ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

Τμήμα Μηχανικών Πληροφορικής και
Ηλεκτρονικών Συστημάτων

Αντικειμενοστρεφής Προγραμματισμός (Object Oriented Programming)

Αλγόριθμοι ταξινόμησης (Sorting Algorithms)

Παναγιώτης Σφέτσος, PhD
http://aetos.it.teithe.gr/~sfetsos/
sfetsos@it.teithe.gr

Αλγόριθμοι Ταξινόμησης

Στην ενότητα αυτή θα μελετηθούν οι παρακάτω αλγόριθμοι ταξινόμησης:

- **Bubble Sort -** Ταξινόμηση φυσαλίδας
- **Insertion Sort -** Ταξινόμηση με Εισαγωγή
- **Selection Sort -** Ταξινόμηση με Επιλογή
- **Quick Sort** – Γρήγορη Ταξινόμηση
- **Merge Sort -** Ταξινόμηση με Συγχώνευση

Αλγόριθμοι Ταξινόμησης

Το πρόβλημα της ταξινόμηση θεωρείται ***θεμελιώδες αλγοριθμικό πρόβλημα***:

- Η ταξινόμηση απαραίτητη σε πολλές εφαρμογές *(περίπου το 25% του υπολογιστικού χρόνου)*.

- Λειτουργίες όπως η αναζήτηση γίνεται γρηγορότερα όταν τα στοιχεία είναι ταξινομημένα. Είδαμε ότι η αναζήτηση σε ταξινομημένα στοιχεία. (σε πίνακα) ολοκληρώνεται σε χρόνο O(log n), έναντι O(n) στην σειριακή.

- Ενδιάμεση λειτουργία για σχεδιασμό άλλων αλγορίθμων π.χ. δενδρικών δομών.

- Διαφορετικές τεχνικές, με διαφορετικούς αλγόριθμους, που βρίσκουν εφαρμογή σε άλλες σημαντικές εφαρμογές.

Αλγόριθμοι Ταξινόμησης

Τεχνικές ταξινόμησης:

- **Τεχνικές ανταλλαγής** *(Exchange techniques)*, π.χ. οι bubble και insertion sort, με χρονική πολυπλοκότητα $O(n^2)$.

- **Τεχνικές δένδρικών δομών** *(Tree-based techniques)*, π.χ. η δυαδική δενδρική ταξινόμηση *(binary tree sorting)* με πολυπλοκότητα $O(n \log 2^n)$.

- **Αναδρομικές τεχνικές** *(Recursive techniques)*, π.χ. quick και merge sort με πολυπλοκότητα $O(n \log 2^n)$ αλλά καλύτερης απόδοσης από άλλες τεχνικές.

Bubble Sort *(1/6)*

Ταξινομήσεις με Ανταλλαγή *(Exchange Sorting)*

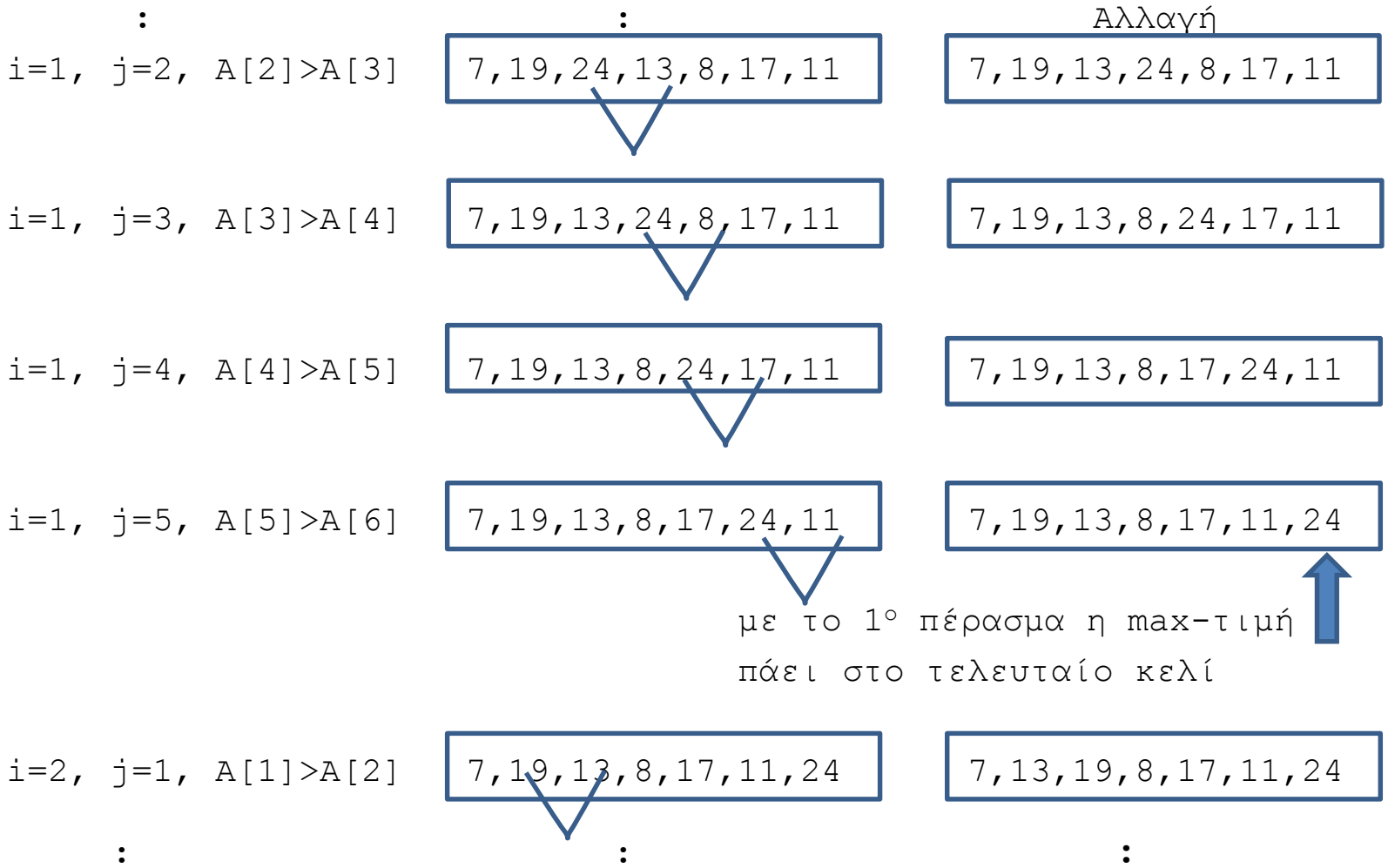
Ταξινόμηση φυσαλίδας *(Bubble Sort)*

- Ο αλγόριθμος αυτός συγκρίνει κάθε στοιχείο με το επόμενό του σαρώνοντας τον πίνακα, αναδύοντας τις μικρότερες τιμές στις κατάλληλες θέσεις σαν 'φυσαλίδες'.
- Στην καλύτερη περίπτωση, με την πρώτη επανάληψη (αν ο πίνακας είναι ταξινομημένος), επομένως δεν χρειάζεται άλλη επανάληψη, και τότε η χρονική πολυπλοκότητα είναι O(n).
- Στην χειρότερη περίπτωση ο αλγόριθμος απαιτεί n-1 επαναλήψεις. Η πρώτη επανάληψη κάνει n-1 συγκρίσεις, η δεύτερη επανάληψη κάνει n-2 συγκρίσεις, κ.ο.κ. Η τελευταία επανάληψη κάνει 1 σύγκριση. Έτσι, ο συνολικός αριθμός των συγκρίσεων είναι:

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

Bubble Sort *(2/6)*

- Έστω ο πίνακας `int nums[] = {7,19,24,13,8,17,11};`

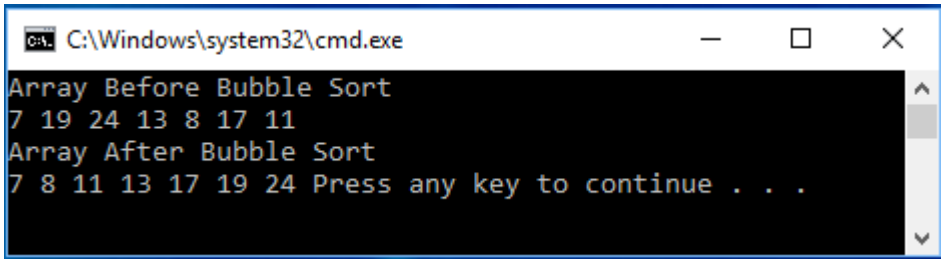


Bubble Sort *(3/6)*

```
class BubbleSortExample {
  static void bubbleSort(int[] arr) {
  int n = arr.length;
  int temp = 0;
  for(int i=0; i < n; i++){
    for(int j=1; j < (n-i); j++){
       if(arr[j-1] > arr[j])
        {
          //allagi timon
          temp = arr[j-1];
          arr[j-1] = arr[j];
          arr[j] = temp;
        }
   }}}
```

Bubble Sort *(4/6)*

```
public static void main(String[] args) {
   int arr[] ={7,19,24,13,8,17,11};
   System.out.println("Prin tin taxinomisi");
   for(int i=0; i < arr.length; i++){
     System.out.print(arr[i] + " ");}
    System.out.println();
    bubbleSort(arr);
    System.out.println("Meta tin taxinomisi");
    for(int i=0; i < arr.length; i++){
      System.out.print(arr[i] + " ");}
    }}
```



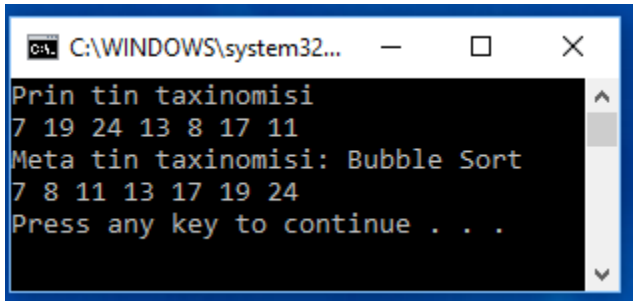
Bubble Sort *(5/6)*

Παραλλαγή: Έλεγχος για ήδη ταξινομημένο πίνακα (flag)

```
class BubbleSort {
 public static void main(String[] args) {
   int nums[] = {7,19,24,13,8,17,11};
   System.out.println("Prin tin taxinomisi");
   for (int i = 0; i < nums.length; i++)
      System.out.print(nums[i]+" ");
   System.out.println();
   BSort(nums);
   System.out.println("Meta tin taxinomisi");
   for (int i = 0; i < nums.length; i++)
       System.out.print(nums[i]+" ");
   System.out.println(); }
```

Bubble Sort *(6/6)*

```
public static void BSort(int[] A)
{
 int i, j;
 int temp;
 boolean flag;
 for (i=1; i < A.length; i++) {
   flag=true;
   for (j=0; j < A.length-i; j++)
    if (A[j] > A[j+1])
     {
      temp=A[j];
      A[j]= A[j+1];
      A[j+1]=temp;
      flag=false;
     }
     if (flag) return; }}}
```



Με κάθε αλλαγή αλλάζουμε και την τιμή της μεταβλητής flag σε false. Αν η μεταβλητή διατηρεί την τιμή true, σημαίνει ότι δεν έχει γίνει κάποια ανταλλαγή τιμών, επομένως ο πίνακας είναι ταξινομημένος και δεν χρειάζεται να γίνει άλλο ‘πέρασμα’.

Insertion Sort *(1/6)*

Ταξινομήσεις με Εισαγωγή *(Insertion Sort)*

Η **ταξινόμηση με εισαγωγή** εισάγει ένα-ένα τα στοιχεία της λίστας στη σωστή τους θέση.

Σε οποιοδήποτε βήμα (για μερικώς ταξινομημένο πίνακα A[1..(i-1)] *):*

- εισάγεται το στοιχείο A[i] στην ακολουθία A[1..(i-1)] στη σωστή θέση,
- μετακινούνται όλα τα στοιχεία που είναι μεγαλύτερα του A[i] μια θέση δεξιά.
- έτσι στην επανάληψη – i για να εισαχθεί ένα στοιχείο στον πίνακα μεγέθους n, απαιτούνται n-συγκρίσεις και n-μετακινήσεις.

Αν T(n) είναι η πολυπλοκότητα του αλγορίθμου της ταξινόμησης με εισαγωγή και c μια σταθερά που εκφράζει τις λοιπές εργασίες όπως η εκχώρηση τιμών και λοιπών συγκρίσεων, τότε:

$$
\begin{aligned}
T(n) &= (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n-1) + c) \\
&= 2(1 + 2 + \cdots + n - 1) + c(n-1) \\
&= 2\frac{(n-1)n}{2} + cn - c = n^2 - n + cn - c \\
&= O(n^2)
\end{aligned}
$$

Insertion Sort *(2/6)*

Ταξινομήσεις με Εισαγωγή *(Insertion Sort)*

Ο αλγόριθμος αυτός είναι πιο γρήγορος από τον αλγόριθμο bubble sort:

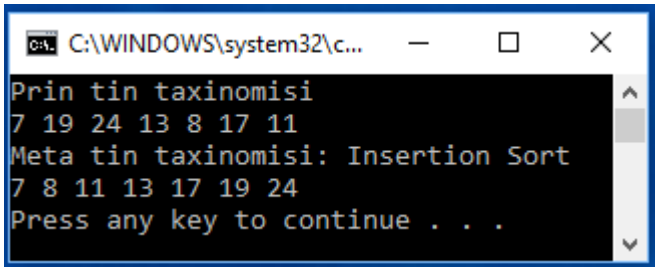
- Έστω ο πίνακας `int nums[]={7,19,24,13,8,17,11};`
- Θεωρώ ότι το 1ο στοιχείο βρίσκεται στην σωστή ταξινομική του θέση **A[0]=7**.
- Ξεκινώ από το 2ο στοιχείο current=19 (για i=1 και j=1) και ελέγχω αν 7>19, δηλ. αν το προηγούμενο από το current στοιχείο είναι μεγαλύτερό του, εδώ δεν είναι άρα είναι ταξινομημένο αριστερά και έτσι δεν μπαίνει στο while-loop.
- Συνεχίζω με το 3ο στοιχείο current=24 (για i=2 και j=2) και ελέγχω αν 19>24, που δεν είναι άρα είναι ταξινομημένο αριστερά και έτσι δεν μπαίνει στο while-loop.
- Συνεχίζω με το 4ο στοιχείο current=13 (για i=3 και j=3) και ελέγχω αν 24>13, που είναι άρα μπαίνει στο while-loop για να 'σπρώξει' τα μεγαλύτερα του 13, πρώτα το 19 και μετά το 24 (while loop), προς τα δεξιά, κ.ο.κ.

Insertion Sort *(3/6)*

```
class InsertSort {
 public static void main(String[] args) {
    int nums[] = {7,19,24,13,8,17,11};
   System.out.println("Prin tin taxinomisi");
   for (int i = 0; i < nums.length; i++)
       System.out.print(nums[i]+" ");
   System.out.println();
   InsSort(nums);
   System.out.println("Meta tin taxinomisi: Insertion Sort");
   for (int i = 0; i < nums.length; i++)
       System.out.print(nums[i]+" ");
    System.out.println(); }
```

Insertion Sort *(4/6)*

```
public static void InsSort(int[] A) {
   for (int i=1; i < A.length; i++) {
     int current = A[i];
     int j=i;

    // metakinisi megalyteron timon dexia
     while (j>0 && A[j-1] > current) {
        A[j] = A[j-1];
        j--;}
        A[j] = current;
    } } }
```



- **Υπάρχουν και άλλες παραλλαγές του αλγορίθμου(άσκηση-1)**

Insertion Sort *(5/6)*

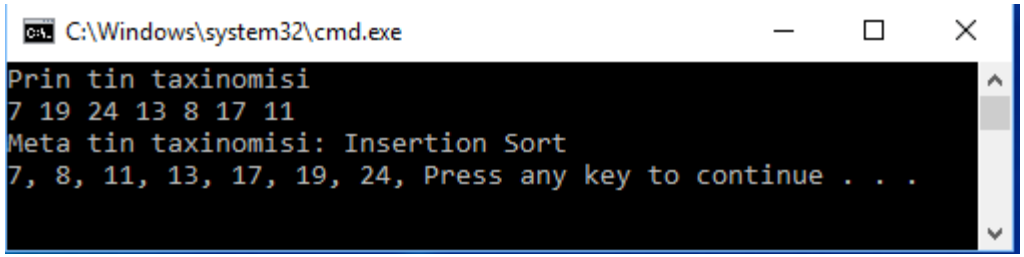
Παραλλαγή:

```
class InsertionSort {
  public static void main(String a[]){
     int[] arr1 = {7,19,24,13,8,17,11};
     System.out.println("Prin tin taxinomisi");
     for (int i = 0; i < arr1.length; i++)
         System.out.print(arr1[i]+" ");
     System.out.println();

     int[] arr2 = insertionsort(arr1);

     System.out.println("Meta tin taxinomisi: Insertion Sort");
     for(int i:arr2){
           System.out.print(i);
           System.out.print(", "); } }
```

Insertion Sort *(6/6)*

Παραλλαγή :

```
public static int[] insertionsort(int[] ar){
   int temp;
   for(int i = 1; i < ar.length; i++) {
      for(int j = i ; j > 0 ; j--){
          if(ar[j] < ar[j-1]){
              temp = ar[j];
              ar[j] = ar[j-1];
              ar[j-1] = temp;
           }
       }
   }
       return ar;
  }
}
```



Selection Sort *(1/4)*

Ταξινομήσεις με Επιλογή *(Selection Sort)*

Ο αλγόριθμος **τοποθετεί κάθε τιμή** *(μία την φορά)* **στην σωστή τελική ταξινομημένη θέση**. Δηλ. ο αλγόριθμος **επιλέγει** την τιμή που θα πάει σε κάθε θέση του πίνακα.

Τα βήματα:

1. Επιλογή του μικρότερου στοιχείου
2. Ανταλλαγή με το πρώτο στοιχείο
3. Επανάληψη των βημάτων 1 και 2 για τα εναπομείναντα στοιχεία

Το μικρότερο στοιχείο μεταξύ n στοιχείων μπορεί να βρεθεί με την χρήση ενός loop σε n-1 συγκρίσεις και ανταλλαγή στην $1^{η}$ θέση. Για το $2^{ο}$ μικρότερο ελέγχονται τα υπόλοιπα n-1 στοιχεία, κ.ο.κ. Άρα ο χρόνος εκτέλεσης του αλγόριθμου selection sort για n στοιχεία είναι:

$(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 = O(n^2)$ *(και από τα εστιασμένα loops)*

Selection Sort *(2/4)*

Ταξινομήσεις με Επιλογή *(Selection Sort)*

- Έστω ο πίνακας `int nums[]={7,19,24,13,8,17,11};`
- Πρώτα βρίσκει το μικρότερο στοιχείο το 7 και το ανταλλάσσει με την τιμή στην πρώτη θέση του πίνακα (στην θέση 0).
- Από τα υπόλοιπα στοιχεία βρίσκει το μικρότερο τους το 8 και το ανταλλάσσει με την τιμή στην δεύτερη θέση του πίνακα (στην θέση 1), ενώ στην θέση που ήταν το 8 (4η) βάζει το 19 που ήταν στην 2η θέση *(swap values)*, κ.ό.κ.

	0	1	2	3	4	5	6
I=0	7	19	24	13	8	17	11
I=1	7	8	24	13	19	17	11
I=2	7	8	11	13	19	17	24
I=3	7	8	11	13	19	17	24
I=4	7	8	11	13	17	19	24
I=5	7	8	11	13	17	19	24

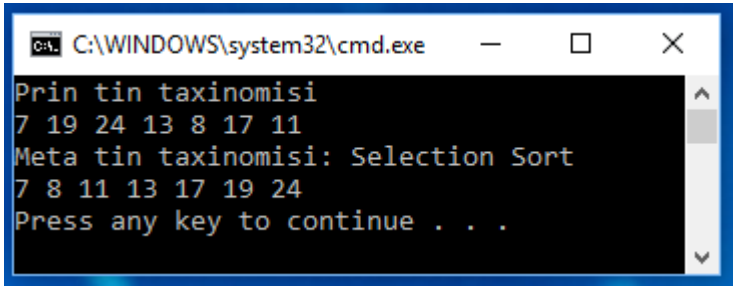
Selection Sort *(3/4)*

```
class SelectionSort {
 public static void main(String[] args) {
    int nums[] = {7,19,24,13,8,17,11};
   System.out.println("Prin tin taxinomisi");
   for (int i = 0; i < nums.length; i++)
      System.out.print(nums[i]+" ");
   System.out.println();
   SelSort(nums);
   System.out.println("Meta tin taxinomisi: Selection Sort");
   for (int i = 0; i < nums.length; i++)
      System.out.print(nums[i]+" ");
      System.out.println(); }
```

Selection Sort *(4/4)*

```
public static void SelSort(int[] A) {
 int pos, temp;
 for (int i = 0; i < A.length-1; i++){
  // thesi tis min timis
  pos=i;
  for (int k=i+1; k<A.length; k++)
    if (A[k] < A[pos]) pos=k;
     //swap tis times "pos" kai "i"
   temp = A[pos];
   A[pos] = A[i];
   A[i] = temp; } } }
```

Το εσωτερικό loop βρίσκει το μικρότερο, το αποθηκεύει στην temp και την θέση του στην pos. Στην τελευταία εντολή του εξωτερικού loop γίνεται η ανταλλαγή των στοιχείων. Με την 1η επανάληψη του εξωτερικού loop το μικρότερο στοιχείο βρίσκεται στην 1η θέση, με την 2η το δεύτερο μικρότερο στην 2η θέση, κ.ο.κ.



Σύγκριση πολυπλοκότητας Insertion Sort και Selection Sort

Θα χρησιμοποιήσουμε δύο κριτήρια για να συγκρίνουμε τους δύο αλγορίθμους *(και οι δύο με χρονική πολυπλοκότητα* $O(n^2)$*)*:

(1) τα **βήματα** που απαιτούνται για την ανεύρεση ενός στοιχείου και

(2) οι **ανταλλαγές** των στοιχείων (swaps)

Insertion Sort:

- Στην καλύτερη περίπτωση ο πίνακας είναι ήδη ταξινομημένος, οπότε η χρονική πολυπλοκότητα είναι της τάξης **O(n) - βημάτων**. Σ' αυτή την περίπτωση ο αλγόριθμος βγαίνει πιο γρήγορα από τον δεύτερο βρόχο.

Selection Sort:

- Απαιτούνται πάντα **$O(n^2)$ – βήματα** (ο αλγόριθμος δεν βγαίνει γρήγορα από τους βρόχους), έτσι η καλύτερη περίπτωση και η χειρότερη είναι ίδιες, ως προς την χρονική πολυπλοκότητα.
- Ως προς τις **ανταλλαγές** των στοιχείων η Selection *(πιο αποδοτική)* απαιτεί **O(n)**, ενώ η Insertion **$O(n^2)$,** στη χειρότερη περίπτωση (ήδη ταξινομημένος πίνακας σε φθίνουσα σειρά).

Άσκηση – 1η

Να γίνει το πρόγραμμα αναζήτησης και εμφάνισης των στοιχείων ενός υπάλληλου σε ένα πίνακα **N** - αντικειμένων τύπου Employee. Ορίστε την κλάση Employee με private πεδία: (1) Όνομα, String, (2) Επώνυμο, String, (3) Κωδικός Υπαλ, int και (4) Μισθός, double.

Το πρόγραμμα θα εκτελεί (1) σειριακή αναζήτηση ως προς το Επώνυμο και (2) δυαδική αναζήτηση ως προς τον Κωδικό Υπαλ. (σε ταξινομημένο ως προς τον Κωδικό-πίνακα).

Οι εργασίες θα υλοποιούνται από ένα **μενού επιλογών**.

Οι εργασίες θα καλούνται κατάλληλα, δηλ. δεν θα μπορείς να κάνεις δυαδική αναζήτηση, αν δεν έχει **εκτελεστεί πρώτα η ταξινόμηση** ως προς τον κωδικό (όποια ταξινόμηση θέλετε εσείς). Θα εμφανίζονται τα κατάλληλα μηνύματα που θα καθοδηγούν τις εργασίες.

Για την σειριακή αναζήτηση: Το πρόγραμμα θα ζητά το Επώνυμο του υπάλληλου που αναζητάμε. Αν το βρει θα εμφανίσει όλα τα στοιχεία του, διαφορετικά το μήνυμα “Den yparxei o Ypallilos ”. **Για την δυαδική αναζήτηση:** Το πρόγραμμα θα ζητά τον κωδικό του και θα υλοποιεί τα ίδια με την σειριακή αναζήτηση (εμφάνιση στοιχείων).

Αναδρομικές Τεχνικές Ταξινόμησης *(Recursive Sorting)*

Οι σημαντικότερες αναδρομικές τεχνικές είναι η γρήγορη ταξινόμηση *(quick sort)* και η ταξινόμηση με συγχώνευση *(merge sort)* που ακολουθούν την διαδικασία **διαίρει και βασίλευε** *(divide and conquer)*, δηλ. με **αναδρομικό αλγόριθμο** το πρόβλημα μοιράζεται σε δύο μέρη τα οποία **ταξινομούνται ξεχωριστά** *(partitioning)* και στο τέλος οι λύσεις συνδυάζονται.

- Στη quick sort δεν χρειάζεται βοηθητικός πίνακας (τα 2 μέρη ταξινομούνται επιτόπου - στον ίδιο πίνακα), ενώ στη merge sort απαιτείται.
- Η ταξινόμηση των δύο τμημάτων γίνεται με την βοήθεια μιας τιμής **άξονα *(pivot)*** που επιλέγουμε (συνήθως η μεσαία του πίνακα).
- Η Quicksort είναι η ταχύτερη τεχνική ταξινόμησης και υλοποιείται στην Java με την μέθοδο **Arrays.sort().** Η υλοποίησή της στην Java είναι διαφορετική από την υλοποίηση που θα διδαχτείτε *(2 pivot)*, και δοκιμασμένη (να την προτιμάτε). ***Προσοχή στους αλγορίθμους ταξινόμισης,*** γιατί κάποιοι δεν υλοποιούνται σωστά (για παράδειγμα υλοποιούνται σωστά μόνο για μονό ή ζυγό πλήθος αριθμών, κλπ.)

Γρήγορη Ταξινόμηση *(Quick Sort) (1/12)*

Γρήγορη Ταξινόμηση *(Quick Sort)*

- Αναπτύχθηκε από τον C.A.R Hoare το 1962 και παρουσιάζεται παρακάτω

Συνοπτικά εκτελούμε **αναδρομικά:**

1) διαλέγουμε ένα στοιχείο p (π.χ. πρώτη ή κεντρική τιμή στον πίνακα) που ονομάζουμε άξονα – ***pivot***.

2) χωρίζουμε τον πίνακα σε δύο μέρη ***(partitioning)*** Q1 και Q2, όπου το Q1 θα περιέχει όλα τα στοιχεία του πίνακα που είναι μικρότερα ή ίσα από τον pivot, και το Q2 θα περιέχει όλα τα στοιχεία που είναι μεγαλύτερα από τον pivot.

Γρήγορη Ταξινόμηση *(Quick Sort) (2/12)*

3) καλούμε **αναδρομικά τον αλγόριθμο QuickSort** στο Q1, και βρίσκουμε λύση A1, και στο Q2 και βρίσκουμε λύση A2.

4) τέλος η λύση θα είναι ο ταξινομημένος πίνακας [A1, p, A2].

Παράδειγμα:

[63, 21, 7, 33, 52, **(31)**, 9, 11, 20, 95, 74]

Διαλέγουμε σαν pivot το 31, και τοποθετούμε:

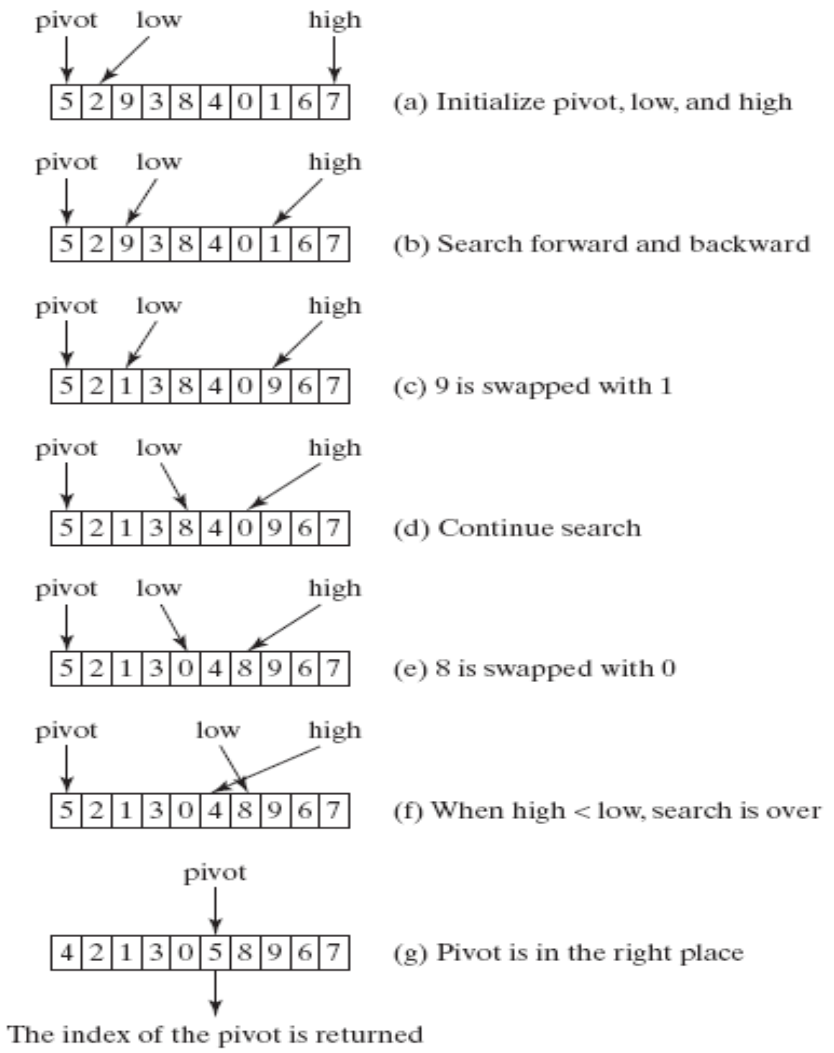
<31 **(31)** **>=31**

Quicksort **Quicksort**

[7,9,11,20,21] **31** **[33,52,63,74,95]**

Γρήγορη Ταξινόμηση *(Quick Sort) (3/12)*

- Κάθε **διαχωρισμός** του πίνακα ***(partition)*** τοποθετεί τον pivot στη σωστή θέση. Η θέση του pivot επιστρέφεται από την μέθοδο partition.



Εδώ σαν pivot διαλέγουμε το 1ο στοιχείο. Προσέξτε τις κινήσεις των δεικτών low και high και τις αλλαγές τιμών (swapping). Όταν *high* < *low*, τότε η αναζήτηση τελειώνει, ο *pivot* τοποθετείται στην σωστή θέση και η νέα θέση του επιστρέφεται.

Παράδειγμα από το βιβλίο του D. Liang (Java Programing, 10th edition)

Γρήγορη Ταξινόμηση *(Quick Sort) (4/12)*

Μια υλοποίηση αλγορίθμου QuickSort (από το βιβλίο του D. Liang)

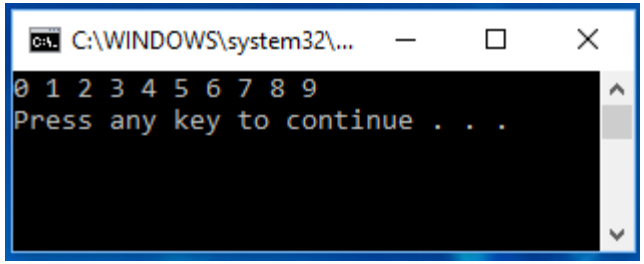
```
public class QuickSortLiang {
 public static void quickSort(int[] list) {
 quickSort(list, 0, list.length - 1);
}
public static void quickSort(int[] list, int first, int last) {
 if (last > first) {
 int pivotIndex = partition(list, first, last);
 quickSort(list, first, pivotIndex - 1);
 quickSort(list, pivotIndex + 1, last);
}}

/** Partition the array list[first..last] */
public static int partition(int[] list, int first, int last) {
 int pivot = list[first];   //Choose the first element as the pivot
 int low = first + 1;       //Index for forward search
 int high = last;           //Index for backward search
```

Γρήγορη Ταξινόμηση *(Quick Sort) (5/12)*

```
while (high > low) {
  // Search forward from left
  while (low <= high && list[low] <= pivot)
    low++;  //ayxisi toy aristerou deikti

  // Search backward from right
  while (low <= high && list[high] > pivot)
     high--; //meiosi toy dexiou deikti

 // Swap two elements in the list
 if (high > low) {
   int temp = list[high];
   list[high] = list[low];
   list[low] = temp;
  }
 }
```

Γρήγορη Ταξινόμηση *(Quick Sort) (6/12)*

```
while (high > first && list[high] >= pivot)
  high--;
 // Swap pivot with list[high]
 if (pivot > list[high]) {
  list[first] = list[high];
  list[high] = pivot;
  return high;}
 else {return first;}}

 /** A test method */
 public static void main(String[] args) {
  int[] list = {5,2,9,3,8,4,0,1,6,7};
  quickSort(list);
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
 }
}
```



Γρήγορη Ταξινόμηση *(Quick Sort) (7/12)*

Μια άλλη υλοποίηση αλγορίθμου QuickSort (χρήση αντικειμένου)

```
import java.util.Arrays;
public class QuickSortDemo{
   public static void main(String args[]) {
      int[] nums = {6, 5, 3, 1, 8, 7, 2, 4};
      System.out.println("Ataxinomitos pinakas");
      System.out.println(Arrays.toString(nums));
      System.out.println();
      QuickSort algorithm = new QuickSort();
      // taxinomisi toy pinaka me ton algorithmo quicksort
      algorithm.sort(nums);

       // emfanisi toy taxinomimenoy pinaka
       System.out.println();
       System.out.println("Taxinomimenos pinakas me Quick Sort ");
        System.out.println(Arrays.toString(nums));
        System.out.println(); } }
```

Γρήγορη Ταξινόμηση *(Quick Sort) (8/12)*

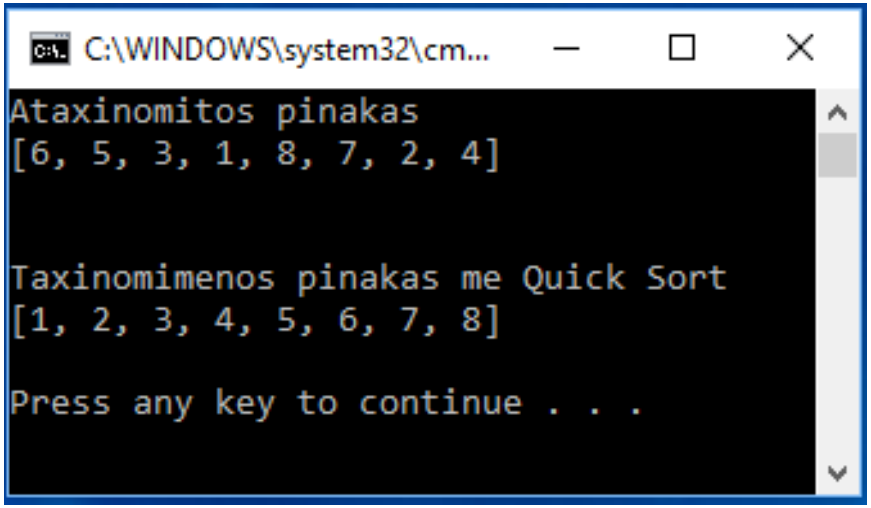
```
class QuickSort {
   private int input[];
   private int length;
   public void sort(int[] numbers) {
       if (numbers == null || numbers.length == 0) {
           return; }
       this.input = numbers;
       length = numbers.length;
       quickSort(0, length - 1); }

   private void quickSort(int low, int high) {
       int i = low;
       int j = high;
       int x=0;
       x=low + (high - low) / 2;
       int pivot = input[low + (high - low) / 2];  //pivot sti mesi
```

Γρήγορη Ταξινόμηση *(Quick Sort) (9/12)*

```
// Diairesi se 2 ypo-pinakes - partitioning
while (i <= j) {
 /* Se kathe epanalipsi tha vriskoume apo ta aristera ena arithmo pou
  * tha einai megalyteros toy pivot kai ena arithmo apo ta dexia pou
  * tha einai mikroteros toy pivot. Otan teleiosei o elehos tha
  * ginetai h antimetathesi(swap) ton timon */

 //prohoroyme apo ta aristera pros dexia mexri na xreiastei ena swap
 while (input[i] < pivot) {i++;}
 //prohoroyme apo ta dexia pros ta aristera mexri na xreiastei ena swap
 while (input[j] > pivot) {j--;}
 if (i <= j) {
    //kane to swap
     swap(i, j);
    // metakinise tous deiktes stin epomeni thesi kai stis 2 kateythinseis
     i++;
     j--;
 }}
```

Γρήγορη Ταξινόμηση *(Quick Sort) (10/12)*

```
  // anadromiki klisi tis quickSort()
  if (low < j) {quickSort(low, j); }
  if (i < high) {quickSort(i, high);}
 }

  // H methodos antikatastasis
   private void swap(int i, int j) {
        int temp = input[i];
        input[i] = input[j];
        input[j] = temp;
   }
}
```



Γρήγορη Ταξινόμηση *(Quick Sort) (11/12)*

Αταξινόμητος	**[6, 5, 3, (1), 8, 7, 2, 4]**
Θέση=3, pivot=**1**, i=0, j=3	[**1**, 5, 3, **6**, 8, 7, 2, 4] (i=1, j=0)
Θέση=4, pivot=**8**, i=4, j=7	[1, 5, 3, 6, **4**, 7, 2, **8**] (i=7, j=6)
Θέση=3, pivot=**6**, i=3, j=6	[1, 5, 3, **2**, 4, 7, **6**, 8] (i=5, j=4)
Θέση=2, pivot=**3**, i=1, j=3	[1, **2**, 3, **5**, 4, 7, 6, 8] (i=2, j=2)
i=2, j=2	[1, 2, 3, 5, 4, 7, 6, 8] (i=3, j=1)
Θέση=3, pivot=**5**, i=3, j=4	[1, 2, 3, **4**, **5**, 7, 6, 8] (i=4, j=3)
Quicksort – 2ο τμήμα	[1, 2, 3, 4, 5, 7, 6, 8]
Quicksort – 1ο τμήμα	[1, 2, 3, 4, 5, 7, 6, 8]
Θέση=5, pivot=**7**, i=5, j=6	[1, 2, 3, 4, 5, **6**, **7**, 8] (i=6, j=5)
Quicksort – 2ο τμήμα	[1, 2, 3, 4, 5, 6, 7, 8]
Quicksort – 1ο τμήμα	[1, 2, 3, 4, 5, 6, 7, 8]
Quicksort – 2ο τμήμα	[1, 2, 3, 4, 5, 6, 7, 8]
Ταξινομημένος	**[1, 2, 3, 4, 5, 6, 7, 8]**

Γρήγορη Ταξινόμηση *(Quick Sort) (12/12)*

Ανάλυση Χρονικής Πολυπλοκότητας

- Ο ορισμός του pivot απαιτεί χρόνο O(1) και η διαδικασία **partition** ενός πίνακα *n*-στοιχείων εκτελείται σε χρόνο **O(n)** σε όλα τα επίπεδα αναδρομής.
- Η εκτέλεση της QuickSort απαιτεί στην χειρότερη περίπτωση χρόνο **$O(n^2)$** και στην καλύτερη περίπτωση χρόνο **O(n*log (n))**.

Χειρότερη περίπτωση:

Σ' αυτή την περίπτωση ο pivot διαιρεί σε ένα μόνο μεγάλο υπο-πίνακα τον αρχικό με στοιχεία n-1. Έτσι ο αλγόριθμος απαιτεί χρόνο: (n - 1) + (n - 2) + g + 2 + 1 = **$O(n^2)$**.

Καλύτερη περίπτωση:

Σ' αυτή την περίπτωση ο pivot διαιρεί τον πίνακα σε 2-τμήματα, του ίδιου μεγέθους περίπου. Έτσι αν *T(n)* συμβολίζει τον χρόνο ταξινόμησης με την quick sort ενός πίνακα *n*- στοιχείων, τότε:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n. = O(n \log n)$$

Αναδρομική QuicSort στους 2 υπο-πίνακες

Χρόνος διαχωρισμού (partition)

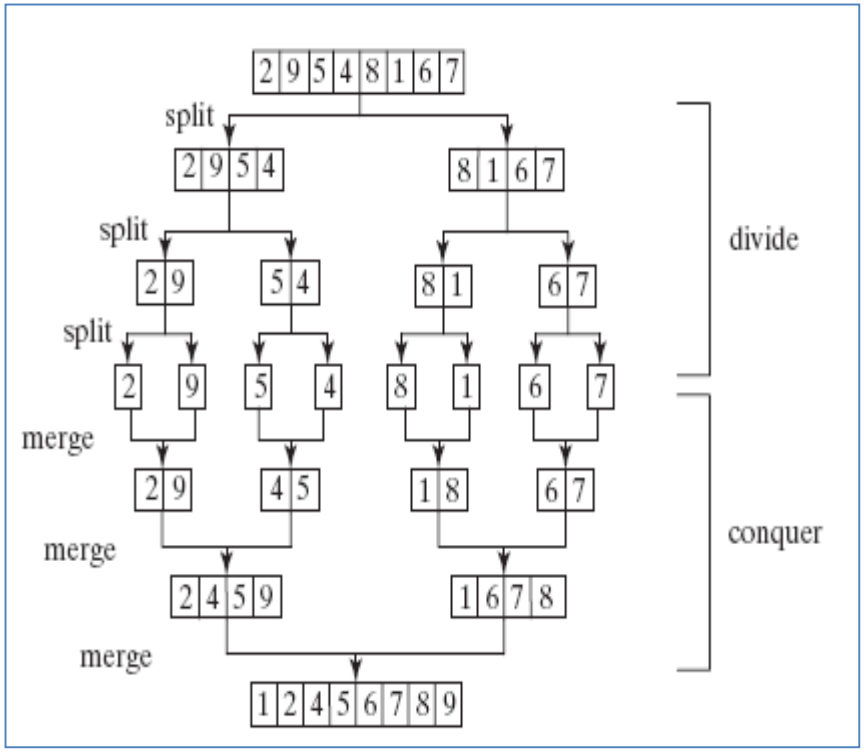
Ταξινόμηση με Συγχώνευση *(Merge Sort)* *(1/7)*

Ο αλγόριθμος της ταξινόμησης με συγχώνευση ακολουθεί επίσης την διαδικασία του **διαίρει και βασίλευε** *(divide and conquer)*. Δηλ., είναι μια αναδρομική διαδικασία όπου ο πίνακας **μοιράζεται** σε δύο μέρη τα οποία ταξινομούνται ξεχωριστά, και μετά αφού έχει γίνει η ταξινόμηση τα δύο μέρη **συγχωνεύονται**.

Ο αλγόριθμος της ταξινόμησης:

- **Διαιρεί** αναδρομικά τον πίνακα σε δύο μέρη μέχρι να φτάσουμε σε πίνακες μεγέθους ενός στοιχείου.
- **Συγχωνεύει,** τους επιμέρους πίνακες που ταξινομούνται αναδρομικά χρησιμοποιώντας ένα βοηθητικό πίνακα.

Παράδειγμα από το βιβλίο του D. Liang:



Ταξινόμηση με Συγχώνευση *(Merge Sort)* (2/7)

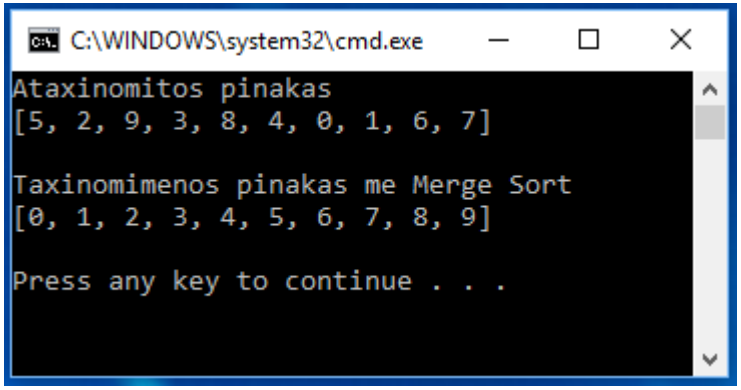
```
import java.util.Arrays;
public class MergeSort {
 /** H methodos taxinomei tous arithmous tou pinaka A */
 public static void mergeSort(int[] A) {
  if (A.length > 1) {
   // anadromiki Merge sort tou 1ou misou tou pinaka(copy)
   int[] firstHalf = new int[A.length/2];//dhmioyrgia toy 1ou temp array
     System.arraycopy(A, 0, firstHalf, 0, A.length/2);
     mergeSort(firstHalf);

     // anadromiki Merge sort tou 2ou misou tou pinaka(copy)
     int secondHalfLength = A.length - A.length/2; //2o temp array
     int[] secondHalf = new int[secondHalfLength];
     System.arraycopy(A, A.length/2, secondHalf, 0, secondHalfLength);

     mergeSort(secondHalf);

    //Merge ton taxinomimenon 1ou-kai 2ou pinaka se ena temp-array
    merge(firstHalf, secondHalf, A);
 } }
```

Ταξινόμηση με Συγχώνευση *(Merge Sort)* *(3/7)*

```
/* Merge tous 2 taxinomimenous pinakes list1 kai list2
* H methodos epanaliptika sygkrinei ta stoixeia ton list1 kai list2
* kai topothetei to mikrotero ston temp.
public static void merge(int[] list1, int[] list2, int[] temp) {
  int current1 = 0;  //trehon deiktis sto list1
  int current2 = 0;  //trehon deiktis sto list2
  int current3 = 0;  //trehon deiktis sto temp

while (current1 < list1.length && current2 < list2.length) {
  if (list1[current1] < list2[current2])
    /*an to mikrotero stoixeio vrethike sto list1 ayxise ton
    * current1 kata 1, diaforetika ayxise ton current2 kata 1 */
    temp[current3++] = list1[current1++];
  else
    temp[current3++] = list2[current2++];
 }
```

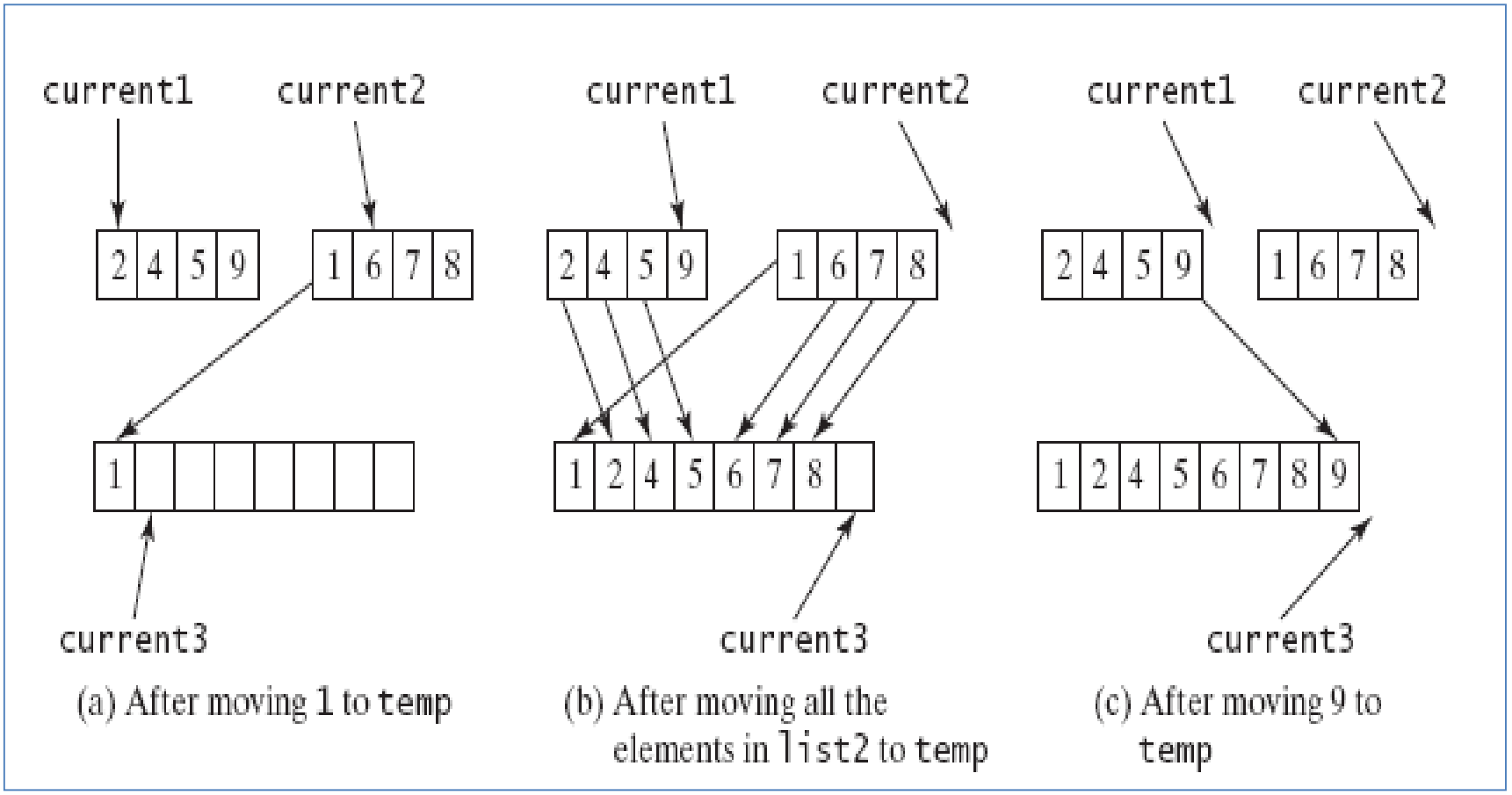
Ταξινόμηση με Συγχώνευση *(Merge Sort)* *(4/7)*

```
/* Ola ta stoixeia metaferontai ston temp
 * akomi kai ayta poy den metakinithikan
 * apo ta list1 kai list2 */
while (current1 < list1.length)
   temp[current3++] = list1[current1++];
while (current2 < list2.length)
   temp[current3++] = list2[current2++];
 }

public static void main(String[] args) {
   int[] nums = {5,2,9,3,8,4,0,1,6,7};
   System.out.println("Ataxinomitos pinakas");
   System.out.println(Arrays.toString(nums));
   System.out.println();
   mergeSort(nums);
   System.out.println("Taxinomimenos pinakas me Merge Sort ");
   System.out.println(Arrays.toString(nums));
   System.out.println(); } }
```



Ταξινόμηση με Συγχώνευση *(Merge Sort)* *(5/7)*

Οι δύο ταξινομημένοι υπο-πίνακες συγχωνεύονται σε ένα ταξινομημένο πίνακα.



Ταξινόμηση με Συγχώνευση *(Merge Sort)* *(6/7)*

Ανάλυση της χρονικής πολυπλοκότητας της Merge Sort *(1/2)*

- Αν *T(n)* συμβολίζει τον χρόνο που απαιτείται για την ταξινόμησης, με την Merge Sort, ενός πίνακα *n*-στοιχείων (όπου n μια δύναμη του 2), τότε ο αλγόριθμος διαιρεί τον πίνακα σε 2 υπο-πίνακες, ***ταξινομεί αναδρομικά τους 2 υπο-πίνακες*** και τους συγχωνεύει σε ένα ταξινομημένο πίνακα, άρα:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \text{Χρόνος Συγχώνευσης}$$

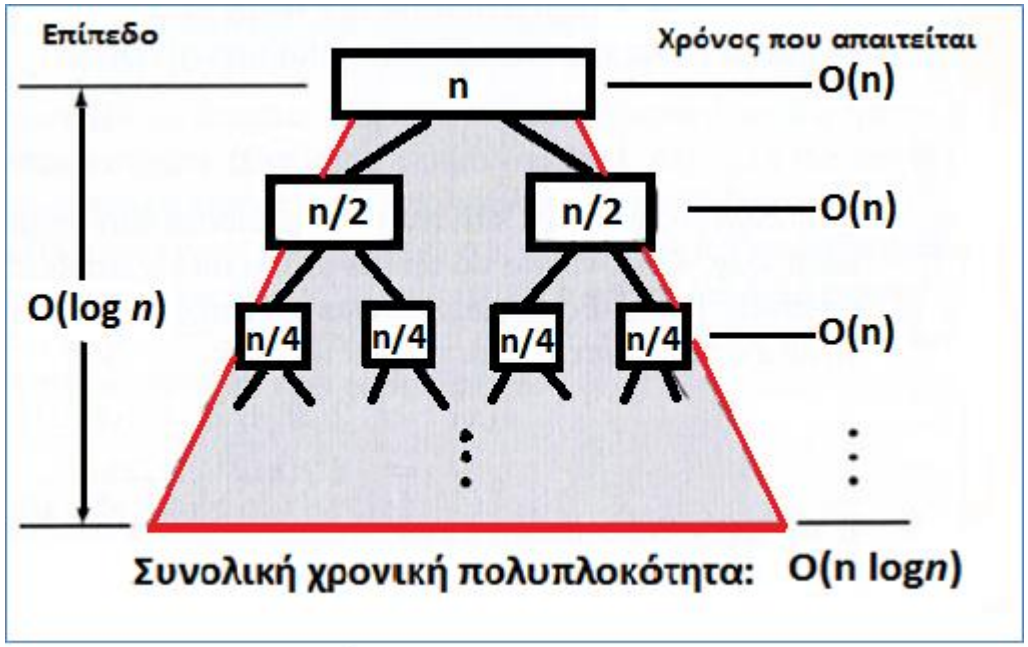
Το 1ο μισό Το 2ο μισό

- Για την ***συγχώνευση των δύο υπο-πινάκων*** απαιτούνται *n-1 συγκρίσεις* των στοιχείων των δύο υπο-πινάκων, και *n-κινήσεις* για την μετακίνηση των στοιχείων στον προσωρινό πίνακα. Έτσι ο συνολικός χρόνος είναι *2n-1*, και η χρονική πολυπλοκότητα:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n - 1 = O(n \log n)$$

Ταξινόμηση με Συγχώνευση *(Merge Sort)* *(7/7)*

Ανάλυση της χρονικής πολυπλοκότητας της Merge Sort *(2/2)*



- ο πίνακας μοιράζεται σε:

n, n/2, n/4, …, 2, 1

- γραμμικός χρόνος O(n) σε κάθε επίπεδο της συγχώνευσης.
- χρειάζεται *βοηθητικός πίνακας* που μπορεί να είναι ο ίδιος σε όλες τις αναδρομικές κλήσεις.
- σε επίπεδο ενός στοιχείου σταματά η αναδρομή και αρχίζει η συγχώνευση.

- **Με συνολική χρονική πολυπλοκότητα O(n log *n*), η Merge Sort είναι καλύτερη από τις selection sort, insertion sort, και bubble sort που έχουν χρονική πολυπλοκότητα της τάξης $O(n^2)$. Μειονέκτημα η χρήση βοηθητικού πίνακα.**
- **Η μέθοδος sort της κλάσης java.util.Arrays υλοποιείται με τον αλγόριθμο Merge Sort.**

Σύγκριση Πολυπλοκότητας Αλγορίθμων Ταξινόμησης

Ταξινόμηση	Μέση Περίπτωση	Καλύτερη Περίπτωση	Χειρότερη Περίπτωση
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Quick Sort	$O(n*\log(n))$	$O(n*\log(n))$	$O(n^2)$
Merge Sort	$O(n*\log(n))$	$O(n*\log(n))$	$O(n*\log(n))$

Όπου:

$$O(1) < O(\log n) < O(\log^2 n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^k) < O(1.1^n) < O(2^n) < O(3^n) < O(n!) < O(n^n)$$