κάθε ψηφίο είναι ένας ακέραιος στο $[0,2^r-1]$ και μπορούμε να υιοθετήσουμε τον counting sort με $k=2^r-1$

Παράδειγμα

- 32-bit word, 4 8-bit digits, b=32, r=8, $k=2^r-1=255$, d=b/r=4

Κάθε πέρασμα του counting sort απαιτεί χρόνο $\Theta(n+k)=\Theta(n+2^r)$

και γίνονται d περάσματα. Οπότε: $\Theta(d(n+2^r))=\Theta((b/r)(n+2^r))$

Radix Sort (6/7)

Γενικά, ψάχνουμε να βρούμε ένα r<=b που να ελαχιστοποιεί την έκφραση $(b/r)(n+2^r)$

Αν b<floor(logn), τότε για κάθε r<=b έχουμε $(n+2^r) = \Theta(n)$. Παίρνοντας r=b έχουμε χρόνο $(b/b)(n+2^r) = \Theta(n)$ που είναι ο βέλτιστος ασυμπτωτικά

Αν b>=floor(logn), τότε παίρνοντας r=floor(logn) έχουμε χρόνο Θ(bn/logn) – παίρνοντας r>floor(logn) έχουμε χρόνο Ω(bn/logn) - παίρνοντας r<floor(logn) έχουμε χρόνο Θ(n)

Radix Sort (7/7)

Demo

https://visualgo.net/en/sorting?slide=1

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Heap Sort

Heap Sort (1/15)

Υιοθετεί τη δομή του σωρού (heap) για την ταξινόμηση

Ο σωρός όχι μόνο είναι χρήσιμος για την υλοποίηση μιας δομής δεδομένων αλλά προσφέρει μια ουρά με προτεραιότητα

Ο σωρός έχει συνδεθεί με την τεχνική του garbage collection της java ή της Lisp

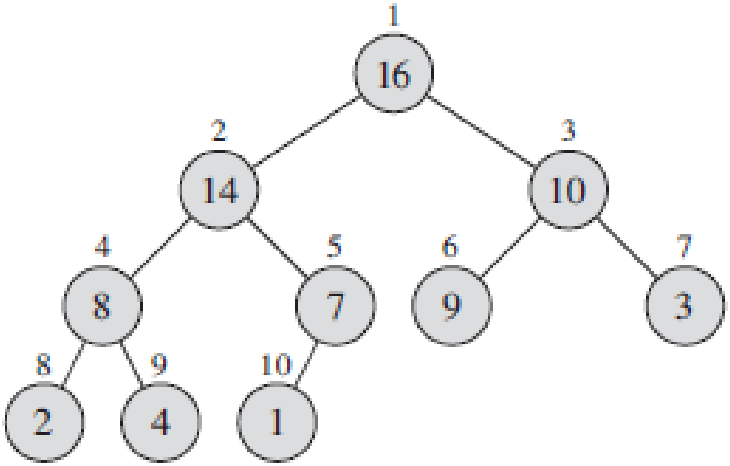
Heap Sort (2/15)

Ο σωρός είναι ένας πίνακας που μπορούμε να τον δούμε σαν ένα δυαδικό δένδρο

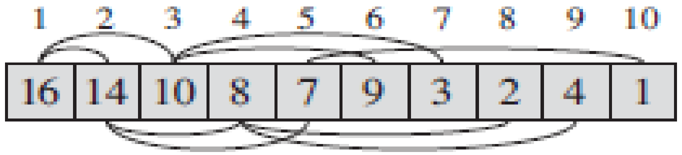
Κάθε κόμβος του δένδρου αντιστοιχεί σε ένα στοιχείο του πίνακα

Το δένδρο έχει γεμίσει σε όλα τα επίπεδα εκτός ίσως από το τελευταίο

Ο πίνακας A χαρακτηρίζεται από 2 στοιχεία: το μέγεθος A.length και το μέγεθος του σωρού A.heap_size που αναπαριστά πόσα στοιχεία του σωρού έχουν αποθηκευτεί στον πίνακα



(a)



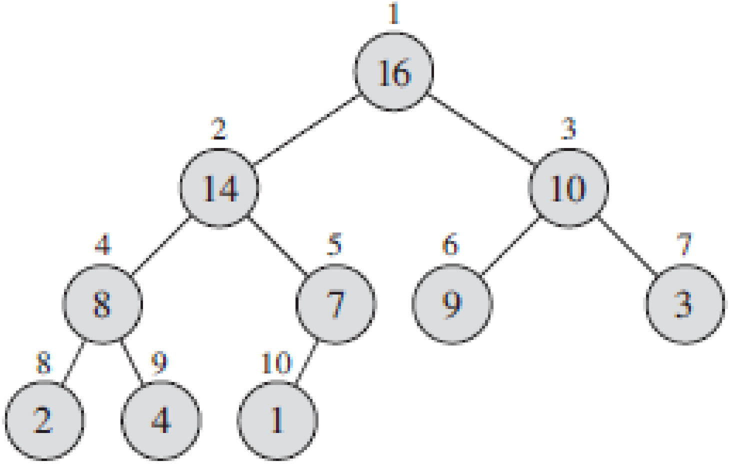
(b)

Heap Sort (3/15)

Μόνο τα A.heap_size είναι στοιχεία του σωρού

Η ρίζα του δένδρου είναι το A[1]

Σε ένα index i, μπορούμε εύκολα να υπολογίσουμε τα indices του γονέα, του αριστερού και του δεξιού παιδιού

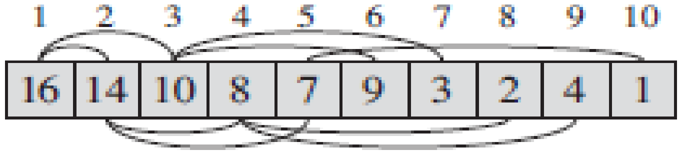


(a)

PARENT(i)
1 return $\lfloor i/2 \rfloor$

LEFT(i)
1 return $2i$

RIGHT(i)
1 return $2i + 1$



(b)

Heap Sort (4/15)

Ο υπολογισμός των indices είναι εύκολος:

- Το 2i υπολογίζεται με μια απλή ολίσθηση αριστερά
- Το 2i+1 υπολογίζεται με μια ολίσθηση αριστερά και πρόσθεση του 1 στο λιγότερο σημαντικό ψηφίο
- Το floor(i/2) υπολογίζεται με μια απλή ολίσθηση δεξιά

Δύο είδη σωρών

- **Max-heap** $A[\textsc{Parent}(i)] \geq A[i]$
  - Το μεγαλύτερο στοιχείο βρίσκεται στη ρίζα
- **Min-heap** $A[\textsc{Parent}(i)] \leq A[i]$
  - Το μικρότερο στοιχείο βρίσκεται στη ρίζα

Heap Sort (5/15)

Ο αλγόριθμος heap sort υιοθετεί την max-heap μέθοδο

Η μέθοδος min-heap υιοθετείται για τις ουρές με προτεραιότητα

Το ύψος ενός κόμβου είναι το πλήθος των ακμών από τον κόμβο μέχρι τα φύλλα

Το ύψος του σωρού είναι το ύψος της ρίζας

Αφού έχουμε n στοιχεία, το ύψος θα είναι Θ(logn)

Ο χρόνος υλοποίησης διαφόρων ενεργειών πάνω σε ένα σωρό εξαρτώνται από το ύψος του

Heap Sort (6/15)

Για τη διατήρηση της ιδιότητας max-heap υιοθετούμε τον ακόλουθο αλγόριθμο

Ο αλγόριθμος υποθέτει ότι τα δένδρα που ξεκινούν από το αριστερό και το δεξιό παιδί υιοθετούν πάλι την max-heap ιδιότητα

Το κάθε στοιχείο A[i] τοποθετείται στο σωστό σημείο του δένδρου

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Heap Sort (7/15)

Σε κάθε βήμα, το μεγαλύτερο στοιχείο ανάμεσα στα A[i], A[left(i)], A[right(i)] αποθηκεύεται στο largest (ο δείκτης του)

Αν το μεγαλύτερο είναι το A[i], ο αλγόριθμος τερματίζει

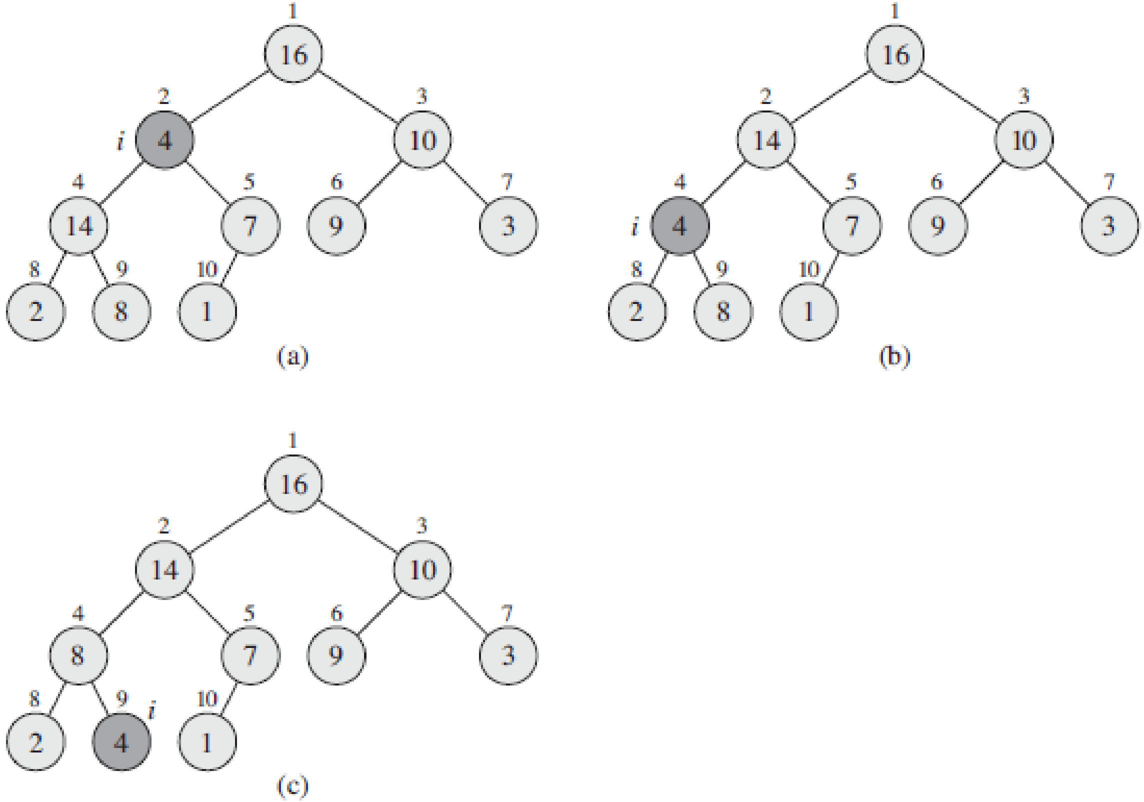
Διαφορετικά, ένα από τα παιδιά έχει το μεγαλύτερο στοιχείο, οπότε το A[i] αντιμετατίθεται με το A[largest]

Έπειτα καλούμε αναδρομικά τον αλγόριθμο

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Heap Sort (8/15)

Η εικόνα παρουσιάζει τη λειτουργία του αλγορίθμου



Heap Sort (9/15)

Ανάλυση

- Ο χρόνος εκτέλεσης για ένα υπο-δένδρο μεγέθους n με ρίζα στο i είναι Θ(1) επιπλέον του χρόνου εκτέλεσης του αλγορίθμου σε ένα από τα παιδιά του κόμβου i
- Τα υπο-δένδρα των παιδιών έχουν μέγεθος το πολύ 2n/3 – η χειρότερη περίπτωση είναι όταν το κάτω επίπεδο μισο-γεμάτο
- Ο χρόνος εκτέλεσης είναι **$T(n) \le T(2n/3) + \Theta(1)$**
- Με βάση το master θεώρημα – περίπτωση 2 – έχουμε ότι **$T(n) = O(\log n)$** – ή εναλλακτικά **$O(h)$** όπου h είναι το ύψος του σωρού

Heap Sort (10/15)

Μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο MAX-HEAPIFY για να μετατρέψουμε ένα πίνακα σε σωρό

Σε κάθε iteration (γραμμές 2-3) ο κάθε κόμβος i+1, i+2, …, n είναι η ρίζα ενός max-heap

Κάθε κλήση στον αλγόριθμο MAX-HEAPIFY κοστίζει O(logn) ενώ η κλήση στον αλγόριθμο που χτίζει το σωρό κοστίζει O(n)

Άρα συνολικό κόστος εκτέλεσης O(n logn)

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Heap Sort (11/15)

Υπολογισμός άνω ορίου χρόνου εκτέλεσης

- Ο χρόνος για τον MAX-HEAPIFY μεταβάλλεται ανάλογα με το ύψος του κόμβου
- Τα ύψη των περισσότερων κόμβων είναι μικρά
- Ένας σωρός με n στοιχεία έχει ύψος floor(logn) και το πολύ ceiling($n/2^{h+1}$) κόμβους ασχέτως ύψους
- Αφού ο MAX-HEAPIFY έχει πολυπλοκότητα O(h) μπορούμε να εκφράσουμε το συνολικό κόστος ως εξής:

Heap Sort (12/15)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$x = 1/2$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Συνεπώς, η δημιουργία του σωρού μπορεί να ολοκληρωθεί σε γραμμικό χρόνο

Heap Sort (13/15)

Ο τελικός αλγόριθμος heap sort έχει ως εξής:

Αφού το μέγιστο στοιχείο έχει αποθηκευτεί στο A[1] μπορούμε να το βάλουμε στη σωστή θέση κάθε φορά ανταλλάσσοντάς το με το A[n]

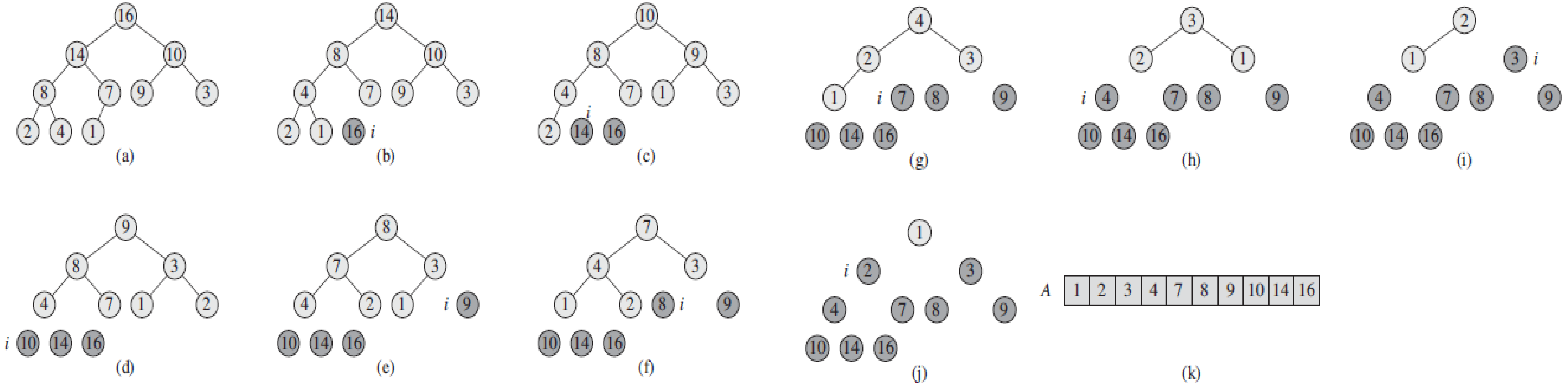
Στη συνέχεια μειώνουμε το μέγεθος του σωρού (δεν λαμβάνουμε υπόψιν το τελευταίο στοιχείο)

Επαναφέρουμε την ιδιότητα max heap καλώντας τον MAX-HEAPIFY

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Heap Sort (14/15)

Παράδειγμα



Heap Sort (15/15)

Demo

http://www.sorting-algorithms.com/heap-sort

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Bucket Sort

Bucket Sort (1/9)

Ο αλγόριθμος υποθέτει ότι οι είσοδοι ακολουθούν την ομοιόμορφη κατανομή

Έχει μέση πολυπλοκότητα ίση με O(n)

Υποθέτει ότι οι είσοδοι έχουν παραχθεί από μια ομοιόμορφη κατανομή στο [0,1)

Διαιρεί το [0,1) σε n ισομεγέθη τμήματα και κατανέμει τις εισόδους στα buckets

Υπόθεση: κάθε στοιχείο A[i] ικανοποιεί τη συνθήκη 0<=A[i]<1

Απαιτεί τη χρήση ενός πίνακα B[0,n-1] συνδεδεμένων λιστών (buckets)

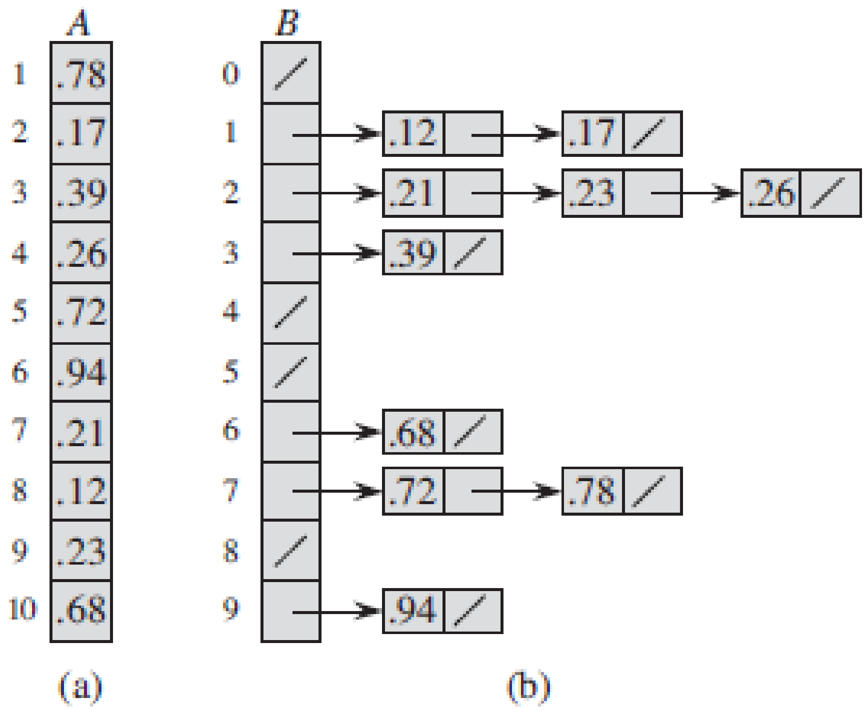
Bucket Sort (2/9)

```
BUCKET-SORT(A)
let B[0 . . n − 1] be a new array
n = A.length
for i = 0 to n − 1
        make B[i] an empty list
for i = 1 to n
        insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
        sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Bucket Sort (3/9)

Παράδειγμα

- Το κάθε bucket αποθηκεύει τιμές στο [i/10, (i+1)/10) με A[1,10]



Bucket Sort (4/9)

Ανάλυση

- Όλα τα τμήματα απαιτούν O(n) εκτός από την κλήση στο insertion sort
- Έστω n_i η τυχαία μεταβλητή που απεικονίζει το πλήθος των στοιχείων που μπαίνουν στο bucket B[i]
- Μια και ο insertion sort απαιτεί τετραγωνικό χρόνο, ο χρόνος εκτέλεσης του bucket sort είναι

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Bucket Sort (5/9)

Η ανάλυση της μέσης περίπτωσης εξάγεται από τον υπολογισμό της αναμενόμενης τιμής

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right)
\end{aligned}
$$

Έχουμε $\mathrm{E}\left[n_i^2\right] = 2 - 1/n$

Bucket Sort (6/9)

Το κάθε bucket έχει το ίδιο $\mathrm{E}\left[n_i^2\right]$

Ορίζουμε την indicator function

$$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$$

for $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Bucket Sort (7/9)

Συνεπώς

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \mathrm{E}\left[X_{ij}X_{ik}\right]
\end{aligned}
$$

Bucket Sort (8/9)

Η X_{ij} είναι 1 με πιθανότητα 1/n και 0 διαφορετικά

Έτσι

$$\begin{aligned} \mathrm{E}\left[X_{ij}^2\right] &= 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} \end{aligned}$$

Όταν k<>j, τα X_{ij} και X_{ik} είναι ανεξάρτητα και συνεπώς

$$\begin{aligned} \mathrm{E}\left[X_{ij} X_{ik}\right] &= \mathrm{E}\left[X_{ij}\right] \mathrm{E}\left[X_{ik}\right] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} \end{aligned}$$

Bucket Sort (9/9)

Τέλος, έχουμε

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

Και η τελική πολυπλοκότητα είναι: $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$

Shell Sort

Shell Sort (1/10)

Πρόκειται για μια γενίκευση του insertion sort

Η ιδέα είναι να τοποθετηθούν τα στοιχεία σε τέτοια σειρά ώστε ξεκινώντας από οποιοδήποτε στοιχείο να προκύπτει μια ταξινομημένη λίστα

Αποφεύγει την απομακρυσμένη μετακίνηση των στοιχείων

Διασπά την αρχική λίστα των στοιχείων σε ένα αριθμό μικρότερων λιστών

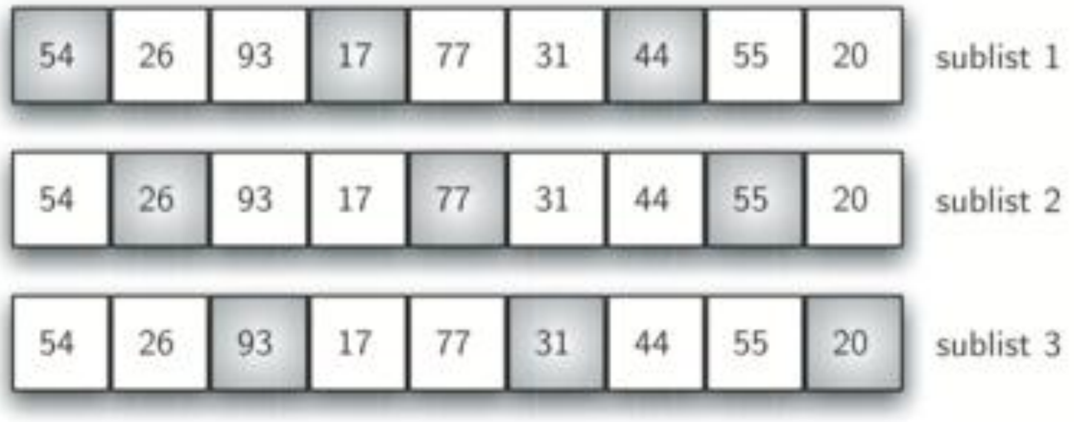
Κάθε μια υπο-λίστα ταξινομείται με τον insertion sort

Shell Sort (2/10)

Αντί να επιλέγει συνεχόμενα στοιχεία για να περιληφθούν στις υπο-λίστες, υιοθετεί μια μεταβλητή i που ονομάζεται κενό (gap) για να δημιουργήσει υπο-λίστες με στοιχεία που είναι i στοιχεία μακριά

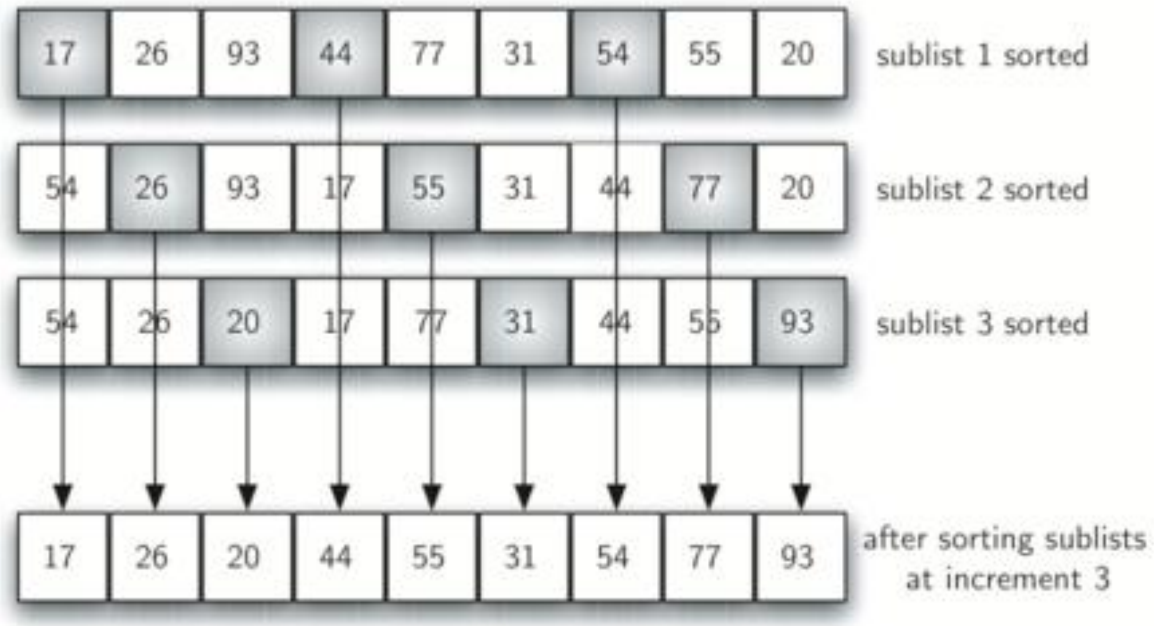
Παράδειγμα:

- i=3



Shell Sort (3/10)

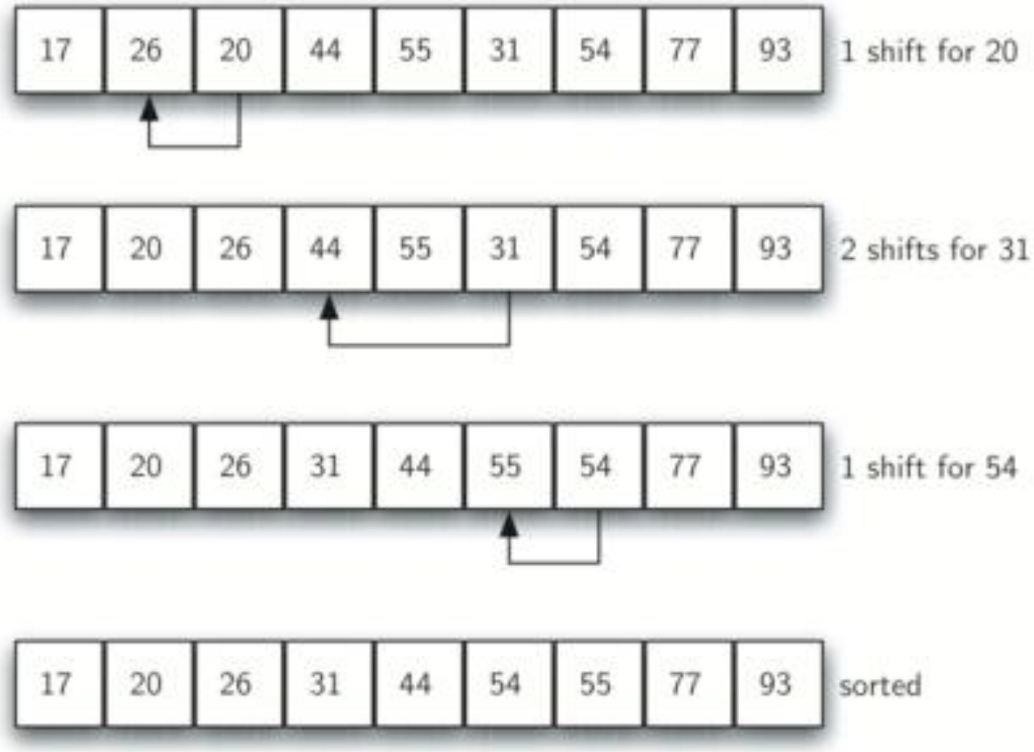
Μετά από την ταξινόμηση με τον insertion sort η λίστες γίνονται



Μετά από κάθε ταξινόμηση το κάθε στοιχείο μετακινείται πιο κοντά στη θέση του

Shell Sort (4/10)

Μια τελική εκτέλεση του insertion sort θα ταξινομήσει τον πίνακα, όμως με μικρότερο πλήθος αντιμεταθέσεων



Shell Sort (5/10)

Σημαντική είναι η επιλογή του gap

Παραδείγματα:

- Μπορούμε να ξεκινήσουμε με n/2, έπειτα με n/4, κ.ο.κ, και στο τέλος με 1

```
public static void
shellSort( Comparable[ ] theArray, int n ) {
// shellSort: sort first n items in array theArray

  for( int gap = n / 2; gap > 0; gap = gap / 2 )
    for( int i = gap; i < n; i++ ) {
      Comparable tmp = theArray[ i ];
      int j = i;

      for( ; j >= gap && tmp.compareTo(theArray[ j - gap ]) < 0 ; j -= gap )
          theArray[ j ] = theArray[ j - gap ];
      theArray[ j ] = tmp;
    }
}
```

Shell Sort (6/10)

Τα καλύτερα αποτελέσματα τα έχουμε όταν όλες οι τιμές του gap είναι πρώτοι – η ακολουθία των gaps δεν έχει κάποιο κοινό διαιρέτη

Όμως, το να βρούμε μια ακολουθία πρώτων αριθμών ίσως να είναι πρακτικά δύσκολο – Προσεγγιστική λύση

Shell Sort (7/10)

Κάποιες λύσεις για την επιλογή του gap

- Shell's suggestion
  - Το πρώτο είναι το n/2 και κάθε επόμενο είναι το μισό του προηγούμενου (χρήση της floor)
- Περιττοί αριθμοί ως gaps
  - Όμοια με την προηγούμενη με καλύτερη επίδοση (αν προκύψει άρτιος αριθμός προσθέτουμε 1)
- Μεθοδολογία 2.2
  - Ίδια με την προηγούμενη – προσθέτουμε 1 στους άρτιους αριθμούς αλλά υιοθετούμε ως διαιρέτη το 2.2
  - Καλύτερη επίδοση ανάμεσα σε όλες

Shell Sort (8/10)

Οι λύσεις που έχουν προταθεί για την επιλογή του gap

General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
$\lfloor N/2^k \rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta(N^2)$ [when $N = 2^p$]	Shell, 1959
$2\lfloor N/2^{k+1} \rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta(N^{3/2})$	Frank & Lazarus, 1960
$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta(N^{3/2})$	Hibbard, 1963
$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta(N^{3/2})$	Papernov & Stasevich, 1965
successive numbers of the form $2^p 3^q$	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta(N \log^2 N)$	Pratt, 1971
$(3^k - 1)/2$, not greater than $\lceil N/3 \rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta(N^{3/2})$	Pratt, 1971
$\prod\limits_{\substack{0 \leq q < r \\ q \neq (r^2+r)/2-k}} a_q$, where $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$, $a_q = \min\{n \in \mathbb{N} \colon n \geq (5/2)^{q+1}, \forall p \colon 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\}$	$1, 3, 7, 21, 48, 112, \ldots$	$O(N^{1+\sqrt{8 \ln(5/2)/\ln N}})$	Incerpi & Sedgewick, 1985, Knuth
$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O(N^{4/3})$	Sedgewick, 1986
$9(4^{k-1} - 2^{\frac{k}{2}}) + 1, 4^{k+1} - 6 \cdot 2^{\frac{k+1}{2}} + 1$	$1, 5, 19, 41, 109, \ldots$	$O(N^{4/3})$	Sedgewick, 1986
$h_k = \max\{\lfloor 5h_{k-1}/11 \rfloor, 1\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11} \left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	?	Gonnet & Baeza-Yates, 1991
$\left\lceil \frac{9^k - 4^k}{5 \cdot 4^{k-1}} \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	?	Tokuda, 1992
unknown (empirically derived)	$1, 4, 10, 23, 57, 132, 301, 701, 1750$	?	Ciura, 2001

Shell Sort (9/10)

Ανάλυση

- Με μια πρώτη ματιά, ο αλγόριθμος φαίνεται πως δεν τα καταφέρνει καλύτερα από τον insertion sort – ***αφού υιοθετεί insertion sort στο τελευταίο βήμα (με gap 1)***
- Όμως, το πλήθος των αντιμεταθέσεων είναι κατά πολύ μικρότερο από τον insertion sort – η λίστα έχει ήδη προ-ταξινομηθεί στα προηγούμενα βήματα
- Η πολυπλοκότητα είναι ανάμεσα στο O(n) και στο $O(n^2)$ εξαρτώμενη από το gap
- Αν υιοθετήσουμε ως gap το 2^k-1, η επίδοση είναι $O(n^{1.5})$
- Για άλλες ακολουθίες οι επιδόσεις είναι $O(n^{4/3})$, $O(n \log^2 n)$

Shell Sort (10/10)

Demo

http://www.sorting-algorithms.com/shell-sort

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Comparison (1/2)

Demo

http://www.sorting-algorithms.com/random-initial-order

http://www.sorting-algorithms.com/nearly-sorted-initial-order

http://www.sorting-algorithms.com/reversed-initial-order

http://www.sorting-algorithms.com/few-unique-keys

Comparison (2/2)

https://en.wikipedia.org/wiki/Sorting_algorithm

Ασκήσεις στους Αλγορίθμους Ταξινόμησης

Άσκηση 1

Υποθέτουμε ότι το n είναι πολλαπλάσιο του 2 και ότι ένας αλγόριθμος ταξινόμησης χαρακτηρίζεται από την αναδρομική σχέση

$$T(n) = \begin{cases} 2 & \text{if } n = 2\,, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

Να αποδείξετε ότι

$$T(n) = n \lg n$$

Λύση

$$T(n) = \begin{cases} 2 & \text{if } n = 2\,, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

Η καλύτερη περίπτωση συναντάται όταν n=2 οπότε έχουμε:

nlogn = 2log2=2

Για την αναδρομική εξίσωση έχουμε ότι T(n/2)=(n/2)log(n/2)

Οπότε

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2)\lg(n/2) + n \\ &= n(\lg n - 1) + n \\ &= n\lg n - n + n \\ &= n\lg n \end{aligned}$$

Άσκηση 2

Ο αλγόριθμος insertion sort μπορεί να υλοποιηθεί αναδρομικά ως εξής: Για να ταξινομηθεί ο πίνακας A[1..n], ταξινομούμε αναδρομικά τον υπο-πίνακα A[1..n-1] και στη συνέχεια εισάγουμε το στοιχείο A[n] στη θέση του στον ταξινομημένο υπο-πίνακα A[1..n-1].

Να γράψετε την αναδρομική σχέση για αυτό τον αλγόριθμο.

Λύση

Στη χειρότερη περίπτωση θα χρειαστούμε χρόνο Θ(n) για να τοποθετήσουμε το A[n] στη θέση του. Συνεπώς, η αναδρομική σχέση θα είναι:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Μέσω της οποίας μπορούμε εύκολα να αποδείξουμε πως $T(n) = \Theta(n^2)$

T(n-1)+n-1 [……… n-1 ……….] ← A[n]

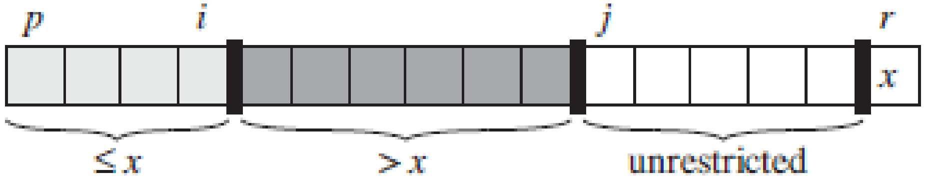
T(n-2)+n-2+n-1 [……… n-2 ……….] ← A[n-1] ← A[n]

Άσκηση 3

Να αποδείξετε ότι η πολυπλοκότητα του quicksort είναι $\Theta(n^2)$ όταν ο πίνακας A περιέχει διακριτά στοιχεία (δεν υπάρχουν δύο ίσα στοιχεία μεταξύ τους) και είναι ταξινομημένα σε φθίνουσα σειρά.

Λύση

Αλγόριθμος quick sort




```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Λύση

Στην περίπτωση που τα στοιχεία είναι σε φθίνουσα σειρά, ο αλγόριθμος PARTITION θα εκτελεί λειτουργία χειρότερης περίπτωσης.

Θα μειώνει το μέγεθος του υπο-πίνακα μόνο κατά 1 σε κάθε βήμα

Για ένα υπο-πίνακα A[p..r], θα παράξει ένα κενό τμήμα στο A[p..r-1] με μόνο ένα λιγότερο στοιχείο σε σχέση με τον αρχικό υποπίνακα

Θα τοποθετεί το pivot στην θέση A[r] αρχικά και μετά στην θέση A[p] και θα παράξει το τμήμα A[p+1..r]

Συνεπώς, η αναδρομική σχέση θα είναι $T(n)=T(n-1) + \Theta(n)$ που έχει πολυπλοκότητα **????**

Άσκηση 4

Να αποδείξετε ότι η πολυπλοκότητα του quick sort στην καλύτερη περίπτωση είναι Ω(nlogn)

Λύση

Έχουμε την αναδρομική σχέση:

$$T(n) = \min_{1 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

Υποθέτουμε ότι

$$T(n) \ge cn \lg n$$

Για κάποια σταθερά c

Με αντικατάσταση στην αναδρομική σχέση παίρνουμε

$$\begin{aligned} T(n) &\ge \min_{1 \le q \le n-1} (cq \lg q + c(n-q-1)\lg(n-q-1)) + \Theta(n) \\ &= c \cdot \min_{1 \le q \le n-1} (q \lg q + (n-q-1)\lg(n-q-1)) + \Theta(n) \end{aligned}$$

Λύση

Θα πρέπει να δείξουμε πως η σχέση

$$q \lg q + (n - q - 1) \lg(n - q - 1)$$

Έχει ρίζα – ελάχιστο στο

$$1 \leq q \leq n-1$$

Όταν

$$q = n - q - 1$$

ή

$$q = (n-1)/2$$

Λύση

Η πρώτη παράγωγος της συνάρτησης είναι 0 όταν

$q = (n-1)/2$

Και η δεύτερη παράγωγος είναι θετική

Αν λοιπόν επιλέξουμε $q = (n-1)/2$

τότε $\min\limits_{1 \le q \le n-1} (q \lg q + (n-q-1)\lg(n-q-1)$

$$
\begin{aligned}
&\ge \frac{n-1}{2}\lg\frac{n-1}{2} + \left(n - \frac{n-1}{2} - 1\right)\lg\left(n - \frac{n-1}{2} - 1\right) \\
&= (n-1)\lg\frac{n-1}{2}
\end{aligned}
$$

Λύση

Αν τώρα προσπαθήσουμε να φράξουμε το T(n) από κάτω θα έχουμε για n>=2

$$\begin{aligned}
T(n) &\geq c(n-1)\lg\frac{n-1}{2} + \Theta(n) \\
&= c(n-1)\lg(n-1) - c(n-1) + \Theta(n) \\
&= cn\lg(n-1) - c\lg(n-1) - c(n-1) + \Theta(n) \\
&\geq cn\lg(n/2) - c\lg(n-1) - c(n-1) + \Theta(n) \\
&= cn\lg n - cn - c\lg(n-1) - cn + c + \Theta(n) \\
&= cn\lg n - (2cn + c\lg(n-1) - c) + \Theta(n) \\
&\geq cn\lg n
\end{aligned}$$

Λύση

Πρέπει να επιλεγεί ένα c αρκετά μικρό ώστε ο όρος Θ(n) να ‘υπερισχύει’ του όρου

$$2cn + c\lg(n-1) - c$$

Οπότε η πολυπλοκότητα της καλύτερης περίπτωσης είναι Ω(nlogn)

Λύση

Όσον αφορά στην παραγώγιση έχουμε:

$$f(q) = q \lg q + (n-q-1)\lg(n-q-1)$$

$$f(q) = \frac{q \ln q + (n-q-1)\ln(n-q-1)}{\ln 2}$$

$$\begin{aligned} f'(q) &= \frac{d}{dq}\left(\frac{q \ln q + (n-q-1)\ln(n-q-1)}{\ln 2}\right) \\ &= \frac{\ln q + 1 - \ln(n-q-1) - 1}{\ln 2} \\ &= \frac{\ln q - \ln(n-q-1)}{\ln 2} \end{aligned}$$

Λύση

Η παράγωγος γίνεται 0 όταν

$q = n - q - 1$ ή $q = (n-1)/2$

Και

$$\begin{aligned}
f''(q) &= \frac{d}{dq}\left(\frac{\ln q - \ln(n-q-1)}{\ln 2}\right) \\
&= \frac{1}{\ln 2}\left(\frac{1}{q} + \frac{1}{n-q-1}\right) \\
f''\left(\frac{n-1}{2}\right) &= \frac{1}{\ln 2}\left(\frac{2}{n-1} + \frac{2}{n-1}\right) \\
&= \frac{1}{\ln 2} \cdot \frac{4}{n-1} \\
&> 0
\end{aligned}$$

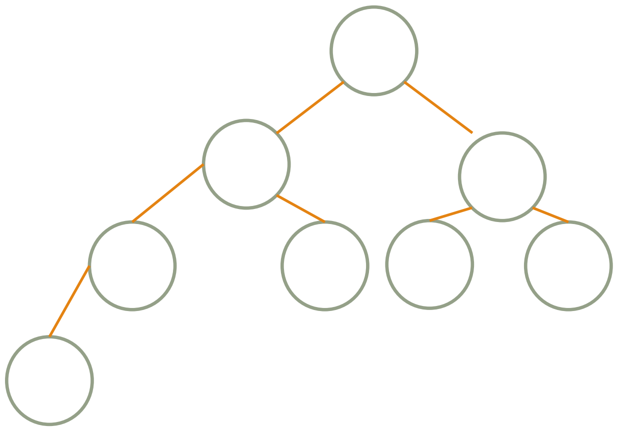
Άσκηση 5

Ποιο είναι το ελάχιστο και το μέγιστο πλήθος των στοιχείων σε ένα σωρό ύψους h?

Λύση

Αφού ο σωρός είναι σχεδόν πλήρες δυαδικό δένδρο, η συμπλήρωση όλων των στοιχείων θα έχει το πολύ $\mathbf{2^h - 1}$ στοιχεία.

Τα στοιχεία θα είναι τουλάχιστον $\mathbf{2^{h-1} - 1 + 1 = 2^{h-1}}$ (όταν το τελευταίο επίπεδο έχει ένα στοιχείο και όλα τα υπόλοιπα επίπεδα είναι γεμάτα)

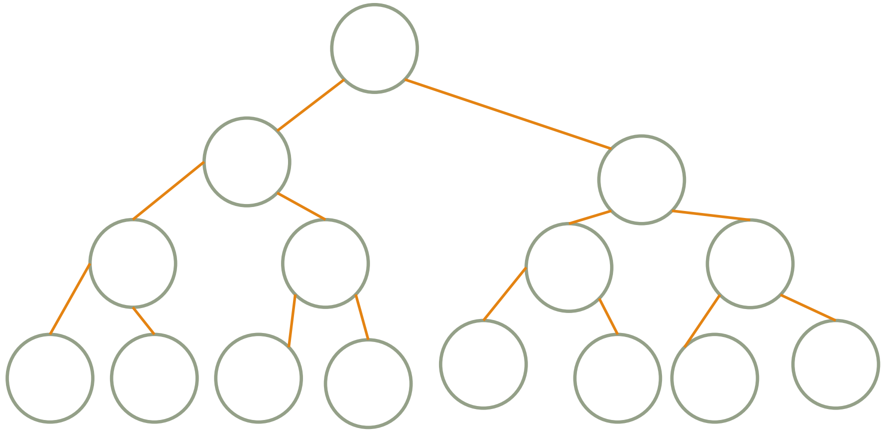


1ο (0)

2ο (1)

3ο (2)

4ο (3)



Άσκηση 5

Να αποδείξετε ότι ένα σωρός που έχει n στοιχεία έχει ύψος **floor(logn)**

Λύση

Με n στοιχεία, έχουμε από την προηγούμενη άσκηση ότι

$\mathbf{2^h <= n <= 2^{h+1}}$

Αν λογαριθμήσουμε κατά μέρη θα πάρουμε:

$h <= \log n < h+1$

Οπότε αφού το h είναι ακέραιος έχουμε πως

h = floor(logn)

Άσκηση 6

Να ταξινομήσετε τη λίστα E X A M P L E με τον αλγόριθμο selection sort

Λύση

$$
\begin{array}{l}
\mid E \quad X \quad \mathbf{A} \quad M \quad P \quad L \quad E \\
A \mid X \quad \mathbf{E} \quad M \quad P \quad L \quad E \\
A \quad E \mid X \quad M \quad P \quad L \quad \mathbf{E} \\
A \quad E \quad E \mid M \quad P \quad \mathbf{L} \quad X \\
A \quad E \quad E \quad L \mid P \quad \mathbf{M} \quad X \\
A \quad E \quad E \quad L \quad M \mid \mathbf{P} \quad X \\
A \quad E \quad E \quad L \quad M \quad P \mid X
\end{array}
$$

Άσκηση 7

Να ταξινομήσετε τη λίστα E X A M P L E με τον αλγόριθμο bubble sort

Λύση

E, X, A, M, P, L, E

$$
\begin{array}{ccccccccccccc}
E & \overset{?}{\leftrightarrow} & X & \overset{?}{\leftrightarrow} & A & & M & & P & & L & & E \\
E & & A & & X & \overset{?}{\leftrightarrow} & M & & P & & L & & E \\
E & & A & & M & & X & \overset{?}{\leftrightarrow} & P & & L & & E \\
E & & A & & M & & P & & X & \overset{?}{\leftrightarrow} & L & & E \\
E & & A & & M & & P & & L & & X & \overset{?}{\leftrightarrow} & E \\
E & & A & & M & & P & & L & & E & & |X \\
E & \overset{?}{\leftrightarrow} & A & & M & & P & & L & & E & & \\
A & & E & \overset{?}{\leftrightarrow} & M & \overset{?}{\leftrightarrow} & P & \overset{?}{\leftrightarrow} & L & & E & & \\
A & & E & & M & & L & & P & \overset{?}{\leftrightarrow} & E & & \\
A & & E & & M & & L & & E & & |P & & \\
A & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & M & \overset{?}{\leftrightarrow} & L & & E & & & & \\
A & & E & & L & & M & \overset{?}{\leftrightarrow} & E & & & & \\
A & & E & & L & & E & & |M & & & & \\
A & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & L & \overset{?}{\leftrightarrow} & E & & & & & & \\
A & & E & & E & & |L & & & & & & \\
A & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & L & & & & & &
\end{array}
$$

Άσκηση 8

Ο selection sort είναι ένας σταθερός (stable) αλγόριθμος?

Λύση

Η απάντηση είναι όχι!

Σκεφτείτε την ακολουθία 2, 2, 1

Ο αλγόριθμος, συνήθως, αντιμεταθέτει μη γειτονικά στοιχεία

Άσκηση 9

Ο bubble sort είναι ένας σταθερός (stable) αλγόριθμος?

Λύση

Η απάντηση είναι ναι!

Ο αλγόριθμος πάντα συγκρίνει γειτονικά στοιχεία μέσω της συνθήκης A[j+1] < A[j]

Συνεπώς σε ισότητα δεν αντιμεταθέτει τα στοιχεία και παραμένουν στη σειρά τους

Άσκηση 10

Να ταξινομήσετε τη λίστα E X A M P L E με τον αλγόριθμο insertion sort

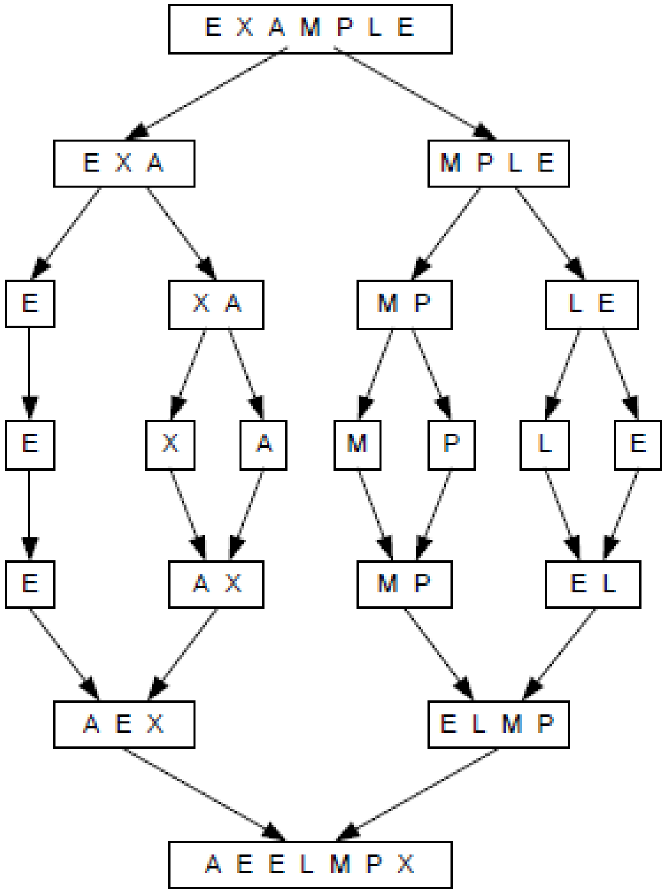
Λύση

$$
\begin{array}{lllllll}
E & X & A & M & P & L & E \\
E \mid & \mathbf{X} & & & & & \\
E & X \mid & \mathbf{A} & & & & \\
A & E & X \mid & \mathbf{M} & & & \\
A & E & M & X \mid & \mathbf{P} & & \\
A & E & M & P & X \mid & \mathbf{L} & \\
A & E & L & M & P & X \mid & \mathbf{E} \\
A & E & E & L & M & P & X
\end{array}
$$

Άσκηση 11

Να ταξινομήσετε τη λίστα E X A M P L E με τον αλγόριθμο merge sort

Λύση



Άσκηση 12

Να επιλύσετε την αναδρομική σχέση των συγκρίσεων στον merge sort στη χειρότερη περίπτωση

Λύση

Η σχέση έχει δοθεί στην προηγούμενη διάλεξη και είναι

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0$$

Η λύση της έχει ως εξής (για n άρτιο):

$$\begin{aligned}
C_w(2^k) &= 2C_w(2^{k-1}) + 2^k - 1 \\
&= 2[2C_w(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 = 2^2C_w(2^{k-2}) + 2 \cdot 2^k - 2 - 1 \\
&= 2^2[2C_w(2^{k-3}) + 2^{k-2} - 1] + 2 \cdot 2^k - 2 - 1 = 2^3C_w(2^{k-3}) + 3 \cdot 2^k - 2^2 - 2 - 1 \\
&= \ldots \\
&= 2^iC_w(2^{k-i}) + i2^k - 2^{i-1} - 2^{i-2} - \ldots - 1 \\
&= \ldots \\
&= 2^kC_w(2^{k-k}) + k2^k \underbrace{- 2^{k-1} - 2^{k-2} - \ldots - 1} = k2^k - (2^k - 1) = n\log n - n + 1
\end{aligned}$$

Άσκηση 13

Να ορίσετε μια αναδρομική σχέση για το πλήθος των συγκρίσεων του merge sort στην καλύτερη περίπτωση

Λύση

Στην καλύτερη περίπτωση ο πίνακας θα είναι ήδη ταξινομημένος. Συνεπώς η αναδρομική σχέση είναι:

$$C_b(n) = 2C_b(n/2) + n/2 \ \text{ for } n > 1 \ , \quad C_b(1) = 0$$

Η λύση της έχει ως εξής (για n άρτιο):

$$\begin{aligned}
C_b(2^k) &= 2C_b(2^{k-1}) + 2^{k-1} \\
&= 2[2C_b(2^{k-2}) + 2^{k-2}] + 2^{k-1} = 2^2C_b(2^{k-2}) + 2^{k-1} + 2^{k-1} \\
&= 2^2[2C_b(2^{k-3}) + 2^{k-3}] + 2^{k-1} + 2^{k-1} = 2^3C_b(2^{k-3}) + 2^{k-1} + 2^{k-1} + 2^{k-1} \\
&= \ldots \\
&= 2^iC_b(2^{k-i}) + i2^{k-1} \\
&= \ldots \\
&= 2^kC_b(2^{k-k}) + k2^{k-1} = k2^{k-1} = \frac{1}{2}n\log n
\end{aligned}$$

Θεώρημα

Οποιοσδήποτε αλγόριθμος ταξινόμησης απαιτεί Ω(nlogn) στη χειρότερη περίπτωση

Απόδειξη

Για την απόδειξη θα υιοθετήσουμε τα δένδρα απόφασης / σύγκρισης (decision trees) για τους αλγορίθμους ταξινόμησης

Ένα δένδρο απόφασης είναι ένα δένδρο που αναπαριστά τις συγκρίσεις που υλοποιεί ένας αλγόριθμος ταξινόμησης

Έλεγχοι, αντιμεταθέσεις, κ.λπ. αγνοούνται σε ένα δένδρο απόφασης

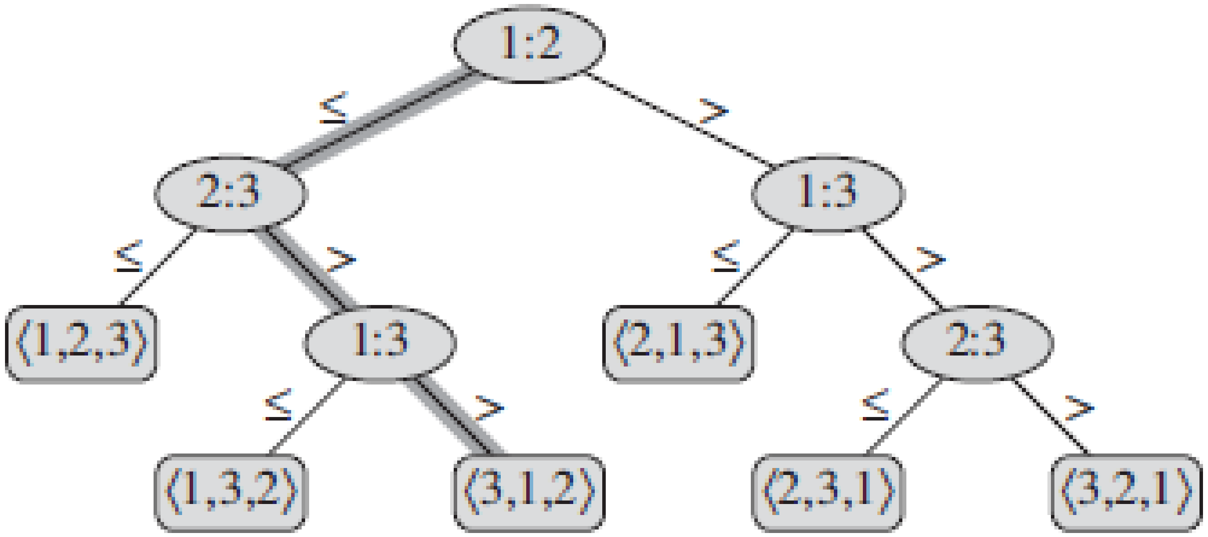
Θεώρημα

Το ακόλουθο δένδρο παρουσιάζει τις συγκρίσεις του insertion sort για ένα πίνακα 3 θέσεων

Στο δένδρο, ο κάθε κόμβος με συμβολισμό i:j παριστάνει τη σύγκριση των a_i, a_j

Ακολουθήστε τη διαδρομή για τον πίνακα

$\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$



Θεώρημα

Συνεπώς, για οποιοδήποτε δένδρο αν έχουμε ύψος h και l φύλλα που αντιστοιχούν στην ταξινόμηση n στοιχείων τότε:

- Οι αναδιατάξεις θα είναι n!
- Το ύψος του δένδρου θα είναι 2^h
- Άρα

$$n! \le l \le 2^h$$

- Λογαριθμίζοντας παίρνουμε (χρησιμοποιούμε τις μαθηματικές πράξεις που παρουσιάστηκαν στις πρώτες διαλέξεις)

$$\begin{aligned} h &\ge \lg(n!) \\ &= \Omega(n \lg n) \end{aligned}$$

Decrease and Conquer (συνέχεια)

Combinatorial Objects

Οι πιο σημαντικοί αλγόριθμοι αφορούν:

- Αναδιατάξεις (permutations)
- Συνδυασμοί (combinations)
- Εύρεση υποσυνόλων ενός δοσμένου συνόλου

Μαθηματικοί τύποι να υποδεικνύουν το πόσα αντικείμενα πρέπει να παραχθούν

Η αύξηση του πλήθους των αντικειμένων αυξάνει εκθετικά σε σχέση με το μέγεθος του προβλήματος

Το ενδιαφέρον μας είναι στους αλγορίθμους που παράγουν τα αντικείμενα αυτά

Αναδιατάξεις (1/7)

Θεωρούμε πως έχουμε στη διάθεσή μας τους αριθμούς από το 1 μέχρι το n

Θέλουμε να παράξουμε όλες τις πιθανές αναδιατάξεις των αριθμών αυτών

Το πλήθος των αναδιατάξεων είναι n!

Η τεχνική της μείωσης κατά 1 μας υποδεικνύει να παράξουμε τις (n-1)! αναδιατάξεις

Αν επιλύσουμε το μικρότερο πρόβλημα, τότε η λύση είναι προφανής: τοποθετούμε το n σε κάθε μια από τις θέσεις των αναδιατάξεων

Αναδιατάξεις (2/7)

Ο συνολικός αριθμός θα είναι n(n-1)!

Μπορούμε να εισάγουμε το n είτε ξεκινώντας από αριστερά προς τα δεξιά ή από τα δεξιά προς τα αριστερά

Είναι αποδοτικό αν ξεκινήσουμε να εισάγουμε το n στο 1,2,…,(n-1) ξεκινώντας από δεξιά προς τα αριστερά και να αλλάζουμε κατεύθυνση κάθε φορά που μια αναδιάταξη παράγεται

Παράδειγμα

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

Αναδιατάξεις (3/7)

Το πλεονέκτημα έχει να κάνει λόγω του γεγονότος ότι αυτή η μέθοδος ικανοποιεί την απαίτηση της μικρότερης αλλαγής (***minimal change requirement)***

- Κάθε αναδιάταξη μπορεί να εξαχθεί από την αμέσως προηγούμενη ανταλλάσσοντας μόνο δύο στοιχεία σε αυτή

Η ιδιότητα αυτή είναι ευεργετική για την ταχύτητα του αλγορίθμου και για τις εφαρμογές που υιοθετούν αναδιατάξεις

Αναδιατάξεις (4/7)

Μπορούμε να αποδώσουμε μια κατεύθυνση σε κάθε αριθμό

Βάζουμε ένα βέλος πάνω από τον κάθε αριθμό

Παράδειγμα: $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$

Ένα στοιχείο k θεωρείται ότι είναι κινητό αν το βέλος του δείχνει σε ένα μικρότερο στοιχείο (διπλανό στοιχείο)

Στο παράδειγμα το 3 και το 4 είναι κινητά στοιχεία

Με την έννοια του κινητού στοιχείου έχουμε τον αλγόριθμο Johnson – Trotter για την παραγωγή αναδιατάξεων

Αναδιατάξεις (5/7)

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Αναδιατάξεις (6/7)

Παράδειγμα εκτέλεσης του αλγορίθμου (n=3)

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}\quad \overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2}\quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}\quad \overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}$$

Πρόκειται για ένα αποδοτικό αλγόριθμο

Εκτελείται σε Θ(n!) που φυσικά δεν πρόκειται για ικανοποιητική πολυπλοκότητα

Το πρόβλημα δεν είναι ο αλγόριθμος αλλά το ίδιο το πρόβλημα που ζητά την παραγωγή πολλών στοιχείων / αναδιατάξεων

Αναδιατάξεις (7/7)

Ερώτημα

- Μπορούμε να δημιουργήσουμε αναδιατάξεις σε λεξικογραφική / αύξουσα σειρά?

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
    let i be its largest index such that $a_i < a_{i+1}$ //$a_{i+1} > a_{i+2} > \cdots > a_n$
    find the largest index j such that $a_i < a_j$ //$j \geq i + 1$ since $a_i < a_{i+1}$
    swap a_i with a_j //$a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
    add the new permutation to the list

Εύρεση Υποσυνόλων (1/6)

Το knapsack problem είναι ένα παράδειγμα

Για ένα σύνολο στοιχείων A={α_1, α_2, …, α_n} θα παράξουμε τα 2^n υποσύνολα

Η τεχνική της μείωσης κατά ένα λειτουργεί ως εξής:

- Το A μπορεί να χωριστεί σε δύο ομάδες: σε αυτό που περιέχει το α_n και σε αυτό που δεν το περιέχει
- Η πρώτη ομάδα είναι το σύνολο των υποσυνόλων των n-1 στοιχείων
- Η δεύτερη παράγεται αν προσθέσουμε το α_n

Εύρεση Υποσυνόλων (2/6)

Παράδειγμα

n	subsets							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Εύρεση Υποσυνόλων (3/6)

Μπορούμε να χρησιμοποιήσουμε μια συμβολοσειρά δυαδικών ψηφίων ως εξής:

- Για τη συμβολοσειρά b_1, b_2, …, b_n το $b_i = 1$ αν το α_i ανήκει στο υποσύνολο και $b_i = 0$ αν το α_i δεν ανήκει στο υποσύνολο

Έχουμε 2^n συμβολοσειρές

Το κενό υποσύνολο συμβολίζεται με το 000 για n=3

Δημιουργούμε όλες τις συμβολοσειρές αν δημιουργήσουμε όλους τους δυαδικούς αριθμούς από το 0 μέχρι το $2^n -1$

Εύρεση Υποσυνόλων (4/6)

Παράδειγμα

bit strings	000	001	010	011	100	101	110	111
subsets	$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Παρά το γεγονός ότι οι συμβολοσειρές παράγονται σε λεξικογραφική σειρά, τα υποσύνολα δεν μοιάζουν να έχουν ‘φυσιολογική’ σειρά

Εύρεση Υποσυνόλων (5/6)

Η squashed order υποδεικνύει ότι το κάθε υποσύνολο που εμπλέκει το αj θα προκύψει μόνο μετά από τα υποσύνολα που εμπλέκουν τα στοιχεία $\alpha_1, \alpha_2, \ldots, \alpha_{j-1}$

Ερώτημα

- Μπορούμε να παράξουμε τον αλγόριθμο της ελάχιστης αλλαγής ώστε κάθε string να διαφέρει από το προηγούμενο κατά μόνο ένα bit?
- Παράδειγμα:

000 001 011 010 110 111 101 100.

Εύρεση Υποσυνόλων (6/6)

Μια ακολουθία που ικανοποιεί την ελάχιστη αλλαγή στην παραγωγή υποσυνόλων ονομάζεται ***binary reflected Gray code***

ALGORITHM $BRGC(n)$

//Generates recursively the binary reflected Gray code of order n
//Input: A positive integer n
//Output: A list of all bit strings of length n composing the Gray code
if $n = 1$ make list L containing bit strings 0 and 1 in this order
else generate list $L1$ of bit strings of size $n - 1$ by calling $BRGC(n - 1)$
    copy list $L1$ to list $L2$ in reversed order
    add 0 in front of each bit string in list $L1$
    add 1 in front of each bit string in list $L2$
    append $L2$ to $L1$ to get list L
return L

Δυναμικός Προγραμματισμός

Εισαγωγή (1/3)

Ο **δυναμικός προγραμματισμός (dynamic programming)** επιλύει τα προβλήματα μέσω του συνδυασμού των λύσεων ενός αριθμοούύ υποπροβλημάτων

Διαίρει και βασίλευε

- Διασπά το πρόβληματα σε διακριτά υποπροβλήματα, επιλύει το καθένα από αυτά αναδρομικά και στη συνέχεια συνδυάζει τις λύσεις

Δυναμικός προγραμματισμός

- Εφαρμόζεται όταν τα υποπροβλήματα επικαλύπτονται
- Επιλύει κάθε υπο-υποπρόβλημα μόνο μια φορά και αποθηκεύει την λύση / απάντηση σε ένα πίνακα

Εισαγωγή (2/3)

- Αποφεύγει την προσπάθεια εύρεσης των λύσεων κάθε φορά που επιλύει ένα υπο-υποπρόβλημα

Εφαρμόζουμε το δυναμικό προγραμματισμό σε προβλήματα βελτιστοποίησης

Τα προβλήματα βελτιστοποίησης μπορεί να έχουν πολλαπλές λύσεις

Κάθε λύση έχει μια τιμή / αξία και προσπαθούμε να βρούμε τη λύση με τη μεγαλύτερη αξία

Η λύση αυτή καλείται η **βέλτιστη (optimal)** λύση

Εισαγωγή (3/3)

Κατά την ανάπτυξη αλγορίθμων δυναμικού προγραμματισμού εφαρμόζουμε τα ακόλουθα βήματα:

- Χαρακτηρίζουμε τη δομή μιας βέλτιστης λύσης
- Ορίζουμε αναδρομικά την τιμή / αξία μιας βέλτιστης λύσης
- Υπολογίζουμε την τιμή των βέλτιστων λύσεων τυπικά μέσω της προσέγγισης bottom-up
- Δημιουργούμε μια βέλτιστη λύση από τις παραπάνω πληροφορίες

Rod Cutting (1/14)

Παράδειγμα: διαίρεση μιας ράβδου (rod cutting)

Ας υποθέσουμε ότι μια επιχείρηση αγοράζει ατσάλινες ράβδους και στη συνέχεια της διαιρεί και τις πουλάει ανεξάρτητα

Γνωρίζουμε την τιμή $p_{i,}$ i=1, 2, 3, … στην οποία η επιχείρηση πουλάει το κάθε κομμάτι

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod Cutting (2/14)

Πρόβλημα

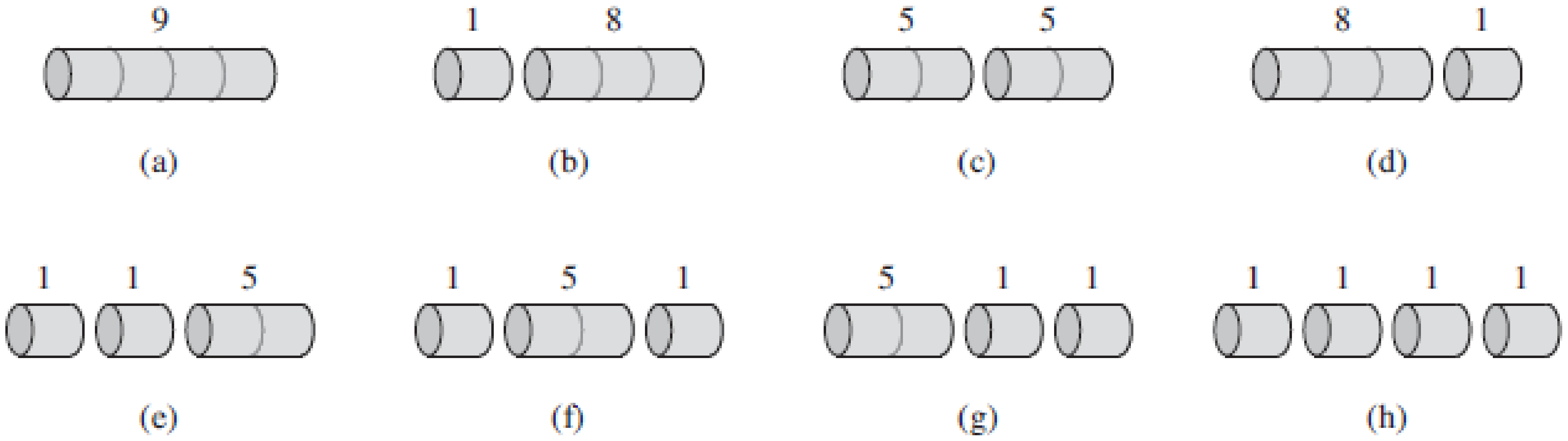
Δοσμένης μιας ράβδου μήκους n και ενός πίνακα τιμών για διάφορα μεγέθη κάποιων τμημάτων της ράβδου, να καθορίσουμε το μέγιστο κέρδος r_n το οποίο θα αποκομίσουμε αν διαιρέσουμε τη ράβδο σε κομμάτια και τα πουλήσουμε

Σημείωση: αν η τιμή p_n είναι αρκετά μεγάλη, τότε ίσως να μην χρειάζεται να διαιρέσουμε τη ράβδο

Rod Cutting (3/14)

Ας υποθέσουμε ότι n=4. Τότε έχουμε το ακόλουθο σχήμα:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



(a) (b) (c) (d)

(e) (f) (g) (h)

Rod Cutting (4/14)

Μπορούμε να διαιρέσουμε τη ράβδο με 2^{n-1} διαφορετικούς τρόπους

Η επιλογή μας θα είναι κάθε φορά να προχωρήσουμε στη διαίρεση ή όχι σε απόσταση i από αριστερά για i=1,2,…,n-1

Παράδειγμα:

- Η διαίρεση 6=1+2+3 σημαίνει ότι διαιρούμε μια ράβδο μήκους 6 σε 1, 2 και 3 μέτρα αντίστοιχα

Rod Cutting (5/14)

Αν μια βέλτιστη λύση είναι να διαιρέσουμε τη ράβδο σε k τμήματα για κάποιο 1<=k<=n, τότε μια βέλτιστη υποδιαίρεση

$$n = i_1 + i_2 + \cdots + i_k$$

Σε τμήματα μήκους

$$i_1,\ i_2,\ \ldots,\ i_k$$

Μας δίνει κέρδος

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

Rod Cutting (6/14)

Γενικά, θα μπορούσαμε να προσδιορίσουμε το κέρδος σε σχέση με το βέλτιστο κέρδος κάθε προηγούμενου διαχωρισμού

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Το p_n αντιστοιχεί στο κέρδος της μη διαίρεσης της ράβδου

Οι υπόλοιπες υποδιαιρέσεις αφορούν στο κέρδος που προκύπτει από την υποδιαίρεση της ράβδου σε 2 κομμάτια

Για να λύσουμε το πρόβλημα που έχει μέγεθος n, πρέπει να λύσουμε ιδίου τύπου υποπροβλήματα αλλά με μικρότερο μέγεθος

Αν κάνουμε την πρώτη υποδιαίρεση, τότε θεωρούμε τα δύο κομμάτια σαν ανεξάρτητες ‘εκδόσεις’ του αρχικού προβλήματος

Rod Cutting (7/14)

Το πρόβλημα επιδεικνύει **optimal substructure**

- Οι βέλτιστες λύσεις σε ένα πρόβλημα ενσωματώνουν τις βέλτιστες λύσεις στα σχετικά υποπροβλήματα τα οποία μπορούν να λυθούν ανεξάρτητα

Προσέγγιση:

- Θεωρούμε μια διαίρεση της ράβδου που αποτελείται από το αριστερό κομμάτι που δεν μπορεί να διαιρεθεί περαιτέρω και το δεξιό το οποίο θα διαιρεθεί στη συνέχεια
- Παίρνουμε την ακόλουθη εξίσωση για το κέρδος $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$
- Η εξίσωση περιλαμβάνει την επίλυση ενός υποπροβλήματος και όχι δύο

Rod Cutting (8/14)

Top-down προσέγγιση

Είσοδοι: πίνακας p[1..n] με τιμές και η τιμή n

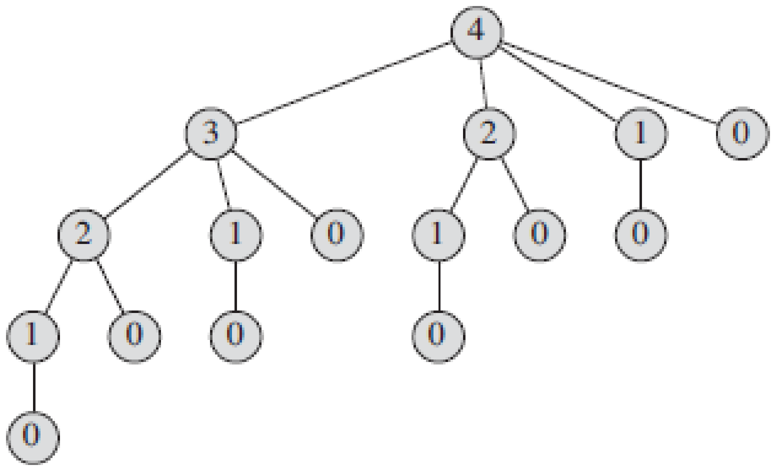
Μη αποδοτική λύση – λύνει τα ίδια υποπροβλήματα συνεχώς μέσω της αναδρομικής κλήσης

Παράδειγμα για n=4

Πολυπλοκότητα $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

και άρα $T(n) = 2^n$

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```



Rod Cutting (9/14)

Χρήση δυναμικού προγραμματισμού

- Καθορίζουμε **την επίλυση κάθε υποπροβλήματος μόνο μια φορά** και όχι πολλαπλές όπως με την αναδρομική κλήση
- Αποθηκεύουμε τη λύση έτσι ώστε αν χρειαστεί **να την προσπελάσουμε μελλοντικά**, τότε να τη δούμε απλά και να μην την ξανα-υπολογίσουμε
- Προφανώς, η λύση μέσω δυναμικού προγραμματισμού **θα απαιτήσει επιπλέον μνήμη αλλά γλιτώνει υπολογιστικό χρόνο**
- Μπορεί μια λύση εκθετικού χρόνου να μετατραπεί σε λύση πολυωνυμικού χρόνου

Rod Cutting (10/14)

Χρήση δυναμικού προγραμματισμού

- Top-down
  - Τροποποιούμε την αναδρομική λύση ώστε να αποθηκεύεται κάθε φορά η λύση των υποπροβλημάτων
- Bottom-up
  - Ταξινομούμε τα υποπροβλήματα κατά μέγεθος και λύνουμε πρώτα αυτά που έχουν μικρότερο μέγεθος

Rod Cutting (11/14)

Top-down with memorization

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Rod Cutting (12/14)

Bottom-up approach

- Πολυπλοκότητα Θ(n^2)
  - **Γιατί?**
  - **Πόσο έχει η top-down?**

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Rod Cutting (13/14)

Οι προηγούμενες λύσεις επιστρέφουν το μέγιστο κέρδος άλλα όχι τη βέλτιστη διαίρεση της ράβδου

Οι διπλανοί αλγόριθμοι υπολογίζουν και τυπώνουν το βέλτιστο διαχωρισμό

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n − s[n]
```

Rod Cutting (14/14)

Παράδειγμα εκτέλεσης

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Τι θα τυπωθεί με τις ακόλουθες κλήσεις?

PRINT-CUT-ROD-SOLUTION$(p, 10)$

PRINT-CUT-ROD-SOLUTION$(p, 7)$

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

Longest Common Subsequence (1/12)

Χρησιμοποιείται κυρίως σε βιολογικές εφαρμογές

Παράδειγμα

- Σύγκριση αλυσίδων DNA

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

Προσπαθούμε να βρούμε πόσο μοιάζουν δύο συμβολοσειρές

Υιοθετούμε συγκεκριμένες μετρικές

Longest Common Subsequence (2/12)

Φορμαλισμός

- Δοσμένης μιας συμβολοσειράς $X=(x_1, x_2, \ldots, x_m)$ μια άλλη συμβολοσειρά $Z=(z_1, z_2, \ldots, z_k)$ είναι μια **υποακολουθία** (subsequence) της X, αν υπάρχει μια αυστηρά 'αυξανόμενη' ακολουθία $<i_1, i_2, \ldots, i_k>$ στοιχείων/δεικτών της X ώστε για όλα τα $j=1,2,\ldots,k$ ισχύει ότι $x_{i_j}=z_j$
- Η Z=(B,C,D,B) είναι μια υποακολουθία της X=(A,B,C,B,D,A,B) για τους δείκτες (2,3,5,7)
- Δοσμένων δύο συμβολοσειρών X, Y, λέμε ότι η Z είναι μια **κοινή υποακολουθία** (common subsequence) των X και Y όταν η Z είναι υποακολουθία της X και της Y

Longest Common Subsequence (3/12)

Παράδειγμα

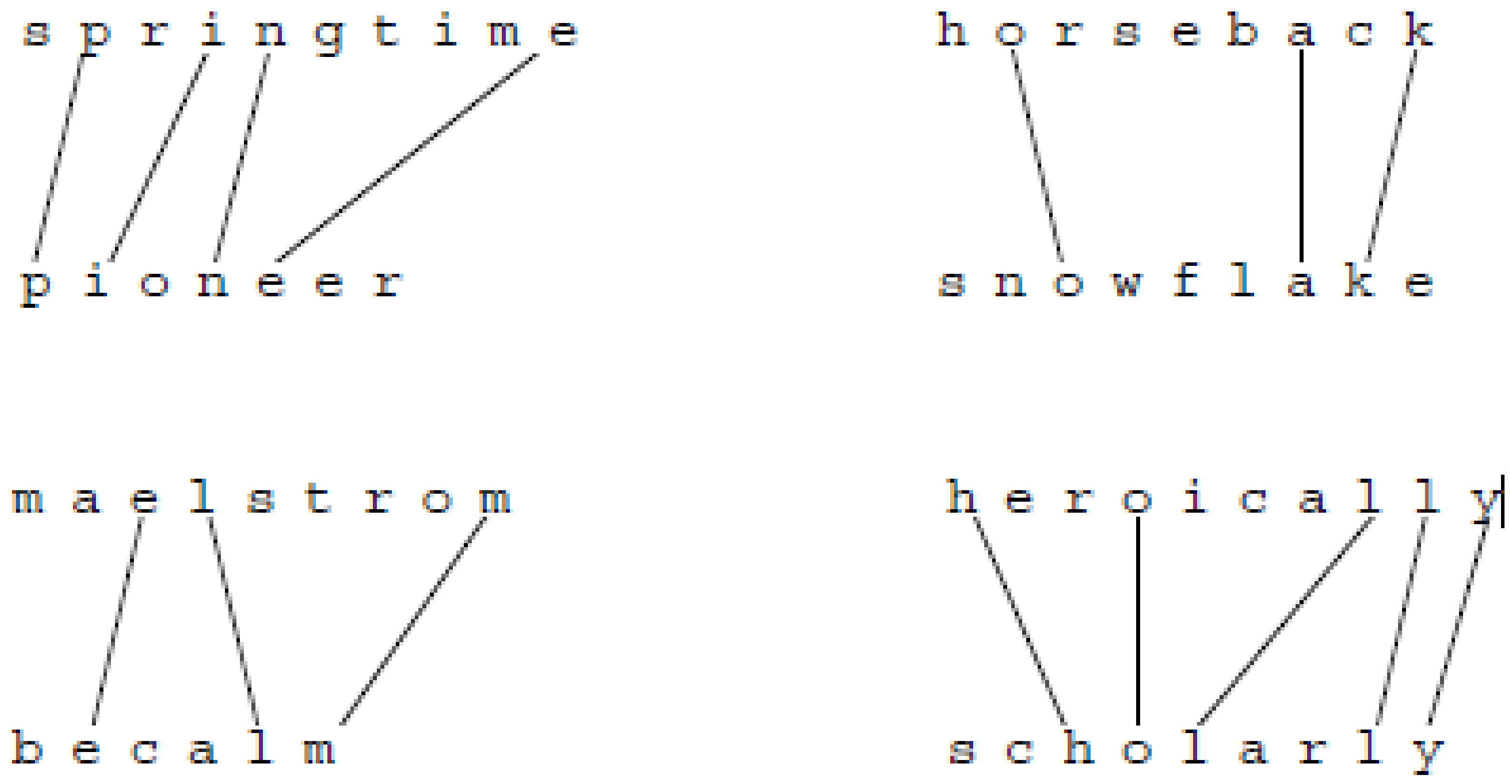
- X=(A,B,C,B,D,A,B), Y=(B,D,C,A,B,A)
  - Η (B,C,A) είναι μια κοινή υποακολουθία αλλά δεν είναι **η μέγιστη κοινή υποακολουθία (longest common subsequence -LCS)**
  - Η (B,C,A) έχει μήκος 3 ενώ η (B,C,B,A) και (B,D,A,B) έχουν μήκος 4 (και οι δύο είναι LCS)

Πρόβλημα

- Μας δίνονται δύο ακολουθίες X=(x_1, x_2, …, x_m) και Y=(y_1, y_2, …, y_n) και μας ζητείται να βρούμε την LCS

Longest Common Subsequence (4/12)

Παραδείγματα



Longest Common Subsequence (5/12)

Μια brute force λύση θα απαριθμούσε όλες τις κοινές υποακολουθίες και στη συνέχεια θα έβρισκε τη μέγιστη

Λόγω του ότι η X έχει 2^m υποακολουθίες, ο χρόνος που απαιτείται είναι εκθετικός

Μέσω δυναμικού προγραμματισμού προσπαθούμε να βρούμε τα prefixes των συμβολοσειρών

Longest Common Subsequence (6/12)

Ισχύει το ακόλουθο θεώρημα

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Longest Common Subsequence (7/12)

Το θεώρημα ουσιαστικά μας λέει:

- Η LCS δύο ακολουθιών περιέχει μια LCS κάποιων prefixes των δύο ακολουθιών
- Με αυτό το σκεπτικό η εύρεση της LCS έχει της ιδιότητα της βέλτιστης υποδομής (optimal substructure property)

Βασιζόμενοι στο προηγούμενο θεώρημα, μπορούμε να ορίσουμε μια επαναληπτική λύση

Longest Common Subsequence (8/12)

Το θεώρημα υποδεικνύει ότι:

- Αν $x_m=y_n$, πρέπει να βρούμε την LCS των X_{m-1}, Y_{n-1}. Στη συνέχεια προσθέτοντας τα x_m, y_n, έχουμε την LCS των αρχικών ακολουθιών
- Αν $x_m<>y_n$, πρέπει να λύσουμε δύο υπο-προβλήματα:
  - Εύρεση της LCS των X_{m-1}, Y
  - Εύρεση της LCS των X, Y_{n-1}
  - Όποια από αυτές τις δύο LCSs είναι μεγαλύτερη θα είναι και η τελική LCS
  - Κάθε ένα από αυτά τα υπο-προβλήματα εμπλέκει την εύρεση της LCS για τα X_{m-1}, Y_{n-1}

Longest Common Subsequence (9/12)

Έστω c[i,j] το μήκος μιας LCS των Xi, Yj

Αν i=0 ή j=0, τότε μια από τις δύο ακολουθίες έχει μήκος 0 οπότε η LCS έχει μήκος 0

Η αναδρομική σχέση έχει ως εξής

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Longest Common Subsequence (10/12)

Αλγόριθμος

- Γεμίζει τον πίνακα c γραμμές
- Ο πίνακας b δείχνει στο κελί που αντιστοιχεί στη βέλτιστη λύση του υπο-προβλήματος όταν υπολογίζουμε το c[I,j]
- Το μήκος της LCS βρίσκεται στο c[m,n]

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

Longest Common Subsequence (11/12)

Παράδειγμα εκτέλεσης

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest Common Subsequence (12/12)

Εκτύπωση της LCS

- Κλήση

  PRINT-LCS$(b, X, X.length, Y.length)$

PRINT-LCS(b, X, i, j)

```
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Knapsack Problem (1/7)

Υιοθετούμε λύση με χρήση δυναμικού προγραμματισμού

Αντικείμενα: n, βάρη: w_i, αξίες: v_j

Χωρητικότητα σάκου: W

Για τη λύση του δυναμικού προγραμματισμού πρέπει να βρούμε την αναδρομική σχέση

Σε κάθε βήμα, παρουσιάζεται ένα στιγμιότυπο του σάκου σε σχέση με τις λύσεις στα μικρότερα προβλήματα

Knapsack Problem (2/7)

Εστιάζουμε σε ένα στιγμιότυπο των πρώτων i αντικειμένων

Αντικείμενα: $1<=i<=n$, Βάρη: $w_1, w_2, \ldots, w_i$, αξίες: $v_1, v_2, \ldots, v_i$

Χωρητικότητα σάκου: $1<=j<=W$

Έστω $F(i,j)$ είναι η αξία μιας βέλτιστης λύσης

Μπορούμε να διαιρέσουμε όλα τα υποσύνολα των πρώτων i αντικειμένων που ταιριάζουν στη χωρητικότητα j του σάκου σε δύο κατηγορίες:

- Αυτά που δεν περιέχουν το i αντικείμενο
- Αυτά που περιέχουν το i αντικείμενο

Knapsack Problem (3/7)

Στα σύνολα που δεν περιέχουν το i αντικείμενο, η αξία ενός βέλτιστου υποσυνόλου είναι F(i-1,j)

Στα σύνολα που δεν περιέχουν το i αντικείμενο, ένα βέλτιστο υποσύνολο αποτελείται από ένα βέλτιστο υποσύνολο των πρώτων i-1 αντικειμένων που χωράνε στο σάκο με χωρητικότητα $j\text{-}w_i$

- Η αξία τους είναι $v_i+F(i-1,j-w_i)$

Συνεπώς

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0 \\ F(i-1, j) & \text{if } j - w_i < 0 \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0$$

Knapsack Problem (4/7)

Για να υπολογίσουμε την τιμή ενός κελιού σε μια γραμμή i και στήλη j, F(i,j), υπολογίζουμε το μέγιστο ανάμεσα στην ***προηγούμενη γραμμή και την ίδια στήλη***, και στην εγγραφή ***στην προηγούμενη γραμμή κα w_i στήλες προς τα αριστερά***

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Knapsack Problem (5/7)

Παράδειγμα

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Knapsack Problem (6/7)

Απαλείφουμε τα πλεονάζοντα υποπροβλήματα με χρήση των memory functions – επιλύουμε μόνο τα απαραίτητα υπο-προβλήματα

ALGORITHM *MFKnapsack*(i, j)

//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
// items being considered and a nonnegative integer j indicating
// the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays $Weights[1..n]$, $Values[1..n]$,
//and table $F[0..n, 0..W]$ whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's
if $F[i, j] < 0$
  if $j < Weights[i]$
    $value \leftarrow MFKnapsack(i-1, j)$
  else
    $value \leftarrow \max(MFKnapsack(i-1, j),$
      $Values[i] + MFKnapsack(i-1, j - Weights[i]))$
  $F[i, j] \leftarrow value$
return $F[i, j]$

Knapsack Problem (7/7)

Παράδειγμα εκτέλεσης

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Άπληστη Μέθοδος

Εισαγωγή (1/2)

Η άπληστη μέθοδος στοχεύει στη λύση που φαίνεται η βέλτιστη μέχρι στιγμής

Επιλέγει την τοπικά βέλτιστη λύση με την ελπίδα ότι αυτή η λύση θα είναι τελικά η βέλτιστη

Μπορούμε να συνθέσουμε μια ολικά βέλτιστη λύση μέσα από την επιλογή τοπικών βέλτιστων λύσεων

Οι άπληστοι αλγόριθμοι δεν δίνουν πάντα τις βέλτιστες

Πρόκειται για μια ισχυρή μέθοδο που χρησιμοποιείται σε μεγάλο εύρος προβλημάτων

Εισαγωγή (2/2)

Διαφορές με το δυναμικό προγραμματισμό

- Στην άπληστη μέθοδο επιλέγουμε την τοπικά βέλτιστη λύση χωρίς να εστιάζουμε στις λύσεις των υπο-προβλημάτων
- Οι επιλογές που κάνουμε στο δυναμικό προγραμματισμό βασίζονται στις λύσεις των υπο-προβλημάτων
- Τυπικά στο δυναμικό προγραμματισμό ακολουθούμε μια bottom-up προσέγγιση όπου από πιο απλά προβλήματα φτάνουμε στη λύση πιο σύνθετων προβλημάτων
- Οι επιλογές στην άπληστη μέθοδο μπορεί να βασίζονται σε παλιές επιλογές αλλά ποτέ δεν βασίζονται σε μελλοντικές
- Ο δυναμικός προγραμματισμός επιλύει τα υπο-προβλήματα πριν κάνει μια επιλογή, ενώ η άπληστη μέθοδος κάνει την επιλογή πριν λύσει τα υπο-προβλήματα

Κώδικες Huffman (1/19)

Οι κώδικες Huffman στοχεύουν στη συμπίεση δεδομένων

Θεωρούμε τα δεδομένα ως ακολουθίες χαρακτήρων

Η προσέγγιση με την άπληστη μέθοδο υιοθετεί ένα πίνακα που αποθηκεύει πόσο συχνά εμφανίζεται ο κάθε ένας χαρακτήρας

Απεικονίζει κάθε χαρακτήρα σαν ένα binary string

Κώδικες Huffman (2/19)

Παράδειγμα

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Με την fixed length μέθοδο χρειαζόμαστε 300.000 bits για ένα αρχείο των 100.000 χαρακτήρων

Με την variable length μέθοδο χρειαζόμαστε

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

Κώδικες Huffman (3/19)

Εστιάζουμε στα prefix codes, κώδικες που δεν είναι προθέματα άλλων κωδίκων

Η τελική κωδικοποίηση είναι η συγχώνευση των κωδίκων για κάθε χαρακτήρα

Αν το prefix code είναι μοναδικό για κάθε χαρακτήρα τότε η αποκωδικοποίηση είναι εύκολη

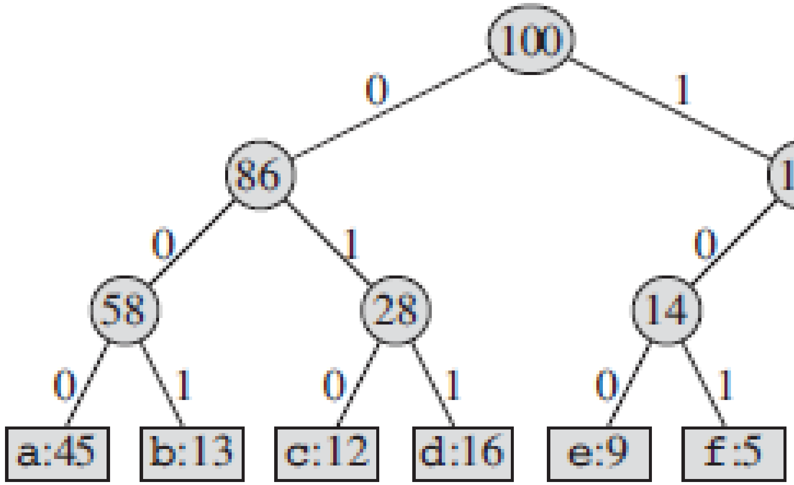
Υιοθετούμε ένα δυαδικό δένδρο στο οποίο τα φύλλα είναι οι χαρακτήρες

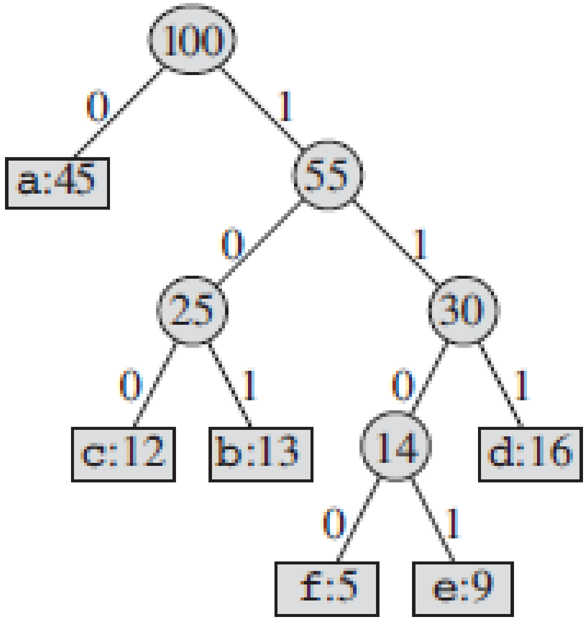
Ο δυαδικός κώδικας είναι η διαδρομή από τη ρίζα του δένδρου προς τα φύλλα

Κώδικες Huffman (4/19)

Το 0 αντιστοιχεί στην κίνηση προς το αριστερό παιδί, ενώ το 1 την κίνηση προς το δεξιό παιδί

Παράδειγμα





Κώδικες Huffman (5/19)

Ο βέλτιστος κώδικας εμπλέκει ένα πλήρες δυαδικό δένδρο στο οποίο ο κάθε κόμβος έχει δύο παιδιά

Η fixed length προσέγγιση δεν οδηγεί σε βέλτιστο κώδικα

Έστω C το αλφάβητο για το οποίο πρέπει να κατασκευάσουμε τον κώδικα

Το δένδρο έχει |C| φύλλα

Το δένδρο έχει ακριβώς |C|-1 εσωτερικούς κόμβους

Κώδικες Huffman (6/19)

Δοσμένου ενός δένδρου T που αντιστοιχεί σε ένα κώδικα, μπορούμε να υπολογίσουμε εύκολα το πλήθος των bits που απαιτούνται για να κωδικοποιήσουμε το αρχείο

Έστω c ένας χαρακτήρας του C, c.freq είναι η συχνότητα του c και $d_T(c)$ είναι το βάθος του c στο δένδρο

Το $d_T(c)$ είναι ταυτόχρονα και το μήκος του κώδικα του c

Το πλήθος των bits θα είναι

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

Κώδικες Huffman (7/19)

Υποθέτουμε ότι το C είναι ένα σύνολο n χαρακτήρων

Ο άπληστος αλγόριθμος ξεκινά από τα φύλλα και εκτελεί |C|-1 συγχωνεύσεις για να δημιουργήσει το τελικό δένδρο

Υιοθετεί μια ουρά ελάχιστης προτεραιότητας (min-priority queue)

Μέσω της ουράς αναγνωρίζει τα δύο ελάχιστα συχνά στοιχεία για να συγχωνεύσει

Όταν γίνεται η συγχώνευση το αποτέλεσμα είναι ένα νέο στοιχείο του οποίου η συχνότητα είναι το άθροισμα των συχνοτήτων των δύο στοιχείων

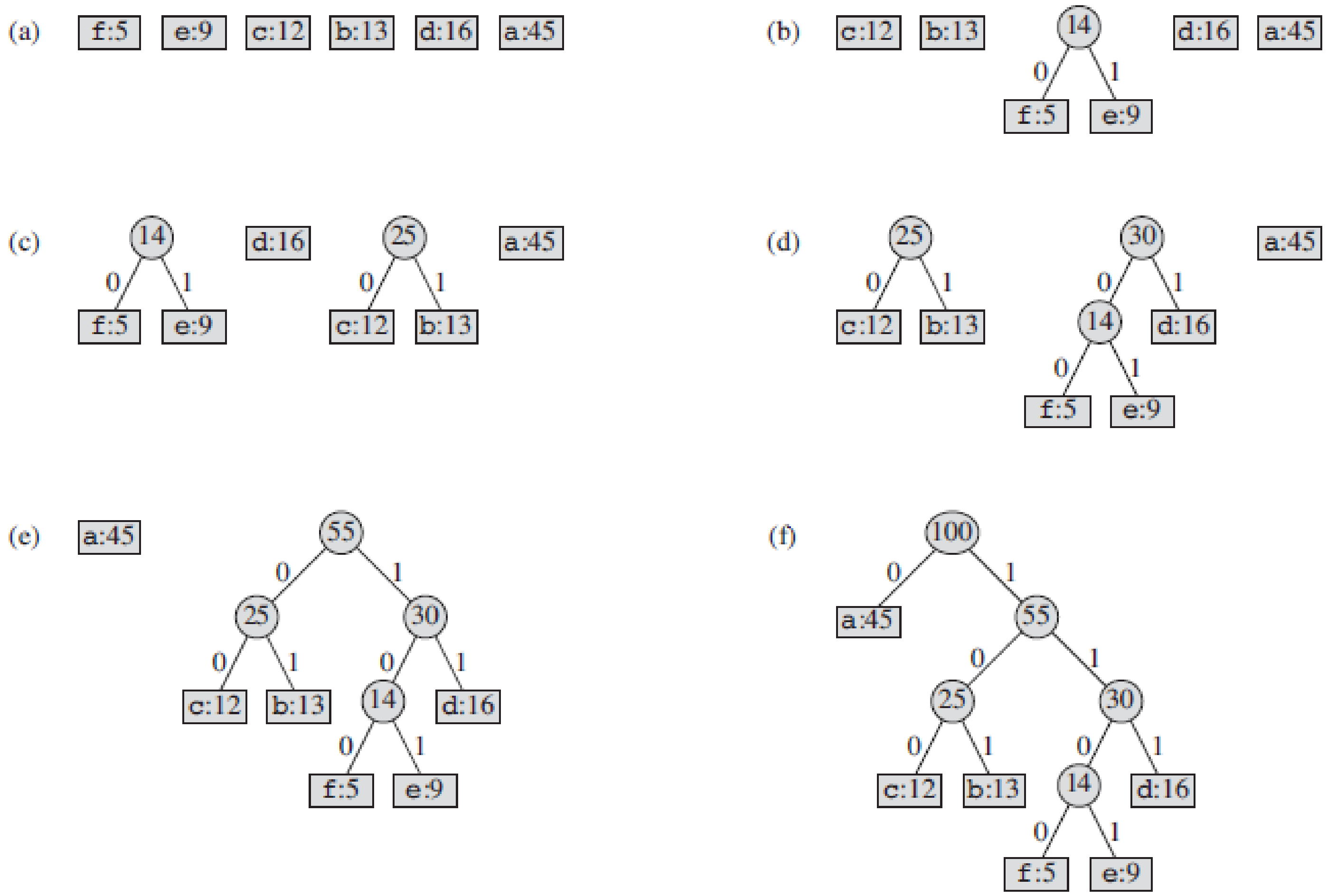
Κώδικες Huffman (8/19)

Αλγόριθμος

- Η αρχική ουρά έχει μέγεθος ίσο με το n
- Οι γραμμές 3-8 εξάγουν επαναληπτικά τους δύο κόμβους x, y με την ελάχιστη συχνότητα αντικαθιστώντας τους στην ουρά με ένα νέο κόμβο z
- Η συχνότητα του z είναι το άθροισμα των συχνοτήτων
- Ο z έχει αριστερό παιδί το x και δεξιό παιδί το y

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)    // return the root of the tree
```

Κώδικες Huffman (9/19)



Κώδικες Huffman (10/19)

Lemma 1

Για ένα αλφάβητο C όπου ο κάθε χαρακτήρας c έχει συχνότητα c.freq, αν x, y είναι οι δύο χαρακτήρες με τη χαμηλότερη συχνότητα, τότε υπάρχει ένα βέλτιστο πρόθεμα (prefix) στο οποίο οι κώδικες για τα x, y έχουν το ίδιο μήκος αλλά διαφέρουν στο τελευταίο bit

Απόδειξη

Έστω a, b δύο χαρακτήρες που είναι αδέρφια σε μέγιστο βάθος στο T.

Υποθέτουμε ότι $a.freq \leq b.freq$ και $x.freq \leq y.freq$.

Κώδικες Huffman (11/19)

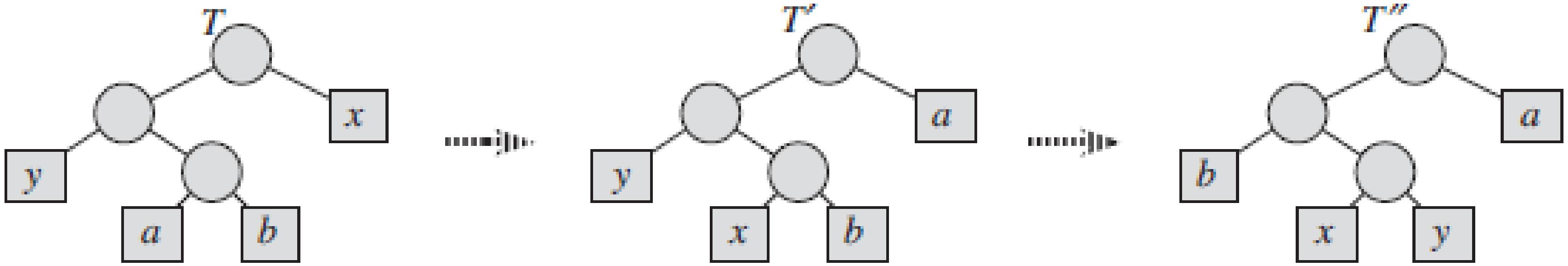
Αφού τα x.freq & y.freq είναι οι ελάχιστες συχνότητες, και a.freq & b.freq είναι δύο συχνότητες χαρακτήρων θα έχουμε x.freq<=a.freq & y.freq<=b.freq

Αν x.freq=a.freq & y.freq=b.freq τότε θα έχουμε x.freq=a.freq=y.freq=b.freq οπότε το λήμμα ισχύει

Αν υποθέσουμε ότι x.freq<>b.freq, τότε θα είναι x<>b

Προχωρούμε σε αλλαγή στο δένδρο με αντιμεταθέσεις θέσεων στα a,x & b,y ως ακολούθως:

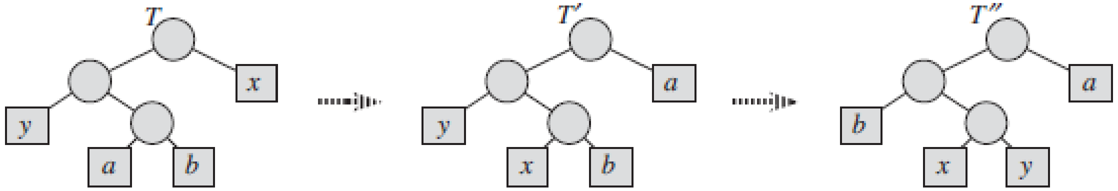
Κώδικες Huffman (12/19)



Αν x=b & y<>a το T’’ δεν έχει τα x,y σαν ‘αδέρφια’

Η διαφορά στο κόστος των δένδρων T, T’ είναι:

Κώδικες Huffman (13/19)



$$
\begin{aligned}
B(T) - B(T') \\
&= \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&= (a.\mathit{freq} - x.\mathit{freq})(d_T(a) - d_T(x)) \\
&\geq 0
\end{aligned}
$$

διότι τα a.freq-x.freq & $d_T(a)$-$d_T(x)$ είναι μη αρνητικά

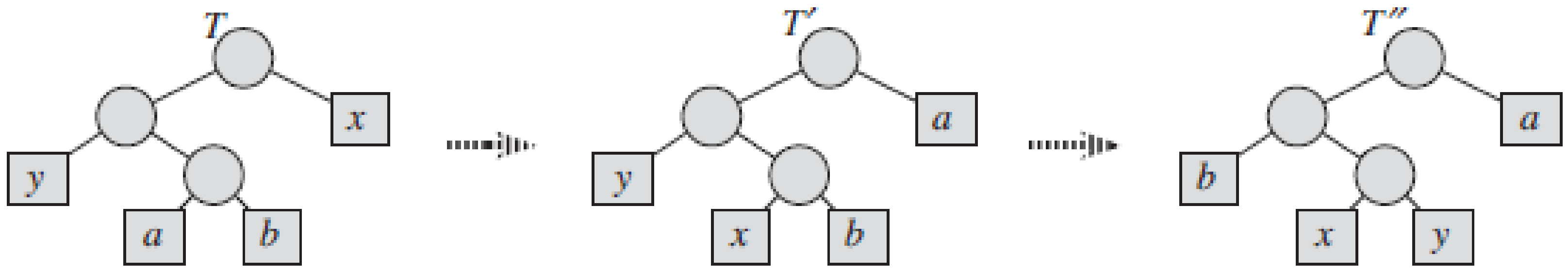
Το a.freq-x.freq είναι μη αρνητικό διότι το x έχει ελάχιστη συχνότητα και το $d_T(a)$-$d_T(x)$ είναι μη αρνητικό αφού το a είναι ένα φύλλο μέγιστου βάθους

Κώδικες Huffman (14/19)

Αν αντιμεταθέσουμε τα y, b, το κόστος δεν αυξάνει, οπότε το B(T’)-B(T’’) είναι μη αρνητικό

Συνεπώς, B(T’’)<=B(T) και αφού το T είναι το βέλτιστο έχουμε B(T)<=B(T’’) το οποίο μας δίνει ότι B(T’’)=B(T)

Οπότε το T’’ είναι βέλτιστο και τα x, y είναι αδέρφια



Κώδικες Huffman (15/19)

Lemma 2

Έστω C είναι ένα αλφάβητο με συχνότητα c.freq για κάθε χαρακτήρα c του αλφαβήτου. Έστω x,y δύο χαρακτήρες με την ελάχιστη συχνότητα. Έστω C’ ένα αλφάβητο χωρίς τα x,y και με ένα νέο χαρακτήρα z τέτοιο ώστε C’=C-{x,y}∪{z} και z.freq=x.freq+y.freq. Έστω T’ οποιοδήποτε δένδρο που αναπαριστά τον κώδικα του βέλτιστου προθέματος για το αλφάβητο C’. Τότε το δένδρο T το οποίο εξάγεται από το T’ αντικαθιστώντας το φύλο z με ένα εσωτερικό κόμβο που έχει παιδιά τα x, y, αναπαριστά ένα βέλτιστο κώδικα για το αλφάβητο C.

Κώδικες Huffman (16/19)

Απόδειξη

Για κάθε χαρακτήρα $c \in C - \{x, y\}$,

Έχουμε $d_T(c) = d_{T'}(c)$

Και έτσι $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$

Αφού $d_T(x) = d_T(y) = d_{T'}(z) + 1$

Έχουμε

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

Κώδικες Huffman (17/19)

Συμπεραίνουμε πως $B(T) = B(T') + x.freq + y.freq$

Ή $B(T') = B(T) - x.freq - y.freq$

Έστω τώρα ότι το T δεν αναπαριστά ένα βέλτιστο κώδικα

Υπάρχει ένα T’’ τέτοιο ώστε $B(T'') < B(T)$

Το T’’ έχει τα x, y ως αδέρφια

Έστω το T’’’ το δένδρο T’’ με το κοινό πατέρα των x, y που έχουν αντικατασταθεί από το z με συχνότητα z.freq=x.freq+y.freq

Κώδικες Huffman (18/19)

Τότε

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

Το οποίο δεν ισχύει αφού το T’ αναπαριστά ένα βέλτιστο κώδικα του C’

Έτσι το T είναι ο βέλτιστος κώδικας του C

Κώδικες Huffman (19/19)

Θεώρημα

Η διαδικασία HUFFMAN παράγει ένα βέλτιστο κώδικα

Απόδειξη

Εξάγεται εύκολα από τα δύο προηγούμενα λήματα

Γράφοι

Εισαγωγή (1/2)

Τα προβλήματα που εμπλέκουν γράφους είναι πολύ σημαντικά για διάφορους τομείς της Επιστήμης των ΗΥ

Ένας γράφος ουσιαστικά απεικονίζεται με το συμβολισμό G=(V,E) όπου V είναι οι κορυφές και E είναι οι ακμές που τις συνδέουν

Όταν χαρακτηρίζουμε το χρόνο εκτέλεσης / πολυπλοκότητα αλγορίθμων πάνω σε γράφους, μετράμε το μέγεθος της εισόδου σε σχέση με το πλήθος των κορυφών $|V|$ και το πλήθος των ακμών $|E|$

Άρα μελετούμε την επίδραση δύο παραμέτρων και όχι μιας

Παράδειγμα: $O(V\ E)$ ή $O(|V||E|)$

Εισαγωγή (2/2)

Δύο τρόποι αναπαράστασης

- **Συλλογή λιστών γειτνίασης** (collection of adjacency lists)
- **Πίνακας γειτνίασης** (adjacency matrix)

Οι λίστες γειτνίασης αποτελούν πιο συμπαγή παρουσίαση ιδιαίτερα των **αραιών (sparse)** γράφων

Οι πίνακες γειτνίασης είναι προτιμότεροι όταν οι γράφοι είναι **πυκνοί (dense)**

Αραιοί γράφοι: το $|E|$ είναι κατά πολύ μικρότερο του $|V|^2$

Πυκνοί γράφοι: το $|E|$ είναι κοντά στο $|V|^2$

Λίστες Γειτνίασης (1/2)

Για ένα γράφο G=(V,E), χρησιμοποιούμε ένα πίνακα |V| λιστών, μια για κάθε κόμβο / κορυφή του G

Για κάθε u ϵ V, η u θέση του πίνακα περιέχει μια **συνδεδεμένη λίστα** με τους κόμβους v για τους οποίους υπάρχει μια ακμή τέτοια ώστε (u,v) ϵ E

Αν ο G είναι **κατευθυνόμενος**, τότε το άθροισμα του μήκους όλων των λιστών γειτνίασης είναι ίσο με |E|

Αν ο G είναι **μη κατευθυνόμενος**, τότε το άθροισμα του μήκους όλων των λιστών γειτνίασης είναι ίσο με 2|E|

Λίστες Γειτνίασης (2/2)

Για όλους τους γράφους, η διατήρηση των λιστών γειτνίασης απαιτεί ποσότητα μνήμης ίση με **Θ(V+E)**

Οι λίστες γειτνίασης μπορούν εύκολα να χρησιμοποιηθούν ώστε να αποτυπώσουν **γράφους με βάρη** (αποθηκεύουμε απλά το βάρος)

- Κάθε ακμή έχει ένα βάρος το οποίο τυπικά αποδίδεται μέσω μιας συνάρτησης $w : E \to \mathbb{R}$

Μειονεκτήματα

- Δεν είναι πολύ γρήγορες στην αποτύπωση του αν υπάρχει μια ακμή που συνδέει δύο κόμβους (οι πίνακες γειτνίασης είναι πιο καλή δομή για την επίλυση του συγκεκριμένου προβλήματος)

Πίνακες Γειτνίασης (1/2)

Υποθέτουμε πως οι κόμβοι αριθμούνται με 1,2,…,|V|

Ο πίνακας γειτνίασης έχει μέγεθος **|V|X|V|**

Έστω $A=(a_{ij})$ ο πίνακας γειτνίασης. Ισχύει ότι:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Ο πίνακας γειτνίασης απαιτεί επιπλέον μνήμη της τάξης του **$\Theta(V^2)$** (προσέξτε ότι δεν επηρεάζεται από το πλήθος των ακμών)

Πρόκειται για ένα συμμετρικό πίνακα

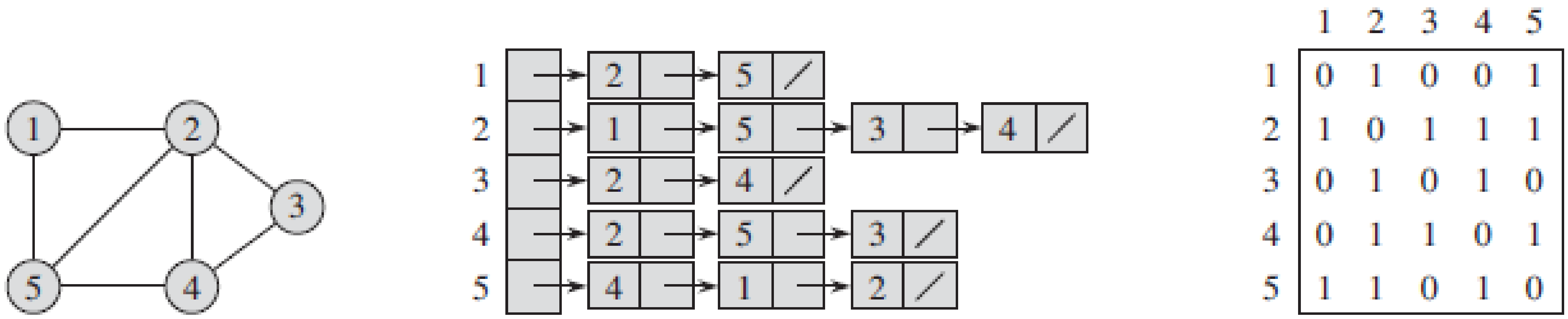
Πίνακες Γειτνίασης (2/2)

Για την αναπαράσταση ενός γράφου με βάρη, απλά αποθηκεύουμε το βάρος και όχι 0 ή 1

Σε μη κατευθυνόμενους γράφους για να ‘γλιτώσουμε’ χώρο επιπλέον μνήμης, αποθηκεύουμε μόνο τα στοιχεία πάνω από τη διαγώνιο (αφού είναι ίδια με τα στοιχεία κάτω από τη διαγώνιο)

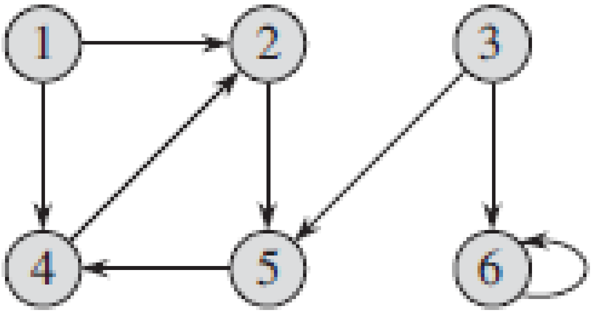
Πρόκειται για πιο απλή δομή σε σχέση με τις λίστες γειτνίασης

Παράδειγμα

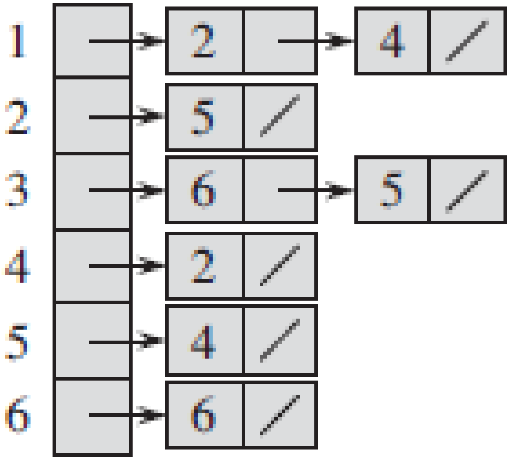


(a) (b) (c)

Παράδειγμα



(a)



(b)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)

Breadth-First Search (1/18)

Η **αναζήτηση κατά πλάτος** (**breadth first search - BFS**) είναι ο πιο εύκολος αλγόριθμος αναζήτησης σε γράφους

Οι αλγόριθμοι του **Prim** & **Dijkstra** χρησιμοποιούν ιδέες όμοιες με του BFS

Δοσμένων ενός γράφου G και ενός κόμβου έναρξης s, ο BFS προσπαθεί να ανακαλύψει κάθε κόμβο που μπορεί να προσπελαστεί από τον s

Υπολογίζει την απόσταση από τον s προς κάθε άλλο προσπελάσιμο κόμβο

Breadth-First Search (2/18)

Επίσης, παράγει ένα δένδρο (breadth-first tree) με ρίζα τον s

Για κάθε κόμβο v που μπορεί να προσπελαστεί από τον s το **απλούστερο μονοπάτι** στο δένδρο από το s στον v αντιστοιχεί στο **συντομότερο μονοπάτι** από τον s στον v πάνω στο γράφο G

Ο αλγόριθμος λειτουργεί είτε για κατευθυνόμενους ή μη κατευθυνόμενους γράφους

Ο αλγόριθμος ανακαλύπτει τους κόμβους που μπορούν να προσπελαστούν από τον s σε απόσταση k πριν ανακαλύψει τους κόμβους που βρίσκονται σε απόσταση k+1

Breadth-First Search (3/18)

Ο αλγόριθμος ‘χρωματίζει’ τους κόμβους με τα εξής χρώματα: **white, gray, black**

Όλοι οι κόμβοι ξεκινούν με λευκό χρώμα και στη συνέχεια μπορεί να γίνουν είτε γκρι είτε μαύρο

Όταν ένας κόμβος ανακαλύπτεται για πρώτη φορά τότε χάνει το λευκό χρώμα

Οι γκρι και μαύροι κόμβοι έχουν ανακαλυφθεί από τον αλγόριθμο αλλά ο διαφορετικός χρωματισμός εξασφαλίζει ότι η αναζήτηση γίνεται κατά πλάτος

Breadth-First Search (4/18)

Αν (u,v) ϵ E και ο u είναι ένας **μαύρος** κόμβος, τότε ο u είναι είτε **γκρι** ή **μαύρος**

Όλοι οι κόμβοι που γειτνιάζουν με μαύρους κόμβους έχουν ανακαλυφθεί από τον αλγόριθμο

Οι γκρι κόμβοι μπορεί να γειτνιάζουν με κάποιους λευκούς και αναπαριστούν το όριο ανάμεσα σε κόμβους που έχουμε ανακαλύψει και σε αυτούς που δεν έχουμε δει ακόμα

Breadth-First Search (5/18)

Το δένδρο αναζήτησης κατασκευάζεται με ρίζα τον κόμβο s

Κάθε φορά που ένας λευκός κόμβος v ανακαλύπτεται, μέσω ενός κόμβου u, ο v και η ακμή (u,v) εισάγονται στο δένδρο

O u είναι ο ‘πατέρας’ (predecessor or parent) του v στο δένδρο

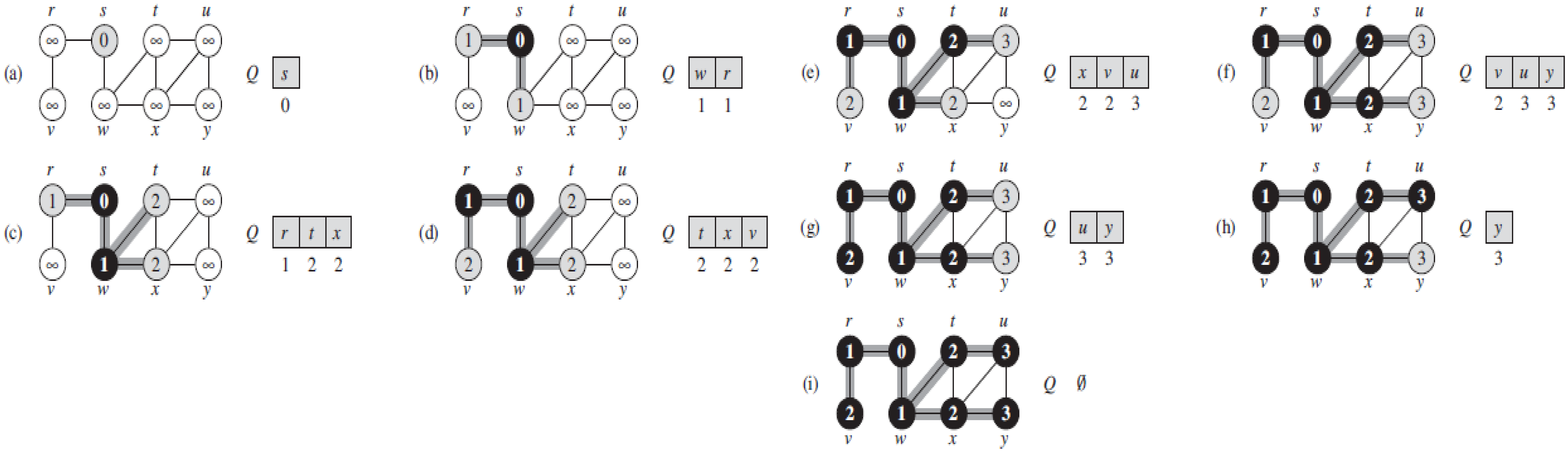
Αφού ο κάθε κόμβος ανακαλύπτεται μόνο μια φορά, θα έχει μόνο ένα πατέρα στο δένδρο

Breadth-First Search (6/18)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search (7/18)

Παράδειγμα



Breadth-First Search (8/18)

Ανάλυση

- Κάθε κόμβος μπαίνει στην ουρά μόνο μια φορά
- Ο χρόνος εισαγωγής / εξαγωγής στην ουρά είναι O(1)
- Συνολικός χρόνος για τις πράξεις που αφορούν στην ουρά είναι O(V)
- Κάθε φορά που εξάγεται ένας κόμβος από την ουρά, τότε οι γείτονες προσπελαύνονται μόνο μια φορά
- Αφού το άθροισμα τους μήκους όλων των λιστών γειτνίασης είναι Θ(E), ο χρόνος για την προσπέλαση όλων των λιστών είναι O(E)
- Συνεπώς, ο συνολικός χρόνος του BFS είναι O(V+E)
- Γραμμικός χρόνος εκτέλεσης

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search (9/18)

Ας υποθέσουμε ότι η μικρότερη απόσταση (shortest-path distance) από τη ρίζα του δένδρου μέχρι ένα κόμβο v είναι **δ(s,v)**

Το συντομότερο μονοπάτι (shortest path) είναι το μικρότερο πλήθος των ακμών που προσπελαύνουμε για να φτάσουμε από το s στο v

Αν δεν υπάρχει κάποιο μονοπάτι που να συνδέει τους κόμβους τότε $\delta(s,v) = \infty$

Breadth-First Search (10/18)

Lemma 1

Αν G=(V,E) είναι ένας γράφος (κατευθυνόμενος ή μη) και s ϵ V είναι ένας κόμβος. Τότε για κάθε ακμή (u,v) ϵ E ισχύει δ(s,v) <= δ(s,u) +1

Απόδειξη

Αν ο u μπορεί να προσπελαστεί μέσω του s τότε το ίδιο ισχύει και για τον v. Τότε το συντομότερο μονοπάτι από τον s στον v δεν μπορεί να είναι μικρότερο από το μονοπάτι που συνδέει τον s με τον u (λόγω της ακμής (u,v)). Αν ο v δεν μπορεί να προσπελαστεί από τον s, τότε **δ(s,v) = ∞**. Και στις δύο περιπτώσεις η αρχική ανίσωση ισχύει.

Breadth-First Search (11/18)

Lemma 2

Αν G=(V,E) είναι ένας γράφος (κατευθυνόμενος ή μη) και ο BFS έχει εκτελεστεί με έναρξη ένα κόμβο s ϵ V. Στον τερματισμό του αλγορίθμου, για κάθε v ϵ V, ισχύει v.d >= δ(s,v)

Απόδειξη

Υιοθετούμε επαγωγή. Εστιάζουμε στην εξαγωγή στοιχείων από την ουρά. Η ανίσωση ισχύει για την εξαγωγή στοιχείων από την ουρά αφού για το πρώτο στοιχείο s έχουμε **s.d=0=δ(s,s)** και για κάθε v ϵ V-{s}, **v.d = ∞ >= δ(s,v)**.

Ας εστιάσουμε τώρα σε κάποιο κόμβο v που έχει προσπελαστεί μέσω του u. Σε αυτή την περίπτωση ισχύει **ότι u.d>=δ(s,u)** οπότε **v.d=u.d+1>=δ(s,u)+1>= δ(s,v)**. Η τιμή v.d δεν αλλάζει στη συνέχεια αφού αν βγει από την ουρά, τότε δεν ξαναβγαίνει.

Breadth-First Search (12/18)

Lemma 3

Ας υποθέσουμε ότι σε κάποια στιγμή εκτέλεσης του BFS στην ουρά υπάρχουν τα στοιχεία (v_1, v_2, …, v_r) με το v_1 να είναι στην κορυφή της ουράς. Τότε, v_r.d<=v_1.d+1 και v_i.d<=v_{i+1}.d για i=1,2,..,r-1

Breadth-First Search (13/18)

Proof of Lemma 3

Υιοθετούμε επαγωγή. Αν το v_1 βγει από την ουρά το v_2 γίνεται η νέα κορυφή της ουράς. Μέσω της επαγωγής έχουμε: $\mathbf{v_1.d<=v_2.d}$ και $\mathbf{v_r.d<=v_1.d+1<=v_2.d}$. Στη συνέχεια το Λήμμα ισχύει για το v_2.

Όταν βάζουμε ένα στοιχείο v τότε αυτό γίνεται το v_{r+1}. Τότε έχουμε ήδη βγάλει το u για το οποίο εξετάζουμε τη λίστα γειτνίασης. Με βάση την επαγωγή, η νέα κορυφή v_1 θα έχει $\mathbf{v_1.d>=u.d}$. Οπότε $\mathbf{v_{r+1}.d=v.d=u.d+1<=v_1.d+1}$. Από την επαγωγική υπόθεση έχουμε πως $\mathbf{v_r.d<=u.d+1}$, συνεπώς $\mathbf{v_r.d<=u.d+1=v_r.d=v_{r+1}.d}$. Οπότε το λήμμα ισχύει όταν το στοιχείο βγαίνει από την ουρά.

Breadth-First Search (14/18)

Πρόταση

Για δύο κόμβους v_i, v_j με τον v_i να έχει εισαχθεί στην ουρά πριν τον v_j ισχύει $v_i.d<=v_j.d$ όταν το v_j βγαίνει από την ουρά.

Θεώρημα

Έστω G=(V,E) είναι ένας γράφος (κατευθυνόμενος ή μη) και ο BFS έχει εκτελεστεί με έναρξη ένα κόμβο s ϵ V. Κατά τη διάρκεια εκτέλεσης, ο BFS ανακαλύπτει κάθε κόμβο v ϵ V που μπορεί να προσπελαστεί από τον s και στον τερματισμό ισχύει v.d=δ(s,v) για κάθε v ϵ V. Για κάθε v≠s, ένα από τα συντομότερα μονοπάτια από το s στο v είναι ένα συντομότερο μονοπάτι από το s στο v.π ακολουθούμενο από την ακμή (v.π,v)

Breadth-First Search (15/18)

Απόδειξη

Ας υποθέσουμε ότι κάποιοι κόμβοι δεν έχουν απόσταση ίση με το ελάχιστο μονοπάτι. Έστω v ο κόμβος με το δ(s,v) που παίρνει αυτή την τιμή. Προφανώς v≠s. Από το Λήμμα 2 έχουμε **v.d>= δ(s,v)** και **v.d> δ(s,v)**. Ο v πρέπει να προσπελαύνεται από τον s, αν όχι τότε δ(s,v)=∞>= v.d. Αν u είναι ο αμέσως προηγούμενος κόμβος στο συντομότερο μονοπάτι με **δ(s,v)= δ(s,u)+1**. Αφού **δ(s,u)<δ(s,v)**, έχουμε **u.d= δ(s,u)**.

Άρα $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$

Breadth-First Search (16/18)

Απόδειξη (συνέχεια)

Ας υποθέσουμε τώρα πως ο αλγόριθμος εξάγει τον u. Σε αυτή τη στιγμή ο v είναι λευκός ή γκρι ή μαύρος. Σε κάθε περίπτωση η εξίσωση $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$ δεν ισχύει. Αν ο v είναι **λευκός** τότε θα έχουμε v.d=u.d+1 (αντίθετα με την εξίσωση). Αν ο v είναι **μαύρος** έχει ήδη βγει από την ουρά και με βάση την προηγούμενη Πρόταση έχουμε **v.d<=u.d** (αντίθετα με την εξίσωση). Αν είναι **γκρι**, τότε αυτό έχει γίνει διότι έχει εξαχθεί από την ουρά ένας άλλος κόμβος w πριν από τον u οπότε **v.d=w.d+1**. Από την Πρόταση 1 έχουμε όμως πως **w.d<=u.d**, οπότε **v.d=w.d+1<=u.d+1** (αντίθετα με την εξίσωση).

Breadth-First Search (17/18)

Απόδειξη (συνέχεια)

Από τα προηγούμενα συμπεραίνουμε πως **v.d=δ(s,v)** για κάθε v ϵ V. Όλοι οι κόμβοι v που προσπελαύνονται μέσω του s μπορούν ν ανακαλυφθούν αλλιώς θα έχουν **v.d=∞> δ(s,v)**. Για να συμπεράνουμε τελικά το θεώρημα, παρατηρούμε πως αν **v.π=u** τότε **v.d=u.d+1**. Τελικά, μπορούμε να έχουμε το συντομότερο μονοπάτι από το s στο v παίρνοντας το συντομότερο μονοπάτι από το s στο v.π και διασχίζοντας την ακμή (v.π,v).

Breadth-First Search (18/18)

Η εκτύπωση του δένδρου που φτιάχνει ο BFS τυπώνεται με τον ακόλουθο αλγόριθμο.

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
    print v
```

Demo

http://visualgo.net/en/dfsbfs.html

Depth-First Search (1/13)

Ο αλγόριθμος **αναζήτησης κατά βάθος (depth first search – DFS)** αναζητά τα στοιχεία στο γράφο προχωρώντας όσο πιο ‘κάτω’ στο γράφο μπορεί

Ακολουθεί ακμές που προκύπτουν από τους πιο πρόσφατα αναζητούμενους κόμβους

Όταν όλες οι ακμές ενός κόμβου έχουν ελεγχθεί τότε οπισθοχωρεί και συνεχίζει από τον επόμενο κόμβο

Αυτή η διαδικασία συνεχίζεται μέχρι να ανακαλυφθούν όλοι οι κόμβοι που μπορούν να προσπελαστούν από τον s

Depth-First Search (2/13)

Όταν ο αλγόριθμος ανακαλύπτει ένα κόμβο v μέσω του u (μέσω της λίστας γειτνίασης) καταγράφει το v.π=u

Αντίθετα με τον BFS όπου ο υπογράφος που ορίζεται με τον πατρικό κόμβο είναι ένα δένδρο, στον DFS ο υπογράφο που ορίζεται από τον πατρικό κόμβο μπορεί να αποτελείται από πολλά δένδρα

Ο υπογράφος του πατρικού κόμβου ορίζει ένα ‘δάσος’ (depth first forest) αποτελούμενο από πολλά DFS trees

Ο αλγόριθμος υιοθετεί επίσης χρώματα: **λευκό**, **γκρι** όταν έχει ανακαλυφθεί και **μαύρο** όταν η επεξεργασία του κόμβου έχει τελειώσει

Depth-First Search (3/13)

Ο αλγόριθμος χρησιμοποιεί **χρονοσφραγίδες** (timestamps) για κάθε κόμβο

Κάθε κόμβος έχει δύο χρονοσφραγίδες

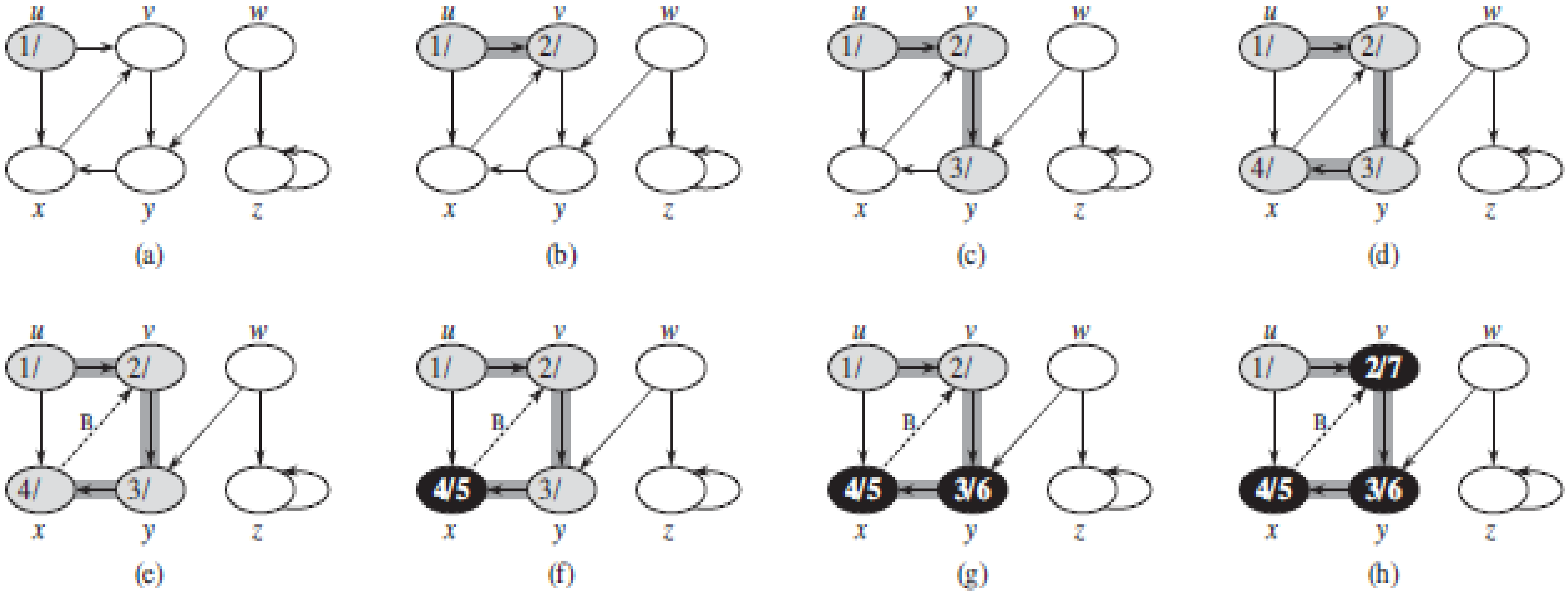
- Η v.d αποθηκεύει πότε ο v έχει ανακαλυφθεί (έχει γίνει γκρι)
- Η v.f αποθηκεύει πότε ο v έχει ολοκληρώσει (έχει τελειώσει η επεξεργασία των λιστών γειτνίασης και έχει γίνει μαύρος)
- Πρόκειται για ακέραιους στο διάστημα [1,2|V|]
- Επίσης, ισχύει u.d<u.f
- Ένας κόμβος u είναι λευκός πριν το u.d, γκρι ανάμεσα στα u.d, u.f και μαύρος μετά το u.f

Depth-First Search (4/13)

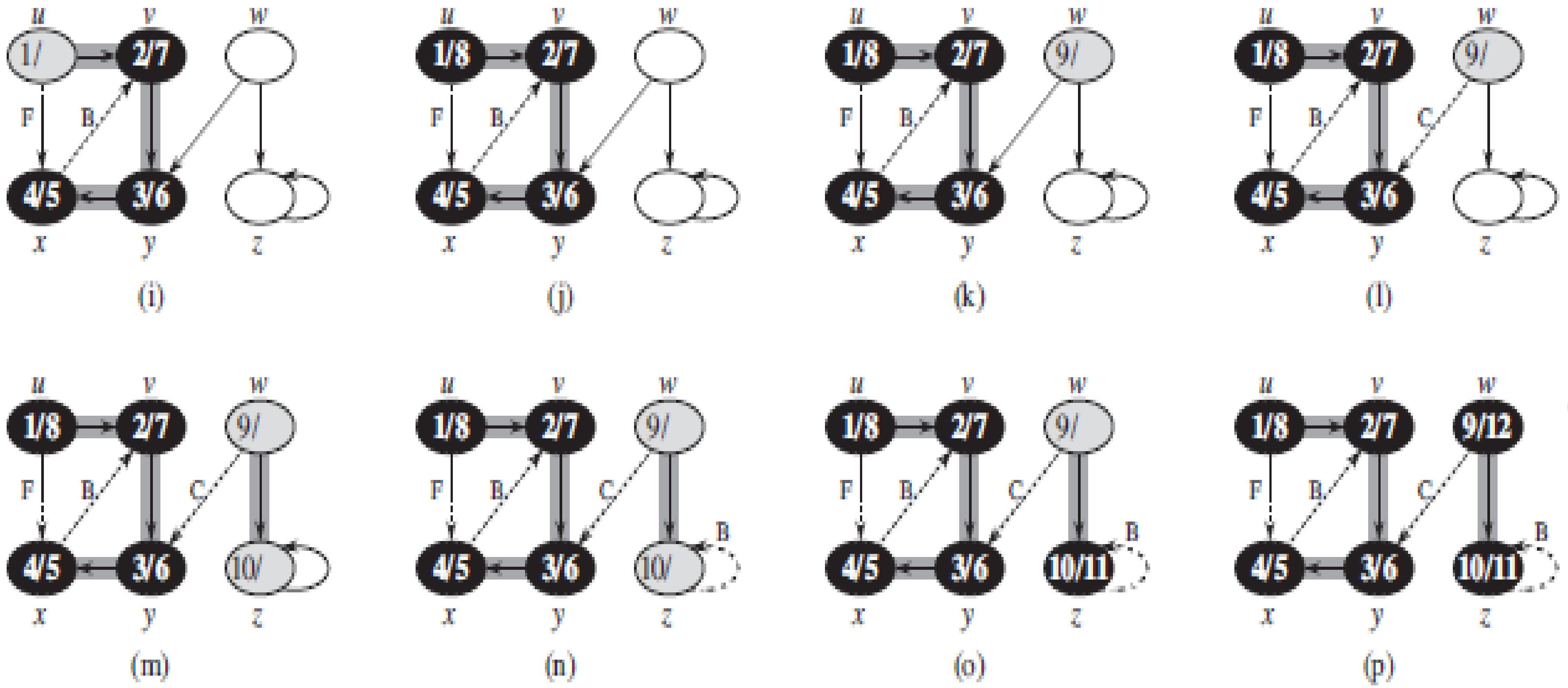
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

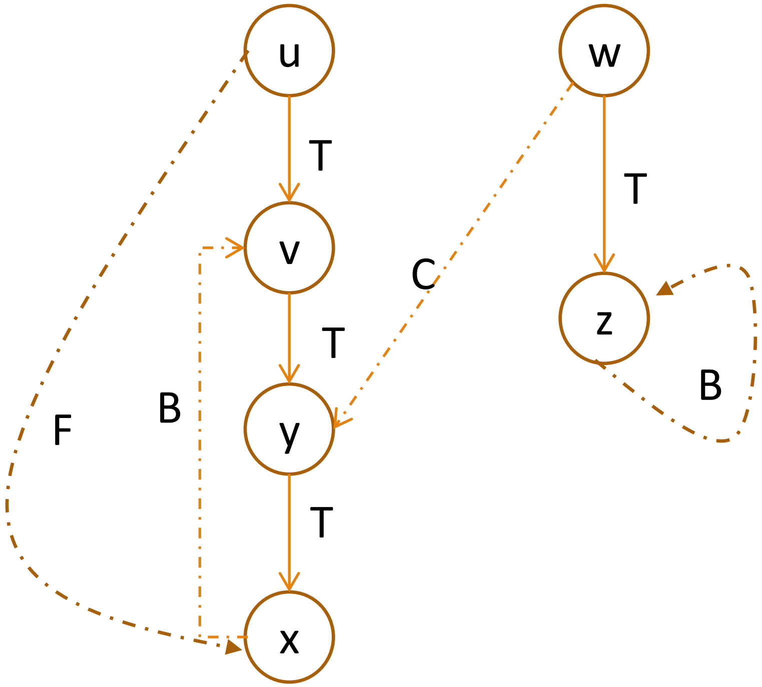
```
DFS-VISIT(G, u)
time = time + 1 // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u] // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK  // blacken u; it is finished
time = time + 1
u.f = time
```

Depth-First Search (5/13)



Depth-First Search (6/13)





Depth-First Search (7/13)

Ανάλυση

- Ο αρχικός ‘χρωματισμός’ των κόμβων απαιτεί Θ(V)
- Η visit καλείται μόνο μια φορά για κάθε κόμβο
- Η εκτέλεση της visit γίνεται |Adj[v]| φορές
- Αφού το άθροισμα όλων των |Adj[v]| για κάθε v ϵ V είναι Θ(E), το κόστος όλων των κλήσεων της visit είναι Θ(E)
- Ο συνολικός χρόνος του αλγορίθμου είναι Θ(V+E)

Depth-First Search (8/13)

Parenthesis Theorem

Στον DFS που εφαρμόζεται πάνω σε ένα γράφο G=(V,E), για οποιουσδήποτε δύο κόμβους u, v, ισχύει ακριβώς μια από τις ακόλουθες περιπτώσεις:

- ***Το διάστημα [v.d,v.f] περιλαμβάνεται στο [u.d,u.f] και ο v είναι απόγονος του u***
- ***Τα διαστήματα [u.d,u.f] & [v.d,v.f] είναι διαφορετικά και κανείς από τους u, v είναι απόγονος του άλλου***
- ***Το διάστημα [u.d,u.f] περιλαμβάνεται στο [v.d,v.f] και ο u είναι απόγονος του v***

Depth-First Search (9/13)

Απόδειξη

Ξεκινάμε από την περίπτωση όπου u.d<v.d. Υπάρχουν 2 υπο-περιπτώσεις: v.d<u.f ή όχι.

Αν **v.d<u.f**, τότε ο v ανακαλύφθηκε όταν ο u ήταν γκρι οπότε ο v είναι απόγονος του u. Αφού ο v ανακαλύφθηκε πιο πρόσφατα όλες οι ακμές του έχουν προσπελαστεί και ο v είναι μαύρος πριν από τον u. Τότε, το [v.d,v.f] περιλαμβάνεται εξ’ ολοκλήρου στο [u.d,u.f].

Depth-First Search (10/13)

Απόδειξη (συνέχεια)

Στη δεύτερη υπο-περίπτωση, **u.f<v.d** και επειδή **u.d<u.f** έχουμε **u.d<u.f<v.d<v.f** οπότε τα διαστήματα **[u.d,u.f], [v.d,v.f]** είναι διαφορετικά και disjoint μεταξύ τους. Αφού είναι disjoint κανένας κόμβος από τους δύο έχει ανακαλυφθεί όταν ο άλλος ήταν γκρι, συνεπώς, κανένας κόμβος δεν είναι απόγονος του άλλου.

Η τρίτη περίπτωση είναι παραπλήσια με τα προηγούμενα, απλά **αντιστρέφουμε τους ρόλους** για τους κόμβους u, v.

Depth-First Search (11/13)

Πρόταση

Ένας κόμβος v είναι απόγονος ενός κόμβου u όταν και μόνο όταν ισχύει $u.d<v.d<v.f<u.f$

Θεώρημα

Στον DFS ο κόμβος v είναι ένας απόγονος του κόμβου u όταν και μόνο όταν σε χρόνο u.d όπου ανακαλύπτεται ο u υπάρχει ένα μονοπάτι από τον u στον v που αποτελείται αποκλειστικά από λευκούς κόμβους.

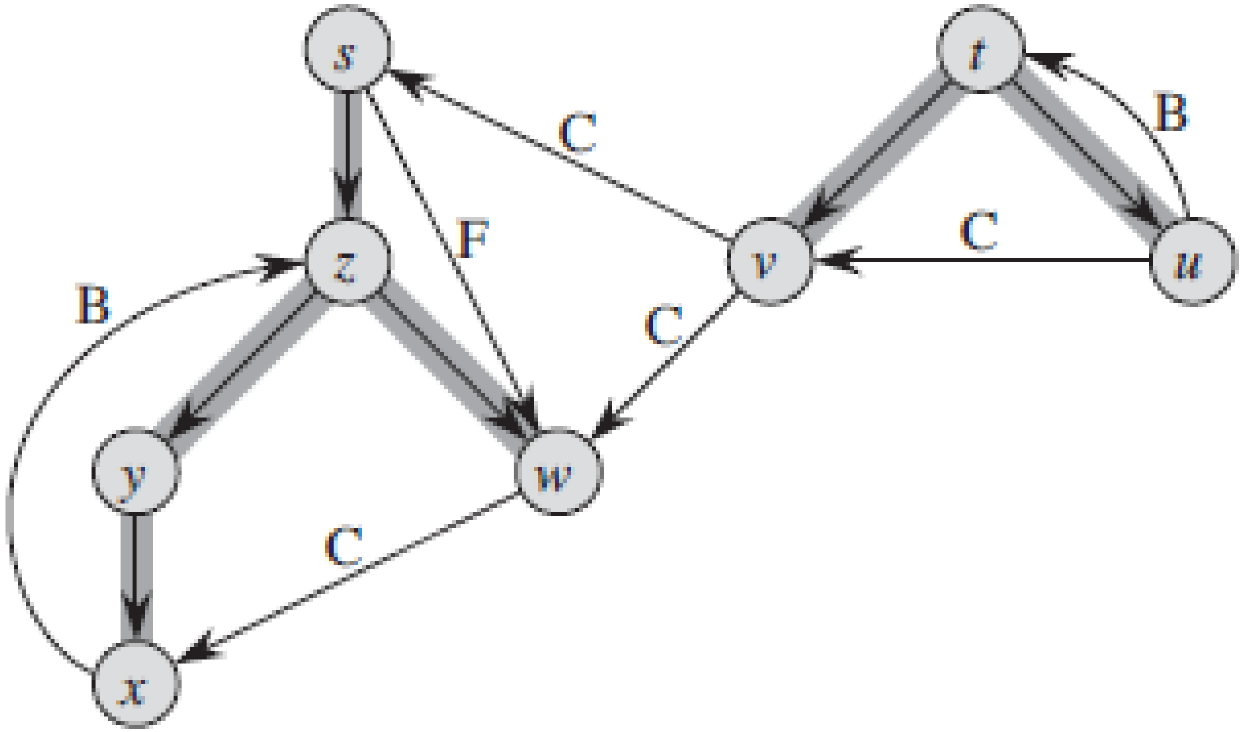
Depth-First Search (12/13)

Κατάταξη των ακμών

- Οι ακμές δίνουν σημαντικές πληροφορίες για ένα γράφο: ένας κατευθυνόμενος γράφος είναι **μη κυκλικός / ακυκλικός** όταν και μόνο όταν στον DFS δεν ανακαλύπτονται ακμές οπισθοχώρησης (back).
- Οι ακμές κατατάσσονται σε:
  - **Tree**: Η ακμή (u,v) είναι δενδρική ακμή όταν ο v έχει πρώτος ανακαλυφθεί
  - **Back**: Η (u,v) είναι ακμή οπισθοχώρησης όταν συνδέει ένα κόμβο με τον πρόγονό του
  - **Forward**: Η (u,v) είναι ακμή προώθησης όταν συνδέει ένα κόμβο με ένα απόγονό του
  - **Cross**: Η (u,v) είναι ακμή διασταύρωσης όταν συνδέει κόμβους στο ίδιο υπο-δένδρο χωρίς οι κόμβοι να έχουν σχέση πρόγονου – απόγονου ή συνδέουν κόμβους σε διαφορετικά υπο-δένδρα

Depth-First Search (13/13)

Παράδειγμα



Demo

http://visualgo.net/en/dfsbfs.html

Minimum Spanning Trees (1/6)

Ορισμένες φορές επιθυμούμε πάνω σε ένα γράφο να βρούμε ένα **μη κυκλικό υποσύνολο ακμών** που να συνδέουν όλους τους κόμβους

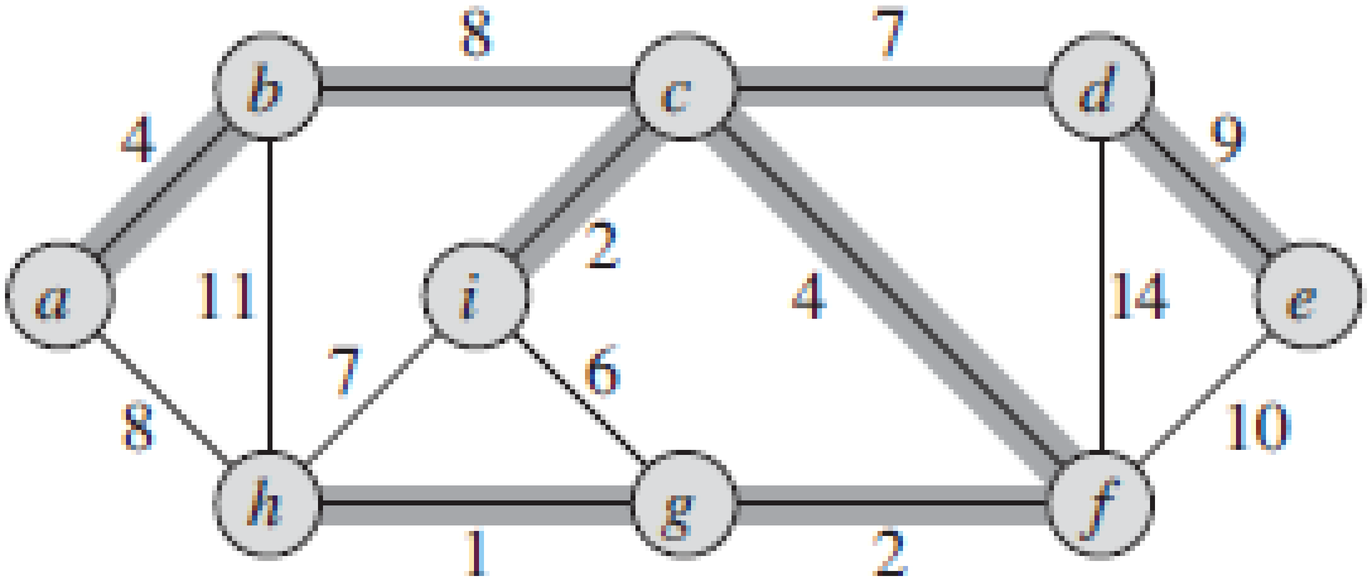
Το άθροισμα των βαρών θέλουμε να είναι το ελάχιστο

Δημιουργείται ένα δένδρο

Το δένδρο ονομάζεται το **δένδρο επικάλυψης ελάχιστου κόστους (minimum spanning tree - MST)**

Minimum Spanning Trees (2/6)

Παράδειγμα



Minimum Spanning Trees (3/6)

Έστω γράφος G=(V,E) και μια συνάρτηση βάρους $w:E \rightarrow R$

Πρόβλημα

- Εύρεση του MST

Τεχνικές

- Άπληστη
  - Το MST μεγαλώνει κατά μια ακμή κάθε φορά
  - Η ακμή προστίθεται σε ένα σύνολο ακμών A που είναι επίσης ένα MST

Minimum Spanning Trees (4/6)

Αλγόριθμος

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
      find an edge (u, v) that is safe for A
      A = A ∪ {(u, v)}
return A
```

Πρόβλημα: πως θα βρούμε μια ακμή που είναι ασφαλές να την τοποθετήσουμε στο MST?

Minimum Spanning Trees (5/6)

Λύση

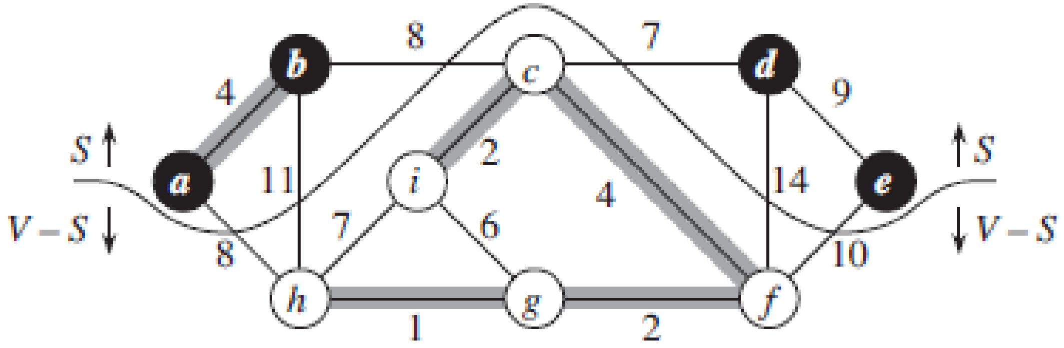
- Για ένα οποιοδήποτε διαχωρισμό των κόμβων του γράφου (S,V-S) λέμε πως μια ακμή (u,v) διασχίζει τα δύο τμήματα αν ένα από τα ακραία σημεία – κόμβοι είναι στο S και το άλλο είναι στο V-S
- Μια light ακμή, είναι η ακμή που συνδέει τα δύο τμήματα του γράφου και έχει το μικρότερο βάρος από όλες που επίσης συνδέουν τα δύο τμήματα

Θεώρημα

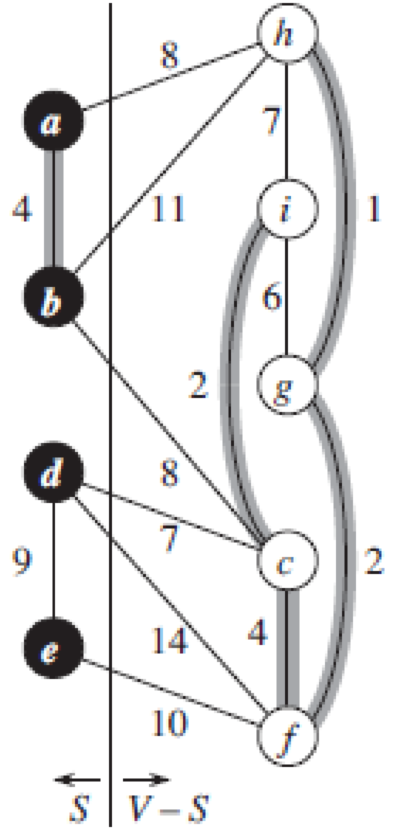
Αν A είναι το τρέχον MST για ένα γράφο G και (S,V-S) είναι ένας οποιοσδήποτε διαχωρισμός του γράφου, η light ακμή (u,v) που συνδέει τα S, V-S είναι ασφαλής για να περιληφθεί στο A.

Minimum Spanning Trees (6/6)

Παράδειγμα



(a)



(b)

Kruskal’s Algorithm (1/4)

- Ο αλγόριθμος βρίσκει κάθε φορά μια ασφαλή ακμή για να την τοποθετήσει στο MST
- Ο αλγόριθμος αρχικά δημιουργεί |V| δένδρα που περιέχουν μόνο ένα κόμβο
- Ταξινομεί τις ακμές σε αύξουσα σειρά βάρους

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

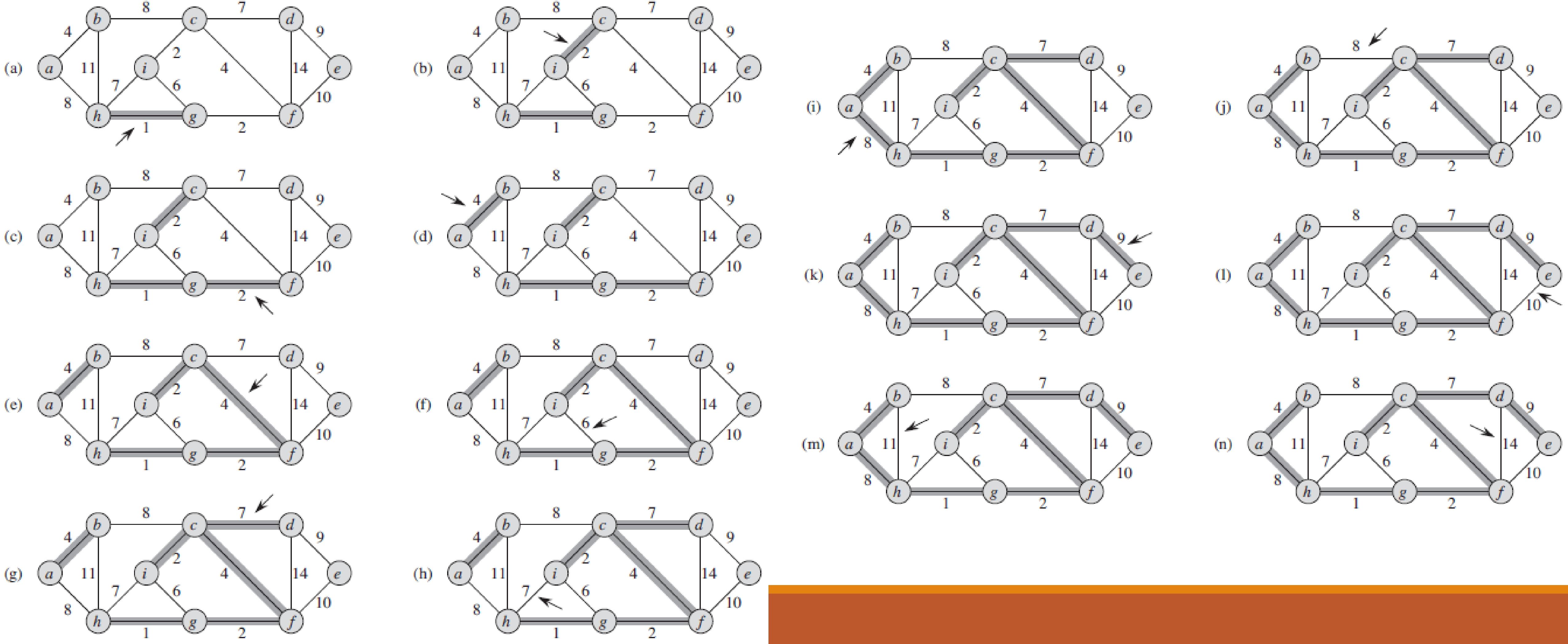
Kruskal’s Algorithm (2/4)

Ελέγχει για κάθε ακμή αν τα σημεία αρχής/τέλους ανήκουν σε διαφορετικά δένδρα

Αν ανήκουν σε διαφορετικά τότε η ακμή μπαίνει στο MST χωρίς να δημιουργεί κύκλο και οι δύο κόμβοι ενώνονται στο ίδιο δένδρο

```
MST-KRUSKAL(G, w)

A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

Kruskal's Algorithm (3/4)



Kruskal’s Algorithm (4/4)

Ο χρόνος εκτέλεσης εξαρτάται από τη μέθοδο που υιοθετούμε για τον υπολογισμό των disjoint sets

Με τον πιο γρήγορο διαθέσιμο αλγόριθμο, ο Kruskal αλγόριθμος απαιτεί O(E logE)

Αν παρατηρήσουμε πως $|E|<|V|^2$, τότε $\log|E| = O(\log V)$

Άρα η πολυπλοκότητα είναι O(E logV)

Demo

https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

Prim’s Algorithm (1/4)

Ενεργεί περίπου όπως ένας αλγόριθμος που βρίσκει το ελάχιστο μονοπάτι

Η βασική ιδιότητα του αλγορίθμου είναι ότι πάντα οι ακμές που μπαίνουν στο σύνολο A σχηματίζουν ένα δένδρο

Σε κάθε βήμα, ο αλγόριθμος προσθέτει στο A μια light ακμή που συνδέει το ήδη υπάρχον δένδρο του A με ένα απομονωμένο κόμβο

Χρειαζόμαστε ένα γρήγορο τρόπο για να βρούμε το ποια ακμή θα μπει στο A

Όλοι οι κόμβοι που δεν βρίσκονται στο MST αποθηκεύονται σε ουρά ελάχιστης προτεραιότητας

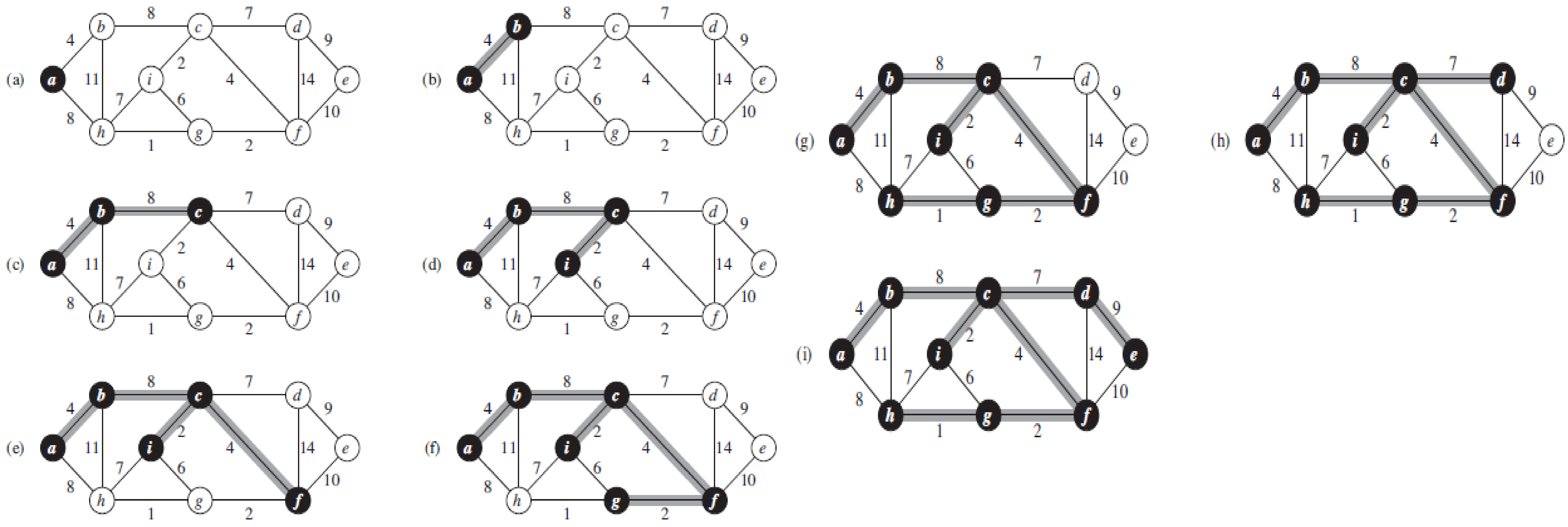
Prim’s Algorithm (2/4)

Το v.key είναι το ελάχιστο βάρος οποιασδήποτε ακμής που συνδέει ένα κόμβο στο δένδρο

Το v.π υποδηλώνει τον πατρικό κόμβο του v στο δένδρο

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Prim’s Algorithm (3/4)



Prim’s Algorithm (4/4)

Ο χρόνος εκτέλεσης εξαρτάται από το πώς υλοποιούμε την ουρά

Μπορούμε να υιοθετήσουμε μια μέθοδο με χρόνο εκτέλεσης O(V)

Η εξαγωγή απαιτεί O(V logV)

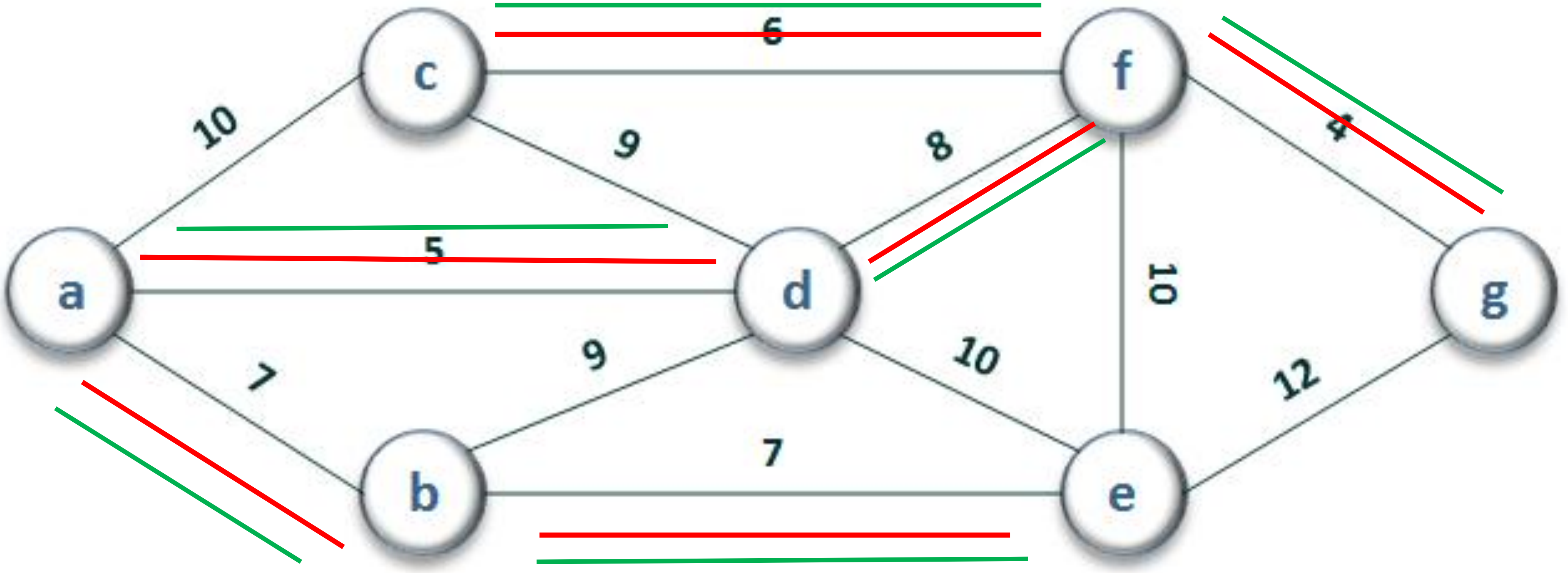
Συνολική πολυπλοκότητα O(E logV)

Demo

https://www.cs.usfca.edu/~galles/visualization/Prim.html

Άσκηση

Να κατασκευαστεί το MST επόμενο γράφο με χρήση των αλγορίθμων Kruskal (κόκκινο) & Prim (πράσινο)



Shortest Paths (1/7)

Δοσμένου ενός γράφου με βάρη αναζητούμε το **συντομότερο μονοπάτι (shortest path)** ανάμεσα σε δύο κόμβους

Γενικά ισχύει

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

και

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Το συντομότερο μονοπάτι είναι οποιοδήποτε μονοπάτι για το οποίο ισχύει $w(p) = \delta(u,v)$

Shortest Paths (2/7)

Τα βάρη μπορεί να συμβολίζουν κόστος, απόσταση ή οποιαδήποτε άλλη μετρική ανάλογα με το application domain

Εστιάζουμε στα **single source shortest paths**

Υπάρχουν 3 εκδόσεις του προβλήματος
- **Single destination shortest path**
- **Single pair shortest path**
- **All pairs shortest path**

Shortest Paths (3/7)

Οι αλγόριθμοι εύρεσης συντομότερων μονοπατιών βασίζονται στην ιδιότητα ενός συντομότερου μονοπατιού ότι περιλαμβάνει άλλα συντομότερα μονοπάτια

Ορισμένες φορές πρέπει να υπολογίσουμε και τους κόμβους που συμμετέχουν στο συντομότερο μονοπάτι

Για κάθε κόμβο που ανήκει στο συντομότερο μονοπάτι υιοθετούμε την ιδιότητα v.π που αποθηκεύει τον προηγούμενο κόμβο

Οι αλγόριθμοι υιοθετούν την τεχνική του **relaxation**

Για κάθε κόμβο v κρατάμε την ιδιότητα v.d που αποτυπώνει ένα **πάνω όριο βάρους του συντομότερου μονοπατιού από τον κόμβο s στον v**

Shortest Paths (4/7)

Η ιδιότητα v.d ονομάζεται **shortest path estimate**

Η αρχικοποίηση των estimates γίνεται ως εξής:

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi =$ NIL
$s.d = 0$

Η τεχνική relaxation περιλαμβάνει τον έλεγχο με βάση μια ακμή (u,v) αν μπορούμε να επεκτείνουμε το συντομότερο μονοπάτι προς τον κόμβο v μέσω του u

Εφόσον αυτό είναι εφικτό, τότε ενημερώνουμε τις ιδιότητες v.d και v.π

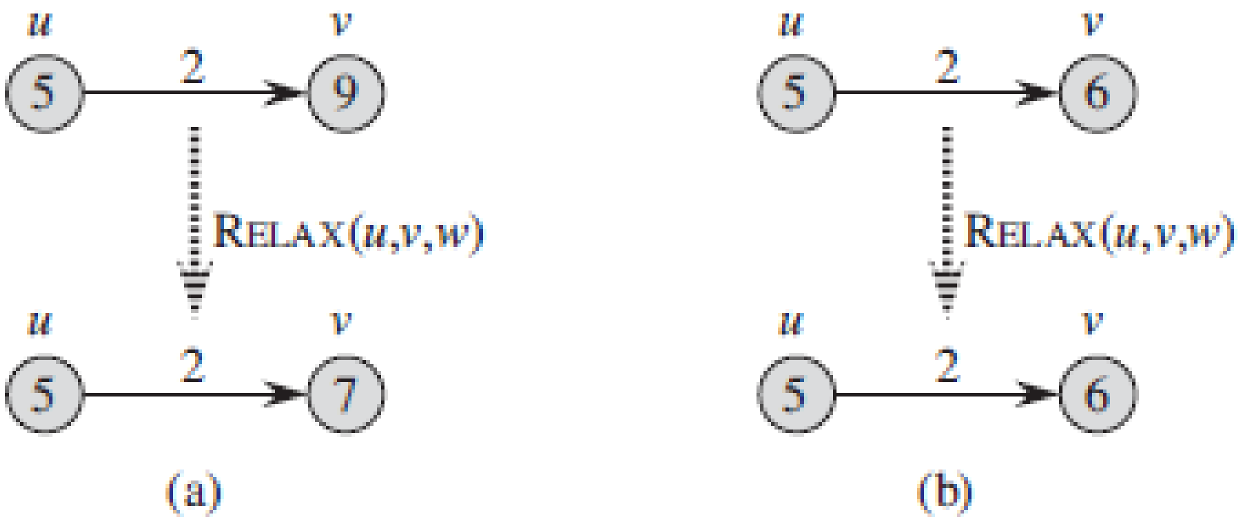
Shortest Paths (5/7)

Κάποιο βήμα του relaxation μπορεί να μειώσει το estimate ενός κόμβου

Αλγόριθμος

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$



Shortest Paths (6/7)

Ιδιότητες των συντομότερων μονοπατιών

Triangle inequality

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property

We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property

If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Shortest Paths (7/7)

Ιδιότητες των συντομότερων μονοπατιών (συνέχεια)

Convergence property

If $s \leadsto u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Bellman-Ford Algorithm (1/10)

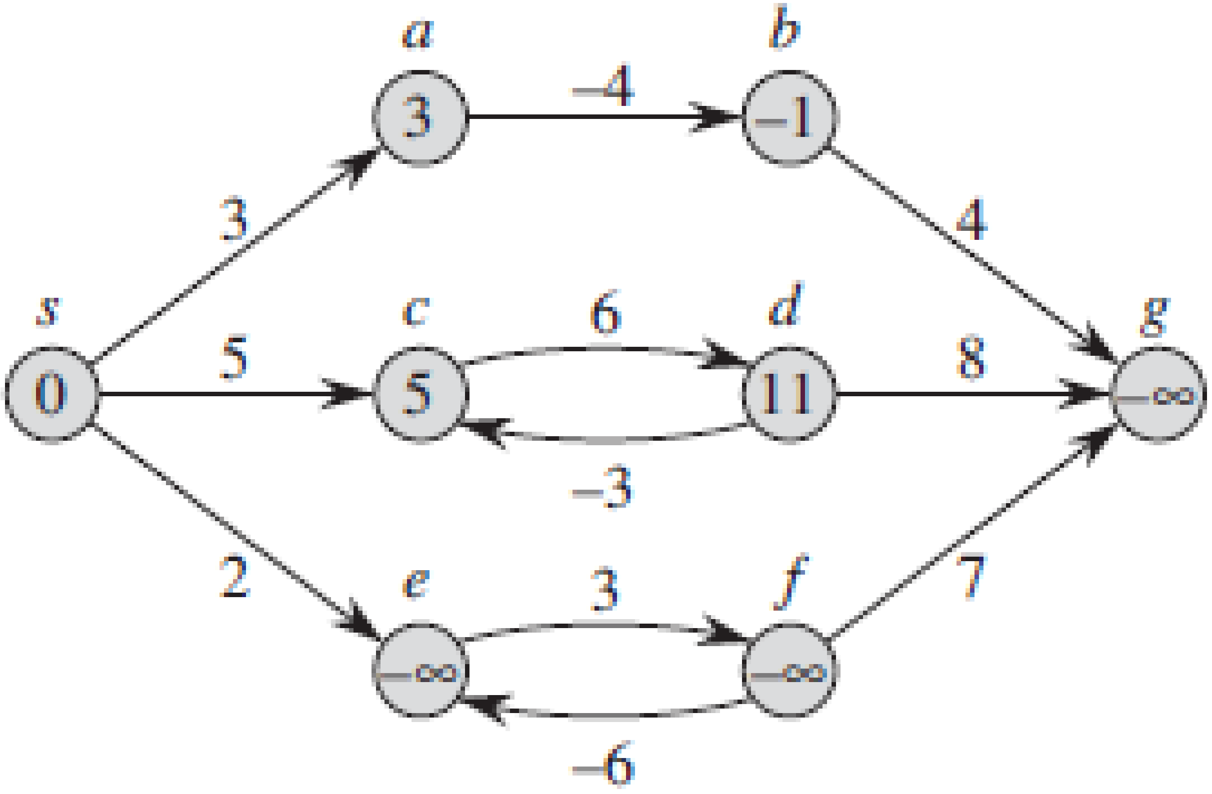
Ο αλγόριθμος επιλύει το πρόβλημα της εύρεσης του συντομότερου μονοπατιού στη γενική περίπτωση όπου τα βάρη μπορεί να είναι και αρνητικές τιμές

Δοσμένου ενός γράφου G=(E,V) με αρχικό κόμβο τον s και μια συνάρτηση βάρους w: E → R, ο αλγόριθμος επιστρέφει μια λογική τιμή που δείχνει αν υπάρχει κύκλος αρνητικών τιμών που μπορεί να προσπελαστεί από τον s

Αν υπάρχει τέτοιος κύκλος, το πρόβλημα δεν έχει λύση

Bellman-Ford Algorithm (2/10)

Παράδειγμα κύκλου με αρνητικές τιμές



Bellman-Ford Algorithm (3/10)

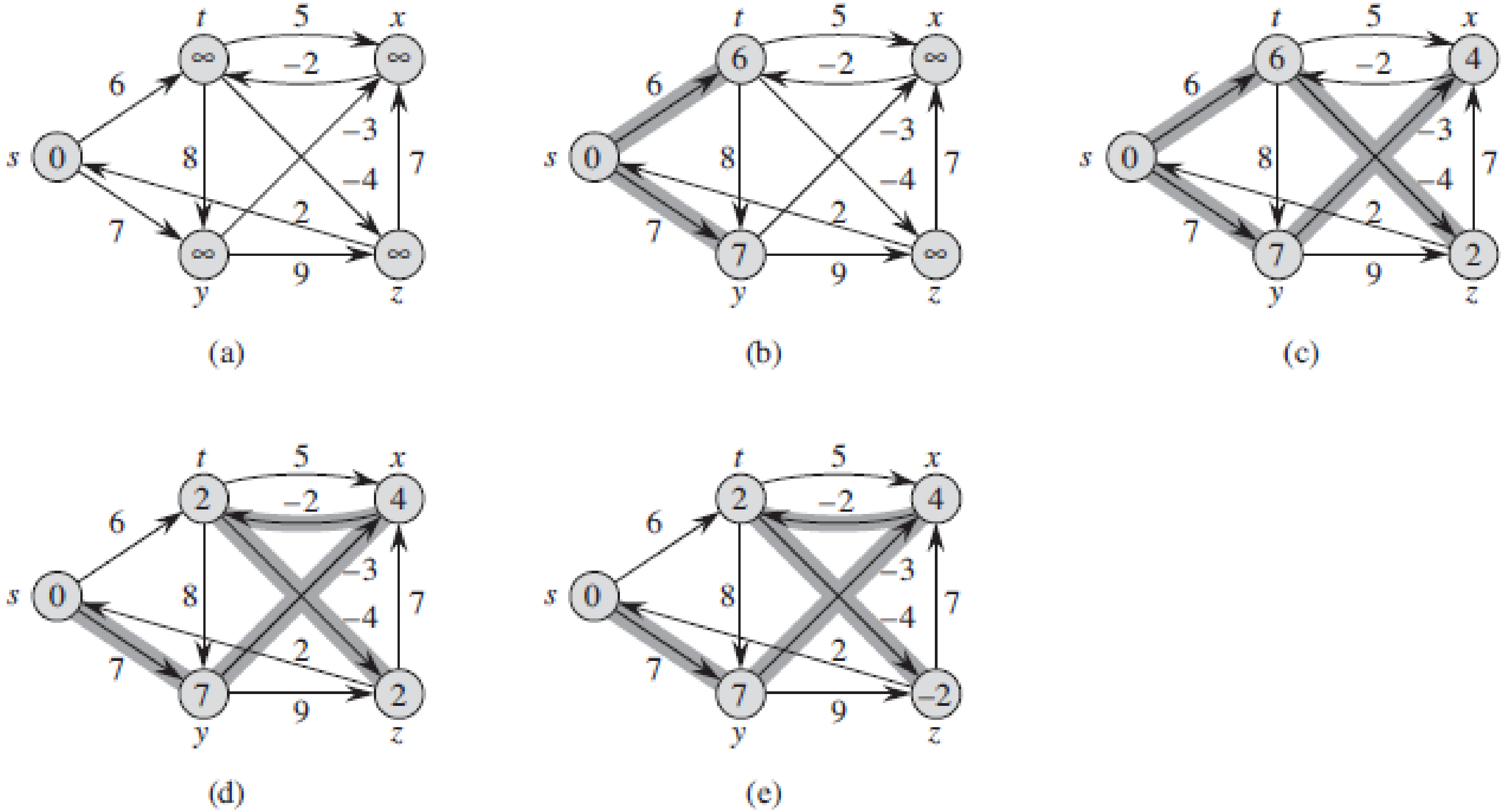
Αν υπάρχει τέτοιος κύκλος, δεν υπάρχει μονοπάτι από τον s που να είναι το συντομότερο μονοπάτι

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Bellman-Ford Algorithm (4/10)



Bellman-Ford Algorithm (5/10)

Demo

https://www.youtube.com/watch?v=obWXjtg0L64

https://www.youtube.com/watch?v=hq3TZInZ5J4

Bellman-Ford Algorithm (6/10)

Lemma

Έπειτα από |V|-1 επαναλήψεις ο αλγόριθμος Bellman-Ford θα έχει τοποθετήσει v.d=δ(s,v) για ένα κόμβο – πηγή s για όλους τους κόμβους v που μπορεί να προσπελαστούν από τον s.

Απόδειξη

Έστω κάθε κόμβος v μπορεί να προσπελαστεί από τον s και P=(s, v_1, v_2, …, v) το οποιοδήποτε συντομότερο μονοπάτι. Το P έχει το πολύ |V|-1 ακμές αφού είναι ένα απλό μονοπάτι. Συνεπώς, k <= |V|-1. Σε κάθε επανάληψη ο αλγόριθμος αλλάζει το βάρος (relaxes) όλων των ακμών |E|. Μέσα σε αυτές τις ακμές, είναι και η (v_{i-1}, v_i). Από την ιδιότητα του path relaxation έχουμε $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$

Bellman-Ford Algorithm (7/10)

Θεώρημα

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Bellman-Ford Algorithm (8/10)

Απόδειξη

Αποδεικνύουμε πρώτα ότι v.d=δ(s,v) για κάθε v ϵ V. Αν ο κόμβος v μπορεί να προσπελαστεί από τον s, τότε το πρηγούμενο Λήμμα αποδεικνύει τον αρχικό ισχυρισμό. Η ιδιότητα του predecessor-subgraph μαζί με τον ισχυρισμό μας δείχνει πως ο υπο-γράφος G_π είναι ένα συντομότερο μονοπάτι. Με το πέρας εκτέλεσης του αλγορίθμου αν επιστρέψει TRUE έχουμε πως:

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\ &= u.d + w(u, v) \end{aligned}$$

Bellman-Ford Algorithm (9/10)

Απόδειξη (συνέχεια)

Ας υποθέσουμε τώρα πως ο γράφος περιέχει ένα κύκλο με αρνητικές τιμές που μπορεί να προσπελαστεί από τον s. Έστω ότι ο κύκλος είναι ο $c = \langle v_0, v_1, \ldots, v_k \rangle$ με $v_0 = v_k$

Τότε $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$

Ας υποθέσουμε πως ο αλγόριθμος επιστρέφει TRUE

Τότε $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \quad i = 1, 2, \ldots, k$

Bellman-Ford Algorithm (10/10)

Απόδειξη (συνέχεια)

Έχουμε
$$\begin{aligned}\sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k}(v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)\end{aligned}$$

Αφού $v_0 = v_k$, κάθε κόμβος στο c εμφανίζεται μια φορά στα αθροίσματα, οπότε

$\displaystyle\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$ και $\displaystyle 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$ (αντίθετο με την αρχική εξίσωση)

Dijkstra’s Algorithm (1/6)

Ο αλγόριθμος Dijkstra επιλύει το πρόβλημα της εύρεσης του single source συντομότερου μονοπατιού όταν τα βάρη στις ακμές δεν είναι αρνητικά

Υποθέτουμε πως (u,v)>=0

Με μια καλή υλοποίηση, ο χρόνος εκτέλεσης του αλγορίθμου είναι καλύτερος από του Bellman-Ford

Ο αλγόριθμος διατηρεί ένα σύνολο κόμβων των οποίων το συντομότερο μονοπάτι από τον κόμβο s έχει ήδη καθοριστεί

Dijkstra’s Algorithm (2/6)

Επαναληπτικά ο αλγόριθμος:

- Επιλέγει τον κόμβο u ϵ V – S με την εκτίμηση του μικρότερου μονοπατιού
- Προσθέτει το u στο S
- Εφαρμόζει την τεχνική relaxation για όλες τις ακμές που φεύγουν από τον u

Υλοποίηση με ουρά μικρότερης προτεραιότητας

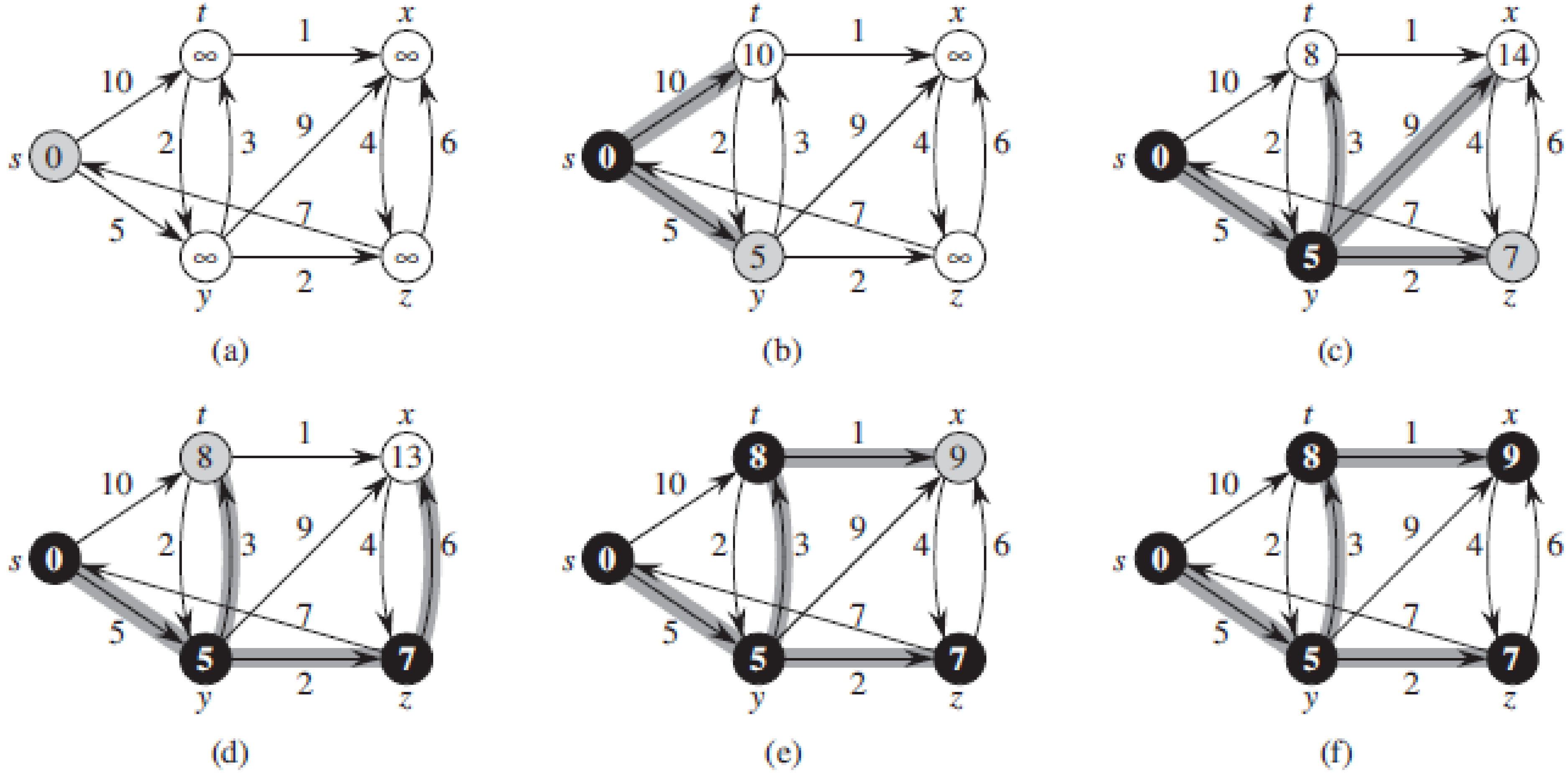
Dijkstra’s Algorithm (3/6)

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Dijkstra’s Algorithm (4/6)



Dijkstra’s Algorithm (5/6)

Demo

http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html

Dijkstra’s Algorithm (6/6)

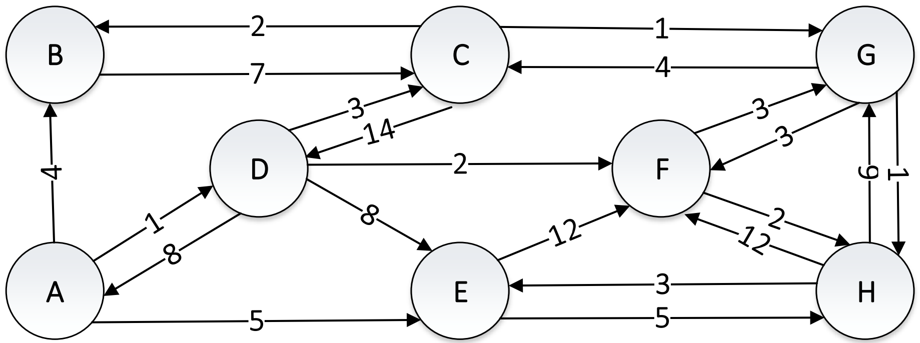
Θεώρημα

Ο αλγόριθμος Dijkstra αν εφαρμοστεί πάνω σε ένα γράφο κατευθυνόμενο με μη αρνητικά βάρη θα τερματίσει με $u.d = \delta(s,u)$ για κάθε $u \in V$.

Απόδειξη

Θα τη βρείτε στο βιβλίο

Άσκηση



S	A	B	C	D	E	F	G	H
A	0	4 (A)		1 (A)	5 (A)			
A D	0	4 (A)	4 (D)	1	5 (A)	3 (D)		
A D F	0	4 (A)	4 (D)	1	5 (A)	3	6 (F)	5 (F)
A D F B	0	4	4 (D)	1	5 (A)	3	6 (F)	5 (F)
A D F B C	0	4	4	1	5 (A)	3	5 (C)	5 (F)
A D F B C E	0	4	4	1	5	3	5 (C)	5 (F)
A D F B C E G	0	4	4	1	5	3	5	5 (F)
A D F B C E G H	0	4	4	1	5	3	5	5

All Pairs Shortest Paths (1/10)

Το πρόβλημα είναι να βρούμε τα συντομότερα μονοπάτια ανάμεσα σε όλους τους κόμβους

Για κάθε ζεύγος u, v θέλουμε να εξάγουμε το συντομότερο μονοπάτι

Τυπικά, θέλουμε το αποτέλεσμα σε μια μορφή πίνακα

Μπορούμε να λύσουμε το πρόβλημα αν τρέξουμε |V| φορές ένα αλγόριθμο εύρεσης του συντομότερου μονοπατιού

Αν ο γράφος έχει αρνητικά βάρη μπορούμε να υιοθετήσουμε τον Bellman-Ford αν όχι τον Dijkstra

Πολυπλοκότητα: $O(V^2 E)$ ή σε πυκνούς γράφους $O(V^4)$

All Pairs Shortest Paths (2/10)

Θα υιοθετήσουμε λύση με βάση πίνακες γειτνίασης

Ένας πίνακας γειτνίασης περιλαμβάνει στοιχεία ως εξής:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \ . \end{cases}$$

Ο πίνακας με τα στοιχεία των συντομότερων μονοπατιών περιέχει στοιχεία d_{ij} που απεικονίζουν το βάρος του συντομότερου μονοπατιού που συνδέει τους αντίστοιχους κόμβους

All Pairs Shortest Paths (3/10)

Πρέπει να υπολογίσουμε όχι μόνο τα βάρη των συντομότερων μονοπατιών αλλά και τον πίνακα των προηγούμενων κόμβων

Τα στοιχεία π_{ij} δείχνουν αν δεν υπάρχει μονοπάτι ανάμεσα σε δύο κόμβους (NIL) ή τον προηγούμενο κόμβο σε κάποιο συντομότερο μονοπάτι

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i == j
    print i
elseif π_ij == NIL
    print "no path from" i "to" j "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
    print j
```

All Pairs Shortest Paths (4/10)

Έστω $l_{ij}^{(m)}$ το ελάχιστο βάρος οποιουδήποτε μονοπατιού από τον κόμβο i στον κόμβο j που περιέχει το πολύ m ακμές

Παίρνουμε σαν είσοδο τον πίνακα $W=(w_{ij})$ και υπολογίζουμε μια σειρά από πίνακες $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ όπου για m=1,2,…,n-1 έχουμε $L^{(m)} = (l_{ij}^{(m)})$

Ο τελικός πίνακας $L^{(n-1)}$ περιέχει τα τελικά βάρη των συντομότερων μονοπατιών

All Pairs Shortest Paths (5/10)

EXTEND-SHORTEST-PATHS(L, W)

$n = L.rows$
let $L' = (l'_{ij})$ be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        $l'_{ij} = \infty$
        for $k = 1$ **to** n
            $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

$$
\begin{aligned}
l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\
&= \min_{1 \le k \le n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\}
\end{aligned}
$$

All Pairs Shortest Paths (6/10)

Με πολ/μο πινάκων

$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W &= W \\
L^{(2)} &= L^{(1)} \cdot W &= W^2 \\
L^{(3)} &= L^{(2)} \cdot W &= W^3 \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W &= W^{n-1}
\end{aligned}
$$

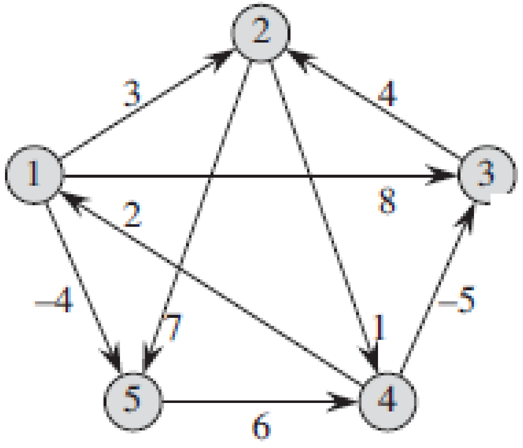
All Pairs Shortest Paths (7/10)

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(1)} = W$
for $m = 2$ **to** $n - 1$
    let $L^{(m)}$ be a new $n \times n$ matrix
    $L^{(m)} =$ EXTEND-SHORTEST-PATHS$(L^{(m-1)}, W)$
return $L^{(n-1)}$

All Pairs Shortest Paths (8/10)

Παράδειγμα



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

EXTEND-SHORTEST-PATHS(L, W)

```
n = L.rows
let L′ = (l′_ij) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        l′_ij = ∞
        for k = 1 to n
            l′_ij = min(l′_ij, l_ik + w_kj)
return L′
```

All Pairs Shortest Paths (9/10)

Βελτίωση του χρόνου εκτέλεσης

- Πράξεις

$$
\begin{aligned}
L^{(1)} &= W\,, \\
L^{(2)} &= W^2 &= W \cdot W \\
L^{(4)} &= W^4 &= W^2 \cdot W^2 \\
L^{(8)} &= W^8 &= W^4 \cdot W^4 \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}
\end{aligned}
$$

- Χρόνος υπολογισμού ceiling(log(n-1))

All Pairs Shortest Paths (10/10)

FASTER-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(1)} = W$
$m = 1$
while $m < n - 1$
    let $L^{(2m)}$ be a new $n \times n$ matrix
    $L^{(2m)} = $ EXTEND-SHORTEST-PATHS$(L^{(m)}, L^{(m)})$
    $m = 2m$
return $L^{(m)}$

Floyd-Warshall Algorithm (1/12)

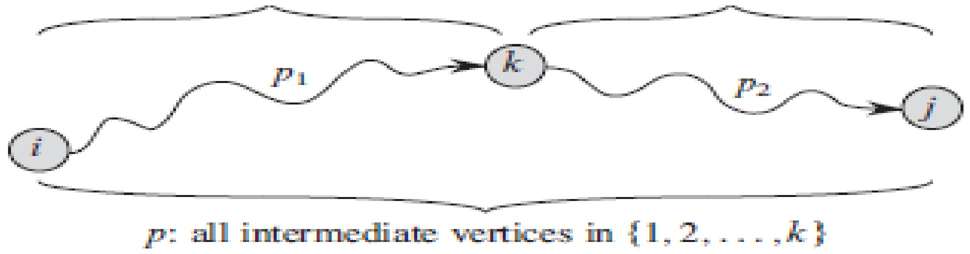
Ο αλγόριθμος βασίζεται στην ακόλουθη παρατήρηση:

- Ας θεωρήσουμε ένα υποσύνολο κόμβων {1,2,…, k}
- Για κάθε ζεύγος κόμβων i, j θεωρούμε όλα τα μονοπάτια που τους συνδέουν και έχουν ενδιάμεσες ακμές από το {1,2,…, k}
- Έστω p ένα μονοπάτι ελάχιστου βάρους ανάμεσά τους
- Ο αλγόριθμος αναζητά τη σχέση ανάμεσα στο p και στα συντομότερα μονοπάτια από το i στο j με ενδιάμεσους κόμβους στο σύνολο {1,2,…, k-1}
- Η σχέση εξαρτάται από το αν ο k κόμβος είναι ένας ενδιάμεσος κόμβος του p

Floyd-Warshall Algorithm (2/12)

Αν ο k **δεν είναι ένας ενδιάμεσος κόμβος** του p, τότε όλοι οι ενδιάμεσοι κόμβοι του p είναι στο σύνολο {1,2,…, k-1}. Έτσι ένα συντομότερο μονοπάτι από το i στο j με ενδιάμεσους κόμβους στο {1,2,…, k-1} είναι επίσης ένα συντομότερο μονοπάτι με ενδιάμεσους κόμβους στο {1,2,…, k}

Αν ο k **είναι ένας ενδιάμεσος κόμβος** του p, τότε χωρίζουμε το p σε δύο τμήματα: i-(p_1)-k-(p_2)-j. Το p_1 είναι ένα συντομότερο μονοπάτι από το i στο k με ενδιάμεσους κόμβους στο {1,2,…, k}. Όλοι οι ενδιάμεσοι κόμβοι του p_1 είναι στο {1,2,…, k-1}. Έτσι, το p_1 είναι ένα συντομότερο μονοπάτι. Ομοίως για το p_2.



Floyd-Warshall Algorithm (3/12)

Έστω $d_{ij}^{(k)}$ το βάρος του συντομότερου μονοπατιού από τον κόμβο i στον j για το οποίο όλες οι ενδιάμεσοι κόμβοι είναι στο {1,2,…,k}.

Όταν k=0, ένα μονοπάτι από το i στο j με μεγαλύτερο βάρος από 0 δεν έχει ενδιάμεσους κόμβους

Ένα τέτοιο μονοπάτι έχει το πολύ μια ακμή και έτσι $d_{ij}^{(0)} = w_{ij}$

Η αναδρομική σχέση έχει ως εξής:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall Algorithm (4/12)

Βασιζόμενοι στην αναδρομική σχέση, μπορούμε χρησιμοποιώντας μια bottom-up προσέγγιση να υπολογίσουμε τις τιμές του $d_{ij}^{(k)}$

Ο αλγόριθμος θα μας επιστρέψει ένα πίνακα με τα βάρη των συντομότερων μονοπατιών

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Floyd-Warshall Algorithm (5/12)

Παράδειγμα

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Floyd-Warshall Algorithm (6/12)

Παράδειγμα

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Floyd-Warshall Algorithm (7/12)

Η δημιουργία των συντομότερων μονοπατιών περιλαμβάνει την κατασκευή ενός πίνακα Π που περιέχει τους προηγούμενους κόμβους στα συντομότερα μονοπάτια

Μπορούμε να υπολογίσουμε μια αλληλουχία πινάκων $\Pi^{(0)}$, $\Pi^{(1)}$, …, $\Pi^{(n)}$ καθώς κατασκευάζουμε τους πίνακες $D^{(i)}$

Ως $\pi_{ij}^{(k)}$ ορίσουμε τον προηγούμενο κόμβο j σε ένα συντομότερο μονοπάτι από το i με όλους τους ενδιάμεσους κόμβους στο {1,2,…,k}

Η εξίσωση που ισχύει έχει ως εξής:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

Floyd-Warshall Algorithm (8/12)

Για k>=1, αν πάρουμε το μονοπάτι i-k-j με k διαφορετικό του j, τότε ο προηγούμενος κόμβος του j είναι ο ίδιος του j που επιλέξαμε στο συντομότερο μονοπάτι από το k με όλους τους ενδιάμεσους στο {1,2,…,k-1}

Διαφορετικά, παίρνουμε τον προηγούμενο κόμβο του j που επιλέξαμε στο συντομότερο μονοπάτι από το i με όλους τους ενδιάμεσους στο {1,2,…,k-1}

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Floyd-Warshall Algorithm (9/12)

Ορισμένες φορές θέλουμε να ορίσουμε αν ένας γράφος περιλαμβάνει ένα μονοπάτι από το i στο j για κάθε ζευγάρι κόμβων

Η ιδιότητα αυτή ορίζεται ως η **μεταβατική κλειστότητα** (transitive closure) του γράφου

Για τον υπολογισμό της αντικαθιστούμε με τη λογική πράξη του OR για το min και την πράξη AND για το + στον αλγόριθμο Floyd-Warshall

Για κάθε i,j,k=1,2,…,n ορίζουμε το $t_{ij}^{(k)}$ που παίρνει την τιμή 1 αν υπάρχει μονοπάτι από τον κόμβο i στον κόμβο j με όλους τους ενδιάμεσους κόμβους στο {1,2,…,k} και 0 διαφορετικά

Floyd-Warshall Algorithm (10/12)

Η αναδρομική σχέση έχει ως εξής:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

$$k \geq 1,$$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$$

Floyd-Warshall Algorithm (11/12)

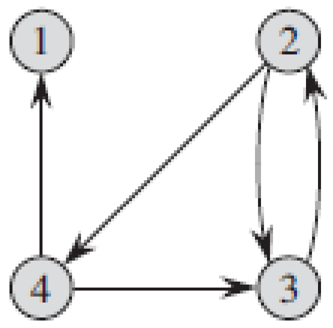
TRANSITIVE-CLOSURE(G)

$n = |G.V|$
let $T^{(0)} = \left(t_{ij}^{(0)}\right)$ be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        if $i == j$ or $(i, j) \in G.E$
            $t_{ij}^{(0)} = 1$
        else $t_{ij}^{(0)} = 0$
for $k = 1$ **to** n
    let $T^{(k)} = \left(t_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$

Floyd-Warshall Algorithm (12/12)

TRANSITIVE-CLOSURE(G)

$n = |G.V|$
let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix
for $i = 1$ to n
    for $j = 1$ to n
        if $i == j$ or $(i, j) \in G.E$
            $t_{ij}^{(0)} = 1$
        else $t_{ij}^{(0)} = 0$
for $k = 1$ to n
    let $T^{(k)} = (t_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ to n
        for $j = 1$ to n
            $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Johnson’s Algorithm (1/4)

Ο αλγόριθμος είτε επιστρέφει ένα πίνακα με τα βάρη των συντομότερων μονοπατιών ή επίστρέφει πως ο γράφος έχει ένα κύκλο αρνητικών τιμών

Υιοθετεί τους αλγορίθμους των Bellman-Ford & Dijkstra

Υιοθετεί την τεχνική του επαναυπολογισμού βαρών (reweighting)

Αν όλα τα βάρη είναι μη αρνητικά, μπορούμε να βρούμε τα συντομότερα μονοπάτια με τη χρήση του Dijkstra υιοθετώντας τον για κάθε κόμβο

Αν υιοθετήσουμε σωρούς Fibonacci η πολυπλοκότητα είναι $O(V^2 \log V + V E)$

Johnson’s Algorithm (2/4)

Αν ο γράφος έχει αρνητικά βάρη αλλά όχι κύκλους αρνητικών τιμών, μπορούμε να υπολογίσουμε ένα νέο σύνολο βαρών που μας επιτρέπει να υιοθετήσουμε την προηγούμενη μέθοδο

Το νέο σύνολο πρέπει να ικανοποιεί τα ακόλουθα:

- Για κάθε ζεύγος κόμβων, ένα μονοπάτι είναι το συντομότερο για μια συνάρτηση βαρών w αν είναι συντομότερο για μια συνάρτηση g – πρέπει να ορίσουμε την $\hat{w}$
- Για κάθε ακμή του γράφου, το νέο βάρος να είναι μη αρνητικό

Johnson’s Algorithm (3/4)

Η συνάρτηση που χρησιμοποιούμε είναι η ακόλουθη:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

Όπου h είναι μια συνάρτηση που απεικονίζει ένα κόμβο σε πραγματικούς αριθμούς

Johnson's Algorithm (4/4)

JOHNSON(G, w)

compute G', where $G'.V = G.V \cup \{s\}$,
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
    $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD(G', w, s) == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
        set $h(v)$ to the value of $\delta(s, v)$
            computed by the Bellman-Ford algorithm
    for each edge $(u, v) \in G'.E$
        $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
    let $D = (d_{uv})$ be a new $n \times n$ matrix
    for each vertex $u \in G.V$
        run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in G.V$
        for each vertex $v \in G.V$
            $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$
    return D

Maximum Flow (1/7)

Μπορούμε να θεωρήσουμε ένα γράφο ως ένα δίκτυο ροής

Παράδειγμα: σε ένα κόμβο κατασκευάζεται ένα υλικό ενώ σε κάποιον άλλο καταναλώνεται

Η ροή σε κάθε σημείο του δικτύου είναι ο ρυθμός με τον οποίο το υλικό μετακινείται

Στο πρόβλημα της μέγιστης ροής (maximum flow) θέλουμε να βρούμε το μεγαλύτερο ρυθμό με το οποίο μπορούμε να μετακινήσουμε το υλικό μέσα στο δίκτυο

Maximum Flow (2/7)

Ένα δίκτυο ροής (flow network) G=(V,E) είναι ένας κατευθυνόμενος γράφος στο οποίο κάθε ακμή έχει μη αρνητική τιμή (χωρητικότητα - capacity)

Αν ο γράφος περιλαμβάνει μια ακμή προς μια κατεύθυνση τότε δεν υπάρχει ακμή προς την αντίθετη κατεύθυνση

Διακρίνουμε δύο ειδών κόμβους: τους κόμβους – πηγές (source) και τους κόμβους καταβόθρες (sink)

Maximum Flow (3/7)

Μια ροή (flow) είναι μια συνάρτηση f: V X V → R που ικανοποιεί τα ακόλουθα κριτήρια:

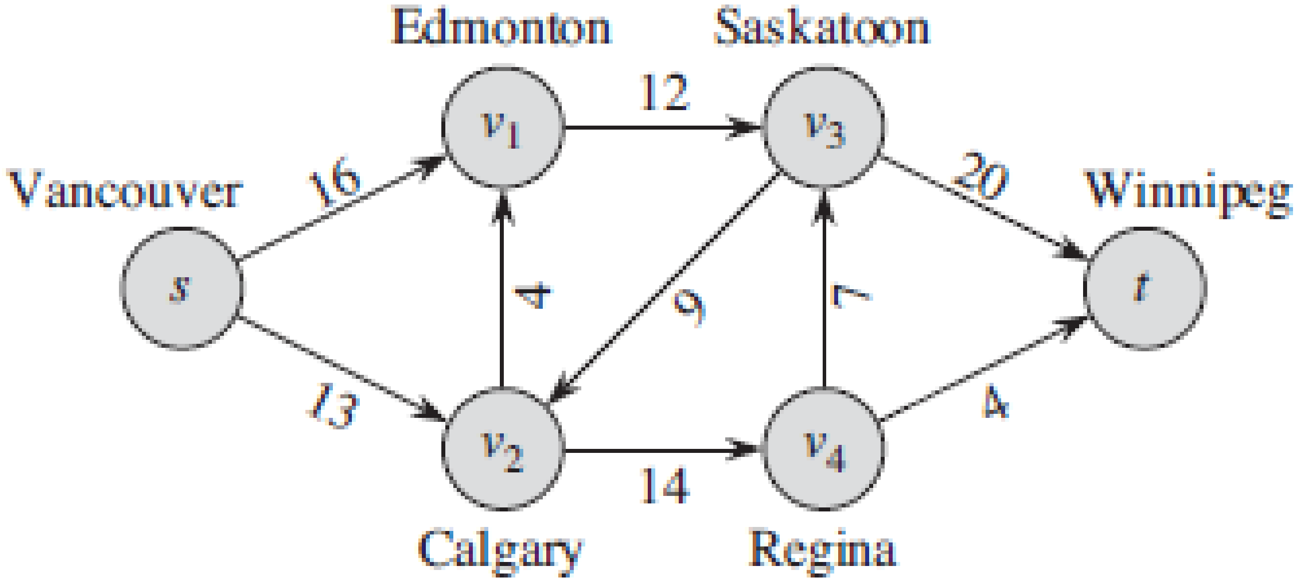
- Για κάθε u,v ϵ V, απαιτείται 0<= f(u,v) <= c(u,v), όπου c(u,v) είναι η χωρητικότητα της διαδρομής (**capacity constraint**)
- Για κάθε v ϵ V – {s, t} απαιτείται (**flow conservation**)

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

- Η ποσότητα |f| (**flow value**) απεικονίζει τη συνολική ροή από το s μείον τη συνολική ροή προς το s
- Όταν μια ακμή δεν ανήκει στο E τότε έχουμε f(u,v)=0

Maximum Flow (4/7)

Παράδειγμα



(a)

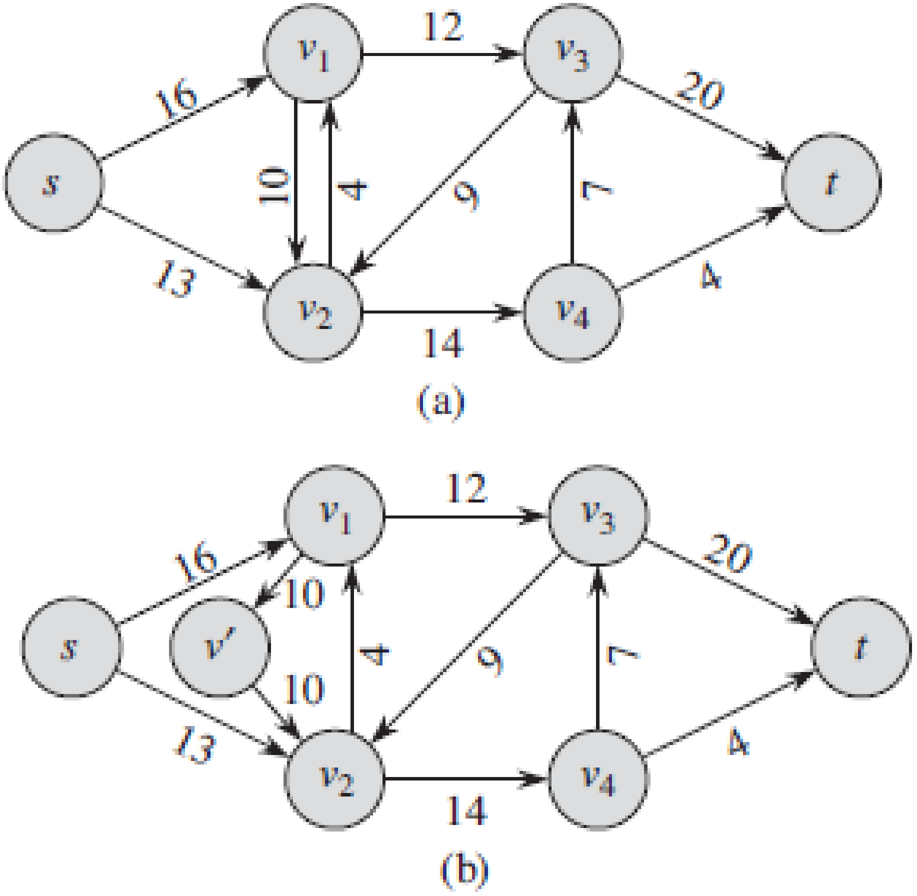
Maximum Flow (5/7)

Ας υποθέσουμε πως σε μια ακμή δημιουργούμε μια ‘αντίθετή’ της

Προφανώς το δίκτυο παραβιάζει την αρχική υπόθεση της μη ύπαρξης αντίθετων ακμών

Ονομάζουμε τις δύο ακμές **αντιπαράλληλες (antiparallel)**

Για είμαστε σύμφωνοι με την αρχική υπόθεση δημιουργούμε ένα ενδιάμεσο κόμβο



Maximum Flow (6/7)

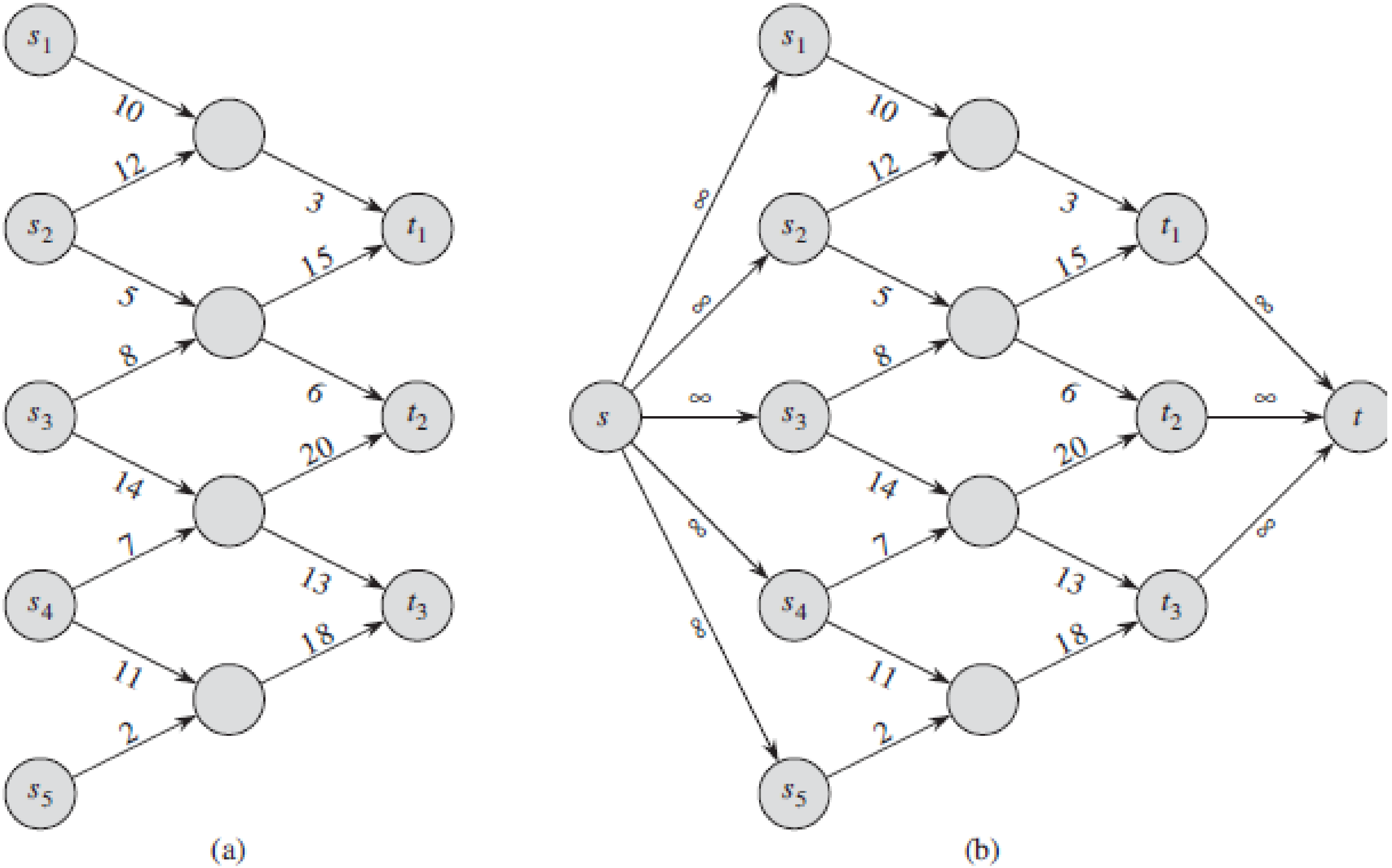
Το πρόβλημα της μέγιστης ροής μπορεί να εμπλέκει πολλαπλές πηγές και πολλαπλά sinks

Μπορούμε όμως να το αντιστοιχίσουμε στο αρχικό μας πρόβλημα εύκολα

Προσθέτουμε μια υπερ-πηγή (super-source) και ένα super-sink θέτοντας τη χωρητικότητά τους στο άπειρο

Maximum Flow (7/7)

Παράδειγμα



Ford-Fulkerson Method (1/17)

Υπάρχουν πολλές υλοποιήσεις με διαφορετικούς χρόνους εκτέλεσης

Βασίζεται στις ακόλουθες ιδέες:

- Residual networks
- Augmenting paths
- Cuts

Η μέθοδος αυξάνει επαναληπτικά την τιμή μιας ροής

Ξεκινάμε με f(u,v) = 0 για κάθε ζεύγος κόμβων

Σε κάθε επανάληψη, αυξάνουμε την τιμή της ροής στο G βρίσκοντας το augmenting path σε ένα residual network G_f

Ford-Fulkerson Method (2/17)

Όταν γνωρίζουμε τις ακμές του G_f είναι εύκολο στη συνέχεια να βρούμε ακμές του G για τις οποίες μπορούμε να αλλάξουμε τη ροή και να αυξήσουμε την τιμή της

Κατά τη διάρκεια εκτέλεσης του αλγορίθμου, η ροή σε οποιαδήποτε ακμή μπορεί να αυξηθεί ή να μειωθεί

Η μείωση μπορεί να είναι απαραίτητη ώστε να ‘αναγκαστεί’ ο αλγόριθμος να στείλει μεγαλύτερη ροή προς το sink

Ο αλγόριθμος θα σταματήσει όταν δεν μπορεί να αναγνωριστούν επιπλέον augmenting paths

Ford-Fulkerson Method (3/17)

FORD-FULKERSON-METHOD(G, s, t)

```
initialize flow f to 0
while there exists an augmenting path p in the residual network G_f
    augment flow f along p
return f
```

Ford-Fulkerson Method (4/17)

Residual networks

- Δοσμένου ενός γράφου G και μιας ροής f, το residual network G_f αποτελείται από τις ακμές με χωρητικότητα που αναπαριστά το πώς μπορούμε να αλλάξουμε τη ροή στις ακμές του G
- Μια ακμή μπορεί να δεχτεί μια ποσότητα επιπρόσθετης ροής ίση με τη χωρητικότητα της ακμής μείον τη ροή σε αυτή την ακμή
- Αν η τιμή αυτή είναι θετική, βάζουμε την ακμή στο G_f με μια εναπομείνουσα χωρητικότητα $c_f(u,v) = c(u,v) - f(u,v)$
- Οι μόνες ακμές του G που μπορούν να μπουν στο G_f είναι αυτές που μπορούν να 'δεχτούν' μεγαλύτερη ροή
- Οι ακμές των οποίων η ροή είναι ίση με τη χωρητικότητά τους έχουν $c_f(u,v) = 0$ και δεν μπαίνουν στο G_f

Ford-Fulkerson Method (5/17)

Residual networks

- Ο αλγόριθμος προσπαθεί σε κάθε βήμα να αυξήσει την τιμή της συνολικής ροής
- Υπάρχει περίπτωση να προσπαθήσει να μειώσει τη ροή σε κάποιες ακμές
- Για την αναπαράσταση της μείωσης μιας θετικής ροής f(u,v) τοποθετούμε την ακμή (v,u) στο G_f με χωρητικότητα $c_f(v,u) = f(u,v)$
- Με αυτό τον τρόπο ‘ακυρώνουμε’ τη ροή της ακμής (u,v)
- Αυτές οι ακμές ονομάζονται αντίστροφες ακμές αφού στέλνουν πίσω τη ροή

Ford-Fulkerson Method (6/17)

Residual networks

- Η εναπομείνουσα χωρητικότητα $c_f(u,v)$ ορίζεται ως εξής:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Παράδειγμα:
  - c(u,v)=16, f(u,v)=11; Η ροή μπορεί να αυξηθεί το πολύ κατά $c_f(u,v) = 5$
- To residual network ορίζεται ως εξής:

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\} \qquad |E_f| \le 2\,|E|$$

Ford-Fulkerson Method (7/17)

Residual networks

- Αν f είναι μια ροή στο G και f' είναι μια ροή στο residual network G_f ορίζουμε $f \uparrow f'$ την αυξητική ροή από την f στην f' ως μια συνάρτηση από το VXV στο R ως εξής:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Ford-Fulkerson Method (8/17)

Lemma

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Ford-Fulkerson Method (9/17)

Augmenting paths

- Ένα augmenting path είναι ένα απλό μονοπάτι από το s στο t στο residual network G_f

Lemma

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \,, \\ 0 & \text{otherwise} \,. \end{cases}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Ford-Fulkerson Method (10/17)

Corollary

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (26.8), and suppose that we augment f by f_p. Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Ford-Fulkerson Method (11/17)

Cuts

- Το βασικό ερώτημα στον αλγόριθμο είναι το πώς θα καταλάβουμε ότι πρέπει να σταματήσουμε
- Με βάση το θεώρημα **max-flow min-cut**, **μια ροή είναι η μέγιστη όταν και μόνο όταν αν το residual network δεν περιέχει κάποιο augmenting path**
- Μια τμηματοποίηση (cut) (S, T) είναι ένας διαχωρισμός των κόμβων V σε S & T=V-S τέτοιος ώστε το s να ανήκει στο S και το t να ανήκει στο T
- Αν η f είναι μια ροή, τότε η ροή δικτύου (net flow) f(S,T) μεταξύ της τμηματοποίησης (S, T) ορίζεται ως εξής:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

Ford-Fulkerson Method (12/17)

Cuts

- Η χωρητικότητα της τμηματοποίησης είναι

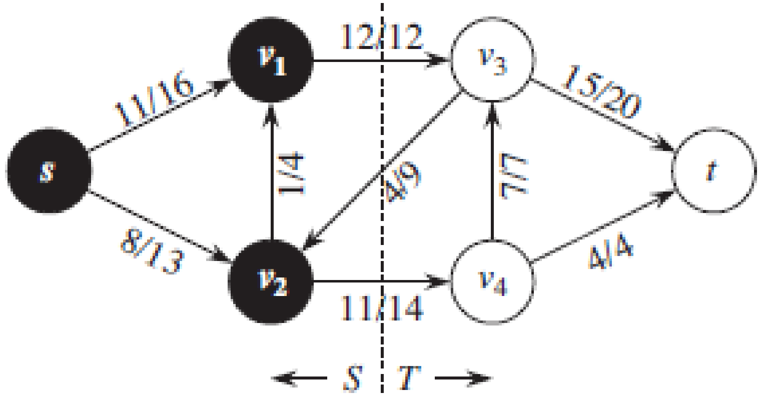
  $$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

- Η ελάχιστη τμηματοποίηση ενός δικτύου κόμβων είναι μια τμηματοποίηση της οποίας η χωρητικότητα είναι η ελάχιστη σε σχέση με όλους τους διαχωρισμούς του δικτύου

Ford-Fulkerson Method (13/17)

Cuts

- Παράδειγμα



$$
\begin{aligned}
f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\
&= 19 \\
c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\
&= 26
\end{aligned}
$$

Ford-Fulkerson Method (14/17)

Lemma

Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Corollary

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Theorem (Max-flow min-cut theorem)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Ford-Fulkerson Method (15/17)

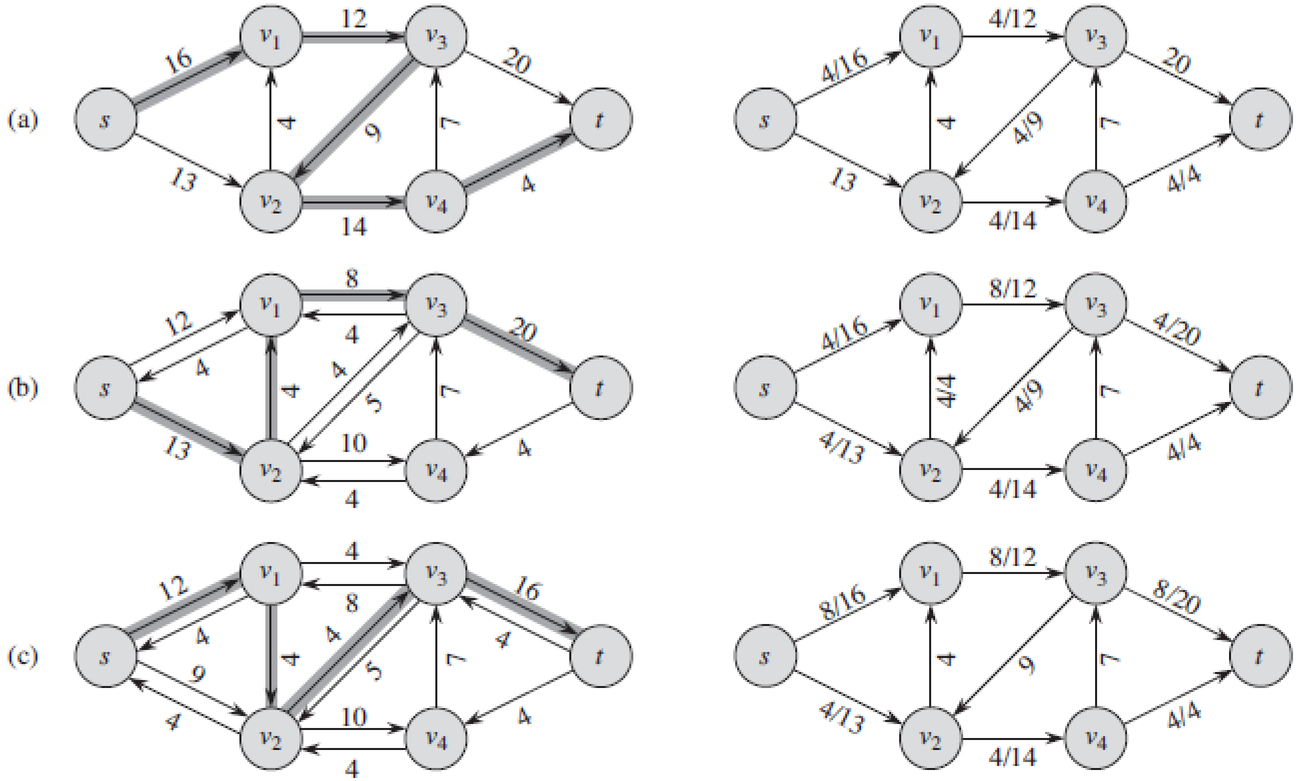
Βασικός αλγόριθμος

- Σε κάθε βήμα βρίσκουμε ένα augmenting path και τροποποιούμε τη ροή f
- Αντικαθιστούμε το f με $f \uparrow f_p$ και παίρνουμε μια νέα ροή με τιμή $|f| + |f_p|$

```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ G.E
    (u, v).f = 0
while there exists a path p from s to t in the residual network G_f
    c_f(p) = min {c_f(u, v) : (u, v) is in p}
    for each edge (u, v) in p
        if (u, v) ∈ E
            (u, v).f = (u, v).f + c_f(p)
        else (v, u).f = (v, u).f − c_f(p)
```

Ford-Fulkerson Method (16/17)

Παράδειγμα εκτέλεσης




```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ G.E
    (u, v).f = 0
while there exists a path p from s to t in the residual network G_f
    c_f(p) = min {c_f(u, v) : (u, v) is in p}
    for each edge (u, v) in p
        if (u, v) ∈ E
            (u, v).f = (u, v).f + c_f(p)
        else (v, u).f = (v, u).f − c_f(p)
```

Ford-Fulkerson Method (17/17)

Ανάλυση αλγορίθμου

- Ο χρόνος εκτέλεσης εξαρτάται από το χρόνο εύρεσης του augmenting path
- Με χρήση του BFS ο αλγόριθμος έχει πολυωνυμικό χρόνο εκτέλεσης
- Ο χρόνος για την εύρεση ενός μονοπατιού είναι O(E) είτε με BFS ή με DFS
- Ο συνολικός χρόνος εκτέλεσης του αλγορίθμου είναι $O(E|f^*|)$

Edmonds-Karp Algorithm (1/2)

Αποτελεί βελτίωση του Ford-Fulkerson

Βρίσκει το augmenting path μέσω αναζήτησης κατά πλάτος

Επιλέγει το augmenting path ως το συντομότερο μονοπάτι από το s στο t

Ο αλγόριθμος εκτελείται σε $O(V\ E^2)$

Edmonds-Karp Algorithm (2/2)

Lemma

If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Theorem

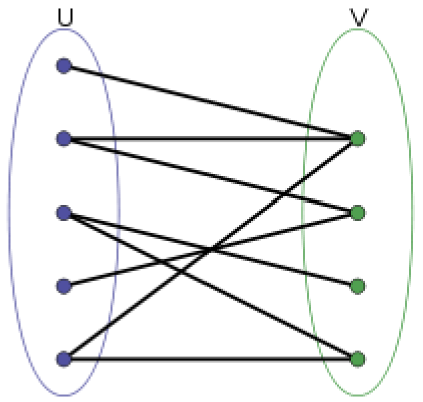
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Maximum Matching in Bipartite Graphs

Μέγιστο Διμερές Ταίριασμα (1/5)

Πρόβλημα

- Εύρεση του μέγιστου ταιριάσματος σε ένα **διμερή γράφο (bipartite graph)**
- Ένας διμερής γράφος είναι ένας γράφος του οποίου οι κόμβοι μπορούν να διαιρεθούν σε δύο αμοιβαία αποκλειόμενα και ανεξάρτητα σύνολα U, V τέτοια ώστε κάθε ακμή συνδέει ένα κόμβο του U με ένα κόμβο του V
- Παράδειγμα



Μέγιστο Διμερές Ταίριασμα (2/5)

Δοσμένου ενός γράφου G=(V,E), ένα **ταίριασμα (matching)** είναι ένα υποσύνολο ακμών M τέτοιο ώστε για όλους τους κόμβους που ανήκουν στο $v \in V$ το πολύ μια ακμή είναι προσπίπτουσα προς τον κόμβο v

Λέμε πως ο v έχει ταιριάξει αν υπάρχει μια ακμή στο M ενώ σε αντίθετη περίπτωση λέμε πως ο κόμβος δεν έχει ταιριάξει

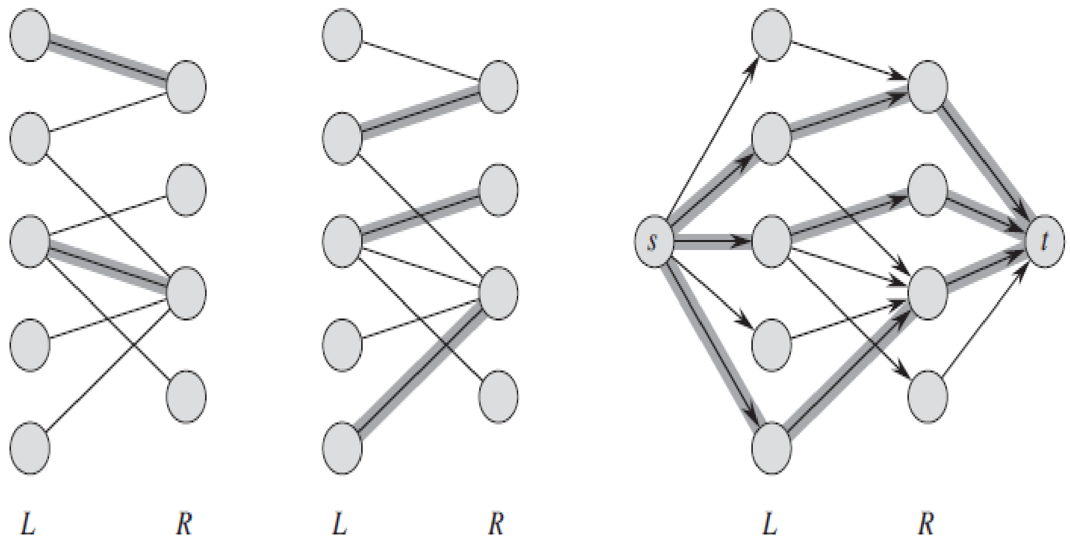
Ένα μέγιστο ταίριασμα είναι ένα ταίριασμα με τη μέγιστη πληθικότητα (cardinality), τέτοιο ώστε $|M| >= |M'|$ για κάθε M'

Εστιάζουμε σε διμερείς γράφους

Μέγιστο Διμερές Ταίριασμα (3/5)

Λύση

- Μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο Ford-Fulkerson για την εύρεση του μέγιστου ταιριάσματος
- Κατασκευάζουμε ένα δίκτυο ροής όπου η ροές αντιστοιχούν με ταιριάσματα

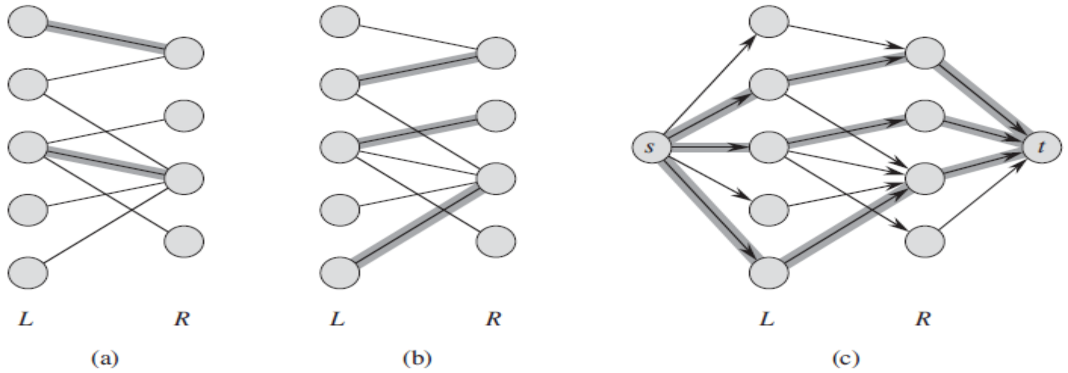


(a) (b) (c)

Μέγιστο Διμερές Ταίριασμα (4/5)

Βήματα

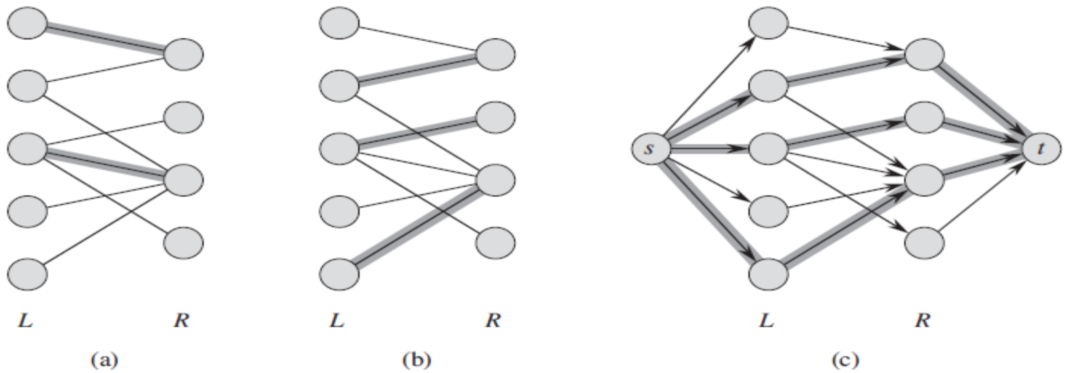
- Ορίζουμε δύο νέους κόμβους s, t
- Το νέο σύνολο των κόμβων του γράφου είναι $V' = V \cup \{s,t\}$
- Αν ο διαχωρισμός των κόμβων είναι $V = L \cup R$, οι κατευθυνόμενες ακμές του νέου γράφου G' θα είναι οι ακμές που ανήκουν στο E που κατευθύνονται από το L στο R
- Προσθέτουμε |V| νέες ακμές στο E ώστε να παραχθεί το νέο σύνολο των ακμών E'
- Το E' ορίζεται ως εξής: $E' = \{(s,u) : u \in L\} \cup \{(u,v) : (u,v) \in E\} \cup \{(v,t) : v \in R\}$



Μέγιστο Διμερές Ταίριασμα (5/5)

Βήματα (συνέχεια)

- Καθορίζουμε τη χωρητικότητα κάθε ακμής που ανήκει στο E’
- Αφού κάθε κόμβος στο V έχει τουλάχιστον μια προσπίπτουσα ακμή, τότε |E| >= |V|/2
- Ισχύει: |E|<=|E’|=|E|+|V|<= 3 |E|
- Συνεπώς |E’| = Θ(E)



Red Black Trees

Red Black Trees (1/26)

Ένα red-black δένδρο, είναι ένα δυαδικό δένδρο αναζήτησης στο οποίο ο κάθε κόμβος έχει ένα bit επιπλέον: το χρώμα του (κόκκινο – red, μαύρο - black)

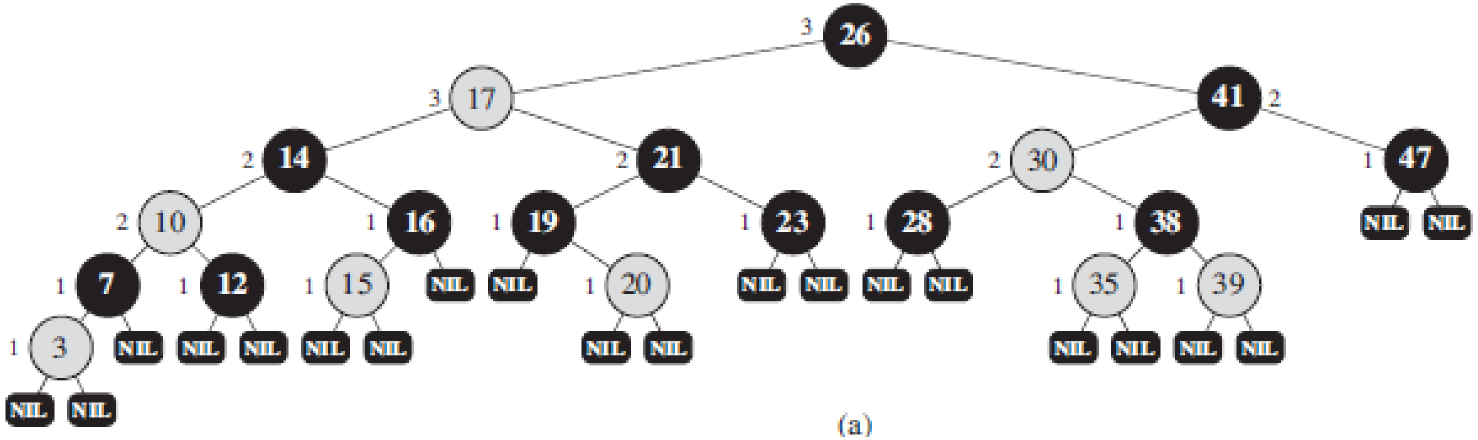
Το δένδρο είναι ισοκατανεμημένο μέσω του περιορισμού του χρώματος των κόμβων από τη ρίζα μέχρι ένα φύλλο

Red Black Trees (2/26)

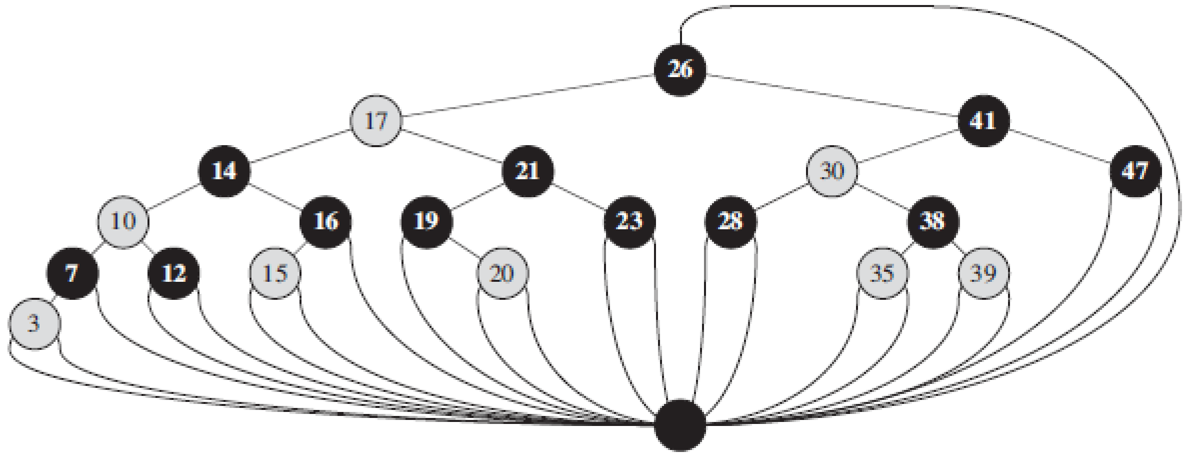
Τα red-black δένδρα ικανοποιούν τις ακόλουθες ιδιότητες:

1. Κάθε κόμβος είναι μαύρος ή κόκκινος
2. Η ρίζα έχει χρώμα μαύρο
3. Κάθε φύλλο είναι μαύρο
4. Αν ένας κόμβος είναι κόκκινος τότε τα παιδιά του είναι και τα δύο μαύρα
5. Για κάθε κόμβο, όλα τα μονοπάτια από τον κόμβο μέχρι τα φύλλα απογόνους περιλαμβάνουν τον ίδιο αριθμό μαύρων κόμβων

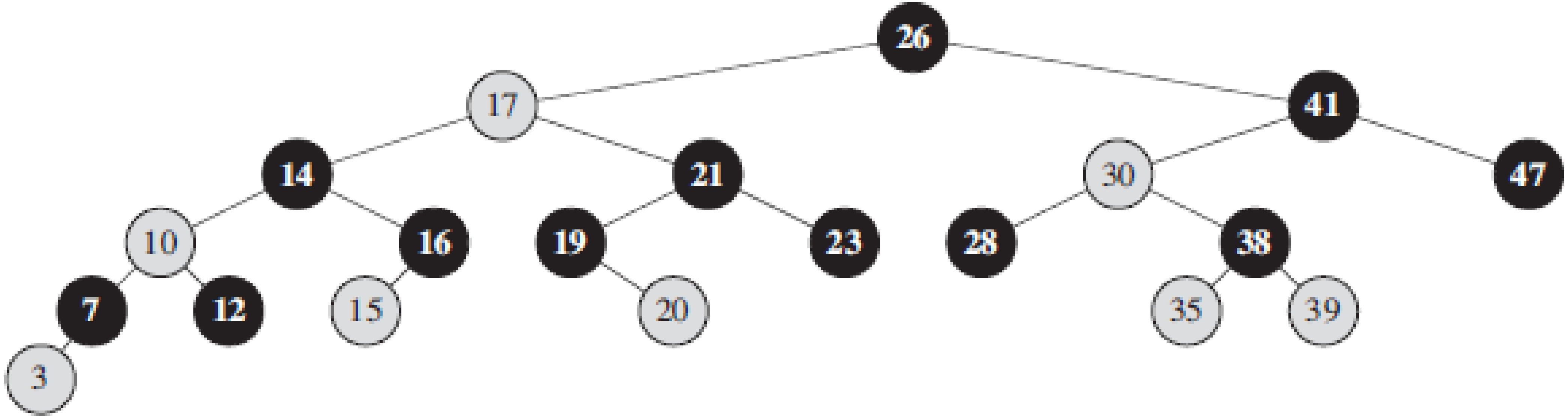
Red Black Trees (3/26)



(a)



Red Black Trees (4/26)



Red Black Trees (5/26)

Ο αριθμός των μαύρων κόμβων σε ένα μονοπάτι από ένα κόμβο προς ένα φύλλο (χωρίς να περιλαμβάνεται ο κόμβος μέσα) ονομάζεται black height – bh(x), x είναι ο κόμβος που εξετάζουμε

Ένα red-black δένδρο με n εσωτερικούς κόμβους έχει ύψος το πολύ 2log(n+1)

Η αναζήτηση, εύρεση μεγίστου / ελαχίστου, εύρεση προηγούμενου / επόμενου κόμβου / κλειδιού ολοκληρώνονται σε χρόνο O(logn)

Red Black Trees (6/26)

Οι αλγόριθμοι TREE-INSERT & TREE-DELETE όταν εκτελούνται σε ένα red-black δένδρο απαιτούν χρόνο O(logn)

Επειδή οι αλγόριθμοι αυτοί τροποποιούν το δένδρο, υπάρχει η πιθανότητα να παριαβάσουμε τα κριτήρια ενός red-black δένδρου

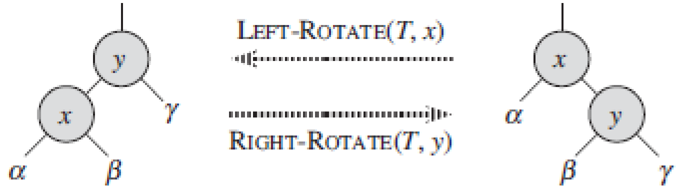
Για να επαναφέρουμε τις ιδιότητες αυτές, πρέπει να αλλάξουμε τα χρώματα σε κάποιους κόμβους όπως επίσης και τους δείκτες προς διάφορους κόμβους

Η αλλαγή στους δείκτες γίνεται με τη βοήθεια της **περιστροφής (rotation)**

Red Black Trees (7/26)

Η περιστροφή διατηρεί τις ιδιότητες του δυαδικού δένδρου

Υπάρχουν δύο είδη περιστροφών: αριστερή και δεξιά



Στην αριστερή περιστροφή, υποθέτουμε πως το δεξιό παιδί του x δεν είναι NIL

Συμμετρική είναι η προσέγγιση για τη δεξιά περιστροφή

Ο χρόνος εκτέλεσης είναι O(1)

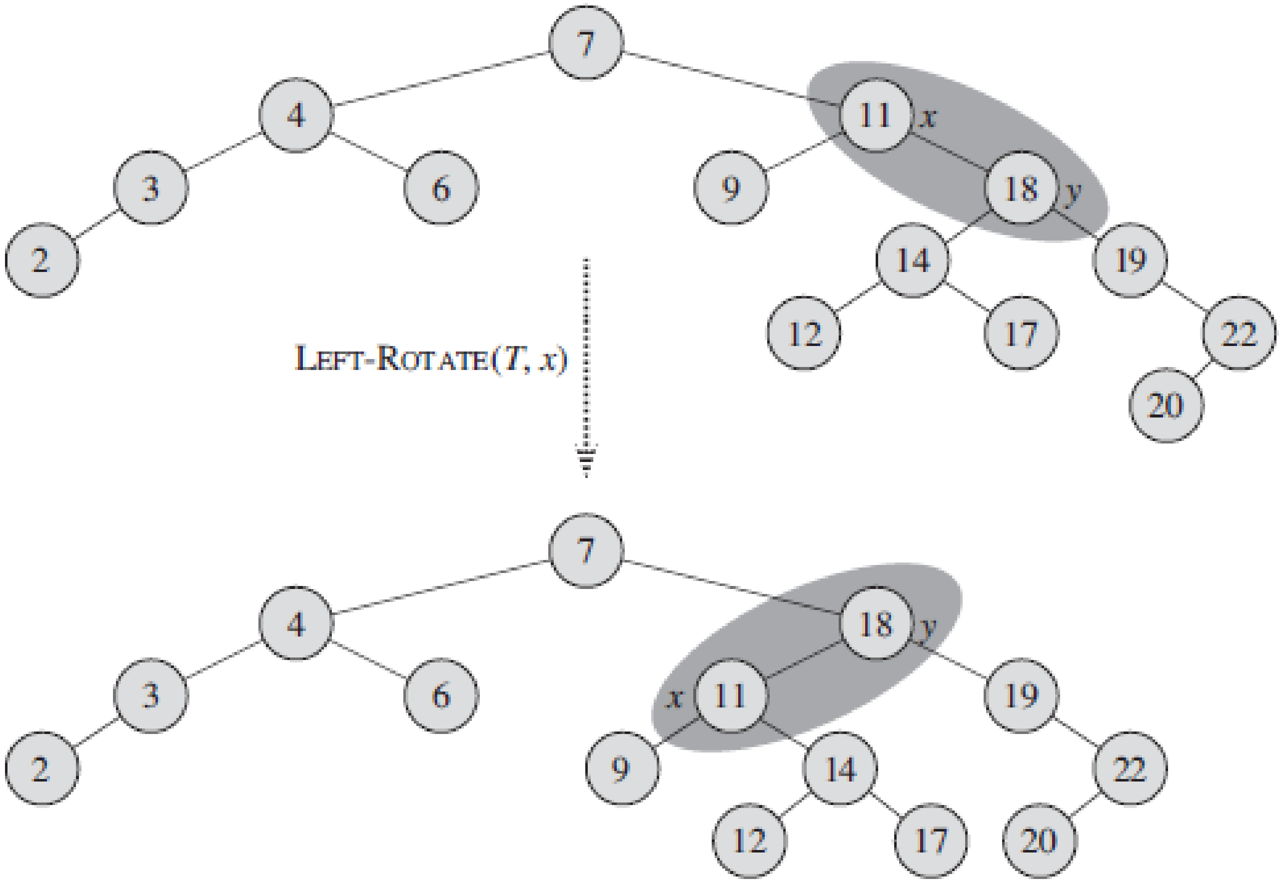
Red Black Trees (8/26)

LEFT-ROTATE(T, x)

```
y = x.right                  // set y
x.right = y.left             // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                    // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                   // put x on y's left
x.p = y
```

Red Black Trees (9/26)

Παράδειγμα



Red Black Trees (10/26)

Η εισαγωγή κόμβου ολοκληρώνεται σε O(logn)

Έστω ότι θέλουμε να εισάγουμε τον κόμβο z

Εισάγουμε τον z όπως και σε ένα ‘κανονικό’ δυαδικό δένδρο και χρωματίζουμε τον z ως κόκκινο

Για να εξασφαλίσουμε πως το δένδρο συνεχίζει να έχει τις αρχικές ιδιότητες εκτελούμε ένα επιπρόσθετο αλγόριθμο (fixup)

Ο fixup αλγόριθμος προβαίνει σε επαναχρωματισμό των κόμβων

Red Black Trees (11/26)

Διαφορές με τον κλασικό αλγόριθμο εισαγωγής:

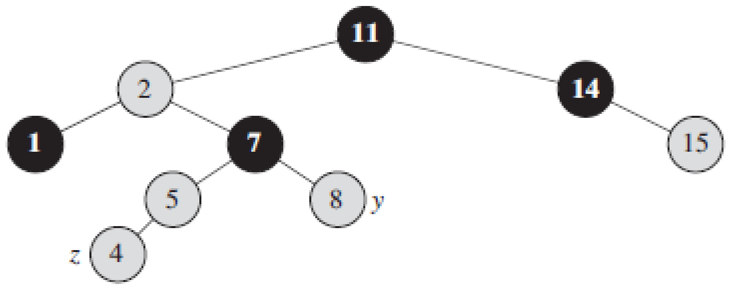
- Οι NIL αναφορές αντικαθίστανται με τον τερματικό κόμβο T.NIL
- Θέτουμε z.left = z.right = T.NIL ώστε να διατηρήσουμε τη δομή του δένδρου
- Χρωματίζουμε ως κόκκινο το νέο κόμβο
- Χρωματίζοντας ως κόκκινο το νέο κόμβο, πιθανόν να παριαβιάσουμε τις ιδιότητες του δένδρου, οπότε καταφεύγουμε στον fixup αλγόριθμο

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Red Black Trees (12/26)

Οι ιδιότητες που μπορεί να παριαβιαστούν είναι:

- Η ρίζα να μην έχει μαύρο χρώμα
  - Ισχύει αν ο z μπει στη ρίζα
- Ένας κόκκινος κόμβος να έχει ένα κόκκινο παιδί
  - Ισχύει αν ο πατρικός του z έχει κόκκινο χρώμα




```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK            // case 1
            y.color = BLACK              // case 1
            z.p.p.color = RED            // case 1
            z = z.p.p                    // case 1
        else if z == z.p.right
                z = z.p                  // case 2
                LEFT-ROTATE(T, z)        // case 2
            z.p.color = BLACK            // case 3
            z.p.p.color = RED            // case 3
            RIGHT-ROTATE(T, z.p.p)       // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red Black Trees (13/26)

Το while διατηρεί ότι:

- Ο κόμβος z είναι κόκκινος
- Αν ο z.p είναι η ρίζα, τότε ο z.p έχει χρώμα μαύρο
- Αν παραβιάζεται κάποια ιδιότητα, τότε παραβιάζεται ακριβώς μια, είτε η 2^{η} ή η 4^{η}

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK              // case 1
            y.color = BLACK                // case 1
            z.p.p.color = RED              // case 1
            z = z.p.p                      // case 1
        else if z == z.p.right
                z = z.p                    // case 2
                LEFT-ROTATE(T, z)          // case 2
            z.p.color = BLACK              // case 3
            z.p.p.color = RED              // case 3
            RIGHT-ROTATE(T, z.p.p)         // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red Black Trees (14/26)

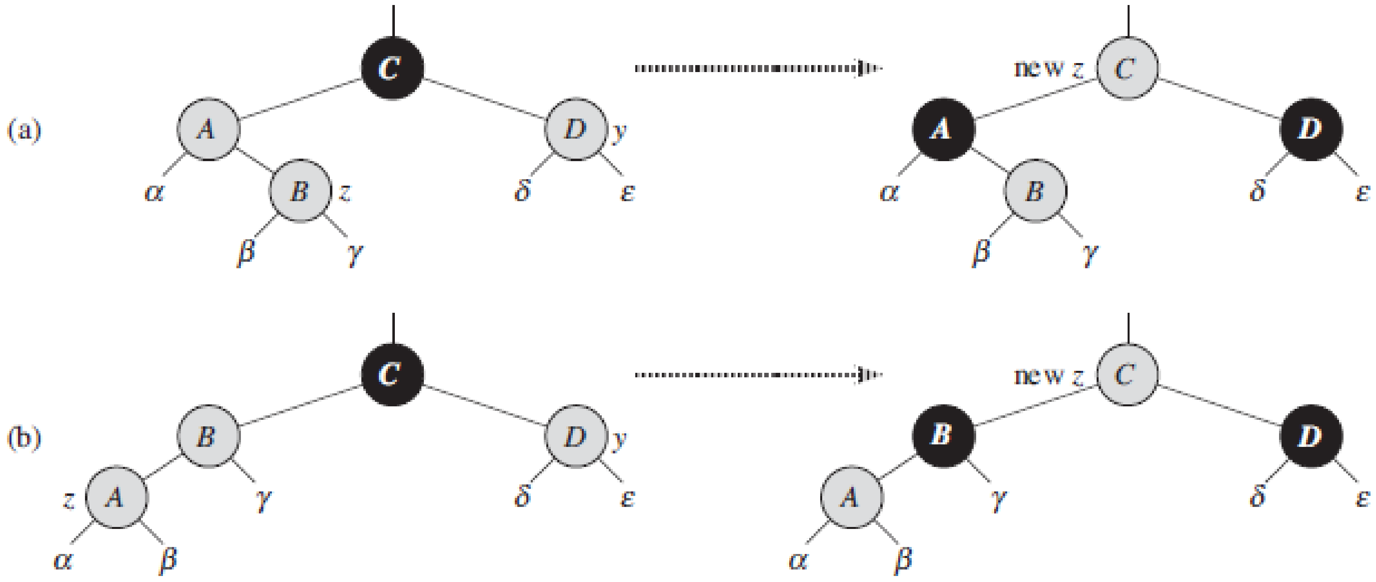
Case 1: Ο ‘θείος’ του z ο κόμβος y είναι κόκκινος

- Αφού ο z.p.p είναι μαύρος, μπορούμε να χρωματίσουμε τους z.p & y ως μαύρους και τον z.p.p ως κόκκινο. Επαναλαμβάνουμε το while για τον z.p.p ως νέο z. Ο δείκτης μετακινείται 2 επίπεδα προς τα πάνω

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK            // case 1
            y.color = BLACK              // case 1
            z.p.p.color = RED            // case 1
            z = z.p.p                    // case 1
        else if z == z.p.right
                z = z.p                  // case 2
                LEFT-ROTATE(T, z)        // case 2
            z.p.color = BLACK            // case 3
            z.p.p.color = RED            // case 3
            RIGHT-ROTATE(T, z.p.p)       // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red Black Trees (15/26)

Case 1: Ο ‘θείος’ του z ο κόμβος y είναι κόκκινος




```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK            // case 1
            y.color = BLACK              // case 1
            z.p.p.color = RED            // case 1
            z = z.p.p                    // case 1
        else if z == z.p.right
                z = z.p                  // case 2
                LEFT-ROTATE(T, z)        // case 2
            z.p.color = BLACK            // case 3
            z.p.p.color = RED            // case 3
            RIGHT-ROTATE(T, z.p.p)       // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red Black Trees (16/26)

Case 2: ο ‘θείος’ του z ο y έχει μαύρο χρώμα και ο z είναι ένα δεξιό παιδί

- Υιοθετούμε μια αριστερή περιστροφή για να μετασχηματίσουμε στην περίπτωση 3

Case 3: ο ‘θείος’ του z ο y έχει μαύρο χρώμα και ο z είναι ένα αριστερό παιδί

- Αφού οι z & z.p έχουν κόκκινο χρώμα, η περιστροφή δεν επηρεάζει ούτε την ιδιότητα bh ούτε την ιδιότητα 5

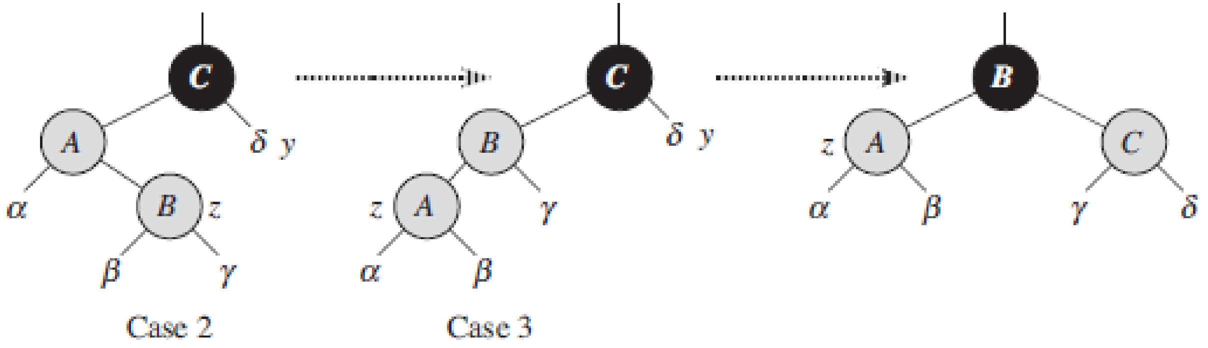
```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK              // case 1
            y.color = BLACK                // case 1
            z.p.p.color = RED              // case 1
            z = z.p.p                      // case 1
        else if z == z.p.right
                z = z.p                    // case 2
                LEFT-ROTATE(T, z)          // case 2
            z.p.color = BLACK              // case 3
            z.p.p.color = RED              // case 3
            RIGHT-ROTATE(T, z.p.p)         // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red Black Trees (17/26)

Case 2: ο 'θείος' του z ο y έχει μαύρο χρώμα και ο z είναι ένα δεξιό παιδί

Case 3: ο 'θείος' του z ο y έχει μαύρο χρώμα και ο z είναι ένα αριστερό παιδί

- Εκτελούμε αλλαγή χρώματος και εκτελούμε δεξιά περιστροφή ώστε να διατηρήσουμε την ιδιότητα 5




```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK              // case 1
            y.color = BLACK                // case 1
            z.p.p.color = RED              // case 1
            z = z.p.p                      // case 1
        else if z == z.p.right
                z = z.p                    // case 2
                LEFT-ROTATE(T, z)          // case 2
            z.p.color = BLACK              // case 3
            z.p.p.color = RED              // case 3
            RIGHT-ROTATE(T, z.p.p)         // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red Black Trees (18/26)

Η διαγραφή κόμβου υλοποιείται σε O(logn)

Βασιζόμαστε στον αλγόριθμο TREE-DELETE των κανονικών δυαδικών δένδρων

Τροποποιούμε την TRANSPLANT ως εξής:

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

Red Black Trees (19/26)

Έστω ότι ο κόμβος προς διαγραφή είναι ο z

Προσπαθούμε να βρούμε ένα κόμβο y ο οποίος μπορεί να προκαλέσει παραβιάσεις των ιδιοτήτων μετά τη διαγραφή

Αν ο z έχει λιγότερα από δύο παιδιά τότε τον διαγράφουμε και τη θέση του παίρνει ο y

Όταν ο z έχει δύο παιδιά, τότε ο y πρέπει να είναι ο επόμενος κόμβος / κλειδί και παόιρνει τη θέση του z στο δένδρο

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Red Black Trees (20/26)

‘Κρατάμε’ επίσης το χρώμα του y πριν την μετακίνησή του και εντοπίζουμε τον κόμβο x που θα πάρει τη θέση ώστε να μην προκαλέσει παραβιάσεις

Μετά τη διαγραφή του z καλούμε ένα αλγόριθμο fixup ώστε να κάνουμε τις απαραίτητες αλλαγές χρωμάτων και περιστροφές

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Red Black Trees (21/26)

O fixup αλγόριθμος διατηρεί τις ιδιότητες 1, 2, 4

Case 1: O ‘αδερφός’ του x o κόμβος w είναι κόκκινος

- Αφού ο w πρέπει να έχει ένα μαύρο παιδί, μπορούμε να αλλάξουμε τα χρώματα των w & x.p και να υλοποιήσουμε μια αριστερή περιστροφή στον x.p
- O νέος ‘αδερφός’ του x είναι ένα από τα παιδιά του w και έχει μαύρο χρώμα
- H περίπτωση μετασχηματίζεται στις 2,3,4

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                // case 1
            x.p.color = RED                // case 1
            LEFT-ROTATE(T, x.p)            // case 1
            w = x.p.right                  // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                  // case 2
            x = x.p                        // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK       // case 3
                w.color = RED              // case 3
                RIGHT-ROTATE(T, w)         // case 3
                w = x.p.right              // case 3
            w.color = x.p.color            // case 4
            x.p.color = BLACK              // case 4
            w.right.color = BLACK          // case 4
            LEFT-ROTATE(T, x.p)            // case 4
            x = T.root                     // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Red Black Trees (22/26)

Case 2: Ο ‘αδερφός’ του x ο κόμβος w είναι μαύρος και τα παιδιά του επίσης

- Αφού ο w και τα παιδιά του έχουν μαύρο χρώμα, παίρνουμε το w ως κόκκινο και επαναλαμβάνουμε το while για τον κόμβο x.p

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK              // case 1
            x.p.color = RED              // case 1
            LEFT-ROTATE(T, x.p)          // case 1
            w = x.p.right                // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                // case 2
            x = x.p                      // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK     // case 3
                w.color = RED            // case 3
                RIGHT-ROTATE(T, w)       // case 3
                w = x.p.right            // case 3
            w.color = x.p.color          // case 4
            x.p.color = BLACK            // case 4
            w.right.color = BLACK        // case 4
            LEFT-ROTATE(T, x.p)          // case 4
            x = T.root                   // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Red Black Trees (23/26)

Case 3: Ο ‘αδερφός’ του x ο κόμβος w είναι μαύρος, το αριστερό παιδί του w είναι κόκκινο και το δεξιό είναι μαύρο

- Αλλάζουμε τα χρώματα του w και του αριστερού παιδιού και εφαρμόζουμε δεξιά περιστροφή στον w
- Ο νέος αδερφός του x έχει μαύρο χρώμα με ένα παιδί κόκκινου χρώματος
- Το σενάριο μετασχηματίστηκε στην περίπτωση 4

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK            // case 1
            x.p.color = RED            // case 1
            LEFT-ROTATE(T, x.p)        // case 1
            w = x.p.right              // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                  // case 2
            x = x.p                        // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK   // case 3
                w.color = RED          // case 3
                RIGHT-ROTATE(T, w)     // case 3
                w = x.p.right          // case 3
            w.color = x.p.color            // case 4
            x.p.color = BLACK              // case 4
            w.right.color = BLACK          // case 4
            LEFT-ROTATE(T, x.p)            // case 4
            x = T.root                     // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Red Black Trees (24/26)

Case 3: Ο ‘αδερφός’ του x ο κόμβος w είναι μαύρος, και το δεξιό του παιδί είναι κόκκινο

- Αλλάζουμε χρώματα και εφαρμόζουμε αριστερή περιστροφή στον x.p
- Θέτουμε τον x ως τη ρίζα για να τερματίσουμε τον αλγόριθμο

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK           // case 1
            x.p.color = RED           // case 1
            LEFT-ROTATE(T, x.p)       // case 1
            w = x.p.right             // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                   // case 2
            x = x.p                         // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK  // case 3
                w.color = RED               // case 3
                RIGHT-ROTATE(T, w)    // case 3
                w = x.p.right               // case 3
            w.color = x.p.color             // case 4
            x.p.color = BLACK               // case 4
            w.right.color = BLACK           // case 4
            LEFT-ROTATE(T, x.p)             // case 4
            x = T.root                      // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Red Black Trees (25/26)

Εφαρμογές

- time-sensitive applications
- computational geometry
- completely Fair Scheduler
- keep track of the virtual memory segments
- Associative Data structures (for e.g C++ STL uses RB tree internally to implement Set and Map)
- TreeMap (Java) uses Red-Black tree as its backend

Red Black Trees (26/26)

Demo

https://www.cs.usfca.edu/~galles/visualization/BST.html

2-3 Trees

2-3 Trees (1/6)

Μια άλλη ιδέα για την κατασκευή ισοκατανεμημένων δένδρων είναι να επιτρέψουμε πολλαπλά κλειδιά σε κάθε κόμβο

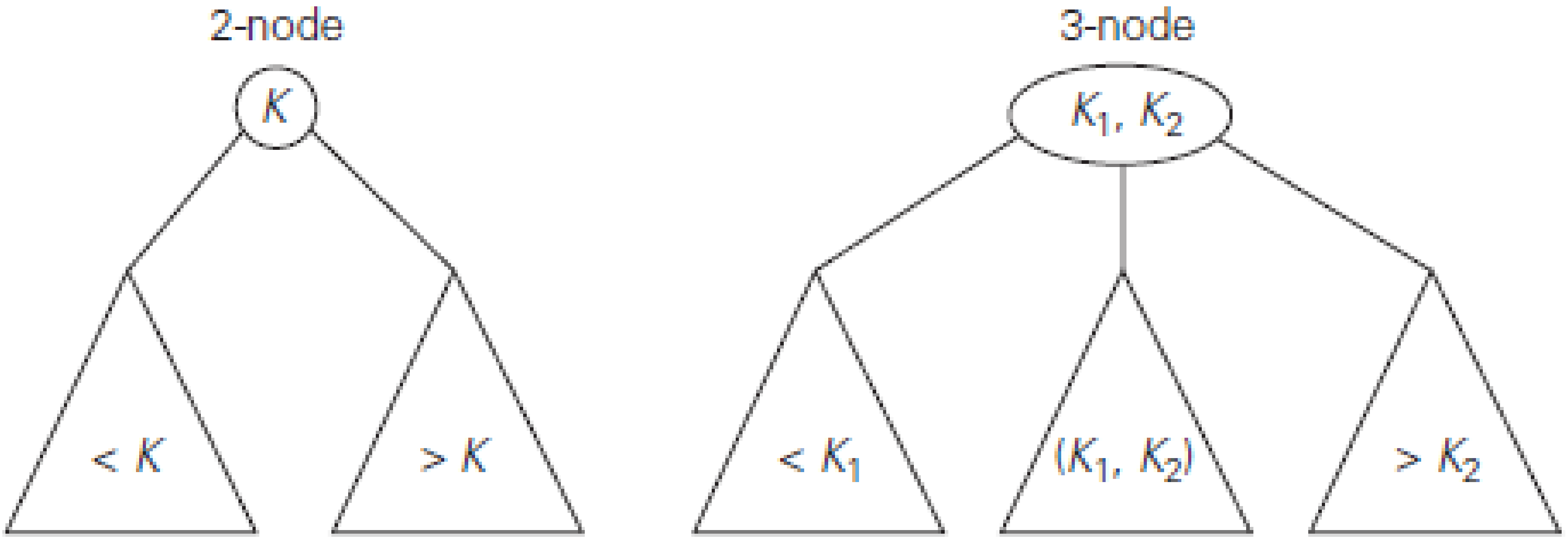
Η πιο απλή υλοποίηση αυτής της ιδέας είναι τα 2-3 δένδρα

Αυτού του είδους τα δένδρα έχουν δύο είδη κόμβων: 2-nodes & 3-nodes

- 2-nodes: έχουν ένα κλειδί K και δύο παιδιά, το αριστερό είναι ρίζα ενός υποδένδρου του οποίου τα στοιχεία είναι μικρότερα από το K και το δεξιό παιδί είναι ρίζα ενός υποδένδρου του οποίου τα στοιχεία είναι μεγαλύτερα του K
- 3-nodes: έχουν δύο κλειδιά K1, K2 (K1<K2) και τρία παιδιά, το τελευταίο αριστερά είναι ρίζα ενός υποδένδρου του οποίου τα στοιχεία είναι μικρότερα από το K1, το μεσαίο είναι ρίζα υποδένδρου με στοιχεία ανάμεσα στα K1, K2 και το δεξιό παιδί είναι ρίζα ενός υποδένδρου του οποίου τα στοιχεία είναι μεγαλύτερα του K2

2-3 Trees (2/6)

Παράδειγμα



2-3 Trees (3/6)

Άλλη ιδιότητα είναι τα φύλλα να τοποθετούνται στο ίδιο επίπεδο

Η αναζήτηση στα 2-3 δένδρα γίνετια ως εξής:

- Ξεκινάμε από τη ρίζα
- Αν πρόκειται για ένα 2-κόμβο, τότε κάνουμε αναζήτηση όπως στα κλασικά δυαδικά δένδρα αναζήτησης
- Αν η ρίζα είναι ένας 3-κόμβος, τότε το πολύ μετά από δύο συγκρίσεις αν το στοιχείο υπάρχει στο δένδρο

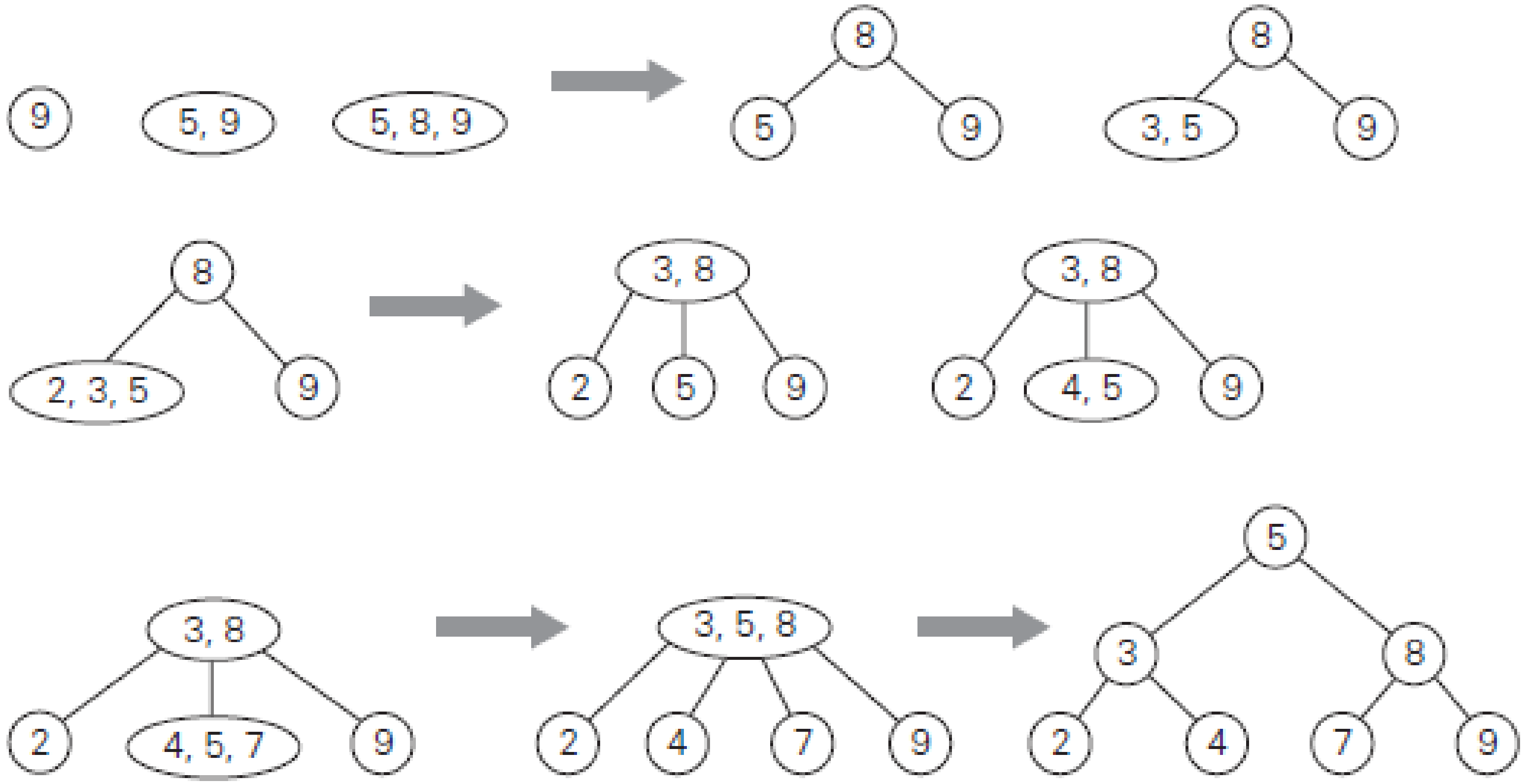
2-3 Trees (4/6)

Εισαγωγή κόμβου

- Εισάγουμε ως φύλλο το νέο κόμβο
- Το κατάλληλο φύλλο εντοπίζεται μετά από αναζήτηση για το κλειδί K
- Αν το φύλλο είναι ένας 2-κόμβος, εισάγουμε το κλειδί στον κόμβο στην κατάλληλη θέση
- Αν οκόμβος είναι ένας 3-κόμβος, διασπάμε τον κόμβο σε δύο κόμβους: το μικρότερο από τα κλειδιά πάει στο πρώτο φύλλο, το μεγαλύτερο πάει στο δεύτερο φύλο και το μεσαίο προωθείται στον πατρικό κόμβο
- Αν ο πατρικός κόμβος είναι ένας 3-κόμβος, η διάσπαση θα πρέπει να συνεχιστεί ώστε το δένδρο να διατηρεί τις αρχικές ιδιότητες

2-3 Trees (5/6)

Παράδειγμα εισαγωγής κόμβων



2-3 Trees (6/6)

Σχετικά με το ύψος των 2-3 δένδρων έχουμε:

- Ένα 2-3 δένδρο με τα ελάχιστα κλειδιά αποτελείται αποκλειστικά από 2-κόμβους και συνεπώς ισχύει:
  $n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1$
- Οπότε $h \leq \log_2(n+1) - 1$
- Ένα δένδρο με το μέγιστο πλήθος κλειδιών αποτελείται αποκλειστικά από 3-κόμβους και συνεπώς ισχύει:
  $n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$
- Οπότε $h \geq \log_3(n+1) - 1$
- Συνεπώς:
  $\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$

$$\mathbf{1 + 3 + \cdots + 3^h = \frac{3^{h+1} - 1}{2}}$$

Hash Tables

Hash Tables (1/24)

Η έννοια του **κατακερματισμού (hashing)** βασίζεται στην ιδέα της διασποράς κλειδιών σε ένα μονοδιάστατο πίνακα που ονομάζεται **πίνακας κατακερματισμού (hash table)**

Η κατανομή γίνεται μέσα από την εφαρμογή μιας συνάρτησης h που ονομάζεται **συνάρτηση κατακερματισμού (hash function)**

Για κάθε κλειδί, η συνάρτηση αποδίδει ένα ακέραιο σε ένα διάστημα [0..m-1] που ονομάζεται **διεύθυνση κατακερματισμού (hash address)**

Παράδειγμα: h(K)=K mod m

Hash Tables (2/24)

Απαιτήσεις για την hash function

- Το μέγεθος του hash table δεν πρέπει να είναι πολύ μεγάλο συγκρινόμενο με το πλήθος των κλειδιών προς κατακερματισμό και κατά συνέπεια να μην διακινδυνεύει την αύξηση του χρόνου εκτέλεσης
- Μια hash function πρέπει να ισοκατανέμει (όσο αυτό γίνεται) τα κλειδιά σε όλες τις θέσεις του πίνακα
- Μια hash function θα πρέπει να υπολογίζεται εύκολα

Hash Tables (3/24)

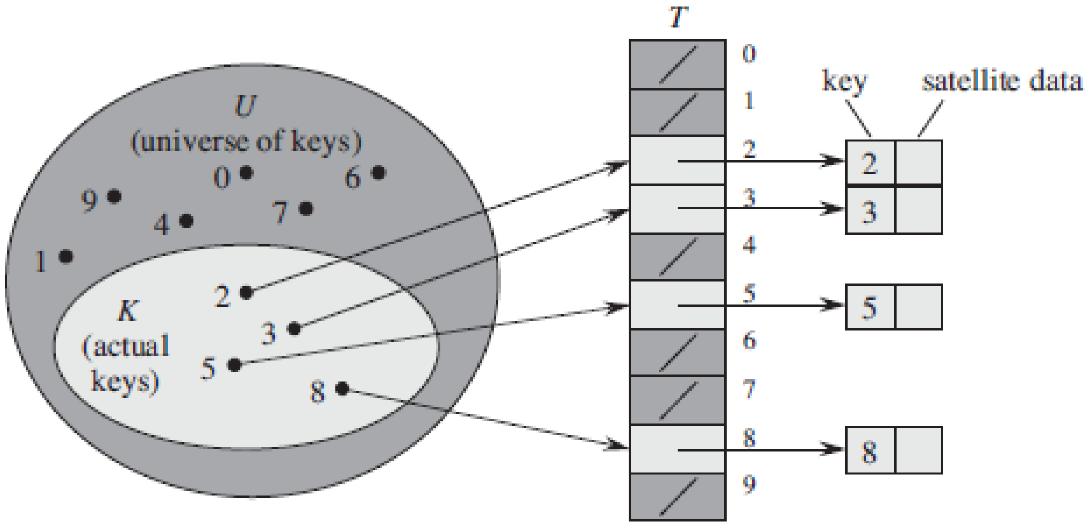
Για ένα hash table T[0..m-1] σε κάθε θέση (**slot**) του οποίου αντιστοιχεί ένα κλειδί k του περιβάλλοντός μας U μπορούμε να εκτελέσουμε τις ακόλουθες ενέργειες:

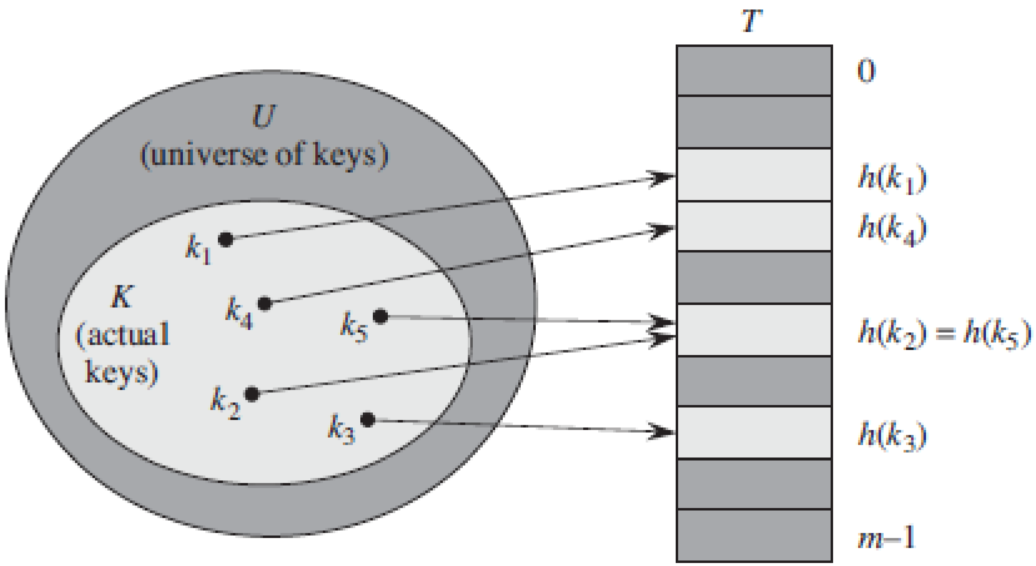
```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
    T[x.key] = NIL
```

Κάθε μια από τις ενέργειες αυτές απαιτεί O(1) χρόνο

Hash Tables (4/24)

Παράδειγμα





Hash Tables (5/24)

Πρόβλημα

- Δύο κλειδιά μπορεί να κατακερματιστούν στην ίδια θέση του πίνακα
  - **Σύγκρουση (collision)**

Προσπαθούμε να βρούμε κατάλληλη και αποδοτική hash function

Εφαρμόζουμε διάφορες αποδοτικές τεχνικές

Hash Tables (6/24)

Λύση: chaining

Τοποθετούμε τα κλειδιά που κατακερματίζονται στην ίδια θέση του πίνακα σε μια συνδεδεμένη λίστα

Κάθε θέση του πίνακα περιέχει ένα δείκτη προς την κορυφή της λίστας

Οι λειτουργίες γίνονται ως εξής:

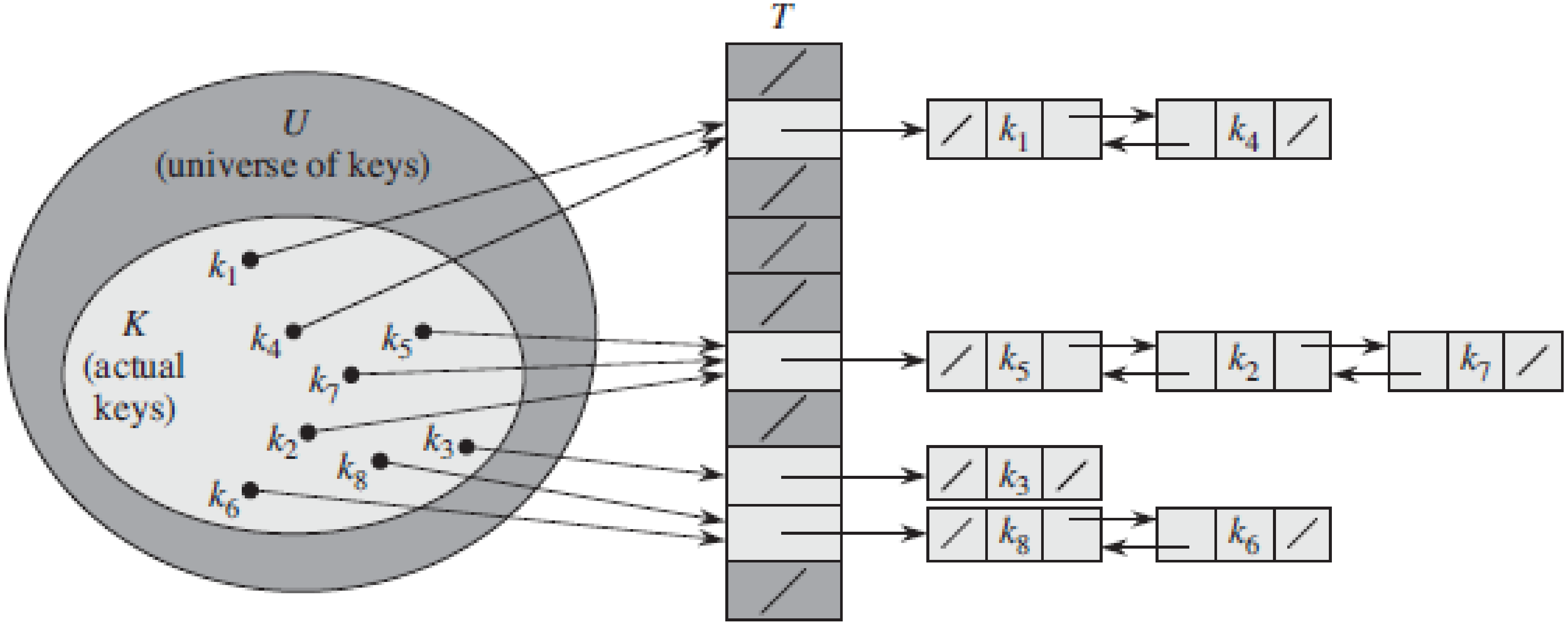
CHAINED-HASH-INSERT(T, x)
insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
delete x from the list $T[h(x.key)]$

Hash Tables (7/24)

Παράδειγμα



Hash Tables (8/24)

Ο χρόνος εισαγωγής στη χειρότερη περίπτωση είναι O(1)

Ο χρόνος αναζήτησης στη χειρότερη περίπτωση εξαρτάται από το μέγεθος της λίστας

Ορίζουμε τον **παράγοντα φόρτου (load factor)** του T ως το λόγο n/m όπου n είναι τα στοιχεία που πρόκειται να κατακερματιστούν και m είναι οι θέσεις του T

Η χειρότερη περίπτωση περιλαμβάνει την τοποθέτηση των n στοιχείων στην ίδια θέση

Η αναζήτηση θα γίνει σε Θ(n) επιπλέον του χρόνου υπολογισμού της συνάρτησης κατακερματισμού

Hash Tables (9/24)

Η μέση περίπτωση περιλαμβάνει την κατανομή των n κλειδιών σε διάφορες θέσεις του T

Αν κάθε κλειδί μπορεί με την ίδια πιθανότητα να τοποθετηθεί σε οποιαδήποτε θέση του T τότε έχουμε τον απλό ομοιόμορφο κατακερματισμό (simple uniform hashing)

Έστω n_j το μέγεθος της λίστας της θέσης T[j]

Συνεπώς έχουμε $n = n_0 + n_1 + \ldots + n_{m-1}$

Επίσης $E[n_j] = \alpha = n/m$

Hash Tables (10/24)

Η αναζήτηση εξαρτάται από τον αναμενόμενο πλήθος των στοιχείων σε κάθε λίστα

Θεώρημα 1: Σε ένα πίνακα κατακερματισμού T μια ανεπιτυχής αναζήτηση απαιτεί στη μέση περίπτωση χρόνο Θ(1+α) (O(1) είναι ο υπολογισμός του h(k))

Απόδειξη

Κάθε κλειδί κατακερματίζεται σε κάθε θέση με την ίδια πιθανότητα. Ο αναμενόμενος χρόνος για την ανεπιτυχή αναζήτηση είναι ίσος με τον αναμενόμενο χρόνο για να φτάσουμε στο τέλος της λίστας T[h(k)] με k να είναι το κλειδί που ψάχνουμε. Η συγκεκριμένη λίστα έχει αναμενόμενο μέγεθος nh(k). Οπότε ο αναμενόμενος αριθμός στοιχείων στα οποία ψάχνουμε είναι α.

Hash Tables (11/24)

Θεώρημα 2: Σε ένα πίνακα κατακερματισμού T, στη μέση περίπτωση, ο χρόνος επιτυχούς αναζήτησης είναι ίσος με Θ(1+α)

Απόδειξη

Έστω ότι έχουμε την ίδια πιθανότητα να αναζητήσουμε το κάθε ένα από τα n κλειδιά. Έστω x το στοιχείο που ψάχνουμε. Αφού κάθε νέο στοιχείο μπαίνει στην κορυφή της λίστας, αυτά τα στοιχεία θα μπαίνουμε προστά από το x και συνεπώς θα πρέπει να ψάχνουμε στο πλήθος των στοιχείων πριν από το x συν 1.

Πρέπει να βρούμε το αναμενόμενο πλήθος των στοιχείων που θα εξετάσουμε.

Hash Tables (12/24)

Απόδειξη (συνέχεια)

Αν x_i είναι το i στοιχείο που εισάγεται στον πίνακα και $k_i = x_i$.key. Ορίζουμε την τυχαία μεταβλητή $X_{ij}=I\{h(k_i)=h(k_j)\}$. Αφού υιοθετούμε ομοιόμορφη κατανομή ισχύει $Pr\{h(k_i)=h(k_j)\}=1/m$ και $E[X_{ij}]=1/m$.

Ο αναμενόμενος αριθμός των στοιχείων που εξετάζονται σε μια επιτυχή αναζήτηση είναι:

Hash Tables (13/24)

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)\\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)
\end{aligned}
$$

$$
\begin{aligned}
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)\\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)\\
&= 1+\frac{n-1}{2m}\\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

$\alpha=n/m \Rightarrow m=n/\alpha$

$$\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$$

Hash Tables (14/24)

Οι περισσότερες hash functions υποθέτουν πως τα κλειδιά είναι φυσικοί αριθμοί

Αν τα κλειδιά δεν είναι φυσικοί αριθμοί τότε εφαρμόζουμε μεθόδους για τη μετατροπή τους σε φυσικούς αριθμούς

Παράδειγμα: ένα σύνολο χαρακτήρων μπορεί να αναπαρασταθεί σαν μια ακολουθία ακεραίων αριθμών

Hash Tables (15/24)

The division method

- Υιοθετούμε το υπόλοιπο της διαίρεσης του κλειδιού k με το πλήθος των θέσεων m
- h(k) = k mod m
- Συχνά αποφεύγουμε κάποιες τιμές για το m
- Για παράδειγμα το m δεν πρέπει να είναι πολλαπλάσιο του 2 αφού $m = 2^p$ και το h(k) είναι τα p κατώτερα bits του k
- Καλή επιλογή επιλογή για το m είναι ένας πρώτος αριθμός όχι κοντά σε μια δύναμη του 2

Hash Tables (16/24)

The multiplication method

- Υιοθετεί δύο βήματα
  - Πολλαπλασιάζουμε το k με μια σταθερά A και παίρνουμε ένα τμήμα του γινομένου
  - Πολλαπλασιάζουμε το αποτέλεσμα με το m και παίρνουμε το floor του αποτελέσματος
- h(k) = floor(m (k A mod 1)), το k A mod 1 απεικονίζει το k A – floor(k A)
- Εδώ η τιμή του m δεν είναι κρίσιμη οπότε την επιλέγουμε να είναι πολλαπλάσιο του 2

Hash Tables (17/24)

Όταν χρησιμοποιούμε μια σταθερή hash function κάποιος μπορεί να επιλέξει κλειδιά ώστε όλα να κατακερματιστούν στην ίδια θέση

Η λύση είναι να επιλέξουμε μια συνάρτηση που να είναι ανεξάρτητη από τα κλειδιά

Η προσέγγιση αυτή ονομάζεται **universal hashing**

Κατά την εκτέλεση επιλέγουμε τυχαία τη συνάρτηση μέσα από ένα σύνολο συναρτήσεων

Αφού γίνεται τυχαία επιλογή, τα αποτελέσματα μπορεί να είναι διαφορετικά ακόμα και για την ίδια είσοδο

Hash Tables (18/24)

Open addressing

- Κάθε στοιχείο του T περιέχει είτε ένα κλειδί ή NIL
- Στην αναζήτηση ψάχνουμε τα στοιχεία του πίνακα ώστε είτε να βρούμε το κλειδί που επιθυμούμε ή να καταλήξουμε ότι το κλειδί δεν υπάρχει
- Δεν χρησιμοποιούμε λίστες εκτός του πίνακα όπως γίνεται με την μέθοδο του chaining
- Ο T μπορεί να γεμίσει, οπότε δεν μπορούν να μπουν νέα στοιχεία

Hash Tables (19/24)

Εισαγωγή κλειδιού στο open addressing

- Εξετάζουμε τον T ώστε να βρούμε μια κενή θέση (probing)
- Η ακολουθία των θέσεων στις οποίες κάνουμε αναζήτηση εξαρτάται από το κλειδί που πρόκειται να εισαχθεί
- Επεκτείνουμε τη συνάρτηση κατακερματισμού ώστε να περιλαμβάνει τον αριθμό αναζήτησης ως δεύτερη είσοδο
- Απαιτείται μια ακολουθία αναζήτησης ως εξής: <h(k,0), h(k,1), ..., h(k,m-1)> που είναι ένα permutation των <1,2, ..., m-1>
- Κάθε θέση εξετάζεται ισοπίθανα

Hash Tables (20/24)

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

Hash Tables (21/24)

Αλγόριθμος αναζήτησης

- Αναζητά στις θέσεις που εξετάζει ο αλγόριθμος εισαγωγής όταν εισάγεται ένα κλειδί k
- Η αναζήτηση τερματίζεται όταν φτάσουμε σε μια κενή θέση

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

Hash Tables (22/24)

Linear probing

- Χρησιμοποιούμε μια βοηθητική συνάρτηση κατακερματισμού
- h(k,i) = (h’(k) + i) mod m
- Αρχικά αναζητούμε στο T[h’(k)]
- Υποφέρει από την **πρωταρχική συσταδοποίηση (primary clustering)**
  - Μεγάλες εκτελέσεις σε κατειλλημένες θέσεις αυξάνουν το μέσο χρόνο
  - Άδειες θέσεις ακολουθούνται από i γεμάτες θέσεις με πιθανότητα (i+1)/m

Hash Tables (23/24)

Quadratic probing

- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- Η h’ είναι μια βοηθητική συνάρτηση κατακερματισμού
- Τα c_1, c_2 είναι βοηθητικές σταθερές
- Αποδίδει καλύτερα από το linear probing αλλά για να γίνει η πλήρης χρήση του T θα πρέπει να περιορίσουμε τις σταθερές και το m
- Αν δύο κλειδιά έχουν την ίδια αρχική θέση αναζήτησης, τότε η ακολουθία τους θα είναι επίσης η ίδια
- Το φαινόμενο αυτό οδηγεί στη **δευτερεύουσα συσταδοποίηση (secondary clustering)**

Hash Tables (24/24)

Double hashing

- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$
- Οι h_1 kai h_2 είναι βοηθητικές συναρτήσεις κατακερματισμού
- Η αρχική θέση είναι η $T[h_1(k)]$
- Η ακολουθία αναζήτησης εξαρτάται δύο φορές στο κλειδί k
- Η τιμή $h_2(k)$ πρέπει να είναι ένας πρώτος σε σχέση με το μέγεθος m
  - Μια καλή λύση είναι να θέσουμε το m ως πολλαπλάσιο του 2 και την h_2 να βγάζει περιττούς αριθμούς
  - Άλλη λύση είναι να θέσουμε
    - $h_1(k) = k \bmod m$
    - $h_2(k) = 1 + (k \bmod m')$, m' επιλέγεται να είναι κατά τι μικρότερο του m ($m'=m-1$)

Ασκήσεις

Linear Probing

- Πρόκειται για την πιο απλή τεχνική
- Αν μια τιμή αποθηκευτεί στη θέση h(k) τότε υιοθετούμε την ακόλουθη συνάρτηση κατακερματισμού για να επιλύσουμε τη σύγκρουση

$$h(k,i)=[h'(k)+i] \bmod m$$

$$h'(k)=k \bmod m$$

- Το i απεικονίζει τον probe αριθμό που κινείται από 0 μέχρι m-1
- Για ένα κλειδί k υπολογίζεται η θέση h’(k)=k mod m διότι την πρώτη φορά έχουμε i=0
- Αν η θέση είναι ελεύθερη, τότε το κλειδί αποθηκεύεται σε αυτή τη θέση
- Αν όχι, τότε υπολογίζουμε το δεύτερο probe [h’(k)+1] mod m

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων
- h’(k) = k mod m, m = 10

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

Step 1 Key = 72

$$h(72, 0) = (72 \bmod 10 + 0) \bmod 10 = (2) \bmod 10 = 2$$

Since T[2] is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2 Key = 27

$$h(27, 0) = (27 \bmod 10 + 0) \bmod 10 = (7) \bmod 10 = 7$$

Since T[7] is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

Step 3 Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10
= (6) mod 10
= 6

Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10
= (4) mod 10
= 4

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72. 27. 36. 24. 63. 81. 92. 101 σε πίνακα 10 θέσεων

Step 5 Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10
= (3) mod 10
= 3

Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10
= (1) mod 10
= 1

Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

Step 7 Key = 92

$h(92, 0) = (92 \bmod 10 + 0) \bmod 10$
$= (2) \bmod 10$
$= 2$

Key = 92

$h(92, 1) = (92 \bmod 10 + 1) \bmod 10$
$= (2 + 1) \bmod 10$
$= 3$

Key = 92

$h(92, 2) = (92 \bmod 10 + 2) \bmod 10$
$= (2 + 2) \bmod 10$
$= 4$

Key = 92

$h(92, 3) = (92 \bmod 10 + 3) \bmod 10$
$= (2 + 3) \bmod 10$
$= 5$

Since T[5] is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

- Μπορείτε να εκτελέσετε τον κατακερματισμό του τελευταίου αριθμού;

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Linear Probing

- Αναζήτηση τιμής
  - Η διαδικασία είναι η ίδια με τη διαδικασία εισαγωγής της τιμής στον πίνακα
  - Επανυπολογίζουμε την τιμή της θέσης του πίνακα
  - Αν η τιμή που ψάχνουμε δεν ταιριάξει με την τιμή που πήραμε από τη θέση τότε εκκινούμε μια σειριακή αναζήτηση
  - Τα αποτελέσματα της διαδικασίας μπορεί να είναι:
    - Να βρεθεί η τιμή
    - Να καταλήξουμε σε θέση που έχει το -1 οπότε η τιμή δεν υπάρχει
    - Να φτάσουμε στο τέλος του πίνακα

Linear Probing

- Άσκηση:
- Στον επόμενο πίνακα σε ποια θέση θα τοποθετηθεί το 458 όταν h(k)=k mod 10;

				104	375	936		738	
0	1	2	3	4	5	6	7	8	9

				104	375	936		738	458
0	1	2	3	4	5	6	7	8	9

Linear Probing

- Άσκηση:
- Στον επόμενο πίνακα σε ποια θέση θα τοποθετηθεί το 194 όταν h(k)=k mod 10;

		342		554		546			799
0	1	2	3	4	5	6	7	8	9

		342		554	194	546			799
0	1	2	3	4	5	6	7	8	9

Linear Probing

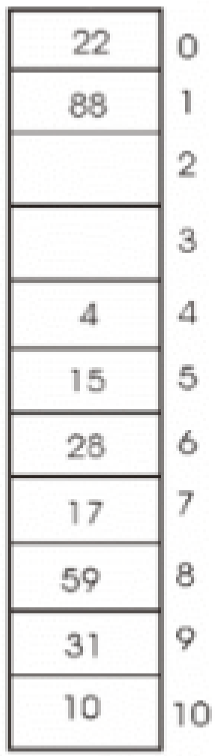
- Άσκηση:
- Στον επόμενο πίνακα σε ποια θέση θα τοποθετηθεί το 693 όταν h(k)=k mod 10;

		512	753	744		866	267	468	
0	1	2	3	4	5	6	7	8	9

		512	753	744	693	866	267	468	
0	1	2	3	4	5	6	7	8	9

Linear Probing

- Άσκηση:
- Να κατακερματίσετε τα κλειδιά: 10, 22, 31, 4, 15, 28, 17, 88, 59 σε πίνακα μεγέθους 11



Quadratic Probing

- Σε αυτή την τεχνική, αν μια τιμή υπάρχει ήδη στη θέση h(k) τότε ψάχνουμε μια ελεύθερη θέση με την ακόλουθη συνάρτηση κατακερματισμού:

$$h(k,i)=[h'(k) + c_1 i + c_2 i^2] \bmod m$$

$$h'(k)=k \bmod m$$

- Τα c_1, c_2 είναι σταθερές

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 1

$$
\begin{aligned}
\text{Key} &= 72 \\
h(72, 0) &= [72 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10 \\
&= [72 \bmod 10] \bmod 10 \\
&= 2 \bmod 10 \\
&= 2
\end{aligned}
$$

Since `T[2]` is vacant, insert the key 72 in `T[2]`. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 2 Key = 27

h(27, 0) = [27 mod 10 + 1 × 0 + 3 × 0] mod 10
= [27 mod 10] mod 10
= 7 mod 10
= 7

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step 3 Key = 36

h(36, 0) = [36 mod 10 + 1 × 0 + 3 × 0] mod 10
= [36 mod 10] mod 10
= 6 mod 10
= 6

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 4

$$\begin{aligned} \text{Key} &= 24 \\ h(24, 0) &= [24 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10 \\ &= [24 \bmod 10] \bmod 10 \\ &= 4 \bmod 10 \\ &= 4 \end{aligned}$$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5

$$\begin{aligned} \text{Key} &= 63 \\ h(63, 0) &= [63 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10 \\ &= [63 \bmod 10] \bmod 10 \\ &= 3 \bmod 10 \\ &= 3 \end{aligned}$$

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 6

$$
\begin{aligned}
\text{Key} &= 81 \\
h(81,0) &= [81 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10 \\
&= [81 \bmod 10] \bmod 10 \\
&= 81 \bmod 10 \\
&= 1
\end{aligned}
$$

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 7

Key = 101

h(101,0) = [101 mod 10 + 1 × 0 + 3 × 0] mod 10
= [101 mod 10 + 0] mod 10
= 1 mod 10
= 1

Key = 101

h(101,0) = [101 mod 10 + 1 × 1 + 3 × 1] mod 10
= [101 mod 10 + 1 + 3] mod 10
= [101 mod 10 + 4] mod 10
= [1 + 4] mod 10
= 5 mod 10
= 5

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

Quadratic Probing

- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 11 θέσεων: 10, 22, 31, 4, 15, 28, 17, 88, 59 (c_1=1, c_2=3)

22	0
	1
88	2
17	3
4	4
	5
28	6
59	7
15	8
31	9
10	10

Quadratic Probing

- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 11 θέσεων: 12,44,13,88,23,94,11,39,20,16 (c_1=1, c_2=3) με τη hash function h(k)=(2k+5) mod m

23	**20**	**11**		**16**	**44**	**94**	**12**	**88**	**13**	**39**

Double Hashing

- Η τεχνική αυτή εξάγει μια τιμή και στη συνέχεια επαναληπτικά προχωρά μέχρι να βρει κενή θέση όπως ακριβώς και οι δύο προηγούμενες τεχνικές
- Υιοθετεί δύο συναρτήσεις κατακερματισμού που είναι ανεξάρτητες
- Η δεύτερη υπολογίζει το βήμα μετακίνησης πάνω στις θέσεις του πίνακα
- Η τελική συνάρτηση κατακερματισμού έχει ως εξής:

$$h(k,i)=[h_1(k) + ih_2(k)] \bmod m$$

$$h_1(k)=k \bmod m$$

$$h_2(k)=k \bmod m'$$

Double Hashing

- Το m’ επιλέγεται να είναι μικρότερο του m
- Μπορούμε να επιλέξουμε m’=m-1, m’=m-2

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με $h_1 = (k \bmod 10)$ και $h_2 = (k \bmod 8)$.

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με $h_1 = (k \bmod 10)$ και $h_2 = (k \bmod 8)$.

Step 1 Key = 72

$$\begin{aligned} h(72, 0) &= [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10 \\ &= [2 + (0 \times 0)] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2 Key = 27

$$\begin{aligned} h(27, 0) &= [27 \bmod 10 + (0 \times 27 \bmod 8)] \bmod 10 \\ &= [7 + (0 \times 3)] \bmod 10 \\ &= 7 \bmod 10 \\ &= 7 \end{aligned}$$

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με $h_1 = (k \bmod 10)$ και $h_2 = (k \bmod 8)$.

Step 3 Key = 36

$$\begin{aligned} h(36, 0) &= [36 \bmod 10 + (0 \times 36 \bmod 8)] \bmod 10 \\ &= [6 + (0 \times 4)] \bmod 10 \\ &= 6 \bmod 10 \\ &= 6 \end{aligned}$$

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

$$\begin{aligned} h(24, 0) &= [24 \bmod 10 + (0 \times 24 \bmod 8)] \bmod 10 \\ &= [4 + (0 \times 0)] \bmod 10 \\ &= 4 \bmod 10 \\ &= 4 \end{aligned}$$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

Step 5 Key = 63

h(63, 0) = [63 mod 10 + (0 × 63 mod 8)] mod 10
= [3 + (0 × 7)] mod 10
= 3 mod 10
= 3

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

h(81, 0) = [81 mod 10 + (0 × 81 mod 8)] mod 10
= [1 + (0 × 1)] mod 10
= 1 mod 10
= 1

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με $h_1 = (k \text{ mod } 10)$ και $h_2 = (k \text{ mod } 8)$.

Step 7

Key = 92

$$
\begin{aligned}
h(92, 0) &= [92 \text{ mod } 10 + (0 \times 92 \text{ mod } 8)] \text{ mod } 10 \\
&= [2 + (0 \times 4)] \text{ mod } 10 \\
&= 2 \text{ mod } 10 \\
&= 2
\end{aligned}
$$

Key = 92

$$
\begin{aligned}
h(92, 1) &= [92 \text{ mod } 10 + (1 \times 92 \text{ mod } 8)] \text{ mod } 10 \\
&= [2 + (1 \times 4)] \text{ mod } 10 \\
&= (2 + 4) \text{ mod } 10 \\
&= 6 \text{ mod } 10 \\
&= 6
\end{aligned}
$$

Key = 92

$$
\begin{aligned}
h(92, 2) &= [92 \text{ mod } 10 + (2 \times 92 \text{ mod } 8)] \text{ mod } 10 \\
&= [2 + (2 \times 4)] \text{ mod } 10 \\
&= [2 + 8] \text{ mod } 10 \\
&= 10 \text{ mod } 10 \\
&= 0
\end{aligned}
$$

Since T[0] is vacant, insert the key 92 in T[0]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	–1	36	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με $h_1 = (k \bmod 10)$ και $h_2 = (k \bmod 8)$.

Step 8

$$
\begin{aligned}
\text{Key} &= 101 \\
h(101, 0) &= [101 \bmod 10 + (0 \times 101 \bmod 8)] \bmod 10 \\
&= [1 + (0 \times 5)] \bmod 10 \\
&= 1 \bmod 10 \\
&= 1 \\
\text{Key} &= 101 \\
h(101, 1) &= [101 \bmod 10 + (1 \times 101 \bmod 8)] \bmod 10 \\
&= [1 + (1 \times 5)] \bmod 10 \\
&= [1 + 5] \bmod 10 \\
&= 6
\end{aligned}
$$

- Συνεχίζουμε με τον ίδιο τρόπο ......

Double Hashing

- Να κατακερματιστούν τα κλειδιά: 14, 17, 25, 37, 34, 16, 26 με M=11, $h_1(k)$ = k mod 11, $h_2(k)$ = k mod 7 + 1

	34		**14**	**37**	**16**	**17**		**25**		**26**

Linear Probing

- Πρόκειται για την πιο απλή τεχνική
- Αν μια τιμή αποθηκευτεί στη θέση h(k) τότε υιοθετούμε την ακόλουθη συνάρτηση κατακερματισμού για να επιλύσουμε τη σύγκρουση

$$h(k,i)=[h'(k)+i] \bmod m$$

$$h'(k)=k \bmod m$$

- Το i απεικονίζει τον probe αριθμό που κινείται από 0 μέχρι m-1
- Για ένα κλειδί k υπολογίζεται η θέση h'(k)=k mod m διότι την πρώτη φορά έχουμε i=0
- Αν η θέση είναι ελεύθερη, τότε το κλειδί αποθηκεύεται σε αυτή τη θέση
- Αν όχι, τότε υπολογίζουμε το δεύτερο probe [h'(k)+1] mod m

Linear Programming

Linear Programming (1/27)

Πολλά προβλήματα έχουν τη μορφή της μεγιστοποίησης ή ελαχιστοποίησης μιας (συνήθως) συνάρτησης κάτω από προϋποθέσεις

Η συνάρτηση είναι συνήθως γραμμική

Οι προυποθέσεις / περιορισμοί (constraint) παίρνουν τη μορφή ισοτήτων ή ανισοτήτων

Σε αυτές τις περιπτώσεις έχουμε **ένα πρόβλημα γραμμικού προγραμματισμού (linear programming problem)**

Linear Programming (2/27)

Η γενική μορφή του προβλήματος είναι να προσπαθήσουμε να βελτιστοποιήσουμε μια γραμμική συνάρτηση όταν ισχύει ένα σύνολο γραμμικών ανισοτήτων

Δοσμένου ενός συνόλου αριθμών $α_1$, $α_2$, …, $α_n$ και ενός συνόλου μεταβλητών x_1, x_2, …, x_n, ορίζουμε μια γραμμική συνάρτηση πάνω σε αυτές τις μεταβλητές ως εξής:

$$f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j$$

Αν b είναι ένας πραγματικός αριθμός και f μια γραμμική συνάρτηση τότε οι γραμμικές ισότητες / ανισώσεις έχουν ως εξής

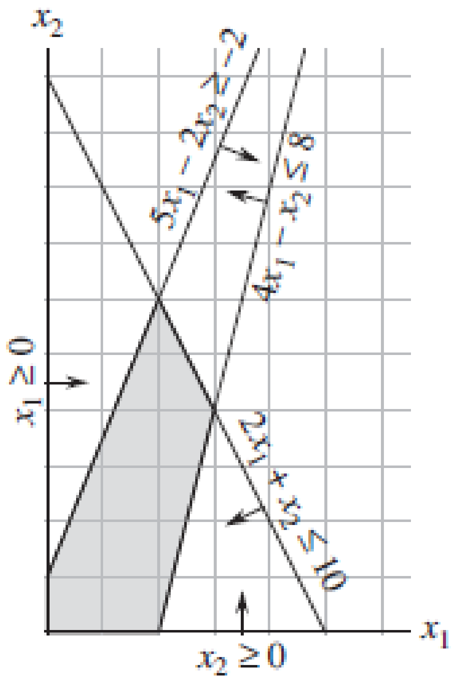
$$f(x_1, x_2, \ldots, x_n) = b \quad f(x_1, x_2, \ldots, x_n) \leq b \quad f(x_1, x_2, \ldots, x_n) \geq b$$

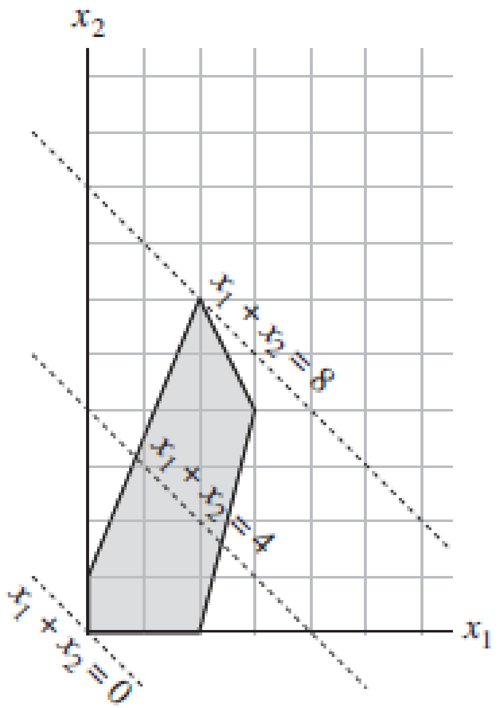
Linear Programming (3/27)

Παράδειγμα

$$
\begin{array}{llcrcr}
\text{maximize} & x_1 & + & x_2 & & \\
\text{subject to} & & & & & \\
& 4x_1 & - & x_2 & \le & 8 \\
& 2x_1 & + & x_2 & \le & 10 \\
& 5x_1 & - & 2x_2 & \ge & -2 \\
& & x_1, x_2 & & \ge & 0
\end{array}
$$

Πιθανές λύσεις





Linear Programming (4/27)

Παραδείγματα εφαρμογών

- Αεροπορική εταιρεία επιθυμεί να διαχειριστεί το πρόγραμμα των πληρωμάτων της. Οι αρχές θέτουν ένα σύνολο περιορισμών σχετικά με τις ώρες εργασίας καθώς και τον τύπο αεροπλάνου όπου θα εργαστεί κάθε πλήρωμα. Η εταιρεία επιθυμεί να μεγιστοποιήσει το όφελος όταν θα αναθέσει εργασίες στον ελάχιστο αριθμό πληρώματος
- Μια εταιρεία εξόρυξης πετρελαίου επιθυμεί να εντοπίσει το σημείο όπου θα κάνει την επόμενη γεώτρηση. Θα πρέπει να λάβει υπόψιν της το κόστος τοποθέτησης των μηχανημάτων και το αναμενόμενο όφελος. Η εταιρεία έχει περιορισμένους χρηματικούς πόρους για να κάνει την εξόρυξη και επιθυμεί να μεγιστοποιήσει την ποσότητα που θα εξάγει και το όφελος.

Linear Programming (5/27)

Η μέθοδος simplex είναι ο κλασικός τρόπος για την επίλυση προβλημάτων γραμμικού προγραμματισμού

Η μέθοδος, πρακτικά, είναι πολύ γρήγορη

Για να εφαρμόσουμε τη μέθοδο σε προβλήματα γραμμικού προγραμματισμού πρέπει να εκφράσουμε το πρόβλημα στην **τυποποιημένη μορφή (standard form)**

Linear Programming (6/27)

Η standard form έχει τις ακόλουθες απαιτήσεις:

- Πρέπει να ορίζεται ένα πρόβλημα **μεγιστοποίησης**
- **Όλοι οι περιορισμοί** (εκτός από τους μη αρνητικούς περιορισμούς) πρέπει να έχουν τη μορφή γραμμικών εξισώσεων με μη αρνητικά δεξιά μέρη
- **Όλες οι μεταβλητές** πρέπει να είναι μη αρνητικές

Το κύριο πλεονέκτημά της είναι στον απλό μηχανισμό που προσφέρει για να αναγνωρίσει τα οριακά σημεία της περιοχής των λύσεων

Linear Programming (7/27)

Υποθέτουμε ότι έχουμε **m** περιορισμούς και **n** μεταβλητές (n>=m)

Η γενική μορφή ενός προβλήματος γραμμικού προγραμματισμού έχει ως εξής:

$$\begin{array}{ll} \text{maximize} & c_1x_1 + \cdots + c_nx_n \\ \text{subject to} & a_{i1}x_1 + \cdots + a_{in}x_n = b_i, \quad b_i \geq 0 \;\; \text{for } i = 1, 2, \ldots, m \\ & x_1 \geq 0, \ldots, x_n \geq 0 \end{array}$$

ή με την υιοθέτηση συμβολισμού πινάκων

$$\begin{array}{ll} \text{maximize} & cx \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

Linear Programming (8/27)

Όπου

$$c = [c_1\ c_2\ \ldots\ c_n],\quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\quad A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix},\quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Οποιοδήποτε πρόβλημα γραμμικού προγραμματισμού μπορεί να γραφεί στην standard form

Linear Programming (9/27)

Όταν πρέπει να ελαχιστοποιήσουμε μια συνάρτηση, μπορεί να αντικατασταθεί από ένα ισοδύναμο πρόβλημα μεγιστοποίησης της ίδιας συνάρτησης αλλά έχουμε αντικαταστήσει τα c_j με $-c_j$

Όταν ένας περιορισμός δίνεται σαν μια ανίσωση, μπορεί να αντικατασταθεί από μια ισοδύναμη εξίσωση προσθέτωντας μια **χαλαρή μεταβλητή** (slack variable) που παριστάνει τη διαφορά των δύο μερών της αρχικής ανίσωσης

Παράδειγμα

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \rightarrow \quad \begin{aligned} s &= b_i - \sum_{j=1}^{n} a_{ij} x_j \\ s &\geq 0 \end{aligned}$$

Linear Programming (10/27)

Παράδειγμα

$$\begin{array}{ll} \text{maximize} & 3x + 5y \\ \text{subject to} & x + \;\; y \leq 4 \\ & x + 3y \leq 6 \\ & x \geq 0, \;\; y \geq 0 \end{array} \quad \rightarrow \quad \begin{array}{ll} \text{maximize} & 3x + 5y + 0u + 0v \\ \text{subject to} & x + \;\; y + \;\; u \qquad\quad = 4 \\ & x + 3y + \qquad + \;\; v = 6 \\ & x, y, u, v \geq 0 \end{array}$$

Άλλη μορφή συμβολισμού

$$x \begin{bmatrix} 1 \\ 1 \end{bmatrix} + y \begin{bmatrix} 1 \\ 3 \end{bmatrix} + u \begin{bmatrix} 1 \\ 0 \end{bmatrix} + v \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

Linear Programming (11/27)

Αν το σύστημα που προκύπτει έχει μια μοναδική λύση, αυτή ονομάζεται **βασική λύση (basic solution)**

Οι συντεταγμένες (στο δισδιάστατο χώρο) που τίθενται ίσες με το 0 πριν τη λύση του συστήματος καλούνται **μη βασικές (non basic)**

Οι συντεταγμένες (στο δισδιάστατο χώρο) που αποτιμούνται με τη λύση του συστήματος καλούνται **βασικές (basic)**

Μια βάση στο δισδιάστατο χώρο αποτελείται από δύο διανύσματα τα οποία είναι ανάλογα μεταξύ τους

Όταν επιλεγεί η βάση, κάθε διάνυσμα μπορεί να εκφραστεί ως ένα άθροισμα πολλαπλασίων των διανυσμάτων βάσης

Linear Programming (12/27)

Οι βασικές και μη βασικές συντεταγμένες δείχνουν ποια από τα δοσμένα διανύσματα περιλαμβάνονται ή αποκλείονται από την επιλογή μιας βάσης

Αν όλες οι συντεταγμένες μιας βασικής λύσης είναι μη αρνητικές, αυτή καλείται **βασική εφικτή λύση (basic feasible solution)**

Για παράδειγμα αν θέσουμε τις μεταβλητές x και y ίσες με 0 και λύσουμε ως προς τις u, υ παίρνουμε

(0, 0, 4, 6)

$$\begin{array}{ll} \text{maximize} & 3x + 5y + 0u + 0v \\ \text{subject to} & x + \ \ y + u \qquad\ \ = 4 \\ & x + 3y + \quad\ + \ v = 6 \\ & x, y, u, v \geq 0 \end{array}$$

Linear Programming (13/27)

Αν θέσουμε τις μεταβλητές x και u ίσες με 0 και λύσουμε ως προς τις y, υ παίρνουμε

(0, 4, 0, -6)

η οποία δεν είναι βασική εφικτή λύση

Οι βασικές εφικτές λύσεις έχουν αντιστοιχία ένα-προς-ένα με τα οριακά σημεία της περιοχής εφικτών λύσεων

$$
\begin{array}{ll}
\text{maximize} & 3x + 5y + 0u + 0v \\
\text{subject to} & x + y + u = 4 \\
& x + 3y + + v = 6 \\
& x, y, u, v \geq 0
\end{array}
$$

Linear Programming (14/27)

Η μέθοδος simplex προχωρά μέσα σε μια σειρά γειτονικών οριακών σημείων (βασικές εφικτές λύσεις) με αυξανόμενες τιμές της συνάρτησης που προσπαθούμε να μεγιστοποιήσουμε

Κάθε ένα από αυτά τα σημεία μπορεί να αναπαρασταθεί από ένα **simplex tableau**

Ένα simplex tableau είναι ένας πίνακας που αποθηκεύει πληροφορίες για τις εφικτές λύσεις που αντιστοιχούν στα οριακά σημεία του χώρου των λύσεων

Linear Programming (15/27)

Για παράδειγμα, για τη λύση (0, 0, 4, 6) του προηγούμενου συστήματος ο simplex tableau έχει ως εξής:

$$
\begin{array}{ll}
\text{maximize} & 3x + 5y + 0u + 0v \\
\text{subject to} & x + y + u = 4 \\
 & x + 3y + + v = 6 \\
 & x, y, u, v \geq 0
\end{array}
$$

	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	–3	–5	0	0	0

↑

Linear Programming (16/27)

Ένας simplex tableau έχει τα ακόλουθα χαρακτηριστικά:

- Περιλαμβάνει m+1 γραμμμές και n+1 στήλες
- Κάθε γραμμή αντιστοιχεί σε περιορισμούς και οι στήλες αντιστοιχούν στις μεταβλητές
- Κάθε γραμμή περιλαμβάνει τα coefficients του κάθε περιορισμού
- Η τελευταία στήλη δείχνει το δεξιό μέρος των περιορισμών
- Οι γραμμές έχουν ετικέτες των βασικών μεταβλητών της εφικτής λύσης

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	–3	–5	0	0	0

Linear Programming (17/27)

- Η τελευταία γραμμή ονομάζεται **objective row**
- Αρχικοποιείται με τα coefficients της συνάρτησης με αντίστροφο πρόσημο (για τις πρώτες n στήλες) και την τιμή της συνάρτησης στην τελευταία στήλη
- Σε επόμενες επαναλήψεις η objective row μετασχηματίζεται με τον ίδιο τρόπο όπως και οι υπόλοιπες γραμμές
- Η συγκεκριμένη γραμμή χρησιμοποιείται από τη μέθοδο για να διαπιστώσει αν η τρέχουσα λύση είναι η βέλτιστη

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	–3	–5	0	0	0

Linear Programming (18/27)

- Η τρέχουσα λύση είναι μια βέλτιστη λύση αν όλες οι τιμές της εκτός ίσως από την τελευταία στήλη είναι μη αρνητικές
- Αν δεν ισχύει αυτό, τότε κάθε αρνητική τιμή αντιπροσωπεύει μια μη βασική μεταβλητή η οποία πρόκειται να γίνει βασική στο επόμενο tableau
- Η εφικτή λύση (0, 0, 4, 6) του πίνακα στο παράδειγμα, δεν είναι βέλτιστη
- Η αρνητική τιμή στη στήλη του x σημαίνει πως μπορούμε να αυξήσουμε την τιμή της συνάρτησης, αυξάνοντας την τιμή της συντεταγμένης x

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	–3	–5	0	0	0

Linear Programming (19/27)

- Αφού ο συντελεστής του x στη συνάρτηση είναι θετικός, όσο μεγαλύτερο είναι το x τόσο θα αυξάνει η τιμή της συνάρτησης
- Η αύξηση του x θα εξαρτηθεί και από τις τιμές των u, υ έτσι ώστε το νέο σημείο της συνάρτησης να είναι εφικτό
- Πρέπει να ικανοποιούνται οι ακόλουθες συνθήκες

$$x + u = 4 \quad u \geq 0$$
$$x + v = 6 \quad v \geq 0$$

- άρα

$$x \leq \min\{4, 6\} = 4$$

	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0
		↑			

Linear Programming (20/27)

- Αν αυξήσουμε το x από το 0 στο 4, βρίσκουμε το σημείο (4, 0, 0, 2) που είναι γειτονικό στο (0, 0, 4, 6) με την τιμή της συνάρτησης να είναι ίση με 12
- Όμοια επεξεργασία ακολουθούμε και για την αρνητική τιμή του y
- Η αύξηση της τιμής του y απαιτεί

$$\begin{aligned} y + u &= 4 \quad u \geq 0 \\ 3y + v &= 6 \quad v \geq 0 \end{aligned}$$

- άρα

$$y \leq \min\{\frac{4}{1}, \frac{6}{3}\} = 2$$

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	–3	–5	0	0	0

Linear Programming (21/27)

- Συνεπώς μπορούμε να αυξήσουμε το y κατά 2 και θα βρούμε τη λύση (0, 2, 2, 0) που είναι ακόμη ένα γειτονικό σημείο στο (0, 0, 4, 6) με τη συνάρτηση να παίρνει την τιμή 10
- Αν υπάρχουν αρκετές αρνητικές τιμές στην objective row, εφαρμόζεται ο ίδιος κανόνας με πρώτη επιλογή το μεγαλύτερο, σε απόλυτη τιμή, αρνητικό αριθμό
- Η τακτική αυτή στοχεύει στη μεταβλητή που έχει την μεγαλύτερη πιθανότητα για μεγαλύτερη αύξηση

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	−3	−5	0	0	0

Linear Programming (22/27)

Οι περιορισμοί θέτουν διαφορετικά κριτήρια ως προς το μέγεθος αύξησης κάθε μεταβλητής

Μια νέα βασική μεταβλητή καλείται **μεταβλητή εισόδου (entering variable)** ενώ η στήλη της ονομάζεται **pivot column** (παίρνουμε τη μεταβλητή που προκαλεί τη μεγαλύτερη αύξηση, π.χ., y)

Ας δούμε όμως πως επιλέγουμε μια **departing variable** (μια βασική μεταβλητή για να γίνει μη βασική)

Βασιζόμαστε στην παρατήρηση για να πάρουμε ένα γειτονικό σημείο στις λύσεις που οδηγεί σε μεγαλύτερες τιμές της συνάρτησης, πρέπει να αυξήσουμε την μεταβλητή εισόδου κατά τη μέγιστη δυνατή ποσότητα

Linear Programming (23/27)

Με αυτό τον τρόπο θα μπορέσουμε να θέσουμε μια από τις παλιές μεταβλητές ίσες με 0 διατηρώντας το κριτήριο της ύπαρξης μη αρνητικών τιμών για τις όλες υπόλοιπες

Προκύπτει ο ακόλουθος κανόνας για την επιλογή της **departing variable**

- Για κάθε θετική τιμή στη pivot column, υπολογίζουμε το **θ-ratio** διαιρώντας την τελευταία τιμή της γραμμής με την τιμή στην pivot column. Στο παράδειγμα έχουμε:

	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	–3	–5	0	0	0
		↑			

$$\theta_u = \frac{4}{1} = 4, \qquad \theta_v = \frac{6}{3} = 2$$

Linear Programming (24/27)

- Η γραμμή **με το μικρότερο θ-ratio** καθορίζει την departing variable, δηλαδή τη μεταβλητή που θα γίνει μη βασική
- Στο παράδειγμα μας είναι η μεταβλητή **υ**
- Αν δεν υπάρχει κάποια θετική τιμή στην pivot column, δεν μπορούμε να υπολογίσουμε το θ-ratio που σημαίνει ότι το πρόβλημα δεν φράσσεται και ο αλγόριθμος σταματά
- Στη συνέχεια μαρκάρουμε τη γραμμή της μεταβλητής που είναι η pivot row

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	−3	−5	0	0	0

$$\theta_u = \frac{4}{1} = 4, \qquad \theta_v = \frac{6}{3} = 2$$

Linear Programming (25/27)

Τα επόμενα βήματα αφορούν για το μετασχηματισμό του τρέχοντος tableau στο νέο

Αρχικά διαιρούμε όλες τις τιμές της pivot row με την τιμή pivot (την τιμή της pivot column) και παίρνουμε νέες τιμές για τη γραμμή

Έπειτα αντικαθιστούμε κάθε μια από τις άλλες γραμμές, μαζί με την objective row, με τη διαφορά $\text{row} - c \cdot \overleftarrow{\text{row}}_{\text{new}}$

όπου c είναι η τιμή της γραμμής στην pivot column

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	–3	–5	0	0	0

$$\overleftarrow{\text{row}}_{\text{new}}: \quad \frac{1}{3} \quad 1 \quad 0 \quad \frac{1}{3} \quad 2$$

Linear Programming (26/27)

Για το παράδειγμά μας έχουμε $\text{row} - c \cdot \overleftrightarrow{\text{row}}_{\text{new}}$

$$\text{row } 1 - 1 \cdot \overleftrightarrow{\text{row}}_{\text{new}}: \quad \frac{2}{3} \quad 0 \quad 1 \quad -\frac{1}{3} \quad 2$$

$$\text{row } 3 - (-5) \cdot \overleftrightarrow{\text{row}}_{\text{new}}: \quad -\frac{4}{3} \quad 0 \quad 0 \quad \frac{5}{3} \quad 10$$

Και ο πίνακας μετασχηματίζεται ως εξής:

	x	y	u	v	
u	1	1	1	0	4
$\leftarrow v$	1	3	0	1	6
	–3	–5	0	0	0
		$\uparrow$			

→

	x	y	u	v	
$\leftarrow u$	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	10
	$\uparrow$				

→

	x	y	u	v	
x	1	0	$\frac{3}{2}$	$-\frac{1}{2}$	3
y	0	1	$-\frac{1}{2}$	$\frac{1}{2}$	1
	0	0	2	1	14

Linear Programming (27/27)

Αλγόριθμος

Step 0 *Initialization* Present a given linear programming problem in standard form and set up an initial tableau with nonnegative entries in the rightmost column and m other columns composing the $m \times m$ identity matrix. (Entries in the objective row are to be disregarded in verifying these requirements.) These m columns define the basic variables of the initial basic feasible solution, used as the labels of the tableau's rows.

Step 1 *Optimality test* If all the entries in the objective row (except, possibly, the one in the rightmost column, which represents the value of the objective function) are nonnegative—stop: the tableau represents an optimal solution whose basic variables' values are in the rightmost column and the remaining, nonbasic variables' values are zeros.

Step 2 *Finding the entering variable* Select a negative entry from among the first n elements of the objective row. (A commonly used rule is to select the negative entry with the largest absolute value, with ties broken arbitrarily.) Mark its column to indicate the entering variable and the pivot column.

Step 3 *Finding the departing variable* For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If all the entries in the pivot column are negative or zero, the problem is unbounded—stop.) Find the row with the smallest θ-ratio (ties may be broken arbitrarily), and mark this row to indicate the departing variable and the pivot row.

Step 4 *Forming the next tableau* Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. (This will make all the entries in the pivot column 0's except for 1 in the pivot row.) Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

String Matching

String Matching (1/42)

Έστω ένα αλφαριθμητικό το οποίο είναι αποθηκευμένο σε ένα πίνακα T[1..n] με μήκος n και μια διάταξη (pattern) P[1..m] μήκους m<=n

Υποθέτουμε πως τα στοιχεία των T και P είναι χαρακτήρες μέσα από ένα αλφάβητο Σ

Λέμε πως το P συναντάται με μετατόπιση s στο T (ισοδύναμα λέμε πως το P ξεκινά στη θέση s+1) εφόσον 0<=s<=n-m και T[s+1..s+m]=P[1..m]

Ισχύει πως T[s+j]=P[j] με 1<=j<=m

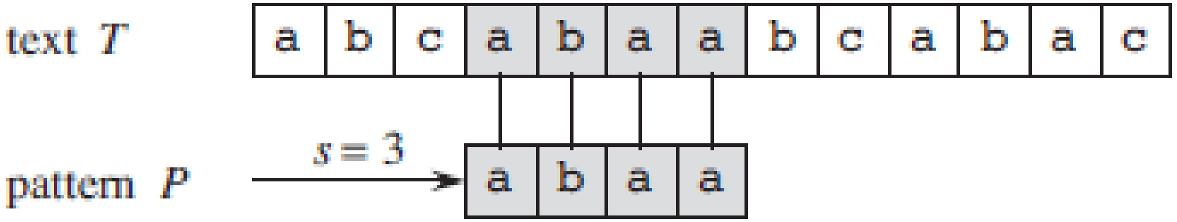
String Matching (2/42)

Αν το P συναντάται στο T τότε το s είναι μια **έγκυρη μετατόπιση (valid shift)**, διαφορετικά είναι μια **μη έγκυρη μετατόπιση (invalid shift)**

Το string matching problem έγκειται στο να βρούμε όλες τις έγκυρες μετατοπίσεις s

String Matching (3/42)

Παράδειγμα



Αλγόριθμοι

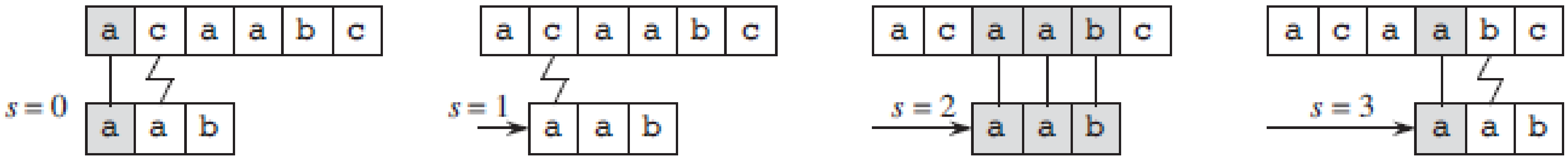
Algorithm	Preprocessing time	Matching time
Naive	0	$O((n - m + 1)m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Finite automaton	$O(m \lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

String Matching (4/42)

Naïve String Matching

- Βρίσκει όλες τις εμφανίσεις του pattern ψάχνοντας σε όλες τις θέσεις του T
- Παράδειγμα εκτέλεσης

```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n − m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```



String Matching (5/42)

Naïve String Matching

- Για κάθε μια από τις πιθανές τιμές του s (n-m+1) , ο αλγόριθμος στη γραμμή 4 πρέπει να ελέγξει όλες τις m θέσεις του pattern
- Η πολυπλοκότητα στη χειρότερη περίπτωση θα είναι Θ((n-m+1)m) το οποίο μπορεί να καταλήξει σε $\Theta(n^2)$ αν m = n/2
- Ο αλγόριθμος δεν απαιτεί preprocessing

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

String Matching (6/42)

Naïve String Matching

- Ο αλγόριθμος δεν εκμεταλλεύεται την πληροφορία που αποκτά για το T κατά τις πρώτες επαναλήψεις
- Αυτή η πληροφορία μπορεί να είναι χρήσιμη κατά τις υπόλοιπες επαναλήψεις

String Matching (7/42)

The Rabin Carp Algorithm

- Υιοθετεί preprocessing κόστους Θ(m) και η πολυπλοκότητα στη χειρότερη περίπτωση είναι Θ((n-m+1)m)
- Βασιζόμενοι σε κάποιες υποθέσεις, η επίδοση στη μέση περίπτωση είναι πολύ καλύτερη
- Ο αλγόριθμος υπολογίζει **μια αριθμητική τιμή (hash)** για το P και για κάθε substring μήκους m του T
- Στη συνέχεια συγκρίνει τα πραγματικά σύμβολα
- Αν βρει κάποιο ταίριασμα, τότε συγκρίνει το P με το συγκεκριμένο substring με τη βοήθεια του naïve matcher

String Matching (8/42)

The Rabin Carp Algorithm

- Έστω p η αριθμητική τιμή του P
- Έστω t_s η αριθμητική τιμή του substring μήκους m του T
- **Προφανώς $p=t_s$ εφόσον T[s+1..s+m] = P[1..m]**
- Συνεπώς το s είναι μια έγκυρη μετατόπιση εφόσον $p=t_s$
- Ο υπολογισμός του p γίνεται με τη βοήθεια του κανόνα του Horner

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))$$

- Ομοίως, μπορούμε να υπολογίσουμε το t_0 από το T[1..m]
- Κάνουμε την υπόθεση πως στο T και το P έχουμε αριθμητικές τιμές

String Matching (9/42)

The Rabin Carp Algorithm

- Για τον υπολογισμό των υπόλοιπων t_1, t_2,…, t_{n-m} παρατηρούμε πως μπορούμε να υπολογίσουμε το t_{s+1} από το t_s σε σταθερό χρόνο αφού

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

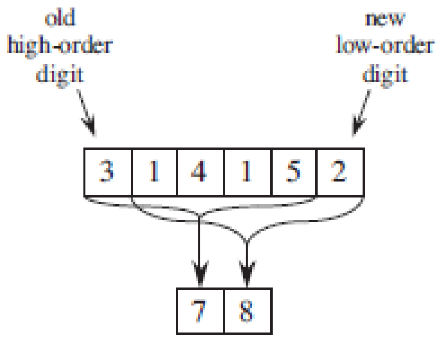
- t_s → κώδικας για τους χαρακτήρες [s+1, s+m]
- Η αφαίρεση του $10^{m-1}T[s+1]$, εξαλείφει τα πιο σημαντικά ψηφία του t_s και ο πολλαπλασιασμός με το 10 προκαλεί μετατόπιση των αριθμών προς τα αριστερά κατά ένα ψηφίο
- Η πρόσθεση του T[s+m+1] μας φέρνει στη σωστή θέση τα λιγότερο σημαντικά ψηφία

String Matching (10/42)

The Rabin Carp Algorithm

- Παράδειγμα
  - m=5, t_s=31415
  - Πρέπει να εξαλείψουμε τα T[s+1] = 3 σημαντικά ψηφία και να φέρουμε στη σωστή θέση τα λιγότερο σημαντικά ψηφία (T[s+5+1] = 2)
  - Παίρνουμε

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \end{aligned}$$



old high-order digit, shift, new low-order digit

$$\begin{aligned} 14152 &\equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\ &\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\ &\equiv 8 \pmod{13} \end{aligned}$$

Η μεταβλητή q = 13 είναι ένας πρώτος αριθμός τέτοιος ώστε το 10q να χωράει στη λέξη του Η/Υ

String Matching (11/42)

The Rabin Carp Algorithm

- Για τον υπολογισμό των p και t_s παίρνουμε το υπόλοιπο της διαίρεσης με μια μεταβλητή q
- Αν επιλέξουμε το q ως ένα πρώτο αριθμό τέτοιο ώστε το 10q να χωράει στη λέξη του H/Y τότε όλοι οι υπολογισμοί θα γίνει με απλή ακρίβεια
- Γενικά, αν έχουμε ένα αλφάβητο d στοιχείων {0,1,2,3, ..., d-1}, επιλέγουμε το q τέτοιο ώστε το **dq** να χωρά στη λέξη του H/Y
- Η εξίσωση υπολογισμού γίνεται

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

- με $h = d^{m-1}$

String Matching (12/42)

The Rabin Carp Algorithm

- **Το d το λαμβάνουμε ίσο με |Σ|**
- **Όλοι οι χαρακτήρες αναπαριστούνται ως d ψηφία**

RABIN-KARP-MATCHER(T, P, d, q)

```
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m    // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (d t_0 + T[i]) mod q
for s = 0 to n - m     // matching
    if p == t_s
        if P[1..m] == T[s+1..s+m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

String Matching (13/42)

The Knuth-Morris-Pratt Algorithm

- Υιοθετεί μια συνάρτηση που υπολογίζεται με βάση το P
- Αποθηκεύουμε την πληροφορία σε ένα πίνακα π[1..m]
- Ο π μας επιτρέπει να υπολογίσουμε μια συνάρτηση μετάβασης δ (δ: QXΣ → Q
- Q είναι ένα σύνολο καταστάσεων σχετικά με ελέγχους που κάνουμε πάνω στο T)
- Για κάθε κατάσταση q = 0,1,2, …, m και κάθε χαρακτήρα α που ανήκει στο Σ, η τιμή π[q] περιέχει την πληροφορία που απαιτείται για να υπολογιστεί το δ(q,α) που δεν εξαρτάται από το α
- Ο π έχει m και η δ έχει Θ(m |Σ|) στοιχεία

String Matching (14/42)

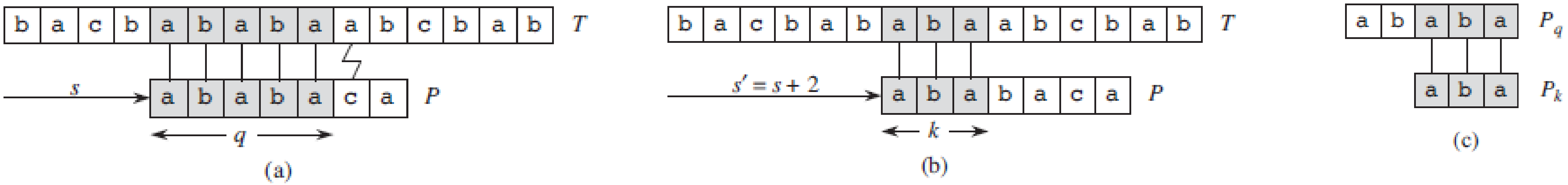
The Knuth-Morris-Pratt Algorithm

- Η π περιλαμβάνει γνώση για το πως το P ταιριάζει σχετικά με μετακινήσεις του
- Αυτή η γνώση υιοθετείται ώστε να αποφευχθούν άσκοπες μετακινήσεις του P
- Επίσης αποφεύγουμε να υπολογίσουμε όλα τα στοιχεία της δ

String Matching (15/42)

The Knuth-Morris-Pratt Algorithm

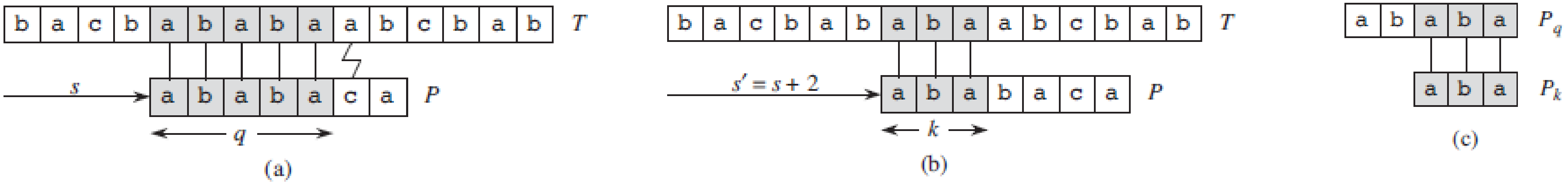
- Στο επόμενο παράδειγμα παίρνουμε q=5 και P=ababaca
- Βλέπουμε πως 5 χαρακτήρες ταιριάζουν επιτυχώς ενώ στον 6ο έχουμε αποτυχία
- Γνωρίζοντας τους q χαρακτήρες μπορούμε να βρούμε κάποιες μη έγκυρες μετακινήσεις



String Matching (16/42)

The Knuth-Morris-Pratt Algorithm

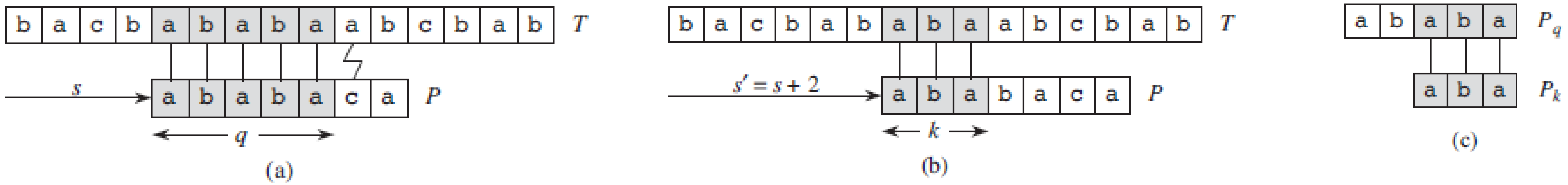
- Η μετακίνηση s+1 δεν είναι έγκυρη αφού ο χαρακτήρας a θα ευθυγραμμιστεί με ένα χαρακτήρα του T που ξέρουμε πως δεν ταιριάζει αλλά ταιριάζει με το 2ο χαρακτήρα (b)
- Η μετακίνηση s’=s+2 ευθυγραμμίζει 3 χαρακτήρες που ταιριάζουν



String Matching (17/42)

The Knuth-Morris-Pratt Algorithm

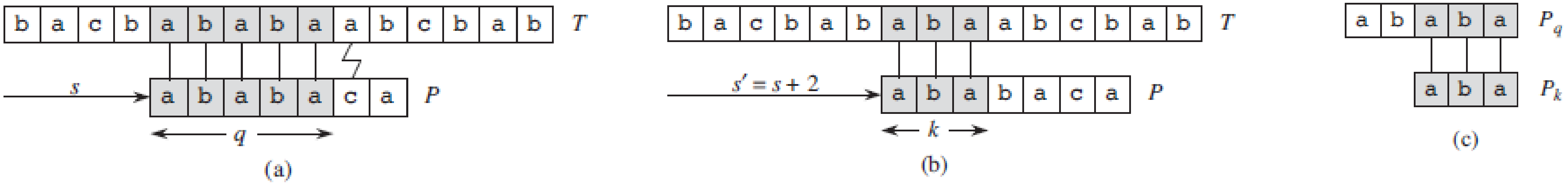
- Το ερώτημα που γεννάται είναι:
  - Δοσμένου των P[1..q] χαρακτήρων που ταιριάζουν με χαρακτήρες του T, T[s+1..s+q], ποια είναι η ελάχιστη μετατόπιση s’>s τέτοια ώστε για κάποιο k<q να έχουμε P[1..k]=T[s’+1..s’+k] όπου s’+k=s+q;



String Matching (18/42)

The Knuth-Morris-Pratt Algorithm

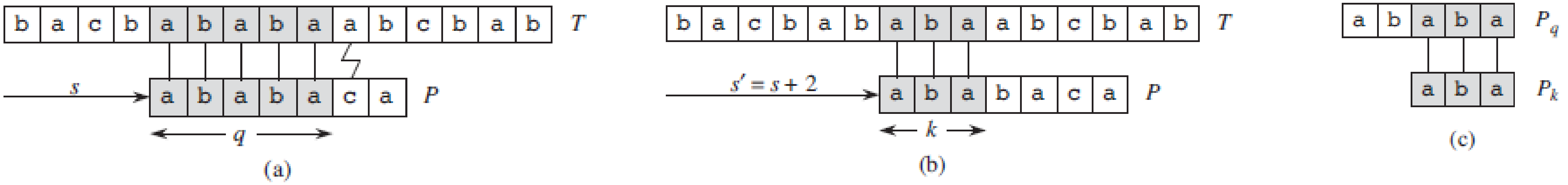
- Προσθέτουμε τη διαφορά q-k του μήκους των prefixes του P στο s έτσι ώστε να παραχθεί μια νέα μετατόπιση s’=s+q και παράγουμε τις μετατοπίσεις s+1, s+2, …, s+q-1
- Σε κάθε μετατόπιση δεν χρειάζεται να συγκρίνουμε τους πρώτους k χαρακτήρες διότι η εξίσωση P[1..k]=t[s’+1..s’+k] εξασφαλίζει το ταίριασμα



String Matching (19/42)

The Knuth-Morris-Pratt Algorithm

- Η τελευταία εικόνα μας δείχνει πως μπορούμε να προ-υπολογίσουμε την απαραίτητη πληροφορία συγκρίνοντας το P με τον εαυτό του
- Αφού το T[s’+1..s’+k] είναι τμήμα του T, ζητάμε το μεγαλύτερο k έτσι ώστε το P_q να περιλαμβάνει το P_k



String Matching (20/42)

The Knuth-Morris-Pratt Algorithm

- Η συνάρτηση π ορίζεται ως εξής:

$$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$$

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

- Το π[q] είναι το μήκος του μεγαλύτερου prefix του P που είναι ένα κατάλληλο επίθημα (suffix) του P_q

String Matching (21/42)

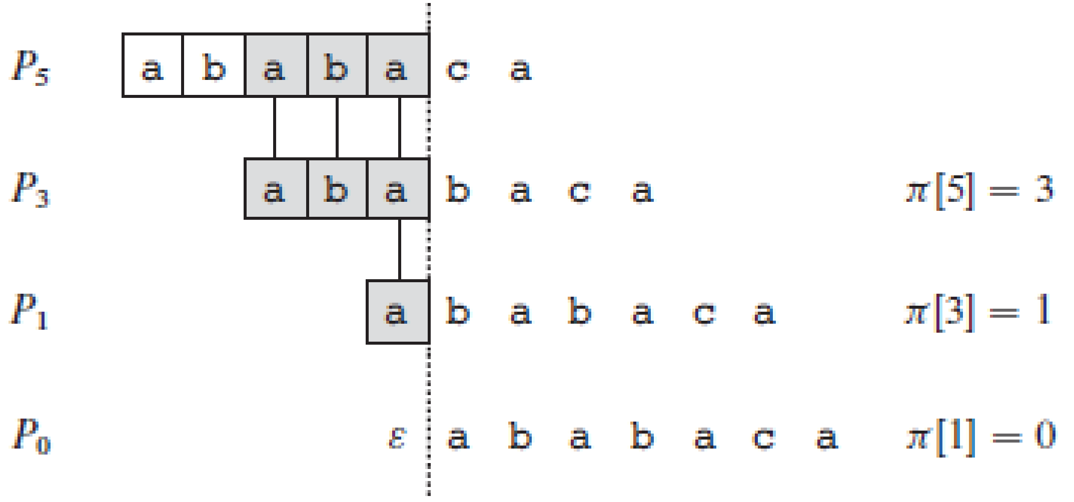
The Knuth-Morris-Pratt Algorithm

- Παράδειγμα

$$\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$$
$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

String Matching (22/42)

The Knuth-Morris-Pratt Algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                   // number of characters matched
for i = 1 to n                          // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                        // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                       // next character matches
    if q == m                           // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                        // look for the next match
```

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

String Matching (23/42)

The Horspool’s Algorithm

- Ας υποθέσουμε πως ψάχνουμε τη συμβολοσειρά BARBER σε ένα κείμενο T
- Μπορούμε να ξεκινήσουμε από το τελευταίο R και να μετακινούμαστε προς τα αριστερά
- Συγκρίνουμε ζεύγη χαρακτήρων
- Αν όλοι οι χαρακτήρες είναι ίδιοι τότε έχουμε βρει τη συμβολοσειρά
- Σε αυτή την περίπτωση η αναζήτηση μπορεί να τερματίσει ή να συνεχιστεί για την εύρεση επόμενων εμφανίσεων

$$\begin{array}{ccccccccccc} s_0 & \ldots & & & & & c & & \ldots & & s_{n-1} \\ & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} \;\; \mathrm{R} & & & & \end{array}$$

String Matching (24/42)

The Horspool's Algorithm

- Αν βρούμε κάποια διαφορά, τότε το P μετακινείται προς τα δεξιά
- Ο αλγόριθμος προσπαθεί να βρει τη μεγαλύτερη δυνατή μετατόπιση
- Κοιτάζει το χαρακτήρα c ο οποίος έχει ευθυγραμμιστεί με τον τελευταίο χαρακτήρα του P
- Ισχύει ακόμα και αν ο c είναι όμοιος με την τελευταία θέση του P

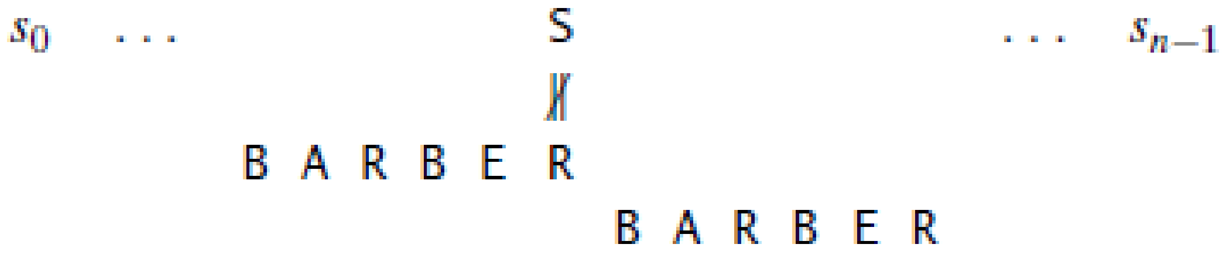
$s_0 \quad \ldots \quad c \quad \ldots \quad s_{n-1}$

B A R B E R

String Matching (25/42)

The Horspool’s Algorithm

- Περίπτωση 1: αν το c δεν ταιριάζει / δεν υπάρχει, τότε μπορούμε να μετακινήσουμε το P ως προς ολόκληρο το μήκος του



String Matching (26/42)

The Horspool's Algorithm

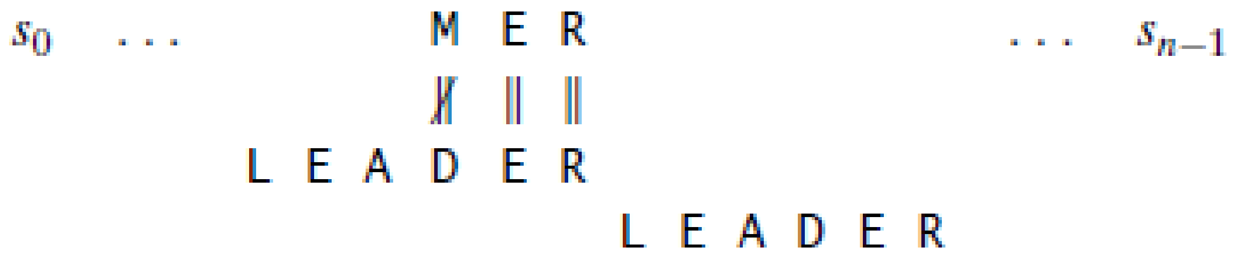
- Περίπτωση 2: Αν υπάρχουν εμφανίσεις του c στο P αλλά δεν είναι στην τελευταία θέση, η μετατόπιση μπορεί να γίνει ως προς την πιο δεξιά εμφάνιση του c στο P

$$
\begin{array}{ccccccccccccc}
s_0 & \ldots & & & & & \mathrm{B} & & & & \ldots & s_{n-1} \\
 & & & & & & \nparallel & & & & & \\
 & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} \;\; \mathrm{R} & & & & & \\
 & & & & \mathrm{B} & \mathrm{A} & \mathrm{R} \;\; \mathrm{B} & \mathrm{E} & \mathrm{R} & & &
\end{array}
$$

String Matching (27/42)

The Horspool’s Algorithm

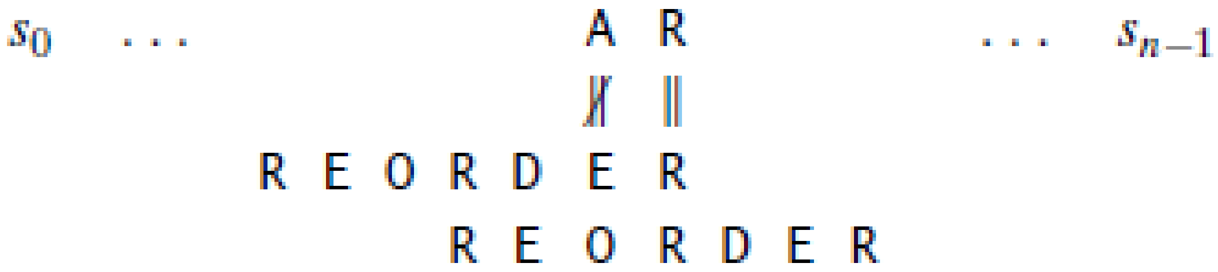
- Περίπτωση 3: Αν το c υπάρχει στην τελευταία θέση του P αλλά δεν υπάχρει το c στις υπόλοιπες θέσεις (m-1), έχουμε όμοια περίπτωση με την 1$^{\eta}$.



String Matching (28/42)

The Horspool's Algorithm

- Περίπτωση 4: αν το c υπάρχει στην τελευταία θέση αλλά και σε κάποιες άλλες ενδιάμεσες θέσεις, τότε έχουμε όμοια περίπτωση με τη 2η



String Matching (29/42)

The Horspool's Algorithm

- Μπορούμε να υπολογίσουμε σε προγενέστερο χρόνο το μήκος των μετατοπίσεων και να τις αποθηκεύσουμε σε ένα πίνακα
- Τα στοιχεία του πίνακα δείχνουν το μήκος των μετατοπίσεων ως ακολούθως:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

String Matching (30/42)

The Horspool's Algorithm

- Αλγόριθμος εύρεσης μήκους μετατοπίσεων

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
//   filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size - 1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m - 2$ **do** $Table[P[j]] \leftarrow m - 1 - j$
return *Table*

String Matching (31/42)

The Horspool's Algorithm

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

//Implements Horspool's algorithm for string matching
//Input: Pattern $P[0..m-1]$ and text $T[0..n-1]$
//Output: The index of the left end of the first matching substring
// or -1 if there are no matches

ShiftTable$(P[0..m-1])$ //generate *Table* of shifts
$i \leftarrow m-1$ //position of the pattern's right end
while $i \leq n-1$ **do**
  $k \leftarrow 0$ //number of matched characters
  while $k \leq m-1$ **and** $P[m-1-k] = T[i-k]$ **do**
    $k \leftarrow k+1$
  if $k = m$
    return $i - m + 1$
  else $i \leftarrow i + Table[T[i]]$
return -1

String Matching (32/42)

The Horspool's Algorithm

- Παράδειγμα

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

String Matching (33/42)

The Boyer-Moore Algorithm

- Αν η σύγκριση του χαρακτήρα που ευθυγραμμίζεται με το c (τελευταίος χαρακτήρας του P), ο αλγόριθμος κάνει τις ίδιες λειτουργίες με τον Horspool
- Μετακινεί το P τόσες θέσεις όσες δείχνει ο πίνακας που έχει ήδη προ-υπολογιστεί
- Οι δύο αλγόριθμοι όμως συμπεριφέρονται διαφορετικά όταν μετά από k χαρακτήρες που έχουν ταιριάξει πριν παρατηρηθεί κάποια διαφορά

$$\begin{array}{cccccccccc}
s_0 & \ldots & & & c & s_{i-k+1} & \ldots & s_i & \ldots & s_{n-1} & \text{text} \\
 & & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \ldots & & p_{m-k-1} & p_{m-k} & \ldots & p_{m-1} & & & \text{pattern}
\end{array}$$

String Matching (34/42)

The Boyer-Moore Algorithm

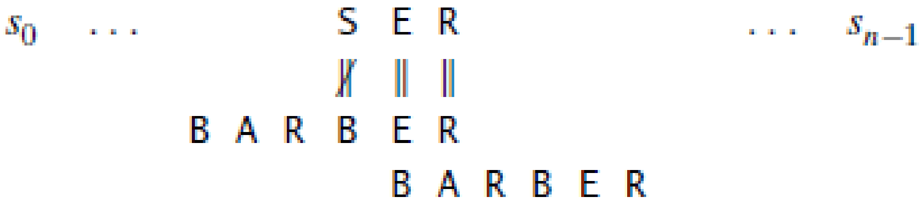
- Σε αυτές τις περιπτώσεις ο αλγόριθμος λαμβάνει υπόψιν του δύο ποσότητες:
  - Η πρώτη εξαρτάται από το c που προκάλεσε μια διαφορά (bad symbol shift). Αν το c δεν είναι στο P, μετατοπίζουμε το P ώστε να ξεπεράσει το c – το μήκος της μετατόπισης μπορεί να υπολογιστεί από το $t_1(c) - k$, όπου $t_1(c)$ είναι η τιμή στον προ-υπολογισμένο πίνακα όπως αυτός υπολογίζεται από τον αλγόριθμο Horspool και k είναι το πλήθος των χαρακτήρων που ταιριάζουν

s_0	$\dots$		c	s_{i-k+1}	$\dots$	s_i	$\dots$	s_{n-1}	text
			$\not\parallel$	$\parallel$		$\parallel$			
	p_0	$\dots$	p_{m-k-1}	p_{m-k}	$\dots$	p_{m-1}			pattern
				p_0	$\dots$		p_{m-1}		

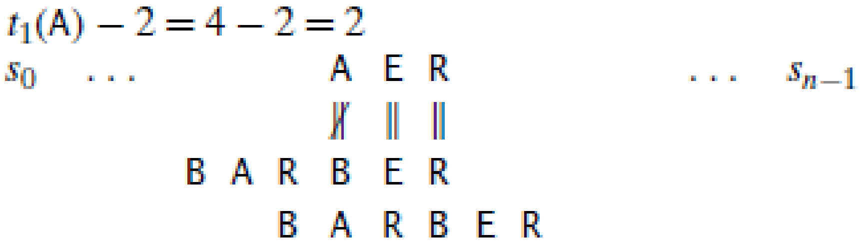
String Matching (35/42)

The Boyer-Moore Algorithm

- Παράδειγμα:
- Αν ψάχνουμε το BARBER και ταιριάξουμε τους τελευταίους 2 χαρακτήρες, μπορούμε να μετατοπίσουμε το P κατά t_1(S)-2=6-2=4 θέσεις



- Ο ίδιος τύπος μπορεί να υιοθετηθεί όταν ο c υπάρχει στο P δεδομένου ότι t_1(c) – k >0
- Παράδειγμα



String Matching (36/42)

The Boyer-Moore Algorithm

- Όταν $t_1(c) - k < 0$, τότε φυσικά δεν θέλουμε να κάνουμε μετατόπιση με αρνητικό αριθμό θέσεων, όποτε υιοθετούμε μια brute force προσέγγιση και μετατοπίζουμε κατά 1
- Γενικά, η μετατόπιση για ένα bad symbol d_1, υπολογίζεται με βάση το πρόσημο της παράστασης $t_1(c) - k$

  $$d_1 = \max\{t_1(c) - k,\ 1\}.$$
- Ο δεύτερος τύπος μετατόπισης εξαρτάται από το επιτυχές ταίριασμα των τελευταίων k χαρακτήρων του P
- Συμβολίζουμε με suff(k) το τερματικό τμήμα του P που αποτελεί ένα επίθημα
- Ονομάζεται good suffix shift

String Matching (37/42)

The Boyer-Moore Algorithm

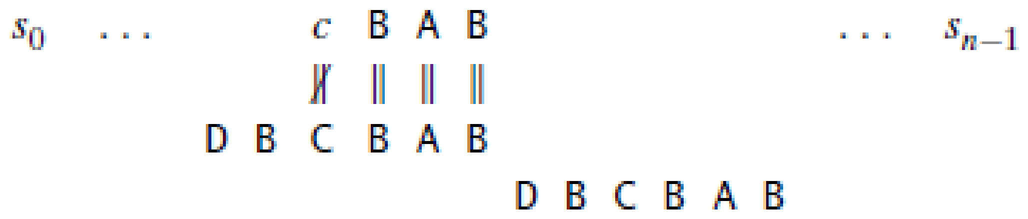
- Ας θεωρήσουμε την περίπτωση όπου υπάρχει και άλλο suff(k) στο P στην οποία δεν προηγείται ο ίδιος χαρακτήρας όπως για το πιο δεξιά suff(k)
- Σε αυτή την περίπτωση μπορούμε να μετατοπίσουμε το P κατά μια απόσταση d_2 ανάμεσα στα δύο suff(k)
- Παράδειγμα

k	**pattern**	d_2
1	$\mathrm{ABC}\overline{\mathrm{B}}\mathrm{A}\underline{\mathrm{B}}$	2
2	$\overline{\mathrm{AB}}\mathrm{CB}\underline{\mathrm{AB}}$	4

String Matching (38/42)

The Boyer-Moore Algorithm

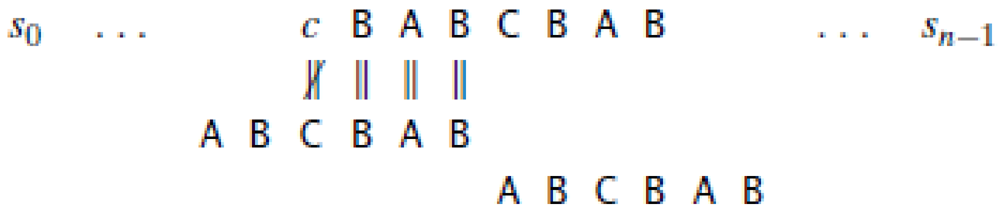
- Τι γίνεται όμως στην περίπτωση όπου δεν υπάρχει άλλη εμφάνιση του suff(k) στην οποία δεν προηγείται ο ίδιος χαρακτήρας όπως στο πιο δεξιά suff(k)?
- Σε αυτή την περίπτωση μπορούμε να μετατοπίσουμε το P ως προς το συνολικό μέγεθός του
- Για παράδειγμα, αν k = 3 και P=DBCBAB μπορεί να μετατοπισθεί κατά 6 χαρακτήρες



String Matching (39/42)

The Boyer-Moore Algorithm

- Δυστυχώς όμως η μετατόπιση ως προς το συνολικό μήκος του P όταν δεν υπάρχει άλλη εμφάνιση του suff(k) στην οποία δεν προηγείται ο ίδιος χαρακτήρας όπως στο πιο δεξιά suff(k), δεν είναι πάντα σωστή
- Για το P=ABCBAB και k=3, η μετατόπιση κατά 6 χαρακτήρες, θα χάσει μια υπο-συμβολοσειρά που ξεκινά με το AB ευθυγραμμισμένο με τους τελευταίους δύο χαρακτήρες του T



String Matching (40/42)

The Boyer-Moore Algorithm

- Για να αποφύγουμε το πρόβλημα αυτό, πρέπει να βρούμε το μεγαλύτερο prefix μεγέθους l < k που ταιριάζει με το suffix ιδίου μήκους l
- Αν υπάρχει τέτοιο πρόθεμα, το μήκος μετατόπισης d_2 υπολογίζεται ως η απόσταση του prefix και του suffix
- Εναλλακτικά, το d_2 τίθεται ίσο με το μήκος του P
- Παράδειγμα

k	**pattern**	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

String Matching (41/42)

The Boyer-Moore Algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

String Matching (42/42)

The Boyer-Moore Algorithm

- Παράδειγμα εύρεσης BAOBAB

c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

k	**pattern**	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(K) - 0 = 6$ B A O B A B

$d_1 = t_1(\_) - 2 = 4$ B A O B A B

$d_2 = 5$ $d_1 = t_1(\_) - 1 = 5$

$d = \max\{4, 5\} = 5$ $d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

P, NP, NP-Complete Problems

P, N, NP-Complete Problems (1/24)

Λέμε πως ένας αλγόριθμος επιλύει ένα πρόβλημα σε πολυωνυμικό χρόνο όταν στη χειρότερη περίπτωση η πολυπλοκότητά του είναι **O(p(n))** όπου p(n) είναι ένα πολυώνυμο του μεγέθους εισόδου n

Τα προβλήματα που μπορούν να επιλυθούν σε πολυωνυμικό χρόνο ονομάζονται **ανιχνεύσιμα (tractable)**

Τα προβλήματα που δεν μπορούν να επιλυθούν σε πολυωνυμικό χρόνο ονομάζονται **μη ανιχνεύσιμα (intractable)**

Η επίλυση των μη ανιχνεύσιμων προβλημάτων δεν μπορούν να επιλυθούν σε ένα λογικό πλαίσιο χρόνου εκτός αν η έκδοση που θέλουμε να επιλύσουμε είναι μικρού μεγέθους

P, N, NP-Complete Problems (2/24)

Σχετικά με τα πολυώνυμα που περιγράφουν την πολυπλοκότητα των ανιχνεύσιμων προβλημάτων, υπάρχει ένας πολύ μεγάλος αριθμός πολυωνύμων αλλά πρακτικά συναντάμε **μέχρι τρίτου βαθμού**

Συνήθως, τα πολυώνυμα που περιγράφουν χρόνους εκτέλεσης **δεν έχουν υψηλές τιμές στους συντελεστές** των όρων

Τα πολυώνυμα έχουν κάποιες ωφέλιμες ιδιότητες όπως η **σύνθεση ή το άθροισμά τους που είναι πολυώνυμο** επίσης

Η επιλογή των πολυωνύμων οδήγησε στην εξέλιξη **της υπολογιστικής πολυπλοκότητας (computational complexity)** που ψάχνει να ταξινομήσει τα προβλήματα σύμφωνα με τη δυσκολία τους

P, N, NP-Complete Problems (3/24)

Τα περισσότερα προβλήματα που έχουμε δει είναι πολυωνυμικού χρόνου

Τα προβλήματα που μπορεί να επιλυθούν σε πολυωνυμικό χρόνο τα θεωρούμε μέρος ενός συνόλου το οποίο συμβολίζουμε με **P**

Η τάξη P είναι μια τάξη προβλημάτων που μπορεί να επιλυθούν σε πολυωνυμικό χρόνο από ένα (ντετερμινιστικό) αλγόριθμο. Η κλάση αυτή ονομάζεται πολυωνυμική.

P, N, NP-Complete Problems (4/24)

Ο ορισμός στοχεύει κυρίως σε προβλήματα απόφασης (η λύση είναι ένα ναι ή όχι) για τους ακόλουθους λόγους

- Είναι λογικό να αποκλείουμε προβλήματα που δεν επιλύονται σε πολυωνυμικό χρόνο **λόγω του εκθετικού μεγέθους της εξόδου** – παράδειγμα: η δημιουργία υποσυνόλων ενός σύνολου μέσω του υπολογισμού των αναδιατάξεων n αντικειμένων
- Πολλά σημαντικά προβλήματα δεν είναι προβλήματα απόφασης αλλά **μπορεί να αναχθούν σε μια σειρά προβλημάτων απόφασης** που μπορούν να μελετηθούν πιο εύκολα

P, N, NP-Complete Problems (5/24)

Δυστυχώς, όλα τα προβλήματα απόφασης δεν μπορούν να επιλυθούν σε πολυωνυμικό χρόνο

Στην πραγματικότητα κάποια προβλήματα δεν μπορούν να λυθούν από κάποιον αλγόριθμο

Τα προβλήματα απόφασης διακρίνονται σε: **αποφασίσιμα (decidable)** και **μη αποφασίσιμα (undecidable)** ανάλογα με το αν λύνονται από κάποιο αλγόριθμο ή όχι

P, N, NP-Complete Problems (6/24)

Παράδειγμα μη αποφασίσιμου προβλήματος

- The **halting problem by Alan Turing**
- Δοσμένου ενός προγράμματος και μιας εισόδου για αυτό, να καθορίσουμε αν το πρόγραμμα θα σταματήσει κάποτε για τη συγκεκριμένη είσοδο ή θα συνεχίσει να εκτελείται επ’ άπειρον
- Ας υποθέσουμε ότι όντως υπάρχει ένας αλγόριθμος που να το λύνει
- Συνεπώς, για κάθε πρόγραμμα P και είσοδο I έχουμε:

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I \\ 0, & \text{if program } P \text{ does not halt on input } I \end{cases}$$

P, N, NP-Complete Problems (7/24)

- Μπορούμε να φανταστούμε το πρόγραμμα σαν είσοδο στον εαυτό του και χρησιμοποιούμε την έξοδο του αλγορίθμου A για το ζεύγος (P,P) για να δημιουργήσουμε ένα πρόγραμμα Q τέτοιο ώστε

$$Q(P) = \begin{cases} \text{halts,} & \text{if } A(P,\ P) = 0, \text{ i.e., if program } P \text{ does not halt on input } P \\ \text{does not halt,} & \text{if } A(P,\ P) = 1, \text{ i.e., if program } P \text{ halts on input } P \end{cases}$$

- Αν αντικαταστήσουμε το Q στο P θα έχουμε:

$$Q(Q) = \begin{cases} \text{halts,} & \text{if } A(Q,\ Q) = 0, \text{ i.e., if program } Q \text{ does not halt on input } Q \\ \text{does not halt,} & \text{if } A(Q,\ Q) = 1, \text{ i.e., if program } Q \text{ halts on input } Q \end{cases}$$

- Όμως, έχουμε φτάσει σε αντίθεση αφού καμία από τις δύο εξόδους για το Q δεν είναι πιθανή

P, N, NP-Complete Problems (8/24)

Το επόμενο ερώτημα που πρέπει να απαντήσουμε είναι: ***υπάρχουν προβλήματα που είναι αποφασίσιμα αλλά όχι ανιχνεύσιμα***;

Η απάντηση είναι, ***ναι αλλά ο αριθμός των παραδειγμάτων είναι μικρός***

Υπάρχουν πολλά προβλήματα για τα οποία δεν έχει βρεθεί αλγόριθμος πολυωνυμικού χρόνου ούτε η μη ύπαρξη τέτοιων αλγορίθμων έχει αποδειχθεί

P, N, NP-Complete Problems (9/24)

Παραδείγματα

- Το πρόβλημα εύρεσης ενός Hamiltonian circuit σε ένα γράφο
  - Εύρεση μονοπατιού που ξεκινά και τελειώνει στον ίδιο κόμβο περνώντας από όλους τους υπόλοιπους κόμβους μόνο μια φορά
- Το πρόβλημα του travelling salesman
  - Εύρεση της συντομότερης διαδρομής σε n πόλεις με γνωστές τις θετικές αποστάσεις μεταξύ τους
- Το πρόβλημα του σακιδίου (knapsack problem)
  - Εύρεση του πιο πολύτιμου υποσυνόλου n αντικειμένων που μπορούν να χωρέσουν σε ένα σακίδιο με δοσμένη χωρητικότητα

P, N, NP-Complete Problems (10/24)

Παραδείγματα (συνέχεια)

- Το πρόβλημα του διαχωρισμού (partition)
  - Δοσμένων n θετικών ακεραίων, να καθορίσουμε αν είναι δυνατόν να τους χωρίσουμε σε δύο αμοιβαία αποκλειόμενα σύνολα με το ίδιο άθροισμα
- The bin packing problem
  - Δοσμένων n αντικειμένων των οποίων το μέγεθος είναι θετικοί αριθμοί όχι μεγαλύτεροι από 1, να τα τοποθετήσουμε στο μικρότερο αριθμό κάδων μεγέθους 1
- Το πρόβλημα του χρωματισμού γράφων
  - Για ένα δοσμένο γράφο, να βρούμε το χρωματικό αριθμό του; Ο μικρότερος αριθμός χρωμάτων που είναι απαραίτητα να ανατεθούν στους κόμβους του γράφου έτσι ώστε οι γειτονικοί κόμβοι να έχουν διαφορετικό χρώμα

P, N, NP-Complete Problems (11/24)

Παραδείγματα (συνέχεια)

- The integer linear programming problem
  - Να βρούμε τη μέγιστο ή την ελάχιστη τιμή μιας γραμμικής συνάρτησης που έχει ένα σύνολο ακεραίων μεταβλητών με δεδομένο ένα πεπερασμένο σύνολο περιορισμών (στη μορφή γραμμικών εξισώσεων και ανισώσεων)

Το κοινό χαρακτηριστικό σε αυτά τα προβλήματα είναι έχουν μια εκθετική αύξηση των επιλογών λύσης ως συνάρτηση του μεγέθους του προβλήματος n

Επίσης, κάποια από αυτά μπορεί να επιλυθούν σε πολυωνυμικό χρόνο

P, N, NP-Complete Problems (12/24)

Ένα άλλο χαρακτηριστικό των προβλημάτων απόφασης είναι ότι παρά το γεγονός ότι η επίλυσή τους είναι δύσκολη υπολογιστικά, ο έλεγχος της λύσης μπορεί να γίνει σε πολυωνυμικό χρόνο

Για παράδειγμα, είναι εύκολο να ελέγξουμε αν μια προτεινόμενη σειρά κόμβων για ένα Hamiltonian circuit είναι μια αποδεκτή λύση

Το μόνο που χρειάζεται είναι να ελέγξουμε ότι η λύση αποτελείται από n+1 κόμβους, όλοι οι ενδιάμεσοι κόμβοι είναι διακριτοί μεταξύ τους, ο τελευταίος είναι ίδιος με τον πρώτο και όλοι οι κόμβοι συνδέονται με μια ακμή

P, N, NP-Complete Problems (13/24)

Ορισμός

- Ένας **μη ντετερμινιστικός (nondeterministic)** αλγόριθμος είναι μια δυο βημάτων διαδικασία που παίρνει σαν είσοδο ένα στιγμιότυπο I ενός προβλήματος απόφασης και εκτελεί τα ακόλουθα:
  - **Μη ντετερμινιστική φάση (υπόθεση - guessing)**: ένα αλφαριθμητικό S παράγεται ως μια υποψήφια λύση για το στιγμιότυπο I
  - **Ντετερμινιστική φάση (επιβεβαίωση - verification):** Ένας ντετερμινιστικός αλγόριθμος παίρνει τα I και S σαν είσοδο και εξάγει την απάντηση ναι αν το S αναπαριστά μια λύση για το στιγμιότυπο I

Λέμε πως ένας μη ντετερμινιστικός αλγόριθμος λύνει ένα πρόβλημα απόφασης **όταν και μόνο όταν για κάθε στιγμιότυπο που απαιτεί θετική απάντηση, επιστρέφει θετικό αποτέλεσμα**

P, N, NP-Complete Problems (14/24)

Ένας μη ντετερμιστικός αλγόριθμος λέμε ότι είναι **πολυωνυμικός μη ντετερμινιστικός (nondeterministic polynomial)** εφόσον η επίδοση χρόνου της επιβεβαίωσης / επαλήθευσης είναι πολυωνυμικός

Ορισμός

- **Η τάξη / κλάση NP είναι η τάξη των προβλημάτων απόφασης που μπορεί να επιλυθούν από μη ντετερμινιστικούς πολυωνυμικούς αλγορίθμους. Αυτή η τάξη προβλημάτων ονομάζεται μη ντετερμινιστική πολυωνυμική (nondeterministic polynomial)**

P, N, NP-Complete Problems (15/24)

Πολλά προβλήματα ανήκουν στην τάξη NP

Η τάξη NP περιλαμβάνει τα προβλήματα που ανήκουν στην τάξη P

$$P \subseteq NP$$

Αν ένα πρόβλημα ανήκει στην τάξη P, μπορούμε να υιοθετήσουμε ένα ντετερμινιστικό αλγόριθμο που το επιλύει στη φάση της επιβεβαίωσης / επαλήθευσης ενός μη ντετερμινιστικού αλγορίθμου που απλά αγνοεί το S που εξάγεται από τη μη ντετερμινιστική φάση

Η τάξη NP περιλαμβάνει: the Hamiltonian circuit, the partition problem, decision versions of the traveling salesman, the knapsack problem, graph coloring, combinatorial problems

P, N, NP-Complete Problems (16/24)

Ερώτημα προς απάντηση από τους επιστήμονες της Θεωρητικής Πληροφορικής

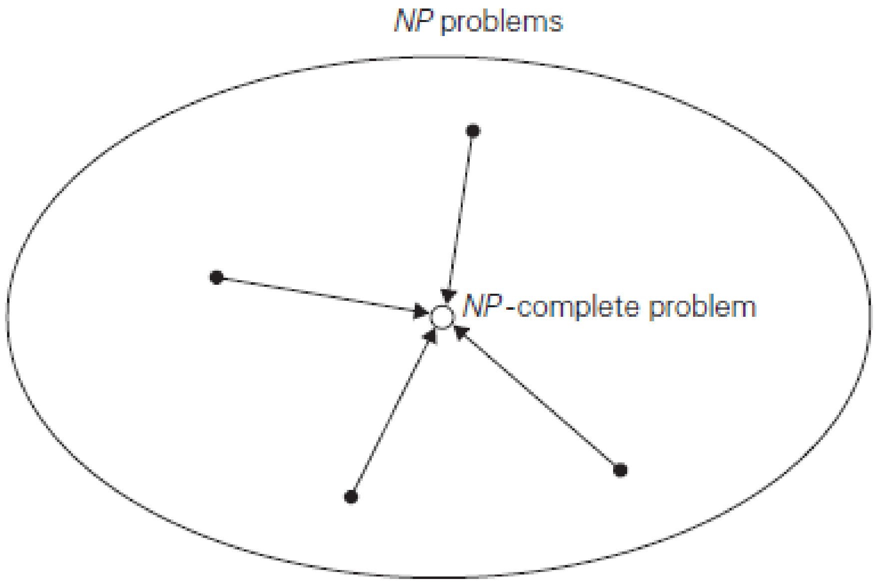
- ***Η τάξη P είναι ένα κατάλληλο υποσύνολο της τάξης NP ή οι δύο τάξεις είναι στην πραγματικότητα οι ίδιες;***

Αν P=NP, τότε κάθε ένα από τα εκατοντάδες δύσκολα προβλήματα συνδυαστικής μπορεί να επιλυθεί σε πολυωνυμικό χρόνο ακόμα και αν οι Επιστήμονες έχουν αποτύχει να βρουν τους αντίστοιχους αλγορίθμους

Επιπρόσθετα, πολλά προβλήματα είναι γνωστό πως είναι NP-complete που προσθέτει αμφιβολίες για τη σχέση P=NP

P, N, NP-Complete Problems (17/24)

Ένα **NP-complete** πρόβλημα, είναι ένα πρόβλημα το οποίο είναι τόσο δύσκολο όσο οποιοδήποτε άλλο πρόβλημα που ανήκει στη τάξη NP επειδή, εξ’ ορισμού, οποιοδήποτε άλλο πρόβλημα στην NP μπορεί να ‘ελαττωθεί’ σε πολυωνυμικό χρόνο



P, N, NP-Complete Problems (18/24)

Ορισμός

- **Ένα πρόβλημα απόφασης D1 λέμε ότι είναι πολυωνυμικά αναγώγιμο (polynomially reducible) σε ένα πρόβλημα D2, αν υπάρχει μια συνάρτηση f που μετασχηματίζει το D1 σε στιγμιότυπα του D2 τέτοια ώστε:**
  - **H f αντιστοιχίζει όλα τα θετικά στιγμιότυπα (απάντηση: ναι) του D1 στα θετικά στιγμιότυπα του D2 και όλα τα μη δυνατά στιγμιότυπα του D1 σε μη δυνατά στιγμιότυπα του D2**
  - **H f υπολογίζεται από ένα πολυωνυμικού χρόνου αλγόριθμο**

Ο ορισμός μας υποδεικνύει ότι αν ένα πρόβλημα D1 είναι πολυωνυμικά αναγώγιμο σε ένα πρόβλημα D2 που μπορεί να επιλυθεί σε πολυωνυμικό χρόνο, τότε το ίδιο ισχύει και για το D1

P, N, NP-Complete Problems (19/24)

Ορισμός

- **Ένα πρόβλημα απόφασης D λέμε ότι είναι NP-complete εάν:**
  - **ανήκει στην τάξη NP**
  - **Κάθε πρόβλημα στην NP είναι πολυωνυμικά αναγώγιμο στο D**

Θα αποδείξουμε ότι το πρόβλημα εύρεσης του Hamiltonian circuit είναι πολυωνυμικά αναγώγιμο στην έκδοση απόφασης του traveling salesman problem

Το Hamiltonian circuit μπορεί να οριστεί σαν ένα πρόβλημα ύπαρξης ενός Hamiltonian circuit όχι μεγαλύτερου από ένα θετικό αριθμό m σε ένα δοσμένο γράφο με θετικά βάρη

P, N, NP-Complete Problems (20/24)

Μπορούμε να απεικονίσουμε ένα γράφο G ενός συγκεκριμένου στιγμιότυπου του Hamiltonian circuit προβλήματος σε ένα πλήρες γράφο με βάρη G' που αναπαριστά ένα στιγμιότυπο του traveling salesman problem

Στον G' αναθέτουμε 1 ως βάρος σε κάθε ακμή του G και προσθέτουμε 2 ανάμεσα σε κάθε ζεύγος μη γειτονικών κόμβων του G

Σαν πάνω όριο m του μήκους του Hamiltonian circuit παίρνουμε m=n, όπου n είναι το πλήθος των κόμβων στα G, G'

Ο μετασχηματισμός μπορεί να ολοκληρωθεί σε πολυωνυμικό χρόνο

P, N, NP-Complete Problems (21/24)

Έστω G είναι ένα θετικό στιγμιότυπο του Hamiltonian circuit

Τότε το G έχει ένα Hamiltonian circuit και η εικόνα του G’ θα έχει μήκος n το οποίο κάνει την εικόνα G’ ένα θετικό στιγμιότυπο της απόφασης του traveling salesman problem

Αντίστοιχα, αν έχουμε ένα Hamiltonian circuit μήκους όχι περισσότερο από n στο G’, τότε το μήκος του πρέπει να είναι n και το Hamiltonian circuit πρέπει να αποτελλείται από ακμές του G δημιουργώντας μια αντίστροφη εικόνα των θετικών στιγμιοτύπων του traveling salesman problem που θα είναι ένα θετικό στιγμιότυπο του προβλήματος της εύρεσης του Hamiltonian circuit

P, N, NP-Complete Problems (22/24)

Η έννοια της NP-completeness απαιτεί πολυωνυμική αναγωγή όλων (γνωστών και αγνώστων) των NP προβλημάτων στο πρόβλημα που εξετάζουμε

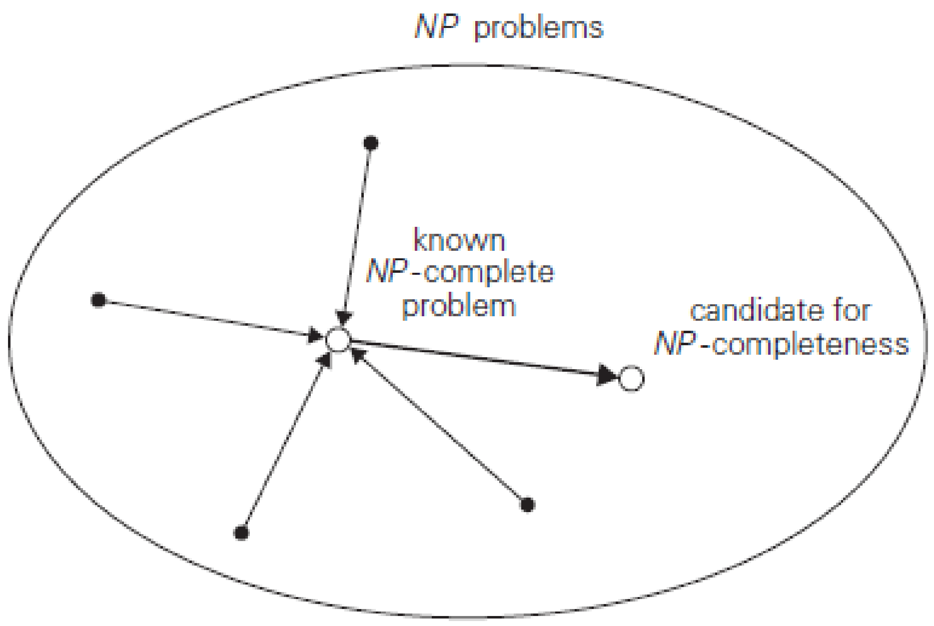
Ένα σύνολο NP-complete προβλημάτων έχουν βρεθεί

Μια δυναμική λίστα τέτοιων προβλημάτων μπορεί κανείς να δει εδώ https://en.wikipedia.org/wiki/List_of_NP-complete_problems

P, N, NP-Complete Problems (23/24)

Για την ανάδειξη ενός προβλήματος ως NP-complete απαιτεί δύο βήματα:

- Πρώτα αποδεικνύουμε ότι το πρόβλημα είναι NP (τυπικά αυτό το βήμα είναι εύκολο)
- Έπειτα αποδεικνύουμε ότι κάθε πρόβλημα στην NP είναι αναγώγιμο στο πρόβλημα που εξετάζουμε σε πολυωνυμικό χρόνο
  - Λόγω της μεταβατικότητας της πολυωνυμικής αναγωγής, το βήμα μπορεί να γίνει αποδεικνύοντας ότι ένα γνωστό NP-complete πρόβλημα μπορεί να αναχθεί σε πολυωνυμικό χρόνο στο πρόβλημα που εξετάζουμε



P, N, NP-Complete Problems (24/24)

Η αναγωγή μέσω ενός γνωστού NP-complete προβλήματος είναι πολύ πιο εύκολη διαδικασία από το να ανάγουμε όλα τα NP προβλήματα στο πρόβλημα που εξετάζουμε

Ο ορισμός της NP-completeness μας δείχνει ότι αν υπάρχει ένας ντετερμινιστικός πολυωνυμικού χρόνου αλγόριθμος για ένα NP-complete πρόβλημα, τότε κάθε NP-complete πρόβλημα μπορεί να λυθεί σε πολυωνυμικό χρόνο μέσω ενός ντετερμινιστικού αλγορίθμου

Χειρισμός των Περιορισμών των Αλγορίθμων

Χειρισμός Περιορισμών (1/32)

Κάποια από τα προβλήματα που είναι δύσκολο να λυθούν αλγοριθμικά, είναι πολύ σημαντικά, συνεπώς θα πρέπει να βρεθεί κάποια λύση

Χρησιμοποιούμε τις τεχνικές **back-tracking** και **branch-and-bound** για να λύσουμε κάποια στιγμιότυπα των προβλημάτων

Και οι δύο τεχνικές βασίζονται στη δημιουργία του **space-state tree**

Οι κόμβοι του δένδρου αντιπροσωπεύουν συγκεκριμένες επιλογές για την επίλυση του προβλήματος

Και οι δύο τεχνικές τερματίζουν ένα κόμβο όταν αυτό δεν μπορεί να εγγυηθεί την επίλυση του προβλήματος

Χειρισμός Περιορισμών (2/32)

Η backtracking χρησιμοποιείται σε προβλήματα που δεν σχετίζονται με βελτιστοποίηση

Η branch-and-bound σχετίζεται και υιοθετείται σε προβλήματα βελτιστοποίησης

Στην backtracking οι κόμβοι του δένδρου κατασκευάζονται σε μια σειρά depth-first

Στην branch-and-bound υιοθετούνται αρκετές τεχνικές μια από τις οποίες είναι ο κανόνας best-first

Χειρισμός Περιορισμών (3/32)

Backtracking

- Η βασική ιδέα είναι να κατασκευάζουμε μια τμηματική λύση τη φορά και να την επαληθεύουμε
- Αν μια τμηματική λύση μπορεί να επεκταθεί περαιτέρω χωρίς να παραβιάζουμε τους διάφορους περιορισμούς, την επιλέγουμε παίρνοντας το επόμενο αποδεκτό στοιχείο της λύσης
- Αν δεν υπάρχει μια αποδεκτή τμηματική επέκταση της λύσης, τότε δεν εξετάζεται καμμία επέκταση
- Σε αυτή την περίπτωση ο αλγόριθμος οπισθοχωρεί ώστε να αντικαταστήσει το τελευταία αποδεκτό στοιχείο της λύσης με την επόμενη επιλογή

Χειρισμός Περιορισμών (4/32)

Backtracking

- Ο αλγόριθμος κατασκευάζει το λεγόμενο **state-space tree**
- Η ρίζα του δένδρου είναι μια αρχική κατάσταση πριν ξεκινήσει η αναζήτηση των τμημάτων της λύσης
- Οι κόμβοι στο πρώτο επίπεδο αντιστοιχούν στις επιλογές που είναι διαθέσιμες για το πρώτο τμήμα της λύσης, οι κόμβοι στο δεύτερο επίπεδο είναι οι επιλογές για το δεύτερο στοιχείο, κ.ο.κ.
- Ένας κόμβος λέμε ότι είναι **υποσχόμενος (promising)** αν αντιστοιχεί σε μια τμηματική λύση που μπορεί να οδηγήσει σε μια πλήρη λύση, αλλιώς καλείται **μη υποσχόμενος (nonpromising)**
- Τα φύλλα αντιστοιχούν είτε σε μη υποσχόμενες λύσεις ή σε αποδεκτές λύσεις

Χειρισμός Περιορισμών (5/32)

Backtracking

- Αν ένας κόμβος είναι υποσχόμενος, τότε κατασκευάζονται τα παιδιά του
- Σε διαφορετική περίπτωση, ο αλγόριθμος οπισθοχωρεί ένα επίπεδο πάνω και για να εξετάσει την επόμενη στη σειρά επιλογή κ.ο.κ.
- Αν ο αλγόριθμος φτάσει σε μια αποδεκτή λύση, είτε σταματά ή συνεχίζει να αναζητά άλλες αποδεκτές λύσεις

Χειρισμός Περιορισμών (6/32)

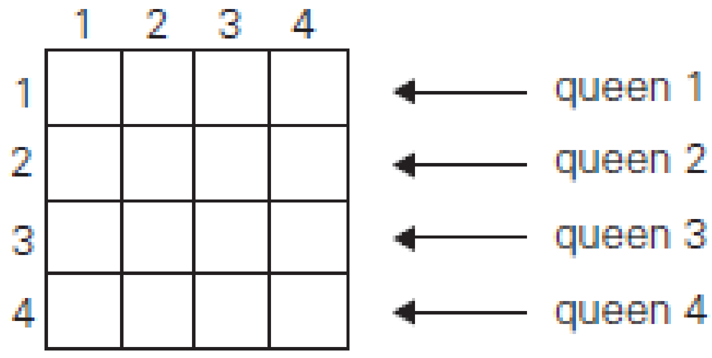
n-Queens Example

- Καλούμαστε να τοποθετήσουμε σε ένα σκάκι nXn, n βασίλισσες, έτσι ώστε να μην μπορούν 2 βασίλισσες να επιτεθούν η μια στην άλλη (ίδια γραμμή, ίδια στήλη, ίδια διαγώνιο)
- Για n=1, η λύση είναι εύκολη
- Επίσης είναι εύκολο να δούμε πως δεν υπάρχει λύση για n=2, n=3
- Ας υποθέσουμε πως έχουμε n=4 και λύνουμε το πρόβλημα με backtracking
- Για n>=4, μια λύση μπορεί να βρεθεί σε γραμμικό χρόνο

Χειρισμός Περιορισμών (7/32)

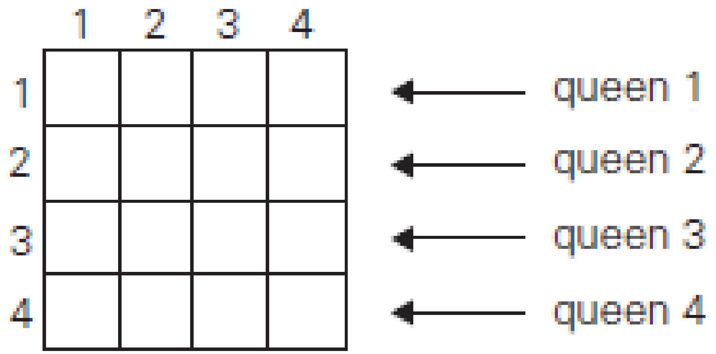
n-Queens Example

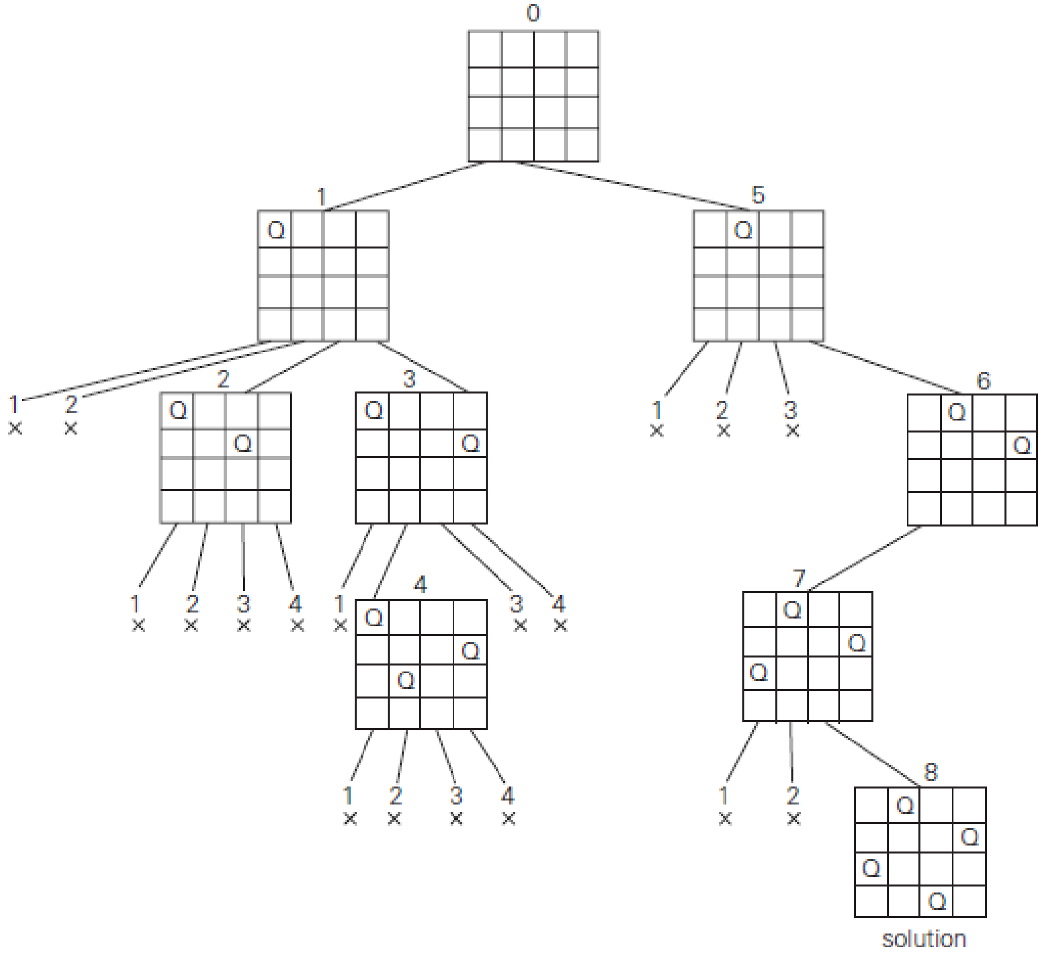
- Ξεκινάμε με άδειο σκάκι και τοποθετούμε την 1η βασίλισσα στην 1η θέση της γραμμής 1.
- Μετά τοποθετούμε τη 2η βασίλισσα (αφού προσπαθήσουμε στις στήλες 1,2) στο κελί (2,3)
- Αποδεικνύεται ότι αυτή η κίνηση δεν είναι αποδεκτή αφού δεν θα μπορεί να τοποθετηθεί η 3η βασίλισσα
- Οπισθοχωρούμε και και βάζουμε τη 2η βασίλισσα στο (2,4)
- Η 3η βασίλισα θα τοποθετηθεί στο (3,2) το οποίο είναι επίσης αδιέξοδο
- Οπσιθοχωρούμε και βάζουμε την 1η βασίλισσα στο (1,2)
- Η 2η θα πάει στο (2,4), η 3η στο (3,1) και η 4η στο (4,3)



Χειρισμός Περιορισμών (8/32)

n-Queens Example

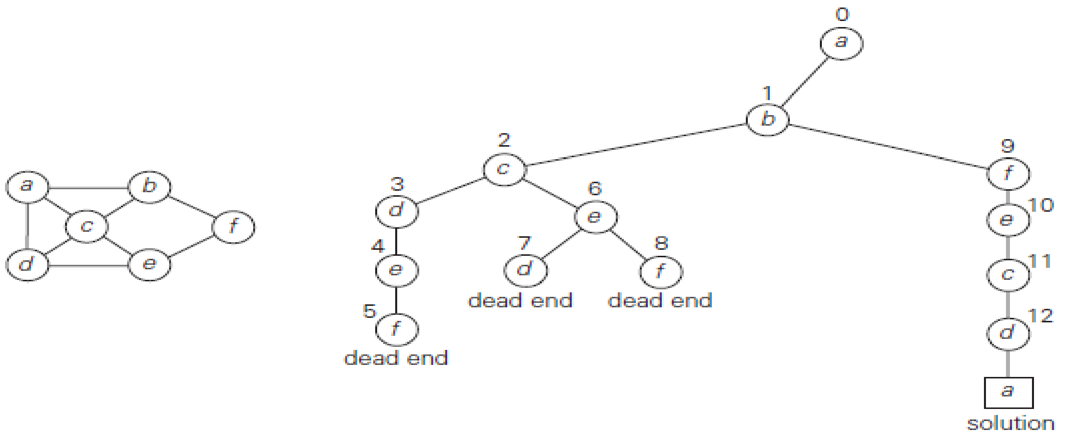




Χειρισμός Περιορισμών (9/32)

Hamiltonian circuit

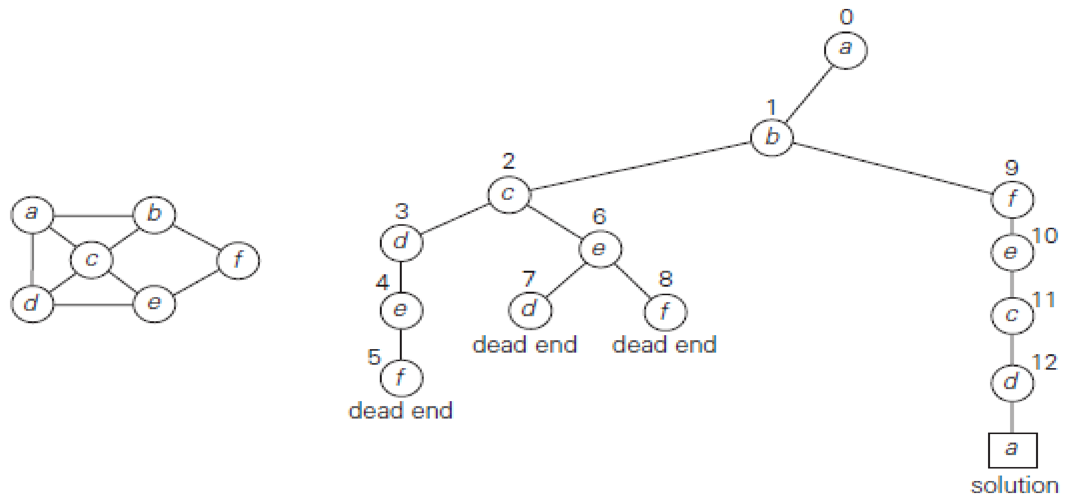
- Χωρίς βλάβη της γενικότητας θεωρούμε πως η διαδρομή ξεκινά από τον κόμβο a
- Θεωρούμε τον a ως τη ρίζα του δένδρου
- Το πρώτο στοιχείο της λύσης μας, εφόσον υπάρχει είναι ο πρώτος άμεσος κόμβος του Hamiltonian circuit



Χειρισμός Περιορισμών (10/32)

Hamiltonian circuit

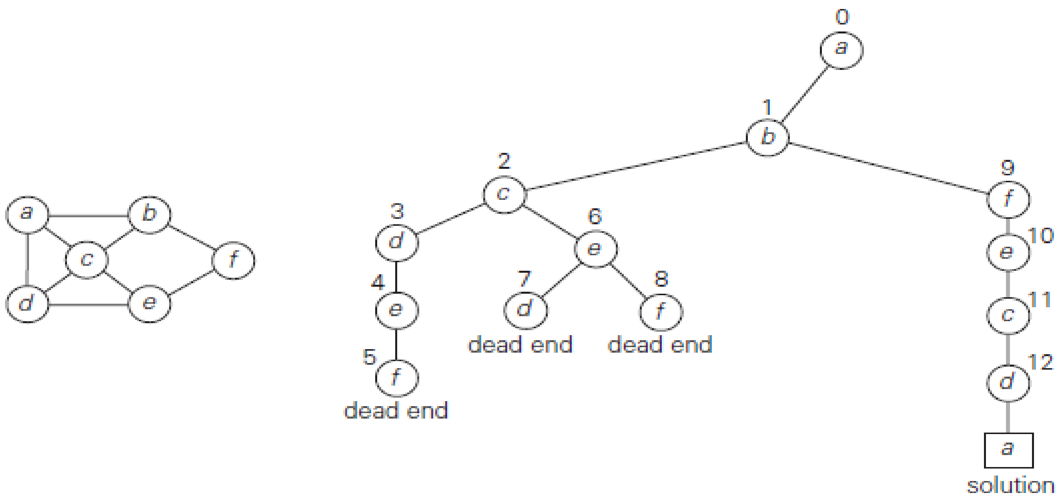
- Αν θεωρήσουμε την αλφαβητική σειρά για τους άμεσους γειτονικούς κόμβους, τότε προχωράμε στον επόμενο κόμβο της διαδρομής που είναι ο b
- Από τον b προχωράμε στον c



Χειρισμός Περιορισμών (11/32)

Hamiltonian circuit

- Μετά πάμε στον d, στον e και τελικά στον f που αποδεικνύεται ότι είναι αδιέξοδο
- Οπισθοχωρούμε και από τον f, στον e, στον d, στον c, κ.ο.κ.



Χειρισμός Περιορισμών (12/32)

Backtracking

- Γενικά η έξοδος του αλγορίθμου είναι ένα διάνυσμα (x_1, x_2, …, x_n) όπου οι συντεταγμένες x_i είναι ένα στοιχείο ενός πεπερασμένου συνόλου S_i
- Εξαρτάται από το πρόβλημα το αν όλα τα διανύσματα θα έχουν το ίδιο μήκος

```
ALGORITHM  Backtrack(X[1..i])
    //Gives a template of a generic backtracking algorithm
    //Input: X[1..i] specifies first i promising components of a solution
    //Output: All the tuples representing the problem's solutions
    if X[1..i] is a solution write X[1..i]
    else    //see Problem 9 in this section's exercises
        for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
            X[i + 1] ← x
            Backtrack(X[1..i + 1])
```

Χειρισμός Περιορισμών (13/32)

Backtracking

- Η τεχνική δεν είναι αποδοτική, γενικά
- Στη χειρότερη περίπτωση θα πρέπει να δημιουργήσουμε όλες τις πιθανές λύσεις
- Ελπίζουμε στο να μην ληφθούν υπόψιν αρκετές διαδρομές προς μη εφικτές λύσεις

ALGORITHM *Backtrack*$(X[1..i])$

```
//Gives a template of a generic backtracking algorithm
//Input: X[1..i] specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if X[1..i] is a solution write X[1..i]
else    //see Problem 9 in this section's exercises
    for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
        X[i + 1] ← x
        Backtrack(X[1..i + 1])
```

Χειρισμός Περιορισμών (14/32)

Backtracking

- Υπάρχει ένα σύνολο μεθόδων για να μειώσουμε το μέγεθος του δένδρου
- Ένας τρόπος είναι να εξερευνήσουμε τη συμμετρία που υπάρχει συχνά σε προβλήματα συνδυαστικά
- Η παρατήρηση αυτή μπορεί να μειώσει στο μισό περίπου το πλήθος πιθανών λύσεων
- Για παράδειγμα, η $1^η$ βασίλισσα δεν χρειάζεται να τοποθετηθεί στις τελευταίες n/2 στήλες

Χειρισμός Περιορισμών (15/32)

Backtracking

- Άλλος τρόπος είναι να προ-αναθέσουμε τιμές σε ένα ή περισσότερα στοιχεία μιας λύσης
- Για παράδειγμα μπορούμε να προ-ταξινομήσουμε κάποια δεδομένα

Χειρισμός Περιορισμών (16/32)

Backtracking

- Είναι δύσκολο να εκτιμήσουμε το μέγεθος του δένδρου
- Ο Knuth πρότεινε να δημιουργήσουμε ένα τυχαίο μονοπάτι από τη ρίζα μέχρι ένα φύλλο και χρησιμοποιώντας την πληροφορία σχετικά με το πλήθος των επιλογών που είναι διαθέσιμες όταν δημιουργούμε το μονοπάτι να εκτιμήσουμε το μέγεθος του δένδρου
- Έστω c_1 το πλήθος των τιμών για το πρώτο στοιχείο
- Επιλέγουμε τυχαία (πιθανότητα $1/c_1$) μια από αυτές
- Επαναλαμβάνουμε τη διαδικασία για τις τιμές c_2, κ.ο.κ.

Χειρισμός Περιορισμών (17/32)

Backtracking

- Υποθέτωντας πως στο επίπεδο i το πλήθος των κόμβων είναι c_i κατά μέσο όρο, η εκτίμηση του πλήθους των κόμβων είναι: $1+c_1+c_1c_2+\ldots+c_1c_2..c_n$
- Αν δημιουργήσουμε πολλές τυχαίες ακολουθίες και πάρουμε το μέσο όρο του μεγέθους, θα έχουμε μια εκτίμηση για το τελικό μέγεθος του δένδρου

Χειρισμός Περιορισμών (18/32)

Branch-and-bound

- Ένα πρόβλημα βελτιστοποίησης αναζητά τη μεγιστοποίηση ή ελαχιστοποίηση κάποιας συνάρτησης με βάση κάποιους περιορισμούς
- Σε σχέση με το backtracking η μέθοδος branch-and-bound απαιτεί δύο επιπρόσθετα βήματα:
  - Ένα τρόπο να καθορίσει για κάθε κόμβο του δένδρου ένα όριο της βέλτιστης τιμής της συνάρτησης βελτιστοποίησης σε σχέση με οποιαδήποτε λύση που μπορεί να προκύψει από το συγκεκριμένο κόμβο
  - Την τιμή της βέλτιστης λύσης μέχρι στιγμής
- Αν αυτές οι πληροφορίες είναι διαθέσιμες, μπορούμε να συγκρίνουμε το όριο του κάθε κόμβου με τη βέλτιστη λύση μέχρι στιγμής

Χειρισμός Περιορισμών (19/32)

Branch-and-bound

- Αν το όριο δεν είναι καλύτερο από τη βέλτιστη τιμή, ο κόμβος είναι μη υποσχόμενος και μπορεί να τερματιστεί
- Τερματίζουμε την αναζήτηση σε ένα κόμβο όταν:
  - Η τιμή του ορίου του **κόμβου δεν είναι καλύτερη από την τιμή της βέλτιστης λύσης** μέχρι στιγμής
  - Ο κόμβος αναπαριστά μια **μη εφικτή λύση** επειδή οι περιορισμοί του προβλήματος έχουν ήδη παραβιαστεί
  - Το υποσύνολο των εφικτών λύσεων που απεικονίζονται από τον κόμβο **είναι ένα απλό σημείο** – συγκρίνουμε την τιμή της συνάρτησης για αυτή την εφικτή λύση με τη βέλτιστη λύση μέχρι στιγμής

Χειρισμός Περιορισμών (20/32)

The Assignment Problem

- Εφαρμόζουμε την branch-and-bound τεχνική στο πρόβλημα της ανάθεσης
- Θέλουμε να αναθέσουμε σε n άτομα, n εργασίες
- Υιοθετούμε τον nXn πίνακα κόστους
- Το κόστος οποιαδήποτε λύσης δεν μπορεί να είναι μικρότερο από το άθροισμα των μικρότερων στοιχείων σε κάθε γραμμή του πίνακα, 2+3+1+4=10
- Αυτό το κόστος μπορεί να μην αντιστοιχεί σε μια αποδεκτή λύση

$$C = \begin{array}{cc} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} & \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} & \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array} \end{array}$$

Χειρισμός Περιορισμών (21/32)

The Assignment Problem

- Εφαρμόζουμε την ίδια λογική σε κάθε τμηματική λύση
- Για παράδειγμα, επιλέγοντας την ανάθεση (α,1) θα προκύψει ένα ελάχιστο κόστος 9+3+1+4=17
- Σε αντίθεση με την backtracking, δημιουργούμε όλα τα παιδιά της πιο υποσχόμενης λύσης ανάμεσα στους μη τερματικούς κόμβους
- Συγκρίνουμε τα ελάχιστα όρια κάθε κόμβου
- Το καλύτερο όριο αντιστοιχεί στον πιο πολλά υποσχόμενο κόμβο

$$\begin{array}{cc} & \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ C = & \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array} \end{array}$$

Χειρισμός Περιορισμών (22/32)

The Assignment Problem

- Η συγκεκριμένη στρατηγική ονομάζεται **best-first branch-and-bound**

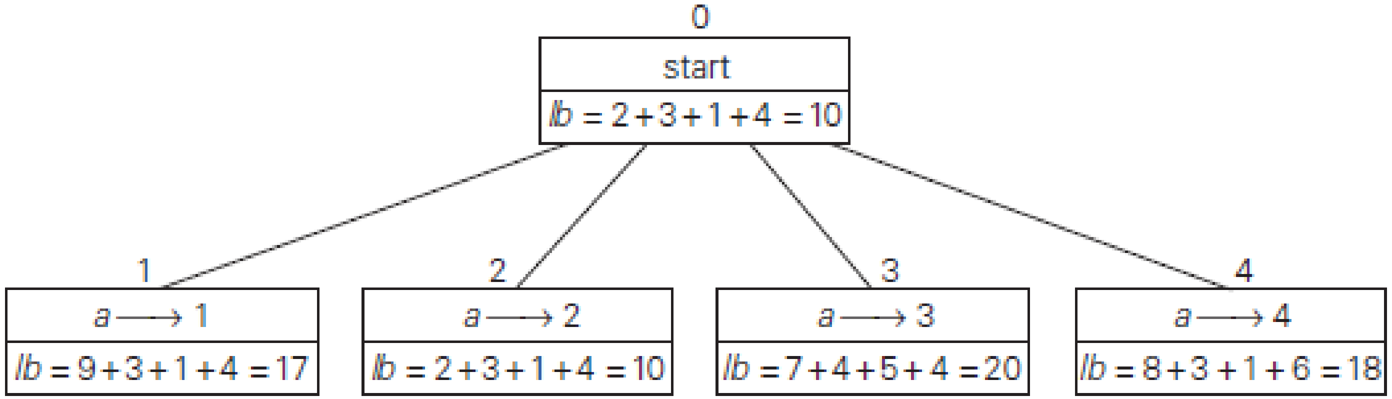
$$
\begin{array}{cc}
 & \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\
C = & \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}
\end{array}
\begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}
$$

Χειρισμός Περιορισμών (23/32)

The Assignment Problem

- Ξεκινάμε από τη ρίζα που δεν περιλαμβάνει κάποιο στοιχείο από τον πίνακα κόστους
- Το ελάχιστο κόστος είναι 10
- Δημιουργούμε 4 φύλλα: κόμβοι 1-4

$$C = \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

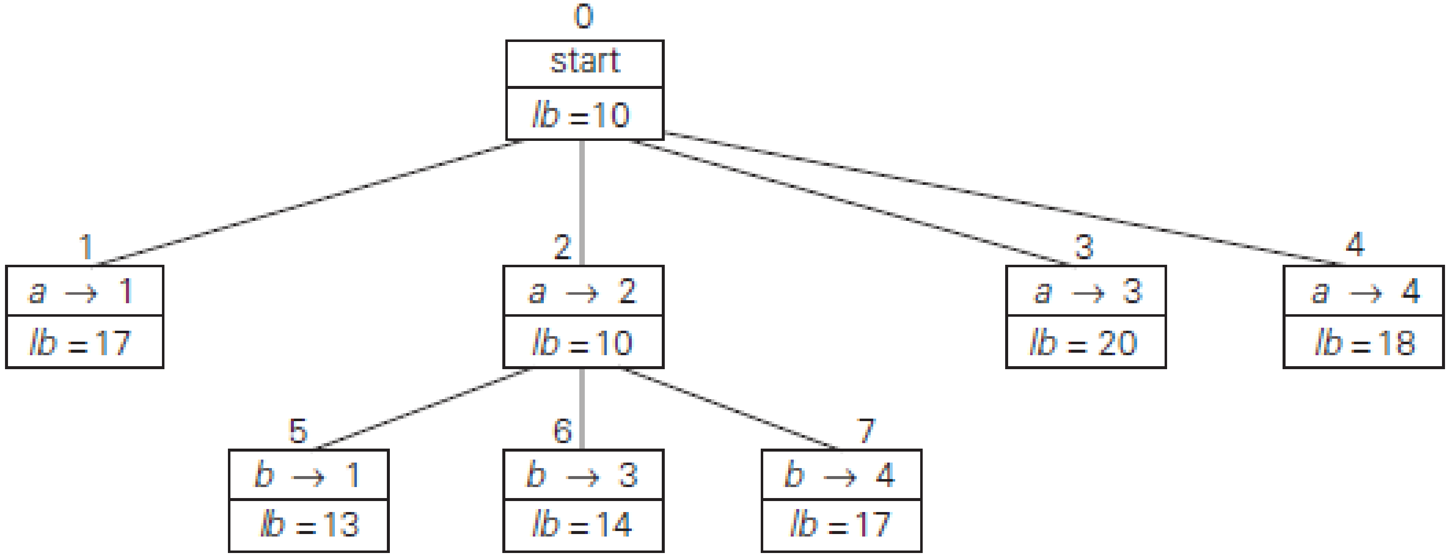


Χειρισμός Περιορισμών (24/32)

The Assignment Problem

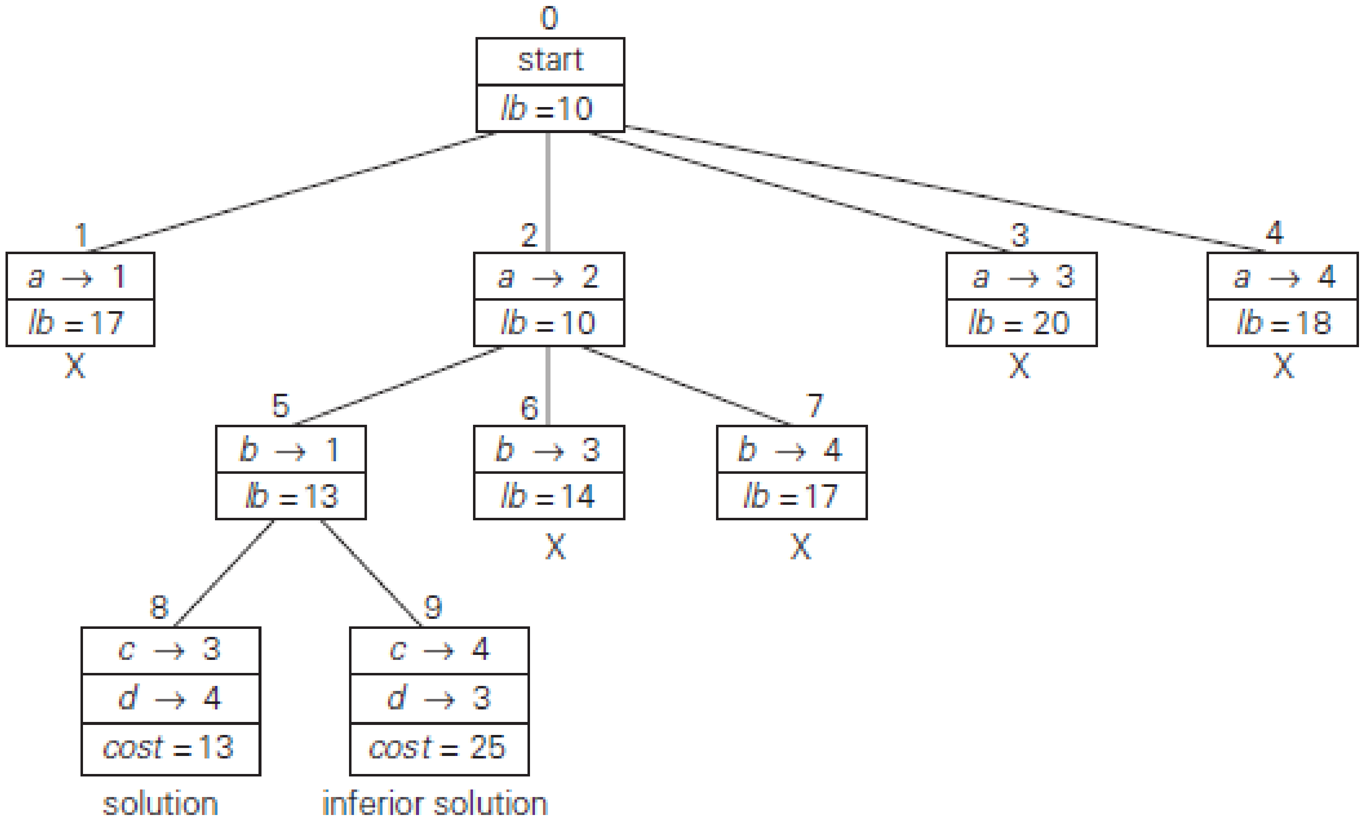
$$C = \begin{array}{c} \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array}$$

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$



Χειρισμός Περιορισμών (25/32)

The Assignment Problem



$$
\begin{array}{c}
\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\
= \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}
\begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}
\end{array}
$$

Χειρισμός Περιορισμών (26/32)

The knapsack problem

- Έχουμε n αντικείμενα βάρους w_i και αξίας $υ_i$ και ένα σακίδιο χωρητικότητας W
- Πρέπει να βρούμε τα μεγαλύτερης αξίας αντικείμενα που χωράνε στο σακίδιο
- Είναι βολικό να ταξινομήσουμε τα αντικείμενα σε φθίνουσα σειρά αξίας προς βάρος – οπότε το πρώτο αντικείμενο έχει τον καλύτερο λόγο
  $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
- Το δένδρο κατασκευάζεται σαν ένα δυαδικό δένδρο αναζήτησης

Χειρισμός Περιορισμών (27/32)

The knapsack problem

- Κάθε κόμβος το i επίπεδο αναπαριστά όλα τα υποσύνολα των αντικειμένων που περιλαμβάνουν μια συγκεκριμένη επιλογή από τα πρώτα i αντικείμενα
- Η συγκεκριμένη επιλογή καθορίζεται μοναδικά μέσω της διαδρομής από τη ρίζα μέχρι ένα κόμβο
- Μια διακλάδωση αριστερά μας δείχνει ότι συμπεριλαμβάνουμε το αντικείμενο ενώ η διασταύρωση προς τα δεξιά μας δείχνει ότι δεν συμπεριλαμβάνουμε το αντικείμενο
- Καταγράφουμε το συνολικό βάρος w και τη συνολική αξία υ μαζί με ένα άνω όριο **ub** της τιμής οποιουδήποτε υποσυνόλου που μπορεί να προκύψει προσθέτωντας μηδέν ή περισσότερα αντικείμενα στην τρέχουσα επιλογή

Χειρισμός Περιορισμών (28/32)

The knapsack problem

- Ένας απλός τρόπος υπολογισμού του ub είναι να προσθέσουμε στο υ τη συνολική τιμή των αντικειμένων που έχουν ήδη επιλεγεί, το γινόμενο της εναπομείνουσας χωρητικότητας του σακιδίου W-w και τον καλύτερο λόγο αξία προς βάρος από τα εναπομείναντα αντικείμενα

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

Χειρισμός Περιορισμών (29/32)

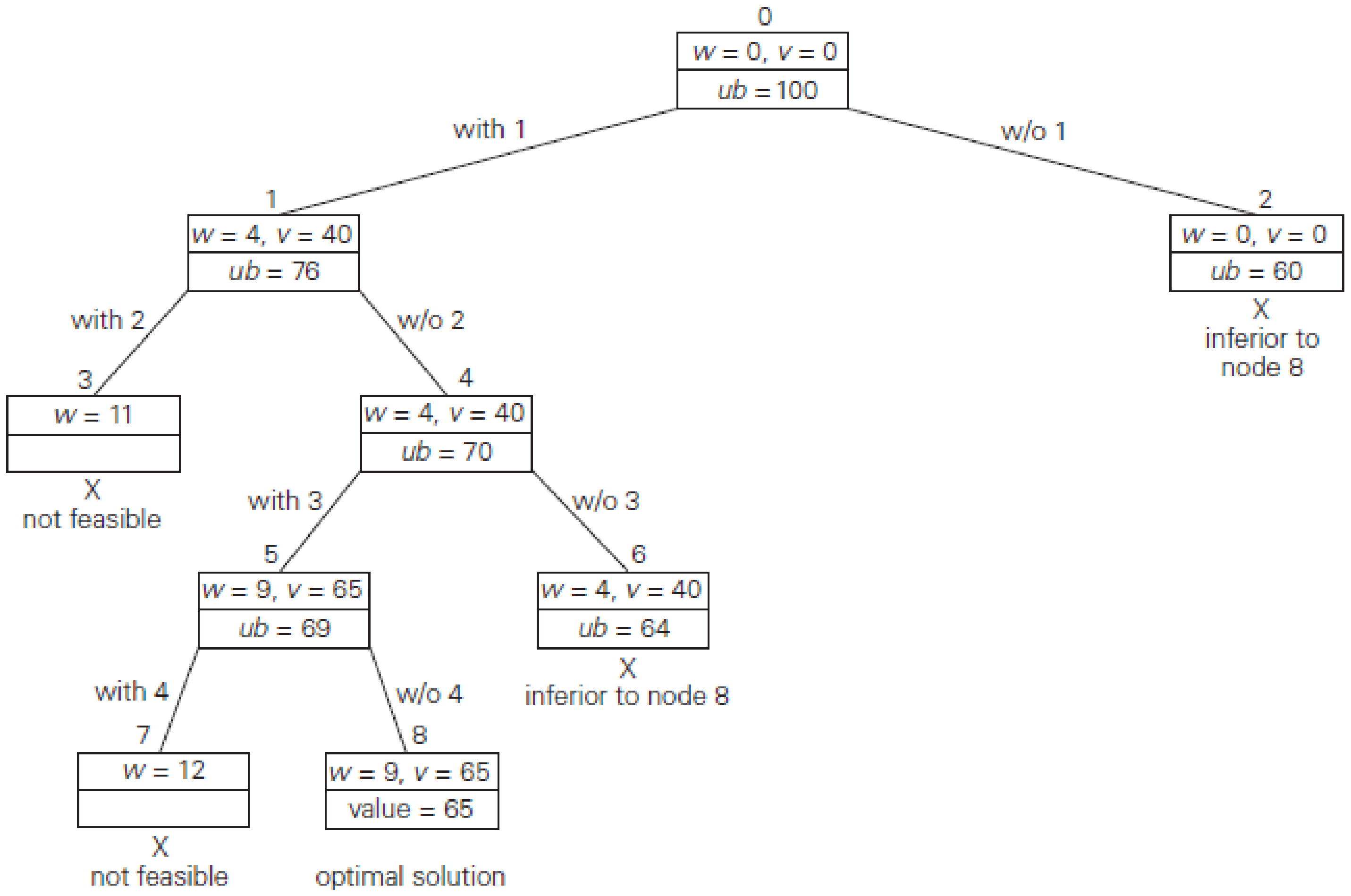
The knapsack problem

- Παράδειγμα:

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$



Χειρισμός Περιορισμών (30/32)

The traveling salesman problem

- Προσπαθούμε να ορίσουμε ένα κάτω όριο για το μήκος της διαδρομής
- Ένας απλός τρόπος είναι να βρούμε τη μικρότερη απόσταση και να την πολλαπλασιάσουμε με το πλήθος των πόλεων n
- Άλλος τρόπος είναι να το υπολογίσουμε ως εξής:
  - Για κάθε πόλη i βρίσκουμε το άθροισμα s_i των αποστάσεων από την πόλη i στις δύο πιο κοντινές πόλεις
  - Αθροίζουμε όλες αυτές τις αποστάσεις (n αριθμοί) και διαιρούμε προς 2

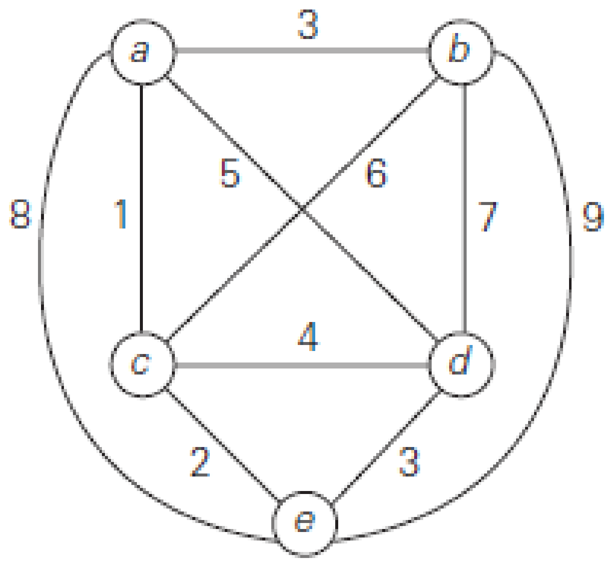
$$lb = \lceil s/2 \rceil$$

Χειρισμός Περιορισμών (31/32)

The traveling salesman problem

- Παράδειγμα
  - Για το διπλανό γράφο έχουμε

$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14$$

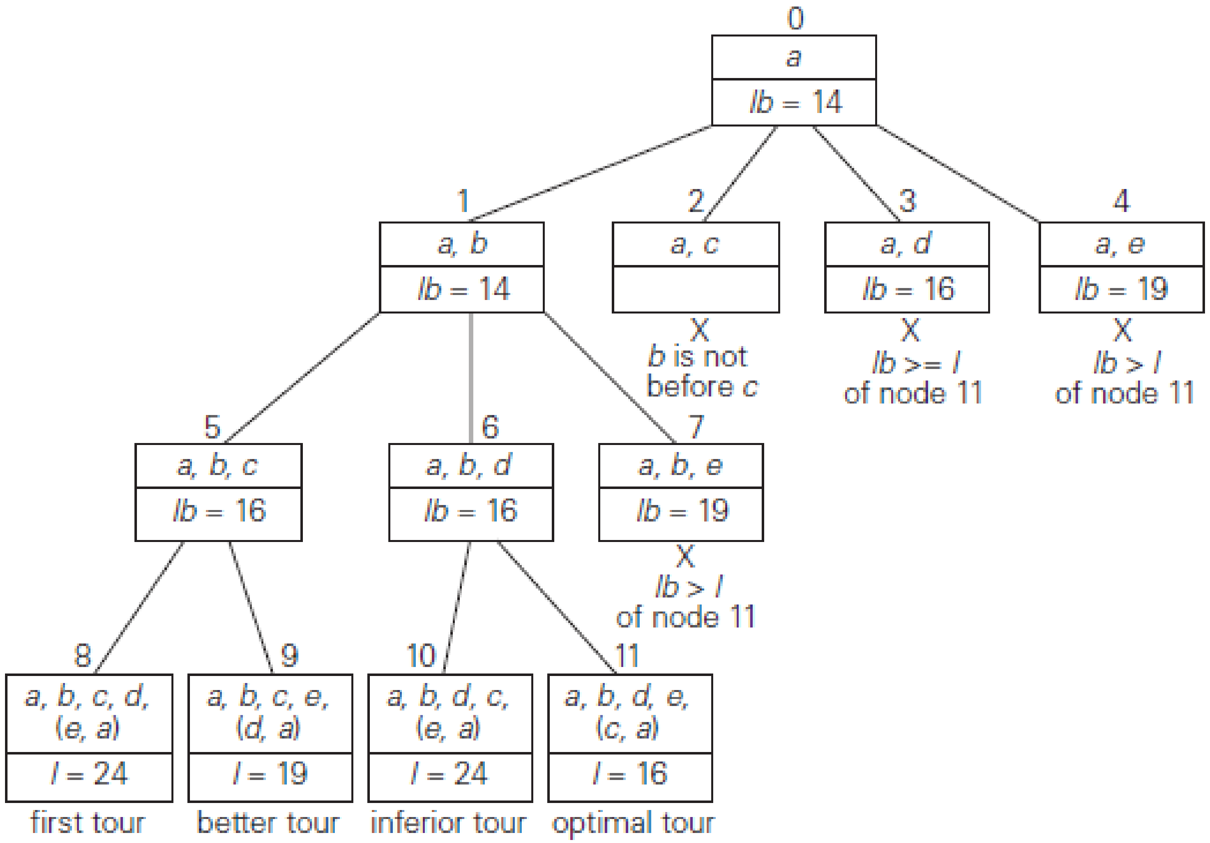


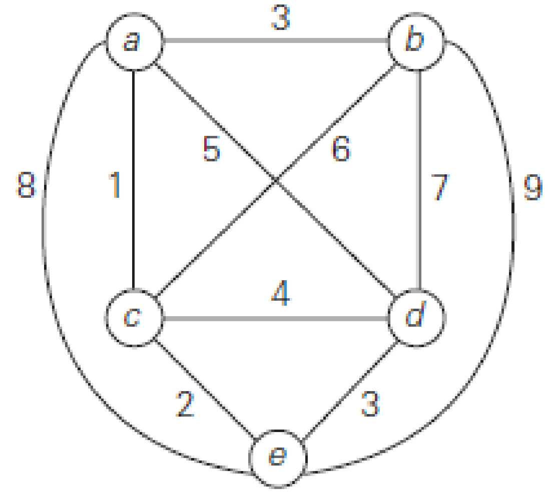
- Τροποιούμε το lb για κάθε υποσύνολο διαδρομών που πρέπει να περιλαμβάνουν μια συγκεκριμένη ακμή
- Παράδειγμα: για όλα τα Hamiltonian circuits που πρέπει να περιλαμβάνουν την ακμή (a,d), παίρνουμε το ακόλουθο lb

$$\lceil [(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2 \rceil = 16$$

Χειρισμός Περιορισμών (32/32)

The traveling salesman problem





Approximation Algorithms for NP-Hard Problems

Approximation Algorithms (1/16)

Η προσέγγιση κάποιων προβλημάτων (π.χ. Traveling salesman) ως προβλήματα απόφασης τα κατατάσσει ως NP-complete προβλήματα ενώ η προσέγγισή τους ως προβλήματα βελτιστοποίησης τα κατατάσσει ως προβλήματα NP-hard

Τα NP-hard προβλήματα είναι δύσκολα τουλάχιστον όπως τα NP-complete προβλήματα

Έτσι, δεν υπάρχει γνωστός αλγόριθμος πολυωνυμικού χρόνου για την επίλυσή τους

Επίσης, μπορεί να υπάρχουν θεωρητικές ενδείξεις ότι τέτοιοι αλγόριθμοι δεν υπάρχουν

Approximation Algorithms (2/16)

Αν ένα στιγμιότυπο αυτών των προβλημάτων είναι αρκετά μικρό μπορούμε να εφαρμόσουμε exhaustive search

Κάποια μπορεί να λυθούν με δυναμικό προγραμματισμό

Η καλή επίδοση της branch-and-bound δεν είναι εγγυημένη

Μια λύση είναι η εφαρμογή προσεγγιστικών αλγορίθμων που είναι αρκετά γρήγοροι

Συνήθως, βασίζονται σε **ευρετικές λύσεις**

Ευρετική είναι η λύση που εξάγεται μέσω της εμπειρίας

Approximation Algorithms (3/16)

Αν υιοθετήσουμε ένα προσεγγιστικό αλγόριθμο το ερώτημα είναι το πόσο κοντά είναι το αποτέλεσμά του με την πραγματική βέλτιστη λύση

Η **ακρίβεια (accuracy)** είναι το βασικό μέγεθος σε αυτές τις περιπτώσεις

Μπορούμε να δούμε την ακρίβεια ως ένα πρόβλημα ελαχιστοποίησης του σφάλματος για μια προσεγγιστική λύση s_a με τη βοήθεια μιας συνάρτησης f (s* είναι οποιαδήποτε λύση του προβλήματος)

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)}$$

Approximation Algorithms (4/16)

Επίσης μπορούμε να υιοθετήσουμε το **λόγο ακρίβειας (accuracy ratio)**

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

Αν πρόκειται για πρόβλημα μεγιστοποίησης τότε υιοθετούμε το λόγο

$$r(s_a) = \frac{f(s^*)}{f(s_a)}$$

Θέλουμε την ακρίβεια σφάλματος κοντά στο 0 ενώ το λόγο ακρίβειας κοντά στο 1

Approximation Algorithms (5/16)

Ορισμός

- **Ένας προσεγγιστικός αλγόριθμος πολυωνυμικού χρόνου λέμε ότι είναι ένας c-προσεγγιστικός αλγόριθμος (c-approximation algorithm), με c>1, αν ο λόγος ακρίβειας που παράγει δεν ξεπερνά το c για οποιοδήποτε στιγμιότυπο του προβλήματος**

$$r(s_a) \leq c$$

Η βέλτιστη τιμή του c καλείται **λόγος επίδοσης (performance ratio)**

Approximation Algorithms (6/16)

Approximations of the Traveling Salesman problem

- Οι απλούστερες προσεγγίσεις βασίζονται στην άπληστη μέθοδο
  - Nearest neighbor algorithm
  - Βήμα 1: επιλογή μιας πόλης ως αφετηρία
  - Βήμα 2: επαναληπτικά μέχρι όλες οι πόλεις να έχουν επισκεφθεί – πηγαίνουμε σε μια πόλη που δεν έχουμε επισκεφθεί που είναι πιο κοντά σε μια πόλη που έχουμε επισκεφθεί τελευταία
  - Βήμα 3: επιστρέφουμε στην αφετηρία

Approximation Algorithms (7/16)

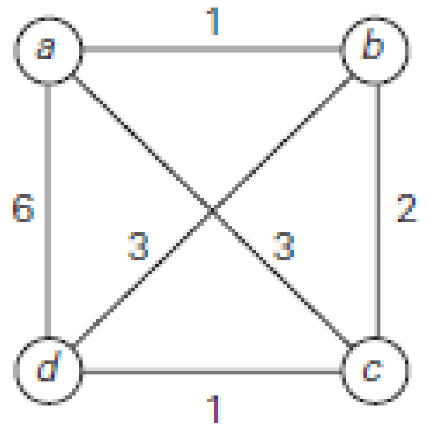
Approximations of the Traveling Salesman problem

- Οι απλούστερες προσεγγίσεις βασίζονται στην άπληστη μέθοδο
  - Nearest neighbor algorithm
  - Παράδειγμα

    s_a: $a - b - c - d - a$ of length 10.
  - Βέλτιστη λύση με exhaustive search

    s^*: $a - b - d - c - a$ of length 8

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$



Approximation Algorithms (8/16)

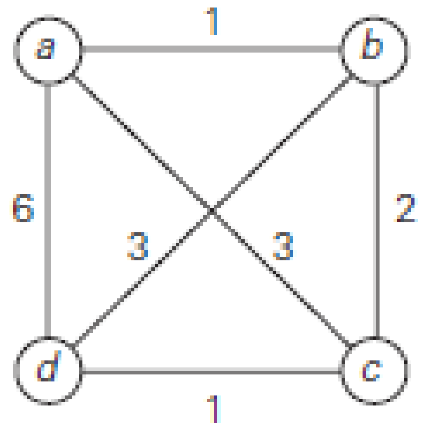
Approximations of the Traveling Salesman problem

- Οι απλούστερες προσεγγίσεις βασίζονται στην άπληστη μέθοδο
  - Multifragment-heuristic algorithm
  - Βήμα 1: ταξινόμηση των ακμών σε αύξουσα σειρά βαρών - Αρχικοποίηση του συνόλου των ακμών της διαδρομής ως το κενό σύνολο
  - Βήμα 2: επαναληπτικά n φορές – προσθήκη της επόμενης ακμής από την ταξινομημένη λίστα στο σύνολο των ακμών της διαδρομής δεδομένου ότι δεν δημιουργούνται κύκλοι; Διαφορετικά απορρίπτουμε την ακμή
  - Βήμα 3: επιστρέφουμε το σύνολο των ακμών

Approximation Algorithms (9/16)

Approximations of the Traveling Salesman problem

- Οι απλούστερες προσεγγίσεις βασίζονται στην άπληστη μέθοδο
  - Multifragment-heuristic algorithm
  - Παράδειγμα
  - Λύση: (a,b), (c,d), (b,c), (a,d)



Approximation Algorithms (10/16)

Approximations of the Traveling Salesman problem

- Minimum spanning tree algorithms
  - Αν διαγράψουμε μια ακμή από ένα Hamiltonian circuit προκύπτει ένα MST
  - Άρα, ένα MST αποτελεί μια καλή βάση για την εύρεση μιας προσέγγισης της τελικής διαδρομής
  - Twice-around-the-tree algorithm
    - Βήμα 1: δημιουργία ενός MST του γράφου
    - Βήμα 2: ξεκινώντας από μια ακμή, εκτελούμε μια διαδρομή στο MST καταγράφοντας τους κόμβους (π.χ. με DFS)
    - Βήμα 3: βλέποντας τους κόμβους της λίστας που προκύπτει από το προηγούμενο βήμα, εξαλείφουμε όλες τις επαναλαμβανόμενες εμφανίσεις κόμβων εκτός από τον πρώτο και τον τελευταίο

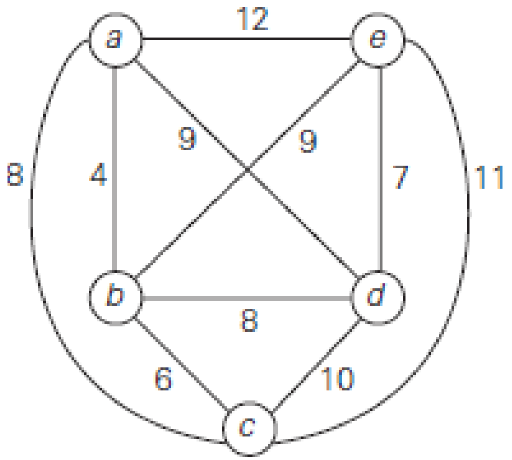
Approximation Algorithms (11/16)

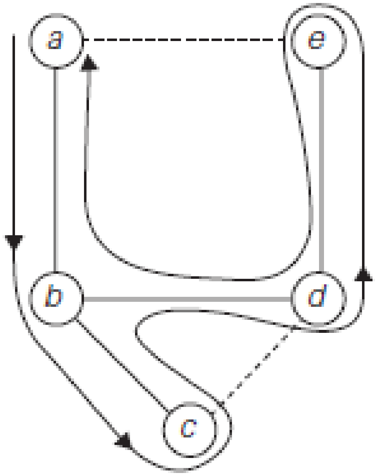
Approximations of the Traveling Salesman problem

- Minimum spanning tree algorithms
  - Twice-around-the-tree algorithm
    - Παράδειγμα
    - Αρχική λίστα κόμβων

      $a, \; b, \; c, \; b, \; d, \; e, \; d, \; b, \; a$
    - Τελική λίστα (μήκος 39)

      $a, \; b, \; c, \; d, \; e, \; a$





Approximation Algorithms (12/16)

Approximations of the Knapsack problem

- Άπληστη μέθοδος
  - Greedy algorithm for the discrete knapsack problem
    - Βήμα 1: υπολογίζουμε το λόγο αξία προς βάρος
    - Βήμα 2: ταξινομούμε τα αντικείμενα σε μη αύξουσα σειρά ως προς το λόγο που υπολογίσαμε
    - Βήμα 3: επαναληπτικά μέχρι να μην υπάρχουν στοιχεία μέσα στη λίστα – αν το επόμενο αντικείμενο χωράει στο σακίδιο, το τοποθετούμε και προχωράμε στο επόμενο; Διαφορετικά, απλά προχωράμε στο επόμενο αντικείμενο

Approximation Algorithms (13/16)

Approximations of the Knapsack problem

- Άπληστη μέθοδος
  - Greedy algorithm for the discrete knapsack problem
    - Παράδειγμα:
    - W=10
    - Επιλέγουμε τα αντικείμενα 1, 3

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

Approximation Algorithms (14/16)

Approximations of the Knapsack problem

- Άπληστη μέθοδος
  - Greedy algorithm for the discrete knapsack problem
    - Παράδειγμα μη εύρεσης της βέλτιστης λύσης:

item	weight	value	value/weight
1	1	2	2
2	W	W	1

The knapsack capacity is $W > 2$.

Approximation Algorithms (15/16)

Approximations of the Knapsack problem

- Άπληστη μέθοδος
  - Greedy algorithm for the continuous knapsack problem
    - Βήμα 1: υπολογίζουμε το λόγο αξία προς βάρος
    - Βήμα 2: ταξινομούμε τα αντικείμενα σε μη αύξουσα σειρά ως προς το λόγο που υπολογίσαμε
    - Βήμα 3: επαναληπτικά μέχρι να μην υπάρχουν στοιχεία μέσα στη λίστα – αν το επόμενο αντικείμενο χωράει στο σακίδιο ολόκληρο, το τοποθετούμε και προχωράμε στο επόμενο; Διαφορετικά, παίρνουμε το ποσοστό του αντικειμένου που χωράει στο σακίδιο και σταματάμε

Approximation Algorithms (16/16)

Approximations of the Knapsack problem

- Άπληστη μέθοδος
  - Greedy algorithm for the discrete knapsack problem
    - Παράδειγμα:
    - W=10
    - Επιλέγουμε τα αντικείμενα 1, 2 (από το δεύτερο παίρνουμε τα 6/7)

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4