

## Κεφάλαιο

# 16

## Αναζήτηση και ταξινόμηση



## Αναζήτηση και ταξινόμηση

Η αναζήτηση και η ταξινόμηση αποτελούν τα πλέον "προκλητικά" θέματα στη διαχείριση δεδομένων. Η αναζήτηση αφορά στον εντοπισμό μιας συγκεκριμένης τιμής σε ένα σύνολο δεδομένων, ενώ η ταξινόμηση αφορά στη διάταξη ενός συνόλου δεδομένων σε αύξουσα ή φθίνουσα σειρά.

Ένα σύνολο δεδομένων σε έναν  $H/Y$ , είναι αποθηκευμένο με έναν συγκεκριμένο τρόπο και καλείται δομή δεδομένων<sup>9</sup>. Η πιο απλή δομή δεδομένων είναι ένας πίνακας.

Η δομή δεδομένων μπορεί να είναι εσωτερική (αποθηκευμένη στην κεντρική μνήμη του  $H/Y$ ) —όπως είναι οι πίνακες, οι συνδεδεμένες λίστες, τα δυαδικά δένδρα κ.λπ. ή εξωτερική, οπότε αναφερόμαστε σε δεδομένα αποθηκευμένα σε αρχεία.

Είτε πρόκειται για εσωτερικές, είτε για εξωτερικές δομές, οι αλγόριθμοι που χρησιμοποιούνται για την αναζήτηση και την ταξινόμηση είναι ίδιοι. Αυτό που διαφέρει είναι ο τρόπος υλοποίησής τους για κάθε περίπτωση.

Βασικός στόχος κάθε αλγόριθμου αναζήτησης και ταξινόμησης είναι η ταχύτητα. Οι διάφοροι αλγόριθμοι που έχουν σχεδιαστεί διαφοροποιούνται μόνο ως προς την απόδοσή τους. Συνήθως οι πιο απλοί στη σύλληψη αλλά και στην υλοποίηση είναι οι λιγότερο αποδοτικοί. Στο κεφάλαιο αυτό παρουσιάζονται αναλυτικά οι πιο "κλασικές" μέθοδοι αναζήτησης και ταξινόμησης.

Σε όλες τις περιπτώσεις τα δεδομένα είναι αποθηκευμένα σε πίνακες. Οι ίδιες όμως μέθοδοι με μικρές παραλλαγές μπορούν να χρησιμοποιηθούν και σε δεδομένα αποθηκευμένα σε αρχεία σε περιφερειακές μονάδες αποθήκευσης.

---

<sup>9</sup> Στις δομές δεδομένων θα αναφερθούμε διεξοδικά στο Κεφάλαιο 18.

## Σειριακή αναζήτηση

Η σειριακή αναζήτηση αποτελεί τον απλούστερο τρόπο αναζήτησης τιμών μέσα σε μια δομή δεδομένων. Η λογική του αλγόριθμου της σειριακής αναζήτησης είναι ότι ελέγχει ένα-ένα τα δεδομένα με τη σειρά (από την αρχή προς το τέλος) μέχρι να εντοπίσει (ή να μην εντοπίσει) την τιμή που αναζητούμε.

Ο αλγόριθμος χρησιμοποιεί μια βοηθητική μεταβλητή (π.χ. τη *found*), στην οποία δίνει μια αρχική τιμή 0. Μόλις εντοπίσει το δεδομένο που αναζητάμε, τότε καταχωρίζει στη μεταβλητή το 1 και σταματάει την αναζήτηση.

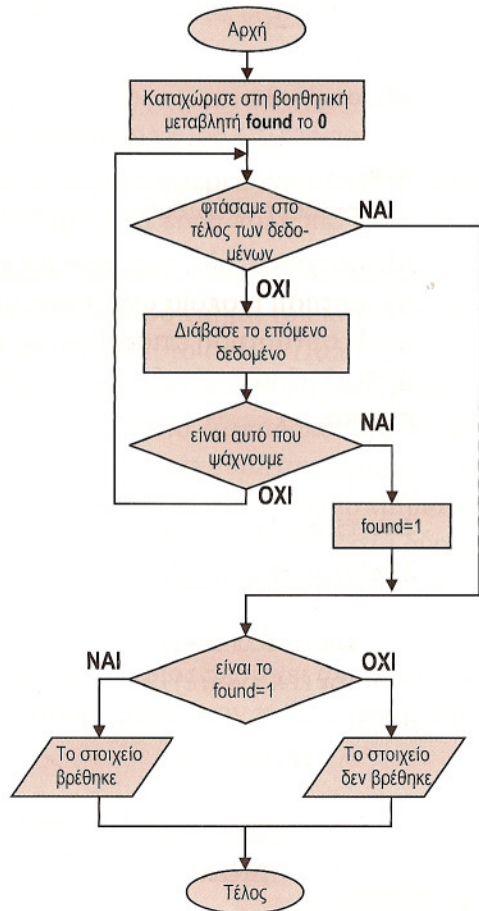
Στο τέλος ελέγχει την τιμή της βοηθητικής μεταβλητής *found*. Αν είναι 1, αυτό σημαίνει ότι το στοιχείο έχει εντοπιστεί, διαφορετικά ότι το στοιχείο δεν εντοπίστηκε μέσα στα δεδομένα που ψάχνουμε.

Το επόμενο τμήμα κώδικα εντοπίζει μέσα σε έναν πίνακα 100 θέσεων έναν αριθμό που ζητάει από το πληκτρολόγιο.

```
printf("Δώσε αριθμό για αναζήτηση:");
scanf("%d", &ar);
found=0;
for (i=0; i<100; i++)
{
    if (a[i]==ar)
```

Δίνεται στη *found* αρχική τιμή το 0.

Εάν βρεθεί η τιμή στον πίνακα, τίθεται η τιμή 1 στη *found* και διακόπτεται η αναζήτηση.





```

    {
        found=1;
        break;
    }
}
if(found==1)
    printf("Ο αριθμός υπάρχει στον πίνακα");
else
    printf("Ο αριθμός δεν υπάρχει στον πίνακα");

```

Στην περίπτωση που η found έχει πάρει τιμή 1 σημαίνει ότι ο αριθμός βρέθηκε στον πίνακα, διαφορετικά όχι.

Με ελάχιστες αλλαγές, το παραπάνω τμήμα κώδικα μπορεί να γραφεί σαν μια συνάρτηση η οποία αναζητάει μέσα σε έναν πίνακα **a**, **n** θέσεων έναν αριθμό **ar**. Η συνάρτηση επιστρέφει ως τιμή τη θέση (την τιμή του **i**) που εντόπισε τον αριθμό μέσα στον πίνακα **a** και τιμή -1 όταν ο αριθμός δεν υπάρχει μέσα στον πίνακα.

```

int find_it(a,n,ar)
int a[],n,ar;
{
    int i,found=-1;
    for(i=0;i<n;i++)
    {
        if(a[i]==ar)
        {
            found=i;
            break;
        }
    }
    return found;
}

```

Αν βρεθεί η τιμή στον πίνακα, καταχωρίζεται στη found η τιμή του **i** και διακόπτεται η αναζήτηση.

Η συνάρτηση επιστρέφει ως τιμή τη θέση που βρήκε τον αριθμό, διαφορετικά επιστρέφει τιμή -1 (την αρχική τιμή της found).

## Δυαδική αναζήτηση (binary search)

Η **δυαδική αναζήτηση** αποτελεί τον πιο αποδοτικό τρόπο αναζήτησης, ο οποίος όμως προϋποθέτει τα δεδομένα να είναι ταξινομημένα είτε κατά αύξουσα, είτε κατά φθίνουσα σειρά.

Φανταστείτε να ψάχνουμε στον τηλεφωνικό κατάλογο για κάποιο όνομα. Δεν θα ήταν καθόλου έξυπνο να αρχίσουμε από την αρχή και να ψάχνουμε ένα-ένα τα ονόματα μέχρι να εντοπίσουμε αυτό που μας ενδιαφέρει. Αντίθετα, αν ακολουθήσουμε την επόμενη τεχνική, ο εντοπισμός θα είναι πολύ πιο γρήγορος:

Ανοίγουμε τον κατάλογο περίπου στο μέσον, βλέπουμε ένα όνομα και αποφασίζουμε αν αυτό που ψάχνουμε είναι στο πρώτο μισό τμήμα του καταλόγου ή στο δεύτερο. Αν π.χ. στη σελίδα που ανοίξαμε είναι το όνομα *Παπαδόπουλος* και εμείς ψάχνουμε το *Ταμβακέλλης*, προφανώς το όνομα που ψάχνουμε θα είναι στο δεύτερο κομμάτι του καταλόγου, μετά από το *Παπαδόπουλος*. Παίρνουμε τώρα το δεύτερο κομμάτι του καταλόγου και κάνουμε ακριβώς τα ίδια. Εντοπίζουμε δηλαδή μια σελίδα στο μέσον του και βλέπουμε σε ποιο τμήμα είναι το όνομα που ψάχνουμε κ.ο.κ. Αυτό γίνεται συνέχεια, κόβοντας στα δύο τα κομμάτια που ψάχνουμε, μέχρι να εντοπίσουμε το όνομα ή να καταλήξουμε ότι δεν υπάρχει στον κατάλογο.

Θεωρούμε τον επόμενο πίνακα **a**, 10 θέσεων, ο οποίος περιέχει ακέραιους αριθμούς με αύξουσα ταξινόμηση και αναζητούμε μέσα στον πίνακα τον αριθμό 135.

Πάντα η αναζήτηση γίνεται σε ένα τμήμα του πίνακα που ξεκινάει από τη θέση που δείχνει το **A** (Από) μέχρι τη θέση που δείχνει το **E** (Εως). Αρχικά τα **A** και **E** δείχνουν την πρώτη (0) και την τελευταία (9) του πίνακα αντίστοιχα.

Παίρνουμε την ενδιάμεση θέση μνήμης **M** στο μέσον

|      | Πρώτος έλεγχος | Δεύτερος έλεγχος | Τρίτος έλεγχος | Τέταρτος έλεγχος |
|------|----------------|------------------|----------------|------------------|
| a[0] | 7 ← A          | 7                | 7              | 7                |
| a[1] | 12             | 12               | 12             | 12               |
| a[2] | 20             | 20               | 20             | 20               |
| a[3] | 48             | 48               | 48             | 48               |
| a[4] | 69 ← M         | 69               | 69             | 69               |
| a[5] | 100            | 100 ← A          | 100 ← A ← M    | 100              |
| a[6] | 135            | 135              | 135 ← E        | 135 ← E ← A ← M  |
| a[7] | 180            | 180 ← M          | 180            | 180              |
| a[8] | 196            | 196              | 196            | 196              |
| a[9] | 205 ← E        | 205 ← E          | 205            | 205              |

Αναζήτηση του αριθμού 135

→ Η τιμή εντοπίστηκε στη θέση M

του τμήματος από το A μέχρι το E ( $M=(A+E)/2$ ). Συγκρίνουμε τον αριθμό που ψάχνουμε αν είναι μικρότερος, μεγαλύτερος, ή ίσος με αυτόν της θέσης M.

- Αν είναι ίσος, τον εντοπίσαμε και η διαδικασία τελειώνει.
- Αν είναι μεγαλύτερος, αυτό σημαίνει ότι αν υπάρχει, θα βρίσκεται στο τμήμα από το M και κάτω (μέχρι το E). Θέτουμε στο A την τιμή M+1 (την επόμενη θέση του M) και συνεχίζουμε την ίδια διαδικασία.
- Αν είναι μικρότερος, αυτό σημαίνει ότι αν υπάρχει, θα βρίσκεται στο τμήμα από το M και πάνω (μέχρι το A). Θέτουμε στο E την τιμή M-1 (την προηγούμενη θέση του M) και συνεχίζουμε την ίδια διαδικασία.

Αυτό επαναλαμβάνεται μέχρι να εντοπιστεί ο αριθμός.

Στην περίπτωση που ο ζητούμενος αριθμός δεν υπάρχει στον πίνακα, θα φτάσουμε σε κάποιο σημείο που το A θα ξεπεράσει το E και τότε η διαδικασία σταματάει χωρίς να εντοπιστεί ο αριθμός.

Στην περίπτωση π.χ. αναζήτησης του αριθμού 137, όταν φτάσουμε στον τέταρτο έλεγχο ( $A=E=M$ ) συγκρίνουμε τον αριθμό που ψάχνουμε αν είναι μικρότερος, μεγαλύτερος, ή ίσος με αυτόν της θέσης M. Το 137 είναι μεγαλύτερο από το 135, οπότε θέτουμε στο A την τιμή M+1 (την επόμενη θέση του M). Τώρα όμως το A ξεπέρασε το E και η διαδικασία πρέπει να τερματιστεί με ταυτόχρονη απάντηση ότι ο ζητούμενος αριθμός (137) δεν υπάρχει στον πίνακα.

Η επόμενη συνάρτηση υλοποιεί τη μέθοδο της δυαδικής αναζήτησης και αναζητάει μέσα σε έναν πίνακα `a`, `n` θέσεων έναν αριθμό `ar`.

Η συνάρτηση επιστρέφει ως τιμή τη θέση που εντόπισε τον αριθμό μέσα στον πίνακα `a`. Αν ο αριθμός δεν υπάρχει μέσα στον πίνακα, η συνάρτηση επιστρέφει τιμή -1.

|      | Πρώτος έλεγχος | Δεύτερος έλεγχος | Τρίτος έλεγχος | Τέταρτος έλεγχος | Πέμπτος έλεγχος |
|------|----------------|------------------|----------------|------------------|-----------------|
| a[0] | 7 ← A          | 7                | 7              | 7                | 7               |
| a[1] | 12             | 12               | 12             | 12               | 12              |
| a[2] | 20             | 20               | 20             | 20               | 20              |
| a[3] | 48             | 48               | 48             | 48               | 48              |
| a[4] | 69 ← M         | 69               | 69             | 69               | 69              |
| a[5] | 100 ← A        | 100              | 100 ← A < M    | 100              | 100             |
| a[6] | 135            | 135              | 135 ← E        | 135 ← E < A < M  | 135 ← E < M     |
| a[7] | 180            | 180 ← M          | 180            | 180              | 180 ← A         |
| a[8] | 196            | 196              | 196            | 196              | 196             |
| a[9] | 205 ← E        | 205 ← E          | 205            | 205              | 205             |

Αναζήτηση του αριθμού 137

→ Εφόσον το A (Από) ξεπέρασε το E (Εως) ο ζητούμενος αριθμός δεν υπάρχει στον πίνακα



```
int binary(a,n,ar)
```

```
int a[],n,ar;
```

```
{
```

```
    int apo,eos,meson;
```

```
    apo=0;
```

```
    eos=n-1;
```

```
    while (apo<=eos)
```

```
    {
```

```
        meson=(apo+eos)/2;
```

```
        if (ar<a[meson])
```

```
            eos=meson-1;
```

```
        else if (ar>a[meson])
```

```
            apo=meson+1;
```

```
        else
```

```
            return meson;
```

```
    }
```

```
    return -1;
```

```
}
```

Η διαδικασία θα σταματήσει όταν το **apo** ξεπεράσει το **eos** (εκτός αν βέβαια προηγουμένως έχει εντοπιστεί η τιμή)

Υπολογισμός της θέσης μνήμης στο μέσον του τμήματος από το **apo** μέχρι το **eos**

Στην περίπτωση που ο ζητούμενος αριθμός είναι μεγαλύτερος από αυτόν στη θέση **meson**, αλλάζουμε την τιμή του **apo**, διαφορετικά αν είναι μικρότερος την τιμή του **eos**.

Η περίπτωση αυτή θα γίνει μόνον όταν το  $ar == a[meson]$ , οπότε ο ζητούμενος αριθμός εντοπίστηκε στη θέση **meson**

Η συνάρτηση θα επιστρέψει -1 στην περίπτωση που δεν εντοπιστεί ο ζητούμενος αριθμός

Μια αναδρομική εκδοχή της προηγούμενης συνάρτησης για τη δυαδική αναζήτηση φαίνεται παρακάτω. Η παράμετρος **apo** προσδιορίζει την πρώτη και η **eos** την τελευταία θέση μνήμης του τμήματος του πίνακα **a**, στον οποίο θέλουμε να γίνει η αναζήτηση του αριθμού **ar**. Η συνάρτηση επιστρέφει ως τιμή τη θέση στην οποία εντόπισε τον αριθμό. Διαφορετικά, αν δεν τον εντοπίσει, επιστρέφει αρνητικό αριθμό.

```
int rbinary(a,apo,eos,ar)
```

```
int a[],apo,eos,ar;
```

```
{
```

```
    int meson;
```

```
    if (apo <= eos)
```

```
    {
```

```
        meson=(apo+eos)/2;
```

```
        if (ar<a[meson])
```

```
            return rbinary(a,apo,meson-1,ar);
```

```
        else if (ar>a[meson])
```

Αναδρομική κλήση της συνάρτησης για το τμήμα που ενδεχομένως να υπάρχει ο αριθμός που ψάχνουμε.

```

        return rbinary(a,meson+1,eos,ar);
    else
        return meson;
}
else
    return -(apo + 1);
}

```

Μη αναδρομική περίπτωση.

Μη αναδρομική περίπτωση.

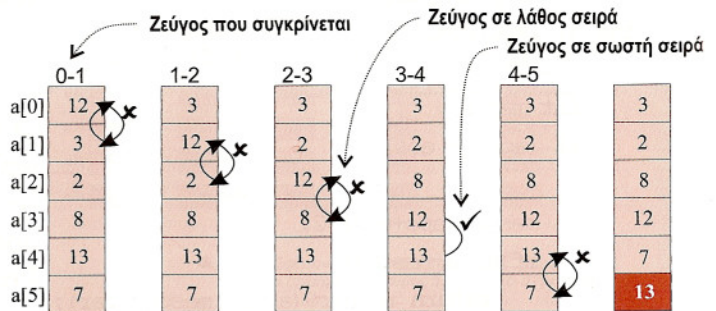
- ❏ Οι παραπάνω συναρτήσεις προϋποθέτουν ότι ο πίνακας **a** είναι ταξινομημένος με αύξουσα σειρά. Με ελάχιστες τροποποιήσεις, μπορούν να εφαρμοστούν και σε πίνακα με φθίνουσα ταξινόμηση. **Σε κάθε περίπτωση όμως, για να εφαρμοστεί η δυαδική αναζήτηση, τα δεδομένα πρέπει να είναι ταξινομημένα.**
- ❏ Στη δυαδική αναζήτηση, το μέγιστο πλήθος αναζητήσεων σε  $n$  στοιχεία είναι  $\log_2 n$ . Για παράδειγμα, σε 8 στοιχεία είναι 3 ( $\log_2 8=3$ ) ενώ σε 1024 στοιχεία είναι 10 ( $\log_2 1024=10$ ).

## Ταξινόμηση φυσαλίδας (bubble sort)

Η λογική στην οποία στηρίζεται η ταξινόμηση φυσαλίδας είναι η εξής:

Ελέγχουμε ανά ζεύγη τα γειτονικά κελιά ενός πίνακα (το κάθε κελί με το επόμενο του) και αν είναι σε διαφορετική σειρά, αντιμεταθέτουμε τα περιεχόμενα τους.

Μετά από το πρώτο πέρασμα, στο τέλος του πίνακα θα βρεθεί η μεγαλύτερη τιμή. Το επόμενο "πέρασμα" γίνεται για τις υπόλοιπες θέσεις του πίνακα (πλην της τελευταίας) και στο τέλος του θα βρεθεί στην προτελευταία θέση του πίνακα η επόμενη μεγαλύτερη τιμή. Κάθε φορά ο έλεγχος εφαρμόζεται σε ολοένα μικρότερο τμήμα του πίνακα με αποτέλεσμα η μεγαλύτερη κάθε φορά τιμή να μετακινείται στο





τέλος του τμήματος που ελέγχεται. Η διαδικασία θα σταματήσει μόλις το τμήμα που ελέγχεται αποτελείται από τις δύο πρώτες θέσεις του πίνακα και τότε πια ο πίνακας θα είναι ταξινομημένος.

Στο σχήμα της επόμενης σελίδα φαίνεται εποπτικά ολόκληρη η διαδικασία αύξουσας ταξινόμησης ενός πίνακα έξι θέσεων.

Η επόμενη συνάρτηση υλοποιεί τη μέθοδο "ταξινόμησης φυσαλίδας" (*bubblesort*) για την αύξουσα ταξινόμηση ενός πίνακα  $x$  μιας διάστασης με  $n$  θέσεις μνήμης:

```
void bubblesort(int x[], int n)
{
    int i, k, temp;
    for(i=0; i<n; i++)
    {
        for(k=0; k<n-i-1; k++)
        {
            if(x[k]>x[k+1])
            {
                temp=x[k];
                x[k]=x[k+1];
                x[k+1]=temp;
            }
        }
    }
}
```

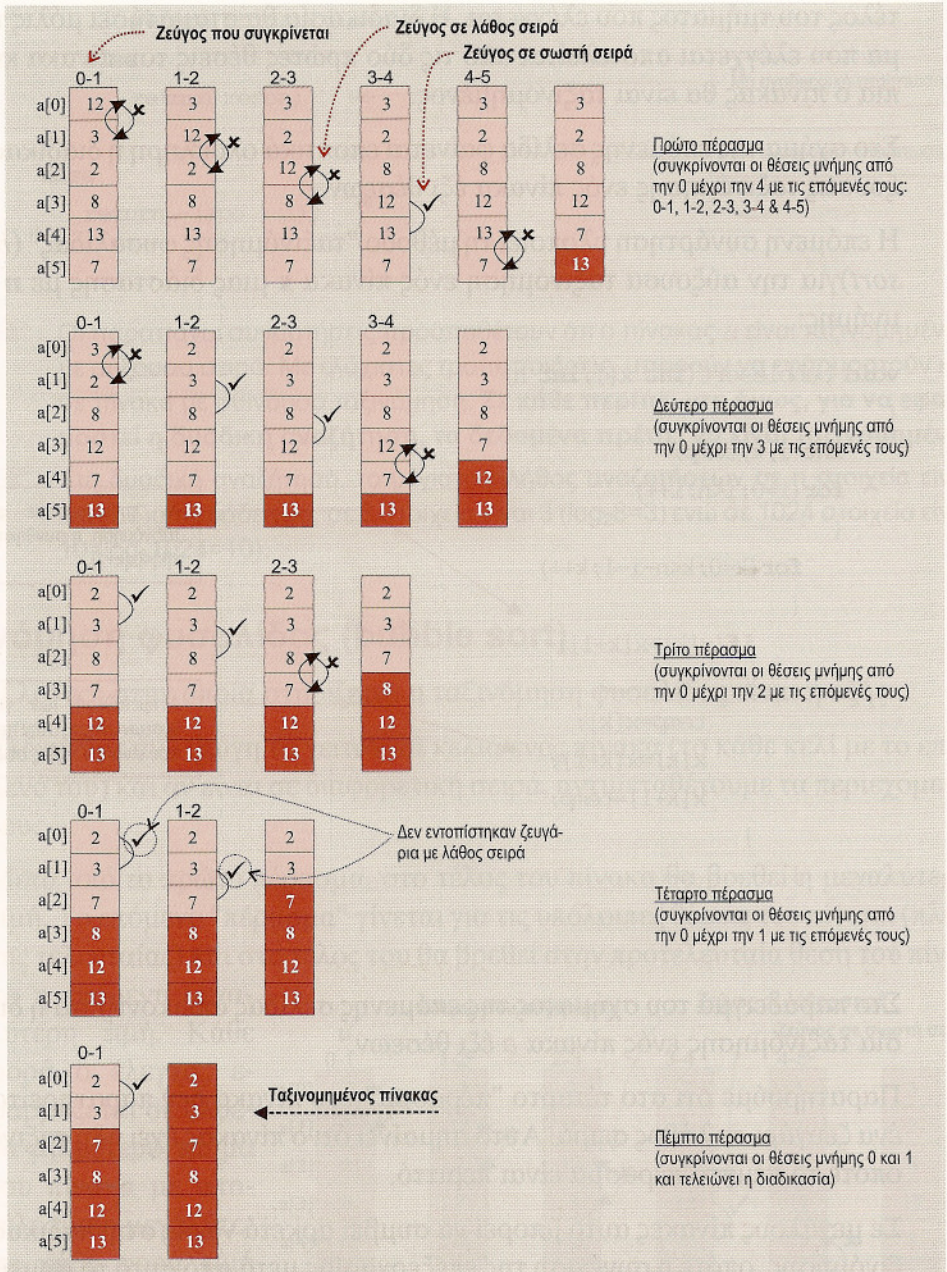
Αν επιθυμούσαμε φθίνουσα ταξινόμηση, η συνθήκη θα ήταν  $x[k]<x[k+1]$ .

Αντιμετάθεση των γειτονικών θέσεων μνήμης στην περίπτωση που είναι σε λάθος σειρά.

Στο παράδειγμα του σχήματος της επόμενης σελίδας απεικονίζεται η διαδικασία ταξινόμησης ενός πίνακα  $a$  έξι θέσεων.

Παρατηρούμε ότι στο τέταρτο "πέρασμα" του πίνακα δεν παρατηρείται ούτε ένα ζευγάρι σε λάθος σειρά. Αυτό σημαίνει ότι ο πίνακας έχει ήδη ταξινομηθεί οπότε το πέμπτο πέρασμα είναι περιττό.

Σε μεγάλους πίνακες αυτό μπορεί να συμβεί αρκετά νωρίς στη διαδικασία ταξινόμησης, οπότε η συνέχιση της επεξεργασίας μετά από αυτό το σημείο είναι εντελώς άσκοπη.





Η επόμενη συνάρτηση είναι μια βελτιωμένη έκδοση της προηγούμενης, η οποία λαμβάνει υπόψη και αυτό το ενδεχόμενο. Με άλλα λόγια, στην περίπτωση που σε κάποιο πέρασμα δεν βρεθεί ούτε ένα ζευγάρι σε λάθος σειρά, ο πίνακας έχει ήδη ταξινομηθεί και η διαδικασία σταματάει.

```
void bubblesort2 (int x[], int n)
```

```
{
    int i, k, temp, found;
    for (i=0; i<n; i++)
    {
        found=0;
        for (k=0; k<n-i-1; k++)
        {
            if (x[k]>x[k+1])
            {
                temp=x[k];
                x[k]=x[k+1];
                x[k+1]=temp;
                found=1;
            }
        }
        if (found==0) break;
    }
}
```

Στη βοηθητική μεταβλητή found καταχωρίζεται το 0.

Όταν βρεθεί έστω και ένα ζευγάρι σε λάθος σειρά στη found, καταχωρίζεται το 1.

Αν η found εξακολουθεί να είναι 0, σημαίνει ότι στο τελευταίο πέρασμα δεν βρέθηκε κανένα ζευγάρι σε λάθος θέση, οπότε ο πίνακας έχει ταξινομηθεί και δεν γίνεται άλλο πέρασμα.

## Ταξινόμηση επιλογής (selection sort)

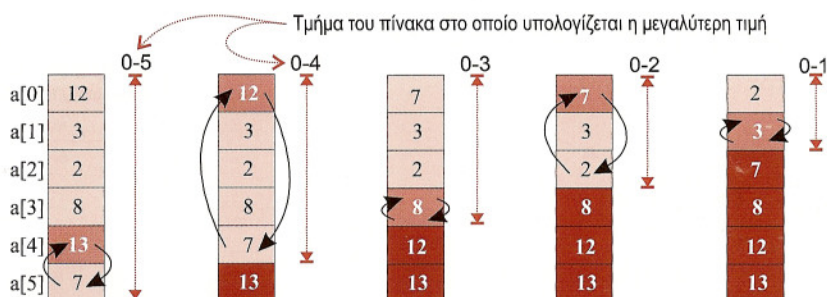
Η λογική στην οποία στηρίζεται η ταξινόμηση επιλογής είναι η εξής:

Ελέγχουμε αρχικά όλο τον πίνακα και εντοπίζουμε το στοιχείο με τη μεγαλύτερη τιμή. Βάζουμε την τιμή αυτή στην τελευταία θέση του πίνακα και το περιεχόμενο της τελευταίας θέσης στη θέση που ήταν η μεγαλύτερη τιμή.

Την επόμενη φορά κάνουμε το ίδιο για τις υπόλοιπες θέσεις του πίνακα (πλην της τελευταίας) και τη μεγαλύτερη τιμή που θα βρούμε τη βάζουμε στην προτελευταία θέση του πίνακα.



Στη συνέχεια, ο έλεγχος εφαρμόζεται κάθε φορά σε ολοένα μικρότερο τμήμα του πίνακα, με αποτέλεσμα να αντιμετωπίζεται η μεγαλύτερη τιμή του τμήματος με την τιμή της τελευταίας θέσης του τμήματος. Η διαδικασία θα σταματήσει όταν φτάσει το τμήμα που ελέγχεται να αποτελείται από τις δύο πρώτες θέσεις του πίνακα — τότε πια, ο πίνακας είναι ταξινομημένος.



Η παρακάτω συνάρτηση υλοποιεί την ταξινόμηση επιλογής για την αύξουσα ταξινόμηση ενός πίνακα  $x$  μιας διάστασης, με  $n$  θέσεις μνήμης:

```
void selectsort(int x[], int n)
```

```
{
    int i, k, temp;
    int max, maxpos;
    for (i=n-1; i>0; i--)
    {
        max=x[0];
        maxpos=0;
        for (k=0; k<=i; k++)
        {
            if (x[k]>max)
            {
                max=x[k];
                maxpos=k;
            }
        }
        temp=x[i];

```

Στις μεταβλητές `max` και `maxpos` καταχωρίζονται η μέγιστη τιμή και η θέση της μέγιστης τιμής για κάθε τμήμα του πίνακα.

Εύρεση της μεγαλύτερης τιμής και της θέσης της στο τμήμα.

Αντιμετάθεση του κελιού με τη μέγιστη τιμή με το τελευταίο κελί του τμήματος.

```

x[i]=x[maxpos];
x[maxpos]=temp;
}
}

```

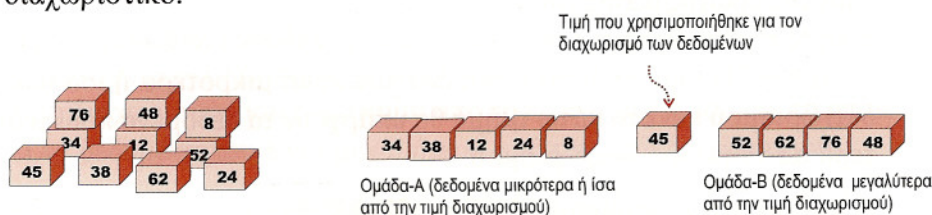
- ☞ Η προηγούμενη συνάρτηση θα μπορούσε με μικρές αλλαγές να ταξινομεί τον πίνακα με φθίνουσα σειρά: αρκεί να υπολογίζουμε το μικρότερο αντί για το μεγαλύτερο αριθμό κάθε τμήματος.

## Ταξινόμηση quick sort

Η **quick sort** ("γρήγορη ταξινόμηση") είναι ένας αλγόριθμος ταξινόμησης που βασίζεται στη φιλοσοφία "διαίρει και βασίλευε" και χρησιμοποιεί εκτενώς τη διαδικασία της αναδρομής.

Η λογική του αλγόριθμου συνοψίζεται στα εξής:

- Από τα δεδομένα που θέλουμε να ταξινομήσουμε, παίρνουμε ένα τυχαίο δεδομένο (συνήθως το πρώτο) το οποίο θα το χρησιμοποιήσουμε σαν διαχωριστικό.
- Χωρίζουμε τα δεδομένα σε δύο ομάδες: Σε αυτά που έχουν τιμή μικρότερη ή ίση με το διαχωριστικό και σε αυτά που έχουν τιμή μεγαλύτερη από το διαχωριστικό.



- Σε κάθε μία από τις ομάδες κάνουμε την ίδια ακριβώς διαδικασία, μέχρι τα τμήματα που θα προκύψουν να έχουν μόνο ένα δεδομένο. Τότε τα δεδομένα μας είναι ταξινομημένα.

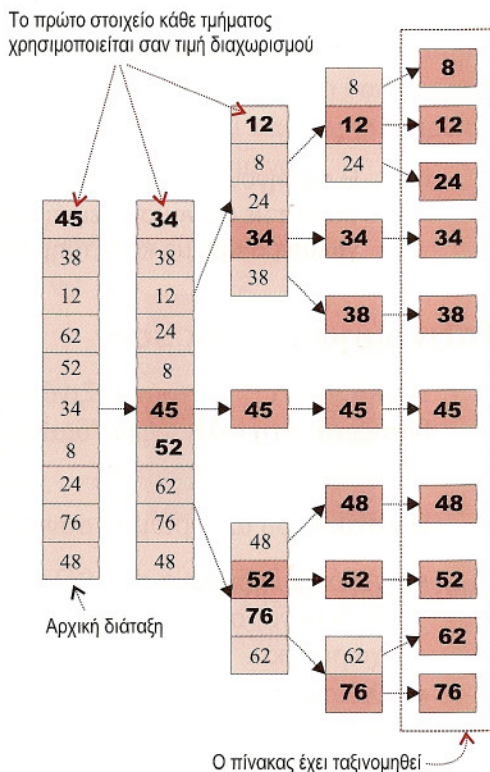
Θεωρούμε τον πίνακα του διπλανού σχήματος. Μετά τον πρώτο διαχωρισμό (με τιμή διαχωρισμού το 45), ο πίνακας χωρίζεται σε τρία βασικά τμήματα. Στο τμήμα με τιμές μικρότερες από την τιμή διαχωρισμού, και στο τμήμα με τιμές μικρότερες από την τιμή διαχωρισμού.

Επαναλαμβάνουμε την ίδια διαδικασία για κάθε τμήμα ξεχωριστά, οπότε τελικά καταλήγουμε σε μια ταξινομημένη διάταξη των στοιχείων του.

Η βάση του αλγόριθμου quick sort είναι μια συνάρτηση η οποία επιτελεί τη λειτουργία που αναφέραμε σε μια συνεχόμενη περιοχή κελιών ενός πίνακα (από το **apo** έως το **eos**). Σαν τιμή διαχωρισμού χρησιμοποιείται η τιμή της πρώτης θέσης του τμήματος

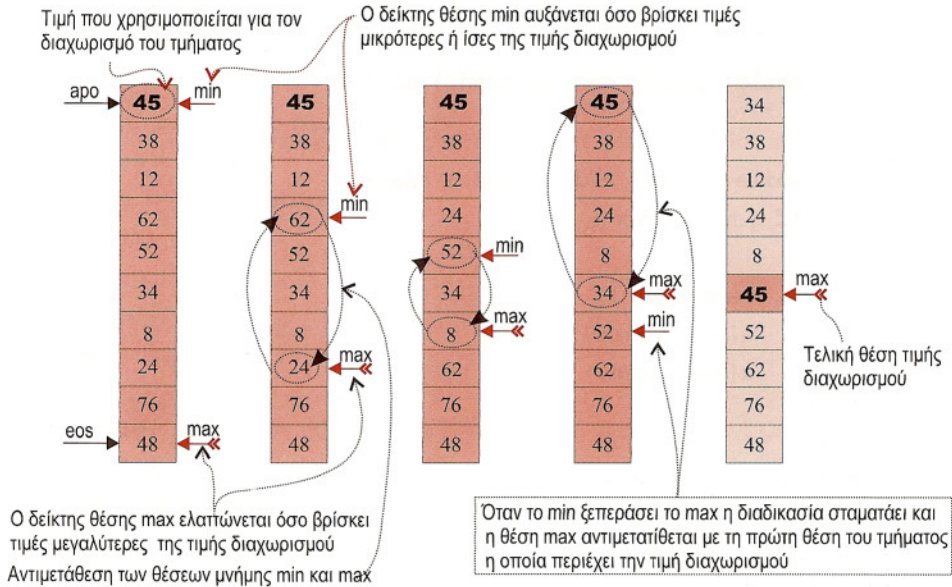
Στόχος της συνάρτησης είναι να χωρίσει τον πίνακα σε δύο τμήματα. Στο επάνω τμήμα θα υπάρχουν τα δεδομένα που είναι μικρότερα ή ίσα από την τιμή διαχωρισμού και στο κάτω τμήμα θα υπάρχουν τα δεδομένα που είναι μεγαλύτερα από την τιμή διαχωρισμού.

Η συνάρτηση χρησιμοποιεί δύο μεταβλητές, τη **min** και τη **max**, οι οποίες αρχικά δείχνουν το πρώτο και το τελευταίο δεδομένο της περιοχής. Στη συνέχεια, η **min** αυξάνεται όσο τα δεδομένα που δείχνει είναι μικρότερα ή ίσα από την τιμή διαχωρισμού (οπότε ουσιαστικά σταματάει μόλις βρει μεγαλύτερη τιμή από την τιμή διαχωρισμού). Αντίστοιχα, η **max** ελαττώνεται μέχρις ότου τα δεδομένα που δείχνει να είναι μεγαλύτερα από την τιμή διαχωρισμού (οπότε ουσιαστικά σταματάει μόλις βρει τιμή μικρότερη ή ίση από την τιμή διαχωρισμού). Όταν σταματήσουν οι δύο δείκτες, τότε αντιμεταθέτουμε τις θέσεις μνήμης **min** και **max** και συνεχίζουμε την ίδια διαδικασία.





Η διαδικασία σταματάει όταν το **min** ξεπεράσει το **max**. Το **max** θα δείχνει τότε τη θέση που πρέπει να έχει η τιμή διαχωρισμού και θα αντιμεταθέτει τη θέση αυτή με την πρώτη θέση της περιοχής (η οποία περιέχει την τιμή διαχωρισμού).



Η περιοχή έχει χωριστεί πλέον σε δύο τμήματα, στο τμήμα που είναι από το **apo** μέχρι το **max-1** και περιέχει τα δεδομένα που είναι μικρότερα ή ίσα από την τιμή διαχωρισμού και στο τμήμα από το **max+1** μέχρι το **eos**, το οποίο περιέχει τα δεδομένα που είναι μεγαλύτερα από την τιμή διαχωρισμού.

Αν τώρα στα δύο τμήματα που προέκυψαν γίνει αναδρομικά ή ίδια διαδικασία, η περιοχή από το **apo** μέχρι το **eos** θα έχει ταξινομηθεί.

Ο επόμενος κώδικας παρουσιάζει τη συνάρτηση που υλοποιεί τη λογική που αναφέρθηκε:

```
int partition( int a[], int apo, int eos )
{
    int min, max, temp;
    int diax_timi;
    diax_timi = a[apo];
```

```

min = apo;
max = eos;
while ( min < max )
{
    /* Μετακίνησε τον δείκτη θέσης min προς τα κάτω
    όσο βρίσκεις τιμές μικρότερες ή ίσες της τιμής διαχωρισμού*/
    while( a[min] <= diax_timi ) min++;
    /* Μετακίνησε το δείκτη θέσης max προς τα πάνω
    όσο βρίσκεις τιμές μεγαλύτερες της τιμής διαχωρισμού*/
    while( a[max] > diax_timi ) max--;
    if ( min < max )
    {
        temp=a[min];
        a[min]=a[max];
        a[max]=temp;
    }
}
/* η θέση max είναι η τελική θέση της τιμής διαχωρισμού*/
a[apo] = a[max];
a[max] = diax_timi;
return max;
}

```

Η συνάρτηση επιστρέφει ως τιμή τη θέση στην οποία κατέληξε η τιμή διαχωρισμού.

Η quick sort υλοποιείται από τον επόμενο κώδικα, με χρήση της παραπάνω συνάρτησης, και με αναδρομικές κλήσεις στα τμήματα που προκύπτουν από τη διαίρεση της αρχικής περιοχής:

```

void q_sort(int a[], int apo, int eos )
{
    int diax_pos;
    /* Μη αναδρομική περίπτωση */
    if(eos <= apo) return;
    diax_pos = partition(a, apo, eos);
    /* Εκτέλεσε αναδρομικά την ίδια διαδικασία για το επάνω τμήμα */
    q_sort(a, apo, diax_pos-1);
}

```

```

/* Εκτέλεσε αναδρομικά την ίδια διαδικασία για το κάτω τμήμα */
q_sort( a, diax_pos+1, eos );
}

```

Ο επόμενος κώδικας ενσωματώνει τις δύο προηγούμενες συναρτήσεις σε μία:

```

void q_sort(int a[], int apo, int eos )
{
    int min, max, diax_pos, temp;
    int diax_timi;
    if (eos <= apo) return;
    diax_timi = a[apo];
    min = apo;
    max = eos;
    while ( min < max )
    {
        while( a[min] <= diax_timi ) min++;
        while( a[max] > diax_timi ) max--;
        if ( min < max )
        {
            temp=a[min];
            a[min]=a[max];
            a[max]=temp;
        }
    }
    a[apo] = a[max];
    a[max] = diax_timi;
    diax_pos=max;
    if (apo<diax_pos)
        q_sort( a, apo, diax_pos-1 );
    if (eos>diax_pos)
        q_sort( a, diax_pos+1, eos );
}

```



Η αποδοτικότητα του αλγόριθμου quick sort εξαρτάται άμεσα από την επιλογή της τιμής διαχωρισμού. Η επιλογή της πρώτης θέσης για τιμή διαχωρισμού συνιστάται στην περίπτωση που τα δεδομένα είναι κατανεμημένα τυχαία. Στην περίπτωση όμως που τα δεδομένα έχουν μια κάποια διατεταγμένη μορφή τότε η τυχαία επιλογή τις τιμής διαχωρισμού οδηγεί σε καλύτερη απόδοση.

Η μέθοδος **quick sort** είναι μακράν ο πιο γρήγορος από τους κλασικούς αλγόριθμους ταξινόμησης όταν τα προς ταξινόμηση δεδομένα έχουν τυχαία διασπορά.

Η quick sort δεν είναι όμως πάντοτε η καλύτερη επιλογή:

- Είναι εκτενώς αναδρομική, γεγονός που σημαίνει ότι για πολύ μεγάλους πίνακες υπάρχει πιθανότητα εξάντλησης της μνήμης στοίβας που χρησιμοποιεί ο Η/Υ για τις αναδρομικές κλήσεις.
- Είναι πολύ πολύπλοκη και δυσνόητη, ειδικά στην περίπτωση που οι απαιτήσεις ταξινόμησης δεν είναι συχνές και το πλήθος των προς ταξινόμηση δεδομένων μικρό.
- Η απόδοση της quick sort είναι απογοητευτική όταν τα προς ταξινόμηση δεδομένα είναι σχεδόν ταξινομημένα. Στην περίπτωση αυτή η καλύτερη επιλογή είναι η μέθοδος bubble sort.

## Ταξινόμηση πινάκων δύο διαστάσεων

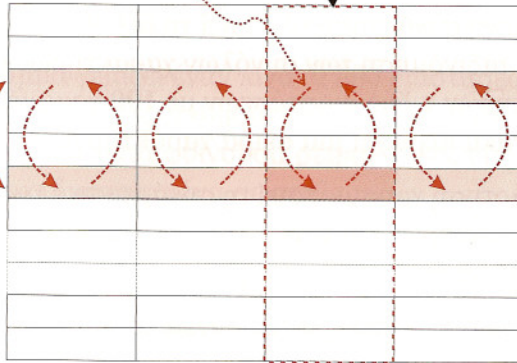
Όλες οι μέθοδοι ταξινόμησης βασίζονται στην αντιμετάθεση θέσεων μνήμης σε έναν πίνακα, έως ότου επιτύχουν την πλήρη τακτοποίησή τους. Σε έναν πίνακα μιας διάστασης, το κλειδί της ταξινόμησης —δηλαδή το δεδομένο που θα καθορίσει τη σειρά διάταξης— είναι προφανώς τα περιεχόμενα της μοναδικής στήλης του πίνακα.

Σε έναν πίνακα δύο διαστάσεων το κλειδί της ταξινόμησης μπορεί να είναι οποιαδήποτε από τις στήλες του πίνακα. Όταν θα χρειαστεί όμως να γίνει η αντιμετάθεση δύο κλειδιών ώστε να πάνε στην νέα τους θέση, **θα πρέπει η αντιμετάθεση αυτή να γίνει σε ολόκληρες τις γραμμές που περιέχουν τα δύο κλειδιά και όχι μόνο στα συγκεκριμένα κελιά.**

ο έλεγχος γίνεται μόνο στα κελιά της στήλης "κλειδί"

Κλειδί ταξινόμησης

Σειρές που  
πρέπει να  
αντιμετατεθούν



Η παρακάτω συνάρτηση, υλοποιεί τη μέθοδο bubblesort για την αύξουσα ταξινόμηση ενός πίνακα  $x$  δύο διαστάσεων με **100** γραμμές και **5** στήλες ως προς την  $m$  στήλη του:

```
void bubblesort2d(int x[][5], int m)
```

```
{
    int i, j, k, temp;
    for(i=0; i<100; i++)
    {
        for(k=0; k<100-i-1; k++)
        {
            if(x[k][m]>x[k+1][m])
            {
                for(j=0; j<5; j++)
                {
                    temp=x[k][j];
                    x[k][j]=x[k+1][j];
                    x[k+1][j]=temp;
                }
            }
        }
    }
}
```

Αν επιθυμούσαμε φθίνουσα ταξινόμηση, η συνθήκη θα ήταν  $x[k]<x[k+1]$ .

Αντιμετάθεση των γειτονικών θέσεων μνήμης στην περίπτωση που είναι σε λάθος σειρά.

## Ταξινόμηση πίνακα με σειρές χαρακτήρων

Η επόμενη συνάρτηση υλοποιεί τη μέθοδο bubblesort για την αύξουσα αλφαβητική ταξινόμηση των συνόλων χαρακτήρων (character strings) ενός πίνακα χαρακτήρων **x** δύο διαστάσεων με **100** γραμμές και **40** στήλες. Κάθε γραμμή του πίνακα περιέχει μια σειρά χαρακτήρων.

Η συνάρτηση χρησιμοποιεί τη συνάρτηση **strcmp()** για τη σύγκριση δύο σειρών χαρακτήρων και την **strcpy()** για την αντιμετάθεσή τους μέσω ενός προσωρινού πίνακα.

```
void bubblesort_Char_strings(char x[][40])
{
    int i,k;
    char temp[40];
    for(i=0;i<100;i++)
    {
        for(k=0;k<100-i-1;k++)
        {
            if(strcmp(x[k],x[k+1])>=1)
            {
                strcpy(temp,x[k]);
                strcpy(x[k],x[k+1]);
                strcpy(x[k+1],temp);
            }
        }
    }
}
```

Αν επιθυμούσαμε φθίνουσα ταξινόμηση, η συνθήκη θα ήταν **strcmp(x[k],x[k+1])<=-1**.

Αντιμετάθεση των γειτονικών συνόλων χαρακτήρων μέσω αντιγραφής τους στον προσωρινό πίνακα **temp**.



## Παραδείγματα

**Π.1** Το επόμενο πρόγραμμα γεμίζει έναν πίνακα δύο διαστάσεων 100 γραμμών και 4 στηλών με τυχαίους αριθμούς και μετά τον ταξινομεί ως προς την τελευταία στήλη του κατά φθίνουσα σειρά. Τέλος εμφανίζει τα περιεχόμενα του πίνακα διατεταγμένα σε στήλες.

```
void bubblesort2d();
```

```
main()
```

```
{
```

```
    int a[100][4];
```

```
    int i, j;
```

```
    for(i=0; i<100; i++)
```

```
        for(j=0; j<4; j++)
```

```
            a[i][j]=rand();
```

```
    bubblesort2d(a, 3);
```

```
    for(i=0; i<100; i++)
```

```
    {
```

```
        for(j=0; j<4; j++)
```

```
        {
```

```
            printf("%10d", a[i][j]);
```

```
        }
```

```
        putchar('\n');
```

```
    }
```

```
}
```

Συμπλήρωση του πίνακα a με τυχαίους αριθμούς.

Κλήση της συνάρτησης bubblesort2d() που ακολουθεί, με παραμέτρους τον πίνακα και το 3 (τελευταία στήλη).

Εμφάνιση του πίνακα σε στήλες.

Αλλαγή γραμμής κάθε 4 στήλες.

```
void bubblesort2d(int x[][4], int m)
```

```
{
```

```
    int i, j, k, temp;
```

```
    for(i=0; i<100; i++)
```

```
    {
```

```
        for(k=0; k<100-i-1; k++)
```

```
        {
```

```
            if(x[k][m]<x[k+1][m])
```

Το < για φθίνουσα ταξινόμηση

```
{
    for (j=0; j<4; j++)
    {
        temp=x[k][j];
        x[k][j]=x[k+1][j];
        x[k+1][j]=temp;
    }
}
}
```

Αντιμετάθεση των γειτονικών θέσεων μνήμης στην περίπτωση που είναι σε λάθος σειρά.

## Ανασκόπηση Κεφαλαίου 16

- Αναζήτηση είναι η διαδικασία εντοπισμού ενός δεδομένου σε μια δομή δεδομένων.
- Στη σειριακή αναζήτηση, ο έλεγχος των δεδομένων γίνεται με τη σειρά από το πρώτο μέχρι το τελευταίο, έως ότου να γίνει ο εντοπισμός του προς αναζήτηση στοιχείου.
- Η δυαδική αναζήτηση μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα.
- Ταξινόμηση είναι η διαδικασία κατά την οποία γίνεται η διάταξη κάποιων δεδομένων σε αύξουσα ή φθίνουσα σειρά.
- Η ταξινόμηση αριθμητικών δεδομένων με αύξουσα σειρά διατάσσει τα δεδομένα από το μικρότερο προς το μεγαλύτερο, ενώ με φθίνουσα σειρά τα διατάσσει αντίστροφα.
- Η ταξινόμηση αλφαριθμητικών δεδομένων διατάσσει τα δεδομένα με απόλυτη αλφαβητική σειρά.

## Ασκήσεις Κεφαλαίου 16

- 16.1** Να γραφεί πρόγραμμα το οποίο να ζητάει μια λέξη, να ταξινομεί τους χαρακτήρες της λέξης κατά αλφαβητική σειρά, και να την εμφανίζει στην οθόνη. Για παράδειγμα, αν πληκτρολογηθεί η λέξη ANANAS θα εμφανίσει τη λέξη AAANNΣ. ★★
- 16.2** Να γραφεί πρόγραμμα (ή συνάρτηση) που να ταξινομεί κατά αύξουσα αλφαβητική ταξινόμηση, **ως προς το δεύτερο χαρακτήρα τους**, τις σειρές χαρακτήρων ενός πίνακα χαρακτήρων δύο διαστάσεων με 100 γραμμές και 40 στήλες. ★★★
- 16.3** Αν υποθέσουμε ότι στον πίνακα χαρακτήρων `lex` υπάρχει καταχωρισμένη η λέξη "ΧΑΡΟΥΜΕΝΟΣ", ποιο θα είναι το περιεχόμενο του πίνακα `lex` μετά από την εκτέλεση του επόμενου κώδικα; ★★

```
int i=0,p1,p2;
char ch1,ch2,temp;
ch1=ch2=lex[0];
p1=p2=0;
while(lex[i]!='\0')
{
    if(ch1>lex[i])
    {
        ch1=lex[i];
        p1=i;
    }
    if(ch2<lex[i])
    {
        ch2=lex[i];
        p2=i;
    }
    i++;
}
temp=lex[0];
```



```
lex[0]=lex[p2];  
lex[p2]=temp;  
temp=lex[i-1];  
lex[i-1]=lex[p1];  
lex[p1]=temp;
```

**16.4** Να γραφεί συνάρτηση η οποία θα προσθέτει ένα χαρακτήρα, παρεμβάλλοντάς τον στη σωστή του θέση, μέσα σε έναν ταξινομημένο πίνακα χαρακτήρων. Για παράδειγμα, αν ο πίνακας περιέχει τους χαρακτήρες "ΑΓΓΔΚΜΧ" και ο χαρακτήρας που πρόκειται να προστεθεί είναι ο 'Ε', η συνάρτηση θα έχει αποτέλεσμα να περιέχει ο πίνακας τους χαρακτήρες "ΑΓΓΔΕΚΜΧ". Η συνάρτηση να δέχεται δύο παραμέτρους: τον πίνακα χαρακτήρων και το χαρακτήρα που πρόκειται να προστεθεί. ★ ★ ★

**16.5** Ποια από τα επόμενα αληθεύουν: ★

- ☐ Η δυαδική αναζήτηση προϋποθέτει ταξινομημένα δεδομένα.
- ☐ Οι διαδικασίες ταξινόμησης είναι γενικά χρονοβόρες σε μεγάλους πίνακες.
- ☐ Σε ένα μεγάλο πίνακα με τυχαίους αριθμούς η καλύτερη μέθοδος ταξινόμησης είναι η μέθοδος της φυσαλίδας.
- ☐ Σε έναν μισοταξινομημένο πίνακα η καλύτερη μέθοδος ταξινόμησης είναι η quick sort.
- ☐ Σε έναν πίνακα με τυχαίους αριθμούς, η μόνη μέθοδος αναζήτησης είναι η σειριακή.