ΤΕΧΝΟΛΟΓΙΚΟ ΙΔΡΥΜΑ ΛΑΡΙΣΑΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ &
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΣΗΜΕΙΩΣΕΙΣ ΓΙΑ ΤΟ ΜΑΘΗΜΑ

ΑΛΓΟΡΙΘΜΟΙ & ΠΟΛΥΠΛΟΚΟΤΗΤΑ

Δρ. Ηλίας Κ. Σάββας
ΛΑΡΙΣΑ, Ιανουάριος 2005

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΑΛΓΟΡΙΘΜΟΙ

ΠΡΟΓΡΑΜΜΑΤΑ

ΣΧΗΜΑΤΑ

1. ΟΡΙΣΜΟΙ & ΟΡΟΛΟΓΙΑ

1.1 Εισαγωγή

Αλγόριθμος είναι ένα σύνολο οδηγιών οι οποίες επιλύουν ένα συγκεκριμένο πρόβλημα ή μία κλάση προβλημάτων. Επιπλέον ένας αλγόριθμος πρέπει να πληροί και τα ακόλουθα κριτήρια:

1) ***Είσοδος***: Για να λειτουργήσει πρέπει να εισαχθούν Ν δεδομένα από κάποια εξωτερική πηγή (το Ν μπορεί να είναι και μηδέν).
2) ***Έξοδος***: Με το τέλος του αλγόριθμου πρέπει να παράγεται τουλάχιστο ένα αντικείμενο σαν αποτέλεσμα.
3) ***Καλά ορισμένος***: Η κάθε οδηγία πρέπει να είναι απόλυτα καθορισμένη, κατανοητή και να μην αφήνει κανένα περιθώριο αμφισβήτησης (πχ σε κάποια απόφαση).
4) ***Πεπερασμένος***: Να τελειώνει σε πεπερασμένο αριθμό βημάτων.
5) ***Αποτελεσματικός***: Κάθε οδηγία του αλγόριθμου πρέπει να είναι απόλυτα βασική και να μπορεί να εκτελεσθεί από μία υπολογιστική μηχανή. Δηλαδή το τρίτο κριτήριο δεν είναι αρκετό: πρέπει η οδηγία επιπλέον να είναι έτσι ώστε να φέρνει κάποιο αποτέλεσμα.
6) ***Γενικός***: Εάν είναι δυνατόν, ο αλγόριθμος πρέπει να λύνει μία κλάση προβλημάτων και όχι ένα και μοναδικό πρόβλημα. Για παράδειγμα, καλός είναι ο αλγόριθμος που επιλύει την εξίσωση πρώτου βαθμού $ax+b=0$ και όχι μία συγκεκριμένη εξίσωση σαν την $5x+6=0$.

Τέλος, ένας αλγόριθμος πρέπει να περιγράφεται αναλυτικά και με τέτοιο τρόπο ώστε να είναι απόλυτα κατανοητός ακόμη και σε κάποιον που δεν ξέρει το πρόβλημα που επιλύει (βλέπε *Αλγόριθμος 1*). Για την ιστορία, η λέξη αλγόριθμος προήλθε από το όνομα ενός Πέρση που είχε γράψει ένα βιβλίο Μαθηματικών (825 μ.Χ.) τον Abu Ja'far Mohammed ibn Musa al Khowarizmi.

Ανάλογα με την τεχνική επίλυσης ενός προβλήματος οι αλγόριθμοι διακρίνονται σε:

- ✓ Αναδρομικοί (recursive): αλγόριθμοι που χρησιμοποιούν αναδρομικές λύσεις προβλημάτων, πχ πολυώνυμα Hermite, υπολογισμός παραγοντικού, κ.α.
- ✓ Διαίρει και Βασίλευε (divide and conquer): επιλύουν το πρόβλημα αναγάγοντάς το σε μικρότερα ανάλογα προβλήματα, πχ quick sort, merge sort, κα
- ✓ Άπληστοι (greedy): επιλύουν προβλήματα επιλέγοντας κάθε φορά την τοπικά βέλτιστη λύση προσδοκώντας την συνολικά βέλτιστη, πχ πρόβλημα επιστροφής ρέστων, χρονικός προγραμματισμός, κ.α.
- ✓ Δυναμικού Προγραμματισμού (dynamic programming): συνήθως αναδρομικοί αλγόριθμοι οι οποίοι χρησιμοποιούν τις ενδιάμεσα παραγόμενες λύσεις, πχ πολλαπλασιασμός αλυσίδας πινάκων, κα.
- ✓ Παράλληλοι (parallel): εύρεση λύσης όπου το πρόβλημα δεν λύνεται σειριακά αλλά πολλές σχέσεις του εκτελούνται παράλληλα, πχ πολλά προβλήματα πινάκων, τεχνικές τύπου «διαίρει και βασίλευε», κ.α.

Επίσης, ανάλογα με την λύση που επιτυγχάνουν μπορούν να διακριθούν σε:

- ✓ Βέλτιστοι ή Άριστοι (optimal): εύρεση της βέλτιστης λύσης του προβλήματος, πχ επίλυσης μίας εξίσωσης δευτέρου βαθμού.
- ✓ Προσεγγιστικοί ή ευρεστικοί (approximation – heuristics): εύρεση «καλών» λύσεων σε άλυτα ή πολύ δύσκολα προβλήματα, πχ χρονοπρογραμματισμός εργασιών, χρωματισμός χάρτη κ.α.

Αλγόριθμος 1: Εξίσωση2 - Επίλυση Εξίσωσης 2ου Βαθμού ($ax^2+bx+c=0$)

1: Εισαγωγή των a, b, και c.
2: Εάν (a=0 ΚΑΙ b=0 ΚΑΙ c=0) Τότε
3: Αόριστη εξίσωση,
4: Τέλος Αλγόριθμου «Εξίσωση2»
5: Τέλος Εάν
6: Εάν (a=0 ΚΑΙ b=0 ΚΑΙ c≠0) Τότε
7: Αδύνατη εξίσωση,
8: Τέλος Αλγόριθμου «Εξίσωση2»
9: Τέλος Εάν
10: Υπολόγισε $d \leftarrow b^2 - 4ac$
11: Υπολόγισε $x_1 \leftarrow \frac{-b+\sqrt{d}}{2a}$
12: Υπολόγισε $x_2 \leftarrow \frac{-b-\sqrt{d}}{2a}$
13: Εκτύπωσε τα x_1 και x_2.
14: Τέλος Αλγόριθμου «Εξίσωση2»

1.2 Έννοιες Κλειδιά

Για να λειτουργήσει ένας αλγόριθμος χρειάζεται πληροφορίες ή αλλιώς δεδομένα τις οποίες και επεξεργάζεται και δίνει τις πιθανές λύσεις. Τα δεδομένα σε έναν αλγόριθμο κατηγοριοποιούνται σε τρεις γενικές κατηγορίες:

⇒ ***Δεδομένα*** (Πληροφορίες)
- Αρχικά: Είναι τα δεδομένα εισόδου που πρέπει να εφοδιαστεί ο αλγόριθμος ώστε να μπορέσει να επεξεργαστεί το πρόβλημα,
- Ενδιάμεσα: Αυτά είναι τα δεδομένα που παράγει ο αλγόριθμος για να μπορέσει να φτάσει στην λύση, και τέλος
- Αποτελέσματα: Είναι οι πληροφορίες που αφορούν την λύση του προβλήματος.

Στον *Αλγόριθμο 1*, αρχικά δεδομένα είναι τα a, b, και c, ενδιάμεσα το d, και αποτελέσματα τα x_1 και x_2.

Τα δεδομένα σε έναν αλγόριθμο περιγράφονται σαν ***μεταβλητές***, δηλαδή υπολογιστικές οντότητες για την αναπαράσταση των δεδομένων σε ένα πρόγραμμα ή αλγόριθμο. Επίσης, τα δεδομένα μπορεί να είναι διαφόρων τύπων. Οι ***τύποι***

δεδομένων καθορίζουν τις τιμές που επιτρέπεται να πάρει μία μεταβλητή πχ ακέραιος αριθμός, πραγματικός κοκ, αλλά και τους τρόπους διαχείρισης ή επεξεργασίας των τιμών αυτών, δηλαδή τις επιτρεπόμενες πράξεις. Οι τύποι των δεδομένων μπορούν να διαχωριστούν στους απλούς και σύνθετους τύπους. Οι κατηγορίες των απλών τύπων είναι οι:

1) Ακέραιος αριθμός (integer),
2) Πραγματικός αριθμός (real – float),
3) Πραγματικός αριθμός μεγάλης ακρίβειας (double),
4) Χαρακτήρας (character), και
5) Boolean (με μοναδικές τιμές: Αληθής / Ψευδής ή 0 / 1).

Οι σύνθετοι τύποι δεδομένων ονομάζονται δομές δεδομένων. ***Δομή Δεδομένων*** είναι ένα τύπος δεδομένων που αποτελείται από σύνθετες τιμές, δηλαδή τιμές που συντίθενται από άλλες απλούστερες επιμέρους τιμές (κόμβοι) και μεταξύ των οποίων υπάρχει ένα οργανωτικό σχήμα. Μερικές από τις πλέον συνηθισμένες δομές δεδομένων είναι οι:

1) Πίνακας (array),
2) Εγγραφή (record – structure),
3) Λίστα (list),
4) Ουρά (queue),
5) Στοίβα ή σωρός (stack),
6) Γράφος (graph),
7) Δένδρα (tree), κλπ

και με επιτρεπτές πράξεις:

1) Διαπέραση,
2) Αναζήτηση,
3) Εισαγωγή,
4) Διαγραφή,
5) Διάταξη, κλπ.

1.3 Πολυπλοκότητα & Απόδοση Αλγορίθμων

Σαν πολυπλοκότητα ή απόδοση αλγορίθμου ορίζεται το κόστος χρήσης του αλγόριθμου για την επίλυση ενός προβλήματος. Μονάδες μέτρησης κόστους μπορεί να θεωρηθούν:

- ✓ Ο υπολογιστικός χρόνος εκτέλεσης του αλγόριθμου, δηλαδή χρήση της CPU μίας υπολογιστικής μηχανής,
- ✓ Η χρήση αποθηκευτικών χώρων,
- ✓ Η χρήση πόρων δικτύου, ή και
- ✓ σε οποιαδήποτε άλλη μονάδα που μπορεί να εκφράζει χρήση οποιονδήποτε πόρων υπολογιστικών και δικτύων.

Βέβαια, ο χρόνος εκτέλεσης ενός αλγόριθμου θεωρείται σαν το πιο σημαντικό μέτρο χαρακτηρισμού του αλγόριθμου. Μάλιστα, επειδή σε πάρα πολλές περιπτώσεις είναι

εντελώς αδύνατο να υπολογισθεί αυτός ο χρόνος με ακρίβεια, υπολογίζονται συναρτήσεις που αποδίδουν τον καλύτερο, μέσο και χειρότερο χρόνο του.

Για παράδειγμα στον *Αλγόριθμο 1*, ο καλύτερος χρόνος είναι όταν η εξίσωση είναι αόριστη και ο χειρότερος όταν θα υπάρχουν οι δύο λύσεις.

1.4 Αλγοριθμική Γλώσσα

Για την περιγραφή των αλγορίθμων έχει προταθεί και χρησιμοποιείται μία «γλώσσα», η λεγόμενη αλγοριθμική γλώσσα η οποία είναι αποδεσμευμένη από τις λεπτομέρειες μιας κανονικής γλώσσας προγραμματισμού σαν την C ή την Pascal, Java και άλλες. Ο στόχος είναι να περιγράφεται η λύση ενός προβλήματος με τέτοιο τρόπο ώστε να μπορεί να μεταφερθεί αργότερα σε οποιαδήποτε γλώσσα προγραμματισμού. Επίσης, στην αλγοριθμική γλώσσα πρέπει να αποφεύγονται εκφράσεις που υπάρχουν σε κάποια γλώσσα προγραμματισμού ενώ δεν υπάρχουν σε άλλες (για παράδειγμα οι τελεστές ++ , -- κ.α. της C).

Σε γενικές γραμμές, τα δομικά στοιχεία της αλγοριθμικής γλώσσας είναι τα ακόλουθα:

1) *Δεδομένα* τα οποία μπορεί να είναι:
   a) Μεταβλητές ποσότητες (πχ x, a, mikos κλπ),
   b) Σταθερές ποσότητες (πχ ένας αριθμός, ένα όνομα κ.α.),

2) *Τελεστές*:
   a) Πρόσθεση,
   b) Αφαίρεση,
   c) Πολλαπλασιασμός,
   d) Διαίρεση,
   e) Υπόλοιπο,
   f) Ισότητα,
   g) Όλες οι πιθανές ανισότητες, κ.λ.π.

3) *Παραστάσεις*:
   a) Εκχώρηση Τιμής, πχ a ← 3x+1 κλπ,
   b) Κριτήρια ή Συνθήκες, πχ α>7, onoma=”Nikos”, κλπ,

4) *Διατάξεις Ελέγχου Ροής* του Αλγόριθμου:
   a) Η πρόταση:
      i) Εάν (συνθήκη) Τότε {
         (1) προτάσεις}
      ii) Αλλιώς
         (1) {προτάσεις}
      iii) Τέλος Εάν (ή If … Then … Else … End If)
   b) Η πρόταση πολλαπλής επιλογής:
      i) Επέλεξε με το …
         (1) Περίπτωση 1: {προτάσεις}
         (2) Περίπτωση 2: {προτάσεις}

(3) …
(4) Περίπτωση ν: {προτάσεις}
ii) Τέλος Επιλογής (Switch ή Case)

5) *Επαναληπτικές Δομές*:
   a) Για … Τέλος Για (For … End For)
      i) Για (προτάσεις ελέγχου επανάληψης)
         (1) {προτάσεις}
      ii) Τέλος Για
   b) Εφόσον (While … End While)
      i) Εφόσον (προτάσεις ελέγχου επανάληψης)
         (1) {προτάσεις}
      ii) Τέλος Εφόσον
   c) Επανέλαβε (Do … While ή Repeat … Until)
      i) Επανέλαβε
         (1) {προτάσεις}
      ii) Μέχρι (προτάσεις ελέγχου επανάληψης)

6) *Διαδικασίες* (procedures – routines):
   a) Διαδικασία «όνομα διαδικασίας με πιθανές παραμέτρους»
      i) {προτάσεις διαδικασίας}
   b) Τέλος Διαδικασίας «όνομα διαδικασίας»

7) *Συναρτήσεις* (functions):
   a) Τύπος επιστρεφόμενου δεδομένου «όνομα συνάρτησης με πιθανές παραμέτρους»
      i) {προτάσεις συνάρτησης
      ii) Επέστρεψε …}
   b) Τέλος Συνάρτησης «όνομα συνάρτησης»

1.5 Παραδείγματα & Λυμένες Ασκήσεις

1. Υπολογισμός Μέγιστου Κοινού Διαιρέτη με χρήση του αναδρομικού αλγόριθμου του Ευκλείδη.

ΛΥΣΗ

Αλγόριθμος 2: ΜΚΔ1 (Υπολογισμός Μ.Κ.Δ. των αριθμών χ και ψ με τον αλγόριθμο του Ευκλείδη)

```
1: Δεδομένα / Είσοδος: ακέραιοι χ, ψ
2: Ακέραιος ΜΚΔ1(ακέραιος χ, ακέραιος ψ)
3: Αρχή
4:      Εάν ψ>0 Τότε
5:          Επέστρεψε ΜΚΔ(ψ, χ υπόλοιπο ψ)
6:      Αλλιώς
7:          Επέστρεψε ψ
8:      Τέλος Εάν
9: Τέλος Συνάρτησης «ΜΚΔ1»
```

2. *Αραιός ονομάζεται ένας πίνακας του οποίου τα περισσότερα στοιχεία είναι μηδενικά σε ποσοστά που υπερβαίνουν το 80%. Για να μην γίνεται αυτή η σπατάλη χώρου ας υποτεθεί ότι μεταφέρονται τα μη μηδενικά στοιχεία του πίνακα σε ένα νέο πίνακα και διαγράφεται ο παλιός. Το πρόβλημα είναι ότι η θέση των στοιχείων στον αρχικό πίνακα είναι σημαντική οπότε για να διατηρηθεί αυτή η πληροφορία, ο νέος πίνακας αντί για μονοδιάστατος θα μπορούσε να είναι δισδιάστατος με την εξής οργάνωση: σε κάθε γραμμή η πρώτη στήλη να περιέχει τα μη μηδενικά στοιχεία του αραιού πίνακα ενώ στην δεύτερη και τρίτη στήλη να τοποθετούνται οι συντεταγμένες αυτών των στοιχείων που είχαν στον αρχικό πίνακα (δηλαδή τον αριθμό της γραμμής και της στήλης τους). Με βάση αυτή τη διαδικασία να δημιουργηθούν οι αλγόριθμοι μεταφοράς των στοιχείων και αναζήτησης στοιχείου με βάση την θέση του.*

Αλγόριθμος 3: Μεταφορά Αραιού Πίνακα

```
1: Δεδομένα / Είσοδος: Αραιός πίνακας Π_NxM, Πίνακας Στόχος Σ_Tx3
2: Διαδικασία «Μεταφορά»
3: Αρχή
4:      Ακέραιοι: ι, κ, λ
5:      λ ← 0
6:      Για ι = 1 Μέχρι N
7:           Για κ = 1 Μέχρι M
8:                Εάν (Π_ι,κ ≠ 0) Τότε
9:                     λ ← λ + 1
10:                    Σ_λ,1 ← Π_ι,κ
11:                    Σ_λ,2 ← ι
12:                    Σ_λ,3 ← κ
13                Τέλος Εάν
14           Τέλος Για (κ)
15      Τέλος Για (ι)
16: Τέλος Διαδικασίας «Μεταφορά»
```

Αλγόριθμος 4: Ανάκτηση Στοιχείου Αραιού Πίνακα από τους Δείκτες του

```
1: Δεδομένα / Είσοδος: Πίνακας  Σ_Tx3
2: Τύπος Στοιχείου Αραιού Πίνακα «Ανάκτηση(ακέραιος χ, ακέραιος ψ)»
3: Αρχή
4:      Ακέραιος: ι ← 0
5:      Boolean: flag ← «Ψευδής»
6:      Τύπος δεδομένων των στοιχείων του αραιού πίνακα: β
7:      Επανέλαβε
8:           ι ← ι + 1
9:           Εάν ( Σι,2 = χ ΚΑΙ Σι,3 = υ) Τότε
10:               Β ← Σι,1
11:               Flag ← «Αληθής»
12:          Τέλος Εάν
13:     Μέχρι (flag = «Αληθής» Η ι = Τ)
14:     Εάν (flag = «Αληθής») Τότε
```

```
15:             Επέστρεψε β
16:     Αλλιώς
17:             Επέστρεψε 0
18:     Τέλος Εάν
19: Τέλος Συνάρτησης «Ανάκτηση»
```

Στο *Πρόγραμμα 1* που ακολουθεί περιγράφεται ένας διαφορετικός αλγόριθμος ανάκτησης στοιχείου (είναι καλύτερος;).

Πρόγραμμα 1: Ανάκτηση Στοιχείου Αραιού Πίνακα

```
#include <stdio.h>

#define M 7 /* Μέγεθος Πίνακα Στόχου */

int S[M][3]={ -3,3,9,
              -12,7,2,
              -5,12,4,
              -7,12,8,
              -31,12,9,
              -5,15,17,
              -9,20,1};
/* S είναι ο πίνακας στον οποίο μεταφέρθηκε ο αραιός πίνακας */

int anaktisi(int,int);

main()
{
        int i;
        int ii,jj; /* Δείκτες του υπό ανάκτηση στοιχείου */
        int s; /* Το αναζητούμενο στοιχείο */

        /* Εκτύπωση πίνακα στόχου */
        for (i=0;i<M;i++)
                printf("\n%5d%5d%5d",S[i][0],S[i][1],S[i][2]);

        /* Είσοδος παραμέτρων */
        printf("\n\nΕισαγωγή γραμμής και στήλης του ζητούμενου στοιχείου:");
        scanf("%d%d",&ii,&jj);

        /* Ανάκτηση στοιχείου */
        s=anaktisi(ii,jj);

        /* Εκτύπωση στοιχείου */
        printf("\n\nStoixeio = %5d\n\n",s);
}


/***************************/
/* ΣΥΝΑΡΤΗΣΗ ΑΝΑΚΤΗΣΗΣ */
```

```
/*************************/

int anaktisi(int i, int j)
{
        int k;

        k=0;
        while (S[k][1]<i && k<M-1)
                k++;

        if (S[k][1]==i)
                while (S[k][2]<j && k<M-1)
                        k++;
        if (S[k][1]==i && S[k][2]==j)
                return S[k][0];
        else
                return 0;
}
```

3. Τριγωνικοί Πίνακες. Ένας τετραγωνικός πίνακας ονομάζεται κάτω (επάνω) τριγωνικός εάν μόνο τα στοιχεία κάτω (επάνω) της κύριας διαγωνίου του είναι μη μηδενικά (η κύρια διαγώνιος συμπεριλαμβάνεται στα μη μηδενικά στοιχεία). Επειδή πάλι υπάρχει σπατάλη χώρου (πολλά μηδενικά στοιχεία) ζητείται μία μέθοδος απεικόνισης του πίνακα ώστε να αποφευχθεί αυτή η σπατάλη χώρου.

ΛΥΣΗ

Εάν ο τριγωνικός πίνακας μεταφερθεί σε ένα νέο μονοδιάστατο πίνακα, τότε το πρώτο ζητούμενο είναι το μέγεθος του πίνακα στόχου. Η γενική μορφή ενός κάτω τριγωνικού πίνακα είναι η ακόλουθη:

$$\begin{pmatrix} a_{1,1},0,0,\cdots\cdots\cdots\cdots,0 \\ a_{2,1},a_{2,2},0,0,\cdots\cdots\cdots,0 \\ a_{3,1},a_{3,2},a_{3,3},0,\cdots\cdots,0 \\ \cdots\cdots\cdots\cdots\cdots \\ a_{n,1},a_{n,2},a_{n,3},\cdots\cdots,a_{n,n} \end{pmatrix}$$

Δηλαδή τα μη μηδενικά στοιχεία ανά γραμμή είναι:

$1^{η}$ γραμμή: 1
$2^{η}$ γραμμή: 2
$3^{η}$ γραμμή: 3
…
$n^{η}$ γραμμή: n

Επομένως το σύνολο των μη μηδενικών στοιχείων είναι:

$$1+2+3+\cdots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}=N\,,$$

επομένως και το μέγεθος του γραμμών του μονοδιάστατου πίνακα στόχου πρέπει να είναι $N=\frac{n(n+1)}{2}$, όπου n το πλήθος των γραμμών / στηλών του τριγωνικού πίνακα.

Ο αλγόριθμος μεταφοράς δίνεται στον *Αλγόριθμο 5*.

Αλγόριθμος 5: Μεταφορά Στοιχείων Κάτω Τριγωνικού Πίνακα

```
1: Δεδομένα / Είσοδος: Κάτω Τριγωνικός Πίνακας  Tnxn, Πίνακας Στόχος ΣN
2: Διαδικασία «Μεταφορά Τριγωνικού»
3: Αρχή
4:      Ακέραιοι: ι, κ, λ = 0
5:      Για ι = 1 Μέχρι n
6:           Για κ = 1 Μέχρι ι
7:                λ ← λ+1
8:                Σλ ← Tι,κ
9:           Τέλος Για (κ)
10:     Τέλος Για (λ)
11: Τέλος διαδικασίας «Μεταφορά Τριγωνικού»
```

Το επόμενο πρόβλημα είναι πως θα γίνεται ανάκτηση ενός στοιχείου του τριγωνικού πίνακα που έχει πλέον μεταφερθεί, από τους δείκτες του, έστω i, j. Εάν i<j τότε το ζητούμενο στοιχείο είναι μηδενικό. Εάν όμως i≥j τότε το στοιχείο που βρίσκεται στη i γραμμή έπεται όλων των στοιχείων των προηγουμένων γραμμών i-1 και το πλήθος των στοιχείων αυτών είναι:

$1^{η}$ γραμμή: 1
$2^{η}$ γραμμή: 2
$3^{η}$ γραμμή: 3
…
$(i-1)^{η}$ γραμμή: i-1

Επομένως το σύνολο των μη στοιχείων είναι:

$$1+2+3+\cdots+(i-1)=\sum_{i=1}^{i-1} i=\frac{i(i-1)}{2}\,,$$

και επιπλέον προηγούνται και τα j-1 στοιχεία της ι γραμμής. Δηλαδή το σύνολο των στοιχείων που προηγούνται του i,j είναι:

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

και αυτό δείχνει ότι το στοιχείο i,j έχει καταχωρηθεί στον πίνακα στόχο στη θέση:

$$l = \sum_{i=1}^{i-1} i = \frac{i(i-1)}{2} + j\,.$$

Στον *Αλγόριθμο 6* περιγράφεται η διαδικασία ανάκτησης.

Αλγόριθμος 6: Ανάκτηση Στοιχείου Τριγωνικού Πίνακα

1: Δεδομένα / Είσοδος: Πίνακας Στόχος Σ_N
2: Τύπος Στοιχείου Τριγωνικού Πίνακα «Ανάκτηση ΚΤΠ(ακέραιος i, ακέραιος j)»
3: Αρχή
4: Ακέραιος k
5: Εάν (i < j) Τότε
6: Επέστρεψε 0
7: Τέλος Εάν
8: k ← $\frac{i(i-1)}{2} + j$
9: Επέστρεψε Σ_k
10: Τέλος συνάρτησης «Ανάκτηση ΚΤΠ»

Στο Πρόγραμμα 2 δίνεται η μεταφορά και ανάκτηση στοιχείου κάτω τριγωνικού πίνακα. Η διαφορά της συνάρτησης ανάκτησης του *Προγράμματος 2* με αυτής του *Αλγόριθμου 6* οφείλεται στο ότι στην C οι δείκτες των πινάκων αρχίζουν από το μηδέν.

Πρόγραμμα 2: Κάτω Τριγωνικός Πίνακας

```
#include <stdio.h>
#define n 4              /* Μέγεθος Κάτω Τριγωνικού Πίνακα */
#define N n*(n+1)/2   /* Μέγεθος Πίνακα Στόχου */

int T[n][n]={   -3,0,0,0,
                -12,7,0,0,
                -5,12,4,0,
                -7,12,8,9};

int S[N];

void metafora(void);
int anaktisi(int,int);

main()
{
  int i, j, l;

  /* Αρχικοποίηση πίνακα στόχου */
  for (i=0; i<N; i++)
    S[i] = 0;

  metafora();

  printf("\n\nΚΑΤΩ ΤΡΙΓΩΝΙΚΟΣ ΠΙΝΑΚΑΣ\n\n");
```

```
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      printf("%5d", T[i][j]);
    printf("\n");
  }

  printf("\n\nΠΙΝΑΚΑΣ ΣΤΟΧΟΣ\n\n");
  for (i=0; i<N; i++)
    printf("%5d", S[i]);
  printf("\n\n");

  printf("\nΕίσοδος δεικτών γραμμής και στήλης του υπό αναζήτηση στοιχείου:");
  scanf("%d%d",&i, &j);

  l = anaktisi(i,j);
  if (l==-1)
    printf("\n\nΤο στοιχείο είναι το: %d\n\n", 0);
  else
    printf("\n\nΤο στοιχείο είναι το: %d\n\n", S[l]);
}

/*************************/
/*  Σ Υ Ν Α Ρ Τ Η Σ Ε Ι Σ  */
/*************************/

void metafora(void)
{
  int i, k, l = 0;

  for (i=0; i<n; i++)
    for (k=0; k<=i; k++)
      S[l++] = T[i][k];
}

int anaktisi(int x, int y)
{
  if (x < y)
    return -1;

  return x*(x+1)/2 + y;
}
```

4. *Ένα ενδιαφέρον πρόβλημα είναι η ενοποίηση δύο ταξινομημένων πινάκων A_N και B_M σε ένα νέο αλλά πάλι ταξινομημένο πίνακα C_{N+M}. Μία απλή λύση είναι η μεταφορά του πρώτου στον C, μετά η μεταφορά του δεύτερου στον C και τέλος η ταξινόμηση του C. Αυτή είναι μία χρονοβόρα λύση γιατί χρειάζεται να ταξινομηθεί πάλι ο C και δεν εκμεταλλεύεται το γεγονός ότι οι A και B είναι ήδη ταξινομημένοι. Προφανώς, η καλύτερη και πιο γρήγορη λύση είναι να μεταφερθούν «ταυτόχρονα»*

οι Α και Β στον C ταξινομημένα. Στο Πρόγραμμα 3 δίνεται μία λύση αυτού του προβλήματος.

ΛΥΣΗ

Πρόγραμμα 3: Ενοποίηση Ταξινομημένων Πινάκων

```
#include <stdio.h>
#define N 5              /* Μέγεθος Α Πίνακα */
#define M 9              /* Μέγεθος Β Πίνακα */
#define F N+M            /* Μέγεθος Πίνακα Στόχου */

int A[N] = {2,6,7,9,9};
int B[M] = {-4,0,1,6,12,12,15,17,19};
int C[F];

void enopoiisi(void);

main()
{
  int i;

  printf("\n\nA ΠΙΝΑΚΑΣ\n\n");
  for (i=0; i<N; i++)
    printf("%5d", A[i]);

  printf("\n\nB ΠΙΝΑΚΑΣ\n\n");
  for (i=0; i<M; i++)
    printf("%5d",B[i]);

  enopoiisi();

  printf("\n\nΕνοποιημένος Πίνακας\n\n");
  for (i=0; i<F; i++)
    printf("%5d", C[i]);
  printf("\n\n");
}

/************************/
/*  Σ Υ Ν Α Ρ Τ Η Σ Ε Ι Σ   */
/************************/

void enopoiisi(void)
{
  int i, j, k;

  for (i=0, j=0, k=0; k<F; k++) {
    if (i == N) { C[k] = B[j++]; continue; }
    if (j == M) { C[k] = A[i++]; continue; }
    C[k] = (A[i]<B[j]) ? A[i++] : B[j++];
  }
```

```
}
```

1.6 Ασκήσεις

1) Οι συμμετρικοί πίνακες είναι μία ιδιαίτερη κατηγορία τετραγωνικών πινάκων οι οποίοι μάλιστα μοιάζουν με τους τριγωνικούς. Σε έναν συμμετρικό πίνακα ισχύει $a_{i,j} = a_{j,i}$. Δηλαδή, υπάρχει μία κατοπτρική συμμετρία των στοιχείων του σε άξονα συμμετρίας την κύρια διαγώνιο του πίνακα. Με βάση το σκεπτικό των τριγωνικών πινάκων και επειδή πάλι υπάρχει σπατάλη μνήμης λόγω τον πολλών όμοιων στοιχείων, να βρεθεί τρόπος οικονομικότερης διαχείρισης αυτού του είδους των πινάκων και να αναπτυχθούν οι αντίστοιχοι αλγόριθμοι και προγράμματα σε οποιαδήποτε γλώσσα προγραμματισμού.

2) Να αναπτυχθούν αλγόριθμοι ταξινόμησης πίνακα με τις παρακάτω μεθόδους:
   a) Φυσαλίδας,
   b) Παρεμβολής, και
   c) Εισαγωγής.

3) Να γραφεί αλγόριθμος υπολογισμού του μεγαλύτερου και μικρότερου στοιχείου πίνακα. Ο αλγόριθμος να επιστρέφει και την θέση στον πίνακα που βρέθηκαν το μεγαλύτερο και μικρότερο στοιχείο αντίστοιχα.

4) Να αναπτυχθούν αλγόριθμοι αναζήτησης στοιχείου πίνακα (και της θέσης που βρέθηκε) με τις παρακάτω μεθόδους. Επίσης να συγκριθούν αυτοί οι αλγόριθμοι ως προς την ταχύτητά τους.
   a) Σειριακή σε αταξινόμητο πίνακα,
   b) Σειριακή σε ταξινομημένο πίνακα,
   c) Δυαδική αναζήτηση (φυσικά σε ταξινομημένο πίνακα).

5) Να τροποποιηθούν οι αλγόριθμοι των μεθόδων (a) και (b) της προηγούμενης άσκησης με τέτοιο τρόπο ώστε εάν υπάρχουν παραπάνω από ένα ίδια στοιχεία με αυτό το υπό αναζήτηση να υπολογίζεται το πλήθος τους αλλά και η θέση τους.

6) *Quiz*: Έστω ότι υπάρχουν δώδεκα αντικείμενα ίδιου μεγέθους και χρώματος. Επίσης, έστω ότι όλα τα αντικείμενα, εκτός από ένα, έχουν το ίδιο βάρος. Εάν διαθέτουμε μία ζυγαριά (παλιού τύπου, με δύο ζυγούς) να βρεθεί ο τρόπος ώστε με το πολύ τέσσερις ζυγίσεις να βρεθεί το αντικείμενο το οποίο είναι διαφορετικού βάρους αλλά και το είδος της διαφορετικότητάς του (πιο ελαφρύ ή πιο βαρύ).

7) Να σχεδιασθεί αλγόριθμος, ο οποίος να επιλύει το προηγούμενο πρόβλημα αλλά με N αντικείμενα (εκ των οποίων ένα είναι διαφορετικού βάρους). Επίσης, να αποδειχθεί ο ελάχιστος αριθμός των απαιτούμενων ζυγίσεων.

2. ΠΟΛΥΠΛΟΚΟΤΗΤΑ

Πριν την σχεδίαση ενός αλγορίθμου, είναι στοιχειωδώς απαραίτητο να ορισθεί το κριτήριο ή κριτήρια τα οποία πρέπει να εκπληρώνει ώστε να μπορεί να χαρακτηρισθεί εάν είναι «καλός» αλγόριθμος ή όχι. Το βασικό στοιχείο ενός καλού αλγόριθμου λοιπόν, είναι το κατά πόσον είναι ή όχι αποδοτικός. Και απόδοση σημαίνει η χρήση υπολογιστικών πόρων που απαιτούνται για την επίλυση του προβλήματος. Οι υπολογιστικοί πόροι (ανάλογα και με την φύση του αλγόριθμου) μπορεί να είναι η CPU, η μνήμη ή ακόμη και πιθανούς δικτυακούς πόρους (πχ bandwidth) που χρησιμοποιούνται. Ουσιαστικά όμως, στα περισσότερα προβλήματα αυτό που εξετάζεται είναι η χρήση της CPU δηλαδή, πόσος χρόνος απαιτείται για την εκτέλεση του αλγόριθμου. Αυτό όμως προϋποθέτει να δοθεί και τυποποιηθεί ένα υπολογιστικό μοντέλο ώστε όλοι να αναφέρονται σε αυτό. Αυτή η υπολογιστική μηχανή θεωρείται όσο πιο απλή όσο είναι δυνατόν. Δηλαδή, είναι σε θέση να εκτελεί μόνο στοιχειώδεις αριθμητικές πράξεις (πχ πρόσθεση, αφαίρεση, κλπ) και επίσης στοιχειώδεις συγκρίσεις (πχ είναι το a>4;). Η υπολογιστική αυτή μηχανή ονομάζεται ***RAM*** από το ακρωνύμιο *Random Access Machine*. Η RAM, αποτελείται από ένα μόνο επεξεργαστή (επομένως δεν είναι δυνατή η παράλληλη επεξεργασία), και εκτελεί τις οδηγίες του αλγόριθμου σειριακά, δηλαδή μία την φορά. Τέλος, θεωρείται ότι η μνήμη της είναι μεγάλη ή τουλάχιστον τόσο μεγάλη όσο απαιτεί ο οποιοσδήποτε αλγόριθμος.

2.1 Ορισμοί

Η έννοια της ***Πολυπλοκότητα και Απόδοσης Αλγόριθμου*** αντιπροσωπεύει το κόστος χρήσης του αλγόριθμου για την επίλυση ενός προβλήματος. Το πρόβλημα χαρακτηρίζεται από τα δεδομένα εισόδου ενώ η συνάρτηση της πολυπλοκότητας *f(n)* εκφράζει την απαίτηση του αλγόριθμου σε χρόνο εκτέλεσης σε σχέση με το μέγεθος των δεδομένων εισόδου *n*. Επειδή είναι ιδιαίτερα δύσκολο ή και αδύνατο σε πάρα πολλές περιπτώσεις να βρεθεί μία ακριβής συνάρτηση, συνήθως σ' αυτήν την περίπτωση ενδιαφέρει η εύρεση της τιμής της *f(n)* στις εξής περιπτώσεις:

⇒ Καλύτερη Περίπτωση, δηλ. η ελάχιστη τιμή της *f(n)*
⇒ Χειρότερη Περίπτωση, δηλ. η μέγιστη τιμή της *f(n)*
⇒ Μέση Περίπτωση, δηλ. η αναμενόμενη τιμή της *f(n)*

με πλέον χρήσιμη την χειρότερη περίπτωση.

Όμως το πρόβλημα της ανάλυσης ενός αλγόριθμου παραμένει όταν τα δεδομένα εισόδου *n*, αυξάνονται πάρα πολύ. Γι' αυτό είναι απαραίτητη η χρήση και επίδειξη ασυμπτωτικών συναρτήσεων οι οποίες να δείχνουν την τάξη μεγέθους της αύξησης του υπολογιστικού χρόνου χωρίς να είναι απαραίτητο να δείχνουν την ακριβή συνάρτηση της πολυπλοκότητάς του. Έτσι, είναι ιδιαίτερα χρήσιμοι οι ορισμοί που ακολουθούν.

- ***Ορισμός 1***. Θεωρούμε ότι *f(n)=o(g(n))* για $n \to \infty$ (μικρό όμικρον – little oh) εάν $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Δηλαδή, η συνάρτηση *f* αυξάνει πιο αργά από την συνάρτηση *g* όταν το *n* είναι πολύ μεγάλο.

Παραδείγματα:
- $n^2 = o(n^5)$
- $\eta\mu(n) = o(n)$

- ***Ορισμός 2***. Θεωρούμε ότι *f(n)=O(g(n))* για $n \to \infty$ (μεγάλο όμικρον – big oh) εάν $\exists C, n_0$, τέτοια ώστε $|f(n)| \le Cg(n), (\forall n > n_0)$. Δηλαδή, η συνάρτηση *f* δεν αυξάνει πιο γρήγορα από την συνάρτηση *g*.

Παραδείγματα:
- $\eta\mu(n) = O(1)$
- $\frac{n(n+1)}{2} = O(n^2)$.
  - *Απόδειξη*: $f(n) = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$ με επικρατούντα όρο τον $\frac{1}{2}n^2$. Επομένως $f(n) = O(n^2)$.
- $\frac{8\log(n) + 4n}{3n^2 \log(n)} = O(\frac{1}{n\log(n)})$.
  - *Απόδειξη*: $f(n) = \frac{8\log(n)}{3n^2 \log(n)} + \frac{4n}{3n^2 \log(n)} = \frac{8}{3n^2} + \frac{4}{3n\log(n)}$. Ο επικρατών όρος είναι ο δεύτερος γιατί $n^2 \ge n\log(n)$ και επομένως $f(n) = O(\frac{1}{n\log(n)})$.

- Γενικά οι σημαντικότερες συναρτήσεις τάξης μεγέθους παρουσιάζουν την ακόλουθη διάταξη (αύξουσα):
  $1 \le \log(n) \le n \le n\log(n) \le n^2 \le n^3 \le 2^n \le \cdots$

- ***Ορισμός 3***. Θεωρούμε ότι $f(n) = \Theta(g(n))$ (θήτα) εάν υπάρχουν σταθερές $c_1 \ne 0, c_2 \ne 0, n_0$ τέτοιες ώστε $\forall n > n_0 \Rightarrow c_1 g(n) < f(n) < c_2 g(n)$. Δηλαδή οι δύο συναρτήσεις, *f* και *g*, αυξάνουν με τον ίδιο περίπου ρυθμό.

Παραδείγματα:
- $(n+1)^2 = \Theta(3n^2)$
- $\frac{n^2 + 5n + 7}{5n^3 + 7n + 2} = \Theta(\frac{1}{n})$

- ***Ορισμός 4***. Θεωρούμε ότι $f(n) \sim g(n)$ (ασυμπτωτικά) εάν $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$. Δηλαδή όχι μόνον ότι οι δύο συναρτήσεις *f* και *g* αυξάνουν με τον ίδιο

περίπου ρυθμό αλλά και ότι το πηλίκο τους $\frac{f}{g}$ προσεγγίζει την μονάδα όσο $n \to \infty$

Παραδείγματα:

- $n^2 + n \sim n^2$
- $(3n+1)^4 \sim 81n^4$

- ***Ορισμός 5***. Θεωρούμε ότι $f(n) = \Omega(g(n))$ (ωμέγα) εάν $\exists \varepsilon > 0$ και μία ακολουθία $n_1, n_2, n_3, \ldots \to \infty$ τέτοια ώστε $\forall i : |f(n_i)| > \varepsilon g(n_i)$. Στην πραγματικότητα αυτό δηλώνει την άρνηση του '*o*'. Δηλαδή, εάν $f(n) = \Omega(g(n)) \Rightarrow not(f(n) = o(g(n)))$. Για παράδειγμα, είναι γνωστό ότι ο πολλαπλασιασμός δύο πινάκων *nXn* δεν μπορεί με κανένα τρόπο να επιτευχθεί με λιγότερους από n^2 πολλαπλασιασμούς. Αυτό σημαίνει ότι υπάρχει κατώτερο όριο και ότι η πολυπλοκότητα του γινομένου εκπεφρασμένου με την συνάρτηση *Ω* είναι $\Omega(n^2)$.

Μερικές χρήσιμες μαθηματικές σειρές:

Αριθμητική σειρά: $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2)$

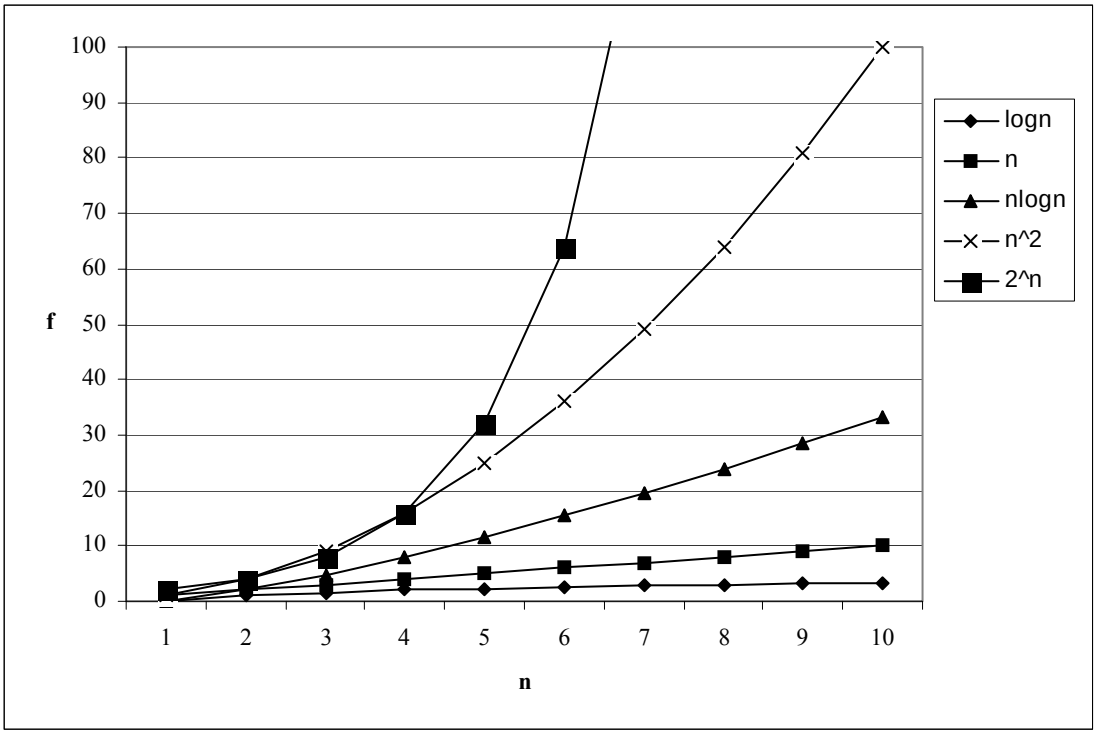
Γεωμετρική σειρά: $\sum_{i=0}^{n} x^i = 1 + x + x^2 + x^3 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1} = \Theta(x^n)$

Αρμονική σειρά: $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) = \Theta(\ln(n))$

Τέλος, στο επόμενο σχήμα (*Σχήμα 2.1*) φαίνεται ο ρυθμός αύξησης των συναρτήσεων $f(n) = \log n$, $f(n) = n$, $f(n) = n \log n$, $f(n) = n^2$, και $f(n) = 2^n$.

2.2 Αλγόριθμοι Brute-Force

Μία κατηγορία αλγορίθμων είναι οι λεγόμενοι brute-force αλγόριθμοι. Με βάση αυτή τη τεχνική, το πρόβλημα επιλύεται με τον πιο απλό και προφανή τρόπο, ο οποίος όμως δεν είναι πάντα (σχεδόν σε όλες τις περιπτώσεις) και ο πιο καλός. Επειδή ακριβώς επιλύει τα προβλήματα με τον πιο απλό και προφανή δυνατό τρόπο, είναι συνήθως εύκολο να υλοποιηθεί και να γίνει κατανοητός. Συνήθως αποτελούν και τον πιο «κουτό» τρόπο επίλυσης ενός προβλήματος. Ένα χαρακτηριστικό παράδειγμα τέτοιας τεχνικής είναι η ταξινόμηση ενός πίνακα με την μέθοδο της φυσαλίδας όπως επίσης και η σειριακή αναζήτηση. Σαν απόδειξη του πόσο αργοί είναι οι αλγόριθμοι αυτού του είδους, η μέθοδος της φυσαλίδας έχει πολυπλοκότητα της τάξης $O(n^2)$ ενώ άλλες τεχνικές ταξινόμησης (πχ quick sort) έχουν πολυπλοκότητα της τάξης $O(n \log n)$.



Σχήμα 2.1 Ρυθμός αύξησης συναρτήσεων

Ένα χαρακτηριστικό παράδειγμα για την διευκρίνιση της τεχνικής Brute-Force είναι το πώς θα υπολόγιζε μία δύναμη, για παράδειγμα την δύναμη a^{16}:

Brute-Force: $\underbrace{a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a}_{16}$

Ενώ με κάποια άλλη τεχνική (πολύ καλύτερη) θα μπορούσε να υπολογισθεί σαν:

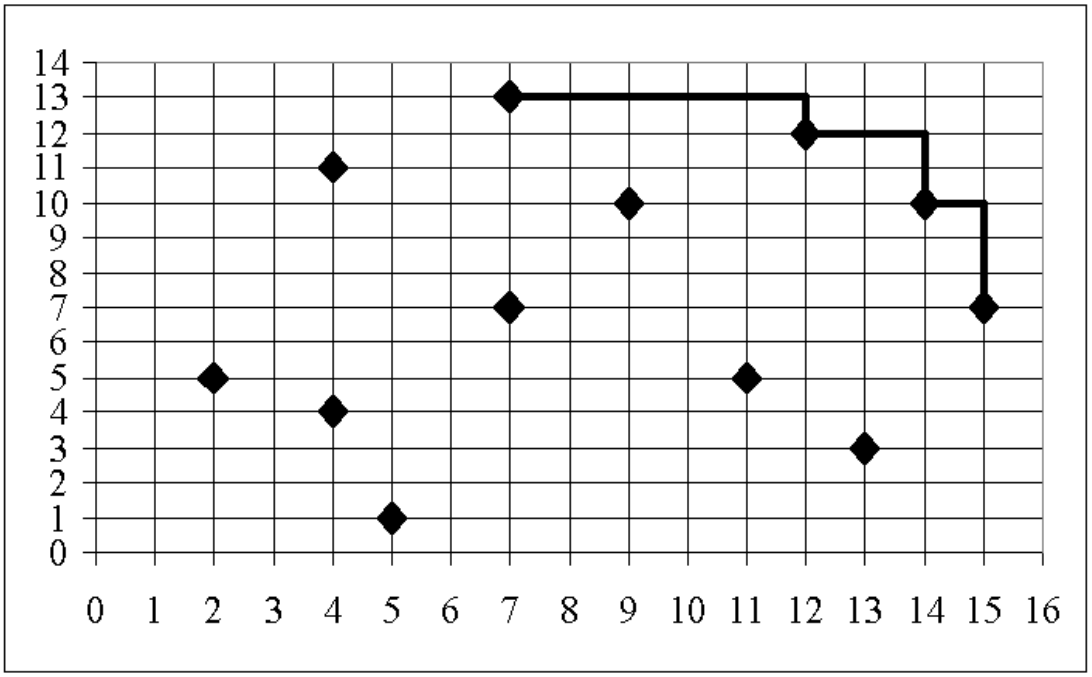
Καλύτερη τεχνική: $(((a^2)^2)^2)^2$

δηλαδή 4 πολλαπλασιασμοί αντί για 16.

2.3 Δισδιάστατα Μέγιστα

Ένα πολύ ενδιαφέρον και χρήσιμο πρόβλημα είναι το πρόβλημα της εύρεσης μεγαλύτερων στοιχείων σε δισδιάστατο χώρο (2-dimension maxima). Ένα απλό παράδειγμα που το περιγράφει αρκετά καλά είναι το ακόλουθο: Εάν υποτεθεί ότι κάποιος θέλει να αγοράσει ένα αυτοκίνητο και ότι αυτό που τον συγκινεί περισσότερο είναι η τελική ταχύτητα του αυτοκινήτου. Βέβαια επειδή δεν διαθέτει πολλά χρήματα τον ενδιαφέρει επίσης να είναι όσο πιο φτηνό γίνεται. Φυσικά τα πιο γρήγορα αυτοκίνητα δεν είναι και φτηνά. Ο υποψήφιος αγοραστής δεν μπορεί να αποφασίσει αν τον ενδιαφέρει πιο πολύ η ταχύτητα ή η τιμή αλλά αυτό που ξέρει σίγουρα είναι ότι δεν θα ήθελε να αγοράσει ένα αυτοκίνητο εάν υπάρχει κάποιο άλλο το οποίο είναι και φτηνότερο και ταχύτερο.

Το πρόβλημα αυτό ανάγεται στην εύρεση μέγιστου ή και μέγιστων σε ένα δισδιάστατο χώρο όπου οι συντεταγμένες αναπαριστούν (στην προκείμενη περίπτωση) η μία τιμή και η άλλη ταχύτητα. Στο προκείμενο πρόβλημα το αυτοκίνητο που ενδιαφέρει είναι αυτό για το οποίο ο συνδυασμός ταχύτητας / τιμής είναι βέλτιστος. Το πρόβλημα δεν είναι δυνατόν να λυθεί εάν δεν τυποποιηθεί πρώτα μαθηματικά. Αυτή η τυποποίηση θα βοηθήσει τόσο στην καλύτερη κατανόηση του προβλήματος όσο και στην γενίκευση της λύσης του.



Σχήμα 2.2 Δισδιάστατα μέγιστα

Η μαθηματική τυποποίηση του προβλήματος είναι η ακόλουθη:

- Εάν p αναπαριστά ένα σημείο στον δισδιάστατο χώρο με συντεταγμένες p=(p.x, p.y) τότε εάν δεν υπάρχει άλλο σημείο q τέτοιο ώστε p.x<q.x ΚΑΙ p.y<q.y τότε το σημείο p θεωρείται μέγιστο (δεν καλύπτεται από κανένα άλλο).

- Στο πρόβλημα της αγοράς αυτοκινήτου, εάν στον άξονα x απεικονισθούν οι ταχύτητες των διαθέσιμων αυτοκινήτων και στον άξονα y οι αρνητικές τιμές των τιμών (ώστε όσο αυξάνει ο y να μειώνονται στην πραγματικότητα οι τιμές) τότε η εύρεση των μέγιστων αποτελούν τις καλύτερες περιπτώσεις αγοράς αυτοκινήτων.

ΠΡΟΣΟΧΗ: Είναι δυνατόν να υπάρχουν πολλά μέγιστα όπως φαίνεται στο παρακάτω *Σχήμα 2.2*.

Ένας σχετικά απλός αλλά καθόλου «έξυπνος» αλγόριθμος είναι το να ελέγχει όλα τα σημεία σε σχέση με όλα τα υπόλοιπα (brute-force algorithm). Δηλαδή για όλα τα σημεία να ελέγχει εάν όλα τα υπόλοιπα έχουν και τις δύο συντεταγμένες τους μικρότερες από το ελεγχόμενο. Η αναλυτική περιγραφή του αλγόριθμου δίνεται στον *Αλγόριθμο 7*.

Αλγόριθμος 7. Δισδιάστατα Μέγιστα – Αλγόριθμος Brute-Force

```
1: Δεδομένα / Είσοδος: Πίνακας Π_n, n = πλήθος σημείων
2: Διαδικασία Δισδιάστατα Μέγιστα1
3: Αρχή
4:      Ακέραιοι: i, j
5:      Boolean: Μέγιστο
6:      Για i από 1 μέχρι n
7:           Έστω Μέγιστο ← Αληθές
8:           Για j από 1 μέχρι n
9:                Εάν (i≠j) ΚΑΙ (Π_i.x ≤ Π_j.x) ΚΑΙ (Π_i.y ≤ Π_j.y) Τότε
10:                    Μέγιστο ← Ψευδές
11:               Τέλος Εάν
12:          Τέλος Για (j)
13:          Εάν (Μέγιστο) Τότε Επέστρεψε Π_ι
14:     Τέλος Για (i)
15: Τέλος Αλγόριθμου «Δισδιάστατα Μέγιστα1»
```

(*Παρατήρηση*: Αποτελεί στοιχειώδης διαδικασία η απόδειξη (και μαθηματική εάν χρειασθεί) της ***ορθότητας ενός αλγόριθμου***, δηλαδή για κάθε αλγόριθμο πρέπει να αποδεικνύεται μαθηματικά ότι είναι ορθός. Βέβαια στην προκείμενη περίπτωση είναι προφανές μιας και εξετάζονται όλα τα στοιχεία σε σχέση με όλα τα υπόλοιπα.)

Ο παραπάνω αλγόριθμος φυσικά είναι σωστός, αλλά το πρόβλημα που τίθεται είναι ***πόσο γρήγορος είναι***. Δηλαδή ποια είναι η πολυπλοκότητά του. Είναι φανερό ότι για κάθε τιμή του εξωτερικού βρόγχου ο εσωτερικός εκτελείται n φορές. Επίσης εάν υποθέσουμε ότι μετράμε το πόσες φορές θα προσπελασθεί κάποιο σημείο του πίνακα Π τότε ο εσωτερικός βρόγχος θα προσπελάσει 4 φορές κάποιο σημείο του πίνακα ενώ ο εξωτερικός 2 (είναι το σημείο που θα επιστραφεί, και στην χειρότερη περίπτωση που όλα τα σημεία είναι μέγιστα). Επομένως, η πολυπλοκότητα του αλγόριθμου, εκφρασμένη ανάλογα με τα δεδομένα εισόδου είναι:

$$T(n) = \sum_{i=1}^{n}(2 + \sum_{j=1}^{n} 4) = n(2+4n) \Rightarrow T(n) = 4n^2 + 2n$$

Επομένως η πολυπλοκότητα της μεθόδου είναι της τάξης $O(n^2)$.

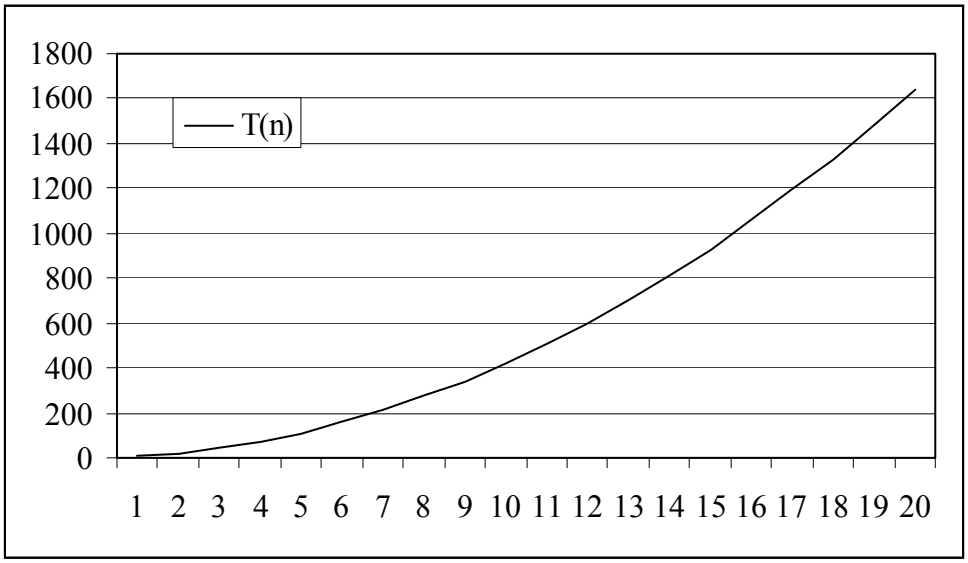
Βέβαια αυτό που πρώτιστα ενδιαφέρει είναι η απόδοση του αλγόριθμου όταν το n γίνει μεγάλο (για μικρό n όλοι οι αλγόριθμοι είναι γρήγοροι). Επομένως με κάποιο τρόπο πρέπει να υπολογισθεί η αύξηση της βραδύτητας του αλγόριθμου για μεγάλα n, δηλαδή ο ρυθμός αύξησης του χρόνου όσο το n αυξάνει. Εάν ο παραπάνω αλγόριθμος βελτιωθεί όπως φαίνεται στην *Αλγόριθμο 8* τότε αυτό που μόλις υπολογίσθηκε αποτελεί την χειρότερη περίπτωση (γιατί;).

Αλγόριθμος 8. Δισδιάστατα Μέγιστα – Δεύτερος Αλγόριθμος

```
1: Δεδομένα / Είσοδος: Πίνακας Π_n, n = πλήθος σημείων
2: Διαδικασία Δισδιάστατα Μέγιστα2
3: Αρχή
4:      Ακέραιοι: i, j
```

```
5:      Boolean: Μέγιστο
6:      Για i από 1 μέχρι n
7:              Έστω Μέγιστο ← Αληθές
8:              Για j από 1 μέχρι n
9:                      Εάν (i≠j) ΚΑΙ (Π_i.x ≤ Π_j.x) ΚΑΙ (Π_i.y ≤ Π_j.y)  Τότε
10:                             Μέγιστο ← Ψευδές
11:                             Τέλος εσωτερικού βρόγχου
12:                     Τέλος Εάν
13:             Τέλος Για (j)
14:             Εάν (Μέγιστο) Τότε Επέστρεψε Π_ι
15:     Τέλος Για (i)
16: Τέλος Αλγόριθμου «Δισδιάστατα Μέγιστα2»
```

Όπως έχει ήδη αποδειχθεί, ο ρυθμός αύξησης του αλγόριθμου είναι: $T(n) = 4n^2 + 2n$. Δηλαδή, όπως φαίνεται στο παρακάτω σχήμα (*Σχήμα 2.3*):



Σχήμα 2.3 Συνάρτηση Πολυπλοκότητας Αλγόριθμου Δισδιάστατων Μεγίστων

Στο *Πρόγραμμα 4* δίνεται ο υπολογισμός δισδιάστατων μεγίστων (*Αλγόριθμος 8*) στην γλώσσα προγραμματισμού C.

Πρόγραμμα 4: Υπολογισμός Δισδιάστατων Μεγίστων

```
#include <stdio.h>
#define N 12 /* πλήθος σημείων */

/* Περιγραφή της δομής «Σημείο» */
struct Simio {
        int x;
        int y;
};
struct Simio P[N]; /* Πίνακας σημείων */

main()
{
```

```
        int i,j,k=0,l=0;
        int megisto;

        /* Εισαγωγή σημείων */
        for (i=0; i<N; i++) {
                printf("\Εισαγωγή των συντεταγμένων του %3d στοιχείου:",i);
                scanf("%d%d",&P[i].x,&P[i].y);
        }

        printf("\n\n ΥΠΟΛΟΓΙΣΜΟΣ ΜΕΓΙΣΤΩΝ \n\n");
        for (i=0; i<N; i++) {
                megisto = 0;
                for (j=0; j<N; j++) {
                        l++; /* μετρητής αριθμού επαναλήψεων */
                        if ((i!=j) && (P[i].x<=P[j].x) && (P[i].y<=P[j].y)) {
                                megisto=1;  break;
                        }
                }
                if (megisto==0) {
                        k++;
                        printf("\n%d-->%5d,%5d",k,P[i].x,P[i].y);
                }

        }
        printf("\n\nΑριθμός επαναλήψεων:%5d\n\n",l);
}
```

2.4 Το Πρόβλημα των Ελαχίστων Αποστάσεων (closest-pair problem)

*Ένα ενδιαφέρον πρόβλημα είναι το εξής: Δοθέντων n σημείων στον δισδιάστατο χώρο, να βρεθούν τα δύο εκείνα σημεία τα οποία η απόσταση μεταξύ τους είναι η πιο μικρή δυνατή (**closest-pair problem**). Μία brute-force τεχνική είναι να υπολογισθούν όλες οι αποστάσεις και στη συνέχεια να επιλεγεί η πιο μικρή. Τα σημεία που την δημιουργούν είναι και τα πιο «κοντινά» σημεία. α) Να αναπτυχθεί αυτός ο αλγόριθμος και στη συνέχεια β) να υλοποιηθεί σε κάποια γλώσσα προγραμματισμού.*

ΛΥΣΗ

Αλγόριθμος 9. Πρόβλημα Ελαχίστων Αποστάσεων (Brute-Force)

1: Δεδομένα / Είσοδος: Πίνακας Π_n, n = πλήθος σημείων
2: Διαδικασία Υπολογισμού Ελαχίστων Αποστάσεων
3: Ακέραιοι: i,j,k,d1,d2, k=0
4: Ακέραιος M = n*(n-1)
5: Πίνακας πραγματικών A_M
6: Πραγματικός: el
7: Για i από 1 μέχρι n

8: Για j από 1 μέχρι n
9: $A_k \leftarrow \sqrt{\left|P_i.x^2 - P_j.x^2\right| + \left|P_i.y^2 - P_j.y^2\right|}$
10: Εάν (k = 0) Τότε *(αρχικοποίηση τιμών)*
11: el ← A_k
12: d1 ← 0
13: d2 ← 1
14: Αλλιώς
15: Εάν (el > A_k)
16: el ← A_k
17: d1 ← I
18: d2 ← j
19: Τέλος Εάν
20: Τέλος Εάν
21: k ← k+1
22: Τέλος Για (j)
23: Τέλος Για (i)
24: Εκτύπωσε «Τα σημεία που απέχουν ελάχιστα είναι τα: d1 και d2»
25: Εκτύπωσε «και η απόστασή τους είναι: el»
26: Τέλος Αλγόριθμου «Υπολογισμού Ελαχίστων Αποστάσεων»

Πρόγραμμα 5: Πρόβλημα Ελαχίστων Αποστάσεων

```
#include <stdio.h>
#include <math.h>

#define N 3

struct Simio {
        float x;
        float y;
};

struct Simio P[N] ;

float apostasi(struct Simio, struct Simio);

main()
{
  int i, j, k=0, M = N*(N-1), d1, d2;
  float el; /* Ελάχιστη τιμή */
  float a[M];

  /* Εισαγωγή σημείων */
  for (i=0; i<N; i++) {
    printf("\n Εισαγωγή των συντεταγμένων του %3d σημείου:",i+1);
    scanf("%f%f",&P[i].x,&P[i].y);
  }

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

```
    for (j=i+1; j<N; j++) {
      a[k] = apostasi(P[i],P[j]);

      if (k == 0) { /* αρχικοποίηση τιμών */
        el = a[k];
        d1 = 0;
        d2 = 1;
      }
      else
      if (el > a[k]) {
        el = a[k];
        d1 = i;
        d2 = j;
      }
      k++;
    }

  printf("\n\n Τα σημεία που απέχουν ελάχιστα είναι τα: %3d και %3d",d1+1,d2+1);
  printf("\n\n και η απόστασή τους είναι: %.2f\n\n",el);

}

/* Σ Υ Ν Α Ρ Τ Η Σ Η υπολογισμού απόστασης */

float apostasi(struct Simio A, struct Simio B)
{
  float d;

  d = (A.x*A.x-B.x*B.x)+(A.y*A.y-B.y*B.y);
  if (d < 0)
    d = -d;
  return sqrt(d);
}
```

2.5 Εύρεση Αλφαριθμητικών

Ένα ιδιαίτερα σημαντικό πρόβλημα είναι η αναζήτηση ενός αλφαριθμητικού μέσα σε ένα κείμενο (ότι ακριβώς η λειτουργία grep στα λειτουργικά συστήματα τύπου UNIX, ή η λειτουργία «Εύρεση» των περισσοτέρων επεξεργαστών κειμένου). Εδώ υπάρχουν δύο ενδεχόμενα: 1) να αναζητά ακριβώς το ίδιο αλφαριθμητικό, και 2) να αναζητά όσο το δυνατόν πλησιέστερα σχήματα με το υπό αναζήτηση αλφαριθμητικό (δηλαδή να βρει κάτι παραπλήσιο). Στην δεύτερη περίπτωση ανάγεται και η αναζήτηση γενετικών κωδίκων. Η μοριακή αλυσίδα του DNA μπορεί να διασπασθεί σε μεγάλες σειρές οι οποίες αποτελούνται από τέσσερις βασικούς τύπους, τους C, G, T και A. Το να βρεθεί στη βιολογία ακριβές ταίριασμα είναι μάλλον απίθανο και αυτό που είναι το ζητούμενο είναι να βρεθεί ένας βαθμός ομοιότητας. Ένας τρόπος μέτρησης της ομοιότητας είναι το μέγεθος της σειράς που είναι ακριβώς ίδιες.

Ένα απλός brute-force αλγόριθμος δίνεται στον *Αλγόριθμο 10* (για απόλυτο ταίριασμα συμβολοσειρών) και η υλοποίησή του στο *Πρόγραμμα 6*. Η απλοϊκή λύση του προβλήματος στηρίζεται στο ψάξιμο όλων των συμβολοσειρών ένα προς ένα χαρακτήρων τους μέχρι να βρεθεί ή όχι η προς αναζήτηση συμβολοσειρά. Έστω ότι $T_M = \{x_1 x_2 \ldots x_M\}$ είναι η συμβολοσειρά που αναζητάτε για το εάν εμφανίζεται στην συμβολοσειρά $S_N = \{y_1 y_2 \ldots y_N\}$, $M < N$. Ο brute-force αλγόριθμος ψάχνει εάν υπάρχει μία ή και περισσότερες *M*-άδες χαρακτήρων ακριβώς ίδιες στην *S*. Για να το επιτύχει, ξεκινάει από τον πρώτο χαρακτήρα και συγκρίνει τον χαρακτήρα του *T* με τον αντίστοιχο του *S*. Εάν είναι ίδιοι προχωράει με τους επόμενους *M*-1 του *T*. Εάν όλοι είναι ίδιοι, τότε το πρώτο «ταίριασμα» βρέθηκε και συνεχίζει με το ίδιο τρόπο μέχρι το τέλος του *S* το οποίο βέβαια είναι ο χαρακτήρας *N*-*M*. Εάν κάποιος από τους ενδιάμεσους χαρακτήρες δεν είναι ίδιος με τον αντίστοιχο του *S*, τότε πάλι αρχίζει από την αρχή την αναζήτηση με τον πρώτο χαρακτήρα του *T* και με τον χαρακτήρα του *S* που σταμάτησε το προηγούμενο ταίριασμα.

Αλγόριθμος 10: Αναζήτηση συμβολοσειράς (αλγόριθμος brute-force)

1: Είσοδος / Δεδομένα: Συμβολοσειρά αναζήτησης T μεγέθους M, και συμβολοσειρά που θα αναζητηθεί S μεγέθους N.
2: Ακέραιοι: i←0, j ←0, k←0
3: Εφόσον (i < N-M+1) /* Η συμβολοσειρά που αναζητάτε να μην ξεπερνάει σε μήκος την συμβολοσειρά στην οποία γίνεται η αναζήτηση */
4: Εφόσον ($T_j = S_i$)
5: i ← i +1 /* προχώρησε στο επόμενο γράμμα */
6: i ← j +1 /* προχώρησε στο επόμενο γράμμα */
7: Τέλος Εφόσον
8: Εάν (j = M) Τότε /* βρέθηκε */
8: k ← k+1 /* αυξάνεται ο αριθμός των φορών που βρέθηκε */
10: Τέλος Εάν
11: Εάν (j > 0 ΚΑΙ j < M) /* Εάν δεν βρέθηκε το i πρέπει να πάρει την τιμή της τελευταίας αναζήτησης */
12: i ← i –1
13: Τέλος Εάν
14: j ← 0 Αρχίζει η αναζήτηση από την αρχή */
15: i ← i +1
16: Τέλος Εφόσον
17: Τέλος Αλγόριθμου «Αναζήτηση συμβολοσειράς (αλγόριθμος brute-force)»

Πρόγραμμα 6: Αναζήτηση συμβολοσειράς (αλγόριθμος brute-force)

```
#include <stdio.h>
#define N 26
#define M 3

main()
{
  char T[M] = "EFG";
  char S[N] = "ABCDEFGHIJKLMNOPEFEFGTVUWX";
  int i=0, j=0, k=0;

  /* String Matching */
```

```
  while (i<N-M+1) {

    while (T[j] == S[i]) {
      i++;
      j++;
    }
    if (j == M) {
      k++;
      printf("\n Βρέθηκε για  %d φορά στην θέση %d", k, i-M);
    }
    if (j>0 && j<M)
      i--;
    j = 0;
    i++;
  }

  if (k == 0)
    printf("\nΔεν βρέθηκε\n");
}
```

Βέβαια, το ζητούμενο είναι να βρεθεί ένας πιο γρήγορος και πιο έξυπνος τρόπος αντιμετώπισης αυτού του προβλήματος επειδή η πολυπλοκότητά του είναι της τάξης $O(M \cdot N)$.

2.6 Ασκήσεις

1. Να υπολογισθεί η πολυπλοκότητα εκπεφρασμένη με τη συνάρτηση O της ταξινόμησης με την μέθοδο της φυσαλίδας.

2. Αφού αναπτυχθούν οι αλγόριθμοι απλής σειριακής και δυαδικής αναζήτησης, να υπολογισθούν και συγκριθούν οι πολυπλοκότητές τους (εκπεφρασμένες με τη συνάρτηση O).

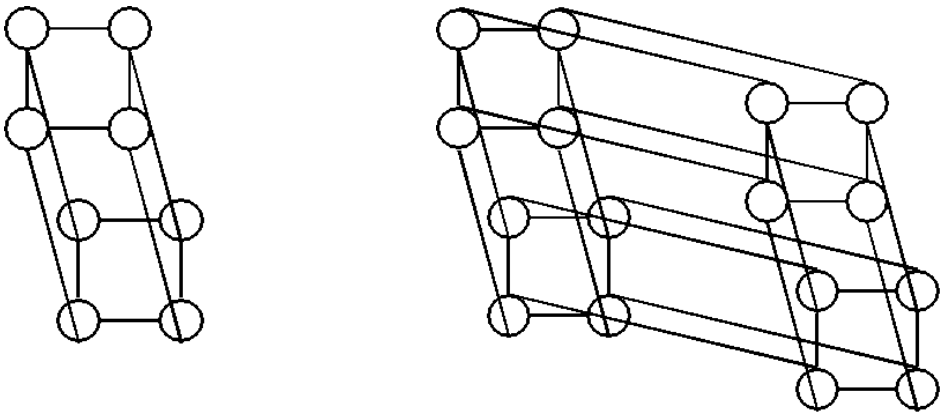
3. Να υπολογισθούν τα ακόλουθα:
   a. $O(0.5n^2 + 7)$ και
   b. $O(\frac{3n + 5\log(n) + 1}{2n}$

4. Να αναπτυχθεί ο αλγόριθμος της τριαδικής αναζήτησης και να συγκριθεί με τον αντίστοιχο της δυαδικής.

5. Με βάση τον αλγόριθμο υπολογισμού δισδιάστατων μεγίστων, να αναπτυχθεί αντίστοιχος για τον υπολογισμό μεγίστων στον τρισδιάστατο χώρο. Πριν την ανάπτυξη του αλγόριθμου να τυποποιηθεί μαθηματικά το πρόβλημα.

6. Οι πίνακες πολλών διαστάσεων (>2) αποτελούν μία χρήσιμη δομή (*Σχήμα 2.4*). Για παράδειγμα εάν πρέπει να κρατηθούν οι μέσες θερμοκρασίες όλων των

ημερών του έτους για μία πόλη τότε θα αρκούσε ένα μονοδιάστατος πίνακας. Εάν θα έπρεπε να κρατηθούν και οι αντίστοιχες θερμοκρασίες για όλες τις πόλεις μίας χώρας, τότε ένας δισδιάστατος πίνακας θα ήταν απαραίτητος όπου οι γραμμές του θα αντιπροσώπευαν τις πόλεις και οι στήλες του τις 365 ημέρες του έτους. Εάν επιπρόσθετα θα έπρεπε να κρατούνται και τα ίδια στοιχεία για πολλά έτη, η τρίτη διάσταση θα μπορούσε να χρησιμεύσει για την αντιπροσώπευση των ετών. Εάν λοιπόν υπάρχει δεδομένος τέτοιος πίνακας, να γραφεί αλγόριθμος ο οποίος να κάνει αναζήτηση μίας θερμοκρασίας εάν δίνεται η πόλη (γραμμή), το έτος (ύψος), ο μήνας και η ημέρα (στήλη).



Σχήμα 2.4 α) Τρεις διαστάσεις β) Τέσσερις διαστάσεις

7. Να δοθούν άλλα παραδείγματα χρήσης πολυδιάστατων πινάκων.

8. Να υπολογισθεί και αποδειχθεί η πολυπλοκότητα της αναζήτησης συμβολοσειράς με την χρήση της brute-force τεχνικής.

3. ΒΕΛΤΙΩΜΕΝΗ ΜΕΘΟΔΟΣ ΥΠΟΛΟΓΙΣΜΟΥ ΔΙΣΔΙΑΣΤΑΤΩΝ ΜΕΓΙΣΤΩΝ (Plane-Sweep Algorithm)

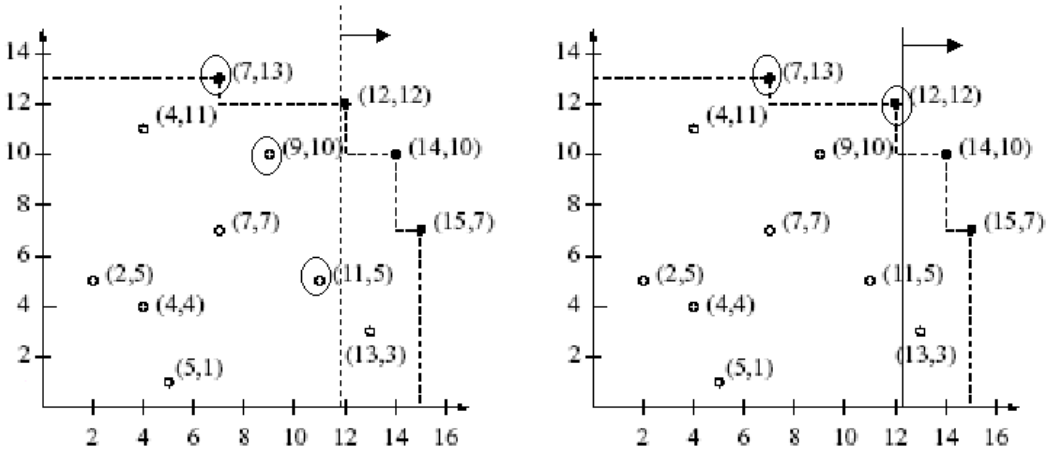
Όπως έχει ήδη αποδειχθεί, η προηγούμενη μέθοδος (brute-force) υπολογισμού δισδιάστατων μεγίστων έχει πολυπλοκότητα $O(n^2)$. Το ερώτημα που τίθεται είναι εάν υπάρχει μία πιο γρήγορη μέθοδος. Το πρόβλημα με τον brute-force αλγόριθμο είναι ότι εξετάζονται όλα τα ζεύγη σημείων όπου ακόμη και στην περίπτωση που ένα σημείο ήδη έχει βρεθεί ότι δεν είναι μέγιστο θα εξεταστεί ξανά και ξανά μέχρι να τελειώσουν όλοι οι πιθανοί συνδυασμοί. Επομένως, είναι ένας μη ευφυής αλγόριθμος που στηρίζεται στη πλήρη και πολλαπλή διερεύνηση του χώρου λύσεων. Εάν ένα σημείο p_x καλύπτεται από ένα άλλο έστω p_y τότε το p_x δεν θα έπρεπε να επανεξεταστεί με κανένα άλλο σημείο. Η παρατήρηση αυτή μπορεί να οδηγήσει σε μια καλύτερη επίλυση του προβλήματος. Εάν βέβαια όλα τα υπό εξέταση σημεία τύχει να είναι μέγιστα τότε αυτή η παρατήρηση δεν οδηγεί σε πιο γρήγορη λύση, αλλά αυτή αποτελεί μια ακραία περίπτωση.

3.1 Αλγόριθμος Σάρωσης του Επιπέδου (Plane-Sweep Algorithm)

Ο αλγόριθμος σάρωσης του επιπέδου στηρίζεται στην προαναφερθείσα παρατήρηση ότι δηλαδή εάν κατά την σύγκριση δύο σημείων κάποιο υπερκαλύπτει το άλλο τότε αυτό το άλλο σημείο δεν πρέπει να επανεξετασθεί. Η τεχνική σάρωσης του επιπέδου λειτουργεί ως εξής: Με μία κάθετη γραμμή σαρώνεται το επίπεδο από αριστερά προς τα δεξιά, κρατώντας τα μέγιστα σημεία που βρίσκονται αριστερά της γραμμής. Όταν η γραμμή φθάσει στο τελευταίο (δεξιότερο) σημείο του επιπέδου τότε θα έχουν βρεθεί όλα τα μέγιστα.

Αν και φαίνεται ότι η γραμμή κινείται κατά ένα συνεχή τρόπο στην πραγματικότητα πρέπει να κινείται με διακριτά βήματα. Για να επιτευχθεί αυτό αρχικά ταξινομούνται τα σημεία σε αύξουσα διάταξη σε σχέση με τις τετμημένες τους. Στη συνέχεια εφαρμόζεται η σάρωση του επιπέδου μετακινώντας την γραμμή από σημείο σε σημείο. Φυσικά η ταξινόμηση επιφέρει ένα επιπλέον κόστος στο όλο πρόβλημα το οποίο όμως είναι της τάξης $O(n \log n)$ εάν χρησιμοποιηθεί κάποια καλή μέθοδος ταξινόμησης όπως η quick sort. Επομένως το πρόβλημα ανάγεται μόνο στο πως θα κρατούνται τα μέγιστα και πως θα ενημερώνονται όταν ένα νέο μέγιστο ανακαλύπτεται. Κατ' αρχήν υποτίθεται ότι κάθε νέο στοιχείο που εξετάζεται είναι εν δυνάμει μέγιστο μιας και η τετμημένη του είναι μεγαλύτερη από όλα τα άλλα (έχει ήδη ταξινομηθεί το σύνολο των σημείων). Το μοναδικό πρόβλημα είναι μήπως αυτό το σημείο που εξετάζεται είναι μεγαλύτερο από κάποιο από τα μέγιστα που έχουν βρεθεί μέχρι τώρα και το οποίο θα πρέπει να αφαιρεθεί από το σύνολο των μεγίστων. Φυσικά κάθε σημείο που βρίσκεται μικρότερο από κάποιο άλλο δεν πρέπει να επανεξετασθεί.

Όπως φαίνεται στο *Σχήμα 3.1*, ενώ πριν εξετασθεί το σημείο (12,12) τα μέγιστα σημεία είναι τα (7,13), (9,10, και (11,5) μετά την εξέταση του (12,12) μέγιστα είναι τα (7,13) και (12,12).



Σχήμα 3.1 Βελτιωμένος Αλγόριθμος Υπολογισμού Δισδιάστατων Μεγίστων

Ανάλυση: Εάν υποτεθεί ότι αυτή τη στιγμή εξετάζεται το σημείο p. Σημειωτέον ότι αφού το p.x είναι μεγαλύτερο από όλα όσα έχουν εξετασθεί μέχρι τώρα τότε το p για να αποτελέσει μέγιστο αρκεί να εξετασθεί μόνο αν η τεταγμένη του είναι μεγαλύτερη ή ίση από όλα τα μέχρι τώρα μέγιστα. Επομένως από όλα τα μέχρι τώρα μέγιστα αρκεί να βρεθούν αυτά με μικρότερη ή ίση τεταγμένη και στη συνέχεια να αφαιρεθούν από το σύνολο των μεγίστων.

Εδώ είναι χρήσιμη η ακόλουθη παρατήρηση. Όπως διαβάζονται τα μέγιστα σημεία από αριστερά προς τα δεξιά και οι τετμημένες είναι σε αύξουσα διάταξη οι τεταγμένες τους μικραίνουν (φθίνουσα διάταξη). Αυτό συμβαίνει γιατί εάν υποτεθεί ότι υπάρχουν δύο μέγιστα έστω p και q, με $p.x \geq q.x$ αλλά $p.y \geq q.y$ τότε θα έπρεπε το p να μην είναι μέγιστο.

Για να γίνει λοιπόν ο υπολογισμός των μεγίστων αρκεί να σαρωθεί το επίπεδο γραμμικά. Η επόμενη και τελευταία ερώτηση είναι η φορά της σάρωσης, δηλαδή από αριστερά προς τα δεξιά ή ανάποδα. Η απάντηση είναι από αριστερά προς τα δεξιά διότι κάθε σημείο που εξετάζεται και η τεταγμένη του είναι μικρότερη από κάποιο άλλο τότε πρέπει να απαλειφθεί από το σύνολο των μεγίστων και δεν υπάρχει λόγος επανεξέτασης. Στην αντίθετη περίπτωση (από δεξιά προς τα αριστερά) και στην χειρότερη δυνατή περίπτωση όπου όλα τα σημεία είναι μέγιστα θα πρέπει το κάθε σημείο του συνόλου των μεγίστων να επανεξετάζεται κάθε φορά και αυτό οδηγεί σε πολυπλοκότητα της τάξης $O(n^2)$.

Παρακάτω περιγράφεται αναλυτικά ο αλγόριθμος. Επειδή τα μέγιστα εισάγονται στο τέλος μίας λίστας και διαγράφονται πάλι από το τέλος της μπορεί να χρησιμοποιηθεί η δομή «στοίβα ή σωρός» και έστω S το όνομά της. Το τελευταίο σημείο του σωρού είναι το S_{top}. Η διαδικασία εισόδου στο σωρό λέγεται Push και εξόδου (κατά συνέπεια και διαγραφής) Pop.

Αλγόριθμος 11: Βελτιωμένος Αλγόριθμος Επίλυσης του Προβλήματος Εύρεσης Μέγιστων Δισδιάστατου Χώρου (Sweep-Plane)

```
1: Δεδομένα:  Πλήθος σημείων n και πίνακας σημείων Π(n) και πίνακας μεγίστων S(n).
2: Αρχή
3:      Ταξινόμηση του πίνακα Π με κλειδί την τετμημένη των σημείων.
4:      S = κενό
5:      Για i από 1 μέχρι n
6:           Εφόσον (S όχι κενή ΚΑΙ  S.top.y ≤ Π(i).y)
7:                Pop(S)
8:           Τέλος Εφόσον
9:      Push(S, Π(i))
10:     Τέλος  Για (i)
11: Τέλος Αλγόριθμου
```

3.2 Πολυπλοκότητα του Αλγόριθμου Σάρωσης του Επιπέδου

Κατ' αρχήν η ταξινόμηση του πίνακα των σημείων είναι της τάξης $O(n\log n)$. Ο αλγόριθμος αυτός, περιλαμβάνει επίσης δύο βρόγχους. Ο εξωτερικός βρόγχος επαναλαμβάνεται n φορές. Το ερώτημα που τίθεται είναι πόσες φορές επαναλαμβάνεται ο εσωτερικός. Αρχικά ενώ δείχνει ότι επαναλαμβάνεται και αυτός το πολύ n-1 φορές για κάθε επανάληψη του εξωτερικού βρόγχου που σημαίνει πολυπλοκότητα της τάξης n^2, στην πραγματικότητα επαναλαμβάνεται το πολύ n-1 φορές συνολικά. Αυτό ισχύει γιατί δεν είναι δυνατό να εξαχθούν πιο πολλά σημεία από την στοίβα από όσα θα εισαχθούν και θα εισαχθούν n ακριβώς σημεία. Και επειδή τουλάχιστον ένα μέγιστο θα υπάρχει κατά συνέπεια θα επαναληφθεί n-1 φορές συνολικά. Επομένως η συνολική πολυπλοκότητα του αλγόριθμου είναι της τάξης $n + n\log n$, δηλαδή $O(n\log n)$.

Συγκριτικά με τον προηγούμενο αλγόριθμο (brute-force) του οποίου η πολυπλοκότητα είναι της τάξης $O(n^2)$ ο sweep-plane είναι σαφώς πιο γρήγορος. Ο λόγος των πολυπλοκοτήτων τους είναι:

$$\frac{n^2}{n\log n} = \frac{n}{\log n}$$

όπου ενώ για μικρό n είναι σχεδόν ισοδύναμοι, όσο όμως αυξάνεται το πλήθος των σημείων η διαφορά τους είναι εμφανής. Για παράδειγμα, εάν n=1.000.000, ο λόγος είναι $\frac{1.000.000}{20} = 50.000$. Αυτό σημαίνει σε πραγματικούς χρόνους ότι εάν ο αλγόριθμος σάρωσης του επιπέδου χρειάζεται 1 δευτερόλεπτο, ο πρώτος αλγόριθμος (brute-force) θα χρειαζόταν 13,88 ώρες.

3.3 Ασκήσεις

1. Να γραφεί πρόγραμμα υλοποίησης του αλγόριθμου σάρωσης του επιπέδου. Η στοίβα να υλοποιείται με πίνακα και τα σημεία να εισάγονται από αρχείο δεδομένων.

ΛΥΣΗ

Πρόγραμμα 7: Πρόγραμμα υλοποίησης αλγόριθμου σάρωσης επιπέδου

```
#include <stdio.h>
#define M 1000 /* Μέγιστος αριθμός σημείων */

struct Simio {
        int x;
        int y;
};

struct Simio P[M], S[M], temp;

main()
{
        FILE f; / Αρχείο αποθήκευσης σημείων */
        int i=0,j=0,k=0,l=0,N;
        int megisto;

        /* Εισαγωγή σημείων */

        f=fopen("2dmax.dat","r");
        while (! feof(f) ) {
                fscanf(f,"%d%d",&P[i].x,&P[i].y);
                if ( ! feof(f)) {
                        printf("\n%2d==>%5d,%5d",i,P[i].x,P[i].y);
                        i++;
                }
        }

        fclose(f);
        printf("\n\n");
        system("pause");

        N = i;

        /* Ταξινόμηση πίνακα σημείων – Bubble sort */
        for (i=0;i<N;i++)
                for (j=N-1;j>0;j--)
                        if (P[j].x<P[j-1].x) {
                                /* Αντιμετάθεση */
                                temp = P[j];
                                P[j] = P[j-1];
```

```
                        P[j-1] = temp;
                }

        /* Ταξινομημένος ως προς την τετμημένη x */
        printf("\n\n Ταξινομημένα σημεία \n");
        for (i=0; i<N; i++)
                printf("\n%3d-->%5d,%5d",i,P[i].x,P[i].y);
        printf("\n\n");
        system("pause");

        /* Αρχικοποίηση S */
        for (i=0; i<N; i++) {
                S[i].x=0;
                S[i].y=0;
        }

        /* Υπολογισμός 2-d μεγίστων (plane sweep algorithm) */
        j=0;
        for (i=0; i<N; i++) {
                while (j>=0 && S[j-1].y<=P[i].y)
                        j--;  /* Απομάκρυνση σημείου */

                S[j++]=P[i]; /* εισαγωγή σημείου */
        }

        /* Εκτυπώσεις */
        printf("\n\n ΜΕΓΙΣΤΑ \n");
        for (i=0; i<j; i++)
                printf("\n%3d-->%5d,%5d",i,S[i].x,S[i].y);

        printf("\n\n Σύνολο μεγίστων:%5d\n\n",j);
}
```

2. Να ξαναγραφεί το παραπάνω πρόγραμμα με χρήση της δομής της στοίβας αντί πίνακα.

3. Να αναπτυχθεί αλγόριθμος βελτίωσης του αλγόριθμου υπολογισμού των ελαχίστων αποστάσεων (closest-pair problem)

4. ΑΝΑΔΡΟΜΙΚΟΙ ΑΛΓΟΡΙΘΜΟΙ

Αναδρομικός λέγεται ο αλγόριθμος ο οποίος επιλύει ένα πρόβλημα επιλύνοντας ένα ή περισσότερα μικρότερα κομμάτια του ίδιου προβλήματος. Για να υλοποιήσουν αναδρομικούς αλγόριθμους, πολλές γλώσσες προγραμματισμού χρησιμοποιούν αναδρομικές συναρτήσεις (συναρτήσεις οι οποίες καλούν τους εαυτούς τους). Οι αναδρομικές συναρτήσεις στη C για παράδειγμα, μοιάζουν με τον μαθηματικό ορισμό των συναρτήσεων. Ίσως ένας από τους πιο τυπικούς αναδρομικούς αλγόριθμους είναι αυτός που υπολογίζει το $N!$ χρησιμοποιώντας τον αναδρομικό τύπο $N! = N \cdot (N-1)!$. Η αντίστοιχη συνάρτηση σε C είναι η ακόλουθη:

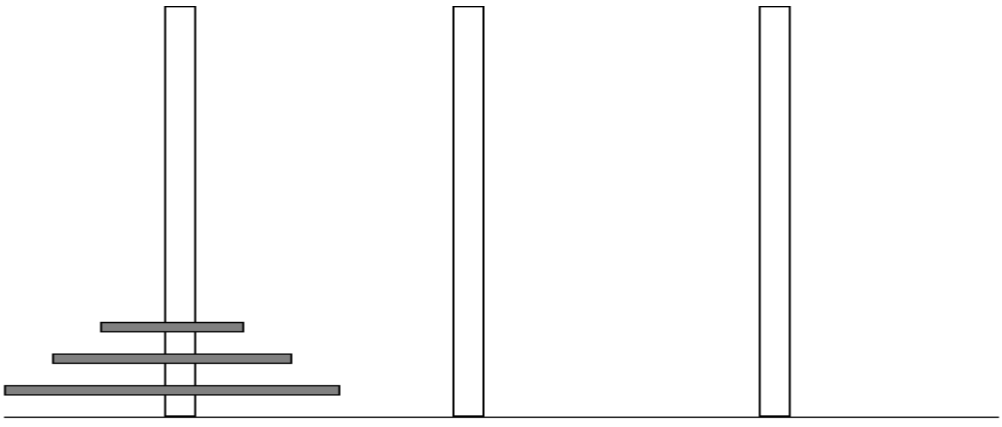
Η παρακάτω αναδρομική συνάρτηση υπολογίζει την συνάρτηση N! με τον πρότυπο ορισμό της αναδρομικότητας. Επιστρέφει σωστό αποτέλεσμα για θετικό N και σχετικά μικρό ώστε να μπορεί να αναπαρασταθεί σαν int.

```
int paragontiko(int N)
{
        if (N==0) return 1;
        return N*paragontiko(N-1);
}
```

Η χρήση της αναδρομής επιτρέπει την περιγραφή δύσκολων αλγορίθμων με ένα συμπαγή τρόπο, χωρίς να χάνεται η αποτελεσματικότητα του αλγόριθμου (αν και μερικές φορές οι αναδρομικοί αλγόριθμοι παρουσιάζουν χειρότερη πολυπλοκότητα από τους μη αναδρομικούς).

4.2 Οι Πύργοι του Hanoi

Ένα από τα πιο κλασικά παραδείγματα χρήσης αναδρομικών συναρτήσεων είναι οι «***Πύργοι του Hanoi***». Η περιγραφή του προβλήματος είναι η ακόλουθη: Υπάρχουν 3 κολόνες (*Σχήμα 4.1*) και σε μία από αυτές είναι περασμένοι N ομόκεντροι δίσκοι διαφορετικών διαμέτρων μειούμενοι από κάτω προς τα πάνω.



Σχήμα 4.1 Πύργοι του Hanoi

Το πρόβλημα έγκειται στην μεταφορά αυτών των δίσκων στη γειτονική κολόνα αλλά με τους παρακάτω περιορισμούς:

1) Επιτρέπεται η μεταφορά ενός δίσκου κάθε φορά, και
2) Απαγορεύετε να βρεθούν δύο διαδοχικού δίσκοι στη ίδια κολόνα με τον μικρότερο κάτω από τον μεγαλύτερο.

Ένα θρύλος λέει πως εάν ένα σύνολο μοναχών καταφέρει να μεταφέρει 40 χρυσούς δίσκους σε ένα ναό χρησιμοποιώντας 3 αδαμάντινες κολόνες και στην διάρκεια της ζωής τους τότε ο κόσμος θα «τελειώσει».

Η ιδέα επίλυσης αυτού του προβλήματος είναι η εξής: Για να μετακινηθούν οι N δίσκοι στη δεξιά κολόνα (έστω 1 ο δίσκος της κορυφής, 2 ο επόμενος οπότε N ο τελευταίος και πιο μεγάλος δίσκος) αρκεί να μετακινηθούν οι N-1 πιο πάνω δίσκοι στην αριστερή κολόνα, στη συνέχεια ο δίσκος N να μετακινηθεί στην δεξιά κολόνα και τέλος η στοίβα των N-1 δίσκων να μετακινηθεί πάνω από τον δίσκο N.

Η πολυπλοκότητα του αλγόριθμου υπολογίζεται σχετικά εύκολα λόγω της αναδρομικής δομής του:

Πρόταση: *Ο αλγόριθμος επίλυσης του προβλήματος «Πύργοι του Hanoi» παράγει λύση η οποία δημιουργεί* $2^N - 1$ *κινήσεις.*

Απόδειξη: Εφόσον πρέπει να μετακινηθεί η στοίβα των N-1 δίσκων μία θέση δεξιά και άλλη μία φορά επάνω από τον δίσκο N, τότε χρειάζεται δύο φορές το πλήθος των κινήσεων εάν είχαμε μόνο μια στοίβα N-1 δίσκων και επιπλέον μία κίνηση για τον δίσκο N. Οπότε εάν απαιτούνται T_K κινήσεις για μια στοίβα K δίσκων τότε για να μετακινηθούν N δίσκοι χρειάζονται:

$T_N = 2T_{N-1} + 1$ κινήσεις για $N \geq 2$ και $T_1 = 1$

Οπότε με επαγωγή έχουμε:

$T_1 = 2^1 - 1 = 2 - 1 = 1$,

έστω ότι ισχύει για $N = k$, δηλαδή $T_k = 2^k - 1$, για $k < N$ τότε

$T_N = 2(2^{N-1} - 1) + 1 = 2 \cdot 2^{N-1} - 2 + 1 = 2^N - 1$. □

Οπότε και η πολυπλοκότητα της μεθόδου είναι $O(2^N)$.

Για να γίνει πιο αντιληπτή η πολυπλοκότητα του παραπάνω αλγόριθμου ας δούμε το εξής: Εάν οι μοναχοί χρειαζόταν ένα δευτερόλεπτο για να μετακινήσουν ένα δίσκο από μια κολόνα σε μια άλλη θα τους έπαιρνε 348 αιώνες για να τελειώσουν (με βάση τον πίνακα που ακολουθεί)

($T_{40} = 2^{40} - 1 = 1.099.511.627.776 - 1 = 1.099.511.627.775$) με την προϋπόθεση ότι δεν έκαναν κανένα λάθος.

Δευτερόλεπτα	
10^2	1,7 λεπτά
10^4	2,8 ώρες
10^5	1,1 ημέρες
10^6	1,6 εβδομάδες
10^7	3,8 μήνες
10^8	3,1 χρόνια
10^9	3,1 δεκαετίες
10^{10}	3,1 αιώνες
10^{11}	ΠΟΤΕ
Μετατροπή δευτερολέπτων	
Η διαφορά χρόνου φαίνεται εάν μετατρέψουμε τις μονάδες σε ποιο κατανοητές χρονικές μονάδες.	

Στο παρακάτω πρόγραμμα φαίνεται καθαρά ο αναδρομικός αλγόριθμος επίλυσης.

Πρόγραμμα 8: Πρόγραμμα υλοποίησης προβλήματος «Πύργοι του Hanoi»

```
/* Towers of Hanoi */
#include <stdio.h>

void hanoi(int, int);
void metakinise(int, int);

char thesi[40]; /* Η τρέχουσα θέση του κάθε δίσκου*/
long met=0;     /* Πλήθος Κινήσεων */
/* ------------------------------------------------------ */

main()
{
  int diskoi;
  int i;

  printf("\n\n Πλήθος δίσκων:");
  scanf("%d",&diskoi);

  for (i=1;i<=diskoi;i++)
     thesi[i]='A';

  hanoi(diskoi,1);
```

```
}

/* ------------------------------------------------------ */
void hanoi(int N, int k)
{
  if (N==0) return;
  hanoi(N-1,-k);
  metakinise(N,k);
  hanoi(N-1,-k);
}
/* ------------------------------------------------------ */
void metakinise(int N, int k)
{
  met++;
  if (k>0) thesi[N]++;
     else thesi[N]--;
  if (thesi[N]>'C') thesi[N]='A';
  if (thesi[N]<'A') thesi[N]='C';
  printf("\n%3d: Δίσκος %5d στη θέση %5c",met,N,thesi[N]);
}
```

Ο αλγόριθμος προσδιορίζει ποιος δίσκος θα μεταφερθεί και προς ποια κατεύθυνση (δηλαδή σε ποια κολόνα) σε κάθε βήμα. Εάν το k είναι θετικό θα μεταφερθεί δεξιά και εάν δεν υπάρχει πιο δεξιά κολόνα τότε στην πρώτη από αριστερά, αλλιώς εάν το k είναι αρνητικό, θα μεταφερθεί αριστερά και πάλι εάν δεν υπάρχει πιο αριστερά στην δεξιότερη κολόνα (στο πρόγραμμα οι κολόνες ονομάζονται A, B, C).

Η εκτέλεση του προγράμματος για N=3 παράγει τις παρακάτω κινήσεις:

```
hanoi(3,1)
        hanoi(2,-1)
                hanoi(1,1)
                        hanoi(0,-1)
                        metakinise(1, +1 άρα δεξιά)
                        hanoi(0,-1)
                metakinise(2,-1 άρα αριστερά)
                hanoi(1,1)
                        hanoi(0,-1)
                        metakinise(1,+1)
                        hanoi(0,-1)
                metakinise(3,+1)
                hanoi(2,-1)
                        hanoi(1,1)
                        hanoi(0,-1)
                metakinise(1,+1)
                hanoi(0,-1)
        metakinise(2,-1)
        hanoi(1,1)
                hanoi(0,-1)
                metakinise(1,+1)
```

```
hanoi(0,-1)
```

Τέλος, ένας μη αναδρομικός αλγόριθμος επίλυσης του προβλήματος των πύργων του Hanoi είναι ο ακόλουθος:

Αλγόριθμος 12: Μη αναδρομικός αλγόριθμος επίλυσης του προβλήματος «Πύργοι του Hanoi»

```
1:    Εφόσον δεν τελείωσε
2:        Μετακίνησε τον μικρότερο δίσκο μια θέση δεξιά, εάν το πλήθος των
          δίσκων n είναι περιττός αριθμός (αριστερά, εάν είναι άρτιος).
3:        Μετακίνησε τον μόνο δίσκο που μπορεί να μετακινηθεί (πλην του
          μικρότερου).
4:    Τέλος Εφόσον
5: Τέλος αλγόριθμου
```

4.2 Ασκήσεις

1. Εξηγήστε τον υπολογισμό του Μέγιστου Κοινού Διαιρέτη με τον αναδρομικό αλγόριθμο του Ευκλείδη. Αναλύστε το βάθος της αναδρομικότητας, να υπολογισθεί η πολυπλοκότητά του και τέλος να υλοποιηθεί σε κάποια γλώσσα προγραμματισμού.

2. Ο υπολογισμός του μέγιστου (ή ελάχιστου) στοιχείου ενός πίνακα με μη αναδρομικό αλγόριθμο είναι ο ακόλουθος:
   a. Είσοδος: Πίνακα Α με Ν στοιχεία
   b. Έστω t ← A[0]
   c. Για i από 1 μέχρι N-1
   d. Εάν A[i]>t Τότε t ← A[i]
   e. Τέλος_Για
   f. Επέστρεψε t

   Ο αναδρομικός αλγόριθμος επίλυσης του παραπάνω προβλήματος στηρίζεται στην αρχή του διαίρει και βασίλευε και η γενική του προσέγγιση είναι να διαιρεί το πίνακα σε δύο υποπίνακες και αφού υπολογίζει τα μέγιστα στοιχεία τους να επιστρέφει σαν μέγιστο όλων τον μεγαλύτερο αυτών των δύο. Να κατασκευάσετε και να υλοποιήσετε αυτόν τον αλγόριθμο.

3. Να αποδειχθεί ότι ο προηγούμενος αλγόριθμος (της άσκησης 2) καλεί τον εαυτό του λιγότερο από Ν φορές.

4. Γράψτε έναν αναδρομικό αλγόριθμο που υπολογίζει το μέγιστο στοιχείο πίνακα βασιζόμενο στο ότι το μέγιστο στοιχείο είναι το μεγαλύτερο που προκύπτει από την σύγκριση του πρώτου στοιχείου με το μεγαλύτερο όλων των υπολοίπων στοιχείων του πίνακα.

5. «ΔΙΑΙΡΕΙ ΚΑΙ ΒΑΣΙΛΕΥΕ»

5.1 Εισαγωγή

Μία από τις κλασικές μεθόδους επίλυσης προβλημάτων είναι η λεγόμενη «Διαίρει και Βασίλευε» (Divide and Conquer). Η μέθοδος οφείλει την ονομασία της στους αρχαίους Ρωμαίους πολιτικούς (οι οποίοι προφανώς δεν σκεφτόταν την ανάπτυξη αλγορίθμων). Η τεχνική ήταν απλή: διαίρεσε (divide) του εχθρούς σου βάζοντάς τους να φιλονικούν μεταξύ τους και στη συνέχεια κατέκτησέ τους έναν έναν (conquer). Στην ανάπτυξη αλγορίθμων, η μέθοδος έγκειται στην διάσπαση ενός προβλήματος σε μικρά κομμάτια του ίδιου προβλήματος και στην συνέχεια στην συνένωση των επιμέρους λύσεων στη γενική λύση του προβλήματος. Βέβαια σε κάθε ένα από τα επιμέρους προβλήματα πρέπει αναδρομικά να εφαρμοσθεί ή ίδια μέθοδος μέχρι να καταλήξει σε προβλήματα τάξης μεγέθους ένα ή το πολύ δύο όπου η λύση είναι εύκολη. Συνοψίζοντας, τα βήματα επίλυσης ενός προβλήματος με αυτή την μέθοδο είναι τα ακόλουθα:

1. Διαίρει (το πρόβλημα σε μικρότερα επιμέρους προβλήματα),
2. Βασίλευε (λύνοντας το κάθε επιμέρους πρόβλημα εφαρμόζοντας αναδρομικά την ίδια τακτική), και τέλος
3. Συνένωσε (τις επιμέρους λύσεις σε μία καθολική λύση του προβλήματος).

5.2 Η Μέθοδος Ταξινόμησης Merge Sort

Αυτή η μέθοδος ταξινόμησης αποτελεί ένα πολύ καλό παράδειγμα εφαρμογής της τεχνικής «διαίρει και βασίλευε». Είναι ένας καλός και πολύ αποτελεσματικός αλγόριθμος ταξινόμησης. Η εφαρμογή της μεθόδου συνοψίζεται ως εξής:

1. Διαίρει: Διαχωρισμός των στοιχείων στην μέση ώστε να προκύψουν δύο πίνακες με τα μισά περίπου στοιχεία ο καθένας,
2. Βασίλευε: Ταξινόμηση κάθε υποπίνακα στοιχείων (καλώντας την ίδια μέθοδο αναδρομικά), και
3. Συνένωσε: Ένωση των ταξινομημένων υποπινάκων σε ένα νέο ταξινομημένο.

Αλγόριθμος 13: Merge Sort

```
1: Διαδικασία MergeSort(πίνακας A, ακέραιος p, ακέραιος r)
2:Αρχή
3:      Εάν (p < r) Τότε
4:              q = (p+r) / 2
5 :             MergeSort(A, p, q)
6:              MergeSort(A, q+1, r)
7:              Merge(A, p, q, r)
8:      Τέλος Εάν
9: Τέλος Διαδικασίας «MergeSort»
```

Επομένως για να ταξινομηθεί ολόκληρος ο πίνακας, ο αλγόριθμος πρέπει να κληθεί: MergeSort(A, 1, N) εάν N είναι το πλήθος των στοιχείων του.

Βέβαια, για να ολοκληρωθεί η διαδικασία της ταξινόμησης, πρέπει να λυθεί και το πρόβλημα της συνένωσης (merge) των δύο ταξινομημένων υποπινάκων. Ο αλγόριθμος επίλυσης αυτού του προβλήματος δίνεται στον *Αλγόριθμο 14* καθώς επίσης και η υλοποίησή του στο *Πρόγραμμα 9*.

Αλγόριθμος 14: Ενοποίηση ταξινομημένων πινάκων

```
1: Διαδικασία Merge(πίνακας A, ακέραιος p, ακέραιος q, ακέραιος r)
2: Αρχή
3:      Πίνακας ακεραίων B_{r-p}
4:      Ακέραιοι: i, j, k
5:      i ← p
6:      k ← p
7:      j ← q+1
8:      Εφόσον (i ≤ q  ΚΑΙ j ≤  r)
9:           Εάν (A_i ≤ A_j) Τότε
10:               B_k ← A_i
11:               k ← k+1
12:               i ← i+1
13:          Αλλιώς
14:               B_k ← A_j
15:               k ← k+1
16:               j ← j+1
17:          Τέλος Εάν
18:     Τέλος Εφόσον
19:     Εφόσον (i ≤ q )
20:          B_k ← A_i
21:          k ← k+1
22:          i ← i+1
23:     Τέλος Εφόσον
24:     Εφόσον (j ≤  r)
25:          B_k ← A_j
26:          k ← k+1
27:          j ← j+1
28:     Τέλος Εφόσον
29:     Για i=1 μέχρι r
30:          A_i ← B_i
31:     Τέλος Για (i)
32: Τέλος Διαδικασίας «Merge»
```

Πρόγραμμα 9: Ενοποίηση ταξινομημένων πινάκων

```
void merge(A[], int p, int q, int r) {
        int B[r-p];
        int i=k=p;
        int j=q+1;
```

```
        while (i <= q && j <=  r)
                if (A[i] <= A[j])
                        B[k++] = A[i++];
                else
                        B[k++] = A[j++];

        while (i <= q) B[k++] = A[i++];
        while (j <= r) B[k++] = A[j++];

        for (i=0; i < r; i++)
                A[i] = B[i];
}
```

Το μειονέκτημα της μεθόδου είναι ότι χρησιμοποιεί ένα βοηθητικό πίνακα (B) και τον οποίο στην συνέχεια τον μεταφέρει στον αρχικό αλλά κατά τα άλλα είναι μία γρήγορη μέθοδος ταξινόμησης που η τάξη πολυπλοκότητάς της είναι $O(n \log n)$.

5.3 Πολυπλοκότητα της Merge Sort

Εάν υποτεθεί ότι ο χρόνος ταξινόμησης ενός πίνακα N στοιχείων είναι T(N) τότε επειδή ο πίνακας πρέπει να διαιρεθεί σε δύο υποπίνακες και επιπλέον να γίνει και η συγχώνευση των N στοιχείων, είναι προφανές ότι:

$$T(N) = \begin{cases} 1, N = 1 \\ T\left\lfloor (N/2) \right\rfloor + T\left\lceil (N/2) \right\rceil + N, \forall N > 1 \end{cases}$$

Πιο αναλυτικά για να δοθεί μία εικόνα της πολυπλοκότητας είναι:

T(1)	=	1		
T(2)	=	2T(1) + 2	=	4
T(3)	=	T(1) + T(2) + 3	=	8
T(4)	=	T(2) + T(2) + 4	=	12
T(5)	=	T(2) + T(3) + 5	=	17

……

Αν και είναι αρκετά δύσκολο να βρεθεί ένας γενικός τύπος από τις παραπάνω τιμές δεν είναι ακατόρθωτο και αυτός είναι $N \lg N + N$. Η απόδειξη πάλι με την μέθοδο της επαγωγής (όπως σε σχεδόν όλες τις περιπτώσεις αναδρομικών αλγόριθμων).

Απόδειξη

Για N=1, ισχύει γιατί T(1) = 1 = 1 lg1.

Έστω ότι ισχύει για κάθε αριθμό μικρότερο του N. Τότε για N έχουμε:

$$T(N) = T\lfloor (N/2) \rfloor + T\lceil (N/2) \rceil + N$$

Επομένως

$$T(N) = 2((N/2)\lg(N/2) + (N/2)) + N$$
$$= (N\lg(N/2) + N) + N$$
$$= N(\lg N - \lg 2) + 2N$$
$$= N(\lg N - 1) + 2N$$
$$= N\lg N - N + 2N$$
$$= N\lg N + N$$

και επομένως αποδείχθηκε.

Δηλαδή, η πολυπλοκότητα της merge sort που είναι της τάξης $(O(N\lg N)$ δείχνει ότι πρόκειται για μία γρήγορη μέθοδο ταξινόμησης.

5.4 Η Μέθοδος Ταξινόμησης Quick Sort

Η ιδέα της γρήγορης ταξινόμησης (quick sort) βασίζεται στο ότι εάν βρεθεί ένα κατάλληλο στοιχείο *οδηγό* και αναδιαταχθεί ο αρχικός πίνακας σε δύο υποπίνακες όπου όλα τα στοιχεία του πρώτου μπορεί μεν να μην είναι ταξινομημένα αλλά είναι μικρότερα του οδηγού στοιχείου και όλα τα στοιχεία του δεύτερου να είναι μεγαλύτερα του οδηγού τότε δεν αρκεί να επαναληφθεί αυτή τη διαδικασία για τους δύο μικρότερους πίνακες ώσπου να καταλήξει σε πίνακες στοιχεία και τότε η διαδικασία θα έχει ολοκληρωθεί (μέθοδος διαίρει και βασίλευε). Για παράδειγμα στον παρακάτω πίνακα

26, 18, 4, 9, 37, 119, 220, 47, 74

ο αριθμός 37 είναι το ζητούμενο στοιχείο.

Βέβαια το πρόβλημα τώρα τίθεται στην εύρεση του οδηγού στοιχείου. Αρχικά πάντως η μέθοδος της γρήγορης ταξινόμησης θα μπορούσε να περιγραφεί ως εξής:

```
ΓρήγορηΤαξινόμηση1(x: πίνακας N στοιχείων)

Εάν N >= 2 Τότε
        Εύρεση του οδηγού στοιχείου x[i]
        ΓρήγορηΤαξινόμηση1(του υποπίνακα x_1, x_2, ..., x_{i-1})
        ΓρήγορηΤαξινόμηση1(του υποπίνακα x_{i+1}, x_{i+2}, ..., x_n)
Τέλος Εάν

Τέλος { ΓρήγορηΤαξινόμηση1}
```

Βέβαια αυτός ο αλγόριθμος δεν δουλεύει αλλά δείχνει καθαρά ότι πρόκειται για έναν αναδρομικό αλγόριθμο (recursive). Για να γίνει λίγο πιο γενικός θα μπορούσε να

δοθεί με τρόπο ώστε να ταξινομεί ένα πίνακα με αρχικό δείκτη *left* και τελικό *right*. Δηλαδή:

```
ΓρήγορηΤαξινόμηση2(x: πίνακας, left,right)

Εάν right-left >= 1 τότε
        Εύρεση του οδηγού στοιχείου x[i]
        ΓρήγορηΤαξινόμηση2(x, left, i-1)
        ΓρήγορηΤαξινόμηση2(x, i+1, right)
Τέλος Εάν

Τέλος { ΓρήγορηΤαξινόμηση2}
```

Και φυσικά για την ταξινόμηση ολόκληρου του πίνακα δεν έχουμε παρά να τεθεί

left = 1, right = N.

Το επόμενο ζητούμενο είναι ο εντοπισμός του οδηγού στοιχείου. Η επιλογή του στοιχείου αυτού είναι όμως πάρα πολύ απλή, είναι τυχαία. Απλώς επιλέγεται με χρήση κάποιας συνάρτησης παραγωγής τυχαίων αριθμών, ένα στοιχείο του προς ταξινόμηση πίνακα έστω *T*. Μόλις επιλεγεί ο οδηγός, η επόμενη διαδικασία είναι η αναδιάταξη του πίνακα ώστε αριστερά να υπάρχουν στοιχεία μόνο μικρότερα του οδηγού ενώ δεξιά μόνο μεγαλύτερα. Αυτή η διαδικασία μπορεί να περιγραφεί με τον παρακάτω αλγόριθμο:

```
ΔιαχωρισμόςΠίνακα(x, left, right, i)

{i είναι η τελική θέση του T}

        L= τυχαίος αριθμός στο διάστημα [left, right]
        Αντιμετάθεσε(x[left], x[L])
        {επομένως, τώρα το οδηγό στοιχείο είναι στην πρώτη θέση}
        T=x[left]
        i=left
        Για j=left+1 μέχρι right
                Εάν x[j] < T τότε
                                i=i+1;
                                αντιμετάθεσε(x[i], x[j])
                Τέλος Εάν
        Τέλος Για
        Αντιμετάθεσε(x[left], x[i])

Τέλος {Διαχωρισμός Πίνακα}
```

Οπότε τελικά η μέθοδος της γρήγορης ταξινόμησης μπορεί να περιγραφεί ως εξής:

```
ΓρήγορηΤαξινόμηση3(x:πίνακας, left, right)
```

```
        Εάν right-left >= 1 τότε
                ΔιαχωρισμόςΠίνακα(x, left, right, i)
                ΓρήγορηΤαξινόμηση3(x:πίνακας, left, i-1)
                ΓρήγορηΤαξινόμηση3(x:πίνακας, i+1, right)
        Τέλος Εάν

Τέλος { ΓρήγορηΤαξινόμηση3}
```

5.5 Quick Sort: Πολυπλοκότητα και Υλοποίηση

Όπως αποδεικνύεται (η απόδειξη είναι πέραν του σκοπού του αυτού του μαθήματος) η πολυπλοκότητα της quick sort είναι στην χειρότερη περίπτωση πάλι $O(N^2)$ αλλά η μέση περίπτωση είναι $O(Nlog(N))$ και αυτή θεωρείται και ως βασική. Δηλαδή η γρήγορη ταξινόμηση είναι ασύγκριτα πιο γρήγορη από τις αργές μεθόδους με πολυπλοκότητες της τάξης του $O(N^2)$ και μάλιστα σε τάξη μεγέθους $\frac{N}{\log(N)}$. Για παράδειγμα εάν μία τράπεζα ήθελε να ταξινομήσει τους 5.000.000 καταθέτες της, η αργή ταξινόμηση θα χρειαζόταν 25.000.000.000.000 συγκρίσεις ενώ η γρήγορη μόλις 105.000.000, δηλαδή 238.000 φορές πιο γρήγορη.

Στο *Πρόγραμμα 10* δίνεται μία υλοποίηση της μεθόδου quick sort με κάπως διαφορετικό τρόπο επιλογής του οδηγού στοιχείου.

Πρόγραμμα 10: Quick sort

```
#include <stdio.h>
#define N 20 /* Μέγεθος Πίνακα */

void quicksort(int *, int, int);
int pivot(int, int);
int partition(int *, int, int);

main()
{
        int A[N] = {4,7,9,0,11,6,2,-8,3,6,-7,23,5,12,-8,-6,3,9,-1,10};
        int i;

        /* Εκτύπωση αταξινόμητου πίνακα */
        printf("\n Αταξινόμητος Πίνακας \n");
        for (i=0; i<N; i++)
                printf("%10d",A[i]);
        printf("\n\n");
        system("pause");

        quicksort(&A[0], 0, N-1);

        /* Ταξινομημένος Πίνακας */
        printf("\n\nTaksinomimenos Pinakas\n");
```

```
	for (i=0; i<N; i++)
		printf("%10d",A[i]);
	printf("\n\n");

}

/* ΣΥΝΑΡΤΗΣΕΙΣ */

void quicksort(int P[], int p, int r)
{
	int temp;
	int q,i;

	if (r <= p) return;
	i = pivot(p,r);

	/* αντιμετάθεση P[i] με P[p] */
	temp = P[i];
	P[i] = P[p];
	P[p] = temp;

	q = partition(&P[0], p, r);
	quicksort(&P[0], p, q-1);
	quicksort(&P[0], q+1, r);
}

int pivot(int a, int b)
{
	int x;

	x = rand() % (b-a) + a;
	return x;
}

int partition(int P[], int p, int r)
{
	int x, q, temp, i;

	x = P[p];
	q = p;

	for (i=p; i<=r; i++)
		if (P[i]<x) {
			q++;
			/* Αντιμετάθεση */
			temp = P[q];
			P[q] = P[i];
			P[i] = temp;
		}
```

```
        /* Αντιμετάθεση P[p] με P[q] */
        temp = P[p];
        P[p] = P[q];
        P[q] = temp;
        return q;
}
```

5.6 Ασκήσεις

1. Εφαρμόστε τον διαχωρισμό του πίνακα x = {2,4,1,10,5,3,9,7,8,6} με οδηγό το πέμπτο στοιχείο, δηλαδή *L=5*, (όχι σε Η/Υ αλλά με χαρτί και μολύβι) και παρατηρήστε τις αλλαγές του πίνακα για κάθε μεταβολή του δείκτη *j* της *Για j=...* ανακύκλωσης.

2. Να βρείτε και στη συνέχεια αξιολογήσετε δύο άλλες μεθόδους ταξινόμησης.

3. Με την μέθοδο «διαίρει και βασίλευε» να σχεδιασθεί αλγόριθμος υπολογισμού του μεγαλύτερου ή μικρότερου στοιχείου ενός αταξινόμητου πίνακα και στη συνέχεια να υπολογισθεί η πολυπλοκότητά του.

4. Να σχεδιάσετε έναν αλγόριθμο ο οποίος δοθέντων δύο πολυωνύμων $p(x)$ και $q(x)$ βαθμού n να υπολογίζει το γινόμενό τους.

6. ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

6.1 Εισαγωγή

Αυτή η τεχνική προϋποθέτει μία αναδρομική επίλυση προβλημάτων αλλά με μία bottom-up εκτίμηση των λύσεων. Οι υπό-λύσεις συνήθως αποθηκεύονται για την επαναχρησιμοποίησή τους. Αυτή η τεχνική χρησιμοποιείται κυρίως σε προβλήματα βελτιστοποίησης τα οποία σχετίζονται με πολλές λύσεις αλλά και μια τιμή κόστους για το κάθε ένα από αυτά. Τυπικό παράδειγμα αποτελεί η επίλυση της συνάρτησης Fibbonaci η οποία ορίζεται ως εξής:

$$f(n) = \begin{cases} f(n-1) + f(n-2), \forall n > 2 \\ f(1) = f(2) = 1 \end{cases}$$

Δηλαδή η $f(6)$ αναλύεται ως εξής:

$$f(6)\begin{cases} f(5)\begin{cases} f(4)\begin{cases} f(3)\begin{cases} f(2) \\ f(1) \end{cases} \\ f(2) \end{cases} \\ f(3)\begin{cases} f(2) \\ f(1) \end{cases} \end{cases} \\ f(4)\begin{cases} f(3)\begin{cases} f(2) \\ f(1) \end{cases} \\ f(2) \end{cases} \end{cases}$$

Εδώ μία top-down προσέγγιση της λύσης δεν είναι καθόλου ικανοποιητική γιατί αναγκαστικά θα επαναλάβει πολλούς ίδιους υπολογισμούς. Για παράδειγμα, στον υπολογισμό του $f(6)$ θα έπρεπε να υπολογισθεί τα $f(4)$ δύο φορές και το $f(3)$ τρεις φορές. Σε αντίθεση, με την τεχνική του δυναμικού προγραμματισμού οι υπό-λύσεις υπολογίζονται από μία φορά και αποθηκεύονται (πχ σε ένα πίνακα) για όλες τις πιθανές φορές που θα πρέπει να χρησιμοποιηθούν.

Τα προβλήματα που αφορούν τον δυναμικό προγραμματισμό είναι συνήθως προβλήματα βελτιστοποίησης, δηλαδή εύρεση μίας βέλτιστης λύσης ενώ ταυτόχρονα η λύση αυτή να υπόκειται και σε κάποιους περιορισμούς. Η τεχνική μοιάζει αρκετά με την μέθοδο του «διαίρει και βασίλευε» σε σχέση με την διαίρεση του προβλήματος σε μικρότερα και απλούστερα αλλά είναι δυνατόν να μην είναι του ίδιου τύπου. Τα βασικά συστατικά των αλγορίθμων δυναμικού προγραμματισμού είναι τα:

1) Διαίρεση του προβλήματος σε μικρότερα και απλούστερα μέρη.

2) Αποθήκευση των λύσεων σε ένα πίνακα. Αυτό γίνεται γιατί πολλές αν όχι όλες από τις λύσεις των υπό-προβλημάτων πρόκειται να ξαναχρησιμοποιηθούν και δεν ενδείκνυται το να επιλυθούν ξανά και ξανά.
3) Συνδυασμός των λύσεων των μικρών κομματιών του προβλήματος ώστε να οδηγούν σε λύσεις μεγαλύτερων κομματιών μέχρι να επιλυθεί όλο το πρόβλημα.

6.2 Πολλαπλασιασμός Πινάκων

Ένα ενδιαφέρον και ιδιαίτερα χρονοβόρο μαθηματικό πρόβλημα (ως προς τον απαιτούμενο χρόνο υπολογισμού του), είναι ο πολλαπλασιασμός πολλών πινάκων, πχ $P = A_1 \cdot A_2 \cdot \cdots \cdot A_N$.

Είναι γνωστό ότι το γινόμενο δύο πινάκων, έστω $A_{Q,R} \cdot B_{R,S}$ δίνεται από τον τύπο:

$$a_{i,k} = \sum_{j=1}^{R} (a_{i,j} \cdot a_{j,k}),\ i = 1,2,\ldots,Q \text{ και } k = 1,2,\ldots,S$$

και απαιτεί $Q \cdot R \cdot S$ πολλαπλασιασμούς. Δηλαδή, ανάλογα και με τα μεγέθη των πινάκων, απαιτεί ένα μεγάλο αριθμό πράξεων

Το πρόβλημα έγκειται στο ότι ανάλογα με την σειρά που θα εκτελεσθούν οι πολλαπλασιασμοί, οι απαιτούμενες πράξεις διαφέρουν και μάλιστα κατά πολύ. Για παράδειγμα, έστω ότι ζητείται το γινόμενο των πινάκων:

$$P = A_{100,1} \cdot B_{1,100} \cdot C_{100,4}$$

Εάν ο πολλαπλασιασμός γίνει με την σειρά:

$$P = (A_{100,1} \cdot B_{1,100}) \cdot C_{100,4}$$

απαιτεί 100x1x100=10000 πράξεις για τον πρώτο πολλαπλασιασμό και 100x100x4=40000 για τον δεύτερο άρα σύνολο 50000 πράξεις.

Ενώ αν γίνει με την σειρά:

$$P = A_{100,1} \cdot (B_{1,100} \cdot C_{100,4})$$

απαιτεί 1x100x4=400 πράξεις για τον πολλαπλασιασμό των B και C ο οποίος θα εκτελεσθεί πρώτος, και 100x1x4=400 για τον πολλαπλασιασμό του A με το αποτέλεσμα των άλλων δύο, δηλαδή σύνολο 800 πράξεις. Όπως φαίνεται, η διαφορά είναι παραπάνω από αρκετή για να δώσει το ερέθισμα του ψαξίματος της σειράς των πολλαπλασιασμών (ή για το που πρέπει να υπάρχουν παρενθέσεις ώστε να προσδιορίζουν την προτεραιότητα των πράξεων).

Η τυποποίηση του προβλήματος αυτού είναι η ακόλουθη: Δεδομένης μίας σειράς πινάκων προς πολλαπλασιασμό, ζητείται να προσδιορισθεί η σειρά των πινάκων ώστε να ελαχιστοποιείται ο αριθμός των πράξεων.

Φυσικά, ο αλγόριθμος που θα επιλύσει το πρόβλημα δεν ασχολείται με τον πολλαπλασιασμό αυτό καθαυτό αλλά με την σειρά των πολλαπλασιαστέων. Η προφανής λύση αυτού του προβλήματος είναι να δοκιμασθούν όλοι οι πιθανοί συνδυασμοί. Εάν το κόστος που απαιτείται αποδίδεται από την συνάρτηση $P(N)$, όπου N το πλήθος των υπό πολλαπλασιασμό πινάκων, τότε (διαίρει και βασίλευε) εάν η αλυσίδα του πολλαπλασιασμού διασπασθεί σε δύο λίστες, η μία k πινάκων και η άλλη N-k (προφανές) αυτό δείχνει ότι $P(N) = P(k) \cdot P(N-k)$ γιατί, για κάθε συνδυασμό της πρώτης λίστας αντιστοιχούν όλοι οι συνδυασμοί της άλλης. Επομένως, η συνάρτηση κόστους μπορεί να αποδοθεί αναδρομικά ως εξής:

$$P(N) = \begin{cases} 1, N = 1 \\ \sum_{k=1}^{N-1} P(k) \cdot P(N-k), N \geq 2 \end{cases}$$

Ο παραπάνω τύπος όμως, συσχετίζεται με μία γνωστή συνάρτηση της Συνδυαστικής Ανάλυσης, την συνάρτηση των *Catalan Numbers*, με $P(N)=C(N-1)$ όπου

$$C(N) = \frac{1}{N+1} \cdot \binom{2 \cdot N}{N}.$$

(Ένα κλασσικό πρόβλημα που σχετίζεται με τους *Catalan numbers* είναι το περίφημο πρόβλημα του *Euler*, όπου τίθεται το ερώτημα με πόσους τρόπους ένα κανονικό πολύγωνο N κορυφών διαιρείται σε N-2 τρίγωνα)

Η πολυπλοκότητα αυτής της μεθόδου είναι της τάξης $O(4^N)$ και κατά συνέπεια απαγορευτική εκτός και εάν το N είναι ένας πολύ μικρός αριθμός. Επομένως θα πρέπει να βρεθεί μία άλλη μέθοδος.

Η λύση που προκύπτει με χρήση των μεθόδων του δυναμικού προγραμματισμού, είναι η διάσπαση του προβλήματος σε μικρότερα και στη συνέχεια ο συνδυασμός αυτών των λύσεων ώστε να οδηγήσει στην συνολική λύση του προβλήματος.

Έστω ότι $A_1 \cdot A_2 \cdot \cdots \cdot A_N$ είναι η λίστα των προς πολλαπλασιασμό πινάκων και έστω ότι $A_{i\ldots j}$ είναι το αποτέλεσμα του πολλαπλασιασμού των πινάκων με δείκτες από i μέχρι j. Τότε, το ζητούμενο είναι:

$$A_{1\ldots N} = A_1 \cdot A_2 \cdot \cdots \cdot A_N$$

Εάν το πρόβλημα διασπασθεί σε δύο επιμέρους προβλήματα, αυτό σημαίνει ότι:

$$A_{1\ldots N} = A_{1\ldots k} \cdot A_{k+1\ldots N} \qquad \text{για } 1 \leq k \leq N$$

Επομένως το πρόβλημα εστιάζεται στον υπολογισμό του βέλτιστου δείκτη *k*. Προφανώς για να βρεθεί αυτός ο βέλτιστος δείκτης πρέπει να ψαχθούν όλοι οι πιθανοί συνδυασμοί, δηλαδή όλοι οι πιθανοί δείκτες και στη συνέχεια επιλογή του βέλτιστου. Εάν αυτό επιτευχθεί, τότε δεν χρειάζεται τίποτα άλλο παρά την διάσπαση των δύο σειρών σε μικρότερες ξανά και ξανά μέχρι η σειρά να αποτελείται από μόνο ένα πίνακα όπου δεν τίθεται θέμα κανενός υπολογισμού (δεν θα υπάρχουν τουλάχιστον δύο πίνακες οπότε δεν θα υπάρχει τίποτα να πολλαπλασιασθεί). Τέλος, από τον συνδυασμό των λύσεων από κάτω προς τα πάνω, θα προκύψει και η συνολική λύση.

Ανάλυση: Προφανώς οι επιμέρους λύσεις των υποπροβλημάτων θα αποθηκεύονται σε πίνακες (αρχή της μεθόδου του δυναμικού προγραμματισμού). Έστω λοιπόν δύο πίνακες *N*x*N*, m και s όπου θα αποθηκεύονται τα κόστη των γινομένων και η προτεραιότητα των πράξεων (θέσεις των παρενθέσεων) αντίστοιχα. Το βέλτιστο κόστος του γινομένου $A_{i\ldots j}$ θα αποθηκεύεται στην θέση m_{ij} (και επομένως ο πίνακας θα πρέπει να αρχικοποιηθεί στο 0). Ο πίνακας m θα πρέπει να ορίζεται αναδρομικά ως εξής:

$$m_{i,i} = 0$$

$$m_{i,j} = \min_{i \le k < j} (m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j)$$

Η διαδικασία δίνεται στην συνάρτηση του *Προγράμματος 11*. Οι αντίστοιχες τιμές του πίνακα s περιγράφουν το σημείο «αποκοπής» της αλυσίδας ώστε να προκύψει η βέλτιστη λύση.

Πρόγραμμα 11: Πολλαπλασιασμός αλυσίδας πινάκων

```
#include <stdio.h>
#define N 100

/* Πολλαπλασιασμός Αλυσίδας Πινάκων */

int p[N];
int s[N][N];
long m[N][N];

void matrix_chain(int);
void mult(int, int);

/* ------------------------------------------------------ */

main()
{

  int x;
  int i,j;

  printf("\n Πλήθος πινάκων:");
  scanf("%d",&x);
```

```
  for (i=0; i<=x; i++) {
    printf("\n Είσοδος %d διάστασης:",i);
    scanf("%d",&p[i]);
  }

  matrix_chain(x);

  printf("\n\n Πίνακας Κόστους");
  for (i=1; i<x; i++) {
    printf("\n");
    for (j=1; j<=x; j++)
      printf("%8d", m[i][j]);
  }


  printf("\n\n Πίνακας Προτεραιοτήτων");
  for (i=1; i<x; i++) {
    printf("\n");
    for (j=1; j<=x; j++)
      printf("%8d", s[i][j]);
  }

}

void matrix_chain(int n)
{
  int i,j,k,L,q;

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

  for (L=2; L<=n; L++) {
    for (i=1; i<= n-L+1; i++) {
      j = i + L -1;
      m[i][j] = 1000000; /* Υποκαθιστά το άπειρο */
      for (k=i; k<=j-1; k++) {
        q = m[i][k]+m[k+1][j] + p[i-1]*p[k]*p[j];
        if (q < m[i][j]) {
          m[i][j] = q;
          s[i][j] = k;
        }
      }
    }
  }
}
```

Έστω, ότι ζητείται να γίνει ο πολλαπλασιασμός των πινάκων:

$$A_{1...5} = A_{3,4} \cdot A_{4,2} \cdot A_{2,7} \cdot A_{7,3} \cdot A_{3,4}$$

Η είσοδος δεδομένων στο *Πρόγραμμα 10* θα είναι η εξής:

⇒ Πλήθος πινάκων: **5**
⇒ Είσοδος 0 διάστασης: **3**
⇒ Είσοδος 1 διάστασης: **4**
⇒ Είσοδος 2 διάστασης: **2**
⇒ Είσοδος 3 διάστασης: **7**
⇒ Είσοδος 4 διάστασης: **3**
⇒ Είσοδος 5 διάστασης: **4**

Και οι δύο πίνακες αποτελέσματα θα είναι:

Πίνακας Κόστους

0	24	66	84	**114**
0	0	56	66	98
0	0	0	42	**66**
0	0	0	0	84

Προφανώς ο βέλτιστος αριθμός πράξεων δίνεται στη θέση 1,5 και είναι 114 (γιατί ο ζητούμενος πολλαπλασιασμός αποδίδεται σαν $A_{1...5}$).

Πίνακας Προτεραιοτήτων

0	1	2	2	**2**
0	0	2	2	2
0	0	0	3	**4**
0	0	0	0	4

Σε συνδυασμό με τον πίνακα s (και στις αντίστοιχες θέσεις), παρατηρείται ότι ο πρώτος διαχωρισμός πρέπει να γίνει στον δεύτερο πίνακα, δηλαδή $A_{1...5} = (A_{3,4} \cdot A_{4,2}) \cdot A_{2,7} \cdot A_{7,3} \cdot A_{3,4}$. Άρα εάν χρειάζεται να βρεθεί το κόστος αυτού του πρώτου γινομένου αυτό αποδίδεται στον πρώτο πίνακα και στην θέση 1,2 δηλαδή 24 πράξεις. Στην συνέχεια για να βρεθεί η σειρά και το κόστος του γινομένου $A_{3...5}$ αρκεί να αναζητηθούν στους πίνακες m και s πάλι στις αντίστοιχες θέσεις. Δηλαδή, κόστος 66 και 4. Τελικά, η σειρά εκτέλεσης των πράξεων είναι:

$$A_{1...5} = (A_{3,4} \cdot A_{4,2}) \cdot ((A_{2,7} \cdot A_{7,3}) \cdot A_{3,4})$$

6.3 Ασκήσεις

1. String editing: Ένα σημαντικό πρόβλημα είναι ο υπολογισμός του κατά πόσο μία συμβολοσειρά εμφανίζεται μέσα σε κάποια άλλη. Στην περίπτωση δε που δεν υπάρχει ακριβές αντίγραφο (έχει δοθεί ο αντίστοιχος brute-force αλγόριθμος) τότε το ζητούμενο είναι πόσο προσεγγίζεται η αναζητούμενη συμβολοσειρά στην συμβολοσειρά στόχος. Για παράδειγμα, ένα η προς αναζήτηση συμβολοσειρά είναι η BCDE και η συμβολοσειρά στόχος είναι η ABQWSAA**BCD**RTY, τότε η προς αναζήτηση συμβολοσειρά απαντάται σε ποσοστό ¾ ή 75% Να σχεδιασθεί

αλγόριθμος με την βοήθεια της τεχνικής του δυναμικού προγραμματισμού που να επιλύει αυτό το πρόβλημα.

2. Έστω ότι υπάρχουν *N* εργασίες και δύο μηχανές *A* και *B*, που μπορούν να τις εκτελέσουν. Εάν η εργασία i εκτελεσθεί από την μηχανή *A* τότε έστω ότι χρειάζεται a_i χρονικές μονάδες για να εκτελεσθεί ενώ εάν εκτελεσθεί από την μηχανή *B* τότε έστω ότι χρειάζεται b_i χρονικές μονάδες. Λόγω της φύσης των διαφόρων εργασιών είναι δυνατόν να ισχύει $a_i \geq b_i$ για κάποιο i ενώ να ισχύει $a_j < b_j$ για κάποιο j. Να σχεδιασθεί ένας αλγόριθμος δυναμικού προγραμματισμού ο οποίος να υπολογίζει τον ελάχιστο χρόνο που απαιτείται για να εκτελεσθούν οι εργασίες.

3. Στο προηγούμενο πρόβλημα, δώστε ένα παράδειγμα 10 εργασιών με τυχαίους χρόνους εκτέλεσης.

7. ΑΠΛΗΣΤΟΙ ΑΛΓΟΡΙΘΜΟΙ

Πολλές φορές η τεχνική του δυναμικού προγραμματισμού είναι εντελώς ασύμφορη για κάποια προβλήματα βελτιστοποίησης και αυτό γιατί ψάχνει όλες τις πιθανές λύσεις επιζητώντας την βέλτιστη. Σε αντίθεση με αυτή την τεχνική, είναι πιθανώς καλό σε κάποιες περιπτώσεις αντί να αναζητά κανείς την συνολικά βέλτιστη λύση, να αναζητά την τοπικά βέλτιστη η οποία μπορεί να οδηγήσει (μπορεί και όχι) στην συνολική βέλτιστη. Αυτή η τεχνική αλγορίθμων ονομάζεται «απληστία» (greedy algorithms) διότι κάθε φορά επιλέγει με καθαρά τοπικά κριτήρια μία λύση η οποία μπορεί να οδηγήσει στην βέλτιστη. Αυτή η μέθοδος χρησιμοποιείται τακτικότατα στην καθημερινή ζωή με πιο χαρακτηριστικό παράδειγμα το πρόβλημα των «ρέστων». Όταν κάποιος επιχειρεί να επιστρέψει ρέστα, αντί να υπολογίσει τον ολικά βέλτιστο αριθμό νομισμάτων, αρχίζει με το μεγαλύτερο νόμισμα που μπορεί να επιστρέψει και συνεχίζει με αυτόν τον τρόπο (*Αλγόριθμος 14* και *Πρόγραμμα 12*). Σε αντίθεση με την τεχνική του δυναμικού προγραμματισμού η οποία εξαρτάται από τις λύσεις των επιμέρους υποπροβλημάτων και κινείται από κάτω προς τα πάνω (bottom up), οι άπληστοι αλγόριθμοι εξαρτώνται από τις επιλογές που έχουν κάνει μέχρι τώρα για να συνεχίσουν (top down). Φυσικά, δεν είναι δυνατή η λύση όλων των προβλημάτων βελτιστοποίησης με αυτή τη τεχνική αλλά ενδείκνυται για κάποια από αυτά.

Αλγόριθμος 14: Το πρόβλημα της επιστροφής ρέστων

1: Δεδομένα: Πίνακας C_8 διαθεσίμων νομισμάτων (200-100-50-20-10-5-2-1) και L πίνακας κερμάτων που απαιτούνται
2: Ακέραιος αριθμός: Συνάρτηση Υπολογισμού ρέστων(ακέραιος N)
3: /* Το N αναπαριστά το ποσό που πρέπει να επιστραφεί */
4: Ακέραιοι: i ← 0, x, S ← 0
5: /* Το x αναπαριστά το εκάστοτε νόμισμα και x το σύνολο που έχει υπολογισθεί. Το i αναπαριστά το σύνολο των νομισμάτων που θα χρειαστούν */
6: Εφόσον ($S < N$)
7: Επέλεξε το μεγαλύτερο νόμισμα x από τον πίνακα C τέτοιο ώστε να μην υπερβαίνει το υπολειπόμενο ποσό N – s
8: i ← i+1
9: S ← S + x
10: Τέλος Εφόσον
11: Επέστρεψε το i
12: Τέλος αλγόριθμου «Υπολογισμού ρέστων»

Πρόγραμμα 12: Το πρόβλημα της επιστροφής ρέστων

```
#include <stdio.h>

int c[8] = {200, 100, 50, 20, 10, 5, 2, 1}; /* Κέρματα */
int L[100]; /* Πίνακας κερμάτων που θα επιστραφούν σαν ρέστα */

int resta(int);
```

```
main()
{

  int N;  /* Επιστρεφόμενο ποσό */
  int p; /* Πλήθος επιστρεφόμενων νομισμάτων */
  int i;
  int s = 0;

  printf("\n Εισαγωγή του επιστρεφόμενου ποσού (σε λεπτά):");
  scanf("%d",&N);

  p = resta(N);

  for (i=0; i<p; i++) {
    printf("\n %5d: %d", i+1, L[i]);
    s += L[i];
  }

  printf("\n\n Χρειάστηκαν %d για το σύνολο του επιστρεφόμενου ποσού %d λεπτά
\n", p,s);
}

/* Συνάρτηση υπολογισμού ρέστων */

int resta(int n)
{
  int i=0,j;
  int x, s=0;

  while (s < n) {
    j = -1;
    while (j<7 && (x=c[++j]) > n-s);
    L[i++] = x;
    s += x;
  }
  return i;
}
```

Γενικά, ένα πρόβλημα που επιδέχεται σαν λύση έναν άπληστο αλγόριθμο, εμφανίζει τα ακόλουθα χαρακτηριστικά:

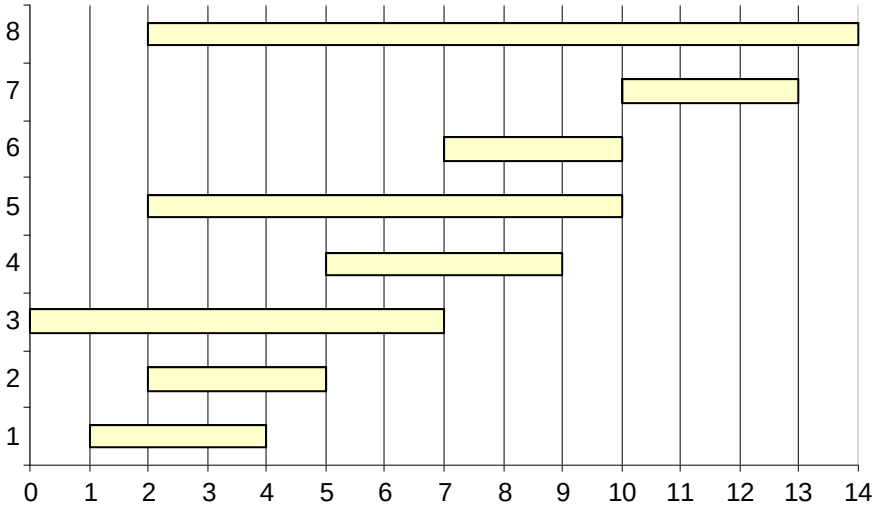
Ιδιότητα Άπληστης Επιλογής: Η ολικά βέλτιστη λύση είναι δυνατή να παραχθεί κατόπιν μίας και μοναδικής τοπικά βέλτιστης επιλογής και προχωρώντας σε επόμενη κίνηση. Δηλαδή, είναι περιττή η διερεύνηση υποπροβλημάτων και σύνθεση των λύσεών τους.

Βέλτιστη Υποδομή: Οι άπληστοι αλγόριθμοι, δρουν επαγωγικά και αποδεικνύεται πως η άπληστη επιλογή παρέχει συνολική βέλτιστη λύση (αφού παράγει την βέλτιστη λύση του κάθε μοναδικού υποπροβλήματος που παράγεται).

7.1 Προγραμματισμός Εργασιών

Έστω ένα σύνολο εργασιών $T = \{t_1, t_2, \cdots, t_N\}$ οι οποίες εκτελούνται από μία μηχανή (πχ προγράμματα που πρέπει να εκτελεσθούν σε έναν υπολογιστή). Η κάθε εργασία t_i ορίζεται από το χρόνο που απαιτεί για να εκτελεσθεί και χαρακτηρίζεται από τον χρόνο έναρξης s_i και το χρόνο λήξης e_i, ενώ η μηχανή μπορεί να εκτελεί μία εργασία κάθε φορά. Το ζητούμενο είναι να βρεθεί το πλήθος των εργασιών που μπορούν να εκτελεσθούν από την μηχανή εντός συγκεκριμένου χρόνου. Αυτό αποτελεί ένα θεμελιώδες πρόβλημα όπου πολλά άλλα αντίστοιχα ανάγονται σε αυτό, όπως για παράδειγμα πόσα μαθήματα (μέγιστος αριθμός) μπορούν να γίνουν σε μία αίθουσα εάν είναι γνωστά τα μαθήματα και οι ώρες έναρξης και λήξης τους.

Επειδή υπάρχει μόνο ένας διαθέσιμος πόρος (πχ μηχανή, αίθουσα, κλπ), είναι πολύ πιθανόν να μην μπορέσουν να εξυπηρετηθούν όλες οι εργασίες. Έστω για παράδειγμα ότι $T = \{t_1, t_2, \cdots, t_8\}$ με χρόνους έναρξης και λήξης {(1,4), (2,5), (0,7), (5,9), (2,10), (7,11), (10,13), (2,14)} αντίστοιχα (*Σχήμα 7.1*). Είναι προφανές ότι τουλάχιστον μία από τις εργασίες t_1 και t_2 δεν πρόκειται να εξυπηρετηθούν. Το ζητούμενο είναι να προγραμματισθούν όσο τον δυνατόν περισσότερες.



Σχήμα7.1: Χρονοπρογραμματισμός εργασιών

Εφόσον πρέπει να μεγιστοποιηθεί ο αριθμός των εργασιών που θα εξυπηρετηθούν είναι εύκολο το συμπέρασμα ότι δεν συμφέρει να προγραμματίζονται χρονοβόρες εργασίες αλλά εργασίες με μικρό χρόνο εκτέλεσης. Αυτή η διαπίστωση οδηγεί στην εξής άπληστη στρατηγική: Επαναληπτική επιλογή της εργασίας με τον μικρότερο χρόνο εξυπηρέτησης (e_i-s_i) και προγραμματισμός της αρκεί να μην αλληλεπιδρά με την προηγούμενη επιλεγείσα εργασία.

Ένας λοιπόν άπληστος αλγόριθμος που βασίζεται σ' αυτή την στρατηγική θα μπορούσε να είναι ο εξής (*Αλγόριθμος 15*). Υποτίθεται ότι το πλήθος των εργασιών είναι *N*, και ότι στην μεταβλητή pr αποθηκεύεται η προηγούμενη εργασία που έχει ήδη προγραμματισθεί. Επίσης, πρέπει να ταξινομηθεί το σύνολο των εργασιών με βάση το χρόνο λήξης τους έτσι ώστε πάντα να επιλέγεται σαν πρώτη εργασία η

πρώτη του νέου ταξινομημένου πλέον συνόλου εργασιών. Τέλος, σε ένα πίνακα, έστω *A*, αποθηκεύονται οι εργασίες που δρομολογούνται προς εξυπηρέτηση.

Αλγόριθμος 15: Χρονοπρογραμματισμός εργασιών

1: Ταξινόμηση όλων των εργασιών με κλειδί τον χρόνο λήξης τους έτσι ώστε $e_1 \leq e_2 \leq \cdots \leq e_N$.
2: i ← 1
3: A_i ← i,
4: pr ← 1
5: Για k ←2 μέχρι *N*
6: Εάν (sk ≥ epr) Τότε
7: i ← i +1
8: A_i ← k
9: pr ← k
10: Τέλος Εάν
11: Τέλος Για (k)
12: Επέστρεψε τον πίνακα *A*
13: Τέλος αλγόριθμου «Χρονοπρογραμματισμός εργασιών»

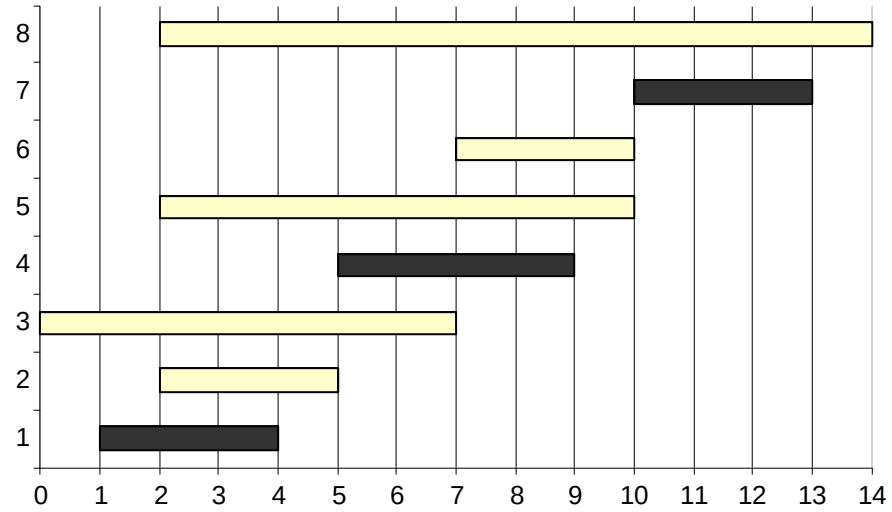
Το *Πρόγραμμα 13* υλοποιεί τον *Αλγόριθμο 15*.

Πρόγραμμα 13: Χρονοπρογραμματισμός εργασιών

```
#include <stdio.h>
/* Προγραμματισμός Εργασιών */

#define N 10

typedef struct {
     int arxi;
     int telos;
  } Task;

Task task[N]; /* Πίνακας εργασιών */
int D[N]; /* Πίνακας δραστηριοτήτων */

int task_sch(int);

main()
{
  int n; /* Πλήθος εργασιών */
  int i,j;
  int p; /* Πλήθος επιλεγέντων εργασιών */
  Task temp;

  /* Εισαγωγή πλήθους εργασιών και χρόνων */
  printf("\n Πλήθος Εργασιών (<%d):",N);
  scanf("%d",&n);
  while (n>N)
    scanf("%d",&n);
```

```
  for (i=0; i<n; i++) {
    printf("\n Εισαγωγή %d εργασίας (χρόνοι έναρξης και λήξης):",i+1);
    scanf("%d%d",&task[i].arxi, &task[i].telos);
  }

  /* Ταξινόμηση εργασιών (αύξουσα) ως προς τον χρόνο λήξης */
  for (i=0; i<n-1; i++)
    for (j=n-1; j>i; j--)
      if (task[j].telos < task[j-1].telos) {
        temp = task[j];
        task[j] = task[j-1];
        task[j-1] = temp;
      }

  printf("\n\n ΤΑΞΙΝΟΜΗΜΕΝΣ ΕΡΓΑΣΙΕΣ\n\n");
  for (i=0; i<n; i++)
    printf("\n%d: %d - %d", i+1, task[i].arxi, task[i].telos);


  p = task_sch(n);
  printf("\n\n ΑΠΟΤΕΛΕΣΜΑΤΑ \n");
  for (i=0; i<=p; i++)
    printf("\n %d Εργασία : %d",i+1,D[i]+1);
}

/* Συνάρτηση χρονοπρογραμματισμού */

int task_sch(int n)
/* Ενημερώνει τον πίνακα δραστηριοτήτων και επιστρέφει το πλήθος των εργασιών
που προγραμματίστηκαν */
{
  int i,k=0;
  int pr; /* Προηγούμενη προγραμματισμένη εργασία */

  D[0] = 0;
  pr = 0;

  for (i=1; i<n; i++)
    if (task[i].arxi >= task[pr].telos) {
      D[++k] = i;
      pr = i;
    }
  return k;
}
```

Εάν στο πρόγραμμα εισαχθούν οι εργασίες του παραδείγματος, η λύση φαίνεται στο *Σχήμα 7.2*.



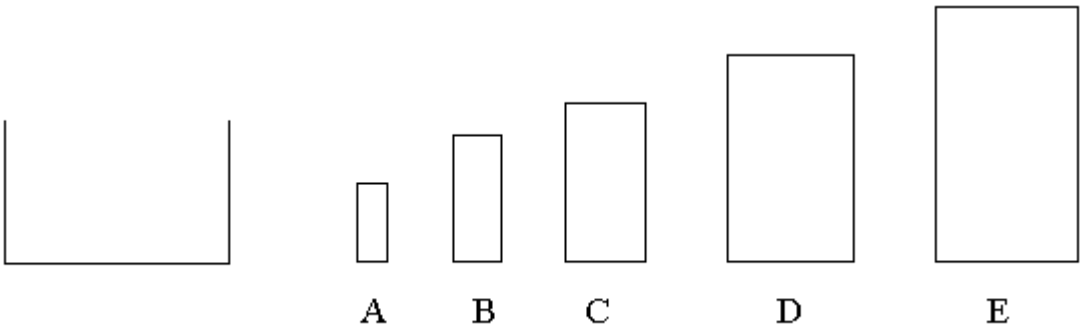
Σχήμα 7.2: Επιλογή εργασιών προς εξυπηρέτηση

7.2 "Knapsack Problem"

Ένα πολύ ενδιαφέρον πρόβλημα το οποίο εμφανίζεται και με πάρα πολλές παραλλαγές είναι το λεγόμενο "knapsack problem": Ας υποτεθεί ότι ένας κλέφτης παραβιάζει ένα θησαυροφυλάκιο το οποίο περιέχει *N* αντικείμενα διαφορετικού μεγέθους αλλά και αξίας. Ο κλέφτης έχει ένα σάκο περιορισμένου φυσικά μεγέθους, και το ζητούμενο είναι ποια αντικείμενα πρέπει να πάρει ώστε να έχουν την μεγαλύτερη δυνατή αξία. Για παράδειγμα, έστω ότι υπάρχουν 5 αντικείμενα με μεγέθη και αξίες όπως στον *Πίνακα 7.1*, και *Σχήμα 7.3*.

Πίνακας 7.1: Παράδειγμα του knapsack problem

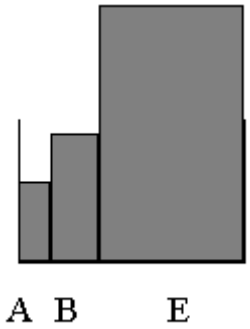
	1	**2**	**3**	**4**	**5**
Αντικείμενο	A	B	C	D	E
Μέγεθος	3	4	7	8	9
Αξία	4	5	10	11	13



Σχήμα 7.3: Knapsack problem

Το ζητούμενο είναι να επιλεγούν αντικείμενα με τέτοιο τρόπο, ώστε και να χωρούν στον σάκο ενώ ταυτόχρονα και η αξία του να είναι η όσο το δυνατόν μεγαλύτερη

(παράδειγμα στο *Σχήμα 7.4*). Βέβαια, ένας τρόπος είναι να ελεγχθούν όλοι οι δυνατοί συνδυασμοί, όμως αυτό οδηγεί σε λύσεις εκθετικού χρόνου και άρα απαγορευτικές ιδιαίτερα για μεγάλες τιμές. Η λύση αυτού του προβλήματος είναι ιδιαίτερα σημαντική γιατί πάρα πολλές εφαρμογές οδηγούν σε ανάλογα προβλήματα όπως για παράδειγμα ο τρόπος που θα φορτωθεί ένα πλοίο ή αεροπλάνο, ο τρόπος που θα τοποθετηθούν αντικείμενα σε μία αποθήκη και πολλά άλλα. Ένας αναδρομικός αλγόριθμος, θα προσέγγιζε το πρόβλημα ως εξής: Για κάθε αντικείμενο που θα επέλεγε θα εξέταζε όλους τους πιθανούς συνδυασμούς και τέλος θα κρατούσε την βέλτιστη λύση. Η πολυπλοκότητα της μεθόδου είναι της τάξης $O(N \cdot M)$ - (πλήθος x χωρητικότητα), δηλαδή ιδιαίτερα «ακριβή» για μεγάλα μεγέθη. Ο αναδρομικός αυτός αλγόριθμος περιγράφεται στο *Πρόγραμμα 14*.



Σχήμα 7.4: Μία λύση του knapsack problem

Πρόγραμμα 14: Knapsack problem – Αναδρομικός Αλγόριθμος

```
#include <stdio.h>
#define N 5
/* Μην εκτελέσετε αυτό το πρόγραμμα για μεγάλο N και x */
/* Ο χρόνος εκτέλεσης αυξάνει εκθετικά */
typedef struct {
     int size;
     int val;
  } Item;

Item items[N] ;
int knap(int);

main()
{
  int i;
  int megisto;
  int x;  /* Χωρητικότητα */

  for (i=0; i<N; i++) {
    printf("\n Εισαγωγή μεγέθους – τιμής του %d αντικειμένου:",i);
    /* Η εισαγωγή με αύξουσα σειρά τάξης μεγέθους */
    scanf("%d%d",&items[i].size,&items[i].val);
  }
```

```
  printf("\n Χωρητικότητα σάκου=");
  scanf("%d",&x);

  /* Προτεινόμενα μεγέθη του Πίνακα ?? και χωρητικότητα έως 40 */

  megisto = knap(x);

  printf("\n Μέγιστο κέρδος =%d\n", megisto);

}

/* Υπολογισμός βέλτιστου */

int knap(int megethos)
{
  int i, space, meg, t;

  for (i=0, meg=0; i<N; i++)
    if ((space=megethos-items[i].size)>=0)
      if ((t=knap(space)+items[i].val)>meg)
        meg = t;

  return meg;
}
```

Βέβαια, ένας άπληστος αλγόριθμος δεν θα λειτουργούσε ποτέ με αυτόν τον τρόπο. Κατ' αρχήν θα επέλεγε το αντικείμενο με την μεγαλύτερη δυνατή αξία και θα συνέχιζε με αυτόν τον τρόπο μέχρι να γεμίσει ο σάκος ή μέχρι να μην μπορεί να βρει άλλο αντικείμενο που να χωράει στον σάκο. Αυτό σημαίνει ότι ή άπληστη προσέγγιση του προβλήματος μπορεί να είναι σαφώς ταχύτερη αλλά δεν βρίσκει πάντα την βέλτιστη λύση. Ο άπληστος αλγόριθμος του προβλήματος αυτού δίνεται στο *Πρόγραμμα 15*.

Πρόγραμμα 15: Knapsack problem – Άπληστος Αλγόριθμος

```
#include <stdio.h>
#define N 5

typedef struct {
     int size;
     int val;
  } Item;

Item items[N] ;
int V[N]; /* Πίνακας που θα κρατάει τα αντικείμενα που εισήχθησαν στον σάκο */

int knap(int);

main()
{
```

```
  int i,p;
  int megisto =0;
  int x;  /* Χωρητικότητα */

  for (i=0; i<N; i++) {
    printf("\n Εισαγωγή μεγέθους – τιμής του %d αντικειμένου:",i);
    /* Η εισαγωγή με αύξουσα σειρά τάξης μεγέθους */
    scanf("%d%d",&items[i].size,&items[i].val);
  }
  printf("\nXoritikotita=");
  scanf("%d",&x);

  p = knap(x);

  for (i=0; i<=p; i++) {
    printf("\n Αντικείμενο: %d", V[i]+1);
    megisto += items[V[i]].val;
  }

  printf("\n Μέγιστο =%d\n", megisto);

}

/* Υπολογισμός βέλτιστου */

int knap(int megethos)
{
  int i, U, k=-1;

  for (i=0; i<megethos; i++)
    V[i] = 0;
  U = megethos;

  i=N-1;
  while (i>=0)
    if ( items[i].size <= U) {
      V[++k] = i;
      U = U - items[i].size;
    }
    else
      i--;

  return k;
}
```

7.3 Ασκήσεις

1. Να τροποποιηθεί ο αναδρομικός αλγόριθμος του knapsack problem με την βοήθεια του σχεδιασμού αλγορίθμων δυναμικού προγραμματισμού ώστε να βελτιωθεί η πολυπλοκότητά του.

2. Υπάρχει περίπτωση ο αλγόριθμος «επιστροφής ρέστων» να μην παράγει βέλτιστη ή και καμία λύση; Αν ναι σε ποιες περιπτώσεις;

3. Το πρόβλημα του χρονοπρογραμματισμού εργασιών λυμένο με τον προταθέντα άπληστο αλγόριθμο παράγει βέλτιστη λύση;

8. ΠΑΡΑΛΛΗΛΟΙ ΑΛΓΟΡΙΘΜΟΙ

Όλοι οι αλγόριθμοι που έχουν αναλυθεί ή σχεδιασθεί μέχρι τώρα αφορούσαν υπολογιστικές μηχανές με ένα επεξεργαστή και κατά συνέπεια η εκτέλεσή τους ήταν σειριακή. Σ' αυτό το κεφάλαιο θα αναπτυχθούν αλγόριθμοι που θα αφορούν υπολογιστές που διαθέτουν παραπάνω από ένα επεξεργαστές οι οποίοι μπορούν να δουλεύουν ταυτόχρονα και να μοιράζονται (αν είναι απαραίτητο) κοινές πληροφορίες. Στην καθημερινή ζωή υπάρχουν πάρα πολλές εφαρμογές οι οποίες πρέπει να δουλεύουν σε πραγματικό χρόνο ενώ ταυτόχρονα πρέπει να αναλύουν και επεξεργάζονται τεράστιες ποσότητες δεδομένων, όπως για παράδειγμα ένα σύστημα ελέγχου εναέριας κυκλοφορίας, ή ένα σύστημα εντοπισμού και παρακολούθησης έντονων μετεωρολογικών φαινομένων. Σε τέτοιες περιπτώσεις ακόμη και οι πιο γρήγοροι υπολογιστές (με έναν επεξεργαστή) δεν θα μπορούσαν να αντεπεξέλθουν. Βέβαια, οι άνθρωποι στην καθημερινή ζωή έχουν εφαρμόσει τέτοια μοντέλα παράλληλης επεξεργασίας. Για παράδειγμα, είναι σε ένα σούπερ μάρκετ οι ταμίες είναι σχεδόν πάντα πάνω από ένας με αποτέλεσμα να εξυπηρετούν παράλληλα πολλούς πελάτες, ή σε βιομηχανικές μονάδες να υπάρχουν παραπάνω από μία γραμμές παραγωγής με αποτέλεσμα την δημιουργία παράλληλα πολλών προϊόντων.

8.1 Εισαγωγικός Αλγόριθμος

Ας υποτεθεί ότι πρέπει να γίνει αλγόριθμος ο οποίος θα προσθέτει 100 αριθμούς οι οποίοι είναι αποθηκευμένοι σε ένα μονοδιάστατο πίνακα, έστω P_{100}. Ένας σειριακός αλγόριθμος θα έλυνε αυτό το πρόβλημα όπως φαίνεται στον *Αλγόριθμο16* .

Αλγόριθμος 16: Σειριακός Αλγόριθμος Πρόσθεσης 100 Αριθμών

```
1: Αριθμός: Σειριακή Πρόσθεση(Πίνακας P100)
2:      Ακέραιος i
3:      Αθροιστής S
4:      S ← 0
5:      Για i από 1 μέχρι 100
6:              S ← S + Pi
7:      Τέλος Για (i)
8:      Επέστρεψε S
9: Τέλος Αλγόριθμου «Σειριακός Αλγόριθμος Πρόσθεσης 100 Αριθμών»
```

Είναι προφανές ότι ο αλγόριθμος χρειάζεται να κάνει μία απόδοση αρχικής τιμής στον αθροιστή και 100 προσθέσεις μέχρι να ολοκληρωθεί και αποδώσει το αποτέλεσμα, δηλαδή σύνολο 101 στοιχειώδεις πράξεις. Σε περίπτωση όμως που υπήρχε η δυνατότητα να εκτελούνται ταυτόχρονα δύο προσθέσεις (από δύο επεξεργαστές) ο αλγόριθμος θα μπορούσε να τροποποιηθεί και να διαμορφωθεί όπως περιγράφεται στον *Αλγόριθμο 17*.

Αλγόριθμος 17: Παράλληλος Αλγόριθμος Πρόσθεσης 100 Αριθμών

```
1: Αριθμός: Σειριακή Πρόσθεση(Πίνακας P100)
2:      Παράλληλη Εκτέλεση (Επεξεργαστές Ε1, Ε2)
3:      Ε1 ΚΑΙ Ε2: Ακέραιος i
```

4: E1: Αθροιστής S1, E2: Αθροιστής S2
5: E1: Αθροιστής S
6: E1: S1 ← 0, E2: S2 ← 0
7: E1: Για i από 1 μέχρι 50, E2: Για i από 51 μέχρι 100
8: E1: S1 ← S1 + P_i, E2: S2 ← S2 + P_i
9: Τέλος Για (i)
10: E1: S ← S1 + S2
11: E1: Επέστρεψε S
12: Τέλος Αλγόριθμου «Παράλληλος Αλγόριθμος Πρόσθεσης 100 Αριθμών»

Ο χρόνος που απαιτεί ο παραπάνω παράλληλος αλγόριθμος είναι για 52 στοιχειώδεις πράξεις και κατά συνέπεια με μια πρώτη εκτίμηση είναι σχεδόν δύο φορές ταχύτερος από τον αντίστοιχο σειριακό. Αυτό όμως το οποίο δεν συνυπολογίσθηκε είναι ο χρόνος επικοινωνίας μεταξύ των επεξεργαστών. Δηλαδή, για να μπορέσει ο E1 να εκτελέσει την γραμμή 10 του Αλγόριθμου ?? πρέπει να έχει την πληροφορία για την τιμή της S2. Επομένως, εάν το κόστος της μεταφοράς αυτής της πληροφορίας θεωρηθεί όσο και το κόστος μίας στοιχειώδους πράξης, τότε ο πραγματικός λόγος της συμπεριφοράς των δύο αλγορίθμων είναι 53/101. Η προηγούμενη παρατήρηση όμως οδηγεί και στην σκέψη ότι όσους πιο πολλούς επεξεργαστές διαθέτει ένα σύστημα, τόσο πιο πολύ αυξάνεται το επικοινωνιακό κόστος και πιθανώς μερικές φορές να καθιστά απαγορευτική την χρήση μεγάλου αριθμού επεξεργαστών (τι θα γινόταν αν το προηγούμενο πρόβλημα επιλυόταν από 50 επεξεργαστές;).

8.2 Κόστος Παράλληλων Αλγόριθμων

Για να εκτιμηθεί σωστά ένας παράλληλος αλγόριθμος πρέπει να συγκριθεί πρώτα από όλα με τον καλύτερο δυνατό αντίστοιχο σειριακό αλγόριθμο. Μια αρκετά καλή ένδειξη του πόσο επαρκής είναι ένας καλός αλγόριθμος είναι η επιτάχυνση που προσφέρει σε σχέση με τον σειριακό και αυτό ορίζεται ως εξής:

$$\text{Επιτάχυνση} = \frac{\text{Χειρότερος χρόνος ττο τταχύτερο σειριακού αλγόριθμου}}{\text{Χειρότερος χρόνος ττο ππαράλληλυ αλγόριθμου}}$$

Επιπλέον, ένας σημαντικός παράγοντας που πρέπει να ληφθεί υπόψη είναι το πλήθος των συμμετεχόντων επεξεργαστών. Προφανώς, ο μεγαλύτερος αριθμός επεξεργαστών συνεπάγεται μεγαλύτερο οικονομικό κόστος αγοράς αλλά και κόστος συντήρησης. Επομένως, το κόστος ενός παράλληλου αλγόριθμου αυξάνεται με τον αριθμό των επεξεργαστών. Συμπερασματικά, το συνολικό κόστος δίνεται από τον τύπο:

Κόστος = (Επιτάχυνση) x (Αριθμός συμμετεχόντων επεξεργαστών)

και τελικά η απόδοση του αλγόριθμου μπορεί να αποδοθεί από τον τύπο:

$$\text{Απόδοση} = \frac{\text{Χειρότερος χρόνος ττοο τταχύτερου σειριακού αλγόριθμου}}{\text{Κόστος ππαράλληλυ αλγόριθμου}}$$

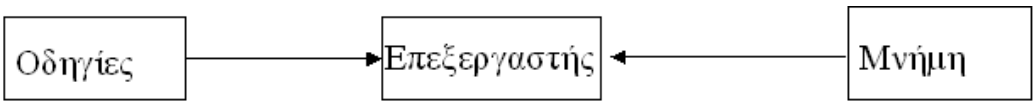
Φυσικά, εκτός από τους παραπάνω ορισμούς του κόστους, μπορούν να προσμετρηθούν και άλλα μεγέθη όπως η καλωδίωση για την δικτύωση των επεξεργαστών, η συνολική μνήμη, κ.α.

8.3 Τύποι Παράλληλων Υπολογιστικών Μηχανών

Μία παράλληλη υπολογιστική μηχανή (αλλά και με έναν επεξεργαστή) λειτουργεί εκτελώντας κάποιες οδηγίες επάνω σε κάποια δεδομένα. Ανάλογα με τον τρόπο διαχείρισης των δεδομένων, οι υπολογιστικές μηχανές διακρίνονται στις εξής κατηγορίες:

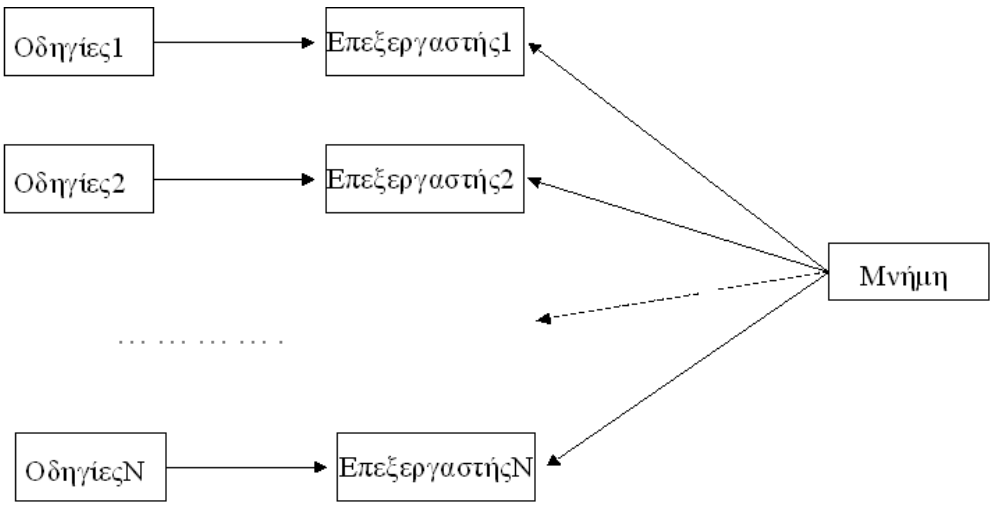
1. SISD: Σειριακή εκτέλεση οδηγιών – Σειριακή διαχείριση δεδομένων, *Σχήμα 8.1* (Single Instruction stream, Single Data stream).
2. MISD: Παράλληλη εκτέλεση οδηγιών – Σειριακή διαχείριση δεδομένων, *Σχήμα 8.2* (Multiple Instruction stream, Single Data stream)
3. SIMD: Σειριακή εκτέλεση οδηγιών – Παράλληλη διαχείριση δεδομένων, *Σχήμα 8.3* (Single Instruction stream, Multiple Data stream)
4. MIMD: Παράλληλη εκτέλεση οδηγιών – Παράλληλη διαχείριση δεδομένων, *Σχήμα 8.4* (Multiple Instruction stream, Multiple Data stream)

SISD: Σ' αυτό το μοντέλο, ένας επεξεργαστής δέχεται μία οδηγία κάθε φορά η οποία με την σειρά της επενεργεί σε δεδομένα που βρίσκονται καταχωρημένα στην μνήμη.



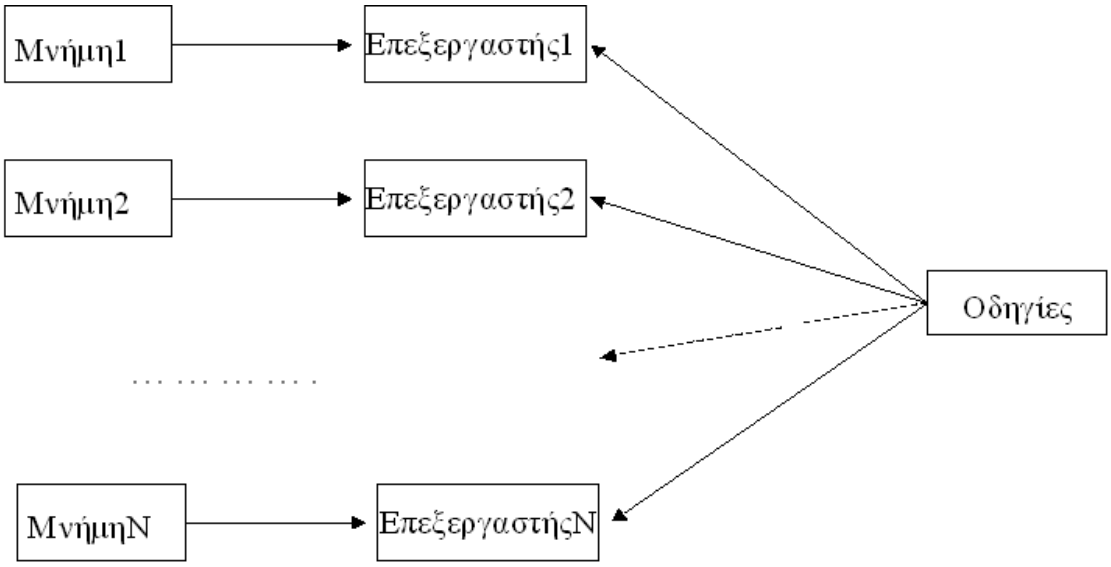
Σχήμα 8.1: SISD

MISD: Πολλοί επεξεργαστές εκτελούν διαφορετικές εργασίες αλλά όλοι μοιράζονται την ίδια μνήμη και αυτές οι διαφορετικές οδηγίες επενεργούν στα ίδια δεδομένα.



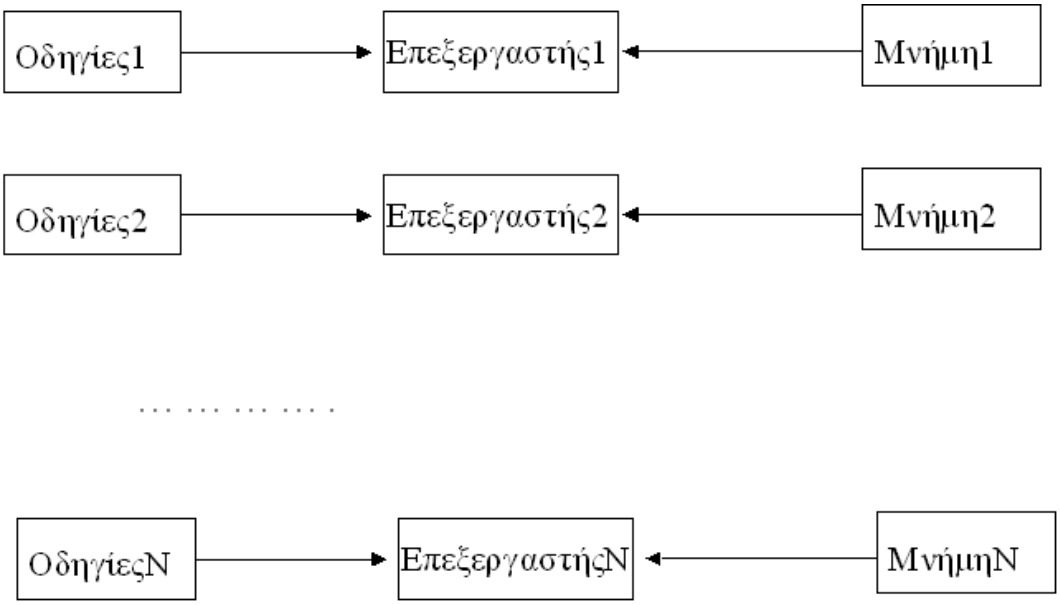
Σχήμα8.2 : MISD

SIMD: Αυτός ο τύπος παράλληλης υπολογιστικής μηχανής λειτουργεί με το να εκτελούν όλοι οι επεξεργαστές τις ίδιες οδηγίες αλλά αυτές να επιδρούν σε διαφορετικά δεδομένα.



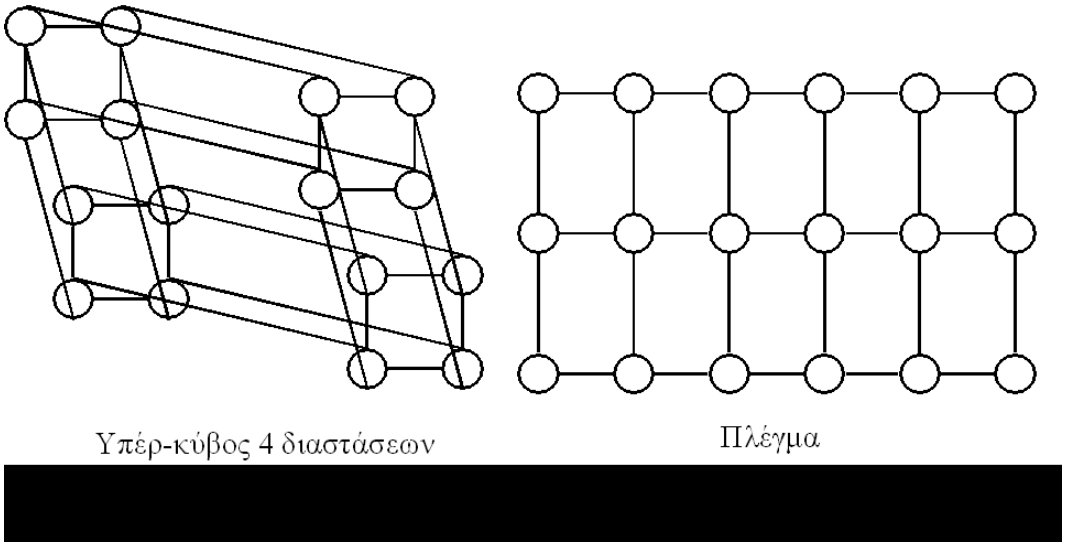
Σχήμα 8.3 : SIMD

MIMD: Τέλος, αυτό είναι το πιο γενικό και ισχυρό μοντέλο παράλληλης επεξεργασίας, όπου ο κάθε επεξεργαστής λειτουργεί με διαφορετικές οδηγίες σε διαφορετικά δεδομένα. Βέβαια αυτό είναι και το πιο πολύπλοκο μοντέλο γιατί οι περισσότεροι αλγόριθμοι πρέπει να επανασχεδιασθούν.



Σχήμα 8.4 : MIMD

Τέλος, στην περίπτωση του MIMD μοντέλου, πρέπει να είναι γνωστή η τοπολογία σύνδεσής των επεξεργαστών έτσι ώστε να αποφεύγονται οι επικοινωνίες μεταξύ των απομακρυσμένων. Με αυτό τον τρόπο μειώνεται και το κόστος μεταφοράς πληροφοριών εάν είναι γνωστό ποιοι είναι οι γειτονικοί κόμβοι. Μερικές από τις πιο συνηθισμένες τοπολογίες φαίνονται στο *Σχήμα 8.5*.



Σχήμα 8.5: Τοπολογίες παράλληλων συστημάτων

8.4 Το Δείπνο των Φιλοσόφων

Οι παράλληλοι αλγόριθμοι και κατ' επέκταση ο παράλληλος προγραμματισμός δεν είναι ούτε μία εύκολη διαδικασία αλλά ούτε και χωρίς προβλήματα. Ένα πολύ ενδιαφέρον πρόβλημα που αναδεικνύει όλα τις πιθανές παγίδες που μπορούν να εμφανισθούν είναι το πρόβλημα «το δείπνο των φιλοσόφων» (Dijkstra): Έστω ότι σε ένα μοναστήρι ζουν πέντε μοναχοί οι οποίοι είναι ταυτόχρονα και φιλόσοφοι. Ο καθένας τους ασχολείται μόνο με τον στοχασμό αλλά πότε πότε πρέπει και να τρώει. Δηλαδή, η ζωή τους είναι ένας ατέλειωτος κύκλος σκέψης και φαγητού. Όσο αναφορά το φαγητό, υπάρχει ένα στρογγυλό τραπέζι όπου στο μέσον υπάρχει μία πιατέλα με μακαρόνια η οποία ανανεώνεται συνέχεια. Στο τραπέζι υπάρχουν πέντε πιάτα και ανάμεσα σε κάθε πιάτο υπάρχει ένα πιρούνι (δηλαδή, σύνολο πάλι πέντε πιρούνια). Ο κάθε μοναχός – φιλόσοφος όταν θέλει να φάει μπαίνει στην αίθουσα, κάθεται σε μία άδεια θέση και χρησιμοποιώντας υποχρεωτικά δύο πιρούνια αυτοσερβίρεται. Τώρα όμως εδώ δημιουργείται το πρόβλημα του ότι πρέπει να είναι ελεύθερα και τα δύο πιρούνια (δεξιά και αριστερά του) γιατί αλλιώς δεν μπορεί να σερβιριστεί. Πάντως, όταν φάει, σηκώνεται και επιστρέφει στο κελί του και στον διαλογισμό του. Το πρόβλημα λοιπόν έγκειται στο να δημιουργηθεί ένα πρωτόκολλο για να μπορούν οι φιλόσοφοι να τρώνε ώστε να μην λιμοκτονήσει ποτέ κανείς τους. Το πρωτόκολλο πρέπει να ικανοποιεί τις ακόλουθες απαιτήσεις:

1) Αμοιβαίος αποκλεισμός: δύο φιλόσοφοι δεν μπορούν να χρησιμοποιούν ταυτόχρονα το ίδιο πιρούνι,
2) Απουσία αδιεξόδου: πρέπει να μπορούν όλοι να φάνε.

Αλγόριθμος 18: Το δείπνο των φιλοσόφων (πρώτη έκδοση)

```
1: Είσοδος / Δεδομένα: Δυαδικός Πίνακας fork5 (0/1), ακέραιος i
2: Για i ← 1 μέχρι 5 παράλληλα
3:        Εκτέλεση διαδικασίας Φιλόσοφος(i)
4: Τέλος Για (i)
5: Διαδικασία Φιλόσοφος(ακέραιος i)
```

6: Επανέλαβε για πάντα
7: Διαλογίσου
8: Περίμενε($fork_i$) /* Περίμενε για πιρούνι */
9: Περίμενε($fork_{(i+1)mod5}$ /* Εξασφάλισε διπλανό πιρούνι */
10: Δείπνησε
11: Απελευθέρωσε($fork_i$)
12: Απελευθέρωσε($fork_{(i+1)mod5}$
13: Τέλος Επανάληψης
14: Τέλος Διαδικασίας «Φιλόσοφος»
15: Τέλος Αλγόριθμου «Το δείπνο των φιλοσόφων (πρώτη έκδοση)»

Ο *Αλγόριθμος 18* υλοποιεί μία πολύ απλή ιδέα και γενικά είναι ικανοποιητικός αλλά μπορεί να οδηγήσει σε αδιέξοδο. Δηλαδή, ο πίνακας fork εξασφαλίζει τον αμοιβαίο αποκλεισμό και η ιδιότητα της ασφάλεια ικανοποιείται γιατί κανείς δεν μπορεί να φάει εάν δεν εξασφαλίσει πρώτα τα δύο πιρούνια. Το πρόβλημα δημιουργείται μόνο όταν και οι πέντε φιλόσοφοι μπουν ταυτόχρονα στο δωμάτιο και καθίσουν όλοι μαζί στο τραπέζι. Τότε, όλοι θα πάρουν από ένα πιρούνι αλλά δεν θα μπορέσουν ποτέ να εξασφαλίσουν το δεύτερο και κατά συνέπεια μάλλον θα οδηγηθούν στην λιμοκτονία. Εδώ αναδεικνύεται το πρόβλημα του τέλειου συγχρονισμού στους παράλληλους αλγόριθμους. Κατά συνέπεια, πρέπει με κάποιο τρόπο να παρακολουθείται το πλήθος των ελεύθερων πιρουνιών πριν ένας φιλόσοφος προσπαθήσει να μπει στο δωμάτιο για να φάει. Για να αντιμετωπισθεί το παραπάνω πρόβλημα, μία λύση είναι το να επιτρέπεται σε ένα φιλόσοφο να παίρνει πιρούνια μόνο εάν και τα δύο πιρούνια είναι ελεύθερα και όχι να παίρνει ένα και να περιμένει το δεύτερο δεσμεύοντας ίσως και έπ' άπειρον το πρώτο. Και αυτό όμως το σενάριο έχει πρόβλημα. Εάν οι φιλόσοφοι 1 και 3 «συνωμοτούσαν» τότε ο φιλόσοφος 2 δεν θα μπορούσε ποτέ να βρει ταυτόχρονα δύο διαθέσιμα πιρούνια. Επομένως και αυτή η λύση οδηγεί σε αδιέξοδο. Πρέπει να επισημανθεί, και μάλιστα με πολλή έμφαση το γεγονός ότι στους παράλληλους αλγόριθμους δεν πρέπει να υπάρχει απολύτως καμία περίπτωση αποκλεισμού ή παραγκωνισμού.

Μία σωστή λύση, θα μπορούσε να βάλλει κάποιους επιπλέον κανόνες ασφάλειας. Δηλαδή, εάν γινόταν έλεγχος σε σχέση με το πόσοι φιλόσοφοι βρίσκονται μέσα στο δωμάτιο. Εάν είναι το πολύ 4 φιλόσοφοι είναι βέβαιο ότι με οποιονδήποτε συνδυασμό, τουλάχιστον ένας θα καταφέρει να φάει. Οπότε ένας σωστός αλγόριθμος θα μπορούσε να είναι όπως αυτός που περιγράφεται στον *Αλγόριθμο 19*.

Αλγόριθμος 19: Το δείπνο των φιλοσόφων (σωστή έκδοση)

1: Είσοδος / Δεδομένα: Δυαδικός Πίνακας $fork_5$ (0/1), ακέραιος i, ακέραιος room=4
2: Για i ← 1 μέχρι 5 παράλληλα
3: Εκτέλεση διαδικασίας Φιλόσοφος(i)
4: Τέλος Για (i)
5: Διαδικασία Φιλόσοφος(ακέραιος i)
6: Επανέλαβε για πάντα
7: Διαλογίσου
8: Περίμενε(room)
9: Περίμενε($fork_i$) /* Περίμενε για πιρούνι */
10: Περίμενε($fork_{(i+1)mod5}$ /* Εξασφάλισε διπλανό πιρούνι */
11: Δείπνησε
12: Απελευθέρωσε($fork_i$)

```
13:             Απελευθέρωσε(fork_(i+1)mod5
14:             Ενημέρωσε(room)
15:     Τέλος Επανάληψης
13: Τέλος Διαδικασίας «Φιλόσοφος»
14: Τέλος Αλγόριθμου «Το δείπνο των φιλοσόφων (σωστή έκδοση)»
```

8.5 Ασκήσεις

1. Περιγράψτε ένα παράλληλο αλγόριθμο ταξινόμησης *N* αντικειμένων. Δικαιολογήστε το μέγιστο αριθμό επεξεργαστών που θα χρειαστούν ώστε να επιτύχει την μέγιστη απόδοση.

2. Περιγράψτε έναν παράλληλο αλγόριθμο για την αναζήτηση στοιχείου σε πίνακα Ν στοιχείων. Να υποτεθεί ότι το πλήθος των διαθέσιμων υπολογιστικών μονάδων είναι *M*, *M*<*N*.

3. Εάν στο πρόβλημα του χρονοπρογραμματισμού εργασιών οι διαθέσιμοι πόροι (πχ μηχανές, υπολογιστές, κλπ) είναι παραπάνω του ενός, έστω δύο, να σχεδιασθεί ένας αλγόριθμος παράλληλης εξυπηρέτησης των εργασιών αυτών.

4. Σχεδιάστε την ακόλουθη παραλλαγή στο πρόβλημα των φιλοσόφων: Ένας φιλόσοφος που θέλει να φάει παίρνει το αριστερό πιρούνι (εάν είναι διαθέσιμο) και στην συνέχεια το δεξί. Εάν το δεξί δεν είναι διαθέσιμο τότε αφήνει και το αριστερό και περιμένει. Σχολιάστε και το κατά πόσο αυτός ο αλγόριθμος είναι σωστός.

5. Σχεδιάστε έναν παράλληλο αλγόριθμο που βρίσκει τον μέσο όρο *N* αριθμών εάν υπάρχουν δυο διαθέσιμοι επεξεργαστές.

9. ΔΕΝΤΡΑ

9.1 Εισαγωγικές Έννοιες - Ορισμοί

Δέντρο είναι ένα σύνολο κόμβων (ίδιου τύπου) και ακμών που συνδέουν τους κόμβους, με βάση κάποια σχέση που δημιουργεί την ιεραρχική δομή των κόμβων. Ένας από τους κόμβους αποτελεί και την ***ρίζα*** του δέντρου. Η δομή του δέντρου αναδρομικά ορίζεται ως εξής:

- Ένας κόμβος αποτελεί από μόνος του ένα δέντρο. Μάλιστα αυτός ο κόμβος αποτελεί και την ρίζα του δέντρου.
- Εάν n είναι ένας κόμβος και $T_1, T_2, \ldots, T_k$ είναι δέντρα με ρίζες $n_1, n_2, \ldots, n_k$ αντίστοιχα, τότε μπορούμε να κατασκευάσουμε ένα νέο δέντρο με κόμβο γονέα τον n των κόμβων $n_1, n_2, \ldots, n_k$. Έτσι, ο κόμβος n αποτελεί την ρίζα του νέου δέντρου, τα $T_1, T_2, \ldots, T_k$ αποτελούν τα ***υποδέντρα*** της ρίζας και οι κόμβοι $n_1, n_2, \ldots, n_k$ τα ***παιδιά*** του κόμβου n.

Κενό είναι το δέντρο που δεν περιέχει καθόλου κόμβους και καμία ακμή.

Εσωτερικοί καλούνται οι κόμβοι στους οποίους μπορούν και να καταλήγουν αλλά και να ξεκινούν ακμές. Στην ρίζα δεν καταλήγουν ακμές. Οι κόμβοι στους οποίους μόνο καταλήγουν ακμές ονομάζονται ***φύλλα*** (ή και τερματικοί κόμβοι). Τα ***υποδέντρα*** σχηματίζονται αρκεί να θεωρήσουμε σαν ρίζα οποιονδήποτε κόμβο του δέντρου.

Εάν $n_1, n_2, \ldots, n_k$ είναι μια σειρά κόμβων ενός δέντρου τέτοια ώστε ο κόμβος n_i να είναι ο γονέας του n_{i+1} για $1 \le i < k$ τότε η ακολουθία αυτή ονομάζεται ***διαδρομή*** (path) από τον n_1 κόμβο προς τον κόμβο n_k. ***Μήκος*** (length) μίας διαδρομής ονομάζεται το σύνολο των περιεχομένων ακμών (που ισούται με το πλήθος των περιεχομένων κόμβων μείον ένα).

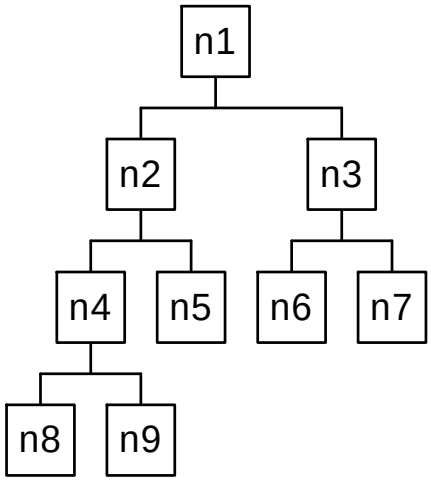
Ύψος (height) κόμβου ενός δέντρου είναι η μεγαλύτερη διαδρομή από τον κόμβο προς κάποιο φύλλο. Ύψος του δέντρου είναι το ύψος της ρίζας του. Τα φύλλα ενός δέντρου έχουν ύψος μηδέν.

Επίπεδο (level) ενός κόμβου είναι το μήκος της μοναδικής διαδρομές από την ρίζα προς αυτόν τον κόμβο. Η ρίζα κάθε δέντρου βρίσκεται στο μηδενικό επίπεδο. Ένα κενό δέντρο έχει ύψος –1.

Βαθμός (degree) ενός κόμβου είναι το πλήθος των παιδιών του.

Ειδικές κατηγορίες δέντρων αποτελούν τα ***δυαδικά*** δέντρα, τα ***τριαδικά*** δέντρα κλπ.

ΠΑΡΑΔΕΙΓΜΑ



Σχήμα 9.1: Δυαδικό δέντρο

Διαδρομή από n_2 προς n_9 είναι η ακολουθία n_2, n_4, n_9.

Το *μήκος* της διαδρομής αυτής είναι 2.

Φύλλα είναι τα n_8, n_9, n_5, n_6, n_7.

Το *ύψος* του δέντρου είναι 3 ενώ ύψος του κόμβου n_2 είναι 2 και του n_9 μηδέν.

Ο *βαθμός* του κόμβου n_9 είναι μηδέν ενώ του κόμβου n_2 είναι 2.

Επίπεδο του κόμβου n_2 είναι ένα και του n_9 είναι τρία.

Δυαδικά Δέντρα είναι τα δέντρα που ο κάθε κόμβος μπορεί να έχει βαθμό το πολύ δύο, δηλαδή ένα αριστερό και ένα δεξιό παιδί. Εάν όλοι οι κόμβοι έχουν από δύο παιδιά πλην ίσως αυτών του τελευταίου επιπέδου αλλά και εδώ όλοι οι κόμβοι να βρίσκονται όσο πιο αριστερά, τότε το δυαδικό δέντρο ονομάζεται ***πλήρες*** (πλήρες είναι το δυαδικό δέντρο του σχήματος 1).

Σε ένα δυαδικό δέντρο, στο επίπεδο μηδέν υπάρχουν $2^0 = 1$κόμβοι. Στο επίπεδο 1 το σύνολο των κόμβων είναι $2^0 + 2^1 = 3$. Επομένως το μέγιστο δυνατό σύνολο των κόμβων ενός πλήρους δυαδικού δέντρου ύψους k είναι $S = 2^0 + 2^1 + \cdots + 2^k = \sum_{p=1}^{k} 2^p = 2^{k+1} - 1$. Επομένως το δέντρο του σχήματος 1 θα μπορούσε να έχει το πολύ κόμβους $2^0 + 2^1 + 2^2 + 2^3 = 2^{3+1} - 1 = 15$ (ύψος 3).

Πρόταση: Για ένα δυαδικό δέντρο n κόμβων και ύψους k, ισχύει $\log(n+1) - 1 \le k \le \log(n)$**.**

Απόδειξη:

Ο ελάχιστος αριθμός κόμβων σε ένα πλήρες δυαδικό δέντρο ύψους k είναι $2^0 + 2^1 + \cdots + 2^{k-1} + 1$εάν υποθέσουμε ότι στη χειρότερη περίπτωση έχει συμπληρωμένους όλους τους κόμβους μέχρι το επίπεδο k-1 και έχει μόνο ένα κόμβο στο επίπεδο k.
Τότε το πλήθος αυτό είναι:

$$(2^k - 1) + 1 = 2^k \Rightarrow n \ge 2^k \Leftrightarrow \log(n) \ge \log(2^k) \Leftrightarrow \log(n) \ge k .$$

Επίσης, αφού το μέγιστο πλήθος κόμβων αυτού του δέντρου είναι:

$2^{k+1}-1 \Rightarrow n \le 2^{k+1}-1 \Leftrightarrow n+1 \le 2^{k+1} \Leftrightarrow \log(n+1) \le \log(2^{k+1}) \Leftrightarrow$
$\Leftrightarrow \log(n+1) \le k+1 \Leftrightarrow \log(n+1)-1 \le k$.

9.2 Διαπεράσεις Δυαδικού Δέντρου

Διαπέραση δέντρου είναι η διαδικασία επίσκεψης των κόμβων του. Εάν θεωρήσουμε τις διαδικασίες:

1. Επίσκεψη ρίζας δέντρου,
2. Επίσκεψη αριστερού υποδέντρου, και
3. Επίσκεψη δεξιού υποδέντρου

τότε διαπέραση σημαίνει την επίσκεψη κάθε κόμβου μία φορά κατά μία συγκεκριμένη σειρά. Έτσι διακρίνουμε τους παρακάτω τρόπους διαπέρασης:

- ✓ ***Προδιατεταγμένη*** (preorder) όπου η σειρά είναι 1-2-3,
- ✓ ***Ενδοδιατεταγμένη*** (inorder) όπου η σειρά είναι 2-1-3, και τέλος
- ✓ ***Μεταδιατεταγμένη*** (postorder) όπου η σειρά είναι 2-3-1.

9.3 Υλοποίηση Δυαδικού Δέντρου με Πίνακα

Έστω *T(n)* ο πίνακας που θα αναπαραστήσει ένα δυαδικό δέντρο ύψους k και πλήθους κόμβων n. Κατ' αρχήν πρέπει $n = 2^{k+1}-1$ δηλαδή, το πλήθος των στοιχείων του πίνακα να είναι όσο και το μέγιστο πλήθος των κόμβων του δέντρου.
Στο πρώτο στοιχείο του πίνακα τοποθετείται η ρίζα του δέντρου. Στη συνέχεια και στις επόμενες θέσεις του πίνακα τοποθετούνται με την σειρά πρώτα το αριστερό παιδί και μετά το δεξιό. Για παράδειγμα, οι κόμβοι του δέντρου του σχήματος 1 θα τοποθετούταν ως εξής: $[n_{,}, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9]$.

Τώρα, το κυρίαρχο θέμα που προκύπτει για το κάθε στοιχείο με δείκτη i του πίνακα T είναι ποιο είναι το αριστερό και ποιο το δεξί παιδί του καθώς και ποιος είναι ο γονέας του. Παρατηρούμε τα ακόλουθα:

- ✓ Αν $i = 1$ τότε είναι η ρίζα του δέντρου.
- ✓ Αν $i > 1$ τότε ο γονέας του είναι το στοιχείο με δείκτη το ακέραιο πηλίκο $\frac{i}{2}$.
- ✓ Αν $2i > n$ τότε το στοιχείο αποτελεί φύλλο του δέντρου (δεν υπάρχουν παιδιά).
- ✓ Αν $2i \le n$ τότε αριστερό παιδί του είναι το $2i$ και δεξί του το $2i+1$.

Αναδρομικός Αλγόριθμος Προδιατεταγμένης Διαπέρασης Δυαδικού Δέντρου
(Τ=Δέντρο, n=πλήθος κόμβων)

ΠΔΔΔ(r)
root=r

```
Εάν root<=n τότε
        Επίσκεψη(T(root))
        left= 2r /* Αριστερό παιδί */
        ΠΔΔΔ(left)

        right=2r+1 /* Δεξιό Παιδί */
        ΠΔΔΔ(right)
Τέλος Εάν
Τέλος ΠΔΔΔ
```

Υλοποίηση σε Pascal

```
program tree1;
const n=9;
var t:array[1..n] of char; {Δυαδικό Δέντρο}
   i:integer;

procedure pddd(r:integer);
{Προδιατεταγμένη Διαπέραση}
var root,left,right:integer;
begin
  root:=r;
  if root<=n then
  begin
      write(t[root],' ');
      left:=2*r; pddd(left);
      right:=2*r+1; pddd(right);
  end;
end; {pddd}

begin {main}
  for i:=1 to n do
  begin
      write(i:5,'-->');
      readln(t[i]);
      writeln;
  end;
  writeln;writeln;
  pddd(1);
end.
```

Υλοποίηση σε C

```
#include <stdio.h>
#define N 10

void pddd(int r);

int T[N];

main()
{
  int i=1;

  do {
      printf("\n\nInput node-->");
      scanf("%d",&T[i]);
      i++;
  } while (i<N);

  printf("\n\n\nPreorder: ");
  pddd(1);
  printf("\n\n\n");
}

void pddd(int r)
{
  int root=r,left,right,node;

  if (root<N) {
      node=T[root];
      printf(" %d ",node);
      left=2*root;
      pddd(left);
      right=2*root+1;
      pddd(right);
  }
}
```

9.4 Δυαδικά Δέντρα Αναζήτησης

Δυαδικό Δέντρο Αναζήτησης (binary search trees) είναι ένα δυαδικό δέντρο με κόμβους διατεταγμένους με τέτοιο τρόπο ώστε η τιμή του στοιχείου του κάθε κόμβου να είναι μεγαλύτερη από τις τιμές όλων των στοιχείων των κόμβων του αριστερού του υποδέντρου και μικρότερη από αυτές των στοιχείων του δεξιού του υποδέντρου. Τέτοιο δέντρο αποτελεί το δέντρο του σχήματος 1.

Βασικό πλεονέκτημα αυτής της δομής είναι ότι συνδυάζει γρήγορες αναζητήσεις ενώ παράλληλα επιτρέπει την εύκολη εισαγωγή και διαγραφή στοιχείων.

⇒ ***Αλγόριθμος Αναζήτησης***:
- Σύγκριση του στοιχείου της ρίζας με το υπό αναζήτησης στοιχείο. Εάν είναι ίσα τότε ο αλγόριθμος τελειώνει (επιτυχής αναζήτηση).
  - Εάν το υπό αναζήτηση στοιχείο είναι μεγαλύτερο τότε συνεχίζουμε με το δεξιό υποδέντρο αλλιώς με το αριστερό (θέτοντας σαν ρίζα την ρίζα του αριστερού ή δεξιού υποδέντρου αντίστοιχα).
- Το προηγούμενο βήμα επαναλαμβάνεται μέχρι ή να βρεθεί το στοιχείο ή την συνάντηση κενού υποδέντρου (ανεπιτυχής αναζήτηση).

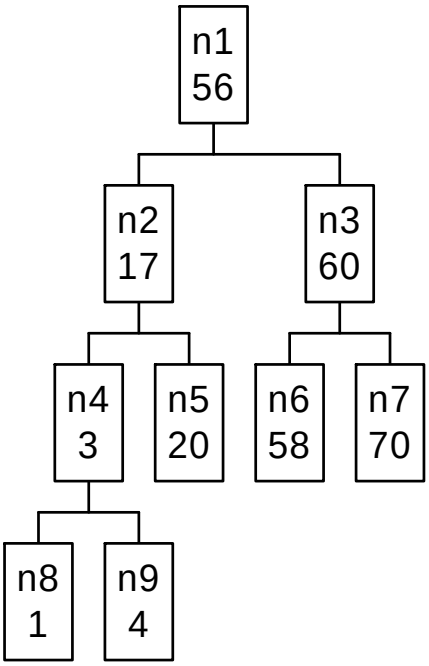
⇒ ***Αλγόριθμος Εισαγωγής***:
- Σύγκριση του στοιχείου της ρίζας με το υπό εισαγωγή στοιχείο. Εάν είναι ίσα τότε ο αλγόριθμος τελειώνει (δεν υπάρχει λόγος εισαγωγής).
  - Εάν το υπό εισαγωγή στοιχείο είναι μεγαλύτερο τότε συνεχίζουμε με το δεξιό υποδέντρο αλλιώς με το αριστερό (θέτοντας σαν ρίζα την ρίζα του αριστερού ή δεξιού υποδέντρου αντίστοιχα).
- Το προηγούμενο βήμα επαναλαμβάνεται μέχρι την συνάντηση κενού υποδέντρου και εισάγεται σ' αυτό ακριβώς το σημείο, δηλαδή σαν ρίζα αυτού του κενού υποδέντρου.

⇒ Ο αλγόριθμος εισαγωγής δημιουργεί το δυαδικό δέντρο αναζήτησης το οποίο εν προσπελασθεί με την ενδοδιατεταγμένη διάταξη εμφανίζει ταξινομημένα τα περιεχόμενα – κόμβους του. Αυτή η μέθοδος ταξινόμησης ονομάζεται **heap sort** και η πολυπλοκότητά της είναι $O(N \log N)$, δηλαδή όσο και της quick sort.

⇒ ***Αλγόριθμος Διαγραφής***:
- Κατ' αρχήν εκτελείται ο αλγόριθμος αναζήτησης και στην περίπτωση μη εύρεσης του προς διαγραφή στοιχείου ο αλγόριθμος τελειώνει. Εάν όμως βρεθεί στον κόμβο n_i τότε:
- Εάν ο n_i δεν έχει παιδιά απλώς διαγράφεται και την θέση του καταλαμβάνει ένα κενό υποδέντρο
- Εάν ο n_i έχει μόνο ένα παιδί, τότε ο n_i διαγράφεται και την θέση του καταλαμβάνει το μοναδικό παιδί του.
- Εάν ο n_i έχει δύο παιδιά, τότε:
  - Σύμφωνα με την ενδοδιατεταγμένη διάταξη (in order) βρίσκεται ο αμέσως επόμενος κόμβος του n_i έστω n_j και διαγράφεται ο n_i και τη θέση του

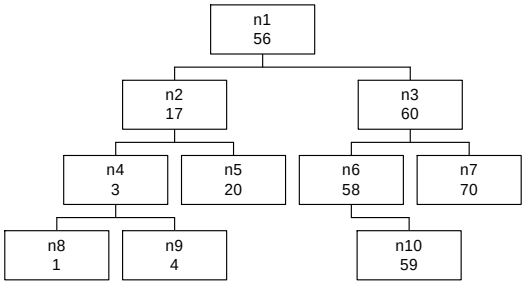
καταλαμβάνει ο n_j. (Φυσικά διαγράφεται και ο nj από την παλιά του θέση).

ΠΑΡΑΔΕΙΓΜΑ

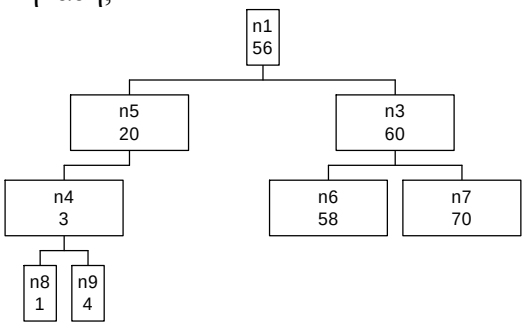


Αναζήτηση του 4: n_1, n_2, n_4, n_9.

Εισαγωγή του 59: n_1, n_3, n_6 και στη συνέχεια εισαγωγή σαν δεξιό υποδέντρο του n_6. Δηλαδή,



Διαγραφή του 17: n_1, n_2, n_5 και στη συνέχεια διαγραφή του n_2 και τοποθέτηση του n_5 στη θέση του. Δηλαδή,



⇒ **<u>ΠΟΛΥΠΛΟΚΟΤΗΤΑ</u>** Η πολυπλοκότητα και των τριών αλγορίθμων ουσιαστικά εξαρτάται από την πολυπλοκότητα της αναζήτησης. Εάν υποτεθεί ότι το ύψος του δέντρου είναι k τότε ισχύει ότι $\log(n+1)-1 \le k \le \log(n)$. Επομένως και ο αριθμός αναζητήσεων $C(n)$ περιορίζεται από την σχέση $\log(n+1)-1 \le C(n) \le \log(n)$. Οπότε η πολυπλοκότητα εκφράζεται με τον τύπο $\log(n) \le O(n) \le \log(n)$ και με μέση συνάρτηση πολυπλοκότητας την $O(n) = \log(n)$.

9.5 Ασκήσεις

1. Να γίνουν αλγόριθμοι εισαγωγής, διαγραφής και αναζήτησης σε δυαδικά δέντρα με χρήση πίνακα όπως έχει περιγραφεί.

2. Εάν υποθέσουμε ότι υλοποιούμε ένα δυαδικό δέντρο με ένα πίνακα εγγραφών. Η εγγραφή περιέχει τουλάχιστον 3 πεδία τέτοια ώστε σε κάθε εγγραφή το προτελευταίο πεδίο να δείχνει το αριστερό παιδί του κόμβου ενώ το τελευταίο πεδίο να δείχνει τον δεξιό αδερφό του αριστερού παιδιού του κόμβου. Να γίνει αλγόριθμος διαπέρασης (προδιατεταγμένης) του δέντρου που υλοποιείται όπως παραπάνω.

3. Με την βοήθεια του αλγόριθμου εισαγωγής να σχηματίσετε δυαδικό δέντρο αναζήτησης με στοιχεία την ακόλουθη λίστα: $\Ε, \Η, \Δ, \Ι, \Π, \Α, \Φ, \Γ, \Ν, \Ρ, \Σ, \Β$. Στη συνέχεια να διαγραφεί το στοιχείο Π και να εισαχθεί το Λ.

4. Να υλοποιηθούν οι αλγόριθμοι Αναζήτησης, Εισαγωγής και Διαγραφής σε Pascal ή C με την χρήση δεικτών και πίνακα

10. ΓΡΑΦΟΙ (Graphs)

10.1 Εισαγωγικές Έννοιες - Ορισμοί

Πρώτος που χρησιμοποίησε σε θεωρητικό επίπεδο γράφους ήταν ο Euler το 1736 στην δημοσίευση με τίτλο "Solutio Problematis ad Geometrian Situs Pertinentis". Προσπάθησε, στην αρχή από απλή μαθηματική περιέργεια, να λύσει το πρόβλημα: «Πως μπορεί κάποιος να διασχίσει τις επτά γέφυρες του ποταμού Konigsberg σε μία βόλτα αλλά να περάσει μόνο και μία μόνο φορά από την καθεμία».

Ένας ***γράφος (graph)*** $G=(V,E)$ αποτελείται από δύο σύνολα: ένα πεπερασμένο σύνολο V από στοιχεία που ονομάζονται κορυφές ή κόμβοι (nodes ή vertices) και ένα πεπερασμένο επίσης σύνολο E ακμών. Εάν οι ακμές ορίζονται σαν διατεταγμένα ζευγάρια κορυφών τότε ο γράφος ονομάζεται *προσανατολισμένος (directed ή oriented)* ενώ σε αντίθετη περίπτωση ονομάζεται *μη προσανατολισμένος (undirected ή nonoriented, Σχήμα 10.1)*. Στις προκείμενες σημειώσεις γίνεται λόγος για μη προσανατολισμένους γράφους.

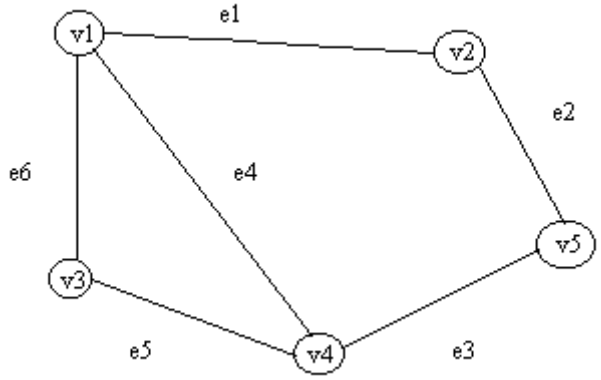
Συνήθως οι κορυφές συμβολίζονται σαν $V=\{v_1,v_2,\ldots,v_n\}$ και οι ακμές σαν $E=\{e_1,e_2,\ldots,e_m\}$. Εάν οι κορυφές v_i και v_j συσχετίζονται με την ακμή e_l τότε ονομάζονται καταληκτικές κορυφές αυτής της ακμής και συμβολίζεται σαν $e_l=(v_i,v_j)$. Όλες οι ακμές που έχουν τις ίδιες καταληκτικές κορυφές ονομάζονται ***παράλληλες ακμές***. Επιπλέον, εάν $e_l=(v_i,v_i)$ τότε η ακμή e_l αποτελεί ένα ***αυτο-βρόγχο*** της κορυφής v_i.Ένας γράφος που δεν έχει παράλληλες ακμές αλλά ούτε και αυτο-βρόγχους ονομάζεται ***απλός γράφος***. Ένας γράφος χωρίς ακμές ονομάζεται άδειος γράφος ενώ χωρίς κορυφές (οπότε και χωρίς ακμές) ονομάζεται ***κενός γράφος***.

$\Rightarrow$ Ο αριθμός των ακμών που έχει μία κορυφή ονομάζεται ***βαθμός της κορυφής*** και συμβολίζεται σαν $d(v_i)$. Εάν ισχύει $d(v_i)=0$ τότε αυτή η κορυφή ονομάζεται ***απομονωμένη κορυφή***.

ΠΑΡΑΔΕΙΓΜΑ

Στο γράφο του παραδείγματος ισχύουν τα ακόλουθα:

1. Είναι ένας απλός γράφος,
2. $V=\{v_1,v_2,v_3,v_4,v_5\}$,
3. $E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$,
4. $e_1=(v_1,v_2),e_2=(v_2,v_5),e_3=(v_5,v_4),e_4=(v_1,v_4),e_5=(v_3,v_4),e_6=(v_1,v_3)$,
5. $d(v_1)=3,d(v_2)=2,d(v_3)=2,d(v_4)=3,d(v_5)=2$.



Σχήμα 10.1: Γράφος 5 κορυφών

Περίπατος (walk) σε ένα γράφο *G* θεωρείται κάθε ακολουθία κορυφών $v_0, v_1, \ldots, v_k$ όπου το κάθε ζευγάρι κορυφών $(v_{i-1}, v_i), 1 \le i \le k,$ αποτελεί ακμή του γράφου *G*. Συνήθως ένα περίπατος συμβολίζεται με τις καταληκτικές κορυφές του, πχ $v_0 - v_k$. Κάθε κορυφή, σε ένα περίπατο μπορεί να εμφανίζεται περισσότερες από μία φορές. Ένας περίπατος ονομάζεται ***ανοικτός*** εάν οι καταληκτικές κορυφές του είναι διαφορετικές αλλιώς ονομάζεται ***κλειστός***. Στο γράφο του σχήματος 1 ένας ανοικτός περίπατος είναι ο $v_2, v_5, v_4, v_1, v_2, v_5$ ενώ ένας κλειστός θα μπορούσε να είναι ο v_1, v_4, v_3, v_1.

Ένας περίπατος ονομάζεται ***πέρασμα*** (trail) εάν όλες οι κορυφές του είναι διαφορετικές μεταξύ τους πχ. v_1, v_4, v_3. Ένα ανοικτό πέρασμα ονομάζεται ***μονοπάτι*** (path) ενώ ένα κλειστό, ***κύκλωμα*** (circuit).

Ο αριθμός των ακμών σε ένα μονοπάτι ορίζουν το ***μήκος*** του μονοπατιού. ***Απόσταση*** μεταξύ δύο κορυφών v και u, συμβολίζεται σαν $d(v,u)$ και ορίζεται σαν το μήκος του συντομότερου μονοπατιού μεταξύ των δύο αυτών κορυφών. Εάν δεν υπάρχει τέτοιο μονοπάτι, αυτή η απόσταση ορίζεται σαν άπειρη. ***Διάμετρος*** ενός γράφου *G*, ονομάζεται η μέγιστη απόσταση μεταξύ δύο οποιονδήποτε κορυφών αυτού του γράφου και συμβολίζεται σαν $diam(G)$.

Ισχύουν τα ακόλουθα:

- Σε ένα μονοπάτι ο βαθμός κάθε εσωτερικής κορυφής είναι 2 ενώ των καταληκτικών είναι 1,
- Σε ένα μονοπάτι ο αριθμός των κορυφών είναι κατά ένα μεγαλύτερος από το μήκος του μονοπατιού.

Η ***αναπαράσταση*** ενός γράφου μπορεί να επιτευχθεί με τον ονομαζόμενο ***πίνακα γειτνίασης*** (adjacency matrix) *A*. Ο πίνακας αποτελείται από $nXn = n^2$ στοιχεία, εάν υποθέσουμε ότι έχουμε ένα γράφο *n* κορυφών. Το στοιχείο $A[i, j] = 1$, εάν υπάρχει η ακμή (v_i, v_j) ενώ αλλιώς είναι 0. Προφανώς αυτός ο πίνακας είναι συμμετρικός. Το μόνο μειονέκτημα αυτής της αναπαράστασης είναι ότι απαιτεί $\Omega(n^2)$ αποθηκευτικό

χώρο ενώ συνήθως ο αριθμός των ακμών είναι μικρότερος του n^2, και φυσικά η προσπέλαση του πίνακα απαιτεί $O(n^2)$ χρόνο. Για το γράφο του σχήματος 1, ο πίνακας γειτνίασης είναι ο ακόλουθος:

Επίσης ένας γράφος μπορεί να αναπαρασταθεί και με την λίστα γειτνίασης όπου για τον γράφο του σχήματος 1, η λίστα γειτνίασης είναι:

1: 2,3,4
2: 1,5
3: 1,4
4: 1,3,5
5: 2,4

Εφαρμογές Γράφων: Αν και οι εφαρμογές της θεωρίας των γράφων είναι πάρα πολλές αναφέρονται μερικές από τις πιο ενδιαφέρουσες: Δίκτυα (υπολογιστών, οδικά, ηλεκτρικών κυκλωμάτων, τηλεπικοινωνιών,…), Μηχανική, Ηλεκτρισμός, Χημεία, Βιολογία,…

10.2 Αλγόριθμος Ψαξίματος Γράφου Κατά Βάθος (depth-first-search ή dfs)

Αυτός ο αλγόριθμος φροντίζει να επισκεφτεί όλους τους κόμβους ενός γράφου μετακινούμενος πάντα προς κάποιο γειτονικό κόμβο, εάν τέτοιος υπάρχει και δεν τον έχει ήδη ξαναεπισκεφθεί. Εάν δεν υπάρχει τέτοιος κόμβος ο αλγόριθμος επιστρέφει προς τα πίσω και εκτελείται αναδρομικά. Χρησιμοποιεί δε ένα μονοδιάστατο πίνακα S με πλήθος στοιχείων όσο και το πλήθος των κορυφών του γράφου. Ο πίνακας αυτός αρχικοποιείται θέτοντας μηδέν όλα τα στοιχεία του μηδέν και μετατρέπονται σε ένα αν κατά την διάρκεια εκτέλεσης του αλγόριθμου πραγματοποιηθεί επίσκεψη της αντίστοιχης κορυφής του γράφου. Ο αλγόριθμος χρησιμοποιεί τον πίνακα γειτνίασης του γράφου.

Δεδομένα: Ο πίνακας γειτνίασης A[ν X ν], ν το πλήθος των κορυφών, και S[ν] ο πίνακας που θα κρατάει την πληροφορία αναφορικά με ποιες κορυφές έχει ήδη επισκεφτεί ο αλγόριθμος. Αρχικά S[i]=0, $\forall i \in \{1,2,\ldots,\nu\}$

Αλγόριθμος 20: Αλγόριθμος Depth First Search

```
1: Διαδικασία DFS(k)
2: Αρχή
3:      Για i=k έως ν
4:      Εάν S[i]=0
5:              Τότε
6:                      S[k]=1
7:                      Εμφάνισε k
8:      Τέλος Εάν
9:      Για j=i+1 έως ν
10:             Εάν A[i,j]=1
11:                     Τότε DFS(j)
```

12: ΤέλοςΕάν
13: ΤέλοςΓια (j=…)
14: Τέλος Για (i=…)
15: Τέλος Διαδικασίας (DFS)

10.3 Ο Αλγόριθμος του DIJKSTRA (Οικονομικότεροι δρόμοι)

Σε δεδομένο γράφο $G=(V,E)$ ένα από τα πιο σημαντικά προβλήματα αποτελεί τον υπολογισμό του κόστους των διαδρομών από κάποιο κόμβο (πηγαίο / source) προς όλους τους άλλους κόμβους του γράφου. Το κόστος του κάθε μονοπατιού είναι το άθροισμα των αντιστοίχων κοστών των συμμετεχόντων ακμών. Για την επίλυση αυτού του προβλήματος, ο προτεινόμενος αλγόριθμος ανήκει στην κατηγορία των άπληστων (greedy) αλγορίθμων και συχνά αναφέρεται σαν ο αλγόριθμος του Dijkstra (1959).

Η γενική ιδέα του αλγόριθμου είναι η εξής: Ο αλγόριθμος δημιουργεί ένα σύνολο *S* κόμβων των οποίων τα μονοπάτια από τον πηγαίο κόμβο είναι ήδη γνωστά. Αρχικά το σύνολο *S* περιέχει μόνο τον πηγαίο κόμβο. Σε κάθε επανάληψη του αλγόριθμου προστίθεται ένας κόμβος του οποίου το μονοπάτι από τον πηγαίο έχει όσο πιο μικρό κόστος γίνεται. Επίσης, σε κάθε βήμα ο αλγόριθμος ενημερώνει ένα πίνακα *D* με το κόστος της οικονομικότερης διαδρομής για κάθε κόμβο. Υποτίθεται ότι υπάρχει ένας πίνακας δύο διαστάσεων *C[n,n]* εάν *n* είναι το πλήθος των κόμβων του γράφου όπου το στοιχείο *C[i,j]* δίνει το κόστος της ακμής e_{ij}. Σε περίπτωση που η ακμή δεν υπάρχει τότε $C[i,j]=\infty$ ή ένας πολύ μεγάλος αριθμός ανάλογα με την υλοποίηση του αλγόριθμου.

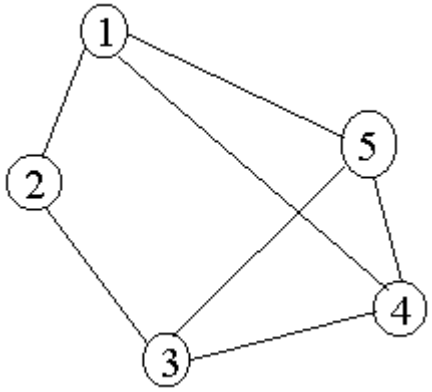
Αλγόριθμος 21: Αλγόριθμος Dijkstra – Οικονομικότεροι δρόμοι

1: Έστω Γράφος n κόμβων
2: Έστω S={1} /Ο εναρκτήριος κόμβος
3: Για i=2 μέχρι n
4: $D_i=C_{1i}$ / Αρχικοποίηση του D
5: Τέλος Για (i)
6: Για i=1 μέχρι n-1
7: Επιλογή κόμβου από το σύνολο V-S τέτοιον ώστε το αντίστοιχο D_w να είναι ελάχιστο
8: Πρόσθεσε τον κόμβο w στο σύνολο S, δηλαδή S=S+{w}
9: Για κάθε κόμβο v του συνόλου V-S
10: D_v=ελάχιστο(D_v, D_w+C_{wv})
11: Τέλος Για
12: Τέλος του αλγόριθμου «Αλγόριθμος Dijkstra – Οικονομικότεροι δρόμοι»

Παρατήρηση: Εάν ζητείται ο οικονομικότερος δρόμος προς κάποιο συγκεκριμένο κόμβο ο αλγόριθμος θα μπορούσε να διακόπτεται μόλις φθάσει σ' αυτόν τον κόμβο.

⇒ **Παράδειγμα:**
Έστω ο γράφος του *Σχήματος 10.2* και έστω ότι

C[1,2]=10, c[1,4]=30, C[1,5]=100
C[2,3]=50
C[3,4]=20
C[4,3]=20, C[4,5]=60



Σχήμα 10.2: Αλγόριθμος Dijkstra

Κατά την εφαρμογή του αλγόριθμου, αρχικά είναι S={1}, D[2]=10, D[3]=∞, D[4]=30 και D[5]=100. Μετά την πρώτη επανάληψη επιλέγεται ο κόμβος 2 επειδή παρουσιάζει την ελάχιστη τιμή στον D πίνακα. Επίσης το D[3] γίνεται 60 επειδή D[3]= ελάχιστο(∞,10+50). Τα D[4] και D[5] δεν αλλάζουν διότι δεν επικοινωνούν άμεσα με τον δεύτερο κόμβο οπότε διατηρείται η αρχική τους τιμή. Η εξέλιξη των τιμών του πίνακα D είναι ως εξής:

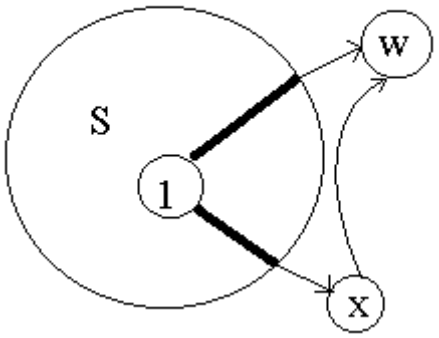
Επαναλήψεις	**S**	**W**	**D[2]**	**D[3]**	**D[4]**	**D[5]**
Αρχική	1		10	∞	30	100
1	1,2	2	10	60	30	100
2	1,2,4	4	10	50	30	90
3	1,2,4,3	3	10	50	30	60
4	1,2,4,3,5	5	10	50	30	60

Στην περίπτωση που θέλουμε να κρατάμε και τις διαδρομές αυτό μπορεί να συμβεί με την χρήση ενός ακόμη πίνακα P ο οποίος αρχικά θα περιέχει σε όλα τα στοιχεία του τον πηγαίο κόμβο και μετά από την γραμμή 9 του *Αλγόριθμου 21* θα τοποθετεί P[v]=w. Αυτό σημαίνει ότι από κάθε στοιχείο του πίνακα θα φαίνεται ο αμέσως προηγούμενος κόμβος που θα οδηγήσει προς το πηγαίο. Οπότε εάν θέλουμε να υπολογίσουμε την διαδρομή 1-5 αρκεί να διαβαστεί ο πίνακας από το στοιχείο 5 μέχρι να μας οδηγήσει στον κόμβο 1. Για το γράφο του παραδείγματος είναι P[1]=1, P[2]=1, P[3]=4, P[4]=1 και P[5]=3. Οπότε η διαδρομή 1-5 δίνεται εάν διαβασθεί ο πίνακας από το πέμπτο στοιχείο και μέχρι να οδηγήσει στο πρώτο, δηλαδή 5,3,4,1 οπότε και η διαδρομή είναι 1,4,3,5

⇒ **<u>Πολυπλοκότητα:</u>** Ο αλγόριθμος επαναλαμβάνει *n-1* φορές τους απαιτούμενους υπολογισμούς (γραμμή 3) και για κάθε επανάληψη η εσωτερική επανάληψη εκτελείται το πολύ *n-2* φορές. Οπότε έχουμε ένα μέγιστο *(n-1)(n-2)* επαναλήψεων επομένως η πολυπλοκότητα του αλγόριθμου είναι της τάξης του $O(n^2)$.

⇒ **<u>Απόδειξη ορθότητας του αλγόριθμου:</u>**

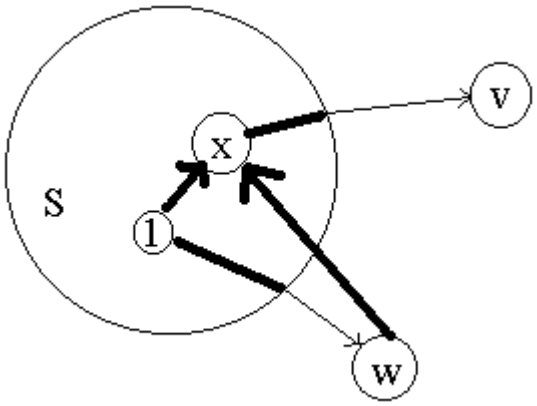
Περίπτωση 1: Εάν υποθέσουμε ότι σε κάποια φάση έχει επιλεγεί ο κόμβος w και έχει ήδη προστεθεί στο σύνολο S ενώ ταυτόχρονα υπάρχει ένας άλλος κόμβος $x \notin S$ όπου η διαδρομή S-x-w είναι οικονομικότερη όπως φαίνεται στο *Σχήμα 10.3*.



Σχήμα 10.3: Ορθότητα αλγόριθμου Dijkstra, 1η περίπτωση

Αλλά εάν αυτό ίσχυε τότε η διαδρομή S-x θα ήταν οικονομικότερη από την διαδρομή S-w οπότε και θα έπρεπε να είχε επιλεγεί ο κόμβος x αντί του w οπότε και καταλήγουμε σε άτοπη υπόθεση.

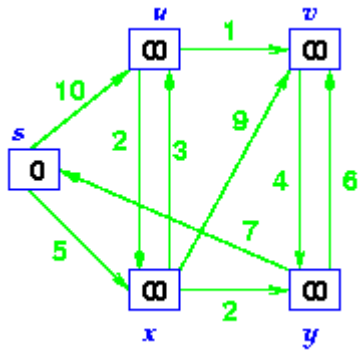
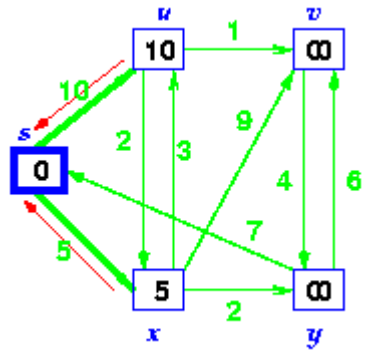
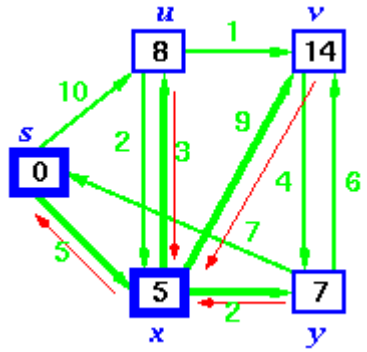
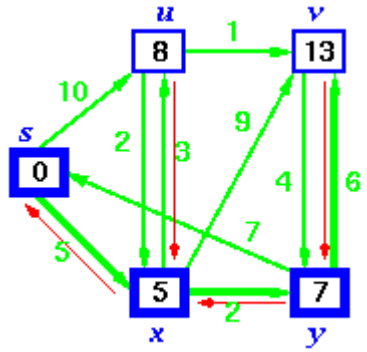
Περίπτωση 2: Εάν υποθέσουμε τώρα ότι πάλι έχει επιλεγεί ο κόμβος w και έχει ήδη προστεθεί στο σύνολο S. Υπάρχει το ενδεχόμενο να υπάρχει οικονομικότερος δρόμος

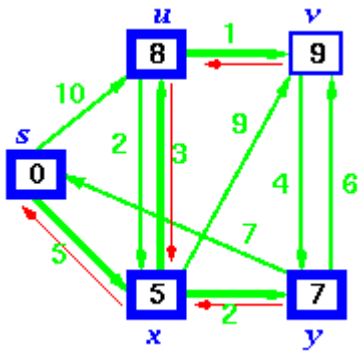
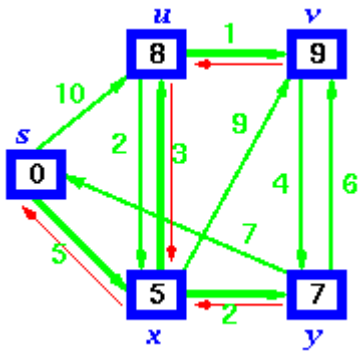


Σχήμα 10.4: Ορθότητα αλγόριθμου Dijkstra, 2η περίπτωση

προς κάποιο κόμβο v ο οποίος διέρχεται μεν από τον w αλλά διέρχεται και από κάποιον κόμβο x που ήδη ανήκει στο S; Αυτό όμως δεν γίνεται γιατί μιάς και ο κόμβος x έχει ήδη επιλεγεί νωρίτερα από τον w οπότε η διαδρομή S-x-v έχει ήδη ελεγχθεί και ενημερωθεί ο πίνακας D.
Οπότε από τα προηγούμενα συνάγεται ότι ο προτεινόμενος αλγόριθμος είναι σωστός.

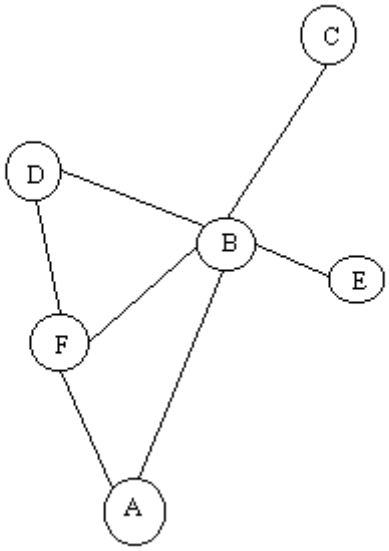
⇒ **Παράδειγμα 2:**

	Αρχικός Γράφος. Το κόστος είναι σε όλους τους κόμβους άπειρο εκτός από τον κόμβο αφετηρία
	Αρχικοποίηση
	Επιλογή του πρώτου κόμβου (x) και ενημέρωση του πίνακα κόστους
	Επιλογή του δεύτερου κόμβου (y) και ενημέρωση του πίνακα κόστους

	Επιλογή του τρίτου κόμβου (u) και ενημέρωση του πίνακα κόστους
	Επιλογή του τελευταίου κόμβου (v) και ενημέρωση του πίνακα κόστους. Τελική μορφή του πίνακα κόστους D[s] = 0 D[u] = 8 D[v] = 9 D[x] = 5 D[y] = 7

10.4 Ασκήσεις

1. Ας υποθέσουμε ότι το οδικό δίκτυο που συνδέει τις πόλεις Αθήνα (A), Λάρισα (B), Θεσσαλονίκη (C), Τρίκαλα (D), Βόλο (E), Καρδίτσα (F) είναι το ακόλουθο:



Να δημιουργηθεί ο πίνακας γειτνίασης καθώς επίσης και τα μονοπάτια Αθήνα-Θεσσαλονίκη, Βόλος-Καρδίτσα και Τρίκαλα-Καρδίτσα. Επίσης να υπολογισθούν η διάμετρος του παραπάνω γράφου όπως και ο βαθμός του κάθε κόμβου (κορυφής).

2. Εάν στο παραπάνω γράφο οι αποστάσεις μεταξύ των πόλεων / κόμβων είναι AB=360, AF=310, BF=55, BD=60, BE=65 και BC=150, να δημιουργηθεί αντίστοιχος πίνακας με τον γειτνίασης όπου οι ακμές θα δείχνουν τις πραγματικές αποστάσεις. Στη συνέχεια να γίνει αλγόριθμος που να υπολογίζει όλα τα πιθανά μονοπάτια / διαδρομές μεταξύ δύο οποιονδήποτε πόλεων μαζί με τις αντίστοιχες αποστάσεις.

3. Να υλοποιηθεί ο αλγόριθμος DFS σε κάποια γλώσσα προγραμματισμού και να υπολογισθεί και η πολυπλοκότητά του.

Πρόγραμμα 16: Ο αλγόριθμος Dijkstra (Pascal)

```
program dijkstra_algorithm;
uses crt;
const nn=20; {μέγιστο πλήθος κόμβων}
var
   c:array[1..nn,1..nn] of integer; {Πίνακας Κόστους}
   d:array[1..nn] of integer;  {Πίνακας ελάχιστου κόστους}
   s:array[1..nn] of integer; {Πίνακας όπου προστίθενται οι κόμβοι}
   pr:array[1..nn] of integer; {Πίνακας μονοπατιού}
   n : integer; {Πραγματικός αριθμός κόμβων}
   i : integer;

procedure init;
var i,j:integer;
   gr:text; {αρχείο που περιέχει τον αριθμό κόμβων και τον πίνακα κόστους}
begin
   assign(gr,'graph.dat');
   reset(gr);
   readln(gr,n);

   for i:=1 to n do
   begin
     for j:=1 to n
       do begin read(gr,c[i,j]);write(c[i,j]:7);end ;
     readln(gr); writeln;
   end;
   close(gr);
end; {init}

procedure da;
{Ο αλγόριθμος}
var i,ii,j,k,min,th:integer;
   l:integer;
   t:array[2..nn] of integer; {Πίνακας που κρατάει την πληροφορία σε σχέση ποιοι
κόμβοι έχουν απομείνει. Σε περίπτωση που t[i]}=-1 ο κόμβος έχει ήδη προστεθεί στο
S}

begin
  {Αρχικοποιήσεις}
```

```
  s[1]:=1; d[1]:=0; t[1]:=-1;
  for i:=2 to n
    do d[i]:=c[1,i];
  for i:=1 to n
    do pr[i]:=1;
  for i:=2 to n
    do t[i]:=i;

  k:=1;
  for l:=2 to n do
  begin
    k:=k+1;
    {Επιλογή εναπομεινάντων κόμβων}
    ii:=2;
    repeat
      i:=t[ii];
      ii:=ii+1;
    until i<>-1;

    min:=d[i];th:=i;
    for j:=2 to n
    do   if t[j]<>-1 then
         if min>d[j] then
         begin
           min:=d[j];
          th:=j;
         end;
    s[k]:=th; d[th]:=min;
    t[th]:=-1; writeln(th);
    for j:=2 to n do
    begin
       if t[j]<>-1 then
      if d[j]>d[th]+c[th,j] then
      begin
        d[j]:=d[th]+c[th,j];
        pr[j]:=th;
      end;
    end;
  end;
end; {da}

begin {main}
 clrscr;
 init;
 da;
 for i:=1 to n
   do writeln(s[i]:2,d[s[i]]:10);
 writeln('---------------------');
 for i:=1 to n
   do writeln(pr[i]);
```

end.

Το αρχείο **Graph.dat** με δεδομένα που αντιπροσωπεύουν τον κόμβο του *Σχήματος ??* (το ∞ εδώ υλοποιείται σαν 32000).

```
5
32000    10 32000    30    100
32000 32000    50 32000 32000
32000 32000 32000 32000    10
32000 32000    20 32000    60
32000 32000 32000 32000 32000
```

11. ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΣ (Hash tables)

11.1 Εισαγωγικές Έννοιες / Ορισμοί

⇒ **Πίνακες Άμεσης Προσπέλασης:** Εάν έχουμε μία συλλογή *n* στοιχείων των οποίων τα κλειδιά είναι μοναδικοί ακέραιοι αριθμοί στο διάστημα *[1,m]*, $m \geq n$, τότε μπορούν να αποθηκευτούν σε ένα πίνακα *T[m]*, όπου κάθε στοιχείο T_i είναι ή άδειο ή περιέχει κάποιο αντικείμενο της συλλογής με κλειδί *i*. Η αναζήτηση είναι προφανές ότι εκτελεί $O(1)$ λειτουργίες. Ο αλγόριθμος αναζήτησης είναι:

Α. Δεδομένου ενός κλειδιού k,

Β. Άμεση προσπέλαση του στοιχείου T[k] όπου εάν δεν είναι άδειο τότε το υπό αναζήτηση στοιχείο βρέθηκε αλλιώς δεν υπάρχει καθόλου.

⇒ Στην περίπτωση κατά την οποία τα κλειδιά δεν είναι μοναδικά τότε πρέπει να δημιουργηθεί ένα σύνολο από *m* λίστες όπου η κεφαλή αυτών των λιστών θα είναι αποθηκευμένη στον πίνακα *T[k]*.

⇒ Όταν η απεικόνιση δεν είναι άμεση αλλά υπάρχει μία συνάρτηση $h(k):[1,m]$, δηλαδή απεικονίζει κάθε τιμή του κλειδιού *k* στο πεδίο τιμών $[1,m]$, τότε μπορούμε να γενικεύσουμε θέτοντας $T[h(k)]$ αντί *T[k]*.

⇒ **Συναρτήσεις Απεικόνισης.** Η άμεση προσπέλαση απαιτεί από την ***hashing*** συνάρτηση *h(k)* μία ένα προς ένα απεικόνιση όπου το κάθε κλειδί *k* απεικονίζεται σε ένα μοναδικό ακέραιο του διαστήματος $[1,m]$. Αυτή είναι η περίπτωση της τέλειας *hashing* συνάρτησης. Δυστυχώς, η εύρεση μίας τέτοιας συνάρτησης δεν είναι πάντα εφικτή. Συνήθως βρίσκουμε μία *hash* συνάρτηση η οποία απεικονίζει τα περισσότερα από τα κλειδιά σε μοναδικούς ακέραιους αλλά υπάρχει και ένας μικρός αριθμός κλειδιών που απεικονίζονται σε ίδιους ακέραιους. Σ' αυτή την περίπτωση θεωρούμε ότι έχουμε **σύγκρουση** (collision) κλειδιών. Εάν ο αριθμός των συγκρούσεων είναι μικρός τότε η τεχνική δουλεύει πολύ καλά και πάλι λειτουργεί με πολυπλοκότητα $O(1)$.

⇒ **Χειρισμός Συγκρούσεων.** Στη περίπτωση όπου πολλαπλά κλειδιά απεικονίζονται στον ίδιο ακέραιο, τότε στοιχεία με διαφορετικά κλειδιά διεκδικούν να αποθηκευτούν στον ίδιο χώρο. Έχουν αναπτυχθεί αρκετές τεχνικές για την αντιμετώπιση αυτού του προβλήματος με βασικότερες τις εξής:

1. Δημιουργία Περιοχών Υπερχείλισης,
2. Re-hashing,
3. Χρήση Γειτονικών Θέσεων,
4. Χρήση Τυχαίων Θέσεων και άλλες.

Για παράδειγμα στην χρήση γειτονικών θέσεων (+1 ή –1) στην περίπτωση σύγκρουσης ψάχνεται η γειτονική προς την κανονική θέση και εάν είναι άδεια το στοιχείο τοποθετείται εκεί αλλιώς συνεχίζεται η αναζήτηση κενής θέσης στην επόμενη γειτονική και ούτω καθ' εξής. Αυτή είναι μία αρκετά καλή μέθοδος αλλά

θα πρέπει να προσεχθεί ιδιαιτέρως η περίπτωση όπου υπάρχουν ενδεχόμενες διαγραφές από τον πίνακα.

ΠΑΡΑΔΕΙΓΜΑ

Εάν υποθέσουμε ότι το κλειδί του κάθε στοιχείου / εγγραφής είναι ένα μοναδικό αλφαριθμητικό, πχ αριθμός ταυτότητας, ΑΦΜ, κωδικός προϊόντος κλπ. Επειδή δεν αντιπροσωπεύει άμεσα κάποιον ακέραιο θα μπορούσε η hashing συνάρτηση να προσθέτει τους αντίστοιχους ASCII κωδικούς και επειδή $n = 10$ να υπολογίζει και το υπόλοιπο της διαίρεσης του ακέραιου που προκύπτει με το 10. Δηλαδή εάν $k = k_1 k_2 \ldots k_x$ όπου k_y είναι κάποιο αλφαριθμητικό, τότε $h(k) = (card(k_1) + card(k_2) + \cdots + card(k_x)) \bmod 10$.

Αλγόριθμος Εισαγωγής

Δεδομένα: Κλειδί k, hashing συνάρτηση h(k), n μέγεθος αποθηκευτικού πίνακα
1. Υπολογισμός h(k)
2. Εάν το T[h(k)] είναι κενό, εισαγωγή της εγγραφής στη θέση h(k) και τέλος.
3. Εάν το T[h(k)] είναι μη κενό τότε έστω x=h(k)
4. x=x+1
5. Εάν x>n τότε x=1
6. Εάν T[x] είναι κενό, εισαγωγή της εγγραφής στη θέση h(x) και τέλος αλλιώς συνέχισε στο βήμα 4.

Αλγόριθμος Αναζήτησης

Δεδομένα: Κλειδί k, hashing συνάρτηση h(k), n μέγεθος αποθηκευτικού πίνακα
1. Υπολογισμός h(k) και έστω x=h(k).
2. Εάν το T[x] είναι το ζητούμενο τότε επιστροφή του x και τέλος.
3. Εάν το T[x] είναι κενό τότε αποτυχία αναζήτησης και τέλος.
4. Εάν το T[x] είναι μη κενό και όχι το ζητούμενο τότε
    a. x=x+1
    b. Εάν x>n τότε x=1
    c. Συνέχιση στο βήμα 2.

Το πρόβλημα που θα προκύψει με ενδεχόμενες διαγραφές, είναι ότι η ανακάλυψη ενός κενού χώρου στον αλγόριθμο αναζήτησης δεν συνεπάγεται πάντα την αποτυχία της αναζήτησης. Εάν ο κενός χώρος δημιουργήθηκε λόγω διαγραφής τότε είναι πιθανό το υπό αναζήτηση στοιχείο να υπάρχει σε ένα από τους επόμενους γειτονικούς χώρους. Μία λύση αυτού του προβλήματος, είναι να τίθεται ένα σύμβολο διαγραφής σε κάθε κενό χώρο που προέκυψε από διαγραφή. Για παράδειγμα, το κενό κελί είναι null ή 0 ενώ το διαγραμμένο '***'.

11.2 Άσκηση

Να περιγραφεί η οργάνωση αρχείων με δείκτες. Στη συνέχεια να γράψετε τους αλγόριθμους «εισαγωγής», «αναζήτησης», «ταξινόμησης», «διόρθωσης», και

«διαγραφής». Τέλος να υλοποιήσετε ένα τέτοιας οργάνωσης αρχείο σε κάποια γλώσσα προγραμματισμού.

12. ΟΙ ΚΛΑΣΕΙΣ *P* ΚΑΙ *NP*

Όπως έχει ήδη ορισθεί, η πολυπλοκότητα ενός αλγόριθμου είναι το σύνολο των βημάτων (στοιχειωδών πράξεων .όπως πρόσθεση, σύγκριση κλπ) που χρειάζεται για να επιλύσει κάποιο πρόβλημα. Επίσης, έχει ήδη συζητηθεί ότι αυτή η πολυπλοκότητα εξαρτάται από το πλήθος των δεδομένων εισόδου. Δηλαδή, μπορεί να αποδοθεί από μία συνάρτηση $f : N \to N$ όπου για κάθε συγκεκριμένο πλήθος δεδομένων εισόδου n επιστρέφει το πλήθος των στοιχειωδών πράξεων $f(n)$ που απαιτούνται για την επίλυση του προβλήματος.

12.1 Αιτιοκρατικοί και μη Αιτιοκρατικοί Αλγόριθμοι

Όλοι οι αλγόριθμοι που έχουν εξετασθεί μέχρι τώρα έχουν την ιδιότητα του να είναι καλά και μάλιστα μοναδικά ορισμένοι. Δηλαδή, να προσδιορίζουν ακριβώς το κάθε βήμα τους και να μην υπάρχει η παραμικρή αμφιβολία σε όλα τα στάδια της εκτέλεσής τους. Αυτοί οι αλγόριθμοι ονομάζονται αιτιοκρατικοί ή ντετερμινιστικοί (*deterministic*). Και βέβαια συμφωνούν απόλυτα με το πώς ένας υπολογιστής αντιλαμβάνεται ένα πρόγραμμα. Από την θεωρητική σκοπιά όμως, είναι δυνατόν να απαλειφθεί αυτός ο παράγοντας. Δηλαδή, να είναι δυνατή η μη καλά ορισμένη λειτουργία κάποιου σημείου του αλγόριθμου και μάλιστα να επιτρέπεται να επιλέξει από ένα σύνολο πιθανών αποφάσεων αντί της μίας και μοναδικής. Αυτή η εξέλιξη οδηγεί στους λεγόμενους μη-αιτιοκρατικούς αλγόριθμους. Η γενική λειτουργία τους συνοψίζεται ως εξής:

1) Επιλογή(*S*): «τυχαία» επιλογή ενός στοιχείου του συνόλου *S*
2) Αποτυχία(): ένδειξη μη επιτυχούς κατάληξης
3) Επιτυχία(): ένδειξη επιτυχούς κατάληξης

Παράδειγμα 1: Αναζήτηση στοιχείου *x* σε πίνακα A_N. Πέραν των κλασσικών προσδιοριστικών αλγόριθμων αναζήτησης σε πίνακα, ένας μη αιτιοκρατικός αλγόριθμος θα μπορούσε να περιγράψει το πρόβλημα όπως φαίνεται στον *Αλγόριθμο 22*. Η πολυπλοκότητα του αλγόριθμου είναι της τάξης *O*(1).

Αλγόριθμος 22: Μη αιτιοκρατικός αλγόριθμος αναζήτησης

```
1: j ← Επιλογή(1,2,…,N)
2: Εάν (x = A_j) Τότε
3:       Επέστρεψε j
4:       Επιτυχία()
5: Αλλιώς
6:       Αποτυχία()
7: Τέλος Εάν
8: Τέλος Αλγόριθμου «Μη αιτιοκρατικός αλγόριθμος αναζήτησης»
```

Παράδειγμα 2: Ταξινόμηση πίνακα A_N. Στον *Αλγόριθμο 23* περιγράφεται ένας μη αιτιοκρατικός αλγόριθμος ταξινόμησης πίνακα. Ο αλγόριθμος ταξινομεί τον πίνακα σε φθίνουσα διάταξη και χρησιμοποιεί έναν βοηθητικό πίνακα B_N. Πρώτα,

αρχικοποιεί τον πίνακα Β και στην συνέχεια τοποθετεί στοιχεία του πίνακα *A* στον *B*. Μετά το τέλος αυτής της διαδικασίας, ο πίνακας Β στην ουσία αποτελεί μία αναδιάταξη των στοιχείων του Α και για αυτό ο αλγόριθμος ελέγχει να δει εάν ο Β είναι ταξινομημένος ή όχι και επιστρέφει αντίστοιχα επιτυχία ή αποτυχία. Η πολυπλοκότητα του αλγόριθμου είναι της τάξης $O(N)$.

Αλγόριθμος 23: Μη αιτιοκρατικός αλγόριθμος ταξινόμησης

```
1: Για i ←1 Μέχρι N
2:       Bi ← 0 /* Αρχικοποίηση Β */
3: Τέλος Για (i)
4: Για i ← 1 Μέχρι N
5:       j ← Επιλογή(1,2,…,N)
6:       Εάν (Bj ≠ 0) Τότε
7:             Bj ← Ai
8:       Τέλος Εάν
9: Τέλος Για (i)
10: Για i ← 1 Μέχρι N
11:      Εάν (Bi > Bi+1) Τότε
12:            Επέστρεψε Αποτυχία()
13:      Τέλος Εάν
14: Τέλος Για (i)
15: Επέστρεψε Επιτυχία()
16: Τέλος αλγόριθμου «Μη αιτιοκρατικός  ταξινόμησης»
```

12.2 Η Κλάση Προβλημάτων *P*

Αν ένα πρόβλημα αποσκοπεί στη λήψη μιας δυαδικής απόφασης (ναι / όχι) τότε λέγεται πρόβλημα απόφασης. Από τα προβλήματα απόφασης όσα επιδέχονται λύση από πολυωνυμικούς αλγόριθμους αποτελούν την κλάση (οικογένεια) *P*. Δηλαδή, ένας αλγόριθμος θεωρείται ότι είναι υπολογιστικής πολυπλοκότητας της τάξης *P*, εάν υπάρχει ένα πολυώνυμο $p()$ τέτοιο ώστε η πολυπλοκότητα του αλγόριθμου να μπορεί να εκφρασθεί σαν $O(p(N))$ για οποιοδήποτε πλήθος δεδομένων *N*.

12.3 Η Κλάση Προβλημάτων *NP*

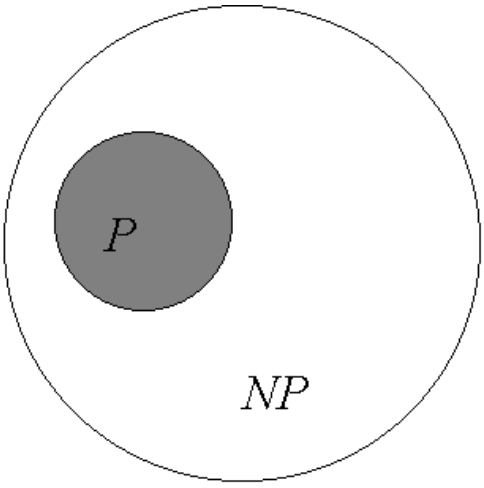
Έστω ένα πρόβλημα απόφασης μεγέθους *n* της μορφής:

«Τα δεδομένα $A=(A(1), A(2), \ldots, A(n))$ ικανοποιούν την συνθήκη *C*»

Δηλαδή, για παράδειγμα ας θεωρηθεί το ακόλουθο πρόβλημα: Δοθέντος ενός φυσικού αριθμού *m* με δυαδική αναπαράσταση $a = a_n a_{n-1} \ldots a_2 a_1$, όπου $a_i \in \{0,1\}$, να ελεγχθεί εάν αυτός ο αριθμός είναι σύνθετος, δηλαδή εάν έχει ακέραιους διαιρέτες.

Έστω τώρα ότι η εν λόγω συνθήκη ικανοποιείται αν βρεθεί έστω και ένας διαιρέτης (εκτός φυσικά της μονάδας και του εαυτού του).

Ας υποτεθεί τώρα ότι ένας «από μηχανής Θεός» προτείνει έναν αριθμό *b* κάποιου μεγέθους ο οποίος ισχυρίζεται ότι διαιρεί τον *a*. Εάν ο «από μηχανής Θεός» είναι αξιόπιστος, το πρόβλημα παίρνει την μορφή του ελέγχου στο εάν ο *b* διαιρεί τον *a*. Εάν το νέο αυτό πρόβλημα ανήκει στην κλάση *P*, τότε ορίζεται πως το αρχικό πρόβλημα ανήκει στην κλάση *NP*. Προφανώς ισχύει $P \subseteq NP$. Ωστόσο παραμένει αναπόδεικτη η εικασία $P \subset NP$. Αυτό δείχνει την άγνοια του ότι ενώ είναι γνωστά προβλήματα της κλάσης *NP* δεν μπορεί να αποδειχθεί ότι δεν μπορεί να βρεθούν κάποιοι καλύτεροι αλγόριθμοι για αυτά τα προβλήματα ώστε να ανήκουν στην κλάση *P* (*Σχήμα 12.1*).



Σχήμα 12.1: Σχέση κλάσεων P και NP με την υπόθεση $P \neq NP$

Ορισμός 12.1: Το σύνολο των προσδιοριστικών αλγόριθμων απόφασης που επιλύονται σε πολυωνυμικό χρόνο, ανήκουν στην κλάση *P*.

Ορισμός 12.2: Το σύνολο των αλγόριθμων απόφασης που επιλύονται σε πολυωνυμικό χρόνο από μη προσδιοριστικούς αλγόριθμους, ανήκουν στην κλάση *NP*.

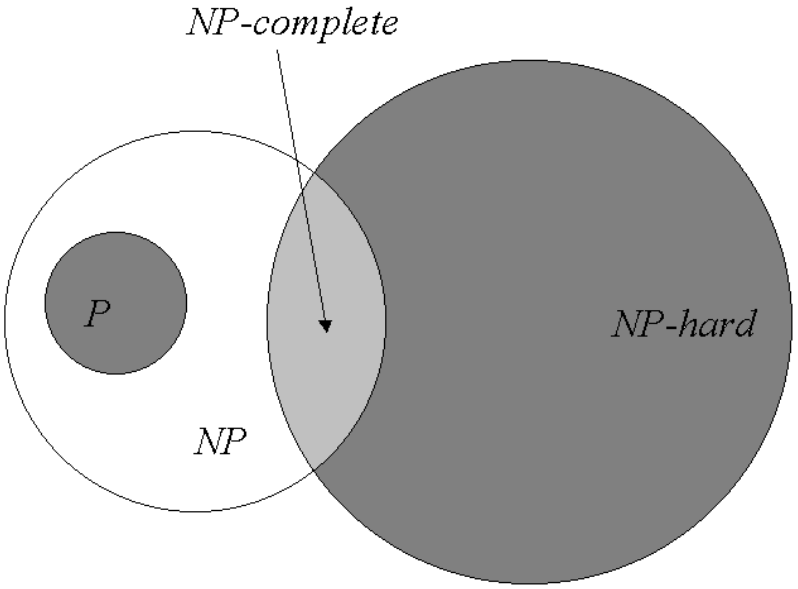
12.4 Η κλάση των *NP-hard* και *NP-complete* Προβλημάτων

Σαν συνέπεια της προαναφερθείσης άγνοιας, ορίζεται μία κλάση προβλημάτων *NP*-complete (*NPC*). Τα προβλήματα που ανήκουν σ' αυτή την κλάση έχουν τις εξής δύο ιδιότητες:

1. Ανήκουν στην κλάση *NP*
2. Αν έστω και για ένα από αυτά αποδειχθεί (στο μέλλον) ότι ανήκει στην κλάση *P* τότε αυτόματα έχει αποδειχθεί ότι $NPC \subset P$, δηλαδή όλα τα *NP*-complete προβλήματα θα ήταν επιλύσιμα σε πολυωνυμικό χρόνο.

Από τα παραπάνω είναι σαφές ότι από τα *NP* προβλήματα, τα *P* είναι τα εύκολα και τα *NP*-complete τα δύσκολα ως προς την υπολογιστική τους πολυπλοκότητα. Από την άλλη μεριά όμως μπορεί να αποδειχθεί ότι όλα τα *NP* προβλήματα, συμπεριλαμβανόμενων και των *NP*-complete επιδέχονται αλγόριθμους με

πολυπλοκότητα $f(n) \in O(2^n)$. Είναι δηλαδή λιγότερο ή το πολύ ισοδύναμοι με αλγόριθμους εκθετικής πολυπλοκότητας



Σχήμα 12.2: Σχέσεις των κλάσεων *P*, *NP*, *NP-hard* και *NP-complete*

Ορισμός 12.3: Έστω L_1 και L_2 είναι δύο προβλήματα απόφασης. Ορίζεται ότι το πρόβλημα L_1 περιορίζει το πρόβλημα L_2 ($L_1 \propto L_2$) εάν και μόνον εάν υπάρχει ένας τρόπος ώστε να επιλυθεί το L_1 από προσδιοριστικό αλγόριθμο σε πολυωνυμικό χρόνο χρησιμοποιώντας έναν προσδιοριστικό αλγόριθμος που επιλύει το L_2.

Ορισμός 12.4: Ένα πρόβλημα *L* θεωρείται ότι ανήκει στην κλάση *NP-hard* εάν η ικανοποίηση του προβλήματος περιορίζει το L (ικανοποίηση $\propto L$)

Ορισμός 12.5: Ένα πρόβλημα *L* θεωρείται ότι ανήκει στην κλάση *NP-complete* εάν και μόνο εάν το *L* ανήκει στην κλάση *NP-hard* και ταυτόχρονα στην κλάση *NP*.

Οι σχέσεις που υπάρχουν μεταξύ των διαφόρων κλάσεων προβλημάτων φαίνονται στο *Σχήμα 12.2*.

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Φ. Αφράτη, και Γ. Παπαγεωργίου, «Αλγόριθμοι: Μέθοδοι Σχεδίασης Και Ανάλυση Πολυπλοκότητας», Εκδόσεις Συμμετρία, Αθήνα, 1993.

2. Μ. Δουκάκης, «Δομές Δεδομένων, Αλγόριθμοι», Εκδόσεις Ζυγός, Θεσσαλονίκη, 1998.

3. Χ. Κοίλιας, «Δομές Δεδομένων και Οργάνωση Αρχείων», Εκδόσεις Νέων Τεχνολογιών, Αθήνα, 1993.

4. Γ. Μανωλόπουλος, «Μαθήματα Θεωρίας Γράφων», Εκδόσεις Νέων Τεχνολογιών, Αθήνα, 1996.

5. Π. Μποζάνης, «ΑΛΓΟΡΙΘΜΟΙ: Σχεδιασμός και Ανάλυση», Εκδόσεις Τζιόλα, 2003.

6. Ι. Χατζηλυγερούδης, «Δομές Δεδομένων, Εισαγωγή Στην Πληροφορική, Τόμος Γ», Ελληνικό Ανοικτό Πανεπιστήμιο, Πάτρα, 2000 .

7. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, “Data Structures And Algorithms”, Addison – Wesley Publishing Company, USA, 1985.

8. M. Ben-Ari, «Ταυτόχρονος Προγραμματισμός», Κλειδάριθμος, 1997

9. M. R. Garey, and D. S. Johnson, “Computers And Intractability, A Guide To The Theory Of NP-Completeness”, W.H. Freeman and Company, New York, USA, 1979.

10. N. D. Jones, “Computability and Complexity. From a Programming Perspective”, Massachusetts Institute of Technology, 1997.

11. M. Gondran, and M. Minoux, “Graphs And Algorithms”, John Wiley & Sons Inc, USA, 1984.

12. E. Horowitz, S. Sahni, and S. Rajasekaran, “Computer Algorithms”, W.H. Freeman and Company, USA, 1998.

13. D. Mount, “Algorithms”, Lecture Notes, Dept. of Computer Science, University of Maryland, 1998.

14. C. Reeves (Edited), “Modern Heuristic Techniques for Combinatorial Problems”, McGraw Hill Book Company, UK, 1995.

15. R. Sedgewick, “Algorithms”, Addison – Wesley Publishing Company, USA, 1984.

16. R. Sedgewick, “Algorithms In C”, Addison – Wesley Publishing Company, USA, 1998.

17. K. Thulasiraman, and M. N. S. Swamy, “Graphs: Theory And Algorithms”, John Wiley & Sons Inc, USA, 1992.

18. H. S. Wilf, “Algorithms And Complexity”, Prentice Hall Inc., USA, 1986.