κεφάλαιο

9

Πίνακες Ι

Ο στόχος μας σε αυτό το κεφάλαιο:

Συχνά έχουμε πολλές μεταβλητές με τις ίδιες ιδιότητες που πρέπει να υποστούν την ίδια επεξεργασία. Θα μάθεις πώς να τις οργανώνεις σε *πίνακες* ώστε να τις χειρίζεσαι με τις γνωστές εντολές επανάληψης.

Προσδοκώμενα αποτελέσματα:

Ο πίνακας είναι ένα πολύ καλό εργαλείο για πάρα πολλές χρήσεις. Εδώ θα δεις μερικές επεξεργασίες πινάκων που θα σου είναι χρήσιμες πολύ συχνά.

Έννοιες κλειδιά:

- *πίνακας*
- *στοιχείο πίνακα, δείκτης*
- *αναζήτηση τιμής σε πίνακα*
- *ταξινόμηση πίνακα*
- *συγχώνευση πινάκων*
- *αποδοτικότητα αλγόριθμου*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει 100 πραγματικούς αριθμούς x_1, x_2, ..., x_{100} και

α) θα υπολογίζει και θα τυπώνει τη Μέση Αριθμητική Τιμή τους <*x*>,

β) θα υπολογίζει και θα τυπώνει τις τιμές της συνάρτησης: $e^{\frac{<x>-x}{<x>}}$ για τους αριθμούς αυτούς. Δηλαδή τα $y_k = e^{\frac{<x>-x_k}{<x>}}$.

Ε, αυτό το λύσαμε στο προηγούμενο κεφάλαιο και μάλιστα όχι για 100 αλλά για όσους αριθμούς και να έχουμε! Γράφουμε σε ένα αρχείο τους αριθμούς όταν πληκτρολογούνται και όταν έχουμε υπολογίσει την <*x*>, διαβάζουμε το αρχείο από την αρχή.

Πόσο καλή είναι αυτή η λύση; Όχι και πολύ καλή. Δεν είναι παραδεκτό να στέλνουμε 100 αριθμούς σε βοηθητική μνήμη και να τους ξαναδιαβάζουμε για να χρησιμοποιηθούν δεύτερη φορά από το ίδιο πρόγραμμα. Πρέπει να τους κρατήσουμε στην κύρια μνήμη όσο τους χρειαζόμαστε.

Αλλά, σύμφωνα με όσα ξέρουμε, θα πρέπει δηλώσουμε 100 μεταβλητές: *x*1, *x*2, ...,*x*100 τύπου `double` και να γράψουμε 100 εντολές:

```
cin >> x1; cin >> x2; ... ; cin >> x100;
```

για να τους διαβάσουμε.

Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνει έναν πιο άνετο τρόπο για να λύσουμε το πρόβλημά μας. Το εργαλείο που θα χρησιμοποιήσουμε λέγεται **πίνακας** και θα το γνωρίσουμε στις παραγράφους που ακολουθούν.

9.1 Πίνακες Στοιχείων

Η έννοια του **πίνακα** (array, matrix) είναι γνωστή από τα μαθηματικά. Στη C++ ο πίνακας δεδομένων αποτελείται από έναν αριθμό *ομοειδών στοιχείων*, δηλ. από *στοιχεία του ίδιου τύπου*. Ο τύπος αυτός καθορίζεται όταν δηλώνουμε τη μεταβλητή - πίνακα.

Τα ομοειδή στοιχεία, που αποτελούν τον πίνακα, έχουν πάντα ένα κοινό όνομα, π.χ. τα pinA, pinB και flNm στο παρακάτω παράδειγμα δήλωσης πινάκων:

```
int    pinA [ 20 ];
double pinB [ 50 ];
char   flNm [ 120 ];
```

Η πρώτη δήλωση του παραδείγματος καθορίζει ότι ο πίνακας pinA αποτελείται από 20 στοιχεία που το καθένα έχει τα χαρακτηριστικά μιας μεταβλητής τύπου `int`, η δεύτερη γραμμή δηλώνει ότι ο πίνακας pinB αποτελείται από 50 στοιχεία που έχουν χαρακτηριστικά μεταβλητής τύπου `double` και η τρίτη ότι ο πίνακας flNm αποτελείται από 120 στοιχεία τύπου `char`.

Σχηματικά μπορούμε να βλέπουμε, π.χ. τον πίνακα pinA, σαν ένα μονοδιάστατο πίνακα που αποτελείται από μια γραμμή και 20 στήλες ή από μια στήλη και 20 γραμμές. Το Σχ. 9-1 δείχνει την πρώτη περίπτωση δομής.

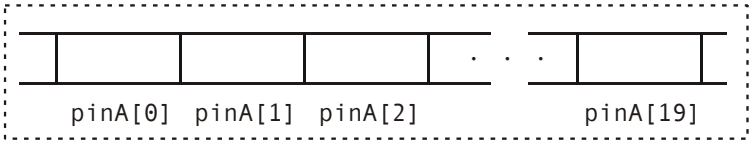
Τα **στοιχεία** (elements) ή οι **συνιστώσες** (components) του πίνακα σημειώνονται πάντα με το κοινό όνομα της μεταβλητής-πίνακα που συνοδεύεται από τον λεγόμενο **δείκτη** (index).

- ***Αν ο πίνακας έχει δηλωθεί με n στοιχεία, οι επιτρεπτές τιμές του δείκτη είναι όλοι οι φυσικοί από 0 μέχρι n – 1.***

Ο δείκτης του στοιχείου πίνακα δίνεται πάντα μέσα σε αγκύλες, όπως δείχνουν τα παρακάτω παραδείγματα:

```
pinA[1],  pinA[5],  pinA[15]
pinB[2],  pinB[10],  pinB[49]
```

Έτσι, τα συμβολικά ονόματα της πρώτης γραμμής καθορίζουν το δεύτερο, το έκτο και το δέκατο έκτο στοιχείο του πίνακα pinA. Ενώ τα ονόματα της δεύτερης γραμμής καθορίζουν το τρίτο, το ενδέκατο και το τελευταίο στοιχείο του πίνακα pinB. Τέλος το pinA`[`*k*`]` καθορί-



Σχ. 9-1 Σχηματική παράσταση πίνακα και των στοιχείων του.

ζει γενικώς το (*k*+1)-οστό στοιχείο του πίνακα pinA, όπου *k* = 0, 1,... 19. Αντιστοίχως το pinB`[`*k*`]` καθορίζει το (*k*+1)-οστό στοιχείο του πίνακα pinB, όπου *k* = 0, 1,... 49.

Το γενικό συντακτικό δήλωσης του μονοδιάστατου πίνακα μπορεί να δοθεί ως εξής:

τύπος, *αναγνωριστικό*, "`[`", *πλήθος*, "`]`"

- Ο *τύπος*, που λέγεται και **τύπος συνιστωσών** (component type), μπορεί να είναι οποιοσδήποτε τύπος, π.χ. `int`, `double`, `char`, `bool`, απαριθμητός κλπ.
- Το *αναγνωριστικό* είναι σαν και αυτά των μεταβλητών.
- Το *πλήθος* είναι μια παράσταση που μπορεί να περιέχει σταθερές, μεταβλητές που έχουν ήδη τιμή και συναρτήσεις, αν φυσικά η δήλωσή μας βρίσκεται μέσα στην εμβέλειά τους. Η τιμή της πρέπει να είναι ακέραιου τύπου και μεγαλύτερη από 0. Το *πλήθος* μπορεί να παραλείπεται.

Κοίταξε τα παρακάτω παραδείγματα:

```
const int N( 50 ), N1( 63 ), N2( 114 );

typedef int  PinAk[ N+1 ];
enum DecDigit { zero = 48, one, two, three, four, five, six,
                seven, eight, nine };

PinAk    p, q, r;
int      l[ N+1 ], m[ N2-N1+1 ];
bool     signal[ (N1+1)/4 ];
ifstream keimena[ 4 ];
DecDigit bv[ N/2+1 ];
PinAk    mat[ 11 ];
```

Κατ' αρχάς ορίζουμε έναν τύπο, τον *PinAk*· κάθε αντικείμενο αυτού του τύπου, όπως οι p, q, r που δηλώνουμε παρακάτω, είναι ένας πίνακας με 51 (= *N* + 1) στοιχεία τύπου `int`, που αριθμούνται από 0 μέχρι 50, π.χ. p`[`0`]`, p`[`1`]`, …, p`[`50`]`.

Στη συνέχεια αντιγράφουμε τον ορισμό του τύπου *DecDigit* από το Κεφ. 4 και παρακάτω δηλώνουμε τον πίνακα bv που έχει 26 (= *N*/2 + 1) στοιχεία τύπου *DecDigit*.

Ο πίνακας `l` έχει 51 στοιχεία τύπου `int`· θα μπορούσαμε να είχαμε δηλώσει "`PinAk l`". Ο πίνακας m έχει 52 στοιχεία τύπου `int`.

Ο keimena είναι ένας πίνακας που κάθε στοιχείο του είναι ένα ρεύμα τύπου *ifstream*.

Τέλος, στην τελευταία δήλωση, ο mat έχει 11 στοιχεία που το καθένα τους είναι ένας πίνακας τύπου *PinAk*· είναι ένας πίνακας πινάκων ή δισδιάστατος πίνακας. Με τέτοιους πίνακες θα ασχοληθούμε αργότερα.

Μαζί με τη δήλωση ενός πίνακα μπορείς να δώσεις και αρχικές τιμές στα στοιχεία του. Π.χ.:

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

Στην περίπτωση αυτή δεν χρειάζεται να δηλώσεις και το πλήθος (ο μεταγλωττιστής ξέρει να μετράει). Θα μπορούσαμε να γράψουμε:

```
int monthLength[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

Αν δηλώσεις το πλήθος των στοιχείων και στη λίστα αρχικών τιμών βάλεις λιγότερες τιμές τότε οι υπόλοιπες θεωρούνται μηδέν. Π.χ. η

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31 };
```

είναι ισοδύναμη με την:

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31, 0, 0, 0, 0 };
```

Όπως καταλαβαίνεις, αν θέλεις να βάλεις αρχική τιμή "`0`" σε όλα τα στοιχεία ενός πίνακα αρκεί να γράψεις ένα "`0`" ανάμεσα στα άγκιστρα:

```
double x[ 100 ] = { 0.0 };
```

Ωραία λοιπόν, δηλώσαμε τον πίνακα. Τώρα τι κάνουμε; Με ολόκληρο τον πίνακα δεν μπορούμε να κάνουμε και πολλά πράγματα, αλλά:

- ***Κάθε συνιστώσα ενός πίνακα έχει όλα τα χαρακτηριστικά μιας μεταβλητής του τύπου συνιστωσών και μπορείς να τη χειριστείς όπως και κάθε μεταβλητή αυτού του τύπου.***

Έτσι, π.χ. το `l[17]` είναι μεταβλητή τύπου `int`. Μπορούμε λοιπόν να της δώσουμε τιμή με μια εντολή εκχώρησης:

```
l[17] = 3215;
```

ή να τη διαβάσουμε από το πληκτρολόγιο:

```
cin >> l[17];
```

Αν η `l[17]` έχει τιμή μπορούμε να τη χρησιμοποιήσουμε σε παραστάσεις:

```
x = l[17] / 5 + 100;
```

να γράψουμε την τιμή της στην οθόνη ή σε κάποιο αρχείο:

```
cout << l[17] << endl;
```

Το signal`[11]` είναι μεταβλητή τύπου `bool` –μπορεί να πάρει τιμές `true` ή `false`, π.χ.:

```
signal[11] = ( l[17] >= x );
if ( signal[11] && x > 1000 )  { ...
```

Όπως είδαμε και πιο πάνω, με τον δείκτη ξεχωρίζουμε τις συνιστώσες ενός πίνακα. Ο δείκτης είναι, γενικά, μια παράσταση. Η τιμή αυτής της παράστασης θα πρέπει να είναι μη αρνητική και μικρότερη από το πλήθος στοιχείων που έχουμε δηλώσει. Μετά τη δήλωση:

```
int  metr[ 26 ];
```

μπορείς να δώσεις:

```
metr[0] = -16;
```

αλλά, δεν μπορείς να δώσεις:

```
metr[27] = 24;
```

ούτε

```
metr[-5] = 33;
```

Αυτό είναι λάθος που, δυστυχώς, *δεν θα το δείξει* ο μεταγλωττιστής. Ακόμη, αν *c* `==` 22, μπορείς να δώσεις:

```
metr[c+1] = c - 17;
```

αλλά, αν *c* `==` 25, η παραπάνω εντολή είναι λάθος: `metr[26]` *δεν υπάρχει!* Αυτό το λάθος φυσικά δεν μπορεί να το εντοπίσει ο μεταγλωττιστής, αλλά, δυστυχώς, μπορεί να μη φανεί ούτε κατά την εκτέλεση. Θα οδηγήσει, πιθανότατα, σε παράλογα αποτελέσματα (μπορεί και να «παγώσει» ο υπολογιστής σου).

- ***Για πίνακα με n στοιχεία είναι υποχρέωση του προγραμματιστή να περιορίζει την τιμή του δείκτη στις επιτρεπτές τιμές (από 0 μέχρι n – 1).***

Όπως φαίνεται από τα παραπάνω παραδείγματα, ο δείκτης του πίνακα μπορεί να είναι, γενικά, μια παράσταση. Έχουμε λοιπόν τη δυνατότητα πρόσβασης στα διάφορα στοιχεία ενός πίνακα, δίνοντας την κατάλληλη τιμή στο δείκτη. Αν θέλουμε να χειριστούμε όλα τα στοιχεία ενός πίνακα, μπορούμε να το πετύχουμε με μια επανάληψη όπου η μεταβαλλόμενη τιμή του δείκτη μας δίνει το ένα στοιχείο μετά το άλλο.

Η C++ δεν μας επιτρέπει διαχείριση ολόκληρου του πίνακα. Για παράδειγμα η εκχώρηση πρέπει να γίνεται στοιχείο προς στοιχείο. Ας πούμε ότι έχουμε δηλώσει:

```
int  a[5], b[5];
```

και κάποτε θέλουμε να εκχωρήσουμε την τιμή που έχει εκείνη τη στιγμή ο a στον b. *Δεν επιτρέπεται να δώσουμε:* `b = a`· θα πρέπει να δώσουμε την εντολή:

```
   for ( k = 0; k <= 4; k = k+1 )  b[k] = a[k];
```

Τώρα, μπορούμε να ξαναγυρίσουμε στο πρόβλημα της εισαγωγικής παραγράφου και να δούμε μια πιο όμορφη λύση.

Παράδειγμα

Ας έλθουμε τώρα στο πρόβλημα που δώσαμε στην εισαγωγή. Στο πρόγραμμα-λύση που γράψαμε στο προηγούμενο κεφάλαιο η *x* ήταν μια απλή μεταβλητή. Κάθε φορά που διαβάζαμε μια νέα τιμή, η προηγούμενη τιμή της *x* χάνονταν, αφού φυσικά την είχαμε φυλάξει στο αρχείο.

Τώρα θα δηλώσουμε τη x ως:

```
   double   x[ N ];
```

Σε κάθε εκτέλεση της περιοχής της (πρώτης) **for** θα δίνουμε τιμή και σε διαφορετικό στοιχείο του *x*. Όταν υπολογίσουμε τη <*x*>, οι 100 αριθμοί που αποθηκεύτηκαν στον x θα είναι στη διάθεσή μας για δεύτερη χρήση:

```
// πρόγραμμα: Μέση Τιμή 1+
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const int N( 100 );// το πλήθος των στοιχείων
   double x[ N ];     // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;        // Το μερικό (τρέχον) άθροισμα.
                      // Στο τέλος έχει το ολικό άθροισμα.
   double avrg;       // Μέση Αριθμητική Τιμή των x (<x>)
   int m;
   double y;

   sum = 0;
   for ( m = 0; m <= N-1; m = m+1 )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x[m];
   // πρόσθεσέ την στη sum
      sum = sum + x[m];
   } // for
   avrg = sum / N;
   cout.setf( ios::fixed,ios::floatfield );  cout.precision( 3 );
   cout << " ΑΘΡΟΙΣΜΑ = " << sum
        << "  <x> = " << avrg << endl;
   for ( m = 0; m <= N-1; m = m+1 )
   {
      y = exp( (avrg - x[m])/avrg );
      cout << " x[";
      cout.width(3);  cout << m << "] = ";
      cout.width(6);  cout << x[m] << "   y[";
      cout.width(3);  cout << m << "] = " ;
      cout.width(6);  cout << y << endl;
   } // for
} // main
```

Το παραπάνω πρόγραμμα έχει δύο **for**, όπου η μεταβλητή *m* παίζει διπλό ρόλο. Χρησιμοποιήθηκε:

α) ως μεταβλητή ελέγχου της **for** και

β) ως δείκτης πίνακα (που διατρέχει τις τιμές: 0 . . *N*-1).

Αυτή η λύση είναι σαφώς καλύτερη από αυτήν της προηγούμενης παραγράφου. Αυτό το πρόγραμμα είναι φανερά πιο γρήγορο από το πρώτο και αυτό διότι αποφεύγει τη χρονοβόρα διαδικασία γραψίματος/διαβάσματος από τη βοηθητική μνήμη.

Στη σύνθεση του προγράμματος, μπορεί κανείς να παρασυρθεί από τον τύπο $y_k = e^{\frac{<x>-x_k}{<x>}}$ και να δηλώσει το y ως πίνακα. «Το x είναι πίνακας, βλέπουμε και δείκτη στο y... Βάλε έναν πίνακα, να τελειώνουμε!» Αυτό που μας ζητάνε, όμως, είναι να υπολογίσουμε και να τυπώσουμε τα y_k. Μετά δεν μας χρειάζονται πια. Το y θα μπορεί να είναι απλή μεταβλητή τύπου `double`. Άλλωστε, «ζωγραφίζουμε» και τα αποτελέσματά μας έτσι ώστε να βγάζουν δείκτες, για να ευχαριστηθεί και ο χρήστης:

```
ΑΘΡΟΙΣΜΑ = 1772.217  <x> = 17.722
x[  0] = -2.217   y[  0] =  3.081
x[  1] = 58.900   y[  1] =  0.098
x[  2] = 35.850   y[  2] =  0.360
x[  3] = -0.879   y[  3] =  2.857
     . . .
```

Το κράτημα 100 θέσεων `double` στην κύρια μνήμη θα ήταν καθαρή σπατάλη.

Παρατήρηση: ▶

Γλυτώσαμε λοιπόν από το αρχείο! Έτσι, νομίζεις. Πληκτρολογείς 100 αριθμούς για να δοκιμάσεις το πρόγραμμα και βλέπεις ότι έχει λάθος. Μετά, στη δεύτερη δοκιμή, τι κάνεις; Τους ξαναπληκτρολογείς; Δεν είμαστε καλά! Διόρθωσε παιδί μου το πρόγραμμα να τους φυλάξει σε ένα αρχείο! ◀

Αυτό που λέμε στην παρατήρηση θα το πούμε ως γενική συμβουλή:

- ***Όταν πληκτρολογούμε δεδομένα, που πιθανότατα θα ξαναχρειαστούμε, τα φυλάγουμε οπωσδήποτε σε αρχείο.***

9.2 Συνηθισμένες Δουλειές με Πίνακες

Ας δούμε τώρα μερικές πολύ συνηθισμένες δουλειές που γίνονται με πίνακες. Για τα παρακάτω κομμάτια προγράμματος υποθέτουμε ότι έχουμε δηλώσει:

```
const int N(...);
double x[N];
```

Φυσικά, ο τύπος μπορεί να μην είναι `double`, αλλά `int`, `char` κλπ.

Πριν προχωρήσουμε να τονίσουμε το εξής: Κατ' αρχήν, μια μεταβλητή που προορίζεται να παίζει ρόλο δείκτη θα πρέπει να δηλώνεται με τύπο `unsigned int`. Όπως θα δεις όμως, συνήθως, θα τη δηλώνουμε με τύπο `int`. Από αμέλεια; Και από αμέλεια, αλλά όπως θα δεις σε κάποια από τα παραδείγματα, μερικές φορές δίνουμε σε τέτοιες μεταβλητές κάποια αρνητική τιμή=φρουρό.

9.2.1 Εισαγωγή Στοιχείων

Στο παράδειγμα της προηγούμενης παραγράφου είδαμε πώς διαβάζουμε τις τιμές των στοιχείων ενός πίνακα από το πληκτρολόγιο. Το ξαναγράφουμε λίγο αλλαγμένο και χωρίς την άθροιση:

```
for ( m = 0; m <= N-1; m = m+1 )
{
   cout << "Δώσε το " << m << "ο στοιχείο: ";
   cin >> x[m];
} // for
```

Με αυτόν τον τρόπο το πρόγραμμα παίρνει τις τιμές όλων των στοιχείων με τη σειρά, καθοδηγώντας το χρήστη. Παράδειγμα εκτέλεσης:

```
Δώσε το 0ο στοιχείο: 5.5
Δώσε το 1ο στοιχείο: 6.3
Δώσε το 2ο στοιχείο: 4
```

. . .

Ας πούμε τώρα ότι ο χρήστης δεν θέλει να δώσει όλα τα στοιχεία αλλά θέλει να σταματήσει με φρουρό (9999). Τι κάνουμε στην περίπτωση αυτή;

```
m = 0;  cin >> x[m];
while ( x[m] != 9999 && m < N-1 )
{
   m = m + 1;  cin >> x[m];
} // while
```

Όπως βλέπεις, η *m* ξεκινάει από 0 και αυξάνεται κατά 1 όσο έχουμε `m < N-1`. Άρα αποκλείεται να πάρει τιμή μεγαλύτερη από *N*–1. Όταν τελειώσει η εκτέλεση της **while** θα έχουμε: `x[m] == 9999 || m >= N-1`.

- Αν έχουμε `x[m] == 9999` τότε στο x[*m*] έχουμε τον φρουρό, που δεν είναι τιμή προς επεξεργασία. Άρα θα έχουν διαβαστεί *m* τιμές στα x[0] .. x[*m*–1].
- Αν έχουμε *m* `>=` *N*–1 τότε θα έχουμε για την ακρίβεια *m* `==` *N*–1 (αφού η *m* παίρνει πρώτη τιμή 0 και αυξάνεται κατά 1 κάθε φορά). Στην περίπτωση αυτή θα έχουν διαβαστεί ήδη *N* (= *m* + 1) τιμές στα x[0] .. x[*N*–1].

Έτσι, αν θέλουμε να έχουμε το πλήθος των τιμών που διαβάστηκαν στη μεταβλητή *count*, θα πρέπει να βάλουμε:

```
if ( x[m] == 9999 )  count = m;
                else  count = m + 1;
```

Αν θέλουμε να δώσουμε στον χρήστη το δικαίωμα να δίνει και τον δείκτη του στοιχείου θα πρέπει να σκεφτούμε διαφορετικά:

```
cout << "Δώσε τη θέση του στοιχείου: ";  cin >> m;
cout << "Δώσε το " << m << "ο στοιχείο: ";  cin >> x[m];
```

Τώρα όμως πρέπει να σκεφτούμε και άλλα δύο πράγματα:

- Μπορεί να πάρουμε παράνομη τιμή του δείκτη –στην περίπτωσή μας μικρότερη από 0 ή μεγαλύτερη από *N*–1. Θα πρέπει λοιπόν να ελέγχουμε την τιμή του *m* πριν διαβάσουμε το x[*m*].
- Δεν είναι καθόλου σίγουρο ότι ο χρήστης θα δώσει τιμές για όλα τα στοιχεία. Δεν μπορούμε λοιπόν να δουλεύουμε με τη **for**. Θα πρέπει να δουλεύουμε μάλλον με φρουρό (π.χ. -1 ή κάποιον άλλον αρνητικό στο *m*).

```
cout << "Δώσε τη θέση του στοιχείου (0.." << (N-1)
     << ") (-1 για ΤΕΛΟΣ): ";
cin >> m;
while ( m != -1 )
{
   if ( 0<= m && m <= N-1 )
   {
      cout << "Δώσε το " << m << "ο στοιχείο: ";
      cin >> x[m];
   }
   else
      cout << "*** Λάθος θέση ***" << endl;
   cout << "Δώσε τη θέση του στοιχείου (0.." << (N-1)
        << ") (-1 για ΤΕΛΟΣ): ";
   cin >> m;
} // while
```

Παράδειγμα εκτέλεσης:

```
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 7
Δώσε το 7ο στοιχείο: -15.7
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 3
Δώσε το 3ο στοιχείο: -3
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 11
*** Λάθος θέση ***
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 1
Δώσε το 1ο στοιχείο: 6.3
```

. . .

Παρατήρηση: ▶

Αν γράφεις ένα «πραγματικό» πρόγραμμα θα πρέπει να προσέχεις αυτά τα «7ο στοιχείο» ή «στοιχείο στη θέση 7» που είναι σίγουρο ότι θα προκαλέσουν σύγχυση και λάθη (το πρώτο στοιχείο βρίσκεται στη θέση 0;!). Αν ο χρήστης είναι κάποιος που ξέρει προγραμματισμό μπορείς να αναφερθείς στο «*στοιχείο με δείκτη* 2»· αν δεν ξέρει, θα πρέπει να βρεις τη γλώσσα που καταλαβαίνει και να του μιλήσεις με αυτήν.◀

Μερικές φορές, όταν ο πίνακας είναι μικρός, μπορεί να θελήσεις να διαβάσεις όλα τα στοιχεία του από μια γραμμή. Αυτό είναι απλό: από τον 1ο τρόπο αφαιρούμε τα μηνύματα:

```
for ( m = 0; m <= N-1; m = m+1 )  cin >> x[m];
```

Σε αυτό μπορείς να απαντήσεις –αν π.χ. $N = 5$– με τη γραμμή:

```
5.5   6.3  4.0   -3  5.1<Enter>
```

9.2.2 Εισαγωγή Στοιχείων από Αρχείο

Τα παραπάνω ισχύουν και για εισαγωγή τιμών από αρχείο. Αλλά:

- Πρέπει να προσθέσουμε ελέγχους για τέλος αρχείου.
- Δεν έχουν νόημα τα μηνύματα προς το χρήστη.

Έστω ότι έχουμε $N = 9$, το ρεύμα:

```
ifstream t( "text1.dta" );
```

και το αρχείο text1.dta με περιεχόμενο:

```
-7   -15      8
    14       33
-8  16   114
    375
```

Αν είναι σίγουρο ότι στο αρχείο υπάρχουν N τιμές, μπορείς να διαβάσεις έτσι:

```
for ( m = 0; m <= N-1; m = m+1 )
{
   t >> x[m];
} // for
```

Αλλά, επειδή οι τιμές μπορεί να είναι λιγότερες ή περισσότερες, πιο σίγουρος είναι ο παρακάτω τρόπος (αντικαταστήσαμε τον έλεγχο του φρουρού με τη `!t.eof()`):

```
m = 0;  t >> x[m];
while ( !t.eof() && m < N-1 )
{
   m = m + 1;  t >> x[m];
} // while
if ( t.eof() )  count = m;
          else  count = m+1;
```

Ας πούμε τώρα ότι $N = 9$ και το αρχείο arrval.txt έχει τα εξής:

```
7   -15.7
3    -3
8     3.75
2     4.0
6     0
11    6.3
4     5.1
5   -13
0     5.5
```

Σε κάθε γραμμή, ο πρώτος αριθμός δείχνει τον δείκτη στον πίνακα και η δεύτερη την τιμή του αντίστοιχου στοιχείου. Διαβάζουμε τις τιμές των στοιχείων, με το 2ο τρόπο, ως εξής:

```
ifstream t( "arrval.txt" );
double y;
int    rNum;
```

```
rNum = 0;
t >> m;
while ( !t.eof() )
{
   rNum = rNum + 1;
   if ( 0<= m && m <= N-1 )
      t >> x[m];
   else
   {
      cout << "Λάθος τιμή δείκτη στη γραμμή " << rNum << endl;
      t >> y;
   }
   t >> m;
} // while
t.close();
```

Αυτές οι εντολές θα δώσουν τιμές σε όλα τα στοιχεία του πίνακα εκτός από το x[1]. Και θα μας δώσουν και ένα μήνυμα λάθους:

```
Λάθος τιμή δείκτη στη γραμμή 6
```

διότι εκεί δίνεται τιμή δείκτη 11.

9.2.3 Γράψιμο Στοιχείων

Το γράψιμο είναι πιο απλό. Ας πούμε ότι θέλουμε να γράψουμε τα στοιχεία του x σε μια γραμμή της οθόνης. Δίνουμε:

```
for ( m = 0; m <= N-1; m = m+1 )  cout << x[m] << "  ";
cout << endl;
```

και παίρνουμε:

```
5.5  6.3  4  -3  5.1  -13  0  -15.7  3.75
```

Καταλαβαίνεις βέβαια ότι το `<< "  "` είναι απαραίτητο για να διαχωρίζονται οι τιμές. Να τι θα βγει αν το παραλείψεις:

```
5.56.34-35.1-130-15.73.75
```

Αν έχεις το *t* (*ofstream*) συνδεμένο με κάποιο αρχείο-κείμενο τότε οι:

```
for ( m = 0; m <= N-1; m = m+1 )  t << x[m] << "  ";
t << endl;
```

γράφουν τις τιμές σε μια γραμμή του αρχείου.

Αν θέλεις να βάζεις μια τιμή σε κάθε γραμμή θα πρέπει να δίνεις `endl` για κάθε στοιχείο:

```
for ( m = 0; m <= N-1; m = m+1 )  cout << x[m] << endl;
```

και για αρχείο:

```
for ( m = 0; m <= N-1; m = m+1 )  t << x[m] << endl;
```

Δες ακόμη πώς γράψαμε τις τιμές των στοιχείων του x στο παράδ. της §9.2.

Ας πούμε όμως ότι έχουμε:

```
const int N( 50 );
int z[N];
```

και θέλεις να τυπώσεις τον πίνακα z όπως φαίνεται στο Σχ. 9-2. Πώς γίνεται αυτό;

Όπως φαίνεται έχουμε να τυπώσουμε 10 γραμμές, αριθμημένες (από την 1η στήλη) από 0 μέχρι 9. Αυτό γίνεται ως εξής:

```
for ( r = 0; r <= 9; r =r + 1 )
{
   Γράψε τη γραμμή r
} // for (r = ...
```

Τώρα ας δούμε πώς θα γράψουμε τη γραμμή *r*. Ας πάρουμε για παράδειγμα τις γραμμές 0 και 1· τι περιέχουν; Τα:

```
 0   35     10    9     20  872     30  232     40  347
 1   18     11   34     21    0     31  667     41   61
 2   15     12   21     22   23     32  139     42  753
 3   80     13   57     23  -34     33  -45     43   73
 4   10     14  239     24  -32     34   -9     44    6
 5   40     15  909     25   56     35  -89     45  -37
 6   23     16  213     26  787     36   34     46   43
 7  789     17  576     27  146     37  576     47  -99
 8  563     18  903     28  589     38  122     48  344
 9    1     19  239     29  568     39   99     49  572
```

Σχ. 9-2 Εκτύπωση των τιμών των στοιχείων του πίνακα z. Πριν από κάθε τιμή γράφεται ο δείκτης. Π.χ. 0 35 σημαίνει ότι το στοιχείο z[0] έχει τιμη 35.

0, z[0], 10, z[10], 20, z[20], 30, z[30], 40, z[40] (γραμμή 0)
1, z[1], 11, z[11], 21, z[21], 31, z[31], 41, z[41] (γραμμή 1)
Καταλαβαίνεις λοιπόν ότι μπορούμε να γράψουμε τη γραμμή r ως εξής:

```
    cout << r << z[r] << (r+10) << z[r+10] << (r+20) << z[r+20]
         << (r+30) << z[r+30] << (r+40) << z[r+40] << endl;
```

(φυσικά θα πρέπει να βάλουμε και κενά για να διαχωρίζονται οι τιμές.)

Να λοιπόν η λύση στο πρόβλημά μας:

```
for ( r = 0; r <= 9; r = r + 1 )
{
   cout << r << z[r] << (r+10) << z[r+10] << (r+20) << z[r+20]
        << (r+30) << z[r+30] << (r+40) << z[r+40] << endl;
} // for (r =…
```

Αλλά μπορούμε να τη γράψουμε πιο κομψά: Η γραμμή δεν βγαίνει με **for**; Για δες αυτήν[1]:

```
    for ( c = 0; c <= 40; c = c + 10 )
        cout << (r + c) << z[r+c];
```

Να ολόκληρο το κομμάτι μαζί με τον καθορισμό πλάτους πεδίου για να ξεχωρίζουν οι τιμές μεταξύ τους:

```
for (r = 0; r <= 9; r = r + 1)
{
   for (c = 0; c <= 40; c = c + 10)
   {
      cout.width(7);  cout << (r + c);
      cout.width(5);  cout << z[r+c];
   } // for (c =...
   cout << endl;
} // for (r =...
```

9.2.4 Απλοί Υπολογισμοί

Οι υπολογισμοί αθροίσματος και γινομένου στοιχείων πίνακα γίνονται όπως ξέρουμε:

```
sum = 0;
for ( m = 0; m <= N-1; m = m + 1 )
   sum = sum + x[m];// άθροισμα
```

[1] Άλλοι προτιμούν να γράψουν το εξής:

```
for (r = 0; r <= 9; r = r + 1)
{
   for (c = 0; c <= 4; c = c + 1)
   {
      cout.width(7);  cout << (r + c*10);
      cout.width(5);  cout << z[r+c*10];
   } // for (c =...
   cout << endl;
} // for (r =...
```

Έτσι έχεις το πλεονέκτημα το *c* να έχει πάντοτε ως τιμή τον αριθμό της στήλης (0 .. 4).

και

```
product = 1;
for ( m = 0; m <= N-1; m = m + 1 )
   product = product * x[m];// γινόμενο
```

Για να βρούμε το μέγιστο (ελάχιστο) δεν χρειάζεται να καταφύγουμε στην τεχνική της απίθανα μικρής (μεγάλης) αρχικής τιμής: Θεωρούμε αρχικώς ότι μέγιστη (ελάχιστη) είναι η τιμή του στοιχείου με δείκτη 0:

```
maxNdx = 0;  xMax = x[maxNdx];
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] > xMax )
   {  maxNdx = m;  xMax = x[m];  }
} // for (m …
```

Στη *xMax* κρατάμε τη μέγιστη από τις τιμές των στοιχείων του πίνακα και στη *maxNdx* τον δείκτη του στοιχείου στον πίνακα. Πρόσεξε όμως το εξής: το βασικό είναι να βρούμε τον *δείκτη* του μεγίστου (*maxNdx*). Αν τον ξέρουμε, αφού έχουμε όλες τις τιμές στη μνήμη, παίρνουμε και τη μέγιστη τιμή:

```
maxNdx = 0;
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] > x[maxNdx] )  maxNdx = m;
} // for (m …
xMax = x[maxNdx];
```

Παρομοίως δουλεύουμε και για το ελάχιστο:

```
minNdx = 0;
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] < x[minNdx] )  minNdx = m;
} // for (m …
xmin = x[minNdx];
```

9.3 Παράμετρος – Πίνακας

Κάτι που θα σου είναι πολύ χρήσιμο είναι το να γράφεις συναρτήσεις με παραμέτρους - πίνακες.

Ας δούμε πώς γίνεται αυτό με ένα παράδειγμα. Η:

```
double vectorSum( double x[], int n )
{
   int m;
   double sum( 0 );

   for ( m = 0; m <= n-1; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

υπολογίζει και επιστρέφει το άθροισμα των τιμών των στοιχείων ενός πίνακα με στοιχεία τύπου `double`.

Θέλήσαμε να κάνουμε τη συνάρτησή μας γενική, ώστε να δουλεύει με οποιονδήποτε πίνακα με στοιχεία τύπου `double`. Βάλαμε τον x ως παράμετρο στη συνάρτηση *αλλά δεν βάλαμε πλήθος στοιχείων*. Το πλήθος *n* των στοιχείων του πίνακα το περνάμε με άλλη παράμετρο.

Με το ίδιο τρόπο μπορούμε να γράψουμε και μια συνάρτηση που θα επιστρέφει τον δείκτη του στοιχείου με τη μέγιστη από τις τιμές ενός πίνακα.

```
int maxNdx( int x[], int n )
{
   int m;
```

```
    int mxp( 0 );

    for ( m = 1; m <= n-1; m = m + 1 )
    {
       if (x[m] > x[mxp])  mxp = m;
    } // for (m ...
    return mxp;
} // maxNdx
```

Ας πούμε ότι σε ένα πρόγραμμα έχουμε:

```
const int N( 9 ), Nd2( N/2 );

double x[N] = { 5.5, 6.3, 4, -3, 5.1, -13, 0, -15.7, 3.75 },
       u[Nd2] = { 5.5, 4, 5.1, 0 };
int ix[N] = { 5, 6, 4, -3, 1, -13, 0, -15, 3 },
    iu[Nd2] = { 5, 4, 1, 0 };
```

και δίνουμε:

```
cout << " Σx: " << vectorSum( x, N ) << endl;
cout << " Σu: " << vectorSum( u, Nd2 ) << endl;

cout << " max(ix): " << ix[maxNdx(ix, N)]
     << " στη θεση: " << maxNdx( ix, N ) << endl;
cout << " max(iu): " << iu[maxNdx(iu, Nd2)]
     << " στη θεση: " << maxNdx( iu, Nd2 ) << endl;
```

Αποτέλεσμα:

```
Σx: -7.05
Σu: 14.6
max(ix): 6 στη θεση: 1
max(iu): 5 στη θεση: 0
```

Πρόσεξε τις κλήσεις των συναρτήσεων: Παίρνουμε το άθροισμα των τιμών των στοιχείων

- του x με τη `vectorSum(x, N)` και
- του u με τη `vectorSum(u, Nd2)`.

Ακόμη:

- Η `maxNdx(ix, N)` μας δίνει τον *δείκτη* του μέγιστου στοιχείου του ix ενώ παίρνουμε την *τιμή* αυτού του στοιχείου με την `ix[maxNdx(ix, N)]`.
- Για τον iu δείκτης μέγιστου στοιχείου: `maxNdx(iu, Nd2)` και *τιμή* μέγιστου στοιχείου: `iu[maxNdx(iu, Nd2)]`.

Αλλά, δυστυχώς, η *vectorSum* δέχεται μόνο πίνακες με στοιχεία τύπου `double`. Αν θέλουμε το άθροισμα των στοιχείων του ix θα πρέπει να γράψουμε άλλη συνάρτηση. Παρομοίως, η *maxNdx* δέχεται πίνακα τύπου `int` μόνον. Αργότερα θα δούμε ότι μπορούμε να βάλουμε τον μεταγλωττιστή να γράφει τις συναρτήσεις που μας ενδιαφέρουν.

Και γιατί βάλαμε τον τύπο της *maxNdx* `int` και όχι `unsigned int`; Αυτό θα ήταν το σωστό, αλλά διάβασε αυτά που ακολουθούν.

Μερικές φορές μας ενδιαφέρει να κάνουμε μια επεξεργασία σε ένα κομμάτι του πίνακα μόνον· ας πούμε να βρούμε το άθροισμα των πρώτων πέντε στοιχείων του x. Θα μπορούσαμε να καλέσουμε τη *vectorSum* ως εξής:

```
q = vectorSum( x, 5 );
```

Και αν θέλουμε το άθροισμα των πέντε τελευταίων στοιχείων; Θα μάθεις αργότερα έναν τρόπο να χρησιμοποιείς τη *vectorSum* και για την περίπτωση αυτή.

Πάντως αυτές οι χρήσεις έχουν ένα πρόβλημα: μπερδεύουν τη δομή του πίνακα με την περιοχή επεξεργασίας, Αυτό μπορεί να οδηγήσει σε μερικά πολύ «δύσκολα» προγραμματιστικά λάθη. Μέχρι να γίνεις μεγάλος/η προγραμματιστής/τρια καλύτερα να διαχωρίζεις αυτά τα στοιχεία στη συνάρτηση. Π.χ.:

```
double vectorSum( double x[], int n, int from, int upto )
```

```
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

Φυσικά, θα πρέπει να έχουμε:

$$0 \le from \le upto \le n-1$$

Και αν δεν ισχύει η συνθήκη τι κάνουμε; Κατ' αρχήν θα πρέπει να καλέσουμε την *exit()*. Επειδή όμως δεν είναι σπάνιο να καλούμε μια συνάρτηση σαν αυτήν με $n == 0$ ή με $upto < from$ και να περιμένουμε από τη συνάρτηση τιμή 0, θα γράψουμε:

```
   if ( 0 <= from && from <= upto && upto < n )
   {
      for ( m = from; m <= upto; m = m + 1 ) sum = sum + x[m];
   }
```

Παρομοίως γράφουμε:

```
int maxNdx( int x[], int n, int from, int upto )
{
   int m;
   int mxp;

   if ( from < 0 || upto < from || n <= upto )
      mxp = -1;
   else // 0 <= from <= upto <= n-1
   {
      mxp = from;
      for ( m = from+1; m <= upto; m = m + 1 )
      {
         if (x[m] > x[mxp])  mxp = m;
      } // for (m ...
   }
   return mxp;
} // maxNdx
```

Όπως βλέπεις και εδώ αποφεύγουμε να καλέσουμε την *exit* αλλά εδώ έχουμε τη δυνατότητα –όταν υπάρχει πρόβλημα με την περιοχή επεξεργασίας– να επιστρέψουμε μια τιμή ("`-1`") που δεν μπορεί να είναι δείκτης στοιχείου πίνακα.[2]

Οι παράμετροι πίνακες έχουν μια σημαντική διαφορά από τις παραμέτρους τιμής: *Η τυπική παράμετρος-πίνακας δεν είναι αντίγραφο της πραγματικής παραμέτρου· είναι το ίδιο αντικείμενο*. Γι' αυτό, προσοχή!

- ***Αν αλλάξεις τιμές στοιχείων παράμετρου-πίνακα μέσα στη συνάρτηση η αλλαγή περναει στη συνάρτηση που έκανε την κλήση.***

Παράδειγμα

Αλλάζουμε τη *vectorSum* ως εξής:

```
double vectorSum( double x[], int n, int from, int upto )
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )  sum = sum + x[m];

   for ( m = 0; m <= n-1; m = m + 1 )  x[m] = 0;

   return sum;
```

[2] Καταλαβαίνεις τώρα γιατί προτιμήσαμε να βάλουμε τύπο αποτελέσματος της συνάρτησης `int` και όχι `unsigned int`.

```
} // vectorSum
```

Δηλαδή, αφού υπολογίσουμε το άθροισμα των στοιχείων, αλλάζουμε τις τιμές τους σε "**0**". Καλούμε τη συνάρτηση και γράφουμε τις τιμές των στοιχείων πριν και μετά την κλήση:

```
cout << " x:  ";
for (m = 0; m <= N-1; m = m + 1)  cout << x[m] << "  ";
cout << endl;

cout << " Σx: " << vectorSum( x, N, 0, N-1 ) << endl;

cout << " x:  ";
for (m = 0; m <= N-1; m = m + 1)  cout << x[m] << "  ";
cout << endl;
```

Αποτέλεσμα:

```
x:  5.5  6.3  4  -3  5.1  -13  0  -15.7  3.75
Σx: -7.05
x:  0  0  0  0  0  0  0  0  0
```

Η C++ σου δίνει ένα εργαλείο για να προστατευθείς από αλλαγές στοιχείων του πίνακα κατά λάθος. Μπορείς να δηλώνεις τις παραμέτρους σου **const** (σταθερές). Αν στο παραπάνω παράδειγμα δηλώσεις:

```
double vectorSum( const double x[], int n, int from, int upto )
{ . . . }
int maxNdx( const int x[], int n, int from, int upto )
{ . . . }
```

τότε αν ο *μεταγλωττιστής* βρει μέσα στο σώμα της συνάρτησης εντολή που να αλλάζει τιμή στοιχείου του x θα βγάλει λάθος: «Cannot modify a const object» (δεν μπορείς να αλλάξεις ένα αντικείμενο **const**).

Παρατήρηση: ▶

Όταν γράφουμε μια συνάρτηση αναδρομικώς μας *είναι απαραίτητη η περιοχή επεξεργασίας* η οποία περιορίζεται σε κάθε αναδρομική κλήση. Δίνουμε στη συνέχεια τις δύο συναρτήσεις σε αναδρομική μορφή:

```
double rVectorSum( const double x[], int n, int from, int upto)
{
   double sum( 0 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      sum = x[upto] + rVectorSum( x, n, from, upto-1 );
   }
   return sum;
} // rVectorSum
```

Σε κάθε αναδρομική κλήση μειώνεται κατά 1 η τιμή της *upto* μέχρι που να γίνει κλήση με *upto* < *from* και η *sum* μένει με το 0 παίρνει αρχικώς.

```
int rMaxNdx( const int x[], int n, int from, int upto )
{
   int mxp;

   if ( from < 0 || upto < from || n <= upto )
      mxp = -1;
   else // 0 <= from <= upto <= n-1
   {
      if ( from == upto )
         mxp = from;
      else
      {
         mxp = rMaxNdx( x, n, from+1, upto );
         if ( x[from] > x[mxp])  mxp = from;
      }
   }
```

```
      return mxp;
   } // rMaxNdx
```

Εδώ ο περιορισμός της περιοχής επεξεργασίας γίνεται με αύξηση της τιμής της *from*. ◀

9.4 Δύο Παραδείγματα με Αριθμούς

Στην παράγραφο αυτή θα δούμε δυο αριθμητικά παραδείγματα, από τα πιο χαρακτηριστικά για πίνακες. Είναι πολύ πιθανό να σου φανούν χρήσιμα σε προγράμματα που θα γράψεις για άλλα ενδιαφέροντα ή άλλες υποχρεώσεις σου.

Πριν προχωρήσουμε όμως να σου τονίσουμε κάτι, αν δεν το έχεις καταλάβει ήδη. Στα προγράμματα με πίνακες τα στοιχεία εισόδου είναι συνήθως πολλά. Μέχρι να κάνεις ένα πρόγραμμα να δουλέψει –μέχρι να διορθώσεις τα διάφορα σημαντικά λάθη– θα χρειαστεί συνήθως να πληκτρολογήσεις αρκετές φορές τα ίδια στοιχεία, πράγμα ενοχλητικό και αντιπαραγωγικό! Μια καλή λύση είναι η εξής: Γράψε τα στοιχεία με τα οποία δοκιμάζεις το πρόγραμμά σου σε ένα αρχείο text. Γράψε το πρόγραμμά σου έτσι που να μην διαβάζει από το πληκτρολόγιο, αλλά από το αρχείο.

Έτσι είναι γραμμένα τα παραδείγματα που ακολουθούν.

Παράδειγμα 1 - Τιμή Πολυωνύμου (αλγόριθμος του Horner)

Ένα πολύ συνηθισμένο πρόβλημα, σε αριθμητικά προγράμματα, είναι ο υπολογισμός της τιμής πολυωνύμου:

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m = \sum_{k=0}^{m} a_k x^k$$

όταν δίνονται οι συντελεστές (a_k, $k = 0..m$) και η τιμή της x.

Θέλουμε μια συνάρτηση που θα παίρνει ως παραμέτρους, τους συντελεστές και το x και θα επιστρέφει ως τιμή το $p(x)$.

Οι συντελεστές θα πρέπει να αποθηκευτούν σε ένα μονοδιάστατο πίνακα τύπου `double` με $m+1$ στοιχεία ($0 .. m$). Η συνάρτηση που γράφουμε είναι αρκετά γενική:

```
   double p1( const double a[], int m, double x )
```

Για να δοκιμάσουμε τη συνάρτηση που θα γράψουμε, ετοιμάζουμε το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

   double p1( const double a[], int m, double x );

int main()
{
   const int N( 20 );

   double   a[N];    // συντελεστές πολυωνύμου
   int      b;       // βαθμός πολυωνύμου
   double   x, y;
   int      k, pa;
   ifstream syntFl;  // από το αρχείο Text με τους συντελεστές

   syntFl.open( "syntFl.txt" );
   syntFl >> b;
   if ( syntFl.eof() )
   {
      syntFl.close();
      cout << "αρχειο συντελεστών άδειο" << endl;
   }
   else if ( b < 0 || b > N-1 )
```

```
   {
      syntFl.close();
      cout << "λάθος βαθμός πολυωνύμου" << endl;
   }
   else   // θέλουμε να διαβάσουμε b+1 συντελεστές
   {      // στις θεσεις a[0]..a[b]
      k = 0;  syntFl >> a[k];
      while ( !syntFl.eof() && k <= b-1 )
      {
         k = k + 1;  syntFl >> a[k];
      } // while
      if ( syntFl.eof() )  pa = k;
                     else  pa = k + 1;
      syntFl.close();
   // διαβάσαμε pa συνετελεστές
      if ( pa < b + 1 )
         cout << "Διάβασα " << pa << " τιμές" << endl;
      else    // όλα εντάξει
      {
         cout << " ΣΥΝΤΕΛΕΣΤΕΣ ΤΟΥ ΠΟΛΥΩΝΥΜΟΥ :" << endl;
         for ( k = 0; k <= b; k = k+1 )  cout << a[k] << "  ";
         cout << endl;

         cout << "ΔΩΣΕ  x = ";   cin >> x;
         while ( x != 0 )
         {
            cout << " ΑΠΟΤΕΛΕΣΜΑ:  p(" << x << ")= "
                 << p1( a, b, x ) << endl;
            cout << "ΔΩΣΕ  x = ";   cin >> x;
         } // while
         cout << " ΑΠΟΤΕΛΕΣΜΑ:  p(" << x << ")= "
              << p1( a, b, x ) << endl;
      } // if (pa ...
   } // !syntFl.eof ...
} // main
```

Το syntfl.txt είναι ένα αρχείο text, όπου γράφουμε, με τον κειμενογράφο μας, στην πρώτη γραμμή το βαθμό του πολυωνύμου και στις επόμενες τους συντελεστές. Βλέπεις ότι διαβάζουμε τα στοιχεία του a όπως είδαμε στην §9.1.1 και τους γράφουμε όπως είδαμε στην §9.1.2. Πριν από αυτό, διαβάζουμε και ελέγχουμε την τιμή του βαθμού του πολυωνύμου· θα πρέπει να είναι: $0 \le b \le N - 1$.

Ας πάρουμε για παράδειγμα το παρακάτω πολυώνυμο, 10ου βαθμού, που αποτελείται από 11 όρους (έχει 11 συντελεστές):

$p(x) = -10 + 9.5x + 7.2x^2 + 6.8x^3 - 6.1x^4 + 5.3x^5 - 4.6x^6 + 3.5x^7 + 2.7x^8 - x^9 + 0.8x^{10}$

Στο αρχείο μας θα γράψουμε:

```
10
-10  9.5  7.2  6.8  -6.1  5.3  -4.6  3.5  2.7  -1  0.8
```

Ελέγχουμε τη **while** με φρουρό το "**0**" στη x. Πάντως δεν ξεχνούμε να γράψουμε και το $p(0)$ στο τέλος.

Και τώρα, να γράψουμε τη συνάρτηση, δηλαδή εντολές που υπολογίζουν το άθροισμα που γράψαμε πιο πάνω. Εύκολο:

```
      px = a[0];
      for ( k = 1; k <= m; k = k+1 )
      {
         Υπολόγισε το x^k
         px = px + a[k]*x^k
      }
```

Πως υπολογίζουμε το x^k; Ε, αυτό το ξέρουμε: **pow(x, k)**. Η συνάρτηση που θέλαμε, γράφτηκε εύκολα:

```
   double p1( const double a[], int m, double x )
   {
```

```
      double px;
      int k;

      px = a[0];
      for ( k = 1; k <= m; k = k+1 )
         px = px + a[k]*pow(x, k);
      return px;
   }  // p1
```

Εισάγουμε τη συνάρτηση στο πρόγραμμά μας και να ένα παράδειγμα εκτέλεσης:

```
 ΣΥΝΤΕΛΕΣΤΕΣ ΤΟΥ ΠΟΛΥΩΝΥΜΟΥ :
-10  9.5  7.2  6.8  -6.1  5.3  -4.6  3.5  2.7  -1  0.8
ΔΩΣΕ  x = 0.3562
 ΑΠΟΤΕΛΕΣΜΑ:  p(0.3562)= -5.46928
ΔΩΣΕ  x = -0.45
 ΑΠΟΤΕΛΕΣΜΑ:  p(-0.45)= -13.8303
ΔΩΣΕ  x = 0
 ΑΠΟΤΕΛΕΣΜΑ:  p(0)= -10
```

Και τώρα να *μετρήσουμε*. Ο αλγόριθμός μας κάνει:

m υψώσεις σε δύναμη (pow),

m πολλαπλασιασμούς *a*[*k*] * ...

m προσθέσεις *px* + *a*[*k*] * ...

Δεν μετράμε εκχωρήσεις και τις πράξεις για τον έλεγχο της **for**.

Τι έγινε όμως εδώ; Αγνοήσαμε τελείως το γεγονός ότι: $x^k = x^{k-1} \cdot x$ και για κάθε όρο υπολογίζαμε το x^k από την αρχή. Ασυγχώρητα σπάταλο πρόγραμμα. Η *p2()* διορθώνει αυτήν την αβλεψία:

```
   double p2( const double a[], int m, double x )
   {
      double px, xPow;
      int    k;

      px = a[0];  xPow = 1;
      for ( k = 1; k <= m; k = k+1 ) // εδώ έχουμε xPow = x^(k-1)
      {
         xPow = xPow * x;            // xPow = x^k
         px = px + a[k]*xPow;
      }
      return  px;
   }  // p2
```

Εδώ υπολογίζουμε το x^k με ένα πολλαπλασιασμό από το x^{k-1} και αποφεύγουμε την χρονοβόρα ύψωση σε δύναμη.

Βάλε την *p2()* στο πρόγραμμά σου και, όπου υπάρχει "`p1`" άλλαξε το σε "`p2`". Δοκίμασέ το και θα πάρεις τα ίδια αποτελέσματα που είδαμε παραπάνω. Αλλά τώρα έχουμε:

$2m$ πολλαπλασιασμούς (`xPow * x` και `a[k] * xPow`),

m προσθέσεις (`px + a[k] ...`),

Εδώ δεν έχουμε υψώσεις σε δύναμη και αυτό είναι σημαντικό κέρδος.

Όμως μπορούμε να έχουμε ένα καλύτερο πρόγραμμα! Πώς; Ας δώσουμε ένα παράδειγμα, με το πολυώνυμο τρίτου βαθμού:

$$p(x) = ax^3 + bx^2 + cx + d$$

Υπολογίζοντας την τιμή του με την *p2()*, θα κάνουμε 6 πολλαπλασιασμούς και 3 προσθέσεις.

Ένας άλλος τρόπος να γράψουμε το $p(x)$ είναι ο εξής:

$$p(x) = ((ax + b)x + c)x + d$$

που μας υποδεικνύει έναν άλλο τρόπο υπολογισμού:

$p(x) \leftarrow a;$

$p(x) \leftarrow p(x)x + b;$

$p(x) \leftarrow p(x)x + c;$
$p(x) \leftarrow p(x)x + d;$

Μετράμε: 3 πολλαπλασιασμοί και 3 προσθέσεις. Θρίαμβος! Αυτός ο νέος τρόπος υπολογισμού, που λέγεται **αλγόριθμος του Horner** ή μέθοδος των **φωλιασμένων πολλαπλασιασμών** (nested multiplication) δίνεται στη συνάρτηση ph:

```
// ph -- Υπολογισμός τιμής πολυωνύμου, βαθμού m, με τον
//       αλγόριθμο του Horner.
double ph( const double a[], int m, double x )
{
   double px;
   int    k;

   px = a[m];
   for ( k = m-1; k >= 0; k = k-1 )  px = px*x + a[k];
   return  px;
}  // ph
```

αλλά τώρα, κάνοντας μόνον:

m πολλαπλασιασμούς (`px*x`),
m προσθέσεις (`px*x + a[k]`),

Τι καταφέραμε λοιπόν; Η περιπλοκότητα αυτού του αλγόριθμου είναι τάξης *N*, όπως και του πρώτου. Αλλά, το ότι ελαττώσαμε τον αριθμό των πολλαπλασιασμών –αφού πρώτα-πρώτα διώξαμε αυτούς που χρειάζονται για την ύψωση σε δύναμη– μας δίνει διπλό κέρδος:

- *λιγότερα λάθη στρογγύλευσης*, δηλ. μεγαλύτερη ακρίβεια του αποτελέσματος· τα λάθη στρογγύλευσης είναι από τα σοβαρότερα προβλήματα με τον τύπο `double`,
- εξοικονόμηση χρόνου, γιατί οι πράξεις στον τύπο `double` είναι πιο αργές.

Συνεπώς ο αλγόριθμος του Horner είναι πολύ καλύτερος από τους προηγούμενους. Για υπολογισμό 500 τιμών του παραπάνω πολυωνύμου οι λόγοι των χρόνων των τριών προγραμμάτων είναι: 210:71:42. Όπως βλέπεις, η μεγάλη διαφορά προέρχεται από τον υπολογισμό της δύναμης, περίπου 3:1. Από τον υποδιπλασιασμό του πλήθους των πολλαπλασιασμών είχαμε περίπου υποδιπλασιασμό του χρόνου, περίπου 7:4.

Ποιό είναι το δίδαγμα από τα παραπάνω; Κατ' αρχήν:

♦ ***Υπάρχει πάντα μια καλύτερη λύση!***

Πάντα; Ναι, εκτός αν μπορείς να αποδείξεις το αντίθετο. Και θα ψάχνουμε πάντα για την καλύτερη λύση; Δεν θα πάρουμε υπ' όψη μας ότι την *p1()* την είχαμε έτοιμη στο κεφάλι μας πριν μας τη ζητήσουν; Και βέβαια!

Ας πούμε τα πράγματα με οικονομικούς όρους: Η *p1()* είχε πολύ μικρό **κόστος ανάπτυξης**, ενώ η *ph()* είχε μικρό **κόστος εκμετάλλευσης**[3]. Δεν θα πρέπει να αγνοήσουμε την *p2()*, όπου επιτύχαμε ικανοποιητικό κόστος εκμετάλλευσης χωρίς μεγάλο κόστος ανάπτυξης.

Αν η συνάρτηση, που έχουμε να γράψουμε, πρόκειται να χρησιμοποιηθεί σε κάποιο πρόγραμμα, που υπολογίζει χιλιάδες τιμές και θα χρησιμοποιείται πολύ καιρό, θα πρέπει να κατεβάσουμε το κόστος εκμετάλλευσης με κάθε θυσία (στην περίπτωσή μας: υψηλό κόστος ανάπτυξης). Αν πρόκειται να χρησιμοποιηθεί μια φορά, τότε δεν είναι και τρομερό να χρησιμοποιήσουμε κάτι που το γράφουμε πολύ γρήγορα, ακόμα κι αν δεν έχει την "ταχύτητα του φωτός".

Ο καλός προγραμματιστής ξέρει να σταθμίσει αυτούς τους παράγοντες και να δώσει τη *συνολικώς* καλύτερη λύση. Βέβαια, ο καλός προγραμματιστής έχει τις απαραίτητες γνώσεις για να γράψει κατ' ευθείαν την *ph()!*[4]

---

[3] Ένας άλλος παράγοντας, το **κόστος συντήρησης**, δεν παίζει σημαντικό ρόλο στο παράδειγμά μας.

[4] Στην πραγματικότητα την έχει ήδη έτοιμη σε κάποια από τις βιβλιοθήκες προγραμμάτων που έχει.

Παράδειγμα 2 - Στατιστικές

Αν μας δοθούν N αριθμοί x_k, $k = 0,\ldots N-1$, έχουμε τη μέση τιμή τους:

$$\langle \mathbf{x} \rangle = \frac{1}{N}\sum_{k=0}^{N-1} x_k$$

και την τυπική απόκλισή τους: $\sigma = \sqrt{\mathrm{Var}(\mathbf{x})}$, όπου:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{N}\sum_{k=0}^{N-1}\left(x_k - \langle \mathbf{x} \rangle\right)^2 \qquad (A)$$

που υπολογίζεται και ως εξής:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{N}\sum_{k=0}^{N-1} x_k^2 - \langle \mathbf{x} \rangle^2 \qquad (B)$$

Θέλουμε δυο προγράμματα τα οποία θα διαβάζουν (το καθένα) από το αρχείο text stat.dta τους N αριθμούς x_k τύπου `double` και θα βρίσκουν και θα τυπώνουν το $\langle \mathbf{x} \rangle$ και σ. Το πρώτο από τα προγράμματα θα χρησιμοποιεί τον τύπο (A) για το σ, ενώ το δεύτερο τον τύπο (B). Και τα δύο προγράμματα θα πρέπει να κάνουν τη μέγιστη δυνατή οικονομία σε μνήμη και υπολογιστικό χρόνο. Ποιό από τα δυό προγράμματα είναι καλύτερο και γιατί;

1η λύση: Έχουμε να γράψουμε δύο συναρτήσεις μια για τη μέση τιμή, ας την πούμε *vectorAvrg*, και μια για την τυπική απόκλιση, ας την πούμε stdDev. Και οι δύο θα βγουν από αλλαγές που θα κάνουμε στην *vectorSum* που γράψαμε πιο πάνω. Πρώτα η:

```
double vectorAvrg( const double x[], int n, int from, int upto)
{
   int m;
   double fv( 0 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      for ( m = from; m <= upto; m = m + 1 )  fv = fv + x[m];
      fv = fv/(upto-from+1);
   }
   return fv;
} // vectorAvrg
```

και μετά η:

```
double stdDev( const double x[], int n, int from, int upto )
{
   int m;
   double fv( 0 ), vAvg;

   if ( 0 <= from && from <= upto && upto < n )
   {
      vAvg = vectorAvrg( x, n, from, upto );
      for ( m = from; m <= upto; m = m + 1 )
         fv = fv + pow( x[m]-vAvg, 2 );
      fv = sqrt( fv/(upto-from+1) );
   }
   return fv;
} // stdDev
```

Στον υπολογισμό του $\frac{1}{N}\sum_{k=0}^{N-1}\left(x_k - \langle \mathbf{x} \rangle\right)^2$ θα μπορούσαμε να είχαμε γράψει:

```
   for ( m = 0; m <= n-1; m = m + 1 )
      sum = sum + pow( x[m]-vectorAvrg(x, n, from, upto), 2 );
```

Αλλά έτσι, ζητούμε να κληθεί n φορές η *vectorAvrg* για να υπολογίσει τη μέση τιμή. Όπως τη γράψαμε τώρα, γίνεται μόνο μια κλήση, όταν δίνουμε αρχική τιμή στη *vAvg*. Βέβαια, ένας καλός μεταγλωττιστής θα κάνει αυτήν τη μετατροπή αυτομάτως.

Η `main` δεν έχει δυσκολίες:

```
#include <iostream>
```

```
#include <fstream>
#include <cmath>
using namespace std;

   double vectorAvrg(const double x[], int n, int from, int upto);
   double stdDev( const double x[], int n, int from, int upto );

int main()
{
   const int Nmax = 100;

   double x[Nmax], y;
   int n, k;
   ifstream  t("stat.dta");

   k = 0;  t >> x[k];
   while ( !t.eof() && k <= Nmax-2 )
   {  k = k + 1;  t >> x[k];  }  // while
   if ( t.eof() )  n = k;
             else  n = k + 1;
   t.close();

   cout << " <x> = " << vectorAvrg( x, Nmax, 0, n-1 ) << endl;
   cout << " σ = " << stdDev( x, Nmax, 0, n-1 ) << endl;
} // main
```

2η λύση: Η πρώτη σκέψη είναι να ξαναγράψουμε την *stdDev* και με αυτόν τον τρόπο να κάνουμε $N - 1$ λιγότερες αφαιρέσεις. Αλλά ας το ξανασκεφτούμε. Το πρόβλημά μας λύνεται αν υπολογίσουμε τα $\sum_{k=0}^{N-1} x_k$ και $\sum_{k=0}^{N-1} x_k^2$. Αυτά όμως μπορούμε να τα υπολογίζουμε όταν διαβάζουμε το αρχείο και *δεν χρειαζόμαστε πίνακα*! Βέβαια στην περίπτωση αυτή χάνουμε τις ωραίες μας συναρτήσεις.

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

int main()
{
   double x, vAvg, sigma, sx, sx2;
   int n, k;
   ifstream  t( "stat.dta" );

   sx = 0;  sx2 = 0;
   k = 0;  t >> x;
   while ( !t.eof() )
   {
      k = k + 1;
      sx = sx + x;   sx2 = sx2 + x*x;
      t >> x;
   }  // while
   n = k;
   t.close();
   cout << "Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      vAvg = sx/n;
      sigma = sqrt( sx2/n - vAvg*vAvg );
      cout << " <x> = " << vAvg << endl;
      cout << " σ = " << sigma << endl;
   } // if
} // main
```

Πρόσεξε ότι εδώ διαβάζουμε όσους αριθμούς υπάρχουν, αφού δεν έχουμε πίνακα.

k	v[k]	k	v[k]	k	v[k]	k	v[k]	k	v[k]
1	35	11	9	21	872	31	232	41	347
2	18	12	34	22	0	32	667	42	61
3	15	13	21	23	23	33	139	43	753
4	80	14	57	24	-34	34	-45	44	73
5	10	15	239	25	-32	35	-9	45	6
6	40	16	909	26	56	36	-89	46	-37
7	23	17	213	27	787	37	34	47	43
8	789	18	576	28	146	38	576	48	-99
9	563	19	903	29	589	39	122	49	344
10	1	20	239	30	568	40	99	50	572

Σχ. 9-3 Ο πίνακας που θα χρησιμοποιούμε στις δοκιμές των προγραμμάτων μας. Είναι ίδιος με αυτόν του Σχ. 9-2 με τη διαφορά ότι `v[0] == INT_MIN` και `v[51] == INT_MAX`.

Η 2η λύση είναι και ταχύτερη και οικονομικότερη σε χρήση μνήμης (αφού δεν χρησιμοποιεί πίνακα).

Πάντως οι δύο συναρτήσεις που γράψαμε για την πρώτη λύση είναι χρήσιμες γενικότερα.

9.5 Και Άλλες Συνηθισμένες Δουλειές με Πίνακες

Στην επεξεργασία στοιχείων με τη βοήθεια του ΗΥ συχνά εφαρμόζουμε τις παρακάτω επεξεργασίες πινάκων:

- **Αναζήτηση** ή **ψάξιμο** (searching) για τον εντοπισμό μιας τιμής μέσα σε έναν πίνακα.
- **Ταξινόμηση** (sorting), σε αύξουσα ή φθίνουσα διάταξη, των τιμών που είναι αποθηκευμένες στον πίνακα.
- **Συγχώνευση** (merging) δυό ταξινομημένων πινάκων σε έναν.

Για τις επεξεργασίες αυτές έχουν επινοηθεί αρκετοί αλγόριθμοι. Μερικοί από αυτούς είναι εξαιρετικά ενδιαφέροντες είτε διότι είναι γρήγοροι είτε διότι είναι οικονομικοί σε μνήμη είτε διότι μπορούν να εφαρμοστούν και σε (σειριακά) αρχεία είτε τέλος διότι είναι *όμορφοι*(!)[5]. Στις επόμενες παραγράφους θα δούμε μερικούς αλγορίθμους και τα αντίστοιχα προγράμματά τους για τέτοιες επεξεργασίες. Σίγουρα οι αλγόριθμοι αυτοί δεν είναι οι καλύτεροι, αλλά είναι αρκετά *απλοί* και *κατανοητοί*.

Για τα παραδείγματά μας θα χρησιμοποιήσουμε έναν πίνακα με 50 στοιχεία τύπου `int`. Επειδή σε ορισμένες περιπτώσεις θα μας χρειαστούν και φρουροί θα δηλώσουμε:

```
const int N( 50 );

int v[N+2];
```

και θα αποθηκεύουμε τις τιμές που θέλουμε στις θέσεις από `v[1]` μέχρι `v[N]`. Στη θέση `v[0]` θα έχουμε το −∞ (*INT_MIN*) και στη θέση `v[N+1]` το +∞ (*INT_MAX*). Οι τιμές των στοιχείων (Σχ. 9-3) βρίσκονται στο αρχείο text, με όνομα στο δίσκο pin.txt και θα τις διαβάζουμε, χωρίς ελέγχους, ως εξής:

```
ifstream a;
   :
a.open( "pin.txt" );
for ( k = 1; k <= N; k = k+1 )       // Διάβασε τον πίνακα
   a >> v[k];
a.close();
v[0] = INT_MIN;  v[N+1] = INT_MAX;   // βάλε τους φρουρούς
```

[5] Αισθητική των αλγορίθμων; Μα σίγουρα θα έχει τύχει να δεις μερικές όμορφες μαθηματικές αποδείξεις. Παρόμοια αισθητική υπάρχει κι' εδώ.

Μπορείς να δεις το περιεχόμενο του πίνακα όπως φαίνεται στο Σχ. 9-3 αν στις εντολές που δώσαμε στο τέλος της §9.2.3 αλλάξεις τη **for** που διατρέχει τις γραμμές:

```
for ( c = 0; c <= 4; c = c+1 )  cout << "     k  v[k]";
cout << endl;
for ( r = 1; r <= 10; r = r+1 )   // τύπωσε τη r-οστή γραμμή
{
   for ( c = 0; c <= 40; c = c+10 )
   {
      cout.width(7);  cout << (r + c);
      cout.width(5);  cout << v[r+c];
   } // for ( c = ...
   cout << endl;                 // ...και πήγαινε στη επόμενη
} // for ( r = ...
```

9.5.1 Αναζήτηση στα Στοιχεία Πίνακα

Όταν λέμε «αναζήτηση» εννοούμε τη διαδικασία που δίνει απάντηση στο εξής πρόβλημα:

Έχουμε έναν πίνακα `T v[N]` *και μια τιμή x (τύπου T). Υπάρχει στοιχείο του* `v` *που να έχει τιμή ίση με x και αν ναι ποια η τιμή του δείκτη για το στοιχείο αυτό;*

Παρατήρηση: ▶

Πολύ συχνά, ένας πίνακας χρησιμοποιείται για την παράσταση κάποιου συνόλου στο πρόγραμμά μας. Στην περίπτωση αυτή η αναζήτηση δίνει απάντηση στο ερώτημα «*ανήκει η τιμή x στο σύνολο (που υλοποιείται με τον πίνακα) v;*»◀

Εμείς θα δουλέψουμε με τον πίνακα τύπου `int` που είδαμε πιο πριν. Θέλουμε λοιπόν μια:

```
int linSearch( int v[], int n, int from, int upto, int x )
```

που θα ψάχνει στα στοιχεία v`[`*from*`]`, v`[`*from*+1`]`,..., v`[`*upto*`]` να βρει την τιμή *x*. Αν τη βρει, θα επιστρέφει τον δείκτη του στοιχείου ως τιμή της συνάρτησης, αλλιώς θα επιστρέφει τιμή –1. Αν δηλαδή δώσουμε:

```
thesi = linSearch( v, N+2, n1, n2, x );
```

και αν $0 < n1 \le n2 < N + 2$ θα πρέπει:

$$(n1 \le \text{linSearch}(v, N{+}2, n1, n2, x) \le n2 \;\&\&\; v[\text{linSearch}(v, N{+}2, n1, n2, x)] == x)$$
$$|| \;(\text{linSearch}(v, N{+}2, n1, n2, x) == -1 \;\&\&\; (\forall j: n1..n2 \bullet v[j] \;!= x))$$

Τα πράγματα είναι πολύ απλά:

Ξεκίνα από την αρχή (*from*)
Όσο (δεν τελείωσε ο πίνακας) και (δεν το βρήκες) κάνε τα εξής:
{ Προχώρησε στο επόμενο στοιχείο του πίνακα

Μεταφράζουμε:

Ξεκίνα από την αρχή (*from*)	→ `k = from`
δεν τελείωσε ο πίνακας	→ `k <= upto`
δεν το βρήκες	→ `v[k] != x`
Προχώρησε στο επόμενο στοιχείο του πίνακα	→ `k = k+1`

Δηλαδή:

```
k = from;
while ( k < upto && v[k] != x )  k = k+1;
```

Δεν τελειώσαμε όμως: Η εκτέλεση της **while** θα τελειώσει:

- είτε διότι `!(k < upto)` ή αλλιώς `k >= upto` (για την ακρίβεια `k == upto` αφού η τιμή της *k* αυξάνεται κατά 1), φτάσαμε δηλαδή στο τέλος της περιοχής αναζήτησης,
- είτε διότι `!(v[k] != x)` ή αλλιώς `v[k] == x`, δηλαδή βρήκαμε την τιμή που ψάχνουμε στο στοιχείο v`[`*k*`]`.

Χρειάζεται λοιπόν και άλλος ένας έλεγχος, ακριβώς μετά τη **while**:

```
if ( v[k] == x )  fv = k;
```

```
               else  fv = -1;
```

Να λοιπόν ολόκληρη η *linSearch*:

```
   // linSearch -- Ψάχνει στα στοιχεία v[from],... v[upto] να
   //              βρει την τιμή x. Αν τη βρει
   //                 επιστρέφει ως τιμή τον δείκτη του στοιχείου
   //              αλλιώς
   //                 επιστρέφει τιμή -1
   int linSearch( const int v[], int n, int from, int upto, int x)
   {
      int k, fv( -1 );

      if ( 0 <= from && from <= upto && upto < n )
      {
         k = from;
         while ( k < upto && v[k] != x )  k = k+1;
         if ( v[k] == x )  fv = k;
                     else  fv = -1;
      // (from <= fv <= upto && v[fv] == x) ||
      //                (fv == -1 && ( j:from..upto " v[j] != x))
      }
      return fv;
   } // linSearch
```

Αυτή είναι η μέθοδος **γραμμικής αναζήτησης** (linear search). Στη συνέχεια βλέπεις και ένα πρόγραμμα που τη δοκιμάζει:

```
#include <iostream>
#include <fstream>
#include <climits>
#include <cstdlib>
using namespace std;

 int linSearch( const int v[], int n, int from, int upto, int x );

int main()
{
   const int N( 50 );

   int  v[N+2];      // ο πίνακας όπου ψάχνουμε
   int  x;           // για την τιμή της ψάχνουμε
   int  ndx;         // ο δείκτης της, αν τη βρούμε
   int  k;
   ifstream a;

// Διάβασε τον πίνακα και βάλε τους φρουρούς
// όπως παραπάνω

   cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   while ( x != 9999 )
   {
      ndx = linSearch( v, N+2, 1, N, x );
      if ( ndx > 0 )
         cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      else
         cout << " ΔΕΝ ΥΠΑΡΧΕΙ" << endl;
      cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   } // while
} // main
```

Παράδειγμα εκτέλεσης:

```
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 23
 ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 7
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): -121
 ΔΕΝ ΥΠΑΡΧΕΙ
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 6
 ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 45
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 9999
```

Παρατηρήσεις ▶

1. Όταν δώσαμε τιμή της $x = 23$ πήραμε απάντηση 7. Αλλά, αν δεις τον πίνακα στο Σχ. 9-3, το 23 υπάρχει και στη θέση 23. Η διαδικασία αυτή θα σου δώσει λοιπόν *μόνο την πρώτη εμφάνιση της τιμής*. Ο πίνακας του παραδείγματος υλοποιεί ένα **πολυσύνολο** (multiset, bag). Για ένα πολυσύνολο B η ερώτηση "$x \in B$;" δεν έχει και τόσο νόημα· η πιο σωστή ερώτηση είναι «*πόσες φορές υπάρχει η τιμή x στο πολυσύνολο B;*» (άσκ. 9-10)

2. Ειδικώς για τη συνάρτηση αυτή, η βίαιη αντίδραση (`exit(1)`) όταν είμαστε εκτός προδιαγραφών δεν είναι καλή. Πολλές φορές καλούμε μια τέτοια συνάρτηση με *μηδενική περιοχή αναζήτησης* (*from* > *upto*) και αυτό που θέλουμε είναι η απάντηση: *δεν υπάρχει η τιμή*. Γι' αυτό, όταν δεν ισχύει η συνθήκη της **if** η *fv* μένει με την τιμή "`-1`" που σημαίνει ακριβώς αυτό. ◀

Που γίνεται η πολλή δουλειά στην *linSearch()*; Προφανώς στη **while**. Για κάθε επανάληψη γίνονται 2 συγκρίσεις, 1 λογική "`&&`", 1 πρόσθεση και 1 εκχώρηση (`k = k+1`). Αν δεν υπάρχει η τιμή που ψάχνουμε –η χειρότερη περίπτωση για τον αλγοριθμό μας– όλα πολλαπλασιάζονται επί N (για την ακρίβεια επί *upto*−*from*+1).

Αν καταφέρουμε να διώξουμε τη μια σύγκριση διώχνουμε αυτόματα και τη λογική πράξη και ο αλγόριθμός μας επιταχύνεται σημαντικά. Ας δοκιμάσουμε. Και πρώτα-πρώτα: Ποια θα διώξουμε; Μάλλον δεν μπορούμε να διώξουμε την "`v[k] != x`". Γι' αυτήν γράφτηκε ολόκληρη η διαδικασία! Αν διώξουμε την "`k < upto`" πώς θα σταματήσουμε στο τέλος του πίνακα; Θα μπορούσαμε να σταματήσουμε με την άλλη συνθήκη. Αλλά αν η x δεν υπάρχει στον πίνακα; Θα τη βάλουμε εμείς με το ζόρι!

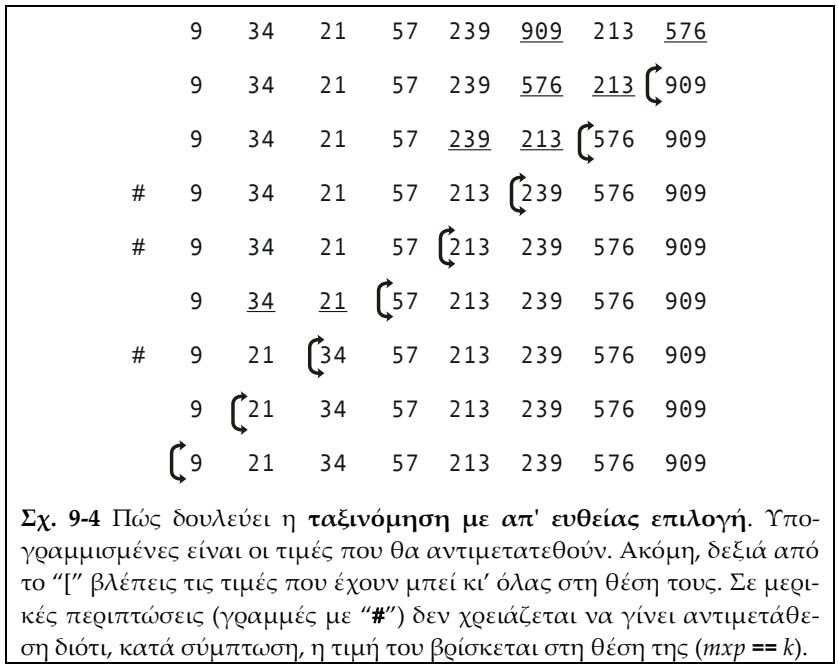
Τώρα μπερδεύτηκες, ε; Λοιπόν, πριν αρχίσουμε την αναζήτηση, θα βάλουμε την x στη θέση `v[`*upto*`+1]`. Όταν η "`while (v[k] != x) k = k+1`" σταματήσει, ελέγχουμε την τιμή της k: Αν $k \le upto$ τότε πραγματικά η x υπάρχει στον πίνακα, αλλιώς, αν $k > upto$ τότε δεν υπάρχει· σταμάτησε από τη x που βάλαμε εμείς στην `v[`*upto*`+1]`. Έξυπνο, έτσι; Αλλά κάτι σου θυμίζει! Τι άλλο; Την **τιμή - φρουρό**. Η τεχνική του φρουρού λύνει πολλά προβλήματα και απλουστεύει τα προγράμματά μας:

```
        save = v[upto+1];    // φύλαξε το v[upto+1]
        v[upto+1] = x;       // φρουρός
        k = from;
        while ( v[k] != x )  k = k+1;
        if ( k <= upto )  fv = k;
                    else  fv = -1;
        v[upto+1] = save;    // όπως ήταν στην αρχή
```

Πρόσεξε ότι πριν βάλουμε το φρουρό (x) στη θέση `v[`*upto* + 1`]` φυλάγουμε την τιμή που υπάρχει εκεί σε μια μεταβλητή (*save*) και αποκαθιστούμε την αρχική κατάσταση μετά το τέλος της αναζήτησης. Αν η *upto* έχει τιμή N δεν υπάρχει πρόβλημα αφού έχουμε προβλέψει μια ακόμη θέση (όπου έχουμε τον φρουρό "+∞").

Αλλά, δυστυχώς, ο μεταγλωττιστής έχει αντιρρήσεις: "`Cannot modify a const object in function linSearch`". Τι συμβαίνει; Φταίει το "`const`" που βάλαμε στην παράμετρο-πίνακα. Αργότερα θα δούμε πώς μπορούμε να το διορθώσουμε. Προς το παρόν θα καταφύγουμε σε «ριζικές θεραπείες»: θα βγάλουμε το "`const`"!

```
int linSearch( int v[], int n, int from, int upto, int x )
{
   int save;
   int k, fv( -1 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      save = v[upto+1];    // φύλαξε το v[upto+1]
      v[upto+1] = x;       // φρουρός
      k = from;
      while ( v[k] != x )  k = k+1;
      if ( k <= upto )  fv = k;
                  else  fv = -1;
```



Σχ. 9-4 Πώς δουλεύει η **ταξινόμηση με απ' ευθείας επιλογή**. Υπογραμμισμένες είναι οι τιμές που θα αντιμετατεθούν. Ακόμη, δεξιά από το "[" βλέπεις τις τιμές που έχουν μπεί κι' όλας στη θέση τους. Σε μερικές περιπτώσεις (γραμμές με "**#**") δεν χρειάζεται να γίνει αντιμετάθεση διότι, κατά σύμπτωση, η τιμή του βρίσκεται στη θέση της (*mxp* == *k*).

```
        v[upto+1] = save;    // όπως ήταν στην αρχή
    // (from <= fv <= upto && v[fv] == x) ||
    //         (fv == -1 && (για κάθε j:from..upto " v[j] != x))
    }
    return fv;
} // linSearch
```

9.5.2 Ταξινόμηση Στοιχείων Πίνακα

Τώρα θέλουμε να ταξινομήσουμε[6] τα στοιχεία του σε αύξουσα διάταξη, δηλ. να έχουμε (στα v[0] και v[*N*+1] έχουμε τα "−∞" και "+∞"):

$$\forall j: 1..n \bullet v[j] \le v[j+1]$$

Ενας απλός τρόπος είναι ο εξής:

Βρες το μέγιστο από τα στοιχεία v[1]...v[*N*] και
αντιμετάθεσε την τιμή του με αυτήν του v[*N*]
Βρες το μέγιστο από τα στοιχεία v[1]...v[*N*−1] και
αντιμετάθεσε την τιμή του με αυτήν του v[*N*−1]
:
Βρες το μέγιστο από τα στοιχεία v[1]...v[2] και
αντιμετάθεσε την τιμή του με αυτήν του v[2]

Αυτή η διαδικασία περιγράφεται και ως εξής:

```
for (k = N; k >= 2; k = k-1)
{
   Βρες το μέγιστο από τα στοιχεία v[1]...v[k] και
   αντιμετάθεσε την τιμή του με αυτήν του v[k]
}
```

Τελειώσαμε! Η "εντολή":

```
   Βρες το μέγιστο από τα στοιχεία v[1]...v[k]
```

μας είναι γνωστή. Χρησιμοποιώντας τη *maxNdx* που είδαμε στην §9.3 μπορούμε να τη γράψουμε ως εξής:

---

[6] Όπως θα δούμε στη συνέχεια, ο πιο συνηθισμένος λόγος για να ταξινομήσουμε τα στοιχεία ενός πίνακα είναι η ευκολία στο ψάξιμό τους είτε με υπολογιστή είτε με το χέρι. Σκέψου, πώς θα έψαχνες τον τηλεφωνικό κατάλογο αν δεν ήταν ταξινομημένος!

```
        mxp = maxNdx( v, N+2, 1, k );
```

Όσο για την "εντολή":

αντιμετάθεσε την τιμή του με αυτήν του v[*k*]

ξέρουμε να τη γράψουμε:

```
        sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
```

Αυτή η μέθοδος λέγεται **ταξινόμηση με απ' ευθείας επιλογή** (straight selection sort). Στο Σχ. 9-4 βλέπεις πως ταξινομείται με τη διαδικασία αυτή ένας πίνακας με 8 στοιχεία.

Το παρακάτω πρόγραμμα:

- διαβάζει τα στοιχεία του v από το pin.txt,
- τα ταξινομεί με τη μέθοδο που είδαμε,
- γράφει τον ταξινομημένο πίνακα στην οθόνη και
- τον φυλάγει στο αρχείο pinsrt.txt.

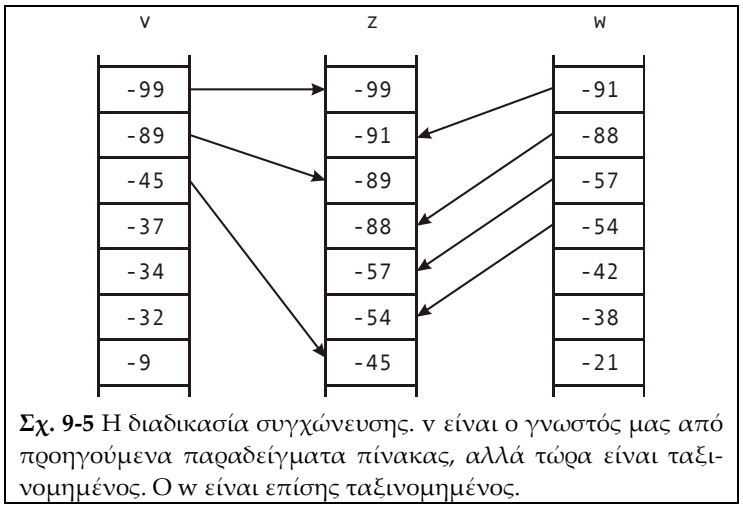
Φυσικά, χρησιμοποιεί τη *maxNdx* που ξέρουμε.

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;

   int maxNdx( int x[], int n, int from, int upto );

int main()
{
   const int N( 50 );

   int  v[N+2], sv;
   int  k, mxp, r, c;
   fstream a;

   a.open( "pin.txt", ios_base::in );
// Διάβασε τον πίνακα και βάλε τους φρουρούς όπως παραπάνω

// ταξινόμηση
   for ( k = N; k >= 2; k = k-1 )
   {
      mxp = maxNdx( v, N+2, 1, k );
      sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
   } // for

// Δείξε τα στοιχεία του ταξινομημένου πίνακα όπως παραπάνω
// φύλαξε τον πίνακα
   a.open( "pinsrt.txt", ios_base::out );
   for ( k = 1; k <= N; k = k+1 )  a << v[k] << endl;
   a.close();
} // main
```

Να τι θα δούμε στην οθόνη:

```
ΣΤΟΙΧΕΙΑ ΤΟΥ ΠΙΝΑΚΑ v ΜΕΤΑ ΤΗΝ ΤΑΞΙΝΟΜΗΣΗ
     k  v[k]     k  v[k]     k  v[k]     k  v[k]     k  v[k]
     1  -99     11    9     21   40     31  146     41  576
     2  -89     12   10     22   43     32  213     42  576
     3  -45     13   15     23   56     33  232     43  589
     4  -37     14   18     24   57     34  239     44  667
     5  -34     15   21     25   61     35  239     45  753
     6  -32     16   23     26   73     36  344     46  787
     7   -9     17   23     27   80     37  347     47  789
     8    0     18   34     28   99     38  563     48  872
     9    1     19   34     29  122     39  568     49  903
    10    6     20   35     30  139     40  572     50  909
```

Ας «μετρήσουμε» λιγάκι αυτή τη μέθοδο. Όπως φαίνεται από τη **for**, θα έχουμε *N*−1 κλήσεις της *maxNdx* και άλλες τόσες αντιμεταθέσεις τιμών του v. Στη *maxNdx* (*from* = 1, *upto*



Σχ. 9-5 Η διαδικασία συγχώνευσης. v είναι ο γνωστός μας από προηγούμενα παραδείγματα πίνακας, αλλά τώρα είναι ταξινομημένος. Ο w είναι επίσης ταξινομημένος.

= *k*), υπάρχει μια άλλη **for** που, για κάθε τιμή της *k*, εκτελεί, *k* – 1 φορές τη σύγκριση v**[***m***]** > v**[***mxp***]**. Αυτή η σύγκριση θα εκτελεσθεί συνολικώς:

$$(N-1) + (N-2) + \ldots + 2 + 1 = \tfrac{1}{2}N(N-1) = \tfrac{1}{2}N^2 - \tfrac{1}{2}N$$

φορές. Για μεγάλα *N*, ο όρος ½*N* γίνεται αμελητέος και ο όρος που κυριαρχεί είναι ο ½N^2. Λέμε ότι ο χρόνος εκτέλεσης του αλγόριθμου αυξάνεται σαν το N^2. Πάντως υπάρχουν και άλλες πράξεις, που πρέπει να εξετασθούν πιο προσεκτικά. Π.χ. πόσες φορές εκτελείται η `mxp = m`; Για πιο πολύπλοκες δομές στοιχείων θα πρέπει να μετρήσουμε τον υπολογιστικό χρόνο για την εκτέλεση της v**[***m***]** > v**[***mxp***]** και ακόμη της αντιμετάθεσης (εκτελείται *N*–1 φορές).

Αυτή η μέθοδος είναι πολύ απλή στη σύλληψη και για μικρές τιμές της *N* είναι ικανοποιητική.

9.5.3 Συγχώνευση Πινάκων

Έστω ότι έχουμε δυό μονοδιάστατους πίνακες v και w, με στοιχεία του ίδιου τύπου, ας πούμε `int`, με *NV* και *NW* στοιχεία αντιστοίχως. Οι πίνακες αυτοί είναι ταξινομημένοι σε αύξουσα τάξη. Θέλουμε να τους συγχωνεύσουμε σε έναν άλλο πίνακα z, με *NZ* (≥ *NV* + *NW*) στοιχεία του ίδιου τύπου, έτσι ώστε ο z να είναι ταξινομημένος επίσης κατ' αύξουσα διάταξη.

Μια πρώτη σκέψη (και η χειρότερη) που θα μπορούσαμε να κάνουμε είναι η εξής: Να βάλουμε στις πρώτες *NV* θέσεις του πίνακα z τα στοιχεία του v, στις επόμενες *NW* θέσεις τα στοιχεία του w και στη συνέχεια να ταξινομήσουμε τον z όπως είδαμε στην προηγούμενη παράγραφο. Με αυτόν τον τρόπο, δεν εκμεταλλευόμαστε το γεγονός ότι οι δύο πίνακες είναι ήδη ταξινομημένοι. Ο υπολογιστικός χρόνος θα είναι ανάλογος του $(NV+NW)^2$, ενώ ήδη έχει καταναλωθεί χρόνος για να ταξινομηθούν οι v και w. Θα πρέπει να ψάξουμε για κάτι καλύτερο.

Για δες την παρακάτω απλή ιδέα:

```
Πάρε την πρώτη τιμή από τον v
Πάρε την πρώτη τιμή από τον w
Ετοιμάσου να γράψεις στην πρώτη θέση του z
while ( δεν έφτασες στο τέλος ούτε του v ούτε του w )
{
  if ( τιμή από τον v < τιμή από τον w )
  {
    Γράψε στον z την τιμή από τον v
    Αντικατάστησε την τιμή από τον v με την επόμενή της
  }
  else   // if τιμή από τον v >= τιμή από τον w
  {
```

```
    Γράψε στον z την τιμή από τον w
    Αντικατάστησε την τιμή από τον wμε την επόμενή της
  }
  Ετοιμάσου να γράψεις στην επόμενη θέση του z
}
```

Στο Σχ. 9-5, βλέπεις πως δουλεύει αυτή η μέθοδος.

Αλλά δεν τελειώσαμε ακόμη: Η εκτέλεση της **while** θα τελειώσει όταν φτάσουμε στο τέλος είτε του v είτε του w. Στη συνέχεια θα πρέπει να αντιγράψουμε στον z τα στοιχεία του άλλου πίνακα, που δεν τελείωσε. Και αν τελειώσουν και οι δυο μαζί; Δεν γίνεται! *Σε κάθε εκτέλεση της περιοχής επανάληψης παίρνουμε ένα στοιχείο από έναν πίνακα*. Θα πρέπει να συμπληρώσουμε τον αλγόριθμό μας με το κομμάτι:

```
Πρόσθεσε στον z τα στοιχεία που περίσσεψαν από τον πίνακα που δεν εξαντλήθηκε
```

Θα γράψουμε ένα πρόγραμμα που θα διαβάζει, όπως ξέρουμε, τις τιμές των στοιχείων του πίνακα v, από το αρχείο pinsrt.txt, όπου τα γράψαμε με το πρόγραμμα της προηγούμενης παραγράφου, μετά την ταξινόμηση. Θα διαβάζει ακόμη, με τον ίδιο τρόπο, τις τιμές των (20) στοιχείων του πίνακα w που βρίσκονται στο αρχείο pinw.dta, ήδη ταξινομημένα. Στη συνέχεια θα συγχωνεύει τα στοιχεία των δύο πινάκων στον z.

Τρεις μεταβλητές *dv*, *dw*, *dz* θα παίζουν ρόλο δεικτών στους v, w, z αντιστοίχως. Οι *dv*, *dw* θα δείχνουν τα στοιχεία των v και w που συγκρίνουμε, ενώ ο *dz* δείχνει το στοιχείο του z όπου θα γίνει η αντιγραφή. Και οι τρεις θα αρχίζουν από 1:

```
   dv = 1;  dw = 1;  dz = 1;
```

Η while που δώσαμε παραπάνω γράφεται:

```
   while ( dv <= NV && dw <= NW )
   {
      if (v[dv] < w[dw])  {  z[dz] = v[dv];  dv = dv + 1;  }
                    else  {  z[dz] = w[dw];  dw = dw + 1;  }
      dz = dz + 1;
   } // while
```

Αν, στο τέλος, τελείωσαν τα στοιχεία του v (`dv > NV`) και έχουν περισσέψει στοιχεία από τον w, τα αντιγράφουμε στον z ως εξής:

```
      while ( dw <= NW )
      {  z[dz] = w[dw];  dw = dw + 1;  dz = dz + 1;  }
```

Πρόσεξε ότι οι αρχικές τιμές των δεικτών *dz* και *dw* γι' αυτήν τη while είναι ακριβώς οι τελικές τιμές τους από την προηγούμενη. Παρόμοια προσθέτουμε στον z και τα στοιχεία του v, αν έχει εξαντληθεί ο w. Μετά από τις πιό πάνω παρατηρήσεις μπορούμε να δώσουμε ολόκληρο το πρόγραμμα.

```
#include <fstream>
using namespace std;

int main()
{
   const int N( 50 ), NV( N ), NW( 20 ), NZ( 70 );

   int  v[N+2], w[NW+2], z[NZ+2];
   int  k;
   int  dv, dw, dz;
   fstream a;

   a.open( "pinsrt.txt", ios_base::in );
   for (k = 1; k <= NV; k = k+1)  a >> v[k];
   a.close();
   a.open( "pinw.txt", ios_base::in);
   for (k = 1; k <= NW; k = k+1)  a >> w[k];
   a.close();

// ΣΥΓΧΩΝΕΥΣΗ
   dv = 1;  dw = 1;  dz = 1;
```

```
   while ( dv <= NV && dw <= NW )
   {
      if (v[dv] < w[dw])  {  z[dz] = v[dv];  dv = dv + 1;  }
                    else  {  z[dz] = w[dw];  dw = dw + 1;  }
      dz = dz + 1;
   } // while
   if ( dv > NV )
      while ( dw <= NW )
      {  z[dz] = w[dw];  dw = dw + 1;  dz = dz + 1;  }
   else
      while ( dv <= NV )
      {  z[dz] = v[dv];  dv = dv + 1;  dz = dz + 1;  }

   a.open( "pinz.txt", ios_base::out );
   for ( k = 1; k <= NZ; k = k+1 )  a << z[k] << endl;
   a.close();
} // main
```

Μπορείς εύκολα να δείς ότι ο χρόνος εκτέλεσης αυτού του αλγόριθμου είναι ανάλογος του *NV*+*NW*.

9.6 Ταχύτερα – Οικονομικότερα – Καλύτερα

Τώρα ας γυρίσουμε για να ξαναδούμε τα προηγούμενα προβλήματά μας.

Πρώτα η αναζήτηση. Αλλάζουμε λίγο το πρόβλημά μας: Έστω π.χ. ότι έχουμε έναν μονοδιάστατο πίνακα, v, με *N* στοιχεία τύπου **int**, *ταξινομημένα κατ' αύξουσα τάξη* και μια τιμή *x* (επίσης τύπου **int**). Θέλουμε έναν αλγόριθμο που θα ψάχνει στα στοιχεία v[*from*], v[*from*+1], ..., v[*upto*] να βρει την τιμή *x*.

Οι δυο παραλλαγές γραμμικής αναζήτησης που είδαμε δουλεύουν μια χαρά. Αλλά παίρνοντας υπόψη μας την ταξινόμηση μπορούμε να τα καταφέρουμε κάπως καλύτερα. Και μάλιστα στη χειρότερη περίπτωση: όταν η τιμή που ψάχνουμε δεν υπάρχει στον πίνακα. Στην περίπτωση αυτήν, οι αλγόριθμοι που είδαμε σαρώνουν ολόκληρον τον πίνακα πριν αποφανθούν.

Αν ο πίνακας είναι ταξινομημένος, δεν χρειάζεται να τον ψάξουμε ολόκληρον. Ας ξεκινήσουμε από τον τηλεφωνικό κατάλογο. Ψάχνουμε το τηλέφωνο του "**ΠΑΠΑΔΗΜΑ**", αλλά μετά το όνομα "**ΠΑΠΑΔΑΝΙΗΛ**" βρίσκουμε το όνομα "**ΠΑΠΑΔΙΑΚΟΣ**". Φυσικά, σταματάμε το ψάξιμο!

Έστω, λοιπόν, ότι v[*from*] ≤ *x* ≤ v[*upto*]. Θα ψάχνουμε μέχρι να βρούμε κάποιο v[*k*] ≥ *x*:

- Αν έχουμε v[*k*] == *x* τότε βρήκαμε την τιμή.
- Αλλιώς, αν v[*k*] > *x*, δεν έχει νόημα να συνεχίσουμε την αναζήτηση αφού όλα τα επόμενα στοιχεία έχουν τιμές μεγαλύτερες από το v[*k*], άρα και από το *x*.

```
      k = from;
      while ( v[k] < x )  // I: ∀j:from-1..k-1 • v[j] < x
         k = k+1;
   // (∀j:0..k-1 • v[j] < x) && (x ≤ v[k])
      if ( v[k] == x )  fv = k;
                  else  fv = -1;
```

Για να τερματίζεται η **while** και στην περίπτωση που *x* > v[*upto*] θα πρέπει να έχουμε φρουρό (+∞) στο v[*upto*+1]. Ο φρουρός (−∞) στο v[*from*−1] χρειάζεται για να έχει νόημα η αναλλοίωτή μας όταν *m* == *from*, αλλά δεν είναι απαραίτητη η φυσική του παρουσία.

Αν θέλεις, μπορείς να ψάξεις και από το τέλος προς την αρχή:

```
      k = upto;
      while ( x < v[k] )  // I: ∀j:k+1..upto+1 • x < v[j]
         k = k-1;
   // (v[k] ≤ x) && (∀j:k+1..upto+1 • x < v[j])
      if ( v[k] == x )  fv = k;
                  else  fv = -1;
```

Στην περίπτωση αυτή, θα πρέπει να έχουμε φρουρό (−∞) στο v[*from*−1] για να τερματιζεται η **while** και στην περίπτωση που *x* < v[*from*]. Εδώ, δεν είναι απαραίτητη η φυσική του παρουσία φρουρού (+∞) στο v[*upto*+1], αλλά πρέπει να υποθέσουμε ότι υπάρχει για να έχει νόημα η αναλλοίωτή μας όταν *m* == *upto*.

Τώρα ας δούμε κάτι καλύτερο.

Έψαξες ποτέ σου τον τηλεφωνικό κατάλογο γραμμικώς; Μάλλον απίθανο. Συνήθως, αν ψάχνεις για τον ΠΑΠΑΔΗΜΑ, ανοίγεις τον κατάλογο εκεί που μαντεύεις ότι είναι το 'Π'. Αν πέσεις στο 'Σ', γυρνάς πιο πίσω. Αν τώρα πέσεις στο 'Ο' ξαναδοκιμάζεις πιο μπροστά κ.ο.κ. Ας προσπαθήσουμε να κάνουμε κάτι παρόμοιο και στον πίνακά μας.

Αρχίζουμε ελέγχοντας το στοιχείο που βρίσκεται στο μέσο του πίνακα, έχει δηλαδή δείκτη *middle* = (*from*+*upto*)/2. Αν v[*middle*] == *x*, βρήκαμε την τιμή που ψάχνουμε, τελειώσαμε. Αν v[*middle*] < *x* τότε, αφού ο πίνακας είναι ταξινομημένος, όλα τα στοιχεία v[*from*],... v[*middle*], είναι σίγουρα < *x*. Θα πρέπει λοιπόν να ψάξουμε στα: v[*middle*+1],... v[*upto*]. Αν βρίσκαμε v[*middle*] > *x* θα έπρεπε να συνεχίζαμε το ψάξιμο στην περιοχή v[*from*],... v[*middle*−1]. Και πώς θα συνεχιστεί το ψάξιμο; Μα με τον ίδιο τρόπο! Αλλά στον μισό πίνακα.

Δες το παράδειγμα στο Σχ. 9-4, όπου σε ένα ταξινομημένο πίνακα 8 στοιχείων ψάχνουμε για την τιμή 100.

Βάζουμε αρχικά, "`from = 1; upto = 8;`" και υπολογίζουμε το *middle* = 4: v[*middle*] = 57. Το 100 –αν υπάρχει– θα βρίσκεται μετά το 57. Συνεχίζουμε με τον ίδιο τρόπο, αφού αλλάξουμε το "`from = middle + 1;`" (= 5). Υπολογίζουμε και πάλι το *middle* = 6. Τώρα έχουμε: v[*middle*] = 239. Το 100 αν υπάρχει, θα υπάρχει πριν από το 239. Τώρα, πριν συνεχίσουμε, αλλάζουμε το "`upto = middle - 1;`" (= 5). Η περιοχή που ψάχνουμε περιορίστηκε σε ένα στοιχείο. Αλλά ας συνεχίσουμε: *middle* = 5, v[*middle*] = 213. Αν υπάρχει το 100, θα υπάρχει πριν από αυτό. Αλλάζουμε λοιπόν το "`upto = middle - 1;`" (= 4). Και εδώ έχουμε το «περίεργο φαινόμενο»: το άνω όριο της περιοχής αναζήτησης (*upto* = 4) να είναι μικρότερη από το κάτω όριο (*from* = 5). Δεν υπάρχει πια περιοχή αναζήτησης. Καιρός να σταματησουμε διότι το 100 δεν υπάρχει στον πίνακά μας.

Τελειώνουμε λοιπόν το ψάξιμο:

- όταν βρούμε το *x* (v[*middle*] == *x*) ή
- όταν η περιοχή που ψάχνουμε δεν υπάρχει πια (*from* > *upto*).

```
middle = (from + upto) / 2;
while ( from <= upto && v[middle] != x )
{
   if (v[middle] < x)  from = middle + 1;
   else if (v[middle] > x)  upto = middle - 1;
   middle = (from + upto) / 2;
}  // while
```

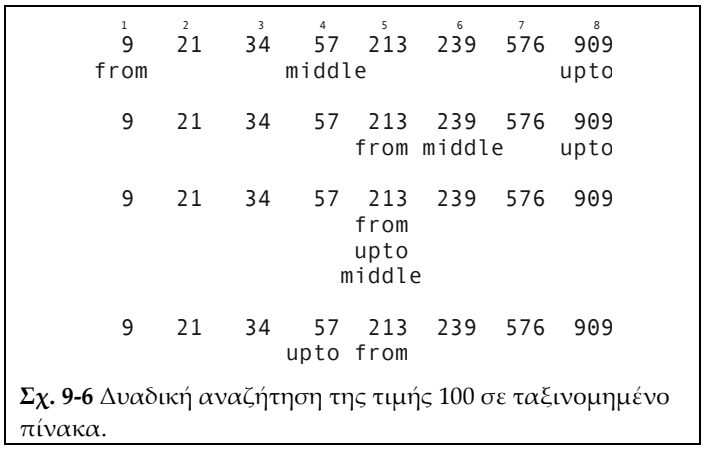
Μετά το τέλος της επαναληπτικής διαδικασίας, θα πρέπει να ελέγξουμε για ποιο λόγο σταμάτησε:

```
if (v[middle] == x)  η x υπάρχει στο Meso
```

Αυτός είναι ο αλγόριθμος **δυαδικής αναζήτησης** (binary search).

Όπως είδες και από το παράδειγμα, οι περισσότερες επαναλήψεις θα γίνουν στην περίπτωση που η τιμή *x* δεν υπάρχει στον πίνακα. Στην περίπτωση αυτήν το ψάξιμο πρέπει να συνεχιστεί ωσότου συναντηθούν οι δείκτες *from* και *upto* (που τους λέμε αντιστοίχως *κάτω δείκτη* και *άνω δείκτη* της περιοχής που ψάχνουμε). Ας πούμε ότι, αρχικά, η περιοχή που ψάχνουμε έχει *N* στοιχεία (*from* = 1 και *upto* = *N*). Όταν οι δείκτες συναντηθούν η περιοχή θα έχει λιγότερα από 1 στοιχεία (*upto* < *from*). Το μήκος της περιοχής μικραίνει με υποδιπλασιασμό (περίπου, γιατί έχουμε ακέραιη διαίρεση). Μετά από *l* επαναλήψεις, η περιοχή που ψάχνουμε θα έχει περίπου:

$$N / 2^l \text{ στοιχεία.}$$



Σχ. 9-6 Δυαδική αναζήτηση της τιμής 100 σε ταξινομημένο πίνακα.

Αυτός ο αριθμός γίνεται μικρότερος από 1 όταν το 2^l γίνει μεγαλύτερο από N, ή:

$$l > log_2 N$$

Έχουμε λοιπόν έναν αλγόριθμο με *λογαριθμική* περιπλοκότητα χρόνου, αντί για την γραμμική (ανάλογη του N) που έχουν οι προηγούμενοι. Τι σημαίνει αυτό; Αν π.χ. πάρουμε $N = 1024$, τότε στη χειρότερη περίπτωση θα έχουμε 10 επαναλήψεις ($log_2 1024 = 10$) της **while**, ενώ με το γραμμική αναζήτηση θα έχουμε 1024!

Η συνάρτηση *binSearch()*, που υλοποιεί τη μέθοδο δυαδικής αναζήτησης, θα είναι διαφορετική από τη *linSearch()* ως προς το εξής: Θα επιστρέφει πάντοτε μια τιμή δείκτη!

Έστω ότι έχουμε:

- πίνακα που δεν είναι «γεμάτος» αλλά έχει τιμές, ταξινομημένες κατ' αύξουσα τάξη, στις θέσεις από `v[1]` μέχρι `v[`*last*`]`, όπου *last* < *N* και
- μια τιμή *x* που πρέπει να εισαχθεί στον πίνακα ώστε να παραμείνει ταξινομημένος.

Η *binSearch()* θα μας δίνει τη θέση *k* στην οποία θα πρέπει να εισαχθεί η *x* διότι όταν σταματησει η **while** θα έχουμε $v[k-1] < x \le v[k]$.

Δες ένα χαρακτηριστικό παράδειγμα χρήσης της *binSearch*:

```
   ndx = binSearch( v, last, x );
   if ( v[ndx] == x )
      cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
   else
   {
      for ( k = last+1; k >= ndx; --k )  v[k+1] = v[k];
      v[ndx] = x;
      last = last + 1;
      cout << " ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
   }
```

Από την τιμή που μας επιστρέφει η συνάρτηση –και φυλάγουμε στην *ndx*– δεν ξέρουμε αν βρέθηκε η *x* στιον πίνακα· χρειάζεται και ο έλεγχος

```
   if ( v[ndx] == x )
```

Πρόσεξε πώς γίνεται η εισαγωγή της νέας τιμής στον πίνακα: Με τη **for** «ανοίγουμε χώρο» για να τη εισαγάγουμε.

Να πώς θα είναι η *binSearch*:

```
// binSearch -- Δυαδική αναζήτηση στα στοιχεία v[1],...v[last]
//              για να βρεθεί θέση fv τέτοια ώστε:
//                       v[fv-1] < x <= v[fv]
//              Επιστρέφει την fv.
//              Ο v πρέπει να είναι ταξινομημένος κατ' αύξουσα
//              τάξη με φρουρούς στα άκρα
//              v[0] == -inf,  v[last+1] == +inf
unsigned int binSearch( const int v[], int last, int x )
{
   unsigned int l( 1 ), r( last+1 );
   unsigned int middle;
```

```
   while ( l < r ) // I: (∀k:[0..l) • v[k] < x) &&
   {               //    (∀k:[r..last+1) • v[k] >= x)
      middle = (l + r) / 2;
      if ( v[middle] < x )  l = middle + 1;
                    else  r = middle;
   }  // while
// v[l-1] < x <= v[l]
   return l;
}  // binSearch
```

Στη συνέχεια δίνουμε μια (όχι και πολύ αυστηρή) απόδειξη ορθότητας της *binSearch*.

Γιατί δεν βάλαμε και εδώ παραμέτρους (*from*, *upto*) που θα μας δίνουν δυνατότητα αναζήτησης σε τμήμα του πίνακα; Αυτή η δυνατότητα έρχεται σε σύγκρουση με τη δυνατότητα να μας δίνεται η θέση εισαγωγής. Δες ένα παράδειγμα. Ας πούμε ότι έχουμε:

0	1	2	3	4	6	7	8	9	10	11	12	13	14	15	16	17
-∞	2	7	11	15	17	19	23	29	31	37	41	44	53	+∞		

και αναζητούμε το 12 με *from* = 7 και *upto* = 12. Αφού δεν υπάρχει θα μας υποδειχθεί η εισαγωγή στη θέση *from* = 7 αλλά αυτό είναι λάθος. Παρομοίως, θα μας υποδειχθεί η εισαγωγή του 61 στη θέση *upto*+1 = 13. Οι υποδείξεις για τη θέση εισαγωγής είναι σωστές μόνον αν v[*from*-1] < *x* < v[*upto*+1].

Δοκιμάζουμε τη *binSearch* με ένα πρόγραμμα που έχει τις εντολές αναζήτησης – εισαγωγής (που είδαμε παραπάνω) σε έναν ταξινομημένο πίνακα που διαβάζουμε από το pinsrt.txt:

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;

unsigned int binSearch( const int v[], int n, int x );

int main()
{
   const unsigned int N( 60 );

   int  v[N];               // ο πίνακας που ψάχνουμε
   unsigned int last(N-10); // με τιμές στις θέσεις v[1]...v[last]
   int  x;                  // για την τιμή της ψάχνουμε
   unsigned int ndx;        // εδώ θα είναι αν τη βρούμε
   int  k;
   ifstream a( "pinsrt.txt" );

   for ( k = 1; k <= last; k = k+1 )
      a >> v[k];                              // Διάβασε τον πίνακα
   a.close();
   v[0] = INT_MIN;  v[last+1] = INT_MAX;      // βάλε τους φρουρούς

   cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   while ( x != 9999 )
   {
      ndx = binSearch( v, last, x );
      if ( v[ndx] == x )
         cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      else
      {
         for ( k = last+1; k >= ndx; --k )  v[k+1] = v[k];
         v[ndx] = x;
         last = last + 1;
         cout << " ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      }
      cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   } // while
} // main
```

Τώρα, κρατούμε για τον πίνακα 60 θέσεις αλλά τον φορτώνουμε στις θέσεις 1..50. Στις θέσεις 0 και 51 μπαίνουν οι φρουροί.

```
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 40
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 21
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 43
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 41
ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 40
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 21
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 43
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 23
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 41
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 9999
```

Παρατήρηση: ▶

Εδώ θα πρέπει να σταματήσουμε για λίγο και να σκεφτούμε αυτά που είδαμε. Η δυαδική αναζήτηση είναι πολύ εντυπωσιακή στην ταχύτητά της αλλά δεν είναι θαύμα. Είναι γρήγορη, διότι ο πίνακας μας έχει υποστεί μια προεργασία, *έχει ταξινομηθεί*. Ας πούμε λοιπόν ότι θέλεις να κάνεις ένα σύστημα, όπου θα πρέπει θα ψάχνεις έναν πίνακα και να δίνεις την απάντησή σου *γρήγορα*. Θα πρέπει να χρησιμοποιήσεις μια γρήγορη μέθοδο, όπως είναι η δυαδική αναζήτηση. Ενώ όμως απαιτείται ταχύτητα στην αναζήτηση, πολύ συχνά, έχεις αρκετή άνεση χρόνου για να προετοιμάσεις κατάλληλα το σύστημά σου: να ταξινομήσεις τον πίνακά σου ή να ενημερώνεις τον πίνακά σου ώστε να είναι ταξινομημένος – διαδικασίες σχετικά χρονοβόρες. Με αυτήν τη φιλοσοφία σχεδιάζονται πολλά πληροφοριακά συστήματα σήμερα και ένα πολύ κοινό παράδειγμα είναι ο τηλεφωνικός κατάλογος. Όταν τον ψάχνεις, η αλφαβητική ταξινόμηση σου είναι απαραίτητη και την έχεις. Έτσι, έχεις τη δυνατότητα να ψάχνεις γρήγορα. Η ταξινόμηση γίνεται πριν τυπωθεί, όταν η υπηρεσία που τον ετοιμάζει έχει όλη τη χρονική άνεση να κάνει κάτι τέτοιο. ◀

Και τώρα, ας ξανάρθουμε στην ταξινόμηση. Ο αλγόριθμος που είδαμε απαιτεί, για μεγάλα N, χρόνο λN^2. Πρόσεξε τώρα το εξής:

- Κόβουμε τον πίνακά μας στη μέση. Έτσι έχουμε δυο πίνακες με μήκος $\frac{1}{2}N$ ο καθένας.
- Ταξινομούμε τον κάθε έναν από αυτούς, σε χρόνο $\lambda(\frac{1}{2}N)^2 = \frac{1}{4}\lambda N^2$. Συνολικώς: $\frac{1}{2}\lambda N^2$.
- Συγχωνεύουμε τους δυο πίνακες σε έναν, σε χρόνο περίπου κN, που για μεγάλα N, είναι αμελητέος μπροστά στο $\frac{1}{2}\lambda N^2$.

Με τις διαδικασίες που γράψαμε αυτό θα μπορούσε να γίνει ως εξής:

```
middle = (from + upto) / 2;
ταξινόμησε το τμήμα v[from]… v[middle]
ταξινόμησε το τμήμα v[middle+1]… v[upto]
συγχώνευσε στον πίνακα z τα δύο τμήματα
```

Βρήκαμε δηλαδή έναν τρόπο να υποδιπλασιάσουμε το χρόνο ταξινόμησης. Τι πληρώσαμε; Διπλασιάσαμε τις απαιτήσεις σε μνήμη (πίνακας z). Ύστερα από αυτό, δεν μπορούμε να μη σκεφτούμε: «γιατί να μην ταξινομήσουμε τα δυο μισά με τον ίδιο τρόπο;» Γιατί όχι; Αυτή ακριβώς είναι η ιδέα για την *ταξινόμηση με συγχώνευση* (merge sort) που ταξινομεί έναν πίνακα με N στοιχεία σε χρόνο ανάλογο του $N\log N$, που φυσικά είναι καλύτερος από N^2. Αυτός ο αλγόριθμος χρειάζεται διπλή μνήμη από αυτήν που απαιτείται για την αποθήκευση του πίνακα. Αργότερα θα μάθεις ότι υπάρχουν αλγόριθμοι ταξινόμησης με χρόνο $N\log N$, χωρίς υπερβολικές απαιτήσεις μνήμης.

Αυτή είναι μια περίπτωση εφαρμογής της αρχής «διαίρει και βασίλευε» (divide and conquer) στον αλγόριθμό μας: Κόβουμε τον πίνακα στα δυο και λύνουμε το αρχικό πρόβλημα (ταξινόμηση) στα δυο κομμάτια.

9.6.1 Απόδειξη Ορθότητας της *binSearch*

Η απόδειξη ορθότητας θα γίνει με τη μέθοδο της επαγωγής.

Αρχικώς, στην [0..*l*) υπάρχει μόνον ο φρουρός "`-∞`" και ισχύει η "`-∞ <` x". Η δεξιά περιοχή είναι κενή: [*last*+1..*last*+1) και η ∀*k*: [*r*..*last*+1) • v`[`*k*`]` ≥ x είναι (τετριμμένως) αληθής.

Αν v`[`*middle*`] <` x βάζουμε `l = middle + 1`. Λόγω της ταξινόμησης του πίνακα η v`[`*k*`] <` x θα ισχύει για κάθε *k* στο [0..*middle*] ή στο [0..*middle*+1) που είναι το [0..*l*). Για την ∀*k*: [*r*..*last*+1) • v`[`*k*`]` ≥ x δεν έχουμε αλλαγή.

Αν v`[`*middle*`]` ≥ x βάζουμε `r = middle`. Λόγω της ταξινόμησης του πίνακα η v`[`*k*`]` ≥ x θα ισχύει για κάθε *k* στο [*middle*..*last*] ή στο [*middle*..*last*+1) που είναι το [*r*..*last*+1). Για την ∀*k*: [0..*l*) • v`[`*k*`] <` x δεν έχουμε αλλαγή.

Μετά τη **while** θα έχουμε:

$$(l >= r)\ \&\&\ (\forall k:\ [0..l) \bullet v[k] < x)\ \&\&\ (\forall k:\ [r..last+1) \bullet v[k] \geq x)$$

Αφού $l \in [r..last+1)$ θα έχουμε: v`[`l`]` ≥ x. Παρομοίως, αφού l-1 ∈ [0..l) θα έχουμε: v`[`l-1`] <` x. Δηλαδή θα ισχύει η v`[`l-1`] <` $x \leq$ v`[`l`]`.

Για την απόδειξη του τερματισμού εξετάζουμε τη διαφορά $d = r - l$ που έχει θετική τιμή (από τη συνθήκη της **while**).

Από την $l <$ `r` παίρνουμε $l \leq$ *middle* `<` r (το "`=`" αριστερά χρειάζεται διότι η (*l*+*r*)`/`2 είναι ακέραιη διαίρεση.)

- Αν εκτελεσθεί η `l = middle + 1` η νέα τιμή διαφοράς $d' = r - (l+r)/2 - 1 < d$.
- Αν εκτελεσθεί η `r = middle` η νέα τιμή διαφοράς $d' = (l+r)/2 - l < d$.

Επομένως η $r - l$ παράγει –όσο εκτελείται η **while**– μια γνησίως φθίνουσα ακολουθία θετικών ακεραίων που δεν είναι δυνατόν να έχει άπειρο πλήθος όρων.

9.7 Ανακεφαλαίωση

Ένας πίνακας είναι μια ακολουθία τιμών ίδιου τύπου, όπως και ένα σειριακό αρχείο, αλλά:

- Ο πίνακας υλοποιείται στην κύρια μνήμη ενώ το αρχείο στη βοηθητική.
- Μπορούμε να έχουμε πρόσβαση σε οποιοδήποτε στοιχείο του πίνακα με την ίδια ευκολία, ενώ στο σειριακό αρχείο για να πάμε στο ν-οστό στοιχείο πρέπει να περάσουμε από τα προηγούμενα $\nu - 1$.[7]
- *Μπορούμε να χειριστούμε οποιοδήποτε στοιχείο ενός πίνακα όπως μια μεταβλητή* ενώ για να χειριστούμε μια τιμή από ένα αρχείο θα πρέπει να τη φέρουμε στην κύρια μνήμη.

Τα ν στοιχεία ενός πίνακα αριθμούνται από 0..ν–1. Όλα τα στοιχεία του πίνακα έχουν το ίδιο όνομα και διακρίνονται μεταξύ τους από τον *δείκτη* που είναι ο αριθμός του στοιχείου μέσα σε αγκύλες.

Οι πίνακες χρησιμοποιούνται φυσικά για την επίλυση με ΗΥ μαθηματικών προβλημάτων όπου χρησιμοποιούνται πίνακες. Πέρα από αυτό όμως οι πίνακες διευκολύνουν τον προγραμματισμό σε περιπτώσεις που έχουμε πολλά ομοειδή (ίδιου τύπου) αντικείμενα που υφίστανται την ίδια επεξεργασία.

Η αναζήτηση τιμών σε έναν πίνακα είναι ένα πολύ συνηθισμένο πρόβλημα. Η ταχύτητα της αναζήτησης αυξάνεται σημαντικά αν τα δεδομένα μας είναι ταξινομημένα. Έτσι, έχουν επινοηθεί πολλοί αλγόριθμοι για ταξινόμηση.

---

[7] Αργότερα θα δούμε ότι μπορούμε να έχουμε και αρχεία *τυχαίας πρόσβασης*.

Ασκήσεις

Α Ομάδα

9-1 Έστω ένας πίνακας

```
double a[10];
```

Γράψε εντολές που θα τον «αναποδογυρίσουν». Δηλαδή να αντιμεταθέσουν τις τιμές των στοιχείων του:

(a[0] ↔ a[9]) (a[1] ↔ a[8]) (a[2] ↔ a[7]) (a[3] ↔ a[6]) (a[4] ↔ a[5])

9-2 Ένας τρόπος να δούμε «πόσο μεγάλος» είναι ένας μονοδιάστατος πίνακας A, με στοιχεία από 0 μέχρι $N - 1$, είναι να υπολογίσουμε το:

$$\|A\| = |A_0| + |A_1| + \ldots + |A_{N-1}|$$

Γράψε πρόγραμμα που θα διαβάζει τις τιμές των στοιχείων του A ($N = 5$) και θα υπολογίζει και θα τυπώνει το $\|A\|$.

Στη συνέχεια, αν $\|A\| \neq 0$, θα κάνει τον A *μοναδιαίο*, διαιρώντας όλα τα στοιχεία του δια $\|A\|$. Τέλος, θα τυπώνει τα στοιχεία του μοναδιαίου πίνακα.

Επανάλαβε τα παραπάνω αν: $\|A\| = \sqrt{A_0^2 + A_1^2 + \ldots + A_{N-1}^2}$.

Το ίδιο αν: $\|A\| = \max_{k=0..N-1} (|A_k|)$.

Τα μαθηματικά ονομάζουν τα τρία αποτελέσματα νόρμα-1, νόρμα-2 και νόρμα-∞ αντίστοιχα.

9-3 Λύσε ξανά την προηγούμενη άσκηση αφού γράψεις τρεις συναρτήσεις *norm*1, *norm*2 και *normInf* που υπολογίζουν τις τρεις νόρμες.

9-4 Τροποποίησε το πρόγραμμα συγχώνευσης για την περίπτωση που ο πίνακας w δίνεται ταξινομημένος σε *φθίνουσα* τάξη. **Προσοχή!** *Μόνον* ο w.

9-5 Γράψε μια

```
double vectorSumIf( const double x[], int n,
                    double lb, double ub )
```

που θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων x[*k*] του x για τα οποία: $lb \le x[k] \le ub$.

Β Ομάδα

9-6 Έστω ότι έχουμε δύο πίνακες x, y με n στοιχεία. Λέμε ότι ο x *προηγείται λεξικογραφικά* του y αν υπάρχει κάποιο k, από 0 μέχρι $n-1$, τέτοιο ώστε να έχουμε: $x_k < y_k$ και $x_j == y_j$ για κάθε j από 0 μέχρι $k-1$.

Γράψε μια:

```
bool lt( int x[], int y[], int n)
```

που θα επιστρέφει τιμή:

- `true` αν ο x προηγείται λεξικογραφικά του y και
- `false` αν όχι.

9-7 *Μέθοδος των ελαχίστων τετραγώνων*. Παραλλαγή της Άσκ. 6-11. Γράψε πρόγραμμα που θα διαβάζει από ένα αρχείο text, με όνομα στο δίσκο lsq.txt, τα (x_k, y_k), $k = 0 .. N-1$ και θα υπολογίζει τα α και β. Σε κάθε γραμμή του αρχείου έχουμε ένα ζεύγος x_k, y_k. Υποθέτουμε το N δεν υπερβαίνει το 100. Οι τιμές των x_k, y_k που διαβάζονται θα αποθηκεύονται σε δύο πίνακες x, y με χώρο για 100 στοιχεία το πολύ.

Ένα μέτρο του πόσο καλή είναι η προσαρμογή ευθείας στα δεδομένα δίνεται από τον **συντελεστή πολλαπλής συσχέτισης** (multiple correlation coefficient) R^2 που ορίζεται ως εξής:

$$R^2 = \frac{\sum_{k=0}^{N-1}(\alpha x_k + \beta - < y >)^2}{\sum_{k=0}^{N-1}(y_k - < y >)^2} \quad \text{όπου: } <y> = \frac{1}{N}\sum_{k=0}^{N-1} Y_k$$

Η τιμή του R^2 είναι από 0 (τέλεια προσαρμογή) μέχρι 1. Συμπλήρωσε λοιπόν το πρόγραμμά σου ώστε μετά τον υπολογισμό των α και β να υπολογίζει και να γράφει το R^2.

***9-8** Απόδειξε ότι η *maxNdx* είναι σωστή, δηλαδή

```
// 0 <= from <= upto <= n-1
   mxp = from;
   m = from+1;
   while ( m <= upto ) )
   {
      if ( x[m] > x[mxp] )  mxp = m;
      m = m + 1;
   }
// from ≤ mxp ≤ upto && ∀j: from..upto • x[j] ≤ x[mxp]
```

Υπόδ.: Απόδειξε ότι η

$$from \le mxp \le m - 1 \;\&\&\; \forall j: from\,..\,m-1 \bullet \mathtt{x[}j\mathtt{]} \le \mathtt{x[}mxp\mathtt{]}$$

είναι αναλλοίωτη της **while**.

9-9 Γράψε μια:

```
int linSearchSrt( const int v, int n, int from, int upto, int x )
```

που θα ψάχνει *γραμμικώς* στον *ταξινομημένο*, σε αύξουσα τάξη, πίνακα v για την τιμή *x*. Απόδειξε ότι είναι σωστή.

9-10 Γράψε μια:

```
unsigned int linSearchMult( const int v, int n,
                            int from, int upto, int x )
```

που θα ψάχνει *γραμμικώς* στον πίνακα v –που υλοποιεί ένα *πολυσύνολο*– για τη *x* και θα επιστρέφει το *πλήθος* των στοιχείων του v που είναι ίσα με τη *x*.

9-11 Απόδειξε ότι το πρόγραμμα ταξινόμησης είναι σωστό, δηλαδή (στα `v[0]` και `v[`N`+1]` έχουμε τα $-\infty$ και $+\infty$)[8]:

```
// N > 0
   k = N;
   while ( k >= 2 ) // ∀j: k..N • v[j] ≤ v[j+1]
   {
      mxp = maxNdx( v, N+2, 1, k );
      sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
      k = k - 1;
   } // for
// ∀j: 1..N • v[j] ≤ v[j+1]
```

Γ Ομάδα

9-12 Έλυσες την Άσκ. 7-15; Λύσε τώρα και το πραγματικό πρόβλημα: Γράψε συνάρτηση

```
double mp( double a[], int N, double x )
```

που να υλοποιεί την παρακάτω συνάρτηση:

[8] Για να είναι σωστός ο αλγόριθμος θα πρέπει να αποδείξουμε ότι δεν κάνουμε εισαγωγή ή διαγραφή τιμών. Αφού όμως ο αλγόριθμός μας κάνει αντιμεταθέσεις τιμών στοιχείων μπορούμε να θεωρήσουμε ότι αυτό είναι εξασφαλισμένο.

$$m_p(x) = \prod_{k=0}^{N-1} \frac{1}{x-a_k} = \frac{1}{x-a_0} \times \frac{1}{x-a_1} \times \ldots \times \frac{1}{x-a_{N-1}}, \text{ όπου } N \geq 1$$

9-13 Έστω ότι έχουμε έναν πίνακα:

```
const int N = 50;

int v[N+2];
```

Στα v[0] και v[N+2] έχουμε τα $-\infty$ και $+\infty$ αντιστοίχως.

Στον πίνακα έχουμε ήδη τιμές στις θέσεις από v[1] μέχρι v[l] ($l < N$), ταξινομημένες κατ' αύξουσα τάξη. Θέλουμε να εισαγάγουμε μία ακόμη τιμή –της μεταβλητής x– στη θέση v[m] όπου $m \leq l + 1$, έτσι ώστε:

- να μη χάσουμε κάποια από αυτές που ήδη υπάρχουν,
- ο πίνακας να συνεχίσει να είναι ταξινομημένος κατ' αύξουσα τάξη.

9-14 (Συνέχεια της προηγούμενης) Με βάση τα παραπάνω μπορούμε να δούμε μιαν άλλη μέθοδο ταξινόμησης ενός πίνακα: την μέθοδο **κατ' ευθείαν εισαγωγής** (straight insertion sort):

```
for ( i = 2; i <= N; i = i+1 ) //I: το v[1]…v[i-1] ταξινομημένο
{
   Βάλε το v[i] στη σωστή θέση στο κομμάτι v[1]…v[i-1]
} // for
```

Άλλαξε το πρόγραμμα της ταξινόμησης ώστε να υλοποιεί αυτήν τη μέθοδο.

9-15 Μας δίνονται δύο αρχεία text, int1.txt και int2.txt, που περιέχουν ακέραιους αριθμούς ταξινομημένους κατ' αύξουσα τάξη. Γράψε πρόγραμμα που θα συγχωνεύσει τα περιεχόμενα των δύο αρχείων σε ένα (int3.txt) ταξινομημένο κατ' αύξουσα τάξη.