Τ.Ε.Ι ΠΕΛΟΠΟΝΝΗΣΟΥ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ Τ.Ε

Πτυχιακή Εργασία

# «ΑΝΑΔΡΟΜΙΚΟΙ ΑΛΓΟΡΙΘΜΟΙ»

Μπαζούκα Ιλμιάν

Α.Μ: 2007031

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Καραγιώργος Γρηγόρης

ΣΠΑΡΤΗ 2014

« Η επανάληψη είναι ανθρώπινη, αλλά η αναδρομή θεϊκή »

Θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου Καραγιώργο Γρηγόρη
για την βοήθεια του και την δυνατοτήτα να εκπονήσω την παρούσα πτυχιακή εργασία.

# ΠΕΡΙΕΧΟΜΕΝΑ

## ΚΕΦΑΛΑΙΟ 1°

## Εισαγωγή


**1.1** Γενικά ..... **6**

**1.2** Διάρθρωση εργασίας ..... **6**

**1.3** Αλγόριθμος ..... **7**

**1.4** Πολυπλοκότητα αλγορίθμων ..... **8**

**1.4.1** Ασυμπτωτικός συμβολισμός Ο ..... **10**

**1.5** Τύποι Προβλημάτων ..... **13**


## ΚΕΦΑΛΑΙΟ 2°

## Αναδρομή


**2.1** Αναδρομή ..... **14**

**2.2** Υπολογισμός Παραγοντικού ..... **16**

**2.3** Σειρά Fibonacci ..... **19**

**2.4** Μέγιστος Κοινός Διαιρέτης ..... **22**

**2.5** Ελάχιστο Κοινό Πολλαπλάσιο ..... **25**

**2.6** Πύργοι Hanoi ..... **27**

**2.7** Ύψωση σε Δύναμη ..... **30**

**2.8** Παλινδρομικός Αριθμός ..... **33**

**2.9** Παλινδρομική Λέξη ..... **35**

**2.10** Αντιστροφή Αριθμού ..... **37**

**2.11** Αντιστροφή Λέξης ..... **39**

**2.12** Άθροισμα Ν Πρώτων Αριθμών ..... **41**

**2.13** Τρίγωνο Pascal ..... **44**

**2.14** Μετατροπή από Δεκαδικό σε Δυαδικό ..... **48**

**2.15** Πόσα Νούμερα έχει ένας αριθμός ..... **50**

**2.16** Πολλαπλασιασμός 2 Ακέραιων αριθμών ..... **52**

## ΚΕΦΑΛΑΙΟ 3ο

## Αναδρομικοί Αλγόριθμοι Αναζήτησης & Ταξινόμησης


**3.1** Αναζήτηση ..... **54**

**3.2** Δυαδική Αναζήτηση ..... **55**

**3.3** Διαίρει και Κυρίευε ..... **59**

**3.3.1** Γρήγορη Ταξινόμηση ..... **60**

**3.3.2** Ταξινόμηση με Συγχώνευση ..... **65**


## ΚΕΦΑΛΑΙΟ 4ο

## Δένδρα


**4.1** Γενικά ..... **70**

**4.2** Δυαδικά Δένδρα ..... **72**

**4.3** Δυαδικά Δένδρα Αναζήτησης ..... **80**

**ΒΙΒΛΙΟΓΡΑΦΙΑ** ..... **85**

**ΣΥΜΠΕΡΑΣΜΑΤΑ** ..... **86**

**ΠΑΡΑΡΤΗΜΑΤΑ** ..... **87**

# ΚΕΦΑΛΑΙΟ 1$^{ο}$

# Εισαγωγή

## 1.1 Γενικά

Η πληροφορική είναι μία επιστήμη ολοένα αναπτυσσόμενη. Κάθε προγραμματιστής που μπαίνει σε αυτόν τον δύσκολο αλλά εντυπωσιακό κόσμο του προγραμματισμού υπολογιστών, θα πρέπει να έρθει σε επαφή με τα εργαλεία που διαθέτει κάθε γλώσσα προγραμματισμού. Ένα από αυτά τα εργαλεία είναι η αναδρομή. Η αναδρομή ως σκέψη, ως έννοια, ως φιλοσοφία και ως προγραμματιστική τεχνική. Όταν κάποιος προγραμματιστής καταφέρει να καταλάβει την φιλοσοφία της αναδρομής και να αποκτήσει οικειότητα με αυτήν την λογική, οι αναδρομικοί αλγόριθμοι αποδεικνύουν πόσο σπουδαίοι και πανίσχυροι είναι.

## 1.2 Διάρθρωση εργασίας

Στην παρούσα εργασία θα ασχοληθούμε με τους αναδρομικούς αλγόριθμους. Στον πρώτο κεφάλαιο θα ασχοληθούμε αρχικά με την έννοια του αλγορίθμου, για την πολυπλοκότητα των αλγορίθμων και της κατηγορίες τους. Καθώς και με το τη είναι πρόβλημα και της κατηγορίες των προβλημάτων. Στο δεύτερο κεφάλαιο θα ασχοληθούμε με τους αναδρομικούς αλγορίθμους αφού πρώτα αποδώσουμε την έννοια της αναδρομής. Σε αυτό θα αναπτύξουμε και απλούς αλλά θεμελιώδεις αναδρομικούς αλγορίθμους. Στο τρίτο κεφάλαιο θα ασχοληθούμε με την αναζήτηση και ταξινόμηση αλλά κυρίως με την τεχνική διαίρει και κυρίευε(βασίλευε) και την υλοποίηση κάποιον αλγορίθμων. Στο τέταρτο και τελευταίο κεφάλαιο της εργασίας

θα αναφέρονται αρχικά βασικές έννοιες για την μελέτη των δενδρικών δομών και στην συνέχεια θα παρουσιάζονται αναδρομικές υλοποιήσεις σε μη γραμμικές δομές δεδομένων(δένδρα). Στο τέλος της εργασίας υπάρχουν όλα τα πρόγραμμα μαζεμένα και υλοποιημένα σε γλώσσα προγραμματισμού C με την σειρά που είναι στην εργασία.

## 1.3 Αλγόριθμος

Η θεωρία των αλγορίθμων έχει μεγάλη παράδοση και μερικοί αλγόριθμοι υπάρχουν χιλιάδες χρόνια, όπως για παράδειγμα ο αλγόριθμος του Ευκλείδη για την εύρεση του μέγιστου κοινού διαιρέτη δύο αριθμών. Σήμερα το πεδίο της Θεωρίας Αλγορίθμων είναι ένα ιδιαίτερα ευρύ και πλούσιο πεδίο.

Η λέξη αλγόριθμος (algorithm) προέρχεται από μια μελέτη του Πέρση μαθηματικού Abu Ja'far Mohammed ibn Musa al Khowarizmi, η μελέτη αυτή μεταφράστηκε στα λατινικά και άρχιζε με τη φράση "Algoritmi dixit..." (ο αλγόριθμος λέει ....). Τη σημερινή του αξία απόκτησε από την αρχή του 20ού αιώνα με την ανάπτυξη της ομώνυμης θεωρίας και φυσικά με την επικαιρότητα των ηλεκτρονικών υπολογιστών.

Αλγόριθμος είναι μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος. 'Οπως αναφέρεται στο βιβλίο του Ι. Μανωλόπουλο, 2004[6].

Κάθε αλγόριθμος απαραίτητα ικανοποιεί τα επόμενα κριτήρια.

- **Είσοδος**: Καμία, μία ή περισσότερες τιμές δεδομένων πρέπει να δίνονται ως είσοδοι στον αλγόριθμο. Η περίπτωση που δεν δίνονται τιμές δεδομένων εμφανίζεται, όταν ο αλγόριθμος δημιουργεί και επεξεργάζεται κάποιες πρωτογενείς τιμές με τη βοήθεια συναρτήσεων παραγωγής τυχαίων αριθμών ή με τη βοήθεια άλλων απλών εντολών.
- **Έξοδος**: Ο αλγόριθμος πρέπει να δημιουργεί τουλάχιστον μία τιμή δεδομένων ως αποτέλεσμα προς το χρήστη ή προς έναν άλλο αλγόριθμο.
- **Καθοριστικότητα**: Κάθε εντολή πρέπει να καθορίζεται χωρίς καμία αμφιβολία για τον τρόπο εκτέλεσής της. Λόγου χάριν, μία εντολή διαίρεσης πρέπει να θεωρεί και την περίπτωση, όπου ο διαιρέτης λαμβάνει μηδενική τιμή.
- **Περατότητα**: Ο αλγόριθμος να τελειώνει μετά από πεπερασμένα βήματα εκτέλεσης των εντολών του. Μία διαδικασία που δεν τελειώνει μετά από ένα συγκεκριμένο αριθμό βημάτων δεν αποτελεί αλγόριθμος, αλλά λέγεται απλά υπολογιστική διαδικασία.
- **Αποτελεσματικότητα**: Κάθε μεμονωμένη εντολή του αλγορίθμου να είναι απλή. Αυτό σημαίνει ότι μία εντολή δεν αρκεί να έχει ορισθεί, αλλά πρέπει να είναι και εκτελέσιμη.

## 1.4 Πολυπλοκότητα Αλγορίθμων

Όταν θέλουμε να επιλύσουμε ένα πρόβλημα με την αλγοριθμική μέθοδο, μας ενδιαφέρει πόσο αποδοτικός είναι ο αλγόριθμος. Άλλωστε όπως θα δούμε και στα επόμενα κεφάλαια για την επίλυση ενός προβλήματος μπορούν να υπάρχουν πολλοί αλγόριθμοι. Κάθε φορά πρέπει να επιλέγουμε τον πιο αποδοτικό. Με τον όρο **απόδοση αλγόριθμου** εννοούμε το πόσο μνήμη και το πόσο υπολογιστικό χρόνο(πόσα βήματα κάνει) που απαιτούνται για να δώσει ο αλγόριθμος αποτέλεσμα.

Γενικώς στην ανάλυση της απόδοσης ενός αλγορίθμου μας ενδιαφέρουν δύο προσεγγίσεις, η αναλυτική και η πειραματική. Στην αναλυτική χρησιμοποιούμε αναλυτικές μεθόδους, ενώ στη μέτρηση της απόδοσης διεξάγουμε πειράματα.

**Η πολυπλοκότητα χώρου** ενός αλγορίθμου είναι το ποσό μνήμης που χρειάζεται για να εκτελεστεί μέχρι να ολοκληρωθεί. **Η χρονική πολυπλοκότητα** ενός αλγορίθμου είναι το ποσό του υπολογιστικού χρόνου που απαιτείται για να εκτελεστεί μέχρι να ολοκληρωθεί. 'Οπως αναφέρεται στο βιβλίο του N.Shani, 2004[8].

Ο υπολογισμός του πλήθους των πράξεων που απαιτεί ένας αλγόριθμος για κάποια είσοδο στοιχείων, η οποία είναι η ευνοϊκότερη για τη λύση του προβλήματος, θα λέμε ότι είναι η καλύτερη περίπτωση. Σε αυτή την περίπτωση προφανώς, θα έχουμε το μικρότερο αριθμό πράξεων. Με την ίδια λογική ενδέχεται να υπάρχει κάποια είσοδος στοιχείων σε ένα πρόβλημα, το οποίο απαιτεί τις περισσότερες δυνατές πράξεις. Η περίπτωση αυτή θα λέμε ότι είναι η χειρότερη (worst-case).Η πολυπλοκότητα πρέπει να υπολογίζεται για τη χειρότερη, την καλύτερη, αλλά και τη μέση περίπτωση σχετικά με το χρόνο και τη μνήμη που απαιτούνται. 'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή, 2010[1].

### 1.4.1 Ασυμπτωτικός συμβολισμός Ο

Τις περισσότερες φορές δεν μας ενδιαφέρουν οι ακριβείς τιμές που μπορούν να προκύψουν, από τις μεθόδους που αναφέρθηκαν στην προηγούμενη παράγραφο. Έτσι επιδιώκουμε να βρούμε μία μέθοδο που θα περιγράφει γενικότερα την συμπεριφορά των αλγορίθμων, δηλαδή την τάξη του αλγορίθμου. Για το λόγο αυτό εισάγεται ο συμβολισμός Ο (O -notation), όπου το γράμμα Ο είναι το αρχικό γράμμα της λέξης order και διαβάζεται συμβολισμός Όμικρον κεφαλαίο.

Οι περισσότεροι αλγόριθμοι έχουν πολυπλοκότητα που ανήκει σε μία από τις κατηγορίες που παρουσιάζονται στον επόμενο πίνακα. Με το n συμβολίζεται το μέγεθος του προγράμματος και εξαρτάται από το πρόβλημα. Για παράδειγμα αν ζητείται να ταξινομηθούν 100 αριθμοί, τότε n = 100.

Να σημειώσουμε εδώ ότι ένας αλγόριθμος λέγεται **άριστος** αν αποδειχθεί ότι είναι τόσο αποτελεσματικός για ένα πρόβλημα, ώστε να μην μπορεί να κατασκευαστεί καλύτερος. Για παράδειγμα ένας αλγόριθμος ταξινόμησης με συγκρίσεις (όπως η φυσαλίδα) απαιτεί Ο ($n^2$) συγκρίσεις (τετραγωνικής πολυπλοκότητας). Όμως υπάρχουν ταχύτεροι αλγόριθμοι (όπως ο Quicksort - § 3.3.1) που απαιτούν Ο (nlogn) συγκρίσεις.

Ανάλογα με το χρόνο εκτέλεσής τους οι αλγόριθμοι διακρίνονται στις παρακάτω κατηγορίες ή κλάσεις πολυπλοκότητας:

*Σταθερού χρόνου 0(1):* στην κατηγορία αυτή οι περισσότερες εντολές εκτελούνται μια μόνο φορά ή το πολύ λίγες φορές. Αν όλες οι εντολές ενός προγράμματος διαθέτουν αυτή την ιδιότητα, θεωρούμε ότι ο χρόνος εκτέλεσης του προγράμματος είναι σταθερός

*Λογαριθμικού χρόνου O(logn):* στην κατηγορία αυτή ανήκουν οι αλγόριθμοι των οποίων ο χρόνος εκτέλεσης είναι λογαριθμικός (με βάση το 2). Είναι αξιοσημείωτο το γεγονός ότι ο χρόνος εκτέλεσης σχεδόν δεν αυξάνεται, καθώς αυξάνεται η τιμή του η.

Οι αλγόριθμοι αυτοί είναι πάρα πολύ γρήγοροι και μπορούν να επιλύσουν προβλήματα με μεγάλο αριθμό στοιχείων σε ελάχιστο χρόνο. Η λογική που ακολουθούν για να επιλύσουν κάποιο μεγάλο πρόβλημα, είναι να το μετασχηματίσουν σε μια σειρά από μικρότερα, μειώνοντας σε κάθε βήμα το μέγεθος του προβλήματος κατά ένα σταθερό ποσοστό.

*Γραμμικού χρόνου O(η):* σε αυτή την κατηγορία ανήκουν οι αλγόριθμοι οι οποίοι σε κάθε στοιχείο της εισόδου εκτελούν μόνο μια πράξη. Στα προβλήματα γραμμικού χρόνου, όταν το μέγεθος *η* του προβλήματος διπλασιάζεται, τότε αντίστοιχα διπλασιάζεται και ο χρόνος εκτέλεσης του αλγορίθμου. Αυτή η περίπτωση είναι βέλτιστη για έναν αλγόριθμο που πρέπει να επεξεργαστεί *η* στοιχεία εισόδου (ή να παραγάγει *η* στοιχεία εξόδου).

*Γραμμολοναριθμικού χρόνου O(nlogn):* Στην κατηγορία αυτή ο χρόνος εκτέλεσης είναι *η* log η και εμφανίζεται, όταν οι αλγόριθμοι για την επίλυση κάποιου προβλήματος διαχωρίζουν το πρόβλημα σε μικρότερα υποπροβλήματα, τα οποία επιλύουν ανεξάρτητα και στη συνέχεια συνδυάζουν τις επιμέρους λύσεις. Οι αλγόριθμοι αυτοί παρουσιάζουν καλή συμπεριφορά και η αύξηση των στοιχείων της εισόδου δε δημιουργεί αδιέξοδο. Όταν το *η* διπλασιαστεί, τότε ο χρόνος εκτέλεσης του αλγορίθμου υπερδιπλασιάζεται (αλλά δεν απέχει και πάρα πολύ από το διπλάσιο). Συνεπώς, δεν έχουμε πολύ σημαντική διαφορά από τη γραμμική περίπτωση.

*Πολυωνυμικού χρόνου $O(n^2)$,O(n3)*: Στην κατηγορία αυτή ανήκουν οι αλγόριθμοι των οποίων ο αριθμός των πράξεων εκφράζεται με κάποιο πολυώνυμο του *η*, όπου *η* είναι το πλήθος των στοιχείων του προβλήματος.

*Εκθετικού χρόνου* $O(2^n)$. Στην κατηγορία αυτή λίγοι αλγόριθμοι με εκθετικούς χρόνους εκτέλεσης είναι κατάλληλοι για χρήση σε πρακτικές εφαρμογές. Βέβαια, τέτοιου είδους αλγόριθμοι προκύπτουν ως «πρωτόγονες» λύσεις σε προβλήματα. Όταν έχουμε εκθετική πολυπλοκότητα ίση με *2η* και το *η* είναι 20, τότε ο χρόνος εκτέλεσης θα είναι περίπου 1.000.000 (διότι $2^{20} \approx 1.000.000$). Κάθε φορά που το *η* διπλασιάζεται, ο χρόνος εκτέλεσης τετραγωνίζεται.

'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή, 2010[1].

| **Κατηγορίες πολυπλοκότητας αλγορίθμων** | |
|---|---|
| O (1) | Σταθερή |
| O (logn) | Λογαριθμική |
| O (n) | Γραμμική |
| O (nlogn) | Δεν υπάρχει επιθετικός προσδιορισμός |
| O ($n^2$) | Τετραγωνική |
| O ($n^3$) | Κυβική |
| O ($2^n$) | Εκθετική |

## 1.5 Τύποι προβλημάτων

Βέβαια πρέπει να επισημάνουμε πως τα προβλήματα μπορούν να προκύπτουν από την καθημερινή μας ζωή ή από διάφορους επιστημονικούς χώρους. Ακόμη τα προβλήματα δεν περιέχουν πάντα μαθηματικές και υπολογιστικές διαδικασίες αν και στο παρόν με τέτοια προβλήματα θα ασχοληθούμε. Επίσης επειδή υπάρχουν διάφορα προβλήματα κρίνεται ανάγκη να κατηγοριοποιηθούν με βάση τρία κριτήρια. Με κριτήριο τη δυνατότητα επίλυσης, το βαθμό δόμησης και το είδος επίλυσης

Υπάρχουν τρεις γενικές περιπτώσεις προβλημάτων που μπορούν ή δεν μπορούν να επιλυθούν, δηλαδή μπορεί να υπάρχει η να μην υπάρχει ο κατάλληλος αλγόριθμος έχουμε τα:

- **Επιλύσιμα**, είναι εκείνα τα προβλήματα για τα οποία η λύση τους είναι ήδη γνωστή και έχει διατυπωθεί, όταν δηλαδή υπάρχει ένας γνωστός και ικανός αλγόριθμος που τα επιλύει.

- **Ανοικτά**, ονομάζονται εκείνα τα προβλήματα για τα οποία αλγόριθμος που να τα επιλύει δεν έχει μεν βρεθεί ακόμα, αλλά παράλληλα δεν έχει αποδειχθεί, ότι δεν επιδέχονται λύση..

- **Μη επιλύσιμα,** χαρακτηρίζονται εκείνα τα προβλήματα που έχει αποδειχθεί, ότι δεν υπάρχει ικανός αλγόριθμος για να τα επιλύσει.

# ΚΕΦΑΛΑΙΟ 2$^{ο}$

# Αναδρομή

## 2.1 Η έννοια της αναδρομής

Για τις περισσότερες αλγοριθμικές στρατηγικές που χρησιμοποιούνται για την επίλυση προγραμματιστικών προβλημάτων υπάρχουν αντίστοιχες στρατηγικές προερχόμενες από την καθημερινότητά μας. Για παράδειγμα η τεχνική της επανάληψης μας έρχεται στο μυαλό αν θέλουμε να εκτελέσουμε κάποιες εντολές πολλές φορές και μας είναι αρκετά οικεία από την εμπειρία μας έξω από τον προγραμματισμό υπολογιστών. Η αναδρομή δεν είναι τόσο απλή στη σκέψη, άμα κάποιος καταλάβει την φιλοσοφία της αναδρομής συνειδητοποιεί πόσο σπουδαία είναι. Δεν είναι τυχαίο που ο Bjarne Stroustrup, ο δημιουργός της γλώσσας προγραμματισμού C++ αναφέρει στο βιβλίο του πως: *Η επανάληψη είναι ανθρώπινη, η αναδρομή θεϊκή!*

Η στρατηγική αυτή που ονομάζεται **αναδρομή** (recursion), ορίζεται ως η τεχνική επίλυσης σύμφωνα με την οποία τα μεγάλα προβλήματα λύνονται με αναγωγή τους σε μικρότερα απλούστερα της ίδιας ακριβώς μορφής. Θα λέγαμε λοιπόν πως η στρατηγική της αναδρομής βασίζεται στην αποδόμηση (decomposition) του προβλήματος, ώστε να προκύψουν τα υποπροβλήματα της αναδρομικής λύσης που έχουν ακριβώς την ίδια μορφή με το αρχικό. 'Οπως αναφέρεται στο βιβλίο του E.Roberts, 2011[11].

Η έννοια της αναδρομής αναφέρεται στον τρόπο της επίλυσης προβλημάτων και υλοποιείται με την χρήση αναδρομικών αλγορίθμων.

**Αναδρομικοί αλγόριθμοι** καλούνται αυτοί οι οποίοι επιλύουν ένα πρόβλημα ανάγοντάς το σε ένα μικρότερο στιγμιότυπο του ίδιου προβλήματος. Οι αλγόριθμοι αυτοί υλοποιούνται με αναδρομικά προγράμματα και συναρτήσεις.

Οι αναδρομικοί αλγόριθμοι είναι κατάλληλοι για την υλοποίηση των λειτουργιών και τη διαχείριση των πληροφοριών σε δομές δεδομένων οι οποίες ορίζονται αναδρομικά.

**Αναδρομική συνάρτηση (recursive function)** είναι η συνάρτηση η οποία καλεί τον εαυτό της έμμεσα ή άμεσα (μέσω μιας άλλης συνάρτησης). Όμοια ορίζεται ένα αναδρομικό πρόγραμμα ως ένα πρόγραμμα το οποίο καλεί τον εαυτό του. Όμως δεν μπορεί να καλεί τον εαυτό του συνεχώς, επειδή δε θα τερματίσει ποτέ. Συνεπώς είναι σαφές ότι πρέπει να υπάρχει μια συνθήκη τερματισμού, η οποία να καθορίζει πότε το πρόγραμμα πρέπει να σταματήσει να καλεί τον εαυτό του. 'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή, 2010[1].

Σε αυτό το κεφάλαιο θα δώσουμε πρακτικά παραδείγματα χρήσης της αναδρομής. Γι' αυτόν τον λόγο στις παρακάτω ενότητες θα παρουσιάσουμε πολύ σημαντικές και κλασικές αναδρομικές δομές που έρχονται κυρίως από τον χώρο των μαθηματικών.

Είναι σημαντικό σε αυτό το σημείο να πούμε πως η αναδρομική σκέψη δίνει αλγορίθμους απλούς, εξαιρετικά κομψούς, σύντομους και πλησιέστερους προς τον μαθηματικό ορισμό αλλά δεν είναι πάντα αποδοτικός.

Συχνά ο αναδρομικός τρόπος επίλυσης είναι λιγότερο αποδοτικός από τον επαναληπτικό. Υπάρχουν δύο κρυμμένες επιβαρύνσεις που σχετίζονται με τις αναδρομικές υλοποιήσεις.

Πρώτον, υπάρχει το κόστος της κλήσης μίας συνάρτησης. Αυτό μπορεί να είναι σημαντικό σε **χρόνο**, εάν το ποσό του υπολογισμού ανά επίπεδο μεγαλώνει. Ένα δεύτερο πρόβλημα έχει να κάνει με τη **μνήμη**. Κατά τη διάρκεια της εκτέλεσης μιας αναδρομικής συνάρτησης, είναι ενεργά πολλαπλά αντίγραφα της συνάρτησης και οι τοπικές μεταβλητές που σχετίζονται με κάθε ενεργό αντίγραφο πρέπει να αποθηκεύονται. Αυτό σε ορισμένες εφαρμογές μπορεί να προκαλέσει σοβαρά προβλήματα . 'Οπως αναφέρεται στο βιβλίο του E.Roberts, 2011[11].

## 2.2 Υπολογισμός Παραγοντικού

Το παραγοντικό ενός μη αρνητικού αριθμού N , που γράφεται ως n!(ν παραγοντικό)είναι το γινόμενο :

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

με το 1! = 1 και 0! = 1.Για παράδειγμα, το 3! Είναι το γινόμενο 5·4·3·2·1 που είναι ίσο με 120. Ένας αναδρομικός ορισμός της παραγοντικής συνάρτησης γίνεται παρατηρώντας την παρακάτω σχέση:

$$n! = n \cdot (n-1)!$$

Για παράδειγμα , το 5 είναι σαφώς ίσο με το 5*4!, όπως φαίνεται από το παρακάτω:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

Το πρόγραμμα του Παραγοντικού που υπάρχει πιο κάτω χρησιμοποιεί αναδρομή για να υπολογίζει και να τυπώνει το παραγοντικό του θετικού αριθμού που δίνει ο χρήστης.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ζητείται από τον χρήστη να δώσει έναν θετικό αριθμό και γίνεται έλεγχος από μια δομή επιλογής *if*.Αν είναι ο αριθμός δεν είναι θετικός τότε το πρόγραμμα σταματάει αλλιώς τρέχει την συνάρτηση *factorial()* και τυπώνεται το παραγοντικό του αριθμού.

Για να γίνει κατανοητή η λειτουργία της αναδρομικής συνάρτησης *factorial()* .Θα υπολογίσουμε το παραγοντικό του αριθμού 3.θα καλέσουμε την συνάρτηση *factorial()* με όρισμα τον αριθμό 3.

Και αφού το n δεν είναι 0 Θα εκτελεστεί η τελευταία γραμμή τις συνάρτησης :

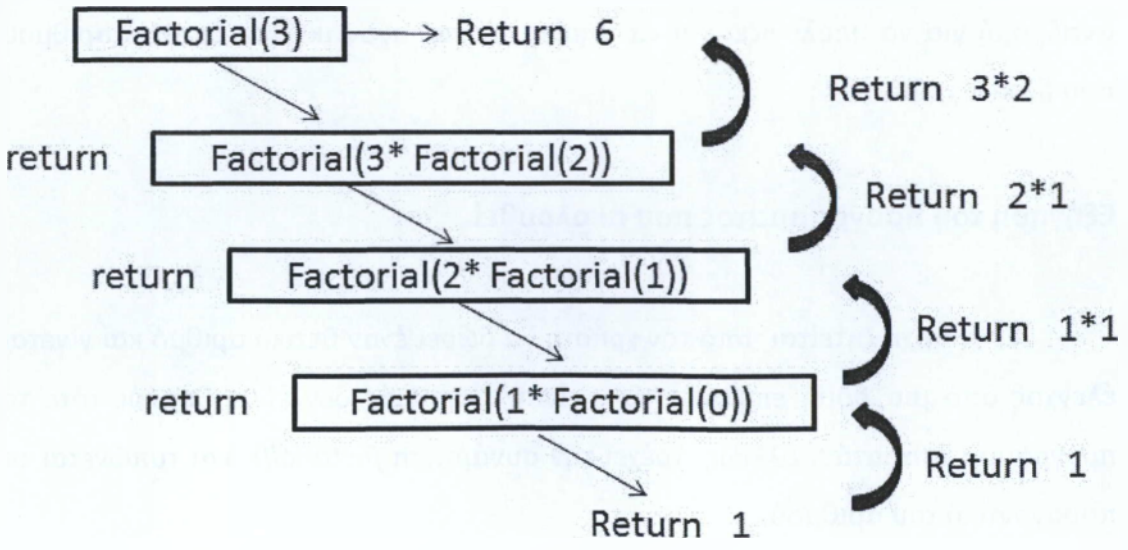
return ( 3 * factorial(3-1)) ;

Στην εντολή αυτή γίνεται κλήση της ίδιας της συνάρτησης *factorial()* , στην οποία ανήκει η εντολή με όρισμα τον αριθμό 2 . Άρα για να υπολογίσουμε τον παραγοντικό του αριθμού 3 πρέπει να υπολογίσουμε πρώτα το παραγοντικό του αριθμού 2 και για να υπολογίσουμε το παραγοντικό του 2 πρέπει να υπολογίσουμε το παραγοντικό του 1 , το οποίο προϋποθέτει τον υπολογισμό του παραγοντικού του 0. Όταν η συνάρτηση κληθεί με n = 0 , εκτελείται η εντολή :

*return 1;*

Και υπολογίζεται το παραγοντικό του αριθμού 0,το οποίο ισούται με 1.Το αποτέλεσμα χρησιμοποιείται στον υπολογισμό του παραγοντικού του αριθμού 1 .Με την ίδια λογική το αποτέλεσμα του παραγοντικού αριθμού 1 χρησιμοποιείται για να υπολογιστεί το παραγοντικό του αριθμού 2 , του οποίου το αποτέλεσμα χρησιμοποιείται για να υπολογιστεί το παραγοντικό του αριθμού 3.

Στην εικόνα φαίνεται η λογική των διαδοχικών αναδρομικών κλήσεων και η επιστροφή των αποτελεσμάτων.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

int factorial(int n);

int main()
{

   int n , fact;

   printf("Δώσε έναν θετικό αριθμό: \n");
   scanf("%d",&n);

   if (n < 0)
     printf("Ο αριθμός που δώσατε δεν είναι θετικός. \n");
   else
   {
      fact = factorial(n);
      printf("Το παραγοντικό του %d είναι : %d \n", n, fact);
   }

 system("PAUSE");
 return 0;
}

int factorial(int n)
{
   if ( n <= 1 )
     return 1;
   else
     return ( n * factorial(n-1)) ;
}
```

## 2.3 Σειρά Fibonacci

Η σειρά Fibonacci ξεκινά με το 0 και το 1 και έχει την ιδιότητα ότι κάθε επόμενος αριθμός Fibonacci είναι το άθροισμα των προηγούμενων δυο αριθμών.

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Η σειρά υπάρχει στην φύση και ιδιαίτερα περιγράφει μια μορφή σπείρας. Η αναλογία των συνεχόμενων αριθμών Fibonacci συγκλίνει στην σταθερή τιμή 1,618 Αυτός ο αριθμός επίσης, συμβαίνει συνεχώς στην φύση και ονομάζεται χρυσή αναλογία ή χρυσή τομή. Στους ανθρώπους χρυσή τομή είναι συνήθως αισθητικά ευχάριστη. Οι αρχιτέκτονες συνήθως σχεδιάζουν παράθυρα, δωμάτια και κτίρια, των οποίων το μήκος και το πλάτος είναι αναλογία της χρυσής τομής.

Η σειρά Fibonacci μπορεί να οριστεί αναδρομικά ως εξής:

fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( n ) = fibonacci( n - 1 ) + fibonacci( n - 2 )

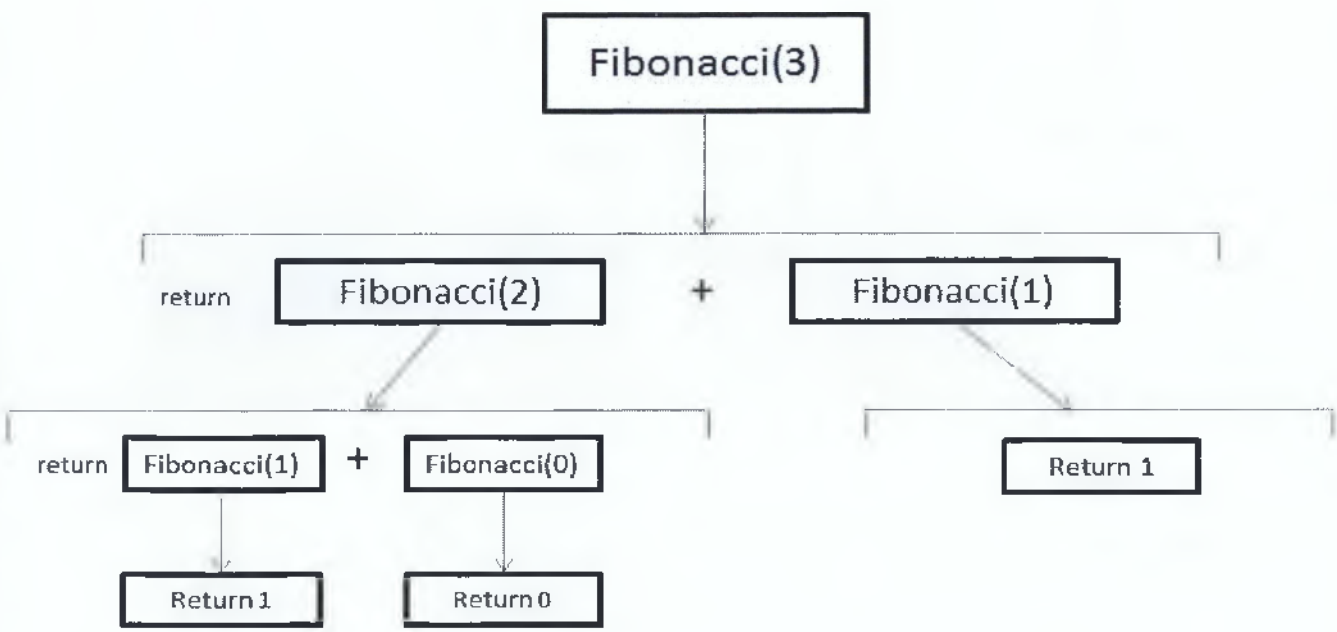
Το παρακάτω πρόγραμμα υπολογίζει το ν-οστό αριθμό Fibonacci αναδρομικά, χρησιμοποιώντας την συνάρτηση Fibonacci().

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ο χρήστης δίνει ποιόν αριθμό τις ακολουθίας Fibonacci ψάχνει και μετά καλείται η συνάρτηση Fibonacci(). Κάθε φορά που καλεί την συνάρτηση ελέγχει αν ο αριθμός που δώσαμε είναι ίσο με 0 ή 1 αν ισχύει τότε επιστρέφει το n.

Όταν όμως το n είναι μεγαλύτερο από το 1 τότε δημιουργεί δυο αναδρομικές κλήσεις, κάθε μια με ένα λίγο απλούστερο πρόβλημα από την αρχική κλήση της Fibonacci().Αυτό γίνεται μέχρι η συνάρτηση fibonacci() να έχει σαν παράμετρο 0 ή 1.Οι τιμές που επιστρέφονται από κάθε αναδρομική κλήση προσθέτονται και επιστρέφουν το αποτέλεσμα(που είναι ο ν-οστός αριθμός του Fibonacci ) στον χρήση.

Στην εικόνα φαίνεται πως θα πρέπει να υπολογίσει η συνάρτηση Fibonacci() το Fibonacci(3).



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

int fibonacci(int n);

int main()
{
   int n , apotelesma ;

   printf("Δώσε έναν αριθμό : \n");
   scanf("%d",&n);

   apotelesma = fibonacci(n);

   printf("Ο αριθμός Fibonacci είναι : %d \n", apotelesma);



 system("PAUSE");
 return 0;
}

int fibonacci(int n)
{
   if ( n==0 || n==1 )
     return n;
   else
      return fibonacci( n-1 ) + fibonacci ( n-2);
}
```

## 2.4 Μέγιστος Κοινός Διαιρέτης

**Μέγιστος κοινός διαιρέτης** δύο φυσικών αριθμών είναι ο μεγαλύτερος φυσικός αριθμός ,ο οποίος διαιρεί και τους δύο χωρίς να αφήνει υπόλοιπο.
Η ιδέα της εύρεσης του αναδρομικού αλγορίθμου για τον υπολογισμό του Μέγιστου Κοινού Διαιρέτη έχει διατυπωθεί από τον αρχαίο Έλληνα μαθηματικό Ευκλείδη.

Το πρόγραμμα του υπολογισμού του Μέγιστου Κοινού Διαιρέτη
που υπάρχει πιο κάτω χρησιμοποιεί αναδρομή για να υπολογίσει και να τυπώσει το Μέγιστο Κοινό Διαιρέτη 2 αριθμών που δίνει ο χρήστης.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ζητείται από τον χρήστη να δώσει τους 2 αριθμούς(a,b) από τους οποίους ψάχνει τον Μέγιστο Κοινό Διαιρέτη και μετά καλείται η συνάρτηση gcd().
Αν ο b αριθμός είναι ίσο με 0 τότε ο αλγόριθμος τελειώνει επιστρέφοντας το a σαν Μέγιστο Κοινό Διαιρέτη.

Αλλιώς ο αλγόριθμος προχωρεί σε μια σειρά από βήματα έτσι ώστε το αποτέλεσμα από κάθε βήμα να χρησιμοποιείται ως δεδομένο για το επόμενο βήμα. Δηλαδή στην πρώτη πράξει (b , a % b ) το υπόλοιπο του ( a % b ) θα είναι το αριθμός b στο επόμενο βήμα και α γίνεται ο αριθμός που ήταν β στο πρώτο βήμα. Πάντα το υπόλοιπο είναι μικρότερο από τον αριθμό b.
Η διαδικασία επαναλαμβάνεται έως ότου το πηλίκο να είναι ίσο με το μηδέν και ο αριθμός a επιστέφει σαν μέγιστος κοινός διαιρέτης . Όταν ο αριθμός a είναι μικρότερος από τον b τότε στο πρώτο βήμα οι αριθμοί αντιστρέφονται και το b γίνεται a.

Για παράδειγμα, για να βρεθεί ο μέγιστος κοινός διαιρέτης a = 100 και b = 22

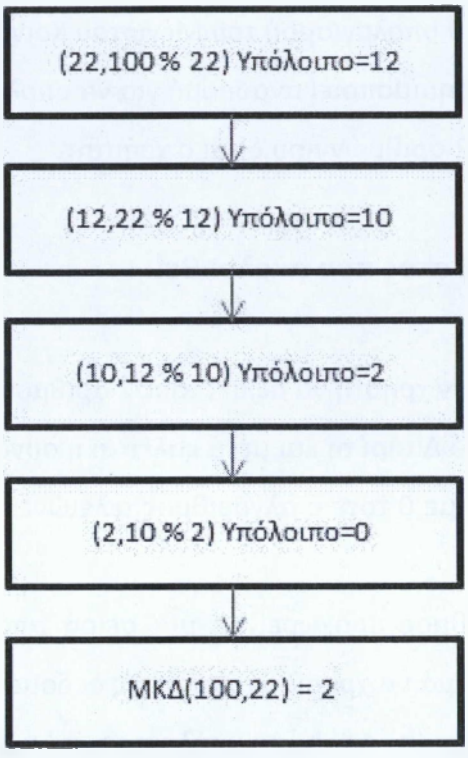
ΜΚΔ(100, 22) υπολογίζεται από τον αντίστοιχο ΜΚΔ(22, 100 % 22)=ΜΚΔ(22, 12)

Ο επόμενος ΜΚΔ υπολογίζεται από τον ΜΚΔ(12, 22 % 12) = ΜΚΔ(12, 10).

Ο επόμενος ΜΚΔ υπολογίζεται από τον ΜΚΔ (10, 12 % 10) = ΜΚΔ (10, 2) .

Ο οποίος με τη σειρά του υπολογίζεται από τον ΜΚΔ (2, 10 % 2) = ΜΚΔ (2, 0 ) .

Άρα ο ΜΚΔ του (100,22 ) είναι το 2.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

int gcd(int a, int b );

int main()
{
   int a , b ,apotelesma;
   printf("Δώσε τον πρώτο αριθμό: \n");
   scanf("%d",&a);
   printf("Δώσε τον δεύτερο αριθμό: \n");
   scanf("%d",&b);


   apotelesma = gcd(a,b);

   printf("Ο Μέγιστος Κοινός Διαιρέτης είναι : %d \n" ,apotelesma);


 system("PAUSE");
 return 0;
}

int gcd(int a, int b )
{
   if ( b==0 )
   return  a;
   else
   return  gcd( b, a % b );

}
```

## 2.5 Ελάχιστο κοινό πολλαπλάσιο

Ελάχιστο κοινό πολλαπλάσιο(ΕΚΠ) δύο ή περισσοτέρων φυσικών αριθμών είναι ο ελάχιστος (μικρότερος), φυσικός αριθμός που διαιρείται ακριβώς με καθένα εξ αυτών. Ελάχιστο κοινό πολλαπλάσιο των αριθμών α και β είναι το μικρότερο κοινό πολλαπλάσιο των α και β. Επειδή, κάθε κοινό πολλαπλάσιο είναι μεγαλύτερο ή ίσο με τους α και β, αποδεικνύεται ότι υπάρχει ελάχιστο πολλαπλάσιο τους.

Το ελάχιστο κοινό πολλαπλάσιο των α, β συμβολίζεται με ΕΚΠ(α,β).

**Εξήγηση του προγράμματος που ακολουθεί.**

Αφού δώσει ο χρήστης τους 2 αριθμούς ελέγχει άμα ο a είναι μεγαλύτερος του b και αναλόγως τρέχει την συνάρτηση lcm().Στην συνάρτηση ελέγχει άμα το πηλίκο της τοπική μεταβλητή temp με το αριθμός b είναι ίσο με 0 και άμα επίσης το πηλίκο της temp με τον αριθμό a είναι ίσο με 0 άμα αυτά ισχύουν τότε επιστρέφει την τιμή που έχει το temp εκείνη την φόρα αλλιώς εκτελεί τον εαυτό της και προσθέτη στην temp συν 1.Ανάλογα πόσες φόρες θα τρέξει η lcm() και επιστρέψει και το temp.Ας πούμε ότι έχουμε τους αριθμούς 6 και 2 .

Temp = 1  1%6≠0 &&  1%2≠0

Temp = 2  2%6≠0 &&  2%2==0

Temp = 3  3%6≠0 &&  3%2≠0

Temp = 4  4%6≠0 &&  4%2==0

Temp = 5  5%6≠0 &&  5%2≠0

Temp = 6  6%6==0 &&  6%2==0

Αφού ισχύει :

temp % b == 0 && temp % a == 0

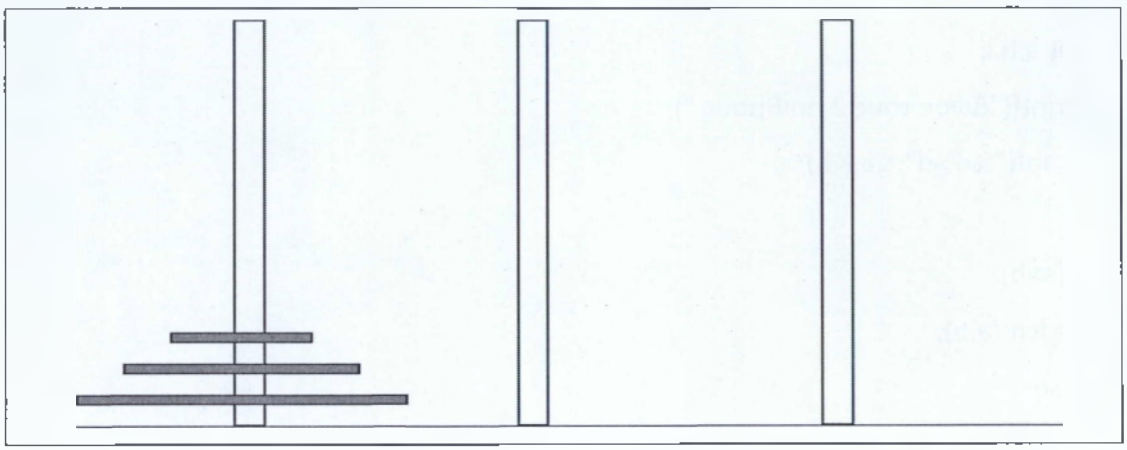
Τότε επιστρέφει το temp που είναι ίσο με 6 άρα το ελάχιστο κοινό πολλαπλάσιο του 6 και του 2 είναι το 6.

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

int lcm(int,int);

int main()
{
   int a,b,l;
   printf("Δώσε τους 2 αριθμούς ");
   scanf("%d%d",&a,&b);

   if(a>b)
   l = lcm(a,b);
   else
      l = lcm(b,a);

   printf("Ελάχιστο κοινό πολλαπλάσιο είναι: %d",l);
   system("PAUSE");
   return 0;
}

int lcm(int a,int b){

  static int temp = 1;

   if(temp % b == 0 && temp % a == 0)
      return temp;
   temp++;
   lcm(a,b);

  return temp;
}
```

## 2.6 Πύργοι του Hanoi

Ένα από τα πιο κλασικά παραδείγματα χρήσης αναδρομικών συναρτήσεων είναι οι Πύργοι του Hanoi. Υπάρχουν 3 κολόνες (Όπως φαίνεται στην εικόνα) και σε μία από αυτές είναι περασμένοι N ομόκεντροι δίσκοι διαφορετικών διαμέτρων μειούμενοι από κάτω προς τα πάνω.



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

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

Με 3 δίσκους μπορεί να λυθεί το πρόβλημα με 7 κινήσεις. Ο μικρότερος αριθμός κινήσεων που χρειάζονται για να λύσουμε το Πύργο του Hanoi είναι $2^n - 1$, όπου n είναι οι αριθμοί τον δίσκων.

### Εξήγηση του προγράμματος που ακολουθεί.

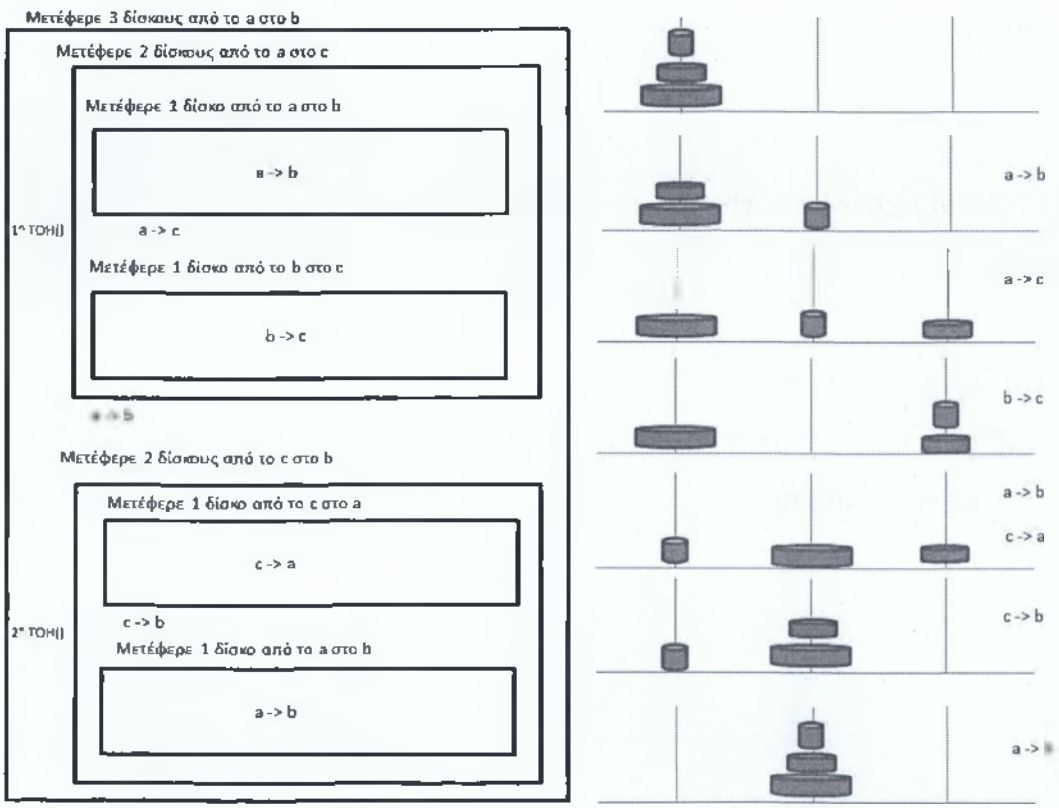
Στην αρχή δίνουμε το πόσους δίσκους θα χρησιμοποιήσουμε και μετά καλείται η συνάρτηση TOH().Αφού ισχύει ότι το n είναι μεγαλύτερο από 0 τότε καλείται η πρώτη αναδρομική κλήση της TOH() μέχρι να γίνει n ίσο με 1 .

Όταν γίνει αυτό τότε τρέχει η printf() και μετά η $2^η$ TOH() ($1^η$ TOH() το n ίσο με 1). Τώρα η $2^η$ TOH() θα εκτελεστεί την πρώτη φόρα θα εκτελέσει την printf() , θα ξανά καλέσει τον αυτό της μειώνοντας το n και αφού n γίνει ίσο με 1 θα εκτελεστεί πάλι την $1^η$ TOH() (αλλά αυτή την φόρα θα πάει ν συν 1). Αυτό θα συνεχιστεί μέχρι να τελειώσουν οι αναδρομικές κλήσεις από την κάθε TOH() και η $1^η$ TOH() να έχει το n ίσο με τον αριθμό που έδωσε ο χρήστης τότε θα τρέξει την printf() και για τελευταία φόρα και τη $2^η$ TOH() και θα τελειώσει το πρόγραμμα.

Για να καταλάβουμε καλύτερα τον αλγόριθμο θα δούμε ένα παράδειγμα με n=3.Όπως φαίνεται και στην εικόνα.

Άρα το αποτέλεσμα από τον αλγόριθμο θα είναι :

A->B A->C B->C A->B C-> A C-> B A->B



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

void TOH(int n,char x,char y,char z);

int main()
{
 int n;
 printf("\n Δώσε πόσους δίσκους θέλουμε:");
 scanf("%d",&n);
 TOH(n,'A','B','C');

 system("PAUSE");
  return 0;
}

void TOH(int n,char from ,char to ,char other) {
 if(n>0)
 {
  TOH(n-1,from,other,to);
  printf("\n%c -> %c",from,to);
  TOH(n-1,other,to,from);
 }
}
```

## 2.7 Ύψωση σε δύναμη

Η ύψωση σε δύναμη είναι μαθηματική πράξη, που συμβολίζεται ως $\alpha^n$ και περιλαμβάνει δύο αριθμούς, την βάση α και τον εκθέτη n. Όταν το n είναι θετικός ακέραιος,η ύψωση σε δύναμη αντιστοιχεί σε επαναλαμβανόμενο πολλαπλασιασμό, με άλλα λόγια είναι το γινόμενο n παραγόντων α:

$$a^n = \underbrace{a \times \cdots \times a}_{n}.$$

Κατά τον ίδιο τρόπο που ο πολλαπλασιασμός αντιστοιχεί σε επαναλαμβανόμενη πρόσθεση:

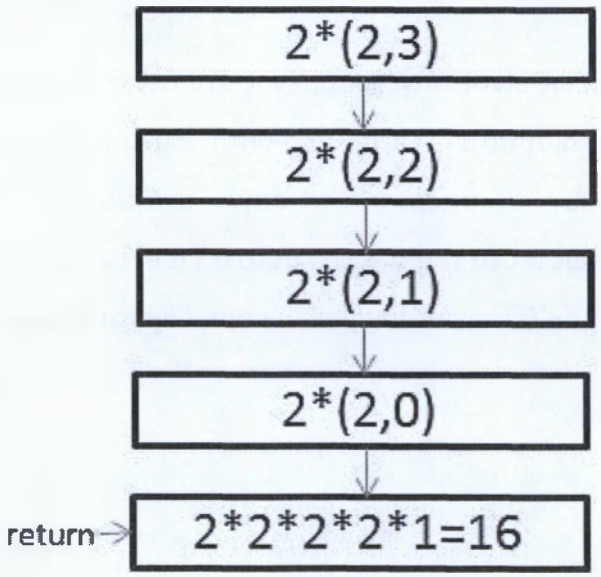
$$a \times n = \underbrace{a + \cdots + a}_{n}.$$

Ο εκθέτης συνήθως είναι επιγεγραμμένος στα δεξιά της βάσης. Η ύψωση σε δύναμη $\alpha^n$ μπορεί να διαβαστεί ως α στη n-οστή δύναμη ή απλά α στη n-οστή. Κάποιοι εκθέτες έχουν ιδιαιτερότητες στην ανάγνωση: για παράδειγμα το $\alpha^2$ διαβάζεται συνήθως α στο τετράγωνο και το $\alpha^3$, α στον κύβο.
Ο αλγόριθμος που υπολογίζει την ύψωση σε δύναμη δίνεται παρακάτω.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ο χρήστης δίνει την βάση και τον εκθέτη( ο οποίος πρέπει να είναι θετικός ).

Αφού γίνει αυτό καλείται η συνάρτηση power() . Η οποία παίρνει σαν παράμετρο δυο interger αριθμούς( την βάση και τον εκθέτη ). Ελέγχει αν ο εκθέτης είναι μεγαλύτερος από ή ίσος με το 1 αν δεν είναι(δηλαδή είναι 0 γιατί δεν μπορούμε να έχουμε αρνητικό αριθμό )επιστρέφει 1 ( $n^0 = 1$ ).Αν όμως είναι μεγαλύτερος από ίσος με 1 τότε καλεί τον εαυτό της και μειώνοντας τον εκθέτη κατά ένα κάθε φόρα μέχρι ο εκθέτης να γίνει 0 και να γυρίσει το αποτέλεσμα του συνόλου της πράξεις, όπως φαίνεται και στην εικόνα(παράδειγμα με $2^4$ ).



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

int power (int base, int exp);

int main(int argc, char *argv[])
{
 int base, exp,result;

 printf("Δώσε την βάση: ");
 scanf("%d",&base);
 printf("Δώσε  τον εκθέτη(θετικό αριθμό ): ");
 scanf("%d",&exp);
 result = power(base, exp);
 printf("Το αποτέλεσμα είναι: %d \n",result);

 system("PAUSE");

 return 0;
}

int power (int base, int exp)
{

        if(exp >= 1)
                return base * (power(base,exp - 1));
        else
                return 1;


}
```

## 2.8 Παλινδρομικός Αριθμός

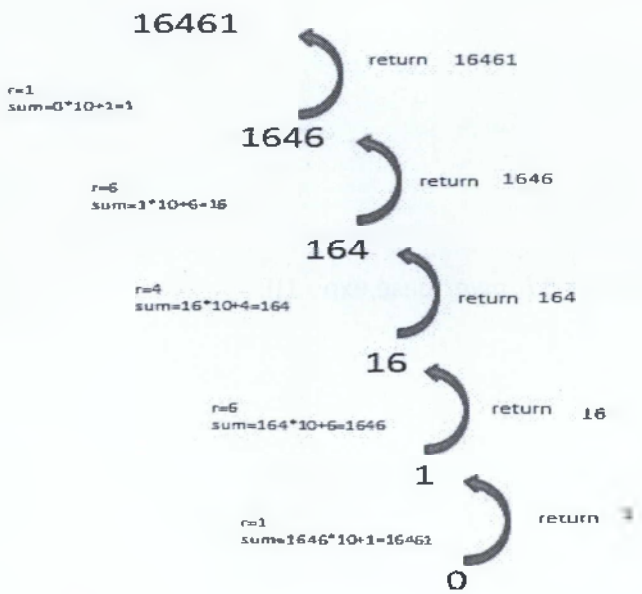
Παλινδρομικός αριθμός είναι ένας "συμμετρικός" αριθμός όπως 16461, αυτός παραμένει ο ίδιος όταν αντιστρέφονται τα ψηφία του. Θα φτιάξουμε ένα πρόγραμμα που θα μας λέει αν ένας αριθμός είναι η όχι παλινδρομικός. Σε μικρούς αριθμούς όπως το 11 η το 22 φαίνεται με το μάτι αν όμως ο αριθμός είναι πολύ μεγάλος θέλουμε πολύ χρόνο για να το διαπιστώσουμε.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή δίνουμε έναν αριθμό και καλείται η συνάρτηση Palindrome(),υπάρχουν δυο τοπικές μεταβλητές r και sum, από της οποίες η r κρατάει κάθε φόρα το πηλίκο του αριθμού και η sum το αποτέλεσμα της πράξεις για να το επιστρέψει στο τέλος της κλήσεις της συνάρτησης rev(). Αφού ελέγξει ότι ο αριθμός δεν είναι 0 παίρνουν τις τιμές οι r και sum και κάθε φόρα που τρέχει η αναδρομική συνάρτηση Palidrome() ο αριθμός που δώσαμε διαιρείται με το 10 επομένως σε κάθε κλήση της αναδρομικής συνάρτησης ο αριθμός είναι διαφορετικός.

Όταν γίνει 0 τότε γίνεται η αναδρομή και επιστρέφει το τελικό sum .

Ας δούμε ένα παράδειγμα με τον αριθμό 16461 όπως φαίνεται στην εικόνα.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

int Palindrome(int);
int main(){
   int num,sum;

   printf("Δώσε έναν αριθμό: ");
   scanf("%d",&num);

   sum = Palindrome(num);

   if(num==sum)
      printf("%d είναι παλινδρομικός \n",num);
   else
   printf("%d δεν είναι παλινδρομικός \n",num);
   system("PAUSE");
   return 0;
}

int Palindrome(int num){

   static int sum=0,r;

   if(num!=0){
      r=num%10;
      sum=sum*10+r;
      Palindrome(num/10);
   }

   return sum;
}
```

## 2.9 Παλινδρομική Λέξη

Παλινδρομική λέξη είναι η λέξη που μπορεί να διαβαστεί και αντίστροφα. Θα φτιάξουμε ένα πρόγραμμα που μας λέει άμα μια λέξη είναι παλινδρομική ή όχι.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ο χρήστης δίνει μια λέξη και αφού κρατήσουμε το μέγεθος της λέξης σε μια μεταβλητή length καλείται η συνάρτηση isPalidrome() με ορίσματα τον πίνακα str[] και το μέγεθος της λέξης length.Ελέγχει άμα το length είναι ίσο με 0 αν όχι πάει στην επόμενη if και ελέγχει την πρώτη φόρα αν το πρώτο και το τελευταίο στοιχείο του πίνακα είναι ίδια άμα ισχύει τότε ξανακαλεί τον εαυτό της για το δεύτερο και προτελευταίο στοιχείο του πίνακα δηλ. κάθε φόρα αυξάνει συν 1 την αρχή και μειώνει κατά 1 το τελευταίο στοιχείο του πίνακα. Άμα δεν ισχύει ότι κάποιό από αυτά τα στοιχεία είναι ίδια τότε επιστρέφει 0 άρα δεν είναι παλινδρομική αν όμως ισχύει μέχρι το length να γίνει 0 ή μικρότερο του 0 τότε επιστρέφει 1 άρα η λέξη είναι παλινδρομική, ας δούμε ένα παράδειγμα.
Π.χ. ότι έχουμε την λέξη αλλα(length =4 ).

Ελέγχει στην αρχή αν το πρώτο στοιχείο του πίνακα είναι ίδιο με το τελευταίο ( δηλ. α == α) ισχύει άρα προσθέτη 1 στο str(δηλ. πάμε στο δεύτερο στοιχείο)και μειώνει κατά 2 το length ( δηλ. πάει στο προτελευταίο στοιχείο) το length είναι ίσο με 2 πλέον επομένως συνεχίζει μέσα στην if ελέγχει άμα το $2^{ο}$ και το προτελευταίο στοιχείο είναι ίδια ( λ == λ ) και αφού ισχύει κάνει την ίδια διαδικασία με πριν και το length γίνεται 0 άρα σταματάει η συνάρτηση και επιστρέφει 1.Άρα η λέξη είναι παλινδρομική.

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

int isPalindrome (char str[],int  length);

int main ()
{
   int result;
   char str[256];
   printf ("\n Δώσε την λέξη: \n");
   gets (str);
int length = strlen (str);
   result = isPalindrome (str, length);
   if (result==1)
      printf ("\n Η λέξη είναι παλινδρομική. \n");
   else
      printf ("\n Η λέξη δεν είναι παλινδρομική \n");

   system("PAUSE");
   return 1;
}

int isPalindrome (char str[],int  length)
{

   if (length<=0)
      return 1;
   if (str[0] == str[length-1])
   {
      return isPalindrome (str+1, length-2);
   }
   else return 0;
}
```

## 2.10 Αντιστροφή Αριθμού

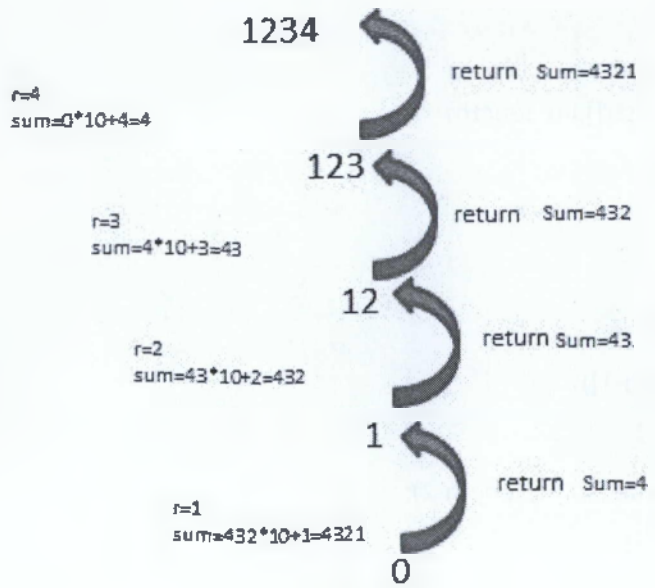
Για να κατανοήσουμε καλύτερα την αναδρομή θα φτιάξουμε ένα πρόγραμμα που παίρνει έναν αριθμό και τον αντιστρέφη.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή δίνουμε έναν αριθμό και καλείται η συνάρτηση rev(), υπάρχουν δυο τοπικές μεταβλητές r και sum οι οποίες η r κρατάει κάθε φόρα το πηλίκο του αριθμού και η sum το αποτέλεσμα της πράξεις για να το επιστρέψει στο τέλος της κλήσεις της συνάρτησης rev() και τέλος καλεί τον εαυτό της η rev() μέχρι να μην ισχύει ότι num μεγαλύτερο του 0.

Κάθε φόρα ο αριθμός που δώσαμε διαιρείται με το 10 επομένως σε κάθε κλήση της αναδρομικής συνάρτησης ο αριθμός είναι διαφορετικός. Όταν γίνει 0 τότε γίνεται η αναδρομή και επιστρέφει το τελικό sum .

Ας δούμε ένα παράδειγμα με τον αριθμό 1234 όπως φαίνεται στην εικόνα.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

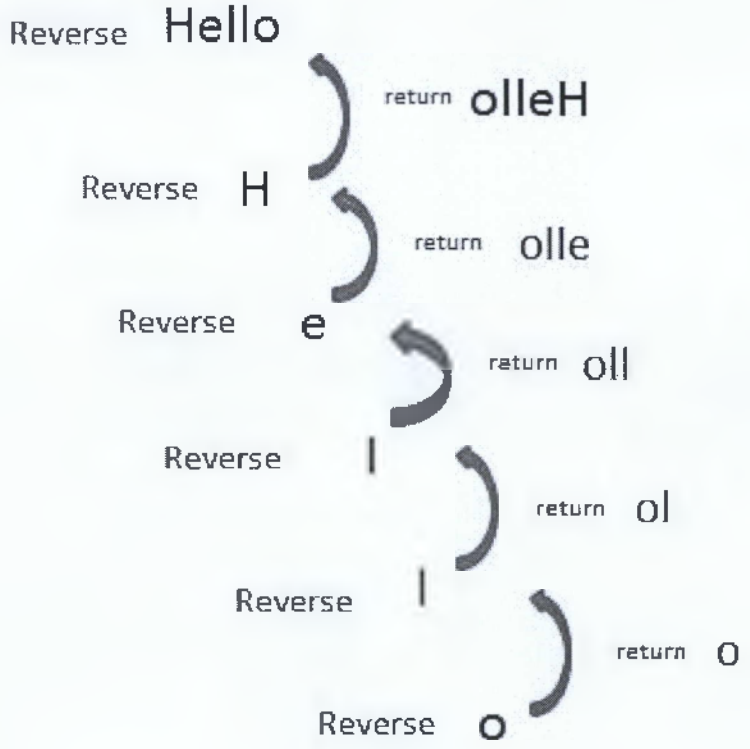
```
#include<stdio.h>

int main(){
   int num,reverse;

   printf("Δώσε τον αριθμό: ");
   scanf("%d",&num);

   reverse=rev(num);
   printf("Ο αντίστροφα ο αριθμός: %d",reverse);
   system("PAUSE");
   return 0;
}

int rev(int num){
   static sum,r;

   if(num>0){
      r=num%10;
      sum=sum*10+r;
      rev(num/10);
   }
   else
      return 0;

   return sum;
}
```

## 2.11 Αντιστροφή Λέξης

Αφού είδαμε πως γίνεται ένας αριθμός να αντιστραφεί θα φτιάξουμε και ένα πρόγραμμα όπου θα παίρνει μια λέξη αυτή την φόρα και θα την αντιστρέφει.

**Εξήγηση του προγράμματος που ακολουθεί.**

Αφού δώσουμε την λέξη που θέλουμε καλείται η συνάρτηση Reverse() .
Η συνάρτηση διαβάζει ένα ένα τα γράμματα καλώντας την αναδρομική συνάρτηση Reverse() μόλις τελειώσει με όλη την λέξη κάνει την αναδρομή και αποθηκεύει στον πίνακα rev[].Η αναδρομική κλήση με την λέξη Hello φαίνεται πιο καλά στην παρακάτω εικόνα.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>
#define MAX 100
char* getReverse(char[]);

int main(){

   char str[MAX],*rev;

   printf("Δώσε την λέξη: ");
   scanf("%s",str);

   rev = Reverse(str);

   printf("Αντίστροφα είναι: %s \n",rev);
   system("PAUSE");
   return 0;
}

char* Reverse(char str[]){

   static int i=0;
   static char rev[MAX];

   if(*str){
      Reverse(str+1);
      rev[i++] = *str;
   }

   return rev;
}
```

## 2.12 Άθροισμα Ν πρώτον αριθμών

Θα γράψουμε έναν αλγόριθμο που υπολογίζει το άθροισμα τον n πρώτον θετικών αριθμών.Θετικοί αριθμοί είναι το 1,2,3,4...n. Άρα το άθροισμα των πρώτον 5 θετικών αριθμών είναι :

Σ = 1+2+3+4+5 = 15

Άρα για n = 5 το άθροισμα είναι 15.

Ένας αναδρομικός ορισμός για άθροισμα Ν αριθμών

συνάρτησης γίνεται παρατηρώντας την παρακάτω σχέση:

1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]

Αυτό που μόλις υπολογίσαμε θα κάνει ο αναδρομικός αλγόριθμος που έχουμε γράψει παρακάτω.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ο χρήστης δίνει ποιού αριθμού το άθροισμα θέλουμε να υπολογίσουμε στην μεταβλητή n.Μετά καλείτε η συνάρτηση sum() που μας δίνει το αποτέλεσμα στην μεταβλητή add.Όταν καλείτε η συνάρτηση sum() ελέγχει αν το n είναι ίσο με 0 με μια δομή επιλογής *if* αν είναι τότε επιστρέφει το n αλλιώς μας επιστρέφει το άθροισμα n . Για να γίνει κατανοητή η λειτουργία της αναδρομικής συνάρτησης *sum()* .

Θα υπολογίσουμε το άθροισμα 5 σαν n.θα καλέσουμε την συνάρτηση *sum()* με όρισμα τον αριθμό 5.Και αφού το n δεν είναι 0 Θα εκτελεστεί η τελευταία γραμμή τις συνάρτησης :

```
return    5+sum(5-1);
```

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

 int sum(int n);

int main(int argc, char *argv[])
{

 int n,add;

    printf("Δώσε έναν ακέραιο αριθμό:\n");
    scanf("%d",&n);

 add=sum(n);
    printf("Το σύνολο είναι : %d",add);
    printf("\n");


 system("PAUSE");
 return 0;
}

 int sum(int n)
{
      if(n==0)
          return n;
      else
         return n+sum(n-1);
}
```

Στην εντολή αυτή γίνεται κλήση της ίδιας της συνάρτησης sum() , στην οποία ανήκει η εντολή με όρισμα τον αριθμό n-1 . Άρα για να υπολογίσουμε το άθροισμα του 5 πρέπει να υπολογίσουμε πρώτα το άθροισμα του 5-1, μετά σαν n θα έχουμε το 4.Άρα κάθε φορά το n θα μειώνεται κατά 1μέχρι η συνάρτηση να κληθεί με n = 0 , τότε εκτελείται η εντολή :

*return n;*

Τα αποτελέσματα από κάθε φορά που έτρεξε η συνάρτηση sum(n-1) από τους αριθμούς 5 ώς 1 προθέτονται και το άθροισμα τους τυπώνεται στον χρήστη.
Στην εικόνα φαίνεται η λογική των διαδοχικών αναδρομικών κλήσεων και η επιστροφή των αποτελεσμάτων.

```
sum(5)
 =5+sum(4)
 =5+4+sum(3)
 =5+4+3+sum(2)
 =5+4+3+2+sum(1)
 =5+4+3+2+1+sum(0)
 =5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
Άρα sum(5)=15
```

## 2.13 Τρίγωνο Pascal

Στα μαθηματικά, το τρίγωνο του Pascal είναι μία τριγωνική γεωμετρική διάταξη των δυωνυμικών συντελεστών, ονομάστηκε έτσι προς τιμήν του μαθηματικού Blaise Pascal.

Οι σειρές στο τρίγωνο του Pascal αριθμούνται ξεκινώντας από την γραμμή 0, και οι αριθμοί κάθε σειράς είναι συνήθως σχετικοί με τις διπλανές τους. Μια απλή κατασκευή του τριγώνου γίνεται με τον ακόλουθο τρόπο. Στην σειρά 0 γράφεται μόνο ο αριθμός 1. Μετά, για την κατασκευή των στοιχείων των ακόλουθων σειρών προστίθεται ο αριθμός που βρίσκεται αμέσως από πάνω και αριστερά με τον αριθμό αμέσως από πάνω και δεξιά. Αν οποιοσδήποτε από τους αριθμούς δεξιά ή αριστερά δεν υπάρχει, υποκαθίσταται με μηδέν.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ζητείται από τον χρήστη να δώσει πόσες σειρές θέλει να είναι το τρίγωνο που θέλει να φτιάξει και αφού δώσει έναν αριθμό ξεκινάει μια δομή επανάληψης for () .Ας πούμε ότι δώσαμε το 6 σαν αριθμό σειρών ( δηλ. num = 6 ).θα ξεκινήσει να τρέχει η for() όπου εκεί υπάρχει μια συνάρτηση space() και ανάλογα σε πιά εκτέλεση της for() είναι αφήνει και τα ανάλογα κενά (δηλ. με num=6 θα αφήσει 6 κενά).Μετά καλείται άλλη μια for() μέσα στην for() που εκτελείται η οποία θα εκτελεστή τόσες φορές όσες είναι το i.
Μέσα σε αυτή την for() εκτελείται μια συνάρτηση pascal() με όρισμα το i και το j .
Στην συνάρτηση pascal() αφού ελέγξει τις if () τότε επιστρέφει την ανάλογη τιμή αλλιώς καλείται η διπλή αναδρομική κλήση της συνάρτησης

(pascal(row-1,column-1)+pascal(row-1,column));

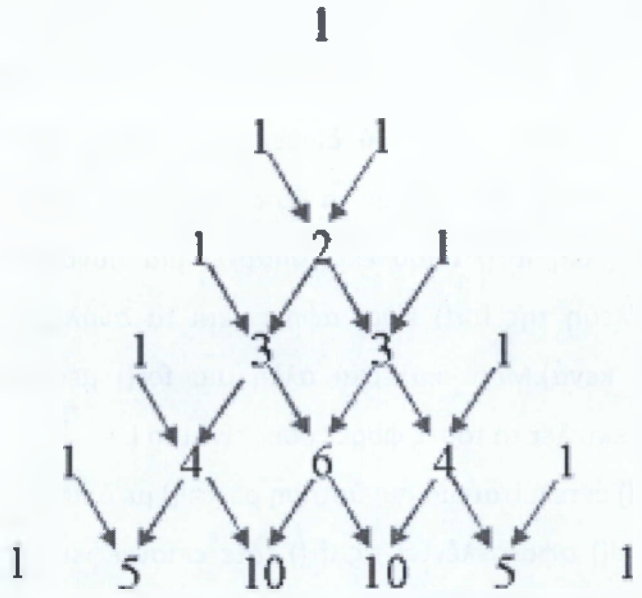
Όταν τελειώσει με τις δυο αυτές κλήσεις κάνει αναδρομή ,τις προσθέτει και γυρνάει το αποτέλεσμα. Αφήνει τρία κενά και γυρνάει στην πρώτη for() με i συν 1 όμως αυτή την φορά.

Αυτή η διαδικασία θα γίνεται μέχρι να τελειώσει η πρώτη for() να τελειώσει. Και το τρίγωνο του Pascal θα έχει φτιαχτεί όπως φαίνεται και στην εικόνα.

Για να δώσουμε τον αριθμό 3 στην 4 σειρά έπρεπε να προσθέσουμε τον 1( pascal(3,1) ) με τον 2(pascal(3,2)) από την τρίτη σειρά η συνάρτηση pascal() θα έχει ορίσμα pascal(4,2) που άμα καλέσουμε την :

(pascal(row-1,column-1)+pascal(row-1,column));

Θα δούμε ότι στην ουσία είναι pascal(3,1) + pascal(3,2) δηλαδή οι αριθμοί 1 και 2 από την από πάνω σειρά.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

int pascal(int,int);
void space(int,int);

int main(int argc, char *argv[])
{
int num,i,j;
printf("\n Δώσε το νούμερο των σειρών: ");
scanf("%d",&num);
for(i=1;i<=num;i++)
{
space(num-i,3);
for(j=1;j<=i;j++)
{
printf("%3d",pascal(i,j));
space(1,3);
}
printf("\n");
}
 system("PAUSE");
  return 0;
}
```

```
int pascal(int row,int column)
{
if(column==0)
return 0;
else if(row==1&&column==1)
return 1;
else if(column>row)
return 0;
else
return (pascal(row-1,column-1)+pascal(row-1,column));
}

void space(int num,int mul)
{
int i;
num*=mul;
for(i=0;i<num;i++)
printf(" ");


}
```

## 2.14 Μετατροπή από δεκαδικό σε δυαδικό

Γενικά όσοι ασχολούνται με υπολογιστές χρειάζεται να ξέρουν να μετατρέπουν έναν αριθμό από δεκαδικό σύστημα σε δυαδικό, μιας και οι υπολογιστές καταλαβαίνουν 0 και 1 .Για αυτό τον λόγο θα φτιάξουμε ένα πρόγραμμα που μετατρέπει έναν δεκαδικό αριθμό σε δυαδικό .

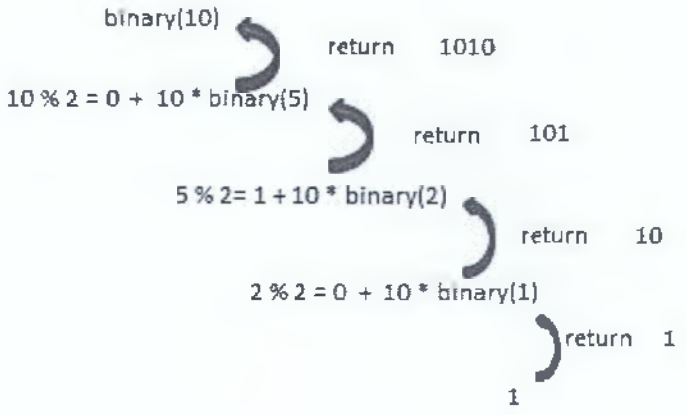
**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή αφού δώσουμε τον αριθμό που θέλουμε να μετατρέψουμε από δεκαδικό σε δυαδικό καλείται η συνάρτηση binary().Ελέγχει άμα είναι 0 αν όχι τότε καλείται η :

return (num % 2) + 10 * binary(num / 2);

Εδώ καλείται αναδρομικά η συνάρτηση binary() και κάθε φορά το όρισμα ( δηλ. ο αριθμός που δώσαμε ) διαιρείται με το 2 άρα κάθε φόρα που θα καλείται θα έχει διαφορετικό όρισμα μέχρι να γίνει 1.Τότε αρχίζει η αναδρομή και γίνονται οι πράξεις .

Ας πούμε ότι έχουμε να μετατρέψουμε τον αριθμό 10 στην πρώτη εκτέλεση θα είναι 10 και μετά θα αρχίσει να διαιρείται με το 2(10/2=5 , 5/2=2 , 2/2 = 1 ).
Στην παρακάτω εικόνα φαίνεται πιο καλά η διαδικασία.



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include <stdio.h>

int binary(int);

int main()
{
  int num, bin;

  printf("Δώσε τον δεκαδικό αριθμό: ");
  scanf("%d", &num);
  bin = binary(num);
  printf("Ο δυαδικός αριθμός του %d είναι %d\n", num, bin);
  system ("PAUSE");
}

int binary(int num)
{
   if (num == 0)
   {
      return 0;
   }
   else
   {
      return (num % 2) + 10 * binary(num / 2);
   }
}
```

## 2.15 Πόσα νούμερα έχει ένας αριθμός

Θα φτιάξουμε ένα πρόγραμμα που θα μετράει από πόσα νούμερα απαρτίζεται ένας αριθμός, σε μικρούς αριθμούς είναι εύκολο το κάνουμε και με το μάτι αλλά σε μεγάλους μας παίρνει πολύ χρόνο.

**Εξήγηση του προγράμματος που ακολουθεί.**

Στην αρχή ο χρήστης δίνει τον αριθμό και καλείται η συνάρτηση countDigits() ελέγχει άμα ο αριθμός είναι ίσο με 0 αν ναι επιστρέφει 0 αλλιώς προσθέτη συν 1 στην μεταβλητή count για κάθε φόρα που ο αριθμός δεν θα είναι 0.Και μετά καλεί τον εαυτό της διαιρώντας τον αρχικό αριθμό με το 10 αυτό θα γίνεται μέχρι ο αριθμός όταν τον διαιρέσουμε να μας δώσει 0 κόμμα κάτι. Όταν συμβεί αυτό τότε επιστρέφει το count και ανάλογα πόσες φόρες έτρεξε η if τόσο θα είναι και το count άρα και το σύνολο που απαρτίζεται ο αρχικός αριθμός που δώσαμε.

Για παράδειγμα ο αριθμός 12345.

12345/10=1234 count=1

1234/10=123 count=2

123/10=12 count=3

12/10=1 count=4

1/10=0,1 εδώ σταματάει η συνάρτηση και γυρνάει το count=5.

Άρα ο αριθμός 12345 αποτελείται από 5 στοιχεία.

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

int countDigits(num);
int main(){
 int num,count;

 printf("Δώσε έναν αριθμό: ");
 scanf("%d",&num);

 count = countDigits(num);

 printf("Το σύνολο στοιχείων:  %d",count);
 system("PAUSE");
 return 0;
}

int countDigits(int num){
   static int count=0;

    if(num!=0){
        count++;
       countDigits(num/10);
   }

   return count;
}
```

## 2.16 Πολλαπλασιασμός 2 ακέραιων αριθμών

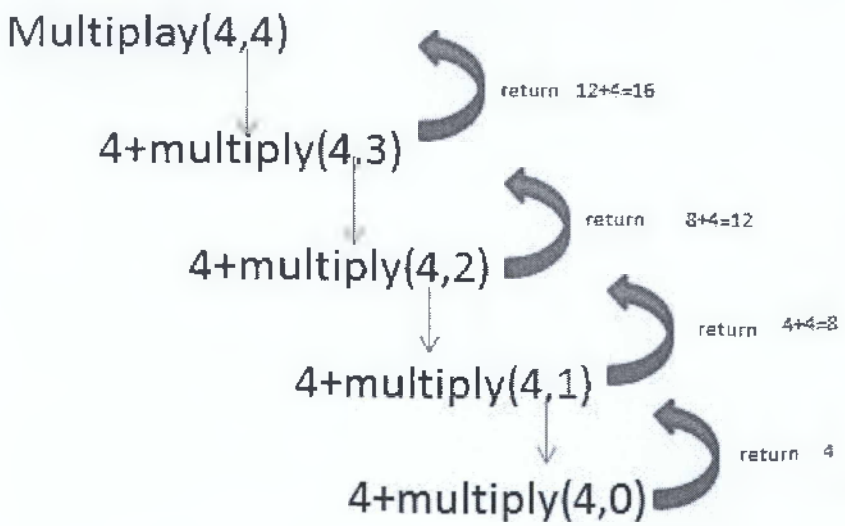
Ένα ακόμα πρόγραμμα που θα μας βοηθήσει να καταλάβουμε την αναδρομή είναι ο πολλαπλασιασμός 2 αριθμών.

**Εξήγηση του προγράμματος που ακολουθεί.**

Αφού δώσουμε τους 2 αριθμούς που θέλουμε να πολλαπλασιάσουμε καλείται η συνάρτηση multiply() με ορίσματα τους 2 αριθμούς που δώσαμε.
Ελέγχει το b αν είναι ίσο με το 0 αν καλεί τον εαυτό της,

return a + multiply(a, b - 1);

εδώ κάθε φόρα που δεν είναι το b ίσο με 0 θα καλεί την multiply() μειώνοντας το b κατά 1 κάθε φόρα μέχρι να γίνει ίσο με 0.Όταν αυτό συμβεί τότε γίνεται αναδρομή ας κάνουμε ένα παράδειγμα για να το καταλάβουμε καλύτερα.
Ας πούμε ότι έχουμε το a=4 b=4 θα γίνει όπως φαίνεται στην εικόνα.




Άρα θα μας επιστρέψει Το αποτέλεσμα του πολλαπλασιασμού είναι : 16

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

int multiply(int,int);

int main(){

   int a,b,product;
   printf("Δώσε τους 2 αριθμους: ");
   scanf("%d%d",&a,&b);

   product = multiply(a,b);

   printf("Το αποτέλεσμα του πολλαπλασιασμού είναι : %d",product);
   system("PAUSE");
   return 0;
}

int multiply(int a,int b)
{
  if (b == 0)
        return 0;

return a + multiply(a, b - 1);
}
```

# ΚΕΦΑΛΑΙΟ 3°

# Αναδρομικοί Αλγόριθμοι Αναζήτησης & Ταξινόμησης

## 3.1 Αναζήτηση

Προβλήματα αναζήτησης εμφανίζονται παντού από την καθημερινότητά μας μέχρι τις πιο εξειδικευμένες εφαρμογές. Υπάρχουν πολλοί μέθοδοι αναζήτησης αλλά στο παρόν θα ασχοληθούμε μόνο με αναδρομικές μεθόδους. **Αναζήτηση** (search) είναι η προσπάθεια εύρεσης ενός στοιχείου σε μία δομή δεδομένων, όπως για παράδειγμα σε ένα πίνακα ή σε ένα δέντρο.Γενικώς το πρόβλημα της αναζήτησης ως προς το είδος επίλυσης είναι ένα πρόβλημα απόφασης μιας και μετά την διαδικασία εύρεσης απαντούμε καταφατικά αν το βρήκαμε ή αρνητικά αν δεν το βρήκαμε.

Οι διάφορες μέθοδοι αναζήτησης εξαρτώνται κυρίως από το αν η δομή δεδομένων είναι διατεταγμένη ή όχι. Μια άλλη παράμετρος είναι, αν η δομή περιέχει μοναδικά στοιχεία ή όχι. Αναζήτηση μπορεί να πραγματοποιηθεί σε αριθμητικά και σε αλφαριθμητικά στοιχεία. 'Οπως αναφέρεται στο βιβλίο του Χ. Κοίλιας, 1993 [10].

## 3.2 Δυαδική Αναζήτηση

Η δυαδική αναζήτηση είναι ένας τρόπος από τους πολλούς που μπορεί να γίνει σε έναν ταξινομημένο πίνακα αυτό που τον κάνει καλό είναι ότι η πολυπλοκότητα του είναι λογαριθμική ( O(n)=logN ) με καλύτερη περίπτωση το στοιχείο που ψάχνουμε να βρίσκεται στην πρώτη θέση του πίνακα ( άρα για O(1)=1 απαιτείται μόνο μια σύγκριση ) και η χειρότερη περίπτωση προκύπτει αν το ζητούμενο είναι στην τελευταία θέση του πίνακα (O(n)=logN όπου n τα στοιχεία του πίνακα).

Η Δυαδική αναζήτηση συγκρίνει το στοιχείο που βρίσκεται στην μέση του πίνακα με το στοιχείο αναζήτησης. Αν δεν προκύψει ισότητα απορρίπτεται το ένα από τα δυο τμήματα του πίνακα και συνεχίζει την αναζήτηση στο άλλο με τον ίδιο τρόπο. Απορρίπτοντας λοιπόν τα τμήματα στα οποία δεν υπάρχει η δυνατότητα ύπαρξης του στοιχείου, καταλήγει σύντομα στην θέση, που υπάρχει το ζητούμενο στοιχείο. Αλλιώς αν δεν το βρει τερματίζει με αποτυχία.

Παρακάτω δίνεται η εξήγηση του αλγόριθμου που έχει υλοποιηθεί σε γλώσσα C καθώς και ένα παράδειγμα.

Στο πρόγραμμα λοιπόν αφού έχουμε φτιάξει έναν ταξινομημένο πίνακα καλείται η συνάρτηση binarysearch() με 4 μεταβλητές(α[] τον ταξινομημένο πίνακα , n το στοιχείο που ψάχνουμε , arxi και telos(no-1)σαν δείκτες των τιμών των ακραίων του πίνακα.

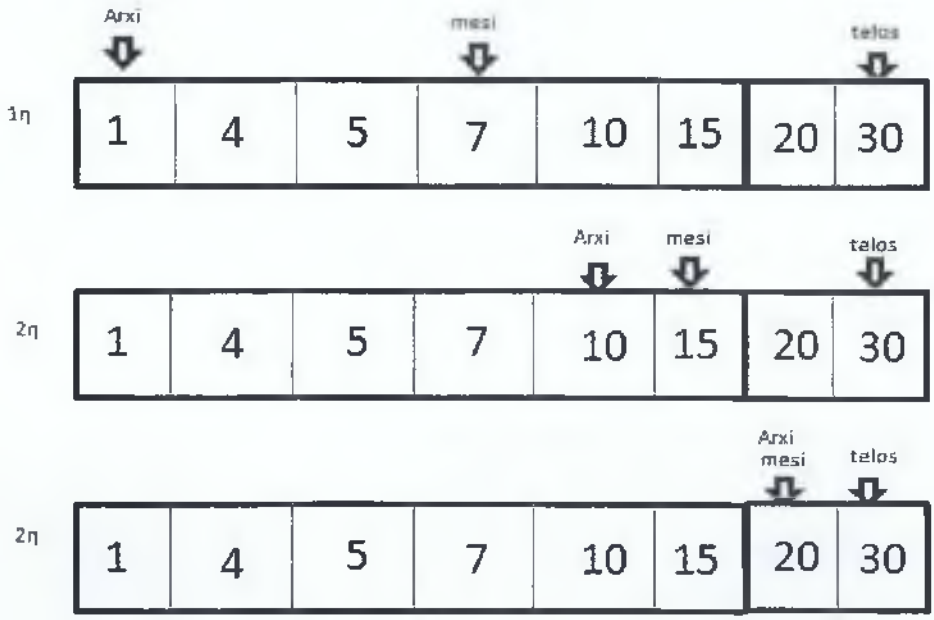
Αν η μεταβλητή arxi είναι μεγαλύτερη από την telos τότε μας επιστρέφει (Ο αριθμός δεν βρέθηκε) και το πρόγραμμα σταματάει. Αλλιώς , οι ακραίες θέσεις arxi και telos διαιρούνται σε δυο σχεδόν ίσα μέρη. Με την μεταβλητή mesi να παίρνει τον αριθμό που βρίσκεται στην μεσαία θέση. Και αριστερά από την μεσαία θέση θα είναι τα μικρότερα στοιχεία ενώ δεξιά τα μεγαλύτερα .Μετά ελέγχουμε αν ο αριθμός που ψάχνουμε είναι ίσος με τον αριθμός στην μεταβλητή mesi αν είναι τότε ο αλγόριθμος τερματίζει και εμφανίζει ότι το στοιχείο βρέθηκε στην θέση mesi+1( όπου το νούμερο της θέσης του πίνακα ).

Αν όμως το στοιχείο που ψάχνουμε είναι μικρότερο από την μέση του πίνακα ξανά καλείται η συνάρτηση binarysearch() Και πλέον η αναζήτηση γίνεται στο αριστερό τμήμα του πίνακα με telos = mesi-1 πλέον. Αν όμως είναι μεγαλύτερο

από την μέση του πίνακα ξανά καλείται η συνάρτηση binarysearch() Και η αναζήτηση γίνεται στο δεξιό τμήμα του πίνακα με arxi=mesi+1. Αυτά τα βήματα επαναλαμβάνονται μέχρι το ο δείκτης telos να είναι μικρότερος ή ίσος με τον δείκτη arxi.

Στο παρακάτω παράδειγμα φαίνεται η διαδικασία.

Παράδειγμα ότι θέλουμε να βρούμε το 20 σε έναν ταξινομημένο πίνακα.




Στην 1η φάση η arxi = 0 (θέση του πίνακα) η mesi=(0+7)/2 = 3 (θέση του πίνακα) και το telos= 7(θέση του πίνακα).Και αφού το στοιχείο που ψάχνουμε είναι μεγαλύτερο από την mesi τότε στην επόμενη φάση θα συνεχίσουμε στα δεξιά του πίνακα.

Στην 2η φάση η arxi = mesi + 1 = 4 (θέση του πίνακα) telos =7(θέση του πίνακα) και mesi = (4+7)/2=5(θέση του πίνακα).Και αφού το στοιχείο που ψάχνουμε είναι μεγαλύτερο από την mesi τότε στην επόμενη φάση θα συνεχίσουμε στα δεξιά του πίνακα.

Στην 3η φάση η arxi = mesi + 1 = 6 (θέση του πίνακα) telos =7(θέση του πίνακα) και mesi = (6+7)/2=6(θέση του πίνακα).Και αφού το στοιχείο που ψάχνουμε είναι ίσο με την mesi θα μας επιστρέψει ένα μήνυμα:

Το στοιχείο βρέθηκε στην θέση : 6 (θέση του πίνακα).

**Το πρόγραμμα υλοποιημένο σε γλώσσα C.**

```
#include <stdio.h>
#include <stdlib.h>
binarysearch(int a[],int n,int arxi,int telos);

int main(int argc, char *argv[])
{
        int a[40];
        int n,no,x,apotelesma;

        printf("Δώσε πόσα στοιχεία θα έχει ο πίνακας : ");
        scanf("%d",&no);

        printf("Δώσε τα στοιχεία του πίνακα :\n");

        for(x=0;x<no;x++)
                scanf("%d",&a[x]);

        printf("Δώσε ποιον αριθμό ψάχνουμε : ");
        scanf("%d",&n);

        apotelesma = binarysearch(a,n,0,no-1);

        if (apotelesma == -1)
           printf("Ο αριθμός δεν βρέθηκε");


system("PAUSE");
return 0;
```

```
binarysearch(int a[],int n,int arxi,int telos)
        {
         int mesi;
         if (arxi > telos)
          return -1;
         mesi = (arxi + telos)/2;
         if(n == a[mesi])
          { printf("Το στοιχείο βρέθηκε στην θέση  %d\n",mesi+1);
            return 0;
          }
         if(n < a[mesi])
          { telos = mesi - 1;
            binarysearch(a,n, arxi, telos);
          }
         if(n > a[mesi])
          { arxi = mesi + 1;
            binarysearch(a,n, arxi, telos);
          }
        }
```

### 3.3 Διαίρει και Κυρίευε

Μια από τις κλασικές μεθόδους επίλυσης προβλημάτων είναι η λεγάμενη «διαιρεί και κυρίευε» (divide and conquer). Η μέθοδος οφείλει την ονομασία της στους αρχαίους Ρωμαίους πολιτικούς .

Πολλοί αλγόριθμοι είναι αναδρομικοί, διότι επιλύουν το πρόβλημα καλώντας τον εαυτό τους μια ή περισσότερες φορές. Αλγόριθμοι αυτής της κατηγορίας ακολουθούν κατά κανόνα τη μέθοδο *διαιρεί και κυρίευε*. Διαιρούν το πρόβλημα σε μικρότερα υποπροβλήματα (διαίρει) τα οποία είναι παρόμοια με αυτό και τα επιλύουν αναδρομικά (κυρίευε). Έπειτα συνδυάζουν σε ένα μόνο αποτέλεσμα τις λύσεις που προέκυψαν. Αυτή η τεχνική αποτελεί τη βάση για πολλούς αποδοτικούς αλγορίθμους, όπως είναι για παράδειγμα η ταξινόμηση με συγχώνευση και γρήγορη ταξινόμηση.

Η μέθοδος χρησιμοποιεί τη διάσπαση των δεδομένων του προβλήματος σε έναν αριθμό από μικρότερα τμήματα. Στη συνέχεια εφαρμόζει τη συνένωση των επιμέρους λύσεων με τέτοιο τρόπο, ώστε να σχηματιστεί η γενική λύση του προβλήματος. Βέβαια, σε καθένα από τα επιμέρους προβλήματα τρέπει να εφαρμοσθεί η ίδια μέθοδος αναδρομικά έως ότου καταλήξει σε προβλήματα τάξης μεγέθους 1 ή το πολύ 2, όπου σε αυτά η λύση είναι τετριμμένη. Συνοψίζοντας, τα βήματα επίλυσης ενός προβλήματος, χρησιμοποιώντας αυτή τη μέθοδο, είναι τα ακόλουθα. 'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή, 2010[1]

- *Διαιρεί (το πρόβλημα σε μικρότερα επιμέρους προβλήματα).*
- *Κυρίευε (επιλύοντας το κάθε επιμέρους πρόβλημα εφαρμόζοντας την ίδια τακτική αναδρομικά) και τέλος*
- *Συνένωσε (τις επιμέρους λύσεις σε μια καθολική η οποία θα αναφέρεται στο πρόβλημα).*

Στην συνέχεια θα δούμε δύο αλγόριθμους ταξινόμησης που βασίζονται στην τεχνική διαίρει και κυρίευε τον αλγόριθμο γρήγορης ταξινόμησης(quicksort) και τον αλγόριθμο ταξινόμησης με συγχώνευση(mergesort).

### 3.3.1 Γρήγορη ταξινόμηση

Η γρήγορη ταξινόμηση (quick sort) είναι μια μέθοδος, η οποία βασίζεται στην τεχνική διαίρει και κυρίευε. Συνεπώς, λειτουργεί με διαδοχικές διασπάσεις του αρχικού συνόλου, χρησιμοποιώντας αναδρομικό τρόπο. Τελικά, με συνενώσεις των μερών ταξινομεί το αρχικό σύνολο. Αποτελεί τη συντομότερη μέθοδος ταξινόμησης πινάκων με συγκρίσεις. Πρόκειται για τον ταχύτερο αλγόριθμο ταξινόμησης( από τους κλασικούς ).Η λειτουργία της βασίζεται στο διαχωρισμό ενός πίνακα( ή μιας λίστας ) σε δύο μέρη(διαίρει),τα οποία ταξινομεί ξεχωριστά και τα συνενώνει σε μια λίστα(κυρίευε).'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή 2010[1].

Ο διαχωρισμός γίνεται επιλέγοντας ένα στοιχείο του πίνακα (συνήθως το πρώτο ή το τελευταίο , αλλά μπορούμε και κάποιο τυχαίο) σαν διαχωριστική τιμή. Χωρίζουμε τον πίνακα σε δυο μέρη, αριστερά τους αριθμούς που έχουν μικρότερη ή ίση τιμή με τη διαχωριστική τιμή και δεξιά τους αριθμούς που είναι μεγαλύτερα.

Ταξινομούμε τα δυο μέρη και τα συνενώνουμε έτσι ώστε ο τελικός πίνακας να περιέχει αριστερά τα στοιχεία που είναι ταξινομημένα και μικρότερα από την διαχωριστική τιμή, την διαχωριστική τιμή και δεξιά τα στοιχεία που είναι ταξινομημένα και μεγαλύτερα από τη διαχωριστική τιμή.

Η πολυπλοκότητα του στην μέση και καλύτερη περίπτωση είναι O(nlog(n)).Ενώ η πολυπλοκότητα του είναι στην χειρότερη περίπτωση O(n2).

Παρακάτω δίνεται η εξήγηση του αλγόριθμου που έχει υλοποιηθεί σε γλώσσα C καθώς και ένα παράδειγμα.

Στην αρχή φτιάχνουμε έναν πίνακα και τον εμφανίζουμε πριν ταξινομηθεί και μετά καλείται η συνάρτηση quicksort() που παίρνει σαν παραμέτρους το μέγεθος του πίνακα , και 2 ακέραιους αριθμούς ο πρώτος( first ) ίσος με το 0 και ο δεύτερος( last )το μέγεθος του πίνακα μείον 1.

Στην αρχή της quicksort δηλώνονται 4 ακέραιους(pivot , j , temp , I ),η pivot θα χρησιμοποιηθεί σαν ένας δείκτης που με αυτόν θα γίνονται συγκρίσεις.
Μετά υπάρχει μια if που ελέγχει αν first < last αν αυτό ισχύει τότε δηλώνει :

pivot=first

i=first

j=last

Αυτό γίνεται για να μπορούμε με το I και j να κουνιούνται σαν δείκτες δεξιά κα αριστερά και first,last,pivot να είναι σταθερά και σαν pivot έχουμε βάλει να είναι ο πρώτος αριθμός .Μετά υπάρχει μια while που όσο ισχύει ότι i<j τότε θα γίνεται έλεγχος στην πρώτη while ελέγχει αν το x[i]<=x[pivot] και αν ισχύει και ότι i<last τότε ο δείκτης i θα πηγαίνει δεξιά για όσο δεν βρίσκει κάποιον αριθμό μεγαλύτερο του pivot αν βρει σταματάει η φτάνει μέχρι το τέλος του πίνακα και σταματάει εκεί. Όταν τελειώσει το i τότε πάει στην επόμενη while και ελέγχει αν το x[j]>x[pivot] και αν αυτό ισχύει τότε ο δείκτης j πηγαίνει αριστερά μέχρι να βρει κάποιόν αριθμό μικρότερο του pivot ή μέχρι το i>j .Όταν σταματήσουν και οι δυο δείκτες και το i<j τότε κάνουν ανταλλαγή τον στοιχείων τους το x[i] με το x[j].Αφού γίνει η ανταλλαγή των στοιχείων ξαναγυρνάει στην while και για όσο συνεχίζουν να ισχύουν οι παράμετροι για τους δείκτες i και j τότε συνεχίζουν το i να πηγαίνει δεξιά και το j αριστερά μέχρι το i>j .

Όταν το i>j τότε σταματάει η while και γίνεται ανταλλαγή των στοιχειών του x[pivot] με x[j] και πλέον έχουμε σαν διαχωρισμό του πίνακα αριστερά του pivot είναι τα μικρότερα νούμερα και δεξιά του pivot τα μεγαλύτερα νούμερα. Για το αριστερά κομμάτι τρέχει η συνάρτηση :

quicksort(x,first,j-1);

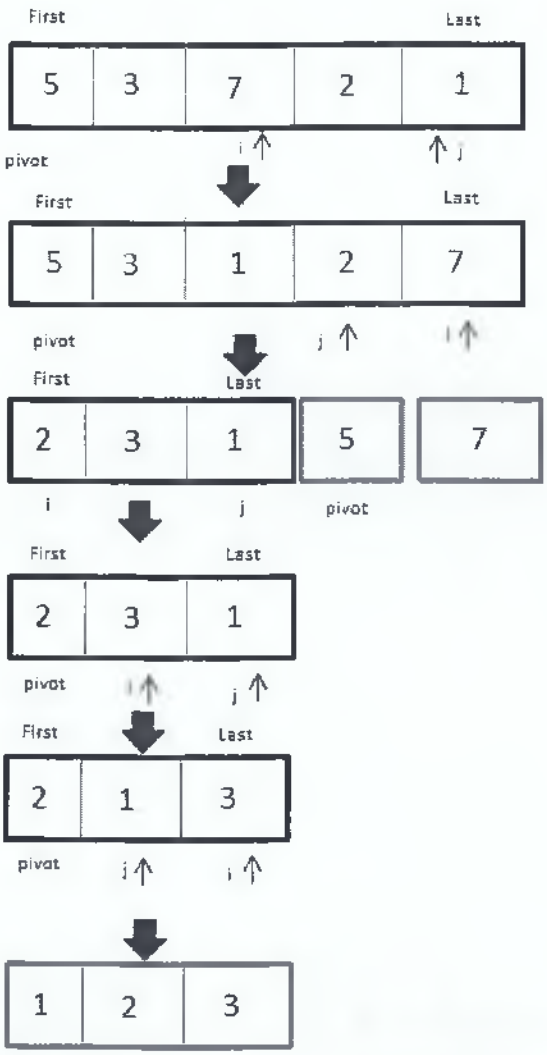
Και παίρνει σαν όρισμα last ίσο με το στοιχείο που έδειχνε το j – 1 .

Και το δεξί κομμάτι του πίνακα τρέχει η συνάρτηση :

quicksort(x,j+1,last);

Και σαν όρισμα του first το στοιχείο που έδειχνε το j+1.Αυτή η διαδικασία συνεχίζεται μέχρι το first να γίνει μεγαλύτερο του last τότε σταματάει η quicksort ο πίνακας πλέον είναι ταξινομημένος και εμφανίζεται στον χρήστη με την βοήθεια μιας εντολής επανάληψης for .

Στο παρακάτω παράδειγμα φαίνεται ποιο κατανοητά πως λειτουργεί:



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>

void quicksort(int x[200],int first,int last);

int main(){
 int x[200],size,i;

 printf("Δώσε το μέγεθος του πίνακα: ");
 scanf("%d",&size);

 printf("Δώσε  %d τα στοιχεία: ",size);
 for(i=0;i<size;i++)
  scanf("%d",&x[i]);

 printf("Μη ταξινομημένος  πίνακας: ");
 for(i=0;i<size;i++)
  printf(" %d",x[i]);

 printf("\n");


 quicksort(x,0,size-1);

 printf("Ταξινομημένος πίνακας: ");
 for(i=0;i<size;i++)
  printf(" %d",x[i]);
  system("pause");
 return 0;
}
```

```
void quicksort(int x[200],int first,int last){
   int pivot,j,temp,i;

    if(first<last){
       pivot=first;
       i=first;
       j=last;

       while(i<j){
          while(x[i]<=x[pivot]&&i<last)
             i++;
          while(x[j]>x[pivot])
             j--;
          if(i<j){
             temp=x[i];
              x[i]=x[j];
              x[j]=temp;
          }
       }

       temp=x[pivot];
       x[pivot]=x[j];
       x[j]=temp;
       quicksort(x,first,j-1);
       quicksort(x,j+1,last);

   }
}
```

### 3.3.2 Ταξινόμηση με συγχώνευση

Η μέθοδος της ταξινόμησης με συγχώνευση ( mergesort ) είναι μια μέθοδος που χρησιμοποιεί την τεχνική διαίρει και κυρίευε.

- **Διαίρει**: Διαιρεί την ακολουθία εισόδου σε δυο υποακολουθίες ίσου μήκους.
- **Κυρίευε:** Ταξινομεί τις δυο υποακολουθίες καλώντας αναδρομικά τον εαυτό της.
- **Συνδύασε:** Συγχωνεύει τις ταξινομημένες ακολουθίες χρησιμοποιώντας την συγχώνευση ταξινομημένων πινάκων.

Θεωρείται ένας αποτελεσματικός αλγόριθμος ταξινόμησης, αλλά απαιτεί το διπλάσιο αποθηκευτικό χώρο από οποιονδήποτε άλλον. 'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή 2010[1].

Το μειονέκτημα της μεθόδου είναι ότι χρησιμοποιεί ένα βοηθητικό πίνακα 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) .

Παρακάτω δίνεται η εξήγηση του αλγόριθμου που έχει υλοποιηθεί σε γλώσσα C καθώς και ένα παράδειγμα.

Αφού δώσουμε το μέγεθος του πίνακα arr[](το οποίο του έχουμε δηλώση σαν μέγεθος 50 στοιχεία με την Define )και τα στοιχεία, εμφανίσουμε τον μη ταξινομημένο πίνακα στον χρήση, και καλείται η συνάρτηση mergeSort() και σαν όρισμα παίρνει τον πίνακα arr[] δυο μεταβλητές Low ίσο με 0 και η high ίσο με n στοιχεία του πίνακα μείον 1 μιας και οι πίνακες ξεκινάνε από το 0 στην C. Μέσα στην mergeSort() δηλώνουμε μια ακέραια μεταβλητή η οποία θα παίρνει την μέση του πίνακα κάθε φόρα. Οι αναδρομικές κλήσεις τον συναρτήσεων διαιρούν τον πίνακα στην μέση η πρώτη mergeSort() το αριστερό κομμάτι και η δεύτερη το δεξιό κομμάτι του πίνακα. Η διαδικασία της αναδρομής σταματάει όταν ο κάθε υποπίνακας περιέχει ένα μόνο στοιχείο και το mid θα είναι ίσο με 0 .Ο πίνακας arr[] δεν έχει αλλάξει απλά η merge() θα ασχολείται με συγκεκριμένα κομμάτια του πίνακα κάθε φορά. Αφού λοιπόν τελειώσουν οι κλήσεις τον mergeSort() καλείται η merge().Στην αρχή θα ασχοληθεί με τα πρώτα δυο στοιχεία και το αριστερό μέρος του πίνακα , αφού συγκρίνει τα δυο στοιχεία βάζει το μικρότερο σε έναν άλλον

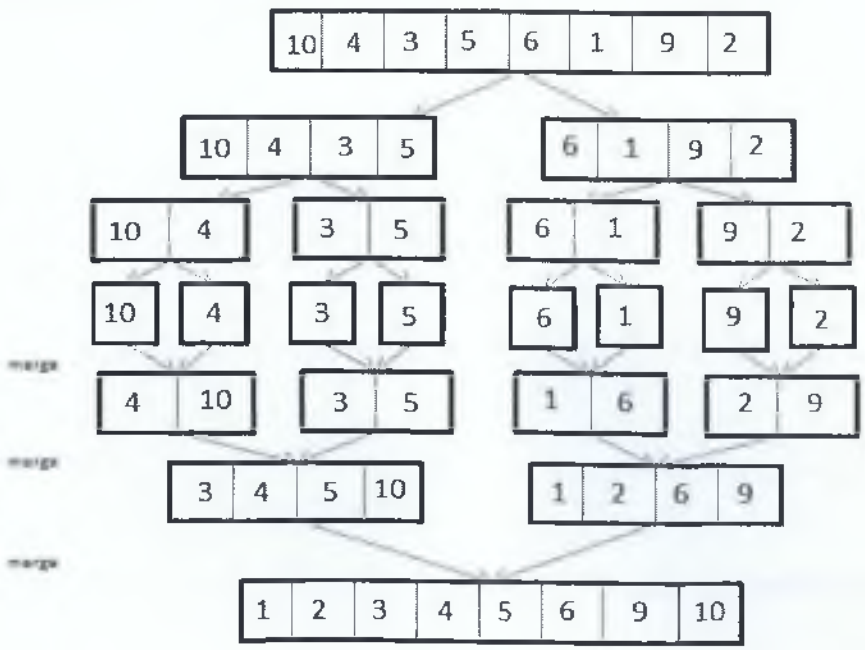
πίνακα temp[] στην αρχή του πίνακα temp[0] και το μεγαλύτερο στοιχείο στο temp[1].

Αυτό θα γίνει σταδιακά για όλα τα στοιχεία από αριστερά προς τα δεξιά και αφού τελειώσει με όλα τα μονά στοιχεία και γίνουν δυάδες μετά τις δυάδες συγχωνεύει και τις κάνει τετράδες πάλι πρώτα το αριστερό κομμάτι του πίνακα και μετά το δεξί. Και όσο η συγχώνευση εξελίσσεται συγκρίνουμε πάντα τα μικρότερα στοιχεία με τα αντίστοιχα μικρότερα του αλλού μέρους ( δηλαδή τον αριστερό υποπίνακα με τον δεξί υποπίνακα )μέχρι να μην έχουμε άλλα στοιχεία να συγκρίνουμε και η συγχώνευση να φτάσει στο τέλος δηλαδή να έχουμε τον πίνακα στο αρχικό μέγεθος. Τέλος αφού συγκρίνει όλα τα στοιχεία και τα έχει ταξινομήση και τοποθετήση στον πίνακα temp[] παίρνει τα στοιχεία από τον πίνακα temp[] και τα αντιγράφη στον πίνακα arr[] με την ίδια ακριβός σειρά (arr[0]=temp[0], arr[1]=temp[1],... arr[n]=temp[n], ) αυτό γίνεται στην τελευταία for() της συνάρτησης της merge(), μετά το πρόγραμμα εμφανίζει τον ταξινομημένο πίνακα στον χρήστη.

Αυτή η διαδικασία φαίνεται καλύτερα στο παράδειγμα.

Παράδειγμα:

Έχουμε έναν πίνακα arr=[10,4,3,5,6,1,9,2].



**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

```
#include<stdio.h>
#define MAX 50

void mergt(int arr[],int low,int mid,int high);
void mergeSort(int arr[],int low,int high);

int main(){

  int arr[MAX],i,n;

  printf("Δώσε το μέγεθος του πίνακα: ");
  scanf("%d",&n);

  printf("Δώσε τα στοιχεία του πίνακα: ");
  for(i=0;i<n;i++){
     scanf("%d",&arr[i]);
  }

  printf("Μη ταξινομημένος ο πίνακας : ");
  for(i=0;i<n;i++)
 {
     printf("%d ",arr[i]);
  }

  mergeSort(arr,0,n-1);

  printf("Ταξινομημένος ο πίνακας : ");
  for(i=0;i<n;i++)
 {
     printf("%d ",arr[i]);
```

```
    }
    system ("pause");
   return 0;
}

void mergeSort(int arr[],int low,int high){

    int mid;

    if(low<high){
        mid=(low+high)/2;
        mergeSort(arr,low,mid);
        mergeSort(arr,mid+1,high);
        merge(arr,low,mid,high);
    }
}

void merge(int arr[],int low,int mid,int high){

    int i,m,k,l,temp[MAX];

    l=low;
    i=low;
    m=mid+1;

    while((l<=mid)&&(m<=high)){

        if(arr[l]<=arr[m]){
            temp[i]=arr[l];
            l++;
        }
        else{
```

```
            temp[i]=arr[m];
            m++;
        }
        i++;
    }

    if(l>mid){
        for(k=m;k<=high;k++){
            temp[i]=arr[k];
            i++;
        }
    }
    else{
        for(k=l;k<=mid;k++){
            temp[i]=arr[k];
            i++;
        }
    }

    for(k=low;k<=high;k++){
        arr[k]=temp[k];
    }
}
```

# ΚΕΦΑΛΑΙΟ 4$^{ο}$

# Δένδρα

## 4.1 Γενικά

Η δομή δεδομένων πίνακας που χρησιμοποιήθηκε και εξετάστηκε σε προηγούμενα κεφάλαια, ως εργαλείο για την υλοποίηση αναδρομικών αλγορίθμων, είναι μία γραμμική δομή δεδομένων. Τα δένδρα που θα εξετάσουμε στο παρόν κεφάλαιο είναι μία μη γραμμική δομή δεδομένων. Η δομή του δένδρου σε γενικές γραμμές είναι η σύνδεση κόμβων με τρόπο ανάλογο ενός πραγματικού δένδρου. Η μόνη διαφορά είναι ότι μια δομή δένδρου είναι ανεστραμμένη, δηλαδή η ρίζα είναι στην κορυφή και τα κλαδιά αναπτύσσονται προς τα κάτω.

Δένδρο είναι ένα πεπερασμένο σύνολο από στοιχεία , το οποίο είτε είναι κενό, είτε περιέχει ένα στοιχείο το οποίο ονομάζεται ρίζα. Τα υπόλοιπα στοιχεία του διαχωρίζονται σε ξένα υποσύνολα, όπου το καθένα από αυτά είναι ένα δένδρο. Κάθε στοιχείο του δένδρου λέγεται κόμβος ή κορυφή. 'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή, 2010[1].

Κένο είναι το δένδρο 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 υποδένδρα σχηματίζονται αρκεί να θεωρήσουμε σαν ρίζα οποιονδήποτε κόμβο του δέντρου. Χαρακτηριστική ιδιότητα του δένδρου είναι ότι ξεκινώντας από την ρίζα μπορούμε να καταλήξουμε σε έναν οποιοδήποτε κόμβο αυτό ονομάζεται **διαδρομή.**

Στα δένδρα υπάρχει μόνο μια διαδρομή που συνδέει δυο οποιουσδήποτε κόμβους. Αν δεν υπάρχει ή υπάρχουν περισσότερες από μια διαδρομές τότε αυτό είναι γράφος.

**Υπόδενδρο** είναι κάθε δένδρο που βρίσκεται κάτω από την ρίζα του δένδρου. Με την ρίζα του υπόδενδρου να ονομάζεται **κόμβος γονέας** και τους κόμβους του υπόδενδρου **κόμβοι παιδιά.**

**Μήκος** μίας διαδρομής ονομάζεται το σύνολο των περιεχομένων ακμών (που ισούται με το πλήθος των περιεχομένων κόμβων μείον ένα).

**Ύψος** κόμβου ενός δένδρου είναι η μεγαλύτερη διαδρομή από τον κόμβο προς κάποιο φύλλο. Ύψος του δένδρου είναι το ύψος της ρίζας του. Τα φύλλα ενός δένδρου έχουν ύψος μηδέν.

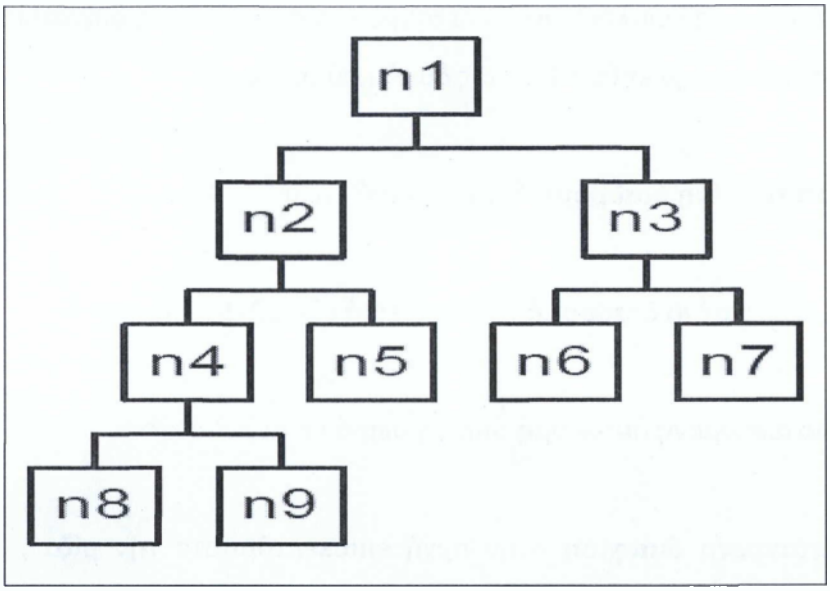
**Επίπεδο** ενός κόμβου είναι το μήκος της μοναδικής διαδρομής από την ρίζα προς αυτόν τον κόμβο. Η ρίζα κάθε δένδρου βρίσκεται στο μηδενικό επίπεδο. Ένα κενό δένδρο έχει ύψος –1.

**Βαθμός** ενός κόμβου είναι το πλήθος των παιδιών του.

Ειδικές κατηγορίες δένδρων αποτελούν τα δυαδικά δένδρα, τα τριαδικά δένδρα κλπ. Τα δυαδικά δένδρα θα μας απασχολήσουν στην επόμενη παράγραφο.

## 4.2 Δυαδικά Δένδρα

Δυαδικό Δένδρο είναι ένα δένδρο το οποίο είτε είναι κενό, είτε αποτελείται από μια ρίζα και δύο υποδένδρα(ξένα μεταξύ τους). Αναφερόμαστε στα δύο υποδένδρα ως το αριστερό και το δεξιό υπόδενδρο.Όπως φαίνεται και στην είκονα.




Οι βασικές διαφορές μεταξύ ενός δυαδικού δένδρου και ενός γενικού δένδρου είναι οι εξής:

- Ένα δυαδικό δένδρο μπορεί να είναι κενό, ενώ ένα δένδρο δεν μπορεί.

- Κάθε στοιχείο ενός δυαδικού δένδρου έχει ακριβώς δύο υποδένδρα, ενώ ενός δένδρου μπορεί να έχει οποιονδήποτε αριθμό υποδένδρων.

- Τα υποδένδρα κάθε στοιχείου ενός δυαδικού δένδρου είναι διατεταγμένα. Δηλαδή διακρίνουμε δεξιό και αριστερό υποδένδρο. Τα υποδένδρα ενός γενικού δένδρου δεν είναι διατεταγμένα.

Διάσχιση δένδρου είναι η διαδικασία επίσκεψης των κόμβων του. Εάν θεωρήσουμε τις διαδικασίες:

1. Επίσκεψη ρίζας δένδρου,
2. Επίσκεψη αριστερού υποδένδρου, και
3. Επίσκεψη δεξιού υποδένδρου

Και από αυτό έχουμε την επιλογή να διαλέξουμε πια από τις τρεις διατάξεις που θα χρησιμοποιήσουμε για την επίσκεψη ενός δυαδικού δένδρου:

- **Προδιατεταγμένη διαδρομή** όπου η σειρά είναι 1-2-3,
- **Ενδοδιατεταγμένη διαδρομή** όπου η σειρά είναι 2-1-3,
- **Μεταδιατεταγμένη διαδρομή** όπου η σειρά είναι 2-3-1.

Στην **προδιατεταγμένη διάσχιση** στην αρχή επισκεπτόμαστε την ρίζα , μετά το αριστερό υπόδενδρο και τέλος το δεξί υπόδενδρο.
Στην **ενδοδιατεταγμένη διάσχιση** στην αρχή επισκεπτόμαστε το αριστερό υπόδενδρο , στην συνέχεια την ρίζα και τέλος το δεξί υπόδενδρο.
Στην **μεταδιατεταγμένη διάσχιση** στην αρχή επισκεπτόμαστε το αριστερό υπόδενδρο μετά το δεξί υπόδενδρο και τέλος την ρίζα του δένδρου.

Στην συνέχεια θα υλοποιήσουμε τον αλγόριθμο για αυτές τις τρεις διατάξεις σε ένα δυαδικό δένδρο.

**Αλγόριθμος για την διάσχιση Δυαδικού δένδρου.**

Παρακάτω δίνεται η εξήγηση του αλγόριθμου για την διάσχιση δυαδικού δένδρου.

Αφού δημιουργήσουμε μια δομή Struct η οποία έχει έναν ακέραιο αριθμό data και δυο δείκτες lchild, rchild που αναφέρονται στα αριστερά και δεξιά παιδία του κόμβου.

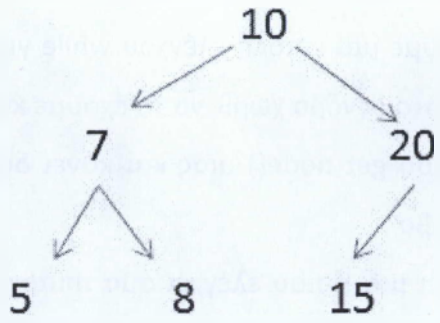
Στην αρχή δηλώνονται όλες οι μεταβλητές και δείκτες που θα χρησιμοποιήσουμε και μετά έχουμε μια εντολή ελέγχου while για να μπορούμε να δώσουμε όσα στοιχεία θέλουμε στο δένδρο χωρίς να το έχουμε καθορίσει από πριν. Σε αυτό βοηθάει και η συνάρτηση get node() μιας και κάνει δυναμική εκχώρηση μνήμης και εκχωρεί έναν νέο κόμβο .

Μέσα στην While υπάρχει μια if που ελέγχει άμα υπάρχει κάποιο στοιχείο στο δένδρο άμα δεν υπάρχει δημιουργεί έναν καινούριο κόμβο, αν υπάρχει καλείται η συνάρτηση insert() και δίνουμε το στοιχείο, η συνάρτηση instert() έχει σαν ορίσματα δυο δείκτες τύπου struct node ( που είχαμε δηλώση στην αρχή) .

Όταν δώσουμε το πρώτο στοιχείο θα μπει σαν ρίζα στο δένδρο από εκεί και πέρα θα ελέγχει κάθε στοιχείο άμα είναι μικρότερο από την ρίζα θα πηγαίνει αριστερά της ρίζας ενώ άμα είναι μεγαλύτερο θα πηγαίνει δεξιά της. Αυτό θα γίνεται για κάθε επίπεδο γονέα να βρει κάποιο γονέα κόμβο που δεν έχει κόμβους παιδία και αναλόγως πάλι θα μπει αριστερά ή δεξιά. Η While θα τελειώσει όταν στην ερώτηση που έχουμε βάλει άμα θέλουμε να δώσουμε άλλο στοιχείο η ans δεν είναι ίσο με y .

Μετά την While τρέχουν οι 3 τρόποι διάσχισης του δένδρου Ενδοδιατεταγμενη Διάσχιση που γίνεται με την κλήση της συνάρτησης inorder() , την Προδιατεταγμένη Διάσχιση με την κλήση της preorder() και την Μεταδιατεταγμενη Διάσχιση από την postorder().Στην inorder() επιστρέφει της τιμές από τα αριστερά υποδένδρα , μετά την ρίζα και τέλος τα δεξιά υποδένδρα αυτό έχει σαν αποτέλεσμα τα στοιχεία να εμφανιστούν με αύξουσα σειρά μιας και εμφανίζει πρώτα τα αριστερά που είναι τα μικρότερα στοιχεία και μετά τα δεξιά. Με την preorder() πρώτα επισκεπτόμαστε την ρίζα μετά τα αριστερά υποδένδρα και τέλος τα δεξιά. Και τέλος η postorder() πρώτα τα αριστερά υποδένδρα μετά τα δεξιά και τέλος την ρίζα και οι τρείς τρόποι καλούνται αναδρομικά.

Στην εικόνα έχουμε ένα δένδρο και πως θα ήταν η απεικόνιση με της τρείς διασχίσεις με τους εξής αριθμούς : 10 , 7 , 5 , 8 , 20 , 15 .




Ενδοδιατεταγμενη Διάσχιση : 5 ,7 ,8,10,15,20

Προδιατεταγμένη Διάσχιση : 10,7,5,8,20,15

Μεταδιατεταγμενη Διάσχιση : 5,8,7,15,20,10

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

typedef struct BST
{
   int data;
   struct BST *lchild,*rchild;
}node;

void insert(node *,node *);
void inorder(node *);
void preorder(node *);
void postorder(node *);

int main(int argc, char *argv[])
{
 char ans='N';
 int i=0 ,j,n , key;
 node *get_node();
 node *new_node,*root;
 root=NULL;

 do
        {

        new_node=get_node();
printf("Δώσε το στοιχείο \n ");

        scanf("%d",&new_node->data);
```

```
        if(root==NULL)
           root=new_node;
        else
        {
           insert(root,new_node);
        }

printf("Θέλετε να δώσετε άλλο στοιχείο?(y/n) \n");
        ans=getch();
  }while(ans=='y');

        if(root==NULL)
           printf("Το δένδρο δεν έχει φτιαχτεί  \n");
        else
          {
          printf("\n Ενδοδιατεταγμένη Διάσχιση  : \n ");
          inorder(root);
          printf("\n Προδιατεταγμένη Διάσχιση  :\n ");
          preorder(root);
          printf("\n Μεταδιατεταγμένη Διάσχιση: \n ");
          postorder(root);
          }

 system("PAUSE");
 return 0;

}

 node *get_node()
 {
 node *temp;
```

```
temp=(node *)malloc(sizeof(node));
temp->lchild=NULL;
temp->rchild=NULL;
return temp;
}


void insert(node *root,node *new_node)
{
 if(new_node->data < root->data)
    {
    if(root->lchild==NULL)
       root->lchild = new_node;
    else
       insert(root->lchild,new_node);
    }

 if(new_node->data > root->data)
    {
    if(root->rchild==NULL)
       root->rchild=new_node;
    else
       insert(root->rchild,new_node);
    }
}

void inorder(node *temp)
{
  if(temp!=NULL)
    {
    inorder(temp->lchild);
    printf("%d",temp->data);
```

```
    inorder(temp->rchild);
    }
}

void preorder(node *temp)
{
 if(temp!=NULL)
    {
    printf("%d",temp->data);
    preorder(temp->lchild);
    preorder(temp->rchild);
    }
}

void postorder(node *temp)
{
 if(temp!=NULL)
    {
    postorder(temp->lchild);
    postorder(temp->rchild);
    printf("%d",temp->data);
    }
}
```

## 4.3 Δυαδικά Δένδρα Αναζήτησης

Ίσως η πιο ενδιαφέρουσα κατηγορία δένδρων είναι τα δυαδικά δένδρα αναζήτησης. Ένα **Δυαδικό δέντρο αναζήτησης** (ΔΔΑ) έχει το χαρακτηριστικό ότι οι τιμές σε οποιοδήποτε αριστερό υπόδενδρο είναι μικρότερες από την τιμή του γονικού κόμβου και οι τιμές σε οποιοδήποτε δεξί υπόδενδρο είναι μεγαλύτερες από την τιμή του γονικού κόμβου. 'Οπως αναφέρεται στο βιβλίο του Ι. Παπουτσή, 2010[1].

Βασικό πλεονέκτημα αυτής της δομής είναι ότι συνδυάζει γρήγορες προσπελάσεις (αναζητήσεις) δεδομένων ενώ παράλληλα επιτρέπει την εύκολη εισαγωγή και διαγραφή στοιχείων. 'Οπως αναφέρεται στο βιβλίο του Ι.Μανωλόπουλου, 2004[6].

**Υλοποίηση Αλγόριθμου δυαδικού δένδρου αναζήτησης**

Παρακάτω δίνεται η εξήγηση του αλγόριθμου δυαδικού δένδρου αναζήτησης.

Αφού δημιουργήσουμε μια δομή Struct η οποία έχει έναν ακέραιο αριθμό data και δυο δείκτες lchild, rchild που αναφέρονται στα αριστερά και δεξιά παιδία του κόμβου.

Στην αρχή δηλώνονται όλες οι μεταβλητές και δείκτες που θα χρησιμοποιήσουμε και μετά έχουμε μια εντολή ελέγχου while για να μπορούμε να δώσουμε όσα στοιχεία θέλουμε στο δένδρο χωρίς να το έχουμε καθορίσει από πριν. Σε αυτό βοηθάει και η συνάρτηση get node() μιας και κάνει δυναμική εκχώρηση μνήμης και εκχωρεί έναν νέο κόμβο .Μέσα στην While υπάρχει μια if που ελέγχει άμα υπάρχει κάποιο στοιχείο στο δένδρο άμα δεν υπάρχει δημιουργεί έναν καινούριο κόμβο, αν υπάρχει καλείται η συνάρτηση insert() και δίνουμε το στοιχείο, η συνάρτηση instert() έχει σαν ορίσματα δυο δείκτες τύπου struct node ( που είχαμε δηλώση στην αρχή) .

Όταν δώσουμε το πρώτο στοιχείο θα μπει σαν ρίζα στο δένδρο από εκεί και πέρα θα ελέγχει κάθε στοιχείο άμα είναι μικρότερο από την ρίζα θα πηγαίνει αριστερά της ρίζας ενώ άμα είναι μεγαλύτερο θα πηγαίνει δεξιά της. Αυτό θα γίνεται για κάθε επίπεδο γονέα να βρει κάποιο γονέα κόμβο που δεν έχει κόμβους παιδία και αναλόγως πάλι θα μπει αριστερά ή δεξιά. Η While θα τελειώσει όταν στην ερώτηση που έχουμε βάλει άμα θέλουμε να δώσουμε άλλο στοιχείο η ans δεν είναι ίσο με y .

Μετά μας ζητάει να δώσουμε τον αριθμό που ψάχνουμε και καλεί την συνάρτηση search().Ξεκινάει από την ρίζα άμα το key είναι ίσο με την ρίζα τότε λέει ότι βρέθηκε και σαν γονέα έχει το 0 μιας και δεν υπάρχει κάποιος αριθμός πάνω από την ρίζα. Αν όμως δεν είναι η ρίζα ελέγχει άμα ο αριθμός που δώσαμε είναι μικρότερος από την ρίζα ψάχνει πρώτα στους αριστερούς κόμβους(γονέα) και μετά στα παιδιά του και μετά ψάχνει τους δεξιούς κόμβους. Όταν βρεθεί μας λέει ποιός είναι και ο γονέας του αριθμού. Και εκεί τελειώνει ο αλγόριθμος.

**Ο αλγόριθμος υλοποιημένο σε γλώσσα C.**

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

typedef struct BST
{
   int data;
   struct BST *lchild,*rchild;
}node;

void insert(node *,node *);

node *search(node *,int,node **);

 int main(int argc, char *argv[])
{
 char ans='N';
 int i=0 ,j,n , key;
 node *get_node();
 node *new_node,*root,*tmp, *parent;
root=NULL;

do
        {

        new_node=get_node();
printf("Δώσε το στοιχείο \n ");

        scanf("%d",&new_node->data);

        if(root==NULL)
           root=new_node;
        else
```

```
        {
           insert(root,new_node);
        }

printf("Θέλετε να δώσετε άλλο στοιχείο?(y/n) \n");
        ans=getch();
   }while(ans=='y');

        printf(" Δώστε το στοιχείο που ψάχνεται : \n");
        scanf("%d",&key);

        tmp = search(root,key,&parent);

        printf("Ο γονεας του %d ειναι %d \n ",
                tmp->data,parent->data);

system("PAUSE");
return 0;

}

node *get_node()
{
 node *temp;
 temp=(node *)malloc(sizeof(node));
 temp->lchild=NULL;
 temp->rchild=NULL;
 return temp;
 }
void insert(node *root,node *new_node)
{
  if(new_node->data < root->data)
    {
    if(root->lchild==NULL)
```

```
      root->lchild = new_node;
    else
      insert(root->lchild,new_node);
    }

 if(new_node->data > root->data)
    {
    if(root->rchild==NULL)
      root->rchild=new_node;
    else
      insert(root->rchild,new_node);
    }
}

node *search(node *root,int key,node **parent)
{
 node *temp;
 temp=root;
   while(temp!=NULL)
   {
    if(temp->data==key)
      {
      printf("Το %d στοιχείο υπάρχει \n",temp->data);
      return temp;
      }
    *parent=temp;

    if(temp->data>key)
      temp=temp->lchild;
    else
      temp=temp->rchild;
   }
 return NULL;
}
```

# ΒΙΒΛΙΟΓΡΑΦΙΑ

**[1] ΠΑΠΟΥΤΣΗΣ Ι.(2010)** «Εισαγωγή στις Δομές Δεδομένων και στους Αλγόριθμους» Εκδόσεις ΑΘ.ΣΤΑΜΟΥΛΗΣ

**[2].Jon Kleinberg & Eva Tardos(2008)** «Σχεδιασμός Αλγορίθμων» Εκδόσεις Κλειδάριθμος

**[3]. Gregory J. E. Rawlins(2004)** «Αλγόριθμοι Ανάλυση και Σύγκριση» Εκδόσεις Κριτική

**[4].Robert Sedgewick(2005)** «Αλγόριθμοι σε C» Εκδόσεις Κλειδάριθμος

**[5]. Μισυρλής Ν.(2008)** «Δομές Δεδομένων με C » Εκδόσεις Μισυρλής Νικόλαος

**[6]. ΜΑΝΩΛΟΠΟΥΛΟΣ Ι. (2004)** «Δομές Δεδομένων» Εκδόσεις ART OF TEXT

**[7]. Brian W. Kernighan, RITCHIE M. DENNIS(2011)** «Η Γλώσσα Προγραμματισμού C » Εκδόσεις Κλειδάριθμος

**[8]. SHANI S.(2004)** «Δομές Δεδομένων Αλγόριθμοι και Εφαρμογές στη C++» Εκδόσεις Τζιόλα

**[9]. WIRTH N.(1990)** «Αλγόριθμοι και Δομές Δεδομένων » Εκδόσεις Κλειδάριθμος

**[10]. ΚΟΙΛΙΑΣ Χ. (1993)** «Δομές Δεδομένων & οργανώσεις αρχείων» Εκδόσεις Νέων Τεχνολογιών

**[11]. ROBERTS E.(2011)** «Η τέχνη και η επιστήμη της C» Εκδόσεις Κλειδάριθμος

# ΣΥΜΠΕΡΑΣΜΑΤΑ

Στην παρούσα πτυχιακή εργασία ασχολήθηκα με τους αναδρομικούς αλγόριθμους. Η αναδρομή είναι ένας από τους πολλούς τρόπους επίλυσης προβλημάτων. Η αναδρομή δεν είναι απλή στην σκέψη αλλά κάποιος που καταλαβαίνει την φιλοσοφία της , διαπιστώνει την σπουδαιότητα της. Αυτό που κάνει την αναδρομική σκέψη τόσο σπουδαία είναι ότι κάνει πολύπλοκα προγράμματα πιο αποδοτικά(όχι πάντα ) επίσης οι αλγόριθμοι είναι πιο απλή, πιο εύκολη για έναν προγραμματιστή να τους διαβάσει , είναι σύντομοι και πιο κοντά στον μαθηματικό ορισμό. Τα προγράμματα που εξηγούνται στην εργασία είναι απλά για πιο εύκολη κατανοήσει της αναδρομής Μια ακόμα μέθοδος επίλυσης προβλημάτων που αναφέρεται και εξετάζεται στην εργασία είναι η διαίρει και κυρίευε( βασίλευε ) είναι αναδρομική μέθοδος , διότι καθώς διαιρεί το πρόβλημα σε μικρότερα(διαίρει) και μετά τα επιλύει αναδρομικά(κυρίευε).Η τεχνική διαίρει και κυρίευε αποτελεί την βάση για πολλούς αποδοτικούς αλγόριθμους όπως τους αλγόριθμους που εξετάσαμε γρήγορη ταξινόμηση(quicksort) και τον αλγόριθμο ταξινόμηση με συγχώνευση ( mergesort) στο $3^0$ κεφάλαιο. Στο τελευταίο κεφάλαιο της πτυχιακής ασχολήθηκα με την μη γραμμική δομή δεδομένων και συγκεκριμένα με δυαδικά δένδρα αναζήτησης(ΔΔΑ). Αυτό που κάνει τα δυαδικά δένδρα αναζήτησης σημαντικά είναι η δομή τους καθώς βοηθάει στην εύκολη αναζήτηση στοιχείων.

Όλα τα προγράμματα είναι που έχω χρησιμοποίηση στην εργασία μπορούν εύκολα να χρησιμοποιηθούν για εκπαιδευτικό σκοπό όπως επίσης από νέους προγραμματιστές για να καταλάβουν πως λειτουργεί η αναδρομή.

# ΠΑΡΑΡΤΗΜΑΤΑ

## 2.2 Υπολογισμός Παραγοντικού

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

int factorial(int n);

int main()
{

  int n , fact;

  printf("Δώσε έναν θετικό αριθμό: \n");
  scanf("%d",&n);

  if (n < 0)
    printf("Ο αριθμός που δώσατε δεν είναι θετικός. \n");
  else
  {
    fact = factorial(n);
    printf("Το παραγοντικό του %d είναι : %d \n", n, fact);
  }

 system("PAUSE");
 return 0;
}
```

```
int factorial(int n)
{
   if ( n <= 1 )
     return 1;
   else
     return ( n * factorial(n-1)) ;
}
```

## 2.3 Σειρά Fibonacci

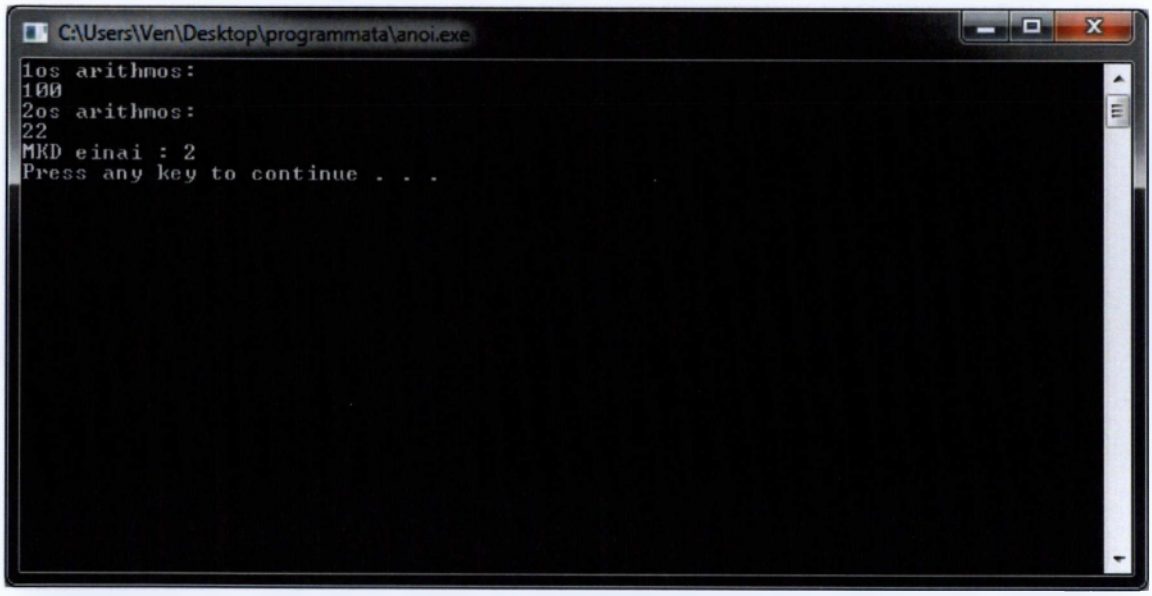
```
#include <stdio.h>
#include <stdlib.h>

int fibonacci(int n);

int main()
{
   int n , apotelesma ;

   printf("Δώσε έναν αριθμό : \n");
   scanf("%d",&n);

   apotelesma = fibonacci(n);

   printf("Ο αριθμός Fibonacci είναι : %d \n", apotelesma);



 system("PAUSE");
 return 0;
}
```

```
int fibonacci(int n)
{
  if ( n==0 || n==1 )
    return n;
  else
     return fibonacci( n-1 ) + fibonacci ( n-2);
}
```

## 2.4 Μέγιστος Κοινός Διαιρέτης

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

int gcd(int a, int b );

int main()
{
   int a , b ,apotelesma;
   printf("Δώσε τον πρώτο αριθμό: \n");
   scanf("%d",&a);
   printf("Δώσε τον δεύτερο αριθμό: \n");
   scanf("%d",&b);


   apotelesma = gcd(a,b);

   printf("Ο Μέγιστος Κοινός Διαιρέτης είναι : %d \n" ,apotelesma);


 system("PAUSE");
 return 0;
}
```

```
int gcd(int a, int b )
{
   if ( b==0 )
   return  a;
   else
   return  gcd( b, a % b );

}
```



## 2.5 Ελάχιστο κοινό πολλαπλάσιο

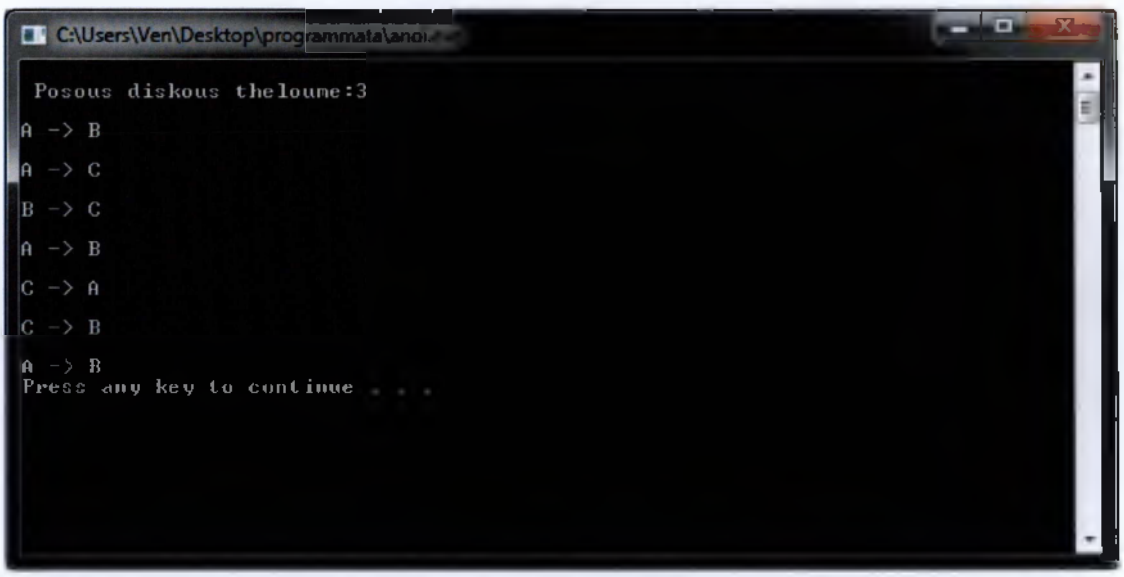
```
#include<stdio.h>

int lcm(int,int);

int main()
{
   int a,b,l;
   printf("Δώσε τους 2 αριθμούς ");
   scanf("%d%d",&a,&b);

   if(a>b)
   l = lcm(a,b);
   else
      l = lcm(b,a);

   printf("Ελάχιστο κοινό πολλαπλάσιο είναι: %d",l);
   system("PAUSE");
   return 0;
}
```

```
int lcm(int a,int b){

  static int temp = 1;

   if(temp % b == 0 && temp % a == 0)
      return temp;
   temp++;
   lcm(a,b);

  return temp;
}
```

## 2.6 Πύργοι του Hanoi

```
#include<stdio.h>

void TOH(int n,char x,char y,char z);

int main()
{
 int n;
 printf("\n Δώσε πόσους δίσκους θέλουμε:");
 scanf("%d",&n);
 TOH(n,'A','B','C');

 system("PAUSE");
  return 0;
}
```

```
void TOH(int n,char from ,char to ,char other) {
 if(n>0)
 {
  TOH(n-1,from,other,to);
  printf("\n%c -> %c",from,to);
  TOH(n-1,other,to,from);
 }
}
```



## 2.7 Ύψωση σε δύναμη

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

int power (int base, int exp);

int main(int argc, char *argv[])
{
 int base, exp,result;

 printf("Δώσε την βάση: ");
 scanf("%d",&base);
 printf("Δώσε  τον εκθέτη(θετικό αριθμό ): ");
 scanf("%d",&exp);
 result = power(base, exp);
 printf("Το αποτέλεσμα είναι: %d \n",result);

 system("PAUSE");

 return 0;
}
```

```
int power (int base, int exp)
{

        if(exp >= 1)
                return base * (power(base,exp - 1));
        else
                return 1;


}
```

## 2.8 Παλινδρομικός Αριθμός

```
#include<stdio.h>

int Palindrome(int);
int main(){
   int num,sum;

   printf("Δώσε έναν αριθμό: ");
   scanf("%d",&num);

   sum = Palindrome(num);

   if(num==sum)
      printf("%d είναι παλινδρομικός \n",num);
   else
   printf("%d δεν είναι παλινδρομικός \n",num);
   system("PAUSE");
   return 0;
}
```

```
int Palindrome(int num){

   static int sum=0,r;

   if(num!=0){
      r=num%10;
      sum=sum*10+r;
      Palindrome(num/10);
   }

   return sum;
}
```

## 2.9 Παλινδρομική Λέξη

```
#include <stdio.h>
#include <string.h>
int isPalindrome (char str[],int  length);

int main ()
{
   int result;
   char str[256];
   printf ("\n Δώσε την λέξη: \n");
   gets (str);
int length = strlen (str);
   result = isPalindrome (str, length);
   if (result==1)
      printf ("\n Η λέξη είναι παλινδρομική.  \n");
   else
      printf ("\n Η λέξη δεν είναι παλινδρομική \n");

   system("PAUSE");
   return 1;
}
```

```
int isPalindrome (char str[],int  length)
{

   if (length<=0)
      return 1;
   if (str[0] == str[length-1])
   {
      return isPalindrome (str+1, length-2);
   }
   else return 0;
}
```

## 2.10 Αντιστροφή Αριθμού

```
#include<stdio.h>

int main(){
   int num,reverse;

   printf("Δώσε τον αριθμό: ");
   scanf("%d",&num);

   reverse=rev(num);
   printf("Ο αντίστροφα ο αριθμός: %d",reverse);
   system("PAUSE");
   return 0;
}
```

```
int rev(int num){
   static sum,r;

   if(num>0){
      r=num%10;
      sum=sum*10+r;
      rev(num/10);
   }
   else
      return 0;

   return sum;
}
```

## 2.11 Αντιστροφή Λέξης

```
#include<stdio.h>
#define MAX 100
char* getReverse(char[]);

int main(){

   char str[MAX],*rev;

   printf("Δώσε την λέξη: ");
   scanf("%s",str);

   rev = getReverse(str);

   printf("Αντίστροφα είναι: %s",rev);
   system("PAUSE");
   return 0;
}
```

```
char* getReverse(char str[]){

   static int i=0;
   static char rev[MAX];

   if(*str){
      getReverse(str+1);
      rev[i++] = *str;
   }

   return rev;
}
```

## 2.12 Άθροισμα Ν πρώτον αριθμών

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

 int sum(int n);

int main(int argc, char *argv[])
{

 int n,add;

    printf("Δώσε έναν ακέραιο αριθμό:\n");
    scanf("%d",&n);

 add=sum(n);
    printf("Το σύνολο είναι : %d",add);
    printf("\n");


 system("PAUSE");
 return 0;
}
```

```
 int sum(int n)
{
      if(n==0)
         return n;
      else
         return n+sum(n-1);
}
```

## 2.13 Τρίγωνο Pascal

```
#include<stdio.h>

int pascal(int,int);
void space(int,int);

int main(int argc, char *argv[])
{
int num,i,j;
printf("\n Δώσε το νούμερο των σειρών: ");
scanf("%d",&num);
for(i=1;i<=num;i++)
{
space(num-i,3);
for(j=1;j<=i;j++)
{
printf("%3d",pascal(i,j));
space(1,3);
}
printf("\n");
}
 system("PAUSE");
  return 0;
}
```

```
int pascal(int row,int column)
{
if(column==0)
return 0;
else if(row==1&&column==1)
return 1;
else if(column>row)
return 0;
else
return (pascal(row-1,column-1)+pascal(row-1,column));
}

void space(int num,int mul)
{
int i;
num*=mul;
for(i=0;i<num;i++)
printf(" ");

}
```

## 2.14 Μετατροπή από δεκαδικό σε δυαδικό

```
#include <stdio.h>

int binary(int);

int main()
{
  int num, bin;

  printf("Δώσε τον δεκαδικό αριθμό: ");
  scanf("%d", &num);
  bin = binary(num);
  printf("Ο δυαδικός αριθμός του %d είναι %d\n", num, bin);
  system ("PAUSE");
}
```

```
int binary(int num)
{
   if (num == 0)
   {
      return 0;
   }
   else
   {
      return (num % 2) + 10 * binary(num / 2);
   }
}
```

## 2.15 Πόσα νούμερα έχει ένας αριθμός

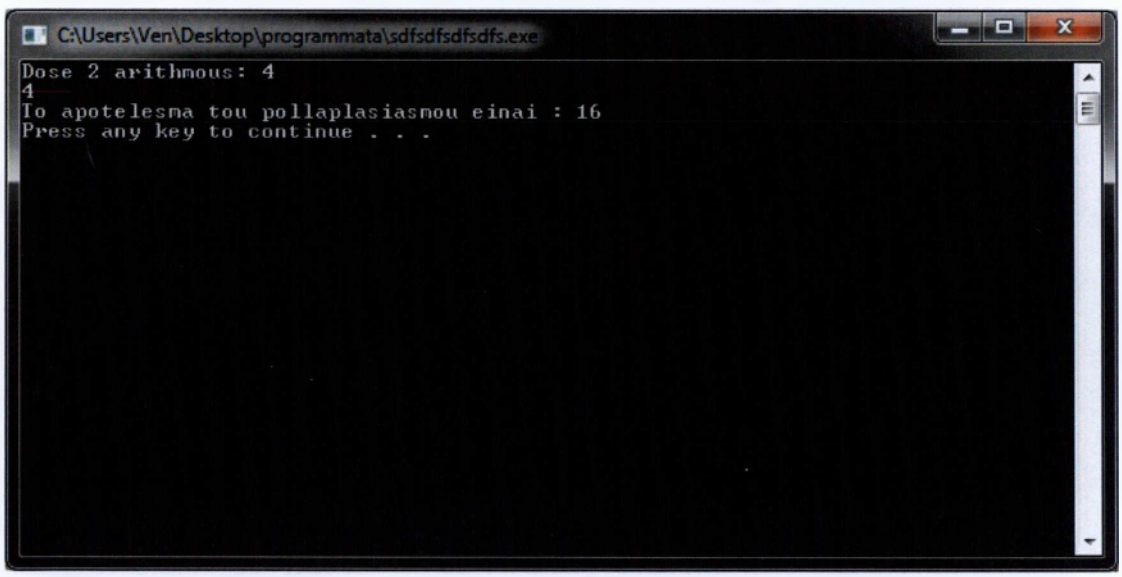
```
#include<stdio.h>

int countDigits(num);
int main(){
  int num,count;

  printf("Δώσε έναν αριθμό: ");
  scanf("%d",&num);

  count = countDigits(num);

  printf("Το σύνολο στοιχείων:  %d",count);
  system("PAUSE");
  return 0;
}
```

```
int countDigits(int num){
   static int count=0;

    if(num!=0){
        count++;
       countDigits(num/10);
    }

   return count;
}
```

## 2.16 Πολλαπλασιασμός 2 ακέραιων αριθμών

```
#include<stdio.h>

int multiply(int,int);

int main(){

   int a,b,product;
   printf("Δώσε τους 2 αριθμους: ");
   scanf("%d%d",&a,&b);

   product = multiply(a,b);

   printf("Το αποτέλεσμα του πολλαπλασιασμού είναι : %d",product);
   system("PAUSE");
   return 0;
}
```

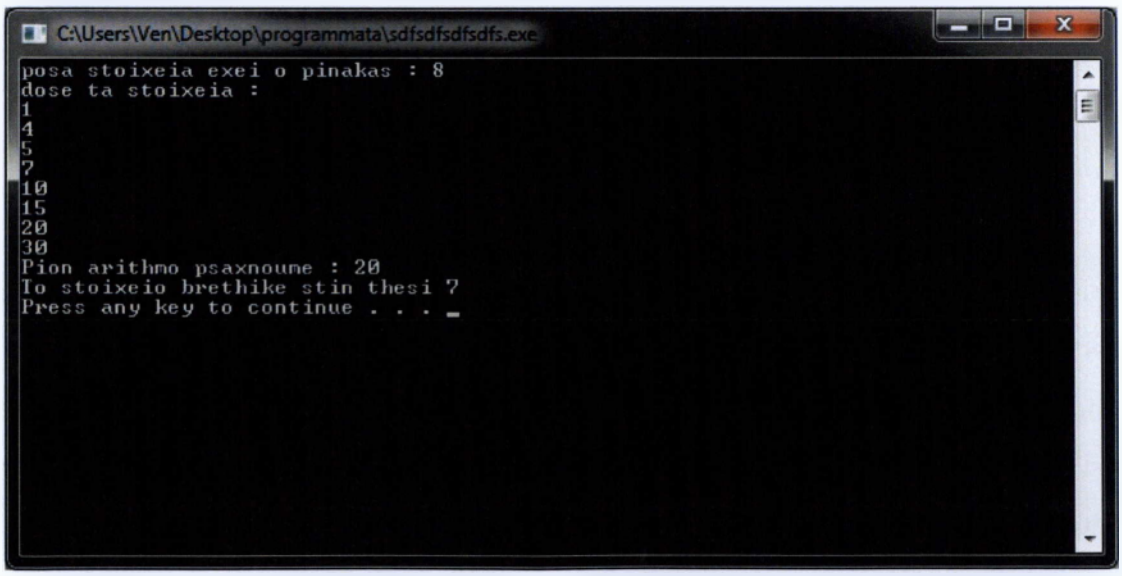
```
int multiply(int a,int b)
{
  if (b == 0)
        return 0;

return a + multiply(a, b - 1);
}
```



### 3.2 Δυαδική Αναζήτηση

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


binarysearch(int a[],int n,int arxi,int telos);


int main(int argc, char *argv[])
{
            int a[40];
           int n,no,x,apotelesma;

           printf("Δώσε πόσα στοιχεία θα έχει ο πίνακας: ");
           scanf("%d",&no);

           printf("Δώσε τα στοιχεία του πίνακα:\n");

           for(x=0;x<no;x++)
                   scanf("%d",&a[x]);

           printf("Δώσε ποιον αριθμό ψάχνουμε: ");
           scanf("%d",&n);

           apotelesma = binarysearch(a,n,0,no-1);

           if (apotelesma == -1)
              printf("Ο αριθμός δεν βρέθηκε ");



 system("PAUSE");
 return 0;
}
```

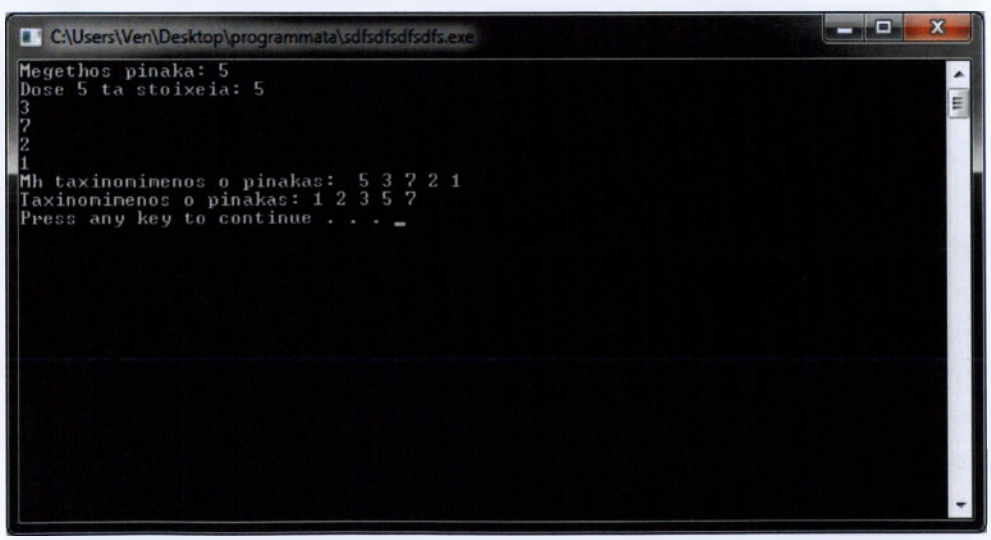
```
binarysearch(int a[],int n,int arxi,int telos)
        {
         int mesi;
        if (arxi > telos)
         return -1;
        mesi = (arxi + telos)/2;
        if(n == a[mesi])
          { printf("Το στοιχείο βρέθηκε στην θέση  %d\n",mesi+1);
            return 0;
          }
        if(n < a[mesi])
          { telos = mesi - 1;
            binarysearch(a,n, arxi, telos);
          }
        if(n > a[mesi])
          { arxi = mesi + 1;
            binarysearch(a,n, arxi, telos);
          }
        }
```



### 3.3.1 Γρήγορη ταξινόμηση

```
#include<stdio.h>
void quicksort(int x[200],int first,int last);

int main(){
  int x[200],size,i;
  printf("Δώσε το μέγεθος του πίνακα: ");
  scanf("%d",&size);

  printf("Δώσε  %d τα στοιχεία: ",size);
  for(i=0;i<size;i++)
    scanf("%d",&x[i]);

  printf("Μη ταξινομημένος  πίνακας: ");
  for(i=0;i<size;i++)
    printf(" %d",x[i]);

  printf("\n");

  quicksort(x,0,size-1);

  printf("Ταξινομημένος πίνακας: ");
  for(i=0;i<size;i++)
    printf(" %d",x[i]);
    system("pause");
  return 0;
}

void quicksort(int x[200],int first,int last){
    int pivot,j,temp,i;
     if(first<last){
```

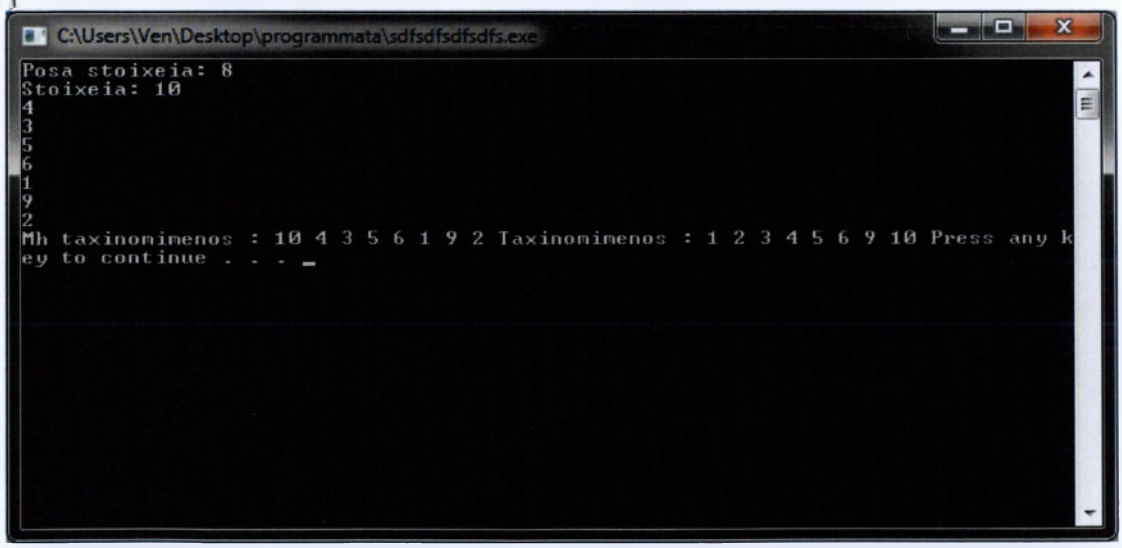
```
    pivot=first;
    i=first;
    j=last;

    while(i<j){
       while(x[i]<=x[pivot]&&i<last)
          i++;
       while(x[j]>x[pivot])
          j--;
       if(i<j){
          temp=x[i];
          x[i]=x[j];
          x[j]=temp;
       }
    }
    temp=x[pivot];
    x[pivot]=x[j];
    x[j]=temp;
    quicksort(x,first,j-1);
    quicksort(x,j+1,last);
  }
}
```



### 3.3.2 Ταξινόμηση με συγχώνευση

```
#include<stdio.h>
#define MAX 50

void merge(int arr[],int low,int mid,int high);
void mergeSort(int arr[],int low,int high);

int main(){

   int arr[MAX],i,n;

   printf("Δώσε το μέγεθος του πίνακα: ");
   scanf("%d",&n);

   printf("Δώσε τα στοιχεία του πίνακα: ");
   for(i=0;i<n;i++){
      scanf("%d",&arr[i]);
   }
   printf("Μη ταξινομημένος ο πίνακας: ");
   for(i=0;i<n;i++)
  {
      printf("%d ",arr[i]);
   }
   mergeSort(arr,0,n-1);
   printf("Ταξινομημένος ο πίνακας: ");
   for(i=0;i<n;i++)
  {
      printf("%d ",arr[i]);
   }
   system ("pause");
  return 0;
}
void mergeSort(int arr[],int low,int high){
   int mid;
   if(low<high){
      mid=(low+high)/2;
      mergeSort(arr,low,mid);
      mergeSort(arr,mid+1,high);
      merge(arr,low,mid,high);
   }
```

```
}
void merge(int arr[],int low,int mid,int high){
   int i,m,k,l,temp[MAX];
   l=low;
   i=low;
   m=mid+1;
   while((l<=mid)&&(m<=high)){
      if(arr[l]<=arr[m]){
         temp[i]=arr[l];
         l++;
      }
      else{
         temp[i]=arr[m];
         m++;
      }
      i++;
   }
   if(l>mid){
      for(k=m;k<=high;k++){
         temp[i]=arr[k];
         i++;
      }
   }
   else{
      for(k=l;k<=mid;k++){
         temp[i]=arr[k];
         i++;
      }
   }
   for(k=low;k<=high;k++){
      arr[k]=temp[k];
   }
}
```



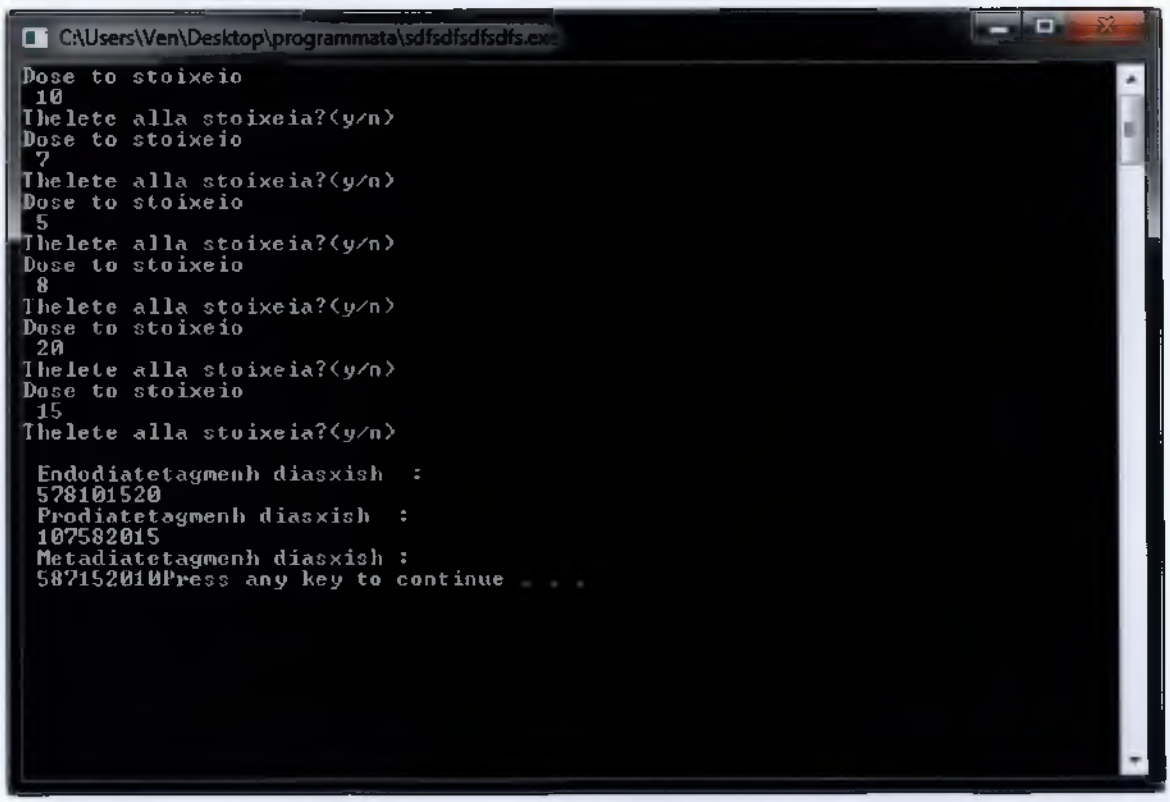
## 4.2 διάσχιση Δυαδικού δένδρου

```
# include <stdio.h>
# include <stdlib.h>
typedef struct BST
{
   int data;
   struct BST *lchild,*rchild;
}node;

void insert(node *,node *);
void inorder(node *);
void preorder(node *);
void postorder(node *);

int main()
{
 char ans='N';
 int i=0 ,j,n , key;
 node *get_node();
 node *new_node,*root;
root=NULL;
do
        {
        new_node=get_node();
printf("Δώσε το στοιχείο \n ");

        scanf("%d",&new_node->data);

        if(root==NULL)
           root=new_node;
```

```
        else
        {
          insert(root,new_node);
        }

printf("Θέλετε να δώσετε άλλο στοιχείο?(y/n) \n");
        ans=getch();
  }while(ans=='y');
       if(root==NULL)
         printf("Το δένδρο δεν έχει φτιαχτεί  \n");
       else
         {
         printf("\n Ενδοδιατεταγμένη Διάσχιση  : \n ");
         inorder(root);
         printf("\n Προδιατεταγμένη Διάσχιση  :\n ");
         preorder(root);
         printf("\n Μεταδιατεταγμένη Διάσχιση: \n ");
         postorder(root);
         }
 system("PAUSE");
 return 0;
}

 node *get_node()
 {
 node *temp;
 temp=(node *)malloc(sizeof(node));
 temp->lchild=NULL;
 temp->rchild=NULL;
 return temp;
 }
 void insert(node *root,node *new_node)
```

```
{
 if(new_node->data < root->data)
   {
   if(root->lchild==NULL)
      root->lchild = new_node;
   else
      insert(root->lchild,new_node);
   }

 if(new_node->data > root->data)
   {
   if(root->rchild==NULL)
      root->rchild=new_node;
   else
      insert(root->rchild,new_node);
   }
}
void inorder(node *temp)
{
  if(temp!=NULL)
   {
   inorder(temp->lchild);
   printf("%d",temp->data);
   inorder(temp->rchild);
   }
}
void preorder(node *temp)
{
 if(temp!=NULL)
   {
   printf("%d",temp->data);
   preorder(temp->lchild);
```

```
    preorder(temp->rchild);
    }
}
void postorder(node *temp)
{
 if(temp!=NULL)
    {
    postorder(temp->lchild);
    postorder(temp->rchild);
    printf("%d",temp->data);
    }
}
```

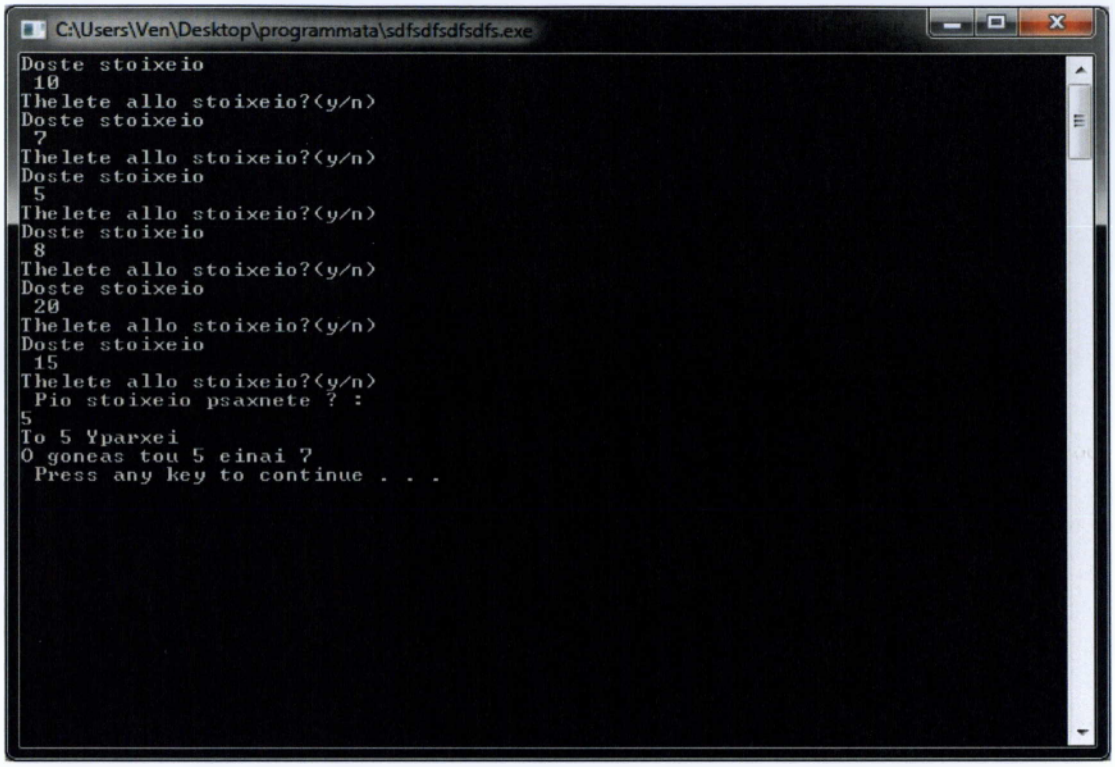


## 4.3 Δυαδικά Δένδρα Αναζήτησης

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

typedef struct BST
{
   int data;
   struct BST *lchild,*rchild;
}node;

void insert(node *,node *);
node *search(node *,int,node **);

 int main(int argc, char *argv[])
{
 char ans='N';
 int i=0 ,j,n , key;
 node *get_node();
 node *new_node,*root,*tmp, *parent;
root=NULL;

do
        {
        new_node=get_node();
printf("Δώσε το στοιχείο \n ");
scanf("%d",&new_node->data);

        if(root==NULL)
           root=new_node;
        else
        {
           insert(root,new_node);
        }
```

```
printf("Θέλετε να δώσετε άλλο στοιχείο?(y/n) \n");
        ans=getch();
  }while(ans=='y');

        printf(" Δώστε το στοιχείο που ψάχνεται : \n");
        scanf("%d",&key);

        tmp = search(root,key,&parent);

        printf("Ο γονεας του %d ειναι %d \n ",
                tmp->data,parent->data);

system("PAUSE");
return 0;
}

node *get_node()
{
 node *temp;
 temp=(node *)malloc(sizeof(node));
 temp->lchild=NULL;
 temp->rchild=NULL;
 return temp;
 }
void insert(node *root,node *new_node)
{
 if(new_node->data < root->data)
   {
   if(root->lchild==NULL)
      root->lchild = new_node;
   else
      insert(root->lchild,new_node);
   }
```

```
 if(new_node->data > root->data)
   {
   if(root->rchild==NULL)
      root->rchild=new_node;
   else
      insert(root->rchild,new_node);
   }
}
node *search(node *root,int key,node **parent)
{
 node *temp;
 temp=root;
   while(temp!=NULL)
   {
    if(temp->data==key)
      {
      printf("Το %d στοιχείο υπάρχει \n",temp->data);
      return temp;
      }

    *parent=temp;
    if(temp->data>key)
      temp=temp->lchild;
    else
      temp=temp->rchild;
   }
 return NULL;

}
```


C:\Users\Ven\Desktop\programmata\sdfsdfsdfsdfs.exe
Doste stoixeio
10
Thelete allo stoixeio?(y/n)
Doste stoixeio
7
Thelete allo stoixeio?(y/n)
Doste stoixeio
5
Thelete allo stoixeio?(y/n)
Doste stoixeio
8
Thelete allo stoixeio?(y/n)
Doste stoixeio
20
Thelete allo stoixeio?(y/n)
Doste stoixeio
15
Thelete allo stoixeio?(y/n)
Pio stoixeio psaxnete ? :
5
To 5 Yparxei
O goneas tou 5 einai 7
Press any key to continue . . .