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

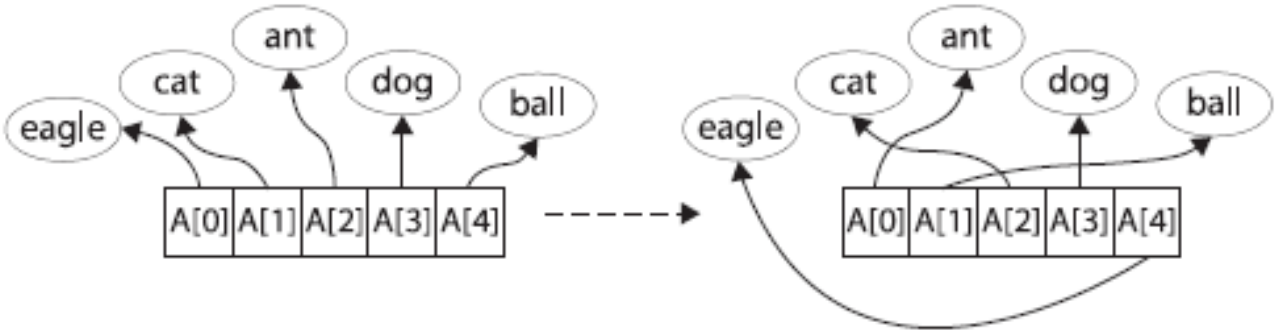
Εισαγωγή (2/3)

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

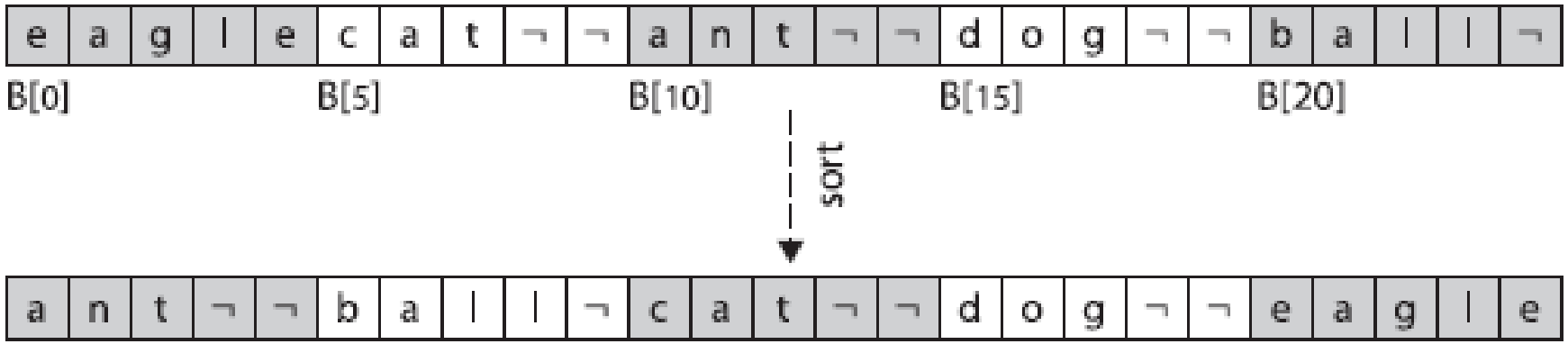
- Αποθηκεύουμε τους δείκτες προς την πληροφορία



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**

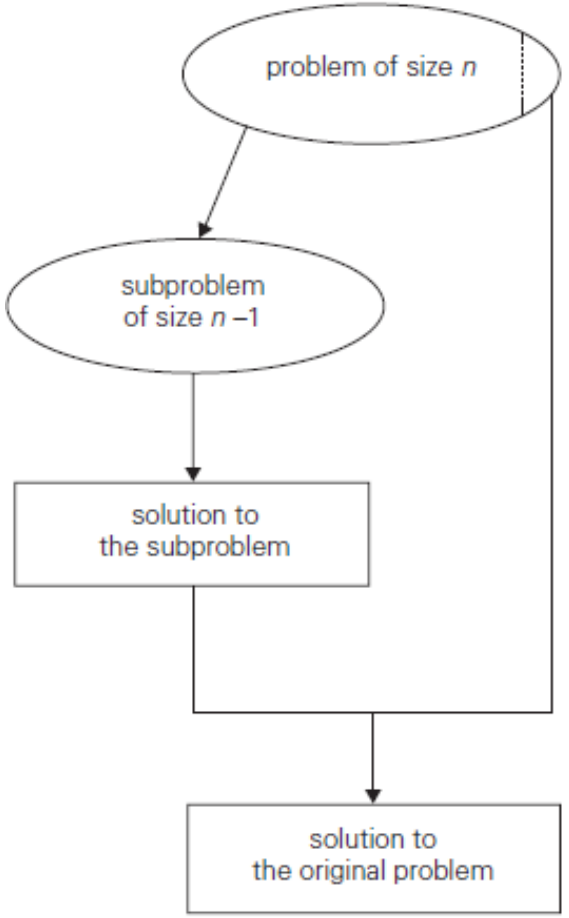
- Αποτιμά τη σχέση μιας λύσης με τη λύση του προβλήματος σε μικρότερο μέγεθος του προβλήματος
- Αν η σχέση οριστεί, τότε η λύση έρχεται είτε ακολουθώντας μια bottom-up ή μια top-down προσέγγιση
- Η top-down οδηγεί σε μια αναδρομική σχέση (υπάρχουν περιπτώσεις που λύνονται με μη αναδρομικές σχέσεις)
- 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}
  - $\alpha^{\nu} = \alpha^{\nu-1}\,\alpha$
  - Οπότε

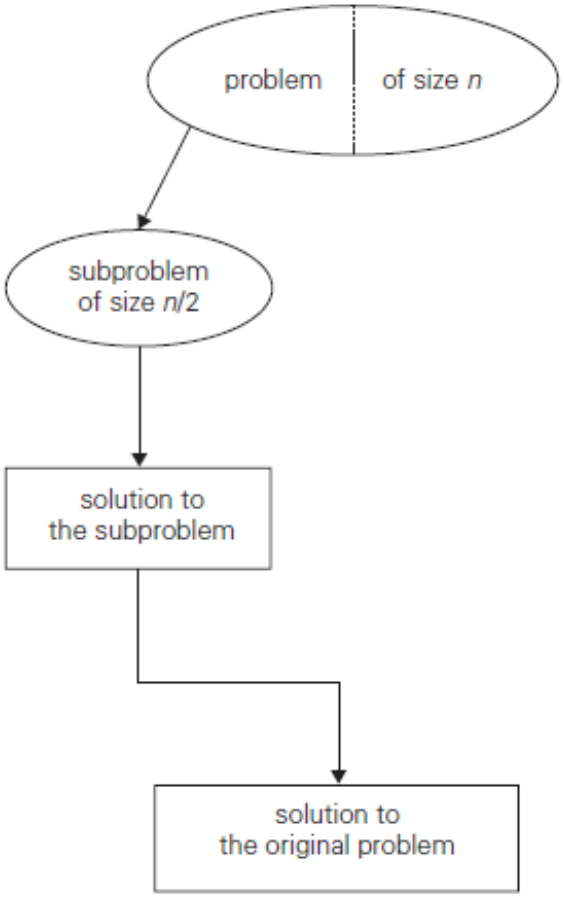
$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$



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

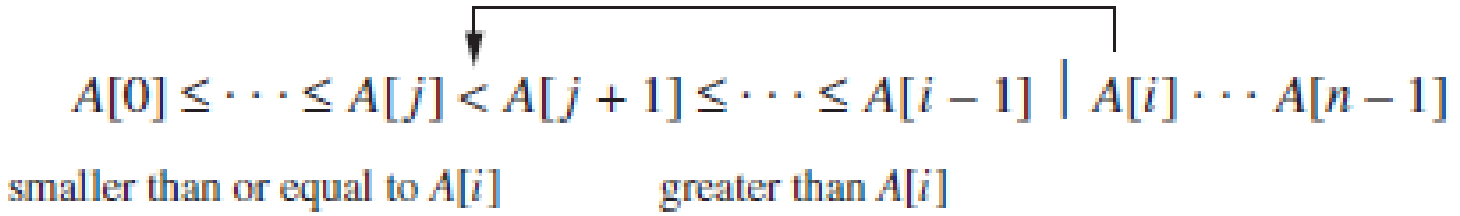
Το στοιχείο A[i] μπαίνει στην κατάλληλη θέση

Η προσπέλαση στα στοιχεία γίνεται ξεκινώντας από αριστερά προς τα δεξιά

Insertion Sort (2/9)

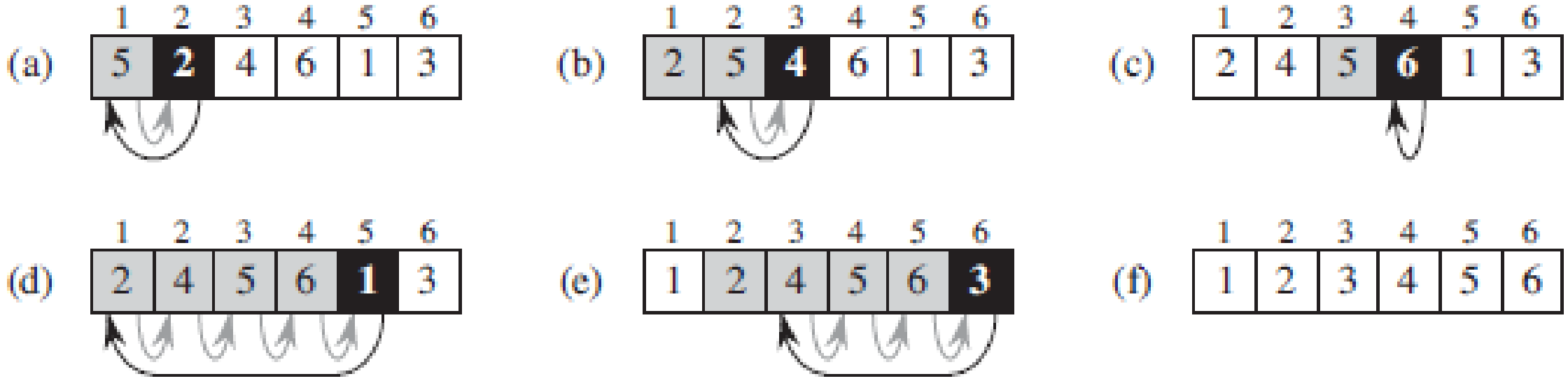
ALGORITHM *InsertionSort*$(A[0..n-1])$

//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**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$



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*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Insertion Sort (8/9)

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

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

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

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

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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right)$$

$$+ c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n$$

$$- (c_2 + c_4 + c_5 + c_8)$$

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

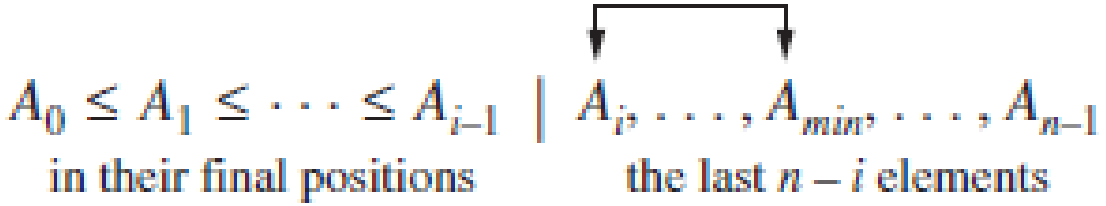
Έπειτα προχωρά στη δεύτερη, την τρίτη, κ.ο.κ.

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



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 $1^{ο}$ πέρασμα, το $1^{ο}$ στοιχείο μπαίνει στη θέση του, στο $2^{ο}$ πέρασμα, το $2^{ο}$ 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**

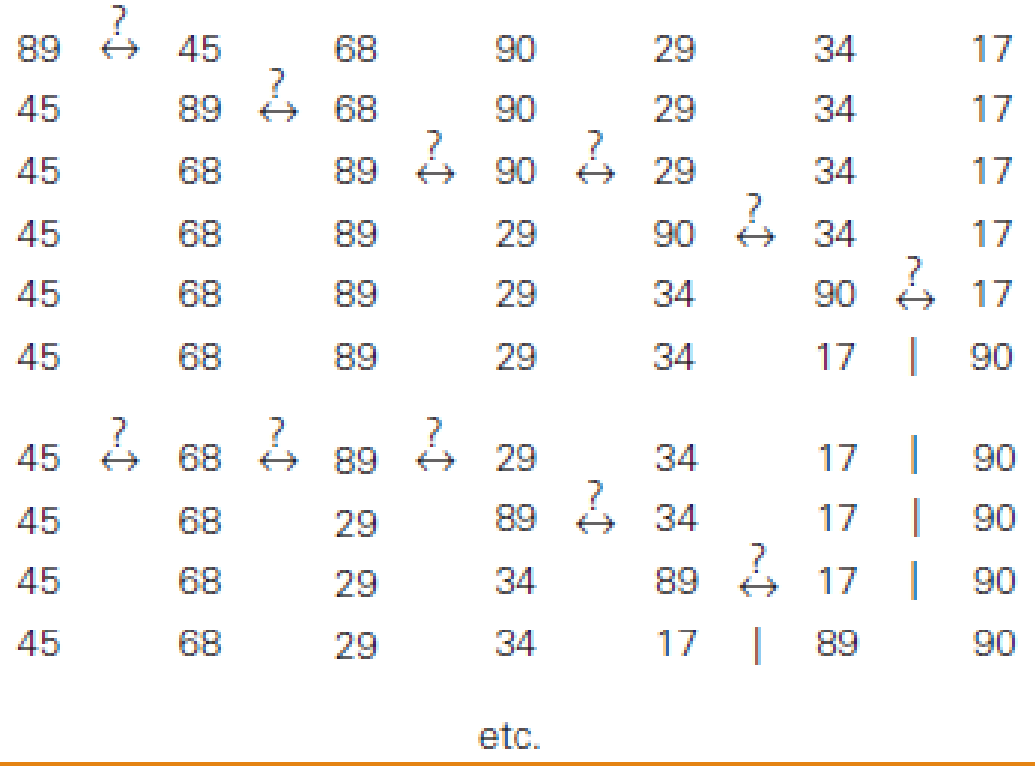
  for $j \leftarrow 0$ **to** $n-2-i$ **do**

    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



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

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΠΡΟΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ
ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ

ΑΛΓΟΡΙΘΜΟΙ

ΔΙΑΛΕΞΗ 6

ΔΙΔΑΣΚΩΝ
ΚΩΣΤΑΣ ΚΟΛΟΜΒΑΤΣΟΣ

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)

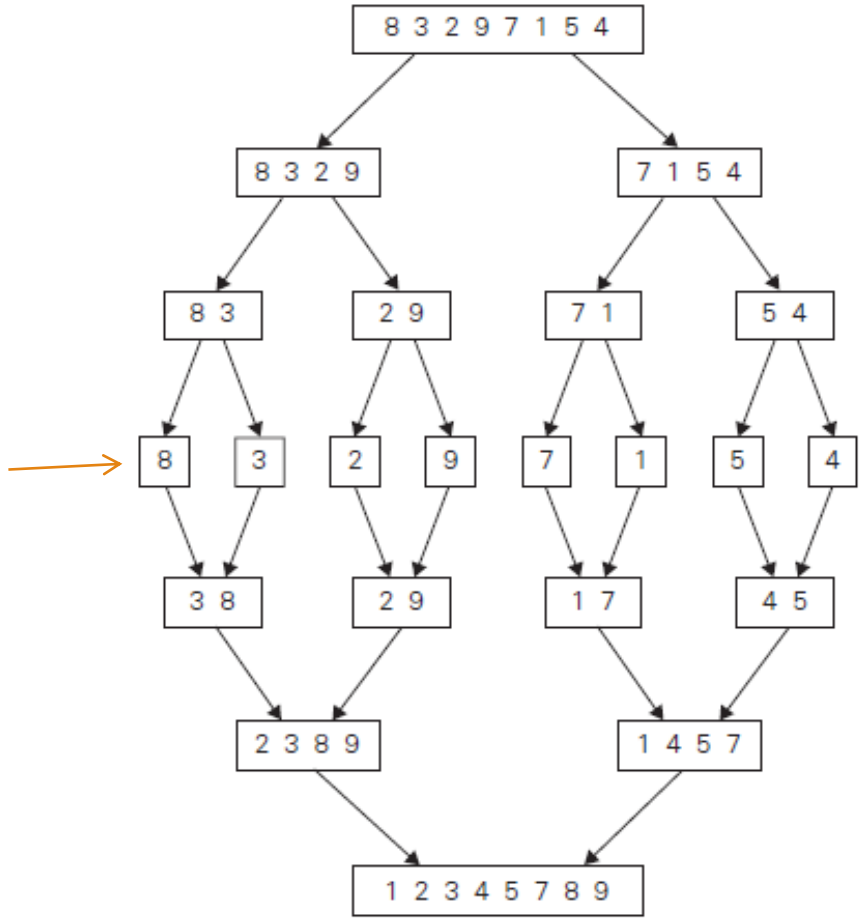
Αλγόριθμος συγχώνευσης

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)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      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

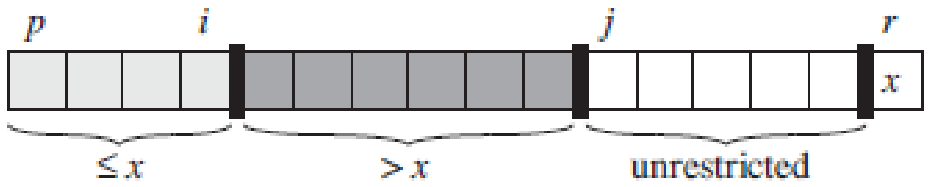
Ο αλγόριθμος διαχωρισμού έχει ως εξής:

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

Quick Sort (5/30)

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

Ο πίνακας χωρίζεται σε 4 περιοχές (πιθανώς άδειες)




```
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
```

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

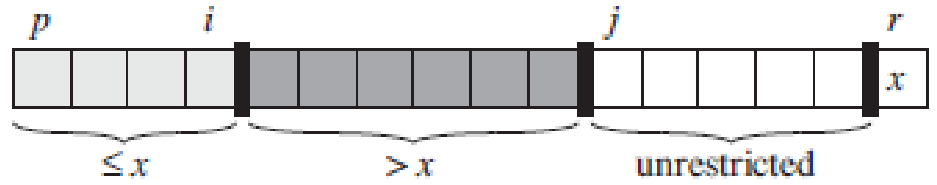
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

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 1η 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

```
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
```



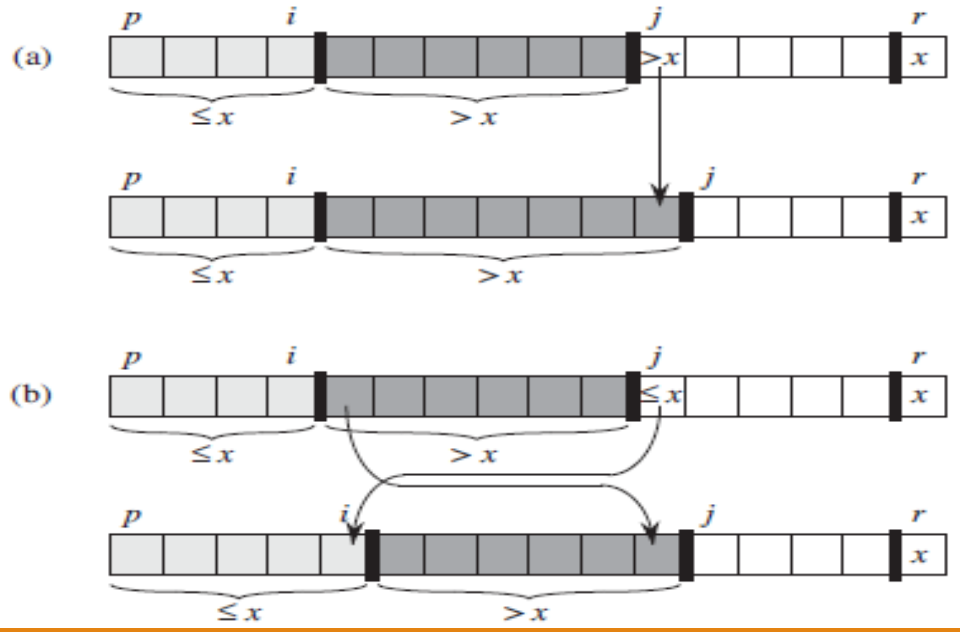
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

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 1η 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)

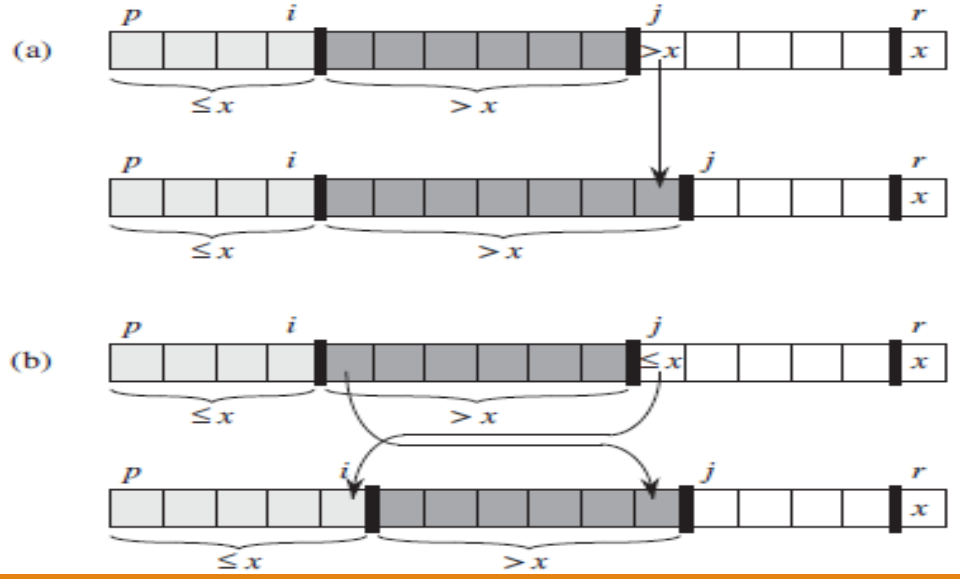
Ο τερματισμός του αλγορίθμου

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

```
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
```



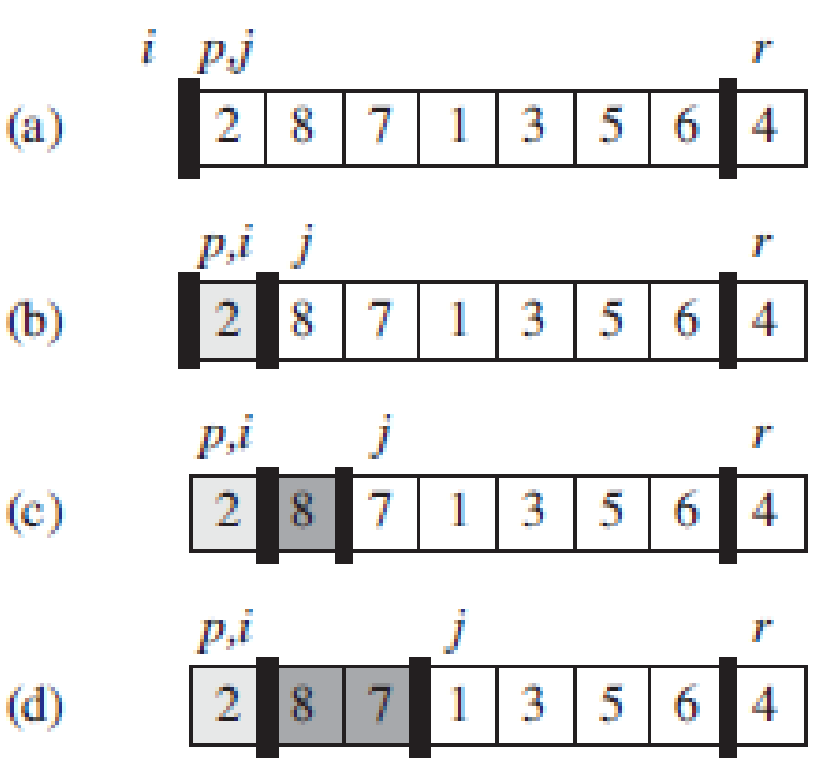
Quick Sort (9/30)

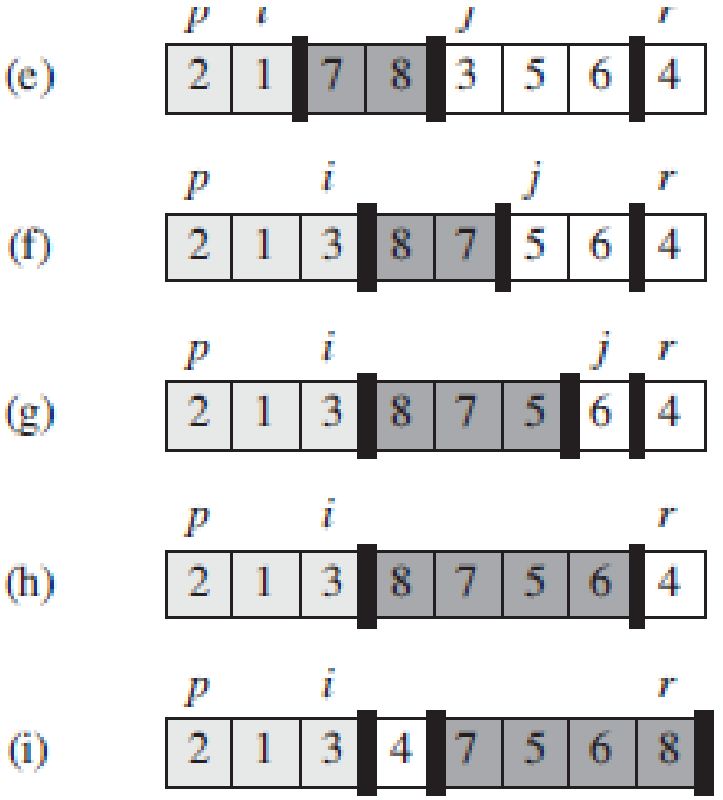
Ο χρόνος εκτέλεσης του τμήματος του διαχωρισμού έχει πολυπλοκότητα Θ(n)

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

Quick Sort (10/30)

Παράδειγμα εκτέλεσης






```
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 (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 T(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)

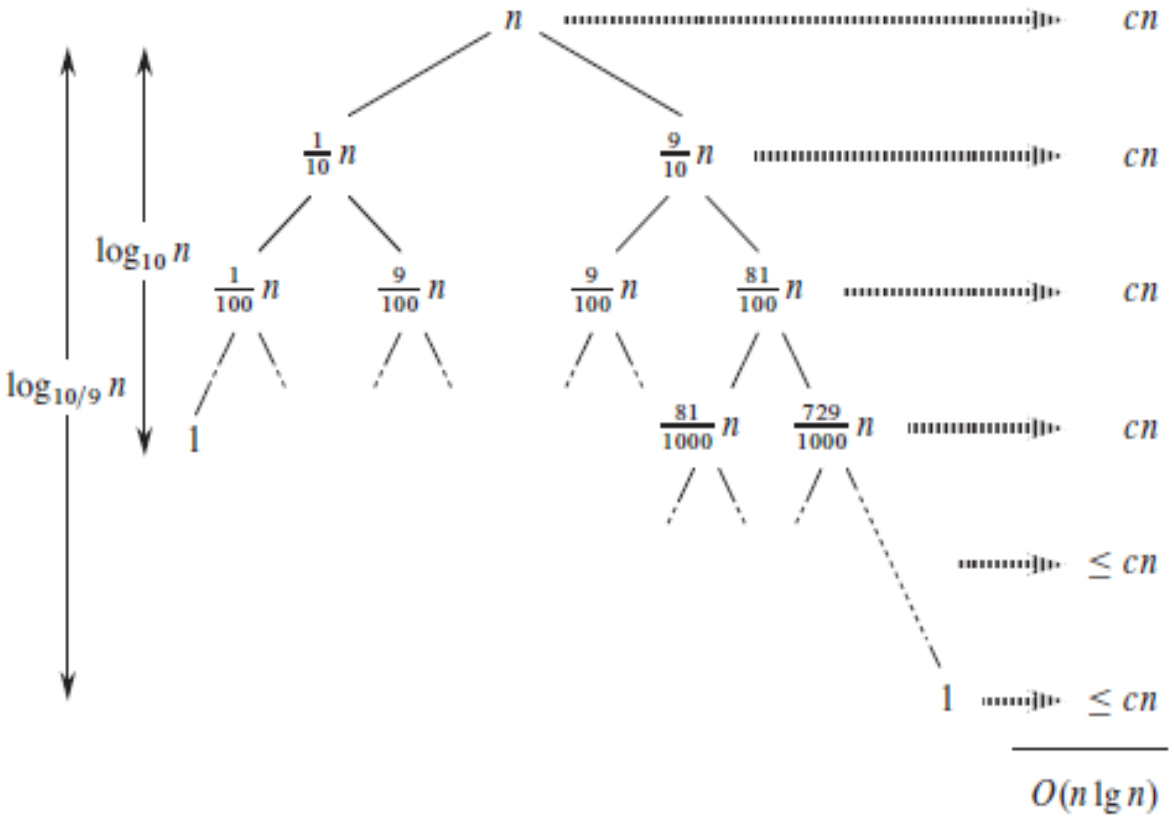
Η μέση περίπτωση είναι πιο κοντά στην καλύτερη περίπτωση παρά στη χειρότερη

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 $\log_{10}n=\Theta(\log n)$

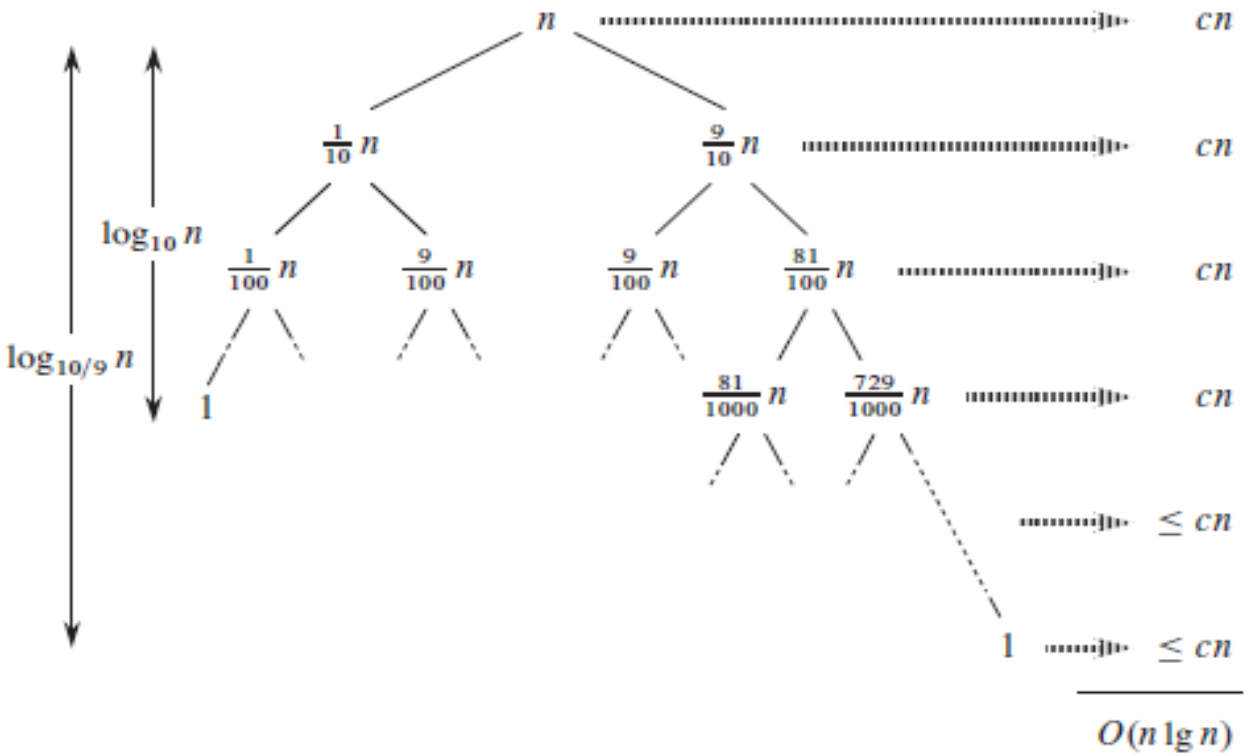


Quick Sort (16/30)

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

Συνολικό κόστος: O(n logn)

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



Quick Sort (17/30)

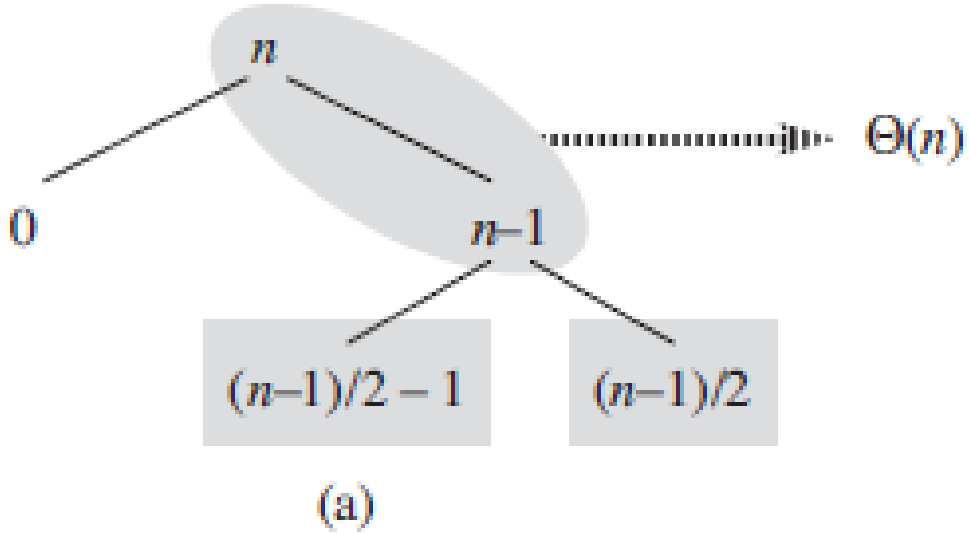
Ο αλγόριθμος εξαρτάται από την κατανομή των στοιχείων στον πίνακα και όχι από το ποια στοιχεία είναι αυτά

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

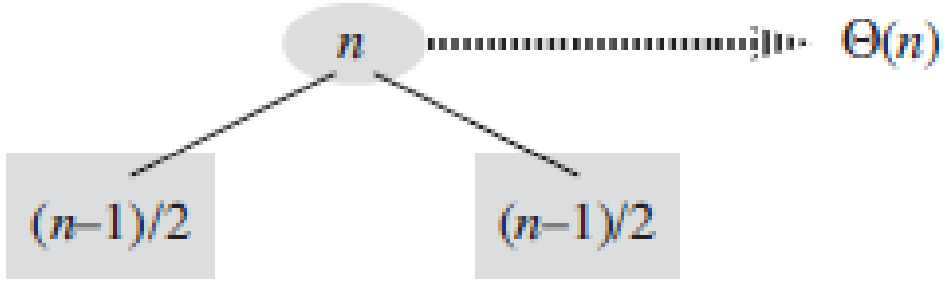
Στη μέση περίπτωση, η κατανομή των ‘καλών’ και ‘κακών’ διαχωρισμών θα είναι τυχαία ‘τοποθετημένες’ στο δένδρο διαχωρισμού

Quick Sort (18/30)

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



(a)



(b)

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

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

Quick Sort (21/30)

Γενική Ανάλυση του αλγορίθμου:

- Χειρότερη περίπτωση
  - Αναδρομική σχέση

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

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

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

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:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

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:

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

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)

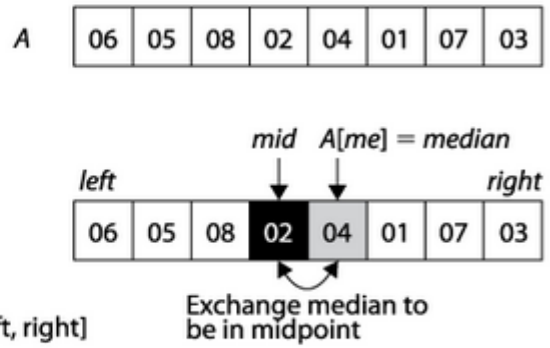
Ο αλγόριθμος ανταλλάσσει τα στοιχεία που είναι στο αριστερό μέρος και είναι μεγαλύτερα από το στοιχείο στη μέση με στοιχεία που είναι στη δεξιά πλευρά και είναι μικρότερα από το στοιχείο στη μέση

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

Median Sort (2/9)

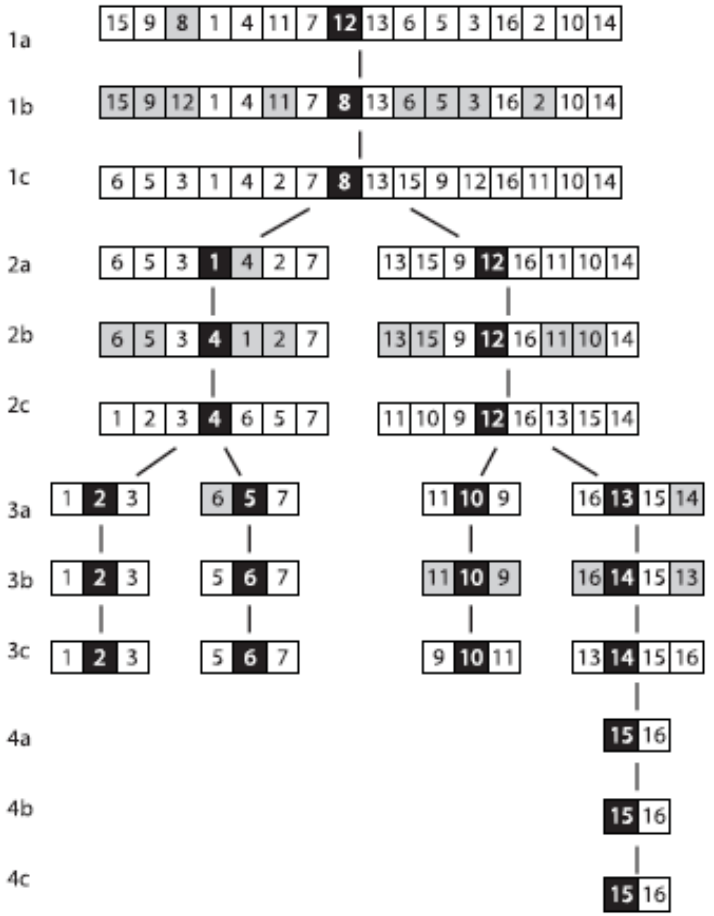
```
sort (A)
1.    medianSort (A, 0, n − 1)
end

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 $1^{ης}$ 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 logn)
- 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)
 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 (3/6)

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

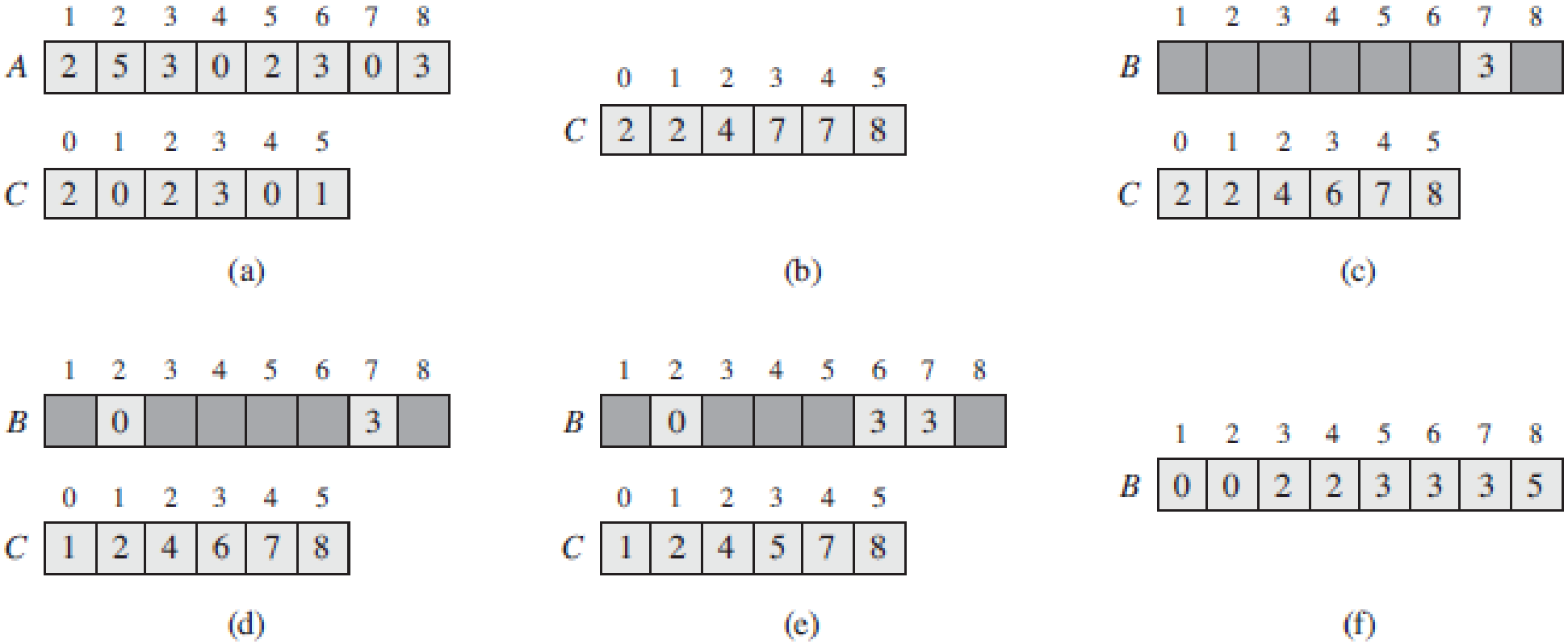
Παράδειγμα: το C[i] έχει το πλήθος των στοιχείων που είναι ίσα με i

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

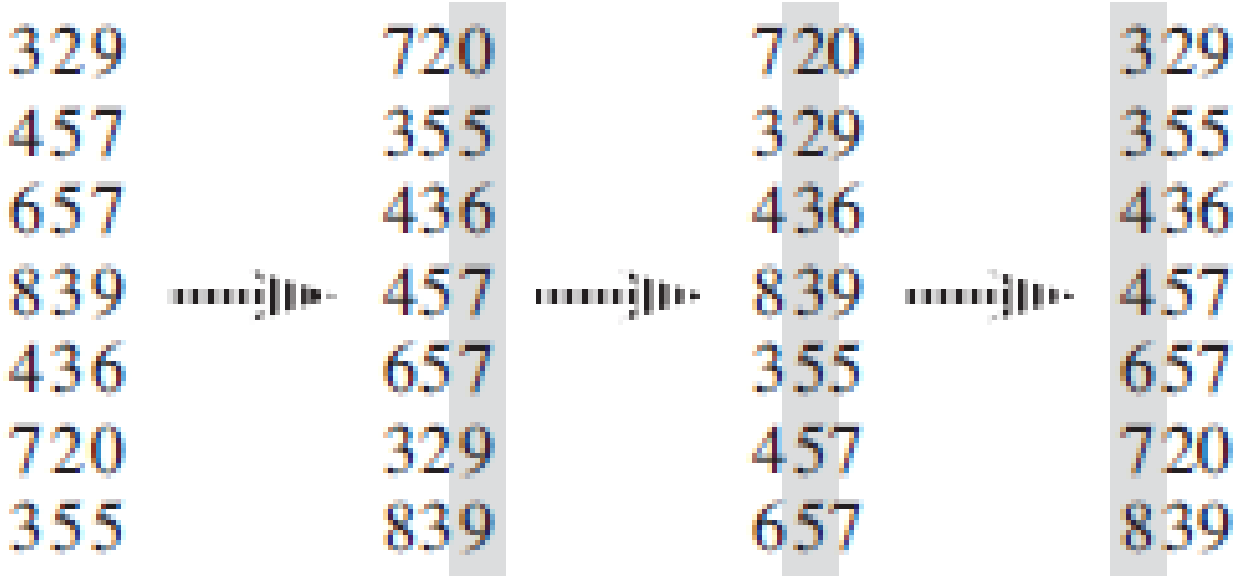
Χρησιμοποιεί ένα σύνολο κάδων (bins)

Για αριθμούς στο δεκαδικό σύστημα κάθε στήλη θα έχει 10 θέσεις

Απαιτούνται d περάσματα για την ταξινόμηση

Radix Sort (2/7)

Παράδειγμα εκτέλεσης



Radix Sort (3/7)

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      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

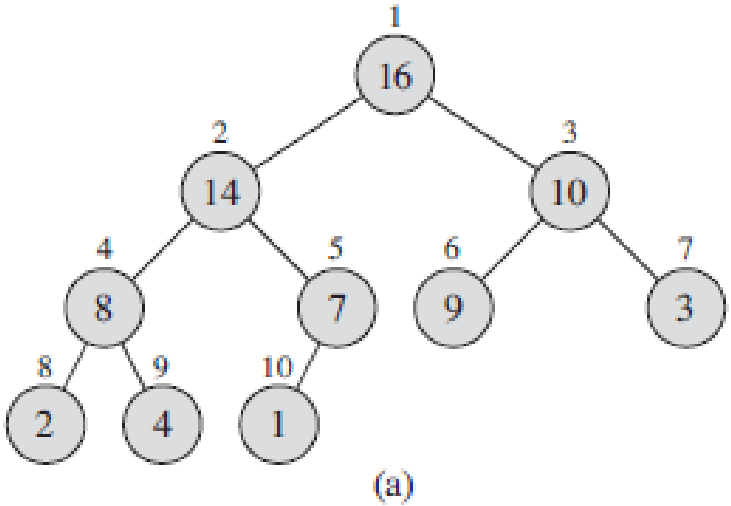
Heap Sort (2/15)

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

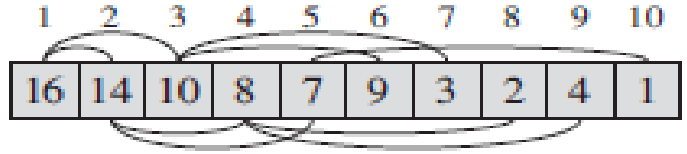
Κάθε κόμβος του δένδρου αντιστοιχεί σε ένα στοιχείο του πίνακα

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)



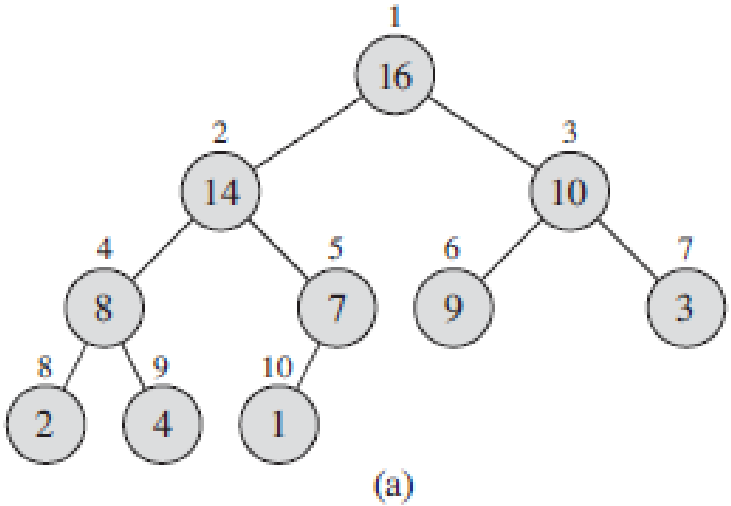
(b)

Heap Sort (3/15)

Μόνο τα A.heap_size είναι στοιχεία του σωρού

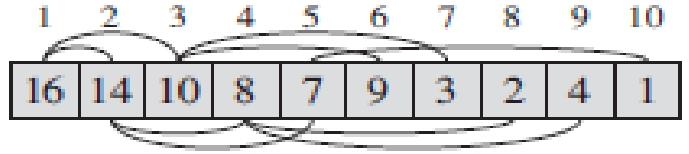
Η ρίζα του δένδρου είναι το A[1]

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



(a)

```
PARENT(i)          LEFT(i)          RIGHT(i)
1  return ⌊i/2⌋    1  return 2i     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** $A[\text{PARENT}(i)] \geq A[i]$
  - Το μεγαλύτερο στοιχείο βρίσκεται στη ρίζα
- **Min-heap** $A[\text{PARENT}(i)] \leq A[i]$
  - 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)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      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

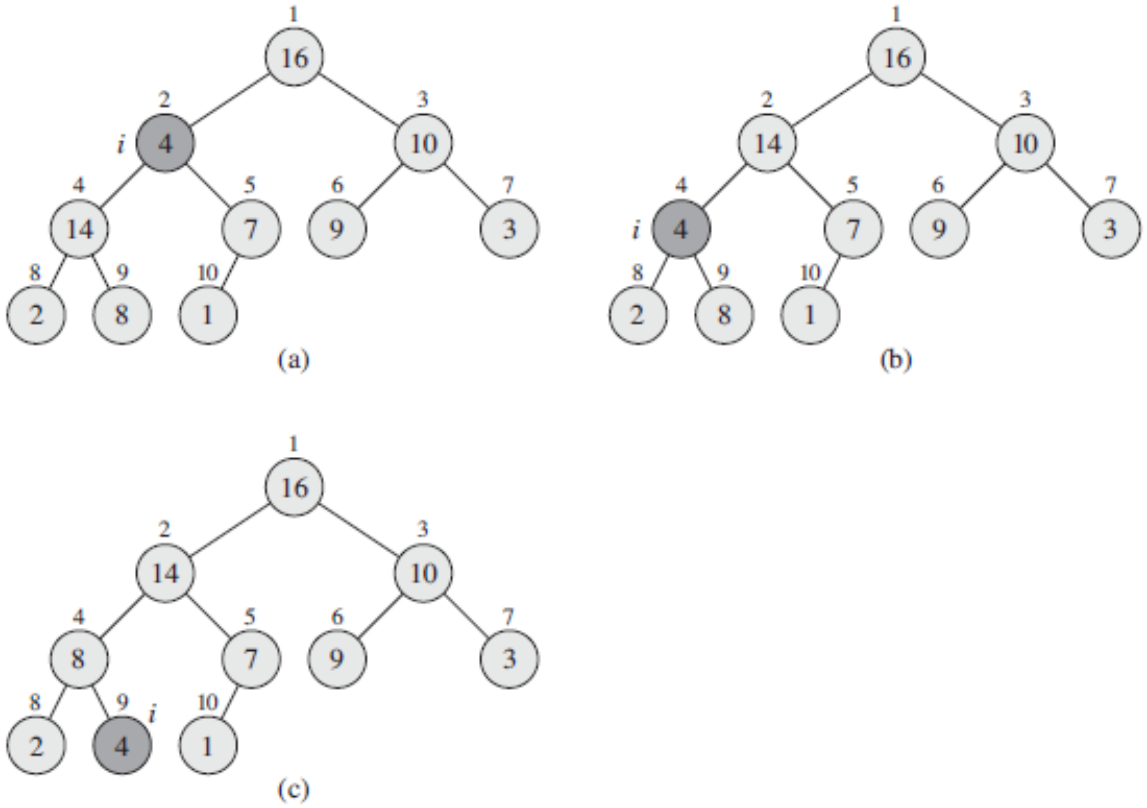
Διαφορετικά, ένα από τα παιδιά έχει το μεγαλύτερο στοιχείο, οπότε το A[i] αντιμετατίθεται με το A[largest]

Έπειτα καλούμε αναδρομικά τον αλγόριθμο

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      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) <= T(2n/3) + Θ(1)**
- Με βάση το master θεώρημα – περίπτωση 2 – έχουμε ότι **T(n) = O(logn)** – 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)

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

$$x = 1/2$$

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

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

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]

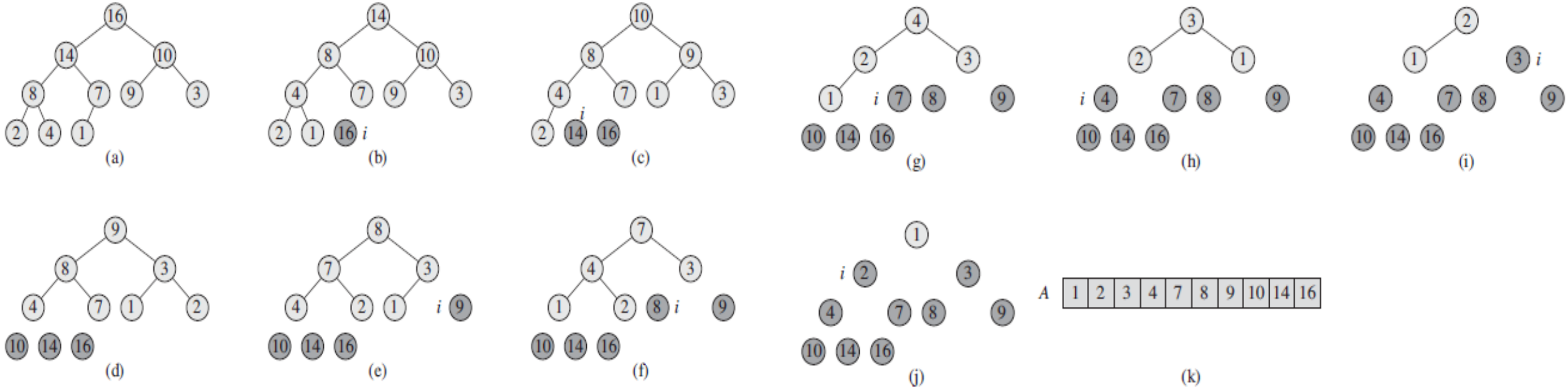
Στη συνέχεια μειώνουμε το μέγεθος του σωρού (δεν λαμβάνουμε υπόψιν το τελευταίο στοιχείο)

Επαναφέρουμε την ιδιότητα max heap καλώντας τον MAX-HEAPIFY

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      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)

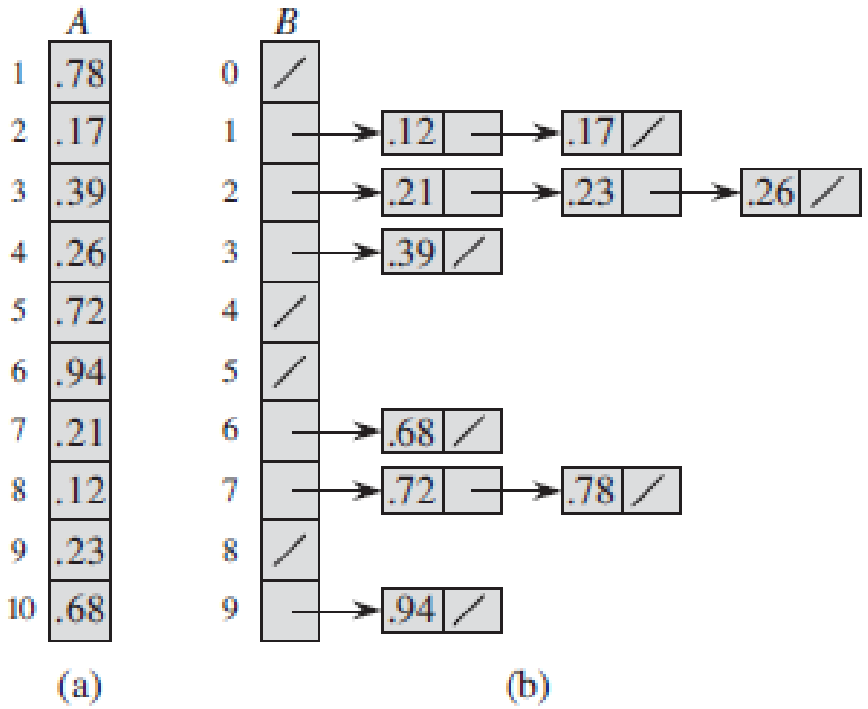
Bucket Sort (2/9)

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  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

$$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$$

$$\text{for } i = 0, 1, \ldots, n-1 \text{ and } j = 1, 2, \ldots, n$$

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Bucket Sort (7/9)

Συνεπώς

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} \mathrm{E}\left[X_{ij}X_{ik}\right]
\end{aligned}
$$

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: $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΠΡΟΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ
ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ

ΑΛΓΟΡΙΘΜΟΙ

ΔΙΑΛΕΞΗ 7

ΔΙΔΑΣΚΩΝ
ΚΩΣΤΑΣ ΚΟΛΟΜΒΑΤΣΟΣ

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

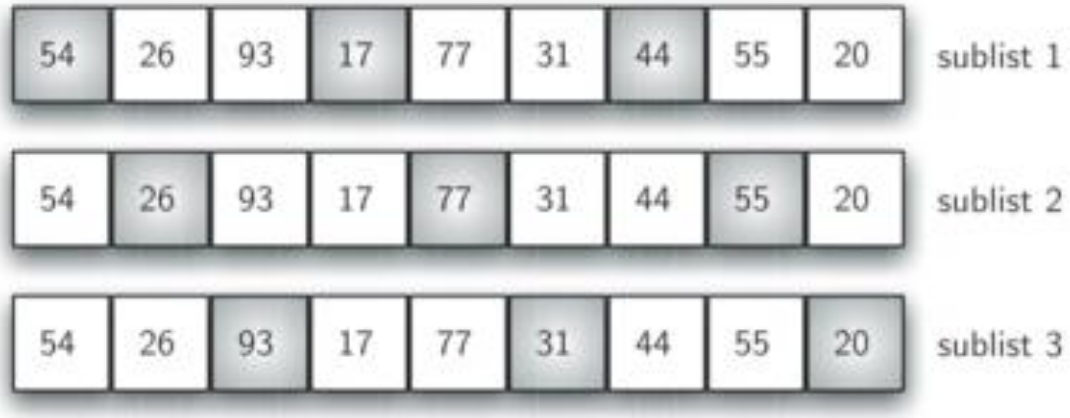
Κάθε μια υπο-λίστα ταξινομείται με τον insertion sort

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

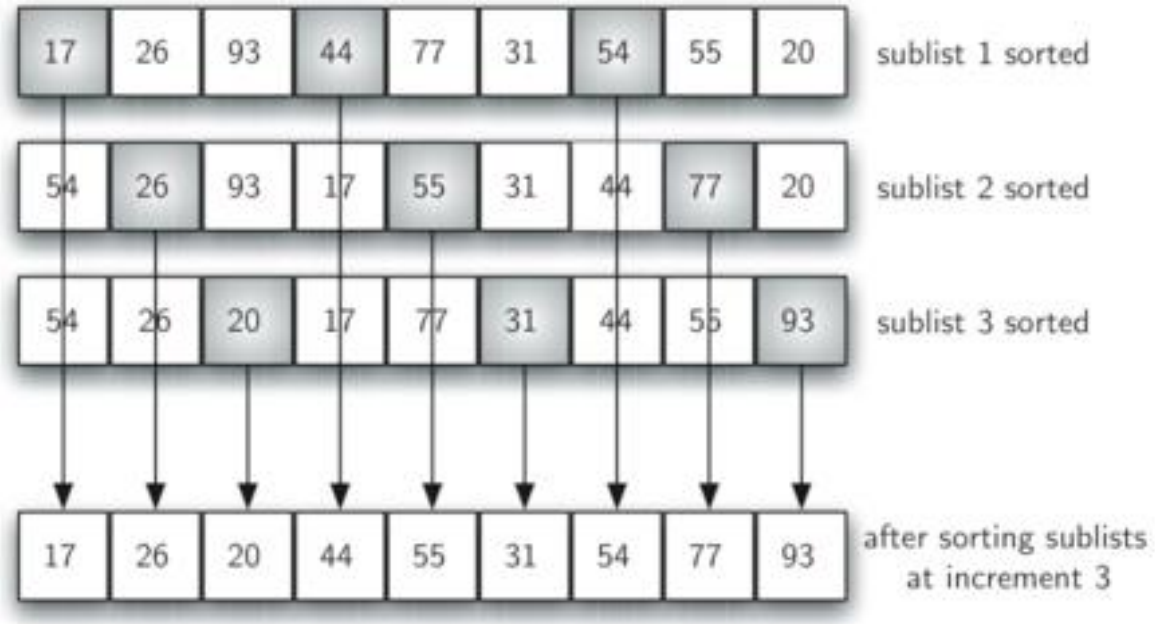
Παράδειγμα:

- i=3



Shell Sort (3/10)

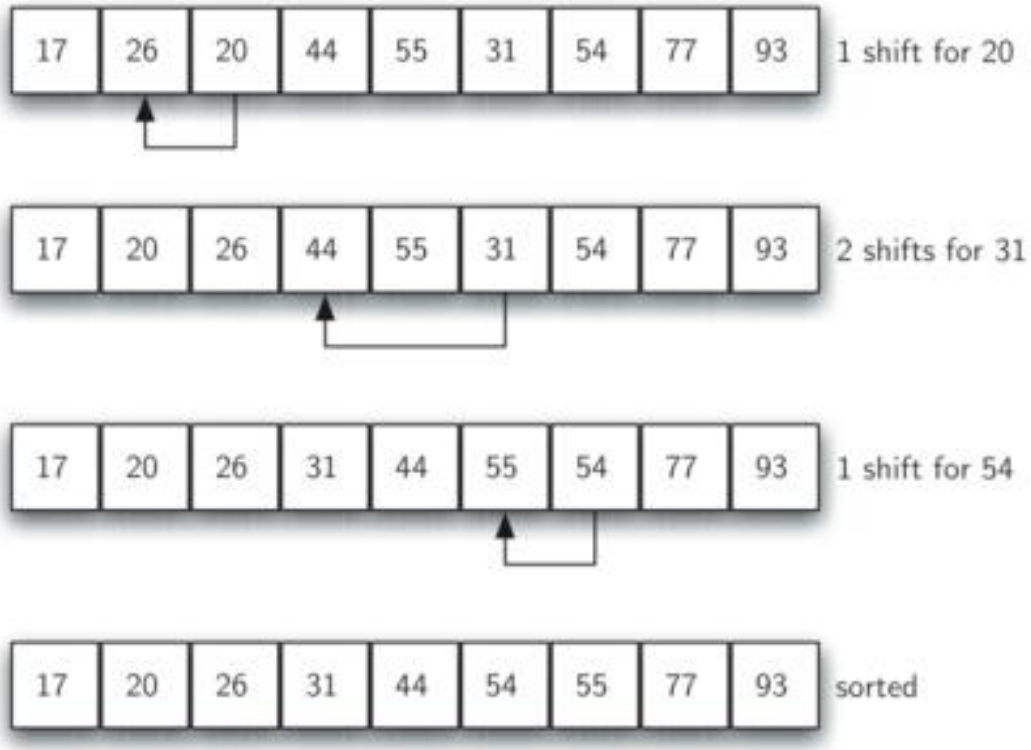
Μετά από την ταξινόμηση με τον insertion sort η λίστες γίνονται



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
$\prod_{\substack{0 \leq q < r \\ q \neq (r^2+r)/2-k}} a_q$, where $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$, $a_q = \min\{n \in \mathbb{N} : n \geq (5/2)^{q+1}, \forall p : 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\}$	$1, 3, 7, 21, 48, 112, \ldots$	$O(N^{1+\sqrt{8 \ln(5/2)/\ln N}})$	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