Ενότητα 8
Ταξινόμηση

Ταξινόμηση

Θεωρούμε έναν πίνακα $A[0..n-1]$ με n στοιχεία στα οποία έχει ορισθεί μια γραμμική διάταξη, δηλαδή $\forall$ ζεύγος στοιχείων x,y του A, είτε $x < y$, ή $x > y$ ή $x = y$.

Η **διαδικασία ταξινόμησης** του A συνίσταται στην αναδιάταξη των στοιχείων του, έτσι ώστε μετά το πέρας της διαδικασίας αυτής να ισχύει $A[0] \leq A[1] \leq \ldots \leq A[n-1]$.

Ταξινόμηση με Χρήση Ουρών Προτεραιότητας
Δεδομένου ότι μια βιβλιοθήκη παρέχει ουρές προτεραιότητας, ζητείται αλγόριθμος που να ταξινομεί τα n στοιχεία του πίνακα A.

Αλγόριθμος Ταξινόμησης με Χρήση Ουράς Προτεραιότητας

```
MakeEmptySet(S);
for (j = 0; j < n; j++)
      εισαγωγή του στοιχείου A[j] στην ουρά προτεραιότητας S;
for (j = 0; j < n; j++)
          Print(DeleteMin(S));
```

Ποια είναι η χρονική πολυπλοκότητα του αλγορίθμου αυτού?

- Η πολυπλοκότητα εξαρτάται από την πολυπλοκότητα των λειτουργιών Insert() και DeleteMin() της ουράς προτεραιότητας.
- Αν η ουρά προτεραιότητας υλοποιείται με τη χρήση ενός σωρού, τότε η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(n \log n)$.

Ταξινόμηση χρησιμοποιώντας άλλες Δομές Δεδομένων

Αλγόριθμος Ταξινόμησης με Χρήση Ταξινομημένων Δένδρων

```
MakeEmptyBinarySearchTree(T);
for (j = 0; j < n; j++)
    BinarySearchTreeInsert(T, A[j]);
Inorder(T) // με τη Visit() να εκτελεί μια εκτύπωση του κλειδιού του εκάστοτε κόμβου
```

Χρονική Πολυπλοκότητα

- Η χρονική πολυπλοκότητα του παραπάνω αλγορίθμου είναι $O(nh)$, όπου h το ύψος του ταξινομημένου δένδρου μετά από τις n εισαγωγές.
- Αν το ταξινομημένο δένδρο είναι AVL, (2-3) δένδρο ή κοκκινόμαυρο δένδρο τότε η χρονική πολυπλοκότητα του παραπάνω αλγορίθμου είναι $O(n\log n)$ (αφού $h = O(\log n)$).

Χωρική Πολυπλκότητα

☹ Τα n στοιχεία που είναι αποθηκευμένα στον πίνακα A απαιτείται να αποθηκευτούν εκ νέου σε μια νέα δομή.

☹ Το ίδιο ισχύει και στην περίπτωση χρήσης ουράς προτεραιότητας.

☹ Αυτό προκαλεί σπατάλη μνήμης!

Ταξινόμηση - Ο Αλγόριθμος InsertionSort

Πρόβλημα

Είσοδος

Μια ακολουθία από n αριθμούς $\langle a_1, a_2, \ldots, a_n \rangle$.

Έξοδος (output):

Μια μετάθεση (αναδιάταξη) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ της ακολουθίας εισόδου έτσι ώστε:

$$a'_1 \le a'_2 \le \ldots \le a'_n$$

Πως λειτουργεί ο αλγόριθμος αν A = <5,2,4,6,1,3>;

Ο αλγόριθμος InsertionSort() και η πολυπλοκότητά του έχουν συζητηθεί αναλυτικά στην Ενότητα 1 του μαθήματος.

```
Algorithm InsertionSort (A[1..n]) {
// Είσοδος: ένας μη-ταξινομημένος
   πίνακας A ακεραίων αριθμών
// Έξοδος: ο πίνακας A με τα στοιχεία
   του σε αύξουσα διάταξη
   int key, i, j;
   for (j = 2; j ≤ n; j++) {
        key = A[j];
        i = j-1;
        while (i > 0 && A[i] > key) {
             A[i+1] = A[i];
             i = i-1;
        }
        A[i+1] = key;
   }
   return A;
}
```

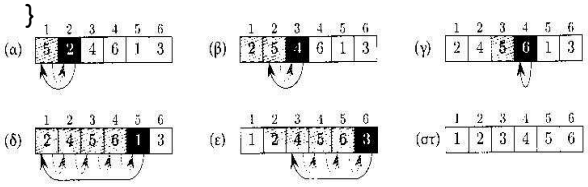
Ταξινόμηση - Ο αλγόριθμος InsertionSort()

```
Algorithm InsertionSort (A[1..n]) {
    int key, i, j;
    for (j = 2; j ≤ n; j++) {
        key = A[j];
        i = j-1;
        while (i > 0 && A[i] > key) {
            A[i+1] = A[i];
            i = i-1;
        }
        A[i+1] = key;
    }
    return A;
}
```

Βασική Ιδέα

Επαναληπτικά, επιλέγεται το επόμενο στοιχείο από το μη-ταξινομημένο κομμάτι A[j..n-1] του πίνακα και τοποθετείται στην κατάλληλη θέση του ταξινομημένου κομματιού A[0..j-1] του πίνακα.

Πως λειτουργεί ο αλγόριθμος αν A = <5,2,4,6,1,3>;



Η χρονική πολυπλοκότητα της InsertionSort() είναι $\Theta(n^2)$.

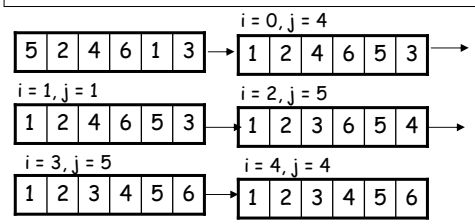
Ταξινόμηση - Ο αλγόριθμος SelectionSort()

Βασική Ιδέα

Επαναληπτική αναζήτηση, στο μη-ταξινομημένο κομμάτι του πίνακα A, του στοιχείου εκείνου που θα αποτελέσει το επόμενο στοιχείο στο ταξινομημένο κομμάτι του πίνακα (το μέγεθος του ταξινομημένου κομματιού του πίνακα αυξάνει κατά ένα μετά από κάθε επανάληψη).

```
Procedure SelectionSort(table A[0..n-1]) {
// sort A by repeatedly selecting the smallest element
// from the unsorted part

  for (i = 0; i < n-1; i++) {
      j = i;
      for (k = i+1; k < n; k++)
          if (A[k] < A[j]) j = k;
      swap(A[i], A[j]);
  }
}
```

Πως λειτουργεί ο αλγόριθμος αν A = <5,2,4,6,1,3>;



Άσκηση για το σπίτι

Αποδείξτε ότι η χρονική πολυπλοκότητα της SelectionSort() είναι $\Theta(n^2)$.

Ταξινόμηση - Ο αλγόριθμος HeapSort()

Βασικές Ιδέες

Αποδοτικότερη έκδοση του SelectionSort(), στην οποία το μη-ταξινομημένο κομμάτι του A διατηρείται ως ένας σωρός που το ελάχιστο στοιχείο του βρίσκεται στη θέση A[n-1] του πίνακα, ενώ τα υπόλοιπα στοιχεία είναι αποθηκευμένα με την κατάλληλη σειρά σε φθίνουσες θέσεις του A ξεκινώντας από την A[n-2].

Υποθέτουμε ότι αρχικά τα στοιχεία που είναι αποθηκευμένα στον πίνακα έχουν την ιδιότητα της μερικής διάταξης (αυτό επιτυγχάνεται με κλήση μιας διαδικασίας που ονομάζεται InitializeHeap()).

Διατρέχουμε όλα τα στοιχεία του A και για κάθε ένα από αυτά το ανταλλάσουμε με το στοιχείο A[n-1] (δηλαδή το ελάχιστο του σωρού) και εκτελούμε μια διαδικασία, που ονομάζεται Heapify(), η οποία επαναφέρει την ιδιότητα της μερικής διάταξης του σωρού (εφαρμόζοντας τις τεχνικές που συζητήθηκαν στην Ενότητα 8).

```
Procedure HeapSort( table A[0..n-1]) {
   InitializeHeap(A);
   // Η InitializeHeap() αναλαμβάνει τη
   // μερική ταξινόμηση του αρχικού
   //πίνακα ώστε αυτός να μετατραπεί
   // σε σωρό με ελάχιστο στην A[n-1]

   for (i = 0; i < n-1; i++) {
      swap(A[i], A[n-1]);
      // επαναληπτική αποθήκευση του
      // μικρότερου στοιχείου του
      // σωρού στη θέση i του πίνακα

      Heapify(A[i+1…n-1]);
      // επαναφορά της μερικής
      // διάταξης του σωρού που μπορεί
      // να έχει καταστραφεί στη ρίζα
   }
}
```

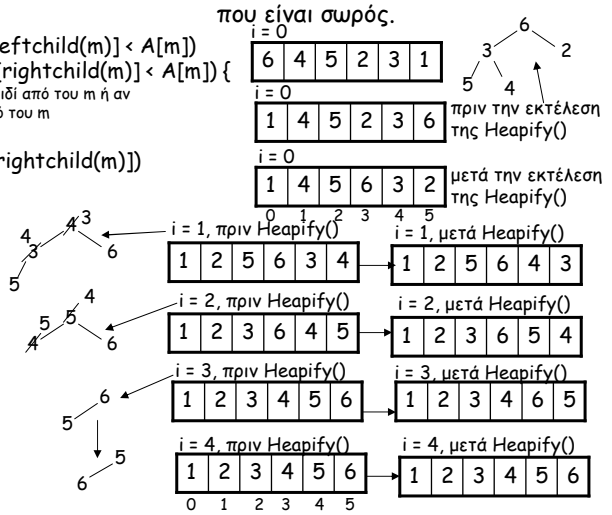

Ταξινόμηση - Ο αλγόριθμος HeapSort()

```
procedure Heapify(table A[i..j]) {
// Αρχικά, το κομμάτι A[i..j-1] είναι μερικώς ταξινομημένο, ενώ μετά την
εκτέλεση της Heapify() το κομμάτι A[i..j] είναι μερικώς ταξινομημένο

   m = j;
   while ((leftchild(m) ≥ i AND A[leftchild(m)] < A[m])
      OR (rightchild(m) ≥ i  AND A[rightchild(m)] < A[m]) {
// αν υπάρχει αριστερό παιδί με μικρότερο κλειδί από του m ή αν
// υπάρχει δεξιό παιδί με μικρότερο κλειδί από του m
      if (rightchild(m) ≥ i ) {
         if (A[leftchild(m)] < A[rightchild(m)])
            p = leftchild(m)
         else p = rightchild(m);
      }
      else p = i;
      swap(A[m], A[p]);
      m = p;
   }
}
```

Στον παραπάνω κώδικα ισχύει ότι:

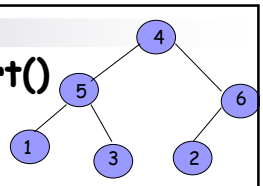
- leftchild(m) = 2m-n
- rightchild(m) = 2m-n-1

Παράδειγμα Εκτέλεσης της HeapSort(A) όπου A = [6,4,5,2,3,1] που είναι σωρός.



Ταξινόμηση - Ο αλγόριθμος HeapSort()

Procedure InitializeHeap(Table A[0..n-1]) {
// Μετατροπή του A σε σωρό
for (i=1; i < n; i++) Heapify(A[0..i]);
}



αρχικά

0	1	2	3	4	5
2	3	1	6	5	4

μετά ανακύκλωση i = 1

0	1	2	3	4	5
2	3	1	6	5	4

μετά ανακύκλωση i = 2

2	3	1	6	5	4

μετά ανακύκλωση i = 3

6	3	1	2	5	4

μετά ανακύκλωση i = 4

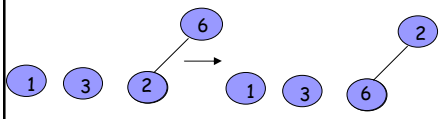
6	3	5	2	1	4

μετά ανακύκλωση i = 5

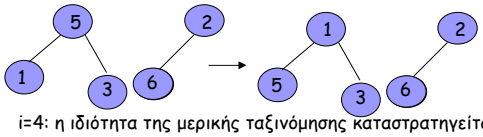
6	4	5	2	3	1

i=1: τα φύλλα 3 και 2 δεν έχουν παιδιά ⇒ η Heapify() δεν επιφέρει καμία αλλαγή!

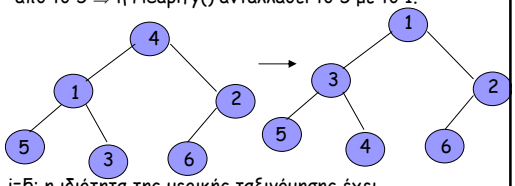
i=2: τα φύλλα 1, 3 και 2 δεν έχουν παιδιά ⇒ η Heapify() δεν επιφέρει καμία αλλαγή!



i=3: ο κόμβος 6 είναι ο μόνος που έχει παιδιά ⇒ η Heapify() ανταλλάσει το 6 με το παιδί του, που έχει τιμή 2 και είναι μικρότερο.



i=4: η ιδιότητα της μερικής ταξινόμησης καταστρατηγείται από το 5 ⇒ η Heapify() ανταλλάσει το 5 με το 1.



i=5: η ιδιότητα της μερικής ταξινόμησης έχει καταστρατηγηθεί στη ρίζα ⇒ η Heapify() ανταλλάσει το 4 με το 1 και στη συνέχεια το 4 με το 3.

Ταξινόμηση - Ο αλγόριθμος HeapSort()

Χρονική Πολυπλοκότητα της HeapSort()

≤ (Χρονική Πολυπλοκότητα της InitializeHeap()) +
((n-1) * Χρονική Πολυπλοκότητα της Heapify(A[0..n-1]))

- (Χρονική Πολυπλοκότητα της InitializeHeap()) ≤
  (n-1) * (Χρονική Πολυπλοκότητα της Heapify(A[0..n-1]))
- Χρονική Πολυπλοκότητα της Heapify(A[0..n-1]) = O(logn).
- Επομένως:

Χρονική Πολυπλοκότητα της HeapSort() = O(nlogn).

Χωρική Πολυπλοκότητα της HeapSort()

- Τα στοιχεία αποθηκεύονται κάθε χρονική στιγμή στον ίδιο τον πίνακα A.
- Ο αλγόριθμος είναι πολύ αποδοτικός ως προς την χωρική πολυπλοκότητα.
- Ο αλγόριθμος είναι βέλτιστος ως προς τη χρονική πολυπλοκότητά του.

Ταξινόμηση - Ο αλγόριθμος MergeSort()

Τεχνική «Διαίρει και Κυρίευε»

Η τεχνική περιλαμβάνει τρία βήματα:

1. **Διαίρεση** του προβλήματος σε διάφορα υποπροβλήματα που είναι παρόμοια με το αρχικό πρόβλημα αλλά μικρότερου μεγέθους.
2. **Κυριαρχία** επί των υποπροβλημάτων, επιλύοντάς τα αναδρομικά, μέχρι αυτά να γίνουν αρκετά μικρού μεγέθους οπότε και τα επιλύουμε απευθείας.
3. **Συνδυασμός** των επιμέρους λύσεων των υποπροβλημάτων ώστε να συνθέσουμε μια λύση του αρχικού προβλήματος.

Ο αλγόριθμος MergeSort() ακολουθεί το μοντέλο «Διαίρει και Κυρίευε»:

1. **Διαίρεση:** Η προς ταξινόμηση ακολουθία των n στοιχείων διαιρείται σε δύο υπακολουθίες των n/2 στοιχείων η κάθε μια.
2. **Κυριαρχία:** Οι δυο υπακολουθίες ταξινομούνται καλώντας αναδρομικά τη MergeSort() για κάθε μια από αυτές. Η αναδρομή εξαντλείται όταν η προς ταξινόμηση υπακολουθία έχει μήκος 1 (αφού κάθε υπακολουθία μήκους 1 είναι ταξινομημένη).
3. **Συνδυασμός:** Συγχωνεύουμε τις δύο ταξινομημένες υπακολουθίες ώστε να σχηματίσουμε την τελική ταξινομημένη ακολουθία.

Ταξινόμηση - Ο αλγόριθμος MergeSort()

Διαδικασία Συγχώνευσης

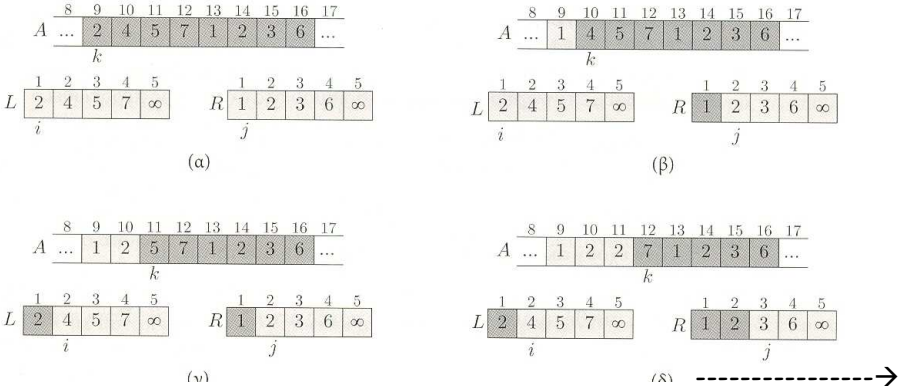
- Χρησιμοποιείται μια βοηθητική διαδικασία Merge(table A, int p,q,r), της οποίας οι παράμετροι είναι ένας πίνακας A και τρεις ακέραιοι, δείκτες στον πίνακα, τ.ω. $p \le q < r$.

- Η διαδικασία προϋποθέτει ότι οι υποπίνακες A[p..q] και A[q+1..r] είναι ταξινομημένοι και συγχωνεύει τα στοιχεία τους ώστε να σχηματίσει μια ενιαία ταξινομημένη υπακολουθία, η οποία αντικαθιστά την αρχική A[p..r].

- Αποθηκεύουμε στο τέλος κάθε υποπίνακα έναν κόμβο φρουρό, ώστε όταν αυτός προσεγγισθεί, η τιμή του να είναι σίγουρα μεγαλύτερη από την τιμή του τρέχοντος στοιχείου στον άλλο υποπίνακα.

```
void Merge(table A, int p,q,r) {
   n1 = q-p+1;
   n2 = r-q;
   // δημιουργία υποπινάκων L[0..n1] και R[0..n2]
   for (i= 0; i < n1; i++)
      L[i] = A[p+i];
   for (i = 0; i < n2; i++)
      R[i] = A[q+1+i];
   i = j = 0;
   L[n1] = R[n2] = ∞; // κόμβοι φρουροί των πινάκων
   for (k = p; k <= r; k++) {
      if (L[i] < R[j]) {
         A[k] = L[i];
         i++;
      }
      else {
          A[k] = R[j];
          j++;
      }
   } /* for */  } /*Merge */
```

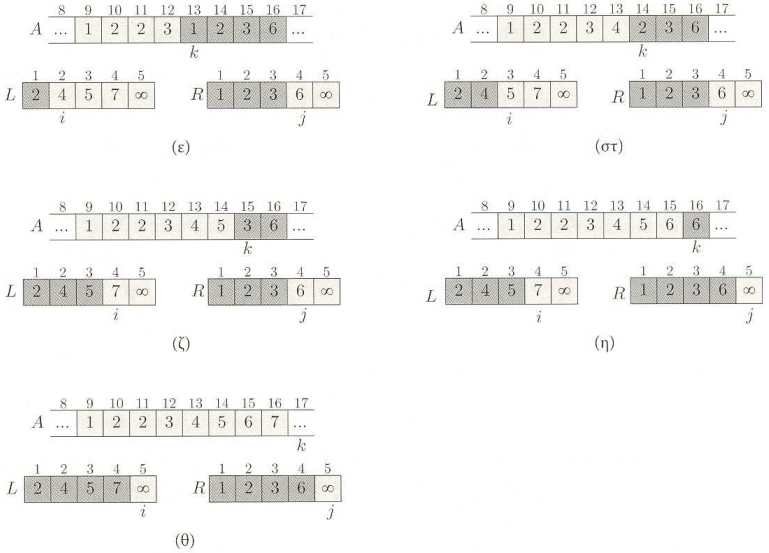
Ταξινόμηση - Παράδειγμα εκτέλεσης της Merge



Παράδειγμα εκτέλεσης της Merge(A, 9, 12, 16), όπου A[9..16] = <2,4,5,7,1,2,3,6>

- Οι ελαφρά σκιασμένες θέσεις στον πίνακα A έχουν λάβει τις τελικές τους τιμές, ενώ οι έντονα σκιασμένες θέσεις θα αντικατασταθούν στη συνέχεια από άλλες τιμές.
- Οι ελαφρά σκιασμένες θέσεις στους πίνακες L,R περιέχουν τιμές οι οποίες δεν έχουν ακόμη επανακαταχωρηθεί στον A, ενώ οι έντονα σκιασμένες το αντίθετο.

Ταξινόμηση - Παράδειγμα εκτέλεσης της Merge



Ταξινόμηση - Ο αλγόριθμος MergeSort()

Συνθήκη που ισχύει αναλλοίωτη κατά την εκτέλεση της Merge()
«Στην αρχή κάθε επανάληψης της 3ης for, ο υποπίνακας A[p..k-1] περιέχει τα k-p μικρότερα στοιχεία των L[0..n1] και R[0..n2], ταξινομημένα. Επιπλέον, τα L[i] και R[j] είναι τα μικρότερα στοιχεία των υποπινάκων αυτών που δεν έχουν ακόμη καταχωρηθεί στον πίνακα A.»

Ο ισχυρισμός αυτός αποδεικνύεται με επαγωγή στο k. Η απόδειξη αφήνεται ως άσκηση για το σπίτι!

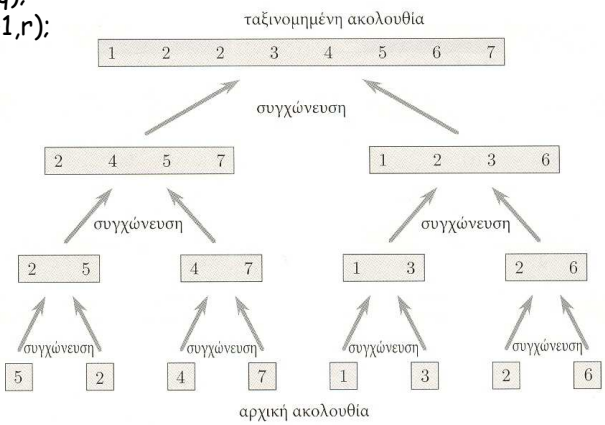
Ποια είναι η χρονική πολυπλοκότητα της Merge();

Θ(n1+n2) = Θ(n).

```
void Merge(table A, int p,q,r)
{
   n1 = q-p+1;  ----------→ Θ(1)
   n2 = r-q;    ----------→ Θ(1)

   for (i=0; i < n1; i++) --
      L[i] = A[p+i];      -→Θ(n1)
   for (i = 0; i < n2; i++) --
      R[i] = A[q+1+i];  -→ Θ(n2)
   i = j = 0;   ----------→ Θ(1)
   L[n1] = R[n2] = ∞; ---→Θ(1)
   for (k = p; k <= r; k++)
      if (L[i] < R[j]) {
         A[k] = L[i];  i++;
      }                          → Θ(n1+n2)
      else {
         A[k] = R[j]; j++;
      }
}
```


Ταξινόμηση - Ο αλγόριθμος MergeSort()

```
void MergeSort(table A, int p, r) {
    if (p < r) {
        q = ⌊(p+r)/2⌋;
        MergeSort(A,p,q);
        MergeSort(A,q+1,r);
        Merge(A,p,q,r);
    }
}
```



Ανάλυση Αλγορίθμων τύπου «Διαίρει και Κυρίευε»

Έστω T(n) ο χρόνος εκτέλεσης ενός αλγορίθμου που λειτουργεί βάσει της τεχνικής «διαίρει και κυρίευε», όπου n είναι το μέγεθος της εισόδου.

Αν το n είναι αρκετά μικρό, π.χ., $n \leq c$, όπου c είναι μια σταθερά, η απευθείας λύση απαιτεί χρόνο $\Theta(1)$.

Έστω ότι κατά τη διαίρεση του προβλήματος προκύπτουν a υποπροβλήματα που το καθένα έχει μέγεθος το 1/b του μεγέθους του πλήρους προβλήματος.

Έστω ότι η διαίρεση του προβλήματος σε υποπροβλήματα απαιτεί χρόνο D(n) και ότι η σύνθεση των επιμέρους λύσεων των υποπροβλημάτων για το σχηματισμό της πλήρους λύσης απαιτεί χρόνο C(n). Τότε:

$$T(n) = \begin{cases} \Theta(1) & \text{αν } n \leq c \\ aT(n/b) + D(n) + C(n) & \textit{διαφορετικά} \end{cases}$$

Ανάλυση αλγορίθμου MergeSort()

- Υποθέτουμε ότι το n είναι κάποια δύναμη του 2 (για λόγους απλότητας).
- Η MergeSort() για ένα μόνο στοιχείο απαιτεί σταθερό χρόνο $\Theta(1)$.
- Όταν εκτελείται σε (υπο)πίνακα με n > 1 στοιχεία, αναλύουμε τη χρονική πολυπλοκότητα ως εξής:
  - **Διαίρει:** D(n) = $\Theta(1)$ (υπολογισμός του μεσαίου στοιχείου του υποπίνακα σε χρόνο $\Theta(1)$)
  - **Κυρίευε:** Αναδρομική επίλυση δύο υπο-προβλημάτων μεγέθους n/2 τα καθένα. Άρα, το χρονικό κόστος του βήματος αυτού είναι 2T(n/2).
  - **Συνδύασε:** Η κλήση της Merge() απαιτεί χρόνο $\Theta(n) \Rightarrow C(n) = \Theta(n)$.

Η αναδρομική σχέση που περιγράφει τη χρονική πολυπλοκότητα της MergeSort() είναι:

$$T(n) = \begin{cases} c_1 & \text{αν } n = 1, \\ 2T(n/2) + c_2 n & \text{αν } n > 1, \end{cases}$$

όπου c_1 και c_2 είναι σταθερές.

Άσκηση για το σπίτι
Αποδείξτε (είτε επαγωγικά ή με τη μέθοδο της αντικατάστασης) ότι T(n) = $\Theta(n \log n)$.

Ταξινόμηση - Ο αλγόριθμος QuickSort()

Βασίζεται στην τεχνική «Διαίρει και Κυρίευε».

Διαίρεση: Επιλέγεται ένα στοιχείο x της προς ταξινόμηση ακολουθίας (π.χ., το τελευταίο) και η αρχική ακολουθία A[p..r] χωρίζεται στις υπακολουθίες A[p..q-1] και A[q+1..r] έτσι ώστε κάθε στοιχείο της A[p..q-1] να είναι μικρότερο του x, ενώ κάθε στοιχείο της A[q+1..r] να είναι μεγαλύτερο του x. Μέρος του βήματος αυτού είναι και ο υπολογισμός της κατάλληλης θέσης q του πίνακα στην οποία θα πρέπει να βρίσκεται το στοιχείο x μετά την ταξινόμηση.

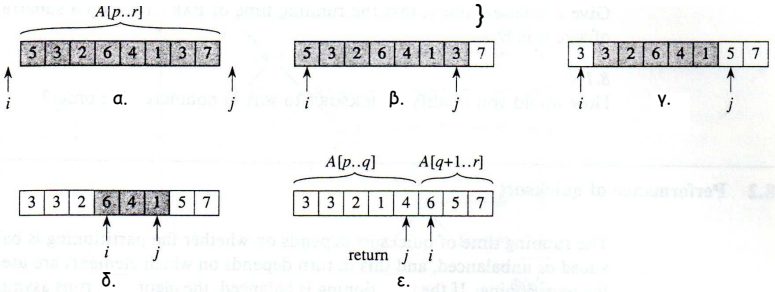
Κυριαρχία: Ταξινομούμε τις δύο υπακολουθίες A[p..q-1] και A[q+1..r] με αναδρομικές κλήσεις της QuickSort().

Συνδυασμός: Δεν χρειάζεται κάποια ιδιαίτερη ενέργεια για το συνδυασμό των δυο υπακολουθιών A[p..q-1] και A[q+1..r], αφότου αυτές ταξινομηθούν (λόγω του τρόπου που αυτές δημιουργήθηκαν). Δηλαδή, η συνολική ακολουθία A[p..r] είναι και αυτή ταξινομημένη χωρίς την εκτέλεση περαιτέρω ενεργειών.

Ταξινόμηση - Ο αλγόριθμος QuickSort()

```
void QuickSort(Table A, int p,
                        int r) {
   if (p < r) {
         q = Partition(A,p,r);
         QuickSort(A,p,q);
         QuickSort(A,q+1,r);
}
```

```
int Partition(Table A, int p, int r) {
  int x, i, j;

  x = A[p]; i = p-1; j = r+1;
  while (TRUE) {
     repeat j = j-1; until A[j] <= x;
     repeat i = i+1; until A[i] >= x;
     if (i < j)  swap(A[i],A[j]);
     else return j;
   }
}
```



Ταξινόμηση - Ο αλγόριθμος QuickSort()

- Ο χρόνος εκτέλεσης χειρότερης περίπτωσης της QuickSort() για ένα πίνακα n στοιχείων είναι $\Theta(n^2)$.
- Ο πίνακας είναι εξ αρχής ταξινομημένος.
  - Η Partition() παράγει ένα τμήμα με n-1 στοιχεία και ένα τμήμα με 1 μόνο στοιχείο.

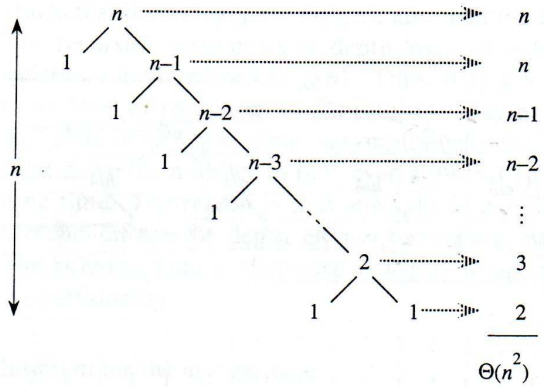
```
int Partition(Table A, int p, int r) {
  int x, i, j;
  x = A[p]; i = p-1; j = r+1;
  while (TRUE) {
      repeat j = j-1; until A[j] <= x;
      repeat i = i+1; until A[i] >= x;
      if (i < j)  swap(A[i],A[j]);
      else return j;
   }
}
```

0	1	2	3	4	5	6	7
10	20	30	40	50	60	70	80

  - Αν το προβληματικό αυτό σενάριο προκύψει σε κάθε εκτέλεση της Partition(), η αναδρομική σχέση που περιγράφει την QuickSort() είναι:

    $T(1) = \Theta(1)$ και $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

Ταξινόμηση - Ο αλγόριθμος QuickSort()



Δένδρο αναδρομής της QuickSort() όπου η Partition() διαμερίζει επαναληπτικά τον πίνακα σε ένα τμήμα μεγέθους 1 και σε ένα τμήμα μεγέθους n-1.

QuickSort()

☺ Ο καλύτερος χρόνος εκτέλεσηςτης QuickSort() είναι Θ(nlogn).

- Η Partition() παράγει δύο τμήματα με n/2 στοιχεία το καθένα.

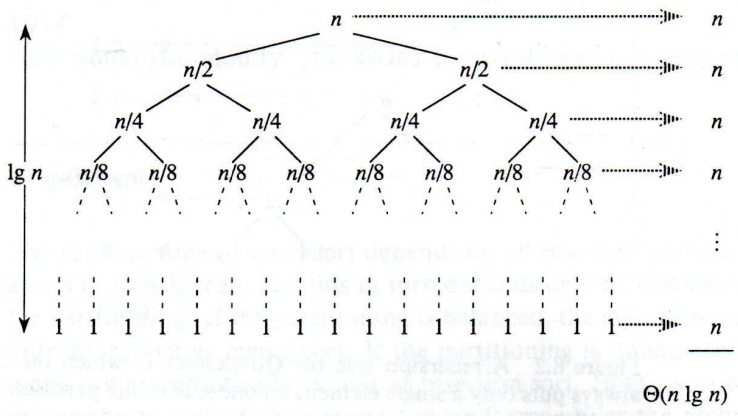
```
int Partition(Table A, int p, int r) {
  int x, i, j;

  x = A[p]; i = p-1; j = r+1;
  while (TRUE) {
      repeat j = j-1; until A[j] <= x;
      repeat i = i+1; until A[i] >= x;
      if (i < j)  swap(A[i],A[j]);
      else return j;
   }
}
```

0	1	2	3	4	5	6	7
50	60	80	70	10	20	40	30

- Η αναδρομική σχέση που περιγράφει τη χρονική πολυπλοκότητα της QuickSort() σε αυτή την περίπτωση είναι:

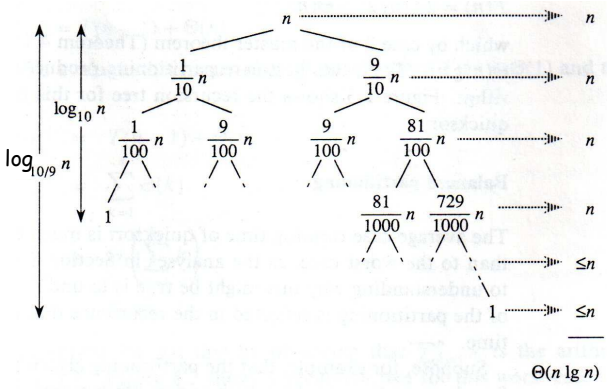
  T(1) = Θ(1) και T(n) = 2T(n/2) + Θ(n) = Θ(nlogn)

Ταξινόμηση - Ο αλγόριθμος QuickSort()



Δένδρο αναδρομής της QuickSort() όπου η Partition() διαμερίζει επαναληπτικά τον πίνακα σε δύο τμήματα μεγέθους n/2 το καθένα.

Ταξινόμηση - Ο αλγόριθμος QuickSort()



- Οποιαδήποτε διαμέριση του πίνακα (από την Partition()) σε δύο τμήματα τ.ω. η αναλογία μεταξύ των μηκών των δύο τμημάτων είναι σταθερή, οδηγεί σε Θ(logn) επίπεδα αναδρομικών κλήσεων που το καθένα κοστίζει Θ(n).

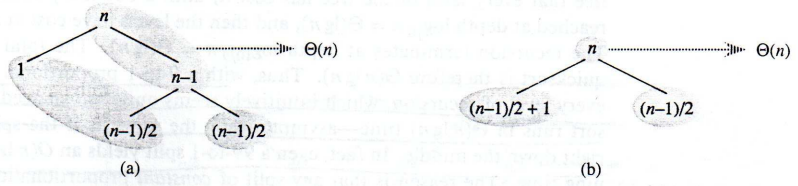
Διαίσθηση για την Μέση Περίπτωση

Υπόθεση

- Κάθε μετάθεση των προς ταξινόμηση αριθμών προκύπτει, ως είσοδος της QuickSort(), με την ίδια πιθανότητα όπως όλες οι υπόλοιπες μεταθέσεις.

Παράδειγμα

- Οι διαμερίσεις που παράγουν εξισορροπημένα, ως προς το μέγεθος, τμήματα του πίνακα εναλλάσσονται με εκείνες που δεν το επιτυγχάνουν αυτό.



Έκδοση QuickSort() που χρησιμοποιεί τυχαιότητα

Βασική Ιδέα

- Σε κάθε βήμα, πριν να εκτελεστεί η διαμέριση του πίνακα, ανταλλάσσουμε το στοιχείο A[p] με ένα άλλο στοιχείο του πίνακα που επιλέγεται με τυχαιότητα.
- Αυτό αναμένεται να οδηγήσει σε διαμέριση του πίνακα σε δύο τμήματα που είναι εξισορροπήμενα ως προς το μέγεθός τους.

```
void RandomizedQuickSort(Table A,
                         int p, int r) {
   if (p < r) {
      q = RandPartition(A,p,r);
      RandomizedQuickSort(A,p,q);
      RandomizedQuickSort(A,q+1,r);
   }
}

void RandomizedPartition(Table A,
                         int p, int r) {
    i = random(p,r);
   swap(A[p], A[i]);
   return Partition(A,p,r);
}
```