Νικόλαος Β. Πλατής
Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής

Τεχνικές Προγραμματισμού

Μάρτιος 2001

1. Εισαγωγή

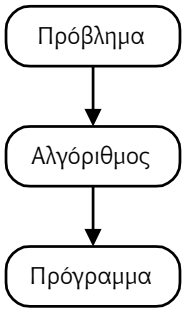
1.1 Βασικές έννοιες

Η επιστήμη της Πληροφορικής μελετά την επίλυση προβλημάτων με τη βοήθεια Ηλεκτρονικών Υπολογιστών (Η/Υ). Η ολοένα αυξανόμενη παρουσία των Η/Υ στην καθημερινή ζωή μας δίνει μια σαφή ένδειξη του εύρους των εφαρμογών στις οποίες η χρήση τους αποδεικνύεται επωφελής. Χαρακτηριστικά αναφέρουμε:

- επιστημονικές εφαρμογές, από τις πιο απλές (επίλυση τριωνύμου, επίλυση συστημάτων εξισώσεων) μέχρι πολυσύνθετες (αστρονομικοί υπολογισμοί, πρόβλεψη καιρού),
- εμπορικές εφαρμογές (μηχανοργάνωση επιχειρήσεων, τήρηση αρχείων γραμματείας),
- βιομηχανικές εφαρμογές (συστήματα βιομηχανικής σχεδίασης, έλεγχος γραμμών παραγωγής),
- εργασίες που μπορεί ο καθένας να πραγματοποιήσει με τον Προσωπικό Υπολογιστή του (συγγραφή κειμένων, επεξεργασία φωτογραφιών, παιχνίδια).

Κάθε Η/Υ μπορεί στην πραγματικότητα να εκτελεί μόνο πολύ απλές πράξεις (π.χ. προσθέσεις, συγκρίσεις, καταχωρίσεις στη μνήμη του) με πολύ μεγάλη ταχύτητα. Επομένως, για να επιλυθεί ένα οποιοδήποτε πρόβλημα όπως τα παραπάνω με τη βοήθεια Η/Υ, πρέπει η διαδικασία επίλυσής του να μπορεί να αναχθεί σε μία (πιθανώς μακριά και πολύπλοκη) σειρά από απλές πράξεις, οι οποίες στη συνέχεια θα γνωστοποιηθούν στον υπολογιστή με κατάλληλο τρόπο.

Μία τέτοια ακολουθία απλών βημάτων των οποίων η επεξεργασία σε μια αυστηρά καθορισμένη σειρά οδηγεί στη λύση ενός προβλήματος, ονομάζεται *αλγόριθμος*. Η περιγραφή ενός αλγορίθμου σε κατάλληλη μορφή ώστε να γίνει κατανοητός και να εκτελεστεί από τον Η/Υ, ονομάζεται *πρόγραμμα*. Αυτή η περιγραφή γίνεται σε μία *γλώσσα προγραμματισμού*, την οποία προφανώς κατανοούν ισοδύναμα τόσο ο *προγραμματιστής* όσο και ο υπολογιστής· είναι κάτι ανάλογο της φυσικής γλώσσας που είναι απαραίτητη για τη συνεννόηση δύο ή περισσότερων ανθρώπων.



΄Οπως οι φυσικές γλώσσες, έτσι και οι γλώσσες προγραμματισμού διέπονται από συγκεκριμένους κανόνες οι οποίοι καθορίζουν αν μία διατύπωση είναι τυπικά σωστή. Καθώς μάλιστα ο Η/Υ δεν διαθέτει τις διανοητικές δυνατότητες του ανθρώπινου εγκεφάλου παρά μόνο τη δυνατότητα να εκτελεί προκαθορισμένες *εντολές*, οι κανόνες των γλωσσών προγραμματισμού είναι αρκετά περιοριστικοί και πρέπει να τηρούνται με ακρίβεια προκειμένου ο Η/Υ να κατανοήσει ένα πρόγραμμα.

1.2 Γλώσσες Προγραμματισμού

1.2.1 Γλώσσες χαμηλού επιπέδου

Στους πρώτους Η/Υ που κατασκευάστηκαν, ο προγραμματισμός δεν ήταν παρά άνοιγμα ή κλείσιμο διακοπτών ρεύματος και παρακολούθηση φωτεινών ενδείξεων. Με τον τρόπο αυτό οι θέσεις μνήμης του υπολογιστή έπαιρναν συγκεκριμένες τιμές, οι οποίες ερμηνεύονταν κατάλληλα από τον υπολογιστή ώστε να εκτελέσει τις απαραίτητες πράξεις. Όταν οι δυνατότητες εισόδου και εξόδου δεδομένων σε έναν Η/Υ εξελίχθηκαν, αυτός ο τύπος προγραμματισμού συνέχισε να υπάρχει, καθώς αποτελεί τον πιο άμεσο τρόπο επικοινωνίας μεταξύ του χρήστη και του υπολογιστή· βέβαια οι τιμές στη μνήμη του υπολογιστή μπορούσαν πλέον να εισάγονται μέσω ενός πληκτρολογίου και να παρακολουθούνται σε μια οθόνη. Αυτή η γλώσσα προγραμματισμού καλείται γενικά *γλώσσα μηχανής*.

Μία εξέλιξη της γλώσσας μηχανής είναι η *συμβολική* γλώσσα (assembly language). Σε αυτή οι στοιχειώδεις πράξεις που κατανοεί ο Η/Υ (προσθέσεις, συγκρίσεις, καταχώριση τιμής σε θέση μνήμης) έχουν μνημονικά ονόματα (add, compare, move) και έτσι είναι δυνατή η περιγραφή του προβλήματος με λίγο πιο προσιτό στο χρήστη τρόπο.

Η γλώσσα μηχανής και η συμβολική γλώσσα καλούνται από κοινού *γλώσσες χαμηλού επιπέδου*, καθώς βρίσκονται αρκετά «κοντά» στον Η/Υ και στον τρόπο λειτουργίας του.

Οι γλώσσες χαμηλού επιπέδου έχουν δύο βασικά μειονεκτήματα: την πολυπλοκότητά τους, καθώς ουσιαστικά ο χρήστης προσαρμόζεται στις απαιτήσεις του υπολογιστή και όχι το αντίστροφο, και επίσης το γεγονός ότι είναι απόλυτα εξειδικευμένες για κάθε τύπο υπολογιστή, καθιστώντας έτσι τα σχετικά προγράμματα δύσκολα *μεταφέρσιμα* σε Η/Υ διαφορετικών τύπων.

1.2.2 Ανώτερες γλώσσες προγραμματισμού

Η δυσκολία χρήσης των γλωσσών προγραμματισμού χαμηλού επιπέδου οδήγησε στη σύλληψη *ανώτερων γλωσσών προγραμματισμού*, γλωσσών στις οποίες η λύση ενός προβλήματος δεν εκφράζεται πλέον με όρους του Η/Υ αλλά με όρους προσιτούς στην ανθρώπινη λογική. Οι ανώτερες γλώσσες προγραμματισμού είναι ουσιαστικά ανεξάρτητες από το συγκεκριμένο Η/Υ στον οποίο τελικά εκτελούνται τα προγράμματά τους και παρέχουν δυνατότητες που διευκολύνουν τη διατύπωση της λύσης των προβλημάτων. Ένα πρόγραμμα γραμμένο σε μία τέτοια γλώσσα μοιάζει με απλοϊκό κείμενο, καθώς αποτελείται από απλές λέξεις και μαθηματικές παραστάσεις, και η ανάγνωσή του παρέχει μία αναλυτική παρουσίαση της λύσης του προβλήματος. Οι γλώσσες αυτές αναφέρονται και ως *γλώσσες προγραμματισμού προσανατολισμένες στο πρόβλημα*.

Όπως έγινε φανερό, οι ανώτερες γλώσσες προγραμματισμού λύνουν και τα δύο βασικά προβλήματα των γλωσσών χαμηλού επιπέδου. Όμως οι γλώσσες αυτές δεν είναι άμεσα κατανοητές στον Η/Υ, ο οποίος πάντα καταλαβαίνει μόνο τη γλώσσα μηχανής. Ο ενδιάμεσος κρίκος είναι ένας *μεταγλωττιστής* ή *διερμηνευτής* (compiler / interpreter), δηλαδή ένα πρόγραμμα που μετατρέπει με αυτόματο τρόπο ένα πρόγραμμα γραμμένο σε ανώτερη γλώσσα προγραμματισμού στην ισοδύναμη ακολουθία εντολών γλώσσας μηχανής του συγκεκριμένου Η/Υ. Πάντως αυτή η αυτόματη μετατροπή δεν είναι δυνατό να παραγάγει τα βέλτιστα προγράμματα γλώσσας μηχανής, κάτι που έχει ως αποτέλεσμα ο προγραμματισμός χαμηλού επιπέδου να βρίσκει ακόμη εφαρμογή σε εξειδικευμένες περιπτώσεις όπου το μέγεθος και η ταχύτητα του εκτελέσιμου προγράμματος είναι κρίσιμοι παράγοντες.

Πρέπει να αναφερθεί ότι υπάρχουν αρκετές διαφορετικές ανώτερες γλώσσες προγραμματισμού. Ένας παράγοντας διαφοροποίησης είναι το συγκεκριμένο πεδίο εφαρμογής που καλούνται να αντιμετωπίσουν. Για παράδειγμα, υπάρχουν γλώσσες που υποστηρίζουν εξελιγμένους μαθηματικούς υπολογισμούς και προτιμώνται για την επίλυση επιστημονικών προβλημάτων· άλλες γλώσσες διευκολύνουν την επεξεργασία δεδομένων και χρησιμοποιούνται για τον προγραμματισμό εμπορικών εφαρ-

μογών· τέλος, υπάρχουν γλώσσες γενικής χρήσης που παρέχουν τα βασικά εργαλεία για την επίλυση μεγάλου φάσματος προβλημάτων.

Ένας δεύτερος παράγοντας διαφοροποίησης μεταξύ των (ανώτερων) γλωσσών προγραμματισμού είναι τα μοντέλα προγραμματισμού που υποστηρίζουν. Με αυτά θα ασχοληθούμε αμέσως παρακάτω.

1.3 Μοντέλα Προγραμματισμού

Οι ανώτερες γλώσσες προγραμματισμού επιτρέπουν, όπως αναφέραμε, την έκφραση της λύσης ενός προβλήματος με όρους σχετικούς με το πρόβλημα και όχι με τις δυνατότητες του Η/Υ. Για το λόγο αυτό αποτελούν ένα μεγάλο βήμα προς την ευκολότερη αξιοποίηση του Η/Υ σε σχέση με τις γλώσσες χαμηλού επιπέδου. Όμως και οι ανώτερες γλώσσες προγραμματισμού έχουν εμφανίσει σημαντικά στοιχεία εξέλιξης στην προσπάθεια να εκφράσουν πιο άμεσα τα προβλήματα του πραγματικού κόσμου που επιλύονται με τη βοήθεια Η/Υ.

1.3.1 Μονολιθικός προγραμματισμός

Οι πρώτες ανώτερες γλώσσες προγραμματισμού που αναπτύχθηκαν έδιναν στον προγραμματιστή τη δυνατότητα να εκφράσει τη λύση ενός προβλήματος χρησιμοποιώντας τις βασικές μαθηματικές πράξεις και σχέσεις καθώς και κάποιες στοιχειώδεις *δομές ελέγχου* της ροής του προγράμματος όπως επανάληψη συγκεκριμένων εντολών κάποιο αριθμό φορών ή εκτέλεση κάποιων εντολών μόνο εφόσον ισχύει μια συνθήκη.

Για παράδειγμα, ένα πρόγραμμα για την εύρεση των ριζών ενός τριωνύμου $\alpha x^2 + \beta x + \gamma = 0$ θα είχε την εξής μορφή: (για απλότητα υποθέτουμε ότι η διακρίνουσα είναι μη αρνητική, $\Delta = \beta^2 - 4\alpha\gamma \geq 0$)

ΘΕΣΕ $\Delta = \beta^2 - 4\alpha\gamma$
ΑΝ $\Delta > 0$ ΤΟΤΕ
    ΘΕΣΕ $\rho_{1,2} = (-\beta \pm \sqrt{\Delta})/2\alpha$
ΔΙΑΦΟΡΕΤΙΚΑ
    ΘΕΣΕ $\rho = -\beta/2\alpha$

Οι δυνατότητες που παρείχαν αυτές οι γλώσσες ήταν αρκετές για την αποτελεσματική περιγραφή της λύσης των προβλημάτων που αντιμετωπίζονταν με τη βοήθεια Η/Υ εκείνη την εποχή: επρόκειτο συνήθως για πολύπλοκα επιστημονικά προβλήματα, των οποίων οι αλγόριθμοι είχαν μεγάλες απαιτήσεις σε ταχύτητα υπολογισμών αλλά σχετικά απλή λογική που μπορούσε να εκφραστεί με μαθηματικούς όρους. Αυτός ο τύπος προγραμματισμού ονομάζεται *μονολιθικός* ή *σειραϊκός* (serial).

Γλώσσες που υποστηρίζουν τον μονολιθικό προγραμματισμό είναι οι Basic και Fortran, στις αρχικές εκδόσεις τους.

1.3.2 Αρθρωτός προγραμματισμός

Καθώς τα προβλήματα που τίθονταν προς επίλυση με τη χρήση Η/Υ γίνονταν ολοένα και πιο πολύπλοκα, έγινε φανερό ότι δεν ήταν εφικτό να εκφραστούν σε ένα μονολιθικό, ενιαίο πρόγραμμα. Αυτό που έλειπε από το μοντέλο ήταν η δυνατότητα που διαθέτει ο ανθρώπινος νους να αντιμετωπίζει τα προβλήματα *διασπώντας* τα σε μικρότερα υποπροβλήματα – διαδικασίες, τα οποία μπορεί πολύ πιο εύκολα να επιλύσει ανεξάρτητα. Για παράδειγμα, ακόμη και στο απλό πρόβλημα του υπολογισμού των ριζών ενός τριωνύμου μπορούμε να διακρίνουμε τρία ανεξάρτητα υποπροβλήματα:

- Υπολογισμός της διακρίνουσας
- Υπολογισμός των δύο ριζών (όταν $\Delta > 0$)
- Υπολογισμός της διπλής ρίζας (όταν $\Delta = 0$)

Αναπτύχθηκαν λοιπόν γλώσσες προγραμματισμού που επεκτείνουν τις μονολιθικές γλώσσες δίνοντας επιπλέον τη δυνατότητα στον προγραμματιστή να αναλύσει το συνολικό πρόβλημα σε απλούστερες επιμέρους ενέργειες. Πρακτικά οι γλώσσες αυτές υποστηρίζουν τη διάσπαση του προγράμματος σε ανεξάρτητες ενότητες – *υποπρογράμματα* που μπορούν να *κληθούν* από οποιοδήποτε σημείο του προγράμματος. Ο σχετικός τρόπος προγραμματισμού ονομάζεται *αρθρωτός* (modular) ή *δομημένος* (structured) ή *διαδικαστικός* (procedural).

Κάνοντας χρήση των δυνατοτήτων μιας τέτοιας γλώσσας προγραμματισμού, η επίλυση του τριωνύμου μπορεί να περιγραφεί ως εξής:

ΘΕΣΕ Δ=*ΔΙΑΚΡΙΝΟΥΣΑ* του $\alpha x^2 + \beta x + \gamma$
ΑΝ $\Delta > 0$ ΤΟΤΕ
    ΘΕΣΕ $\rho_{1,2}$ =*ΔΥΟ ΡΙΖΕΣ* του $\alpha x^2 + \beta x + \gamma$
ΔΙΑΦΟΡΕΤΙΚΑ
    ΘΕΣΕ ρ=*ΔΙΠΛΗ ΡΙΖΑ* του $\alpha x^2 + \beta x + \gamma$

όπου

« *ΔΙΑΚΡΙΝΟΥΣΑ* του $\alpha x^2 + \beta x + \gamma$ » = $\beta^2 - 4\alpha\gamma$
« *ΔΥΟ ΡΙΖΕΣ* του $\alpha x^2 + \beta x + \gamma$ » = $(-\beta \pm \sqrt{\Delta}) / 2\alpha$
« *ΔΙΠΛΗ ΡΙΖΑ* του $\alpha x^2 + \beta x + \gamma$ » = $-\beta / 2\alpha$

είναι τα υποπρογράμματα που χρησιμοποιούνται.

Σε ένα απλό πρόβλημα όπως αυτό μία τέτοια ανάλυση φαίνεται ίσως υπερβολική· όμως είναι σίγουρο ότι σε ένα πιο σύνθετο πρόβλημα μία ανάλογη διάσπαση σε απλούστερα υποπροβλήματα συμβάλλει σημαντικά στην ευκολότερη διατύπωση της λύσης. Για παράδειγμα, έστω ότι διατηρούμε ένα αρχείο με τα στοιχεία κάποιων φοιτητών και θέλουμε να εισαγάγουμε κάποια καινούργια στοιχεία. Για κάθε φοιτητή το πρόγραμμα θα διαβάζει τα στοιχεία του και θα υπολογίζει τον τελικό βαθμό του· κατόπιν θα ελέγχει αν υπάρχουν ήδη στοιχεία του στο αρχείο, οπότε θα τα ενημερώνει, ή στην αντίθετη περίπτωση θα προσθέτει μια νέα εγγραφή. Αν προσπαθούσαμε να υλοποιήσουμε τη λύση αυτού του προβλήματος με ένα ενιαίο πρόγραμμα είναι σίγουρο ότι θα δυσκολευόμασταν πολύ να αντιμετωπίσουμε όλες τις λεπτομέρειες συγχρόνως. Αν τελικά τα καταφέρναμε, θα καταλήγαμε σε ένα πρόγραμμα αρκετών σελίδων του οποίου η λειτουργία και η δομή θα ήταν σαφώς δυσνόητες. Είναι πιο εύκολο να αντιμετωπίσουμε το πρόβλημα διασπώντας το σε επιμέρους διαδικασίες, όπως φαίνεται στην επόμενη περιγραφή σε μία υποθετική γλώσσα αρθρωτού προγραμματισμού:

ΑΝΟΙΓΜΑ ΑΡΧΕΙΟΥ του αρχείου των φοιτητών
ΕΠΑΝΑΛΗΨΗ για όλους τους φοιτητές που δίνονται
    ΕΙΣΑΓΩΓΗ ΣΤΟΙΧΕΙΩΝ του φοιτητή
    ΥΠΟΛΟΓΙΣΜΟΣ ΤΕΛΙΚΟΥ ΒΑΘΜΟΥ του φοιτητή
    ΕΥΡΕΣΗ ΣΤΟΙΧΕΙΩΝ του φοιτητή
    ΑΝ υπάρχει ο φοιτητής ΤΟΤΕ
        ΕΝΗΜΕΡΩΣΗ ΣΤΟΙΧΕΙΩΝ του φοιτητή
    ΔΙΑΦΟΡΕΤΙΚΑ
        ΕΙΣΑΓΩΓΗ ΕΓΓΡΑΦΗΣ με τα στοιχεία του φοιτητή

ΚΛΕΙΣΙΜΟ ΑΡΧΕΙΟΥ του αρχείου των φοιτητών

Με πλάγια στοιχεία σημειώνονται και εδώ τα επιμέρους υποπρογράμματα που χρησιμοποιούνται (τα οποία δεν αναλύονται περαιτέρω για απλότητα). Παρατηρούμε ότι με τον τρόπο αυτό η περιγραφή της λύσης γίνεται σε ένα αρκετά «υψηλό» επίπεδο και η πολυπλοκότητά της επιμερίζεται και κρύβεται στα υποπρογράμματα.

Με τη χρήση του αρθρωτού προγραμματισμού επωφελούμαστε από μερικά σημαντικά χαρακτηριστικά του:

- Τα υποπρογράμματα που κατασκευάζονται χαρακτηρίζονται από *γενικότητα*, δηλαδή μπορούν να επιλύουν μία ολόκληρη κλάση προβλημάτων και όχι μόνο ένα συγκεκριμένο. Αυτό είναι δυνατό μέσω των *παραμέτρων* που δέχονται τα υποπρογράμματα, μέσω των οποίων ο προγραμματιστής μεταφέρει στο υποπρόγραμμα δεδομένα που αφορούν το πρόβλημα που επιλύεται. Για παράδειγμα, το υποπρόγραμμα «*ΔΙΑΚΡΙΝΟΥΣΑ*» παραπάνω έχει σαν παράμετρο το τριώνυμο (ουσιαστικά τους συντελεστές α, β, γ) και έτσι μπορεί να υπολογίσει τη διακρίνουσα οποιουδήποτε τριωνύμου.
- Δίνεται η δυνατότητα *επαναχρησιμοποίησης* των ανεξάρτητων τμημάτων του προγράμματος. Για παράδειγμα, είναι πιθανό σε κάποια εφαρμογή να μην είναι απαραίτητος ο τελικός υπολογισμός των ριζών του τριωνύμου παρά μόνο της διακρίνουσάς του· με διάσπαση όπως αυτή που περιγράφηκε, το υποπρόγραμμα « *ΔΙΑΚΡΙΝΟΥΣΑ* του $\alpha x^2 + \beta x + \gamma$ » μπορεί να χρησιμοποιηθεί ανεξάρτητα, «αντιγράφοντάς» το από το προηγούμενο πρόγραμμα. Αντίστοιχα η «*ΕΙΣΑΓΩΓΗ ΕΓΓΡΑΦΗΣ* του φοιτητή» μπορεί να χρησιμοποιηθεί αυτούσια σε άλλα σημεία ενός ευρύτερου προγράμματος που θα χειρίζεται το αρχείο των φοιτητών.
- Τα υποπρογράμματα λειτουργούν ως *μαύρα κουτιά*: έχουν συγκεκριμένες εισόδους, εκτελούν μία σαφώς ορισμένη λειτουργία και παράγουν κάποιο αποτέλεσμα· ο τρόπος που υλοποιείται η λειτουργία τους δεν ενδιαφέρει καθόλου κατά τη χρήση τους. Έτσι η υλοποίηση των υποπρογραμμάτων μπορεί να αλλάξει χωρίς να επηρεαστεί το υπόλοιπο πρόγραμμα. Αυτό μπορεί να φανεί χρήσιμο για τη βελτιστοποίηση του προγράμματος· για παράδειγμα, η «*ΕΥΡΕΣΗ ΣΤΟΙΧΕΙΩΝ*» μπορεί να υλοποιηθεί με διάφορους τρόπους ανάλογα με το πώς είναι αποθηκευμένα τα στοιχεία στο αρχείο των φοιτητών. Επίσης μπορεί να διευκολύνει την εύρεση σφαλμάτων στο πρόγραμμα, καθώς ο κώδικας που αφορά συγκεκριμένες λειτουργίες είναι συγκεντρωμένος σε λίγα σημεία του προγράμματος· αν για παράδειγμα διαπιστωθεί λάθος κατά την ενημέρωση των στοιχείων των φοιτητών η προσοχή θα εστιαστεί στο υποπρόγραμμα «*ΕΝΗΜΕΡΩΣΗ ΣΤΟΙΧΕΙΩΝ*».

Κατεξοχήν γλώσσες που υποστηρίζουν τον αρθρωτό προγραμματισμό είναι οι Pascal και C. Επίσης νεότερες εκδοχές των γλωσσών Basic και Fortran υποστηρίζουν αυτό το μοντέλο.

1.3.3 Αντικειμενοστρεφής προγραμματισμός

Με τον αρθρωτό προγραμματισμό έγινε δυνατή η αποτελεσματική επίλυση αρκετά σύνθετων προβλημάτων με τη βοήθεια Η/Υ, με την περιγραφή τους ως μια σειρά ανεξάρτητων ενεργειών. Ακόμα και αυτό το μοντέλο όμως αποδείχθηκε ανεπαρκές για την αντιμετώπιση νεώτερων και ακόμη πιο πολύπλοκων προβλημάτων. Τη λύση έδωσε ο *αντικειμενοστρεφής προγραμματισμός* (object oriented programming).

Στον αντικειμενοστρεφή προγραμματισμό κεντρική έννοια είναι το *αντικείμενο*, που αποτελεί προγραμματιστική αναπαράσταση μίας οντότητας του πραγματικού κόσμου. Η λύση του προβλήματος περιγράφεται ως μια σειρά αλληλεπιδράσεων μεταξύ κατάλληλων αντικειμένων, και η πολυπλο-

κότητα διασπάται και ενσωματώνεται σε αυτά τα αντικείμενα. Για παράδειγμα, στο πρόβλημα της εύρεσης των ριζών του τριωνύμου μπορεί να οριστεί ένα αντικείμενο *Τριώνυμο* μέσα στο οποίο θα ενσωματωθούν οι σχετικοί υπολογισμοί. Ομοίως στο πρόβλημα της ενημέρωσης του αρχείου των φοιτητών μπορούμε να διακρίνουμε δύο βασικά αντικείμενα, *Φοιτητής* και *Αρχείο Φοιτητών*.

Ακριβέστερα στον αντικειμενοστρεφή προγραμματισμό κατασκευάζονται *κλάσεις*, που είναι προδιαγραφές, «καλούπια» για την κατασκευή των αντικειμένων που συμμετέχουν στο πρόγραμμα. Μια κλάση περιγράφει ένα *τύπο*, ένα *είδος*, με συγκεκριμένες ιδιότητες και λειτουργίες. Κάθε αντικείμενο που δημιουργείται από αυτή την κλάση διαθέτει τις ιδιότητες και τις λειτουργίες που περιγράφει η κλάση. Έτσι πιο σωστά, στο πρόγραμμα του τριωνύμου συμμετέχει ένα αντικείμενο της κλάσης *Τριώνυμο*, και στο πρόγραμμα των φοιτητών μερικά αντικείμενα της κλάσης *Φοιτητής* και ένα αντικείμενο της κλάσης *Αρχείο Φοιτητών*.

Με τεχνικούς όρους, κάθε κλάση διαθέτει *δεδομένα* και *διαδικασίες*. Τα δεδομένα αποθηκεύουν την κατάσταση των αντικειμένων της κλάσης. Για την κλάση *Τριώνυμο* τα δεδομένα αφορούν τους συντελεστές α, β, γ. Για την κλάση *Φοιτητής* τα δεδομένα αποθηκεύουν τα στοιχεία του φοιτητή (ονοματεπώνυμο, διεύθυνση, βαθμοί, κ.λπ.). Τέλος, για το *Αρχείο Φοιτητών* μπορεί να ενδιαφέρει αν το αρχείο είναι ανοιχτό ή όχι, πόσοι φοιτητές είναι καταχωρισμένοι σε αυτό, κ.ο.κ. Γενικά τα δεδομένα μιας κλάσης συγκροτούν την εσωτερική της αναπαράσταση, όπως είναι χρήσιμη στην ίδια την κλάση για να επιτελέσει τη λειτουργία της στο πρόγραμμα. Με αυτή την έννοια συνήθως τα δεδομένα είναι *ιδιωτικά*, δηλαδή δεν είναι άμεσα προσβάσιμα από τα αντικείμενα άλλων κλάσεων.

Οι διαδικασίες μιας κλάσης υλοποιούν τη συμπεριφορά της. Για παράδειγμα, το *Τριώνυμο* θα έχει διαδικασίες για τον υπολογισμό της διακρίνουσας και των ριζών του. Ο *Φοιτητής* θα έχει διαδικασίες για την εισαγωγή των στοιχείων και την εκτύπωσή τους, για την αποθήκευση και την ανάκτησή τους από το αρχείο, για τον υπολογισμό του τελικού βαθμού, κ.λπ. Το *Αρχείο Φοιτητών* θα έχει διαδικασίες για το άνοιγμα και κλείσιμο του αρχείου, για την εισαγωγή, αναζήτηση, ενημέρωση και διαγραφή κάποιας εγγραφής φοιτητή, κ.λπ. Γενικά τέτοιες διαδικασίες καθορίζουν την αλληλεπίδραση της κλάσης με τις υπόλοιπες κλάσεις πρέπει φυσικά να είναι προσβάσιμες από αυτές, με άλλα λόγια πρέπει να είναι *δημόσιες*. Σημειώνουμε ότι κατάλληλες δημόσιες διαδικασίες παρέχουν έμμεση και ελεγχόμενη πρόσβαση στα ιδιωτικά δεδομένα της κλάσης. Μία κλάση μπορεί επίσης να έχει και ιδιωτικές διαδικασίες, οι οποίες υλοποιούν υπολογισμούς απαραίτητους στην ίδια αλλά που δεν ενδιαφέρουν τον «έξω κόσμο».

Σε μια υποθετική γλώσσα αντικειμενοστρεφούς προγραμματισμού ο παραπάνω αλγόριθμος ενημέρωσης του αρχείου των φοιτητών θα έχει την εξής μορφή: (με *Αντικείμενο*→*Μέθοδος* σημειώνουμε τις κλήσεις των μεθόδων των αντικειμένων)

Αρχείο Φοιτητών → *ΑΝΟΙΓΜΑ*
ΕΠΑΝΑΛΗΨΗ για όλους τους φοιτητές που δίνονται
    Φοιτητής → *ΕΙΣΑΓΩΓΗ ΣΤΟΙΧΕΙΩΝ*
    Φοιτητής → *ΥΠΟΛΟΓΙΣΜΟΣ ΤΕΛΙΚΟΥ ΒΑΘΜΟΥ*
    Αρχείο Φοιτητών → *ΕΥΡΕΣΗ ΣΤΟΙΧΕΙΩΝ* του *Φοιτητή*
    ΑΝ υπάρχει ο φοιτητής ΤΟΤΕ
        Αρχείο Φοιτητών → *ΕΝΗΜΕΡΩΣΗ ΣΤΟΙΧΕΙΩΝ* του *Φοιτητή*
    ΔΙΑΦΟΡΕΤΙΚΑ
        Αρχείο Φοιτητών → *ΕΙΣΑΓΩΓΗ ΕΓΓΡΑΦΗΣ* με τα στοιχεία του *Φοιτητή*
Αρχείο Φοιτητών → *ΚΛΕΙΣΙΜΟ ΑΡΧΕΙΟΥ*

Η διάκριση μεταξύ του εξωτερικού (δημόσιου) και του εσωτερικού (ιδιωτικού) τμήματος των κλάσεων είναι ουσιαστικά αυτή που οδηγεί στην απομόνωση της πολυπλοκότητας στο εσωτερικό των αντικειμένων: κάθε κλάση επικοινωνεί με το περιβάλλον της μέσω του συγκεκριμένου και σταθερού δημόσιου τμήματός της, αλλά υλοποιεί τη λειτουργικότητά της κατά τρόπο αθέατο στο περιβάλλον της μέσα στο ιδιωτικό τμήμα της για το οποίο είναι αποκλειστικά υπεύθυνη.

Στη μείωση της πολυπλοκότητας συμβάλλει σημαντικά και η έννοια της *ιεραρχίας* αντικειμένων. Αυτή είναι το προγραμματιστικό ανάλογο της ιεραρχίας που συναντούμε στη φύση: για παράδειγμα, όλα τα φυτά έχουν κάποιες ιδιότητες και κάποιες λειτουργίες· τα λουλούδια και τα δένδρα είναι φυτά, επομένως *κληρονομούν* τις κοινές ιδιότητες και λειτουργίες των φυτών, συγχρόνως όμως έχουν και τις δικές τους ξεχωριστές ιδιότητες και λειτουργίες. Οι αντικειμενοστρεφείς γλώσσες προγραμματισμού υποστηρίζουν την κληρονομικότητα μεταξύ κλάσεων. Για παράδειγμα, η εφαρμογή που διαχειρίζεται τους φοιτητές είναι πιθανό να χρησιμοποιεί αρχεία διαφόρων τύπων (μαθημάτων, καθηγητών, κ.λπ.). Στην περίπτωση αυτή μπορεί να οριστεί μια γενικότερη κλάση *Αρχείο*, με ιδιότητες και μεθόδους που αφορούν γενικά όλους τους τύπους των αρχείων όπως πλήθος εγγραφών, άνοιγμα και κλείσιμο· τότε το *Αρχείο Φοιτητών* αλλά και οι άλλοι τύπου αρχείων θα κληρονομούν το *Αρχείο*, επομένως θα έχουν τις ιδιότητές του χωρίς να χρειάζεται να τις επαναλάβουν και θα μπορούν να χρησιμοποιήσουν τις υπηρεσίες του χωρίς να τις υλοποιήσουν πάλι. Περαιτέρω κάθε συγκεκριμένος τύπος (κλάση) αρχείων θα παρέχει διαδικασίες σχετικές με τη λειτουργία του: για το *Αρχείο Φοιτητών* τέτοιες αναφέρθηκαν παραπάνω και θα είναι σχετικές με την εισαγωγή, αναζήτηση, ενημέρωση και διαγραφή εγγραφών φοιτητών.

Το αντικειμενοστρεφές μοντέλο προγραμματισμού υποστηρίζεται από πολλές σύγχρονες γλώσσες καθώς και από νεότερες εκδοχές παλαιότερων γλωσσών που επεκτάθηκαν προς αυτή την κατεύθυνση. Ενδεικτικά αναφέρουμε τις γλώσσες Visual Basic, Delphi, Eiffel, Ada, C++, Java.

Αξίζει να σημειώσουμε ότι ο αρθρωτός και ο αντικειμενοστρεφής προγραμματισμός είναι συμπληρωματικά μοντέλα και το ένα δεν αποκλείει το άλλο. Ένα πρόβλημα μπορεί να επιλυθεί με μια σειρά ενεργειών (υποπρογραμμάτων) που υλοποιούνται με τη βοήθεια κατάλληλων αντικειμένων· αντίστροφα, τα αντικείμενα υλοποιούν τη λειτουργικότητά τους σε διαδικασίες στις οποίες εφαρμόζονται οι αρχές του αρθρωτού προγραμματισμού. Κατά την επίλυση ενός προβλήματος πρέπει και μπορεί να χρησιμοποιείται ο κατάλληλος συνδυασμός και των δύο τεχνικών.

2. Εισαγωγή στον Προγραμματισμό

Σε αυτή την ενότητα θα παρουσιάσουμε τις βασικές αρχές του προγραμματισμού. Όλες οι σύγχρονες γλώσσες προγραμματισμού διαθέτουν μια κοινή, περιορισμένη ομάδα δομών και εντολών με τις οποίες συνθέτονται τα προγράμματα, και αυτές θα αναπτύξουμε στα επόμενα. Καθώς τα θέματα που θα αναλύσουμε ισχύουν για όλες τις γλώσσες προγραμματισμού, δεν θα χρησιμοποιήσουμε κάποια συγκεκριμένα γλώσσα στα παραδείγματα αλλά θα εκφράσουμε τους αλγορίθμους σε *ψευδοκώδικα*, μια υποθετική γλώσσα προγραμματισμού· πάντως στο τέλος των σημειώσεων παρέχονται υλοποιήσεις όλων των προγραμμάτων σε C++.

2.1 Βασικά στοιχεία αλγορίθμων

Όπως έχουμε αναφέρει, οι αλγόριθμοι εκφράζουν λεπτομερώς και συστηματικά τη λύση προβλημάτων με Η/Υ. Στα προβλήματα αυτά κάποια δεδομένα υφίστανται επεξεργασία με τη βοήθεια συγκεκριμένων εντολών και παράγεται ένα αποτέλεσμα. Τα δεδομένα και οι εντολές είναι τα κύρια στοιχεία που απαρτίζουν ένα αλγόριθμο.

2.1.1 Μεταβλητές – Σταθερές

Τα δεδομένα που χειρίζεται ένας αλγόριθμος μπορεί να είναι αριθμοί, γράμματα, λέξεις, κ.λπ. και αποθηκεύονται σε *μεταβλητές* και *σταθερές*.

Μια *μεταβλητή* μπορεί να θεωρηθεί ως ένα όνομα με το οποίο μπορούμε να αναφερθούμε σε κάποιο δεδομένο του προβλήματος. Για παράδειγμα στο πρόβλημα επίλυσης του τριωνύμου τα α, β, γ, ρ που εμφανίζονται στον αλγόριθμο είναι μεταβλητές. Οι μεταβλητές μπορούν να πάρουν τιμή από το χρήστη, ως αποτέλεσμα της εισαγωγής δεδομένων, είτε από το ίδιο το πρόγραμμα, ως αποτέλεσμα κάποιων υπολογισμών. Για παράδειγμα,

```
a = 7
b = 5.5
sum = a+b
```

Στα επόμενα θα δίνουμε στις μεταβλητές ονόματα που περιέχουν χαρακτήρες του λατινικού αλφαβήτου και πιθανώς αριθμητικά ψηφία (ξεκινώντας πάντα με ένα γράμμα), π.χ. x, y, d, num1, num2, κ.ο.κ.

Σε κάθε πρόγραμμα που θα κατασκευάζουμε, θα *δηλώνουμε* τις μεταβλητές που θα χρησιμοποιήσουμε, στην αρχή του προγράμματος, γράφοντάς τες μετά τη χαρακτηριστική λέξη DATA. Για παράδειγμα ένα πρόγραμμα που χρησιμοποιεί τις μεταβλητές a, b, x, j, θα ξεκινά με τη δήλωση

```
DATA  a, b, x, j
```

Σταθερές υπάρχουν δύο ειδών: Καταρχήν όλες οι ποσότητες που εμφανίζονται σε ένα αλγόριθμο είναι σταθερές· τέτοιες είναι αριθμοί (π.χ. 4, 30, 7.5), γράμματα (περικλείονται μέσα σε μονά εισαγωγικά, π.χ. 'a', 'γ', '3') ή λέξεις (περικλείονται σε διπλά εισαγωγικά, π.χ. "day", "Τεχνικές προγραμματισμού", "R2D2"). Επίσης υπάρχουν *σταθερές με όνομα*· αυτές μοιάζουν πολύ με τις μεταβλητές

αλλά διαφέρουν στο ότι παίρνουν τιμή μια φορά στη διάρκεια του προγράμματος και αυτή η τιμή δεν μπορεί να αλλάξει. Χρησιμοποιούνται για να αποθηκεύσουν γνωστές ποσότητες ή χαρακτηριστικά μεγέθη του προβλήματος που δεν αλλάζουν κατά την εκτέλεση του προγράμματος. Οι σταθερές γράφονται συνήθως στην αρχή του προγράμματος και η δήλωσή τους γίνεται με τη χαρακτηριστική λέξη CONST, ταυτόχρονα με την απόδοση της (σταθερής) τιμής τους. Για παράδειγμα,

```
CONST pi = 3.14159
CONST maxsize = 500
```

Με όρους του υπολογιστή, μια μεταβλητή ή μια σταθερά με όνομα αντιπροσωπεύει μια θέση στη μνήμη του. Όταν χρησιμοποιούμε την τιμή της μεταβλητής ή της σταθεράς στην πραγματικότητα ανακτούμε την ποσότητα που είναι καταχωρισμένη σε εκείνη τη θέση μνήμης, ενώ όταν δίνουμε τιμή στη μεταβλητή ή στη σταθερά στην πραγματικότητα τοποθετούμε κάποια τιμή στη θέση μνήμης. Είναι παραστατικό να σκεφτόμαστε τις μεταβλητές σαν κουτιά στα οποία αποθηκεύουμε τα δεδομένα μας:

a :	7	pi :	3.14159
b :	5.5	maxsize :	500
sum :	12.5		

2.1.2 Εντολές

Οι εντολές που επεξεργάζονται τα δεδομένα του αλγορίθμου γενικά διαφέρουν μεταξύ των διαφορετικών γλωσσών προγραμματισμού. Υπάρχουν όμως κάποιες βασικές εντολές που είναι κοινές για όλες τις γλώσσες· με τις εντολές αυτές μπορεί να εκφραστεί οποιοσδήποτε αλγόριθμος. Αυτές οι εντολές είναι:

- Εντολές *εισόδου και εξόδου δεδομένων*. Με αυτές εισάγονται δεδομένα στο πρόγραμμα και αντίστοιχα παρουσιάζονται τα αποτελέσματα στο χρήστη. Μπορούμε να υποθέσουμε ότι τα δεδομένα δίνονται από το πληκτρολόγιο και τα αποτελέσματα εμφανίζονται στην οθόνη του υπολογιστή.
- Εντολή *απόδοσης τιμής*. Με αυτή την εντολή δίνεται τιμή στις μεταβλητές και σταθερές του προγράμματος.
- Εντολή *επιλογής*. Εξετάζεται μια συνθήκη και αν ισχύει εκτελείται ένα συγκεκριμένο τμήμα του προγράμματος ενώ αν δεν ισχύει παραλείπεται και πιθανώς εκτελείται ένα άλλο τμήμα.
- Εντολή *επανάληψης*. Ένα συγκεκριμένο τμήμα του προγράμματος εκτελείται επαναληπτικά για όσο διάστημα ισχύει κάποια συνθήκη.

Στα επόμενα θα παρουσιάσουμε αναλυτικά τις εντολές αυτές και τη χρήση τους. Οι γλώσσες προγραμματισμού παρέχουν επιπρόσθετα άλλες εντολές, που είτε είναι τροποποιημένες μορφές των παραπάνω για κάποιες ειδικότερες χρήσεις (μερικές από αυτές είναι μάλιστα αρκετά κοινές και θα τις δούμε παρακάτω), είτε πιο σύνθετες εντολές που διευκολύνουν τον προγραμματιστή.

2.1.3 Παραστάσεις

Οι μεταβλητές και οι σταθερές συνδυάζονται σε *παραστάσεις*. Παραστάσεις σχηματίζονται όταν γίνονται πράξεις μεταξύ μεταβλητών και σταθερών, καθώς και όταν συγκρίνονται μεταβλητές ή σταθερές. Οι μεταβλητές ή σταθερές που συμμετέχουν σε μια παράσταση ονομάζονται *όροι* και τα σύμβολα *τελεστές*.

Ο παρακάτω πίνακας περιέχει τους βασικούς τελεστές που θα χρησιμοποιούμε:

Τελεστής	**Όνομα**	**Παράδειγμα**
+	Θετικό πρόσημο	+12
–	Αρνητικό πρόσημο	– 12
!	Λογική άρνηση	! true
*	Πολλαπλασιασμός	3 * 4
/	Διαίρεση	4 / 7
%	Υπόλοιπο ακέραιας διαίρεσης	5 % 2 (=1)
+	Πρόσθεση	3+5
–	Αφαίρεση	r – 2
<, <=, >, >=	Συγκρίσεις μεγέθους	a < b, c >= d
==	Σύγκριση για ισότητα	a == 3
!=	Σύγκριση για ανισότητα	a != 5
AND ή &&	Λογικό «ΚΑΙ»	a && b
OR ή ||	Λογικό «Ή»	a || b

Η προτεραιότητα των πράξεων είναι αυτή που γνωρίζουμε από τα μαθηματικά· εξάλλου στον παραπάνω πίνακα οι τελεστές παρουσιάζονται κατά σειρά προτεραιότητας. Παρενθέσεις μπορούν να χρησιμοποιηθούν όπου χρειάζεται να ομαδοποιηθούν παραστάσεις.

2.1.4 Ψευδοκώδικας – Λογικά Διαγράμματα

Οι αλγόριθμοι πρέπει να εκφράζονται σε μια τυπική μορφή ώστε η μετατροπή τους σε πρόγραμμα υπολογιστή να είναι εύκολη και άμεση. Για το σκοπό αυτό έχουν αναπτυχθεί διάφοροι τρόποι παράστασής τους. Εδώ θα παρουσιάσουμε και θα χρησιμοποιήσουμε *ψευδοκώδικα* και *λογικά διαγράμματα*.

Ο ψευδοκώδικας είναι μια υποθετική γλώσσα προγραμματισμού· διατηρεί τα βασικά συντακτικά στοιχεία μιας γλώσσας αλλά ταυτόχρονα δεν έχει την αυστηρότητά της και μπορεί να διαμορφωθεί σχετικά ελεύθερα ώστε να εξυπηρετεί στην παράσταση των αλγορίθμων. Παραδείγματα ψευδοκώδικα είναι τα αποσπάσματα «προγραμμάτων» που παρουσιάσαμε στην προηγούμενη ενότητα (τριώνυμο, αρχείο φοιτητών).

Στον ψευδοκώδικα οι εντολές εισόδου και εξόδου δεδομένων θα παριστάνονται με READ και WRITE αντίστοιχα. Για παράδειγμα με τις εντολές

```
READ x, y
WRITE x, 4, y
```

θα εισαχθούν από το χρήστη τιμές στις μεταβλητές x και y και κατόπιν θα τυπωθούν κατά σειρά η τιμή του x, ο αριθμός 4 και η τιμή του y.

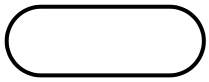
Η εντολή απόδοσης τιμής θα παριστάνεται με =, για παράδειγμα

```
a = 4
D = b*b – 4*a*c
```

Στην πρώτη από αυτές τις εντολές δίνεται η τιμή 4 στη μεταβλητή a. Στη δεύτερη εντολή υπολογίζεται η παράσταση στα δεξιά του = (υποτίθεται ότι τα b, a, c είναι επίσης μεταβλητές που έχουν πάρει τιμές μέχρι εκείνο το σημείο του προγράμματος) και το αποτέλεσμα του υπολογισμού αποδίδεται στη μεταβλητή D. Πρέπει να τονιστεί ότι αντίθετα από το συνηθισμένο «=» που γράφουμε στα Μαθηματικά, η εντολή απόδοσης τιμής δεν είναι συμμετρική, δηλαδή έχει σημασία τι γράφεται στα αριστερά του = και τι στα δεξιά του: στα αριστερά γράφεται πάντα κάποια μεταβλητή και στα δεξιά κάποια παράσταση, η τιμή της οποίας υπολογίζεται και αποδίδεται στη μεταβλητή.

Η παράσταση των υπόλοιπων εντολών θα δοθεί παρακάτω.

Τα λογικά διαγράμματα είναι ένας σχηματικός τρόπος για την παράσταση των αλγορίθμων. Κάθε εντολή παριστάνεται με ένα κουτί και η ροή του προγράμματος δηλώνεται με μια κατευθυνόμενη γραμμή. Βασικά σύμβολα των λογικών διαγραμμάτων είναι τα επόμενα:



Το σχήμα αυτό χρησιμοποιείται για να δηλώσει την αρχή ή το τέλος του λογικού διαγράμματος. Στο εσωτερικό του γράφουμε BEGIN ή END αντίστοιχα.

Το παραλληλόγραμμο χρησιμοποιείται για την είσοδο και έξοδο δεδομένων. Στο εσωτερικό του γράφουμε READ ή WRITE αντίστοιχα και τα στοιχεία που θα διαβαστούν ή εμφανιστούν.

Το ορθογώνιο χρησιμοποιείται για να περιγράψει μια εντολή, με εξαίρεση την επιλογή. Στο εσωτερικό του γράφεται η εντολή, για παράδειγμα «b=5».

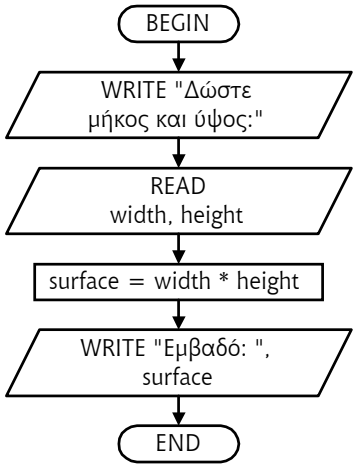
Τα σύμβολα των υπόλοιπων εντολών θα δοθούν στις αντίστοιχες παραγράφους.

Παράδειγμα 1. Εμβαδό παραλληλογράμμου

Το πρώτο πρόγραμμα που θα κατασκευάσουμε υπολογίζει το εμβαδό ενός τυχαίου παραλληλογράμμου.

Όπως γνωρίζουμε, το εμβαδό ενός παραλληλογράμμου ισούται με το γινόμενο του μήκους της βάσης του επί το ύψος του. Επομένως τα δεδομένα που θα επεξεργαστεί ο αλγόριθμος είναι το μήκος και το ύψος του παραλληλογράμμου καθώς και το εμβαδό του· για καθένα από αυτά απαιτείται μια μεταβλητή, και τις καλούμε αντίστοιχα width, height και surface. Ο αλγόριθμος θα ζητεί από το χρήστη το μήκος και το ύψος του παραλληλογράμμου, κατόπιν θα τα πολλαπλασιάζει ώστε να υπολογίσει το εμβαδό, και τέλος θα εμφανίζει το εμβαδό στην οθόνη. Ο ψευδοκώδικας και το λογικό διάγραμμα που ακολουθούν υλοποιούν αυτή την περιγραφή:

```
DATA  width, height, surface

WRITE  "Δώστε μήκος και ύψος:"
READ  width, height
surface = width * height
WRITE  "Εμβαδό: ", surface
```



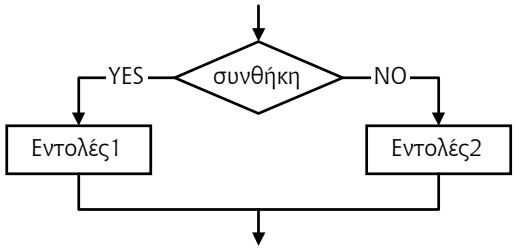
2.2 Εντολές επιλογής

Με τις εντολές επιλογής αποφασίζεται αν θα εκτελεστούν συγκεκριμένα τμήματα ενός αλγορίθμου ανάλογα με συνθήκες που μεταβάλλονται κατά την εκτέλεσή του.

2.2.1 Απλή επιλογή

Η εντολή απλής επιλογής έχει σε ψευδοκώδικα και λογικά διαγράμματα αντίστοιχα τη μορφή

```
IF συνθήκη
      Εντολές1
ELSE
      Εντολές2
END IF
```

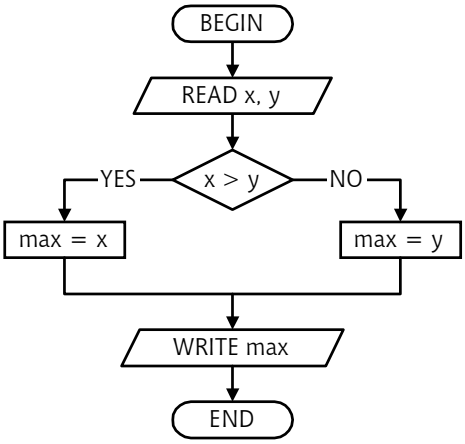


Η «συνθήκη» είναι μια λογική παράσταση που μπορεί να είναι αληθής ή ψευδής. Αν είναι αληθής εκτελούνται οι εντολές «Εντολές1» ενώ αν είναι ψευδής εκτελούνται οι «Εντολές2». Το τμήμα ELSE της εντολής επιλογής είναι προαιρετικό, δηλαδή υπάρχει η δυνατότητα να μην εκτελεστεί καμία εντολή αν η συνθήκη είναι ψευδής και το πρόγραμμα να συνεχιστεί με τις επόμενες εντολές. Το END IF δηλώνει το τέλος της εντολής IF και διαχωρίζει τις εντολές που της αντιστοιχούν από το υπόλοιπο πρόγραμμα.

Παράδειγμα 1. Μέγιστος μεταξύ δύο αριθμών

Έστω ότι ο χρήστης δίνει δύο αριθμούς και ζητούμε τον μεγαλύτερο από αυτούς. Ο παρακάτω αλγόριθμος τους συγκρίνει και τυπώνει τον μεγαλύτερο:

```
DATA  x, y, max

READ  x, y
IF  x > y
      max = x
ELSE
      max = y
END IF
WRITE  max
```



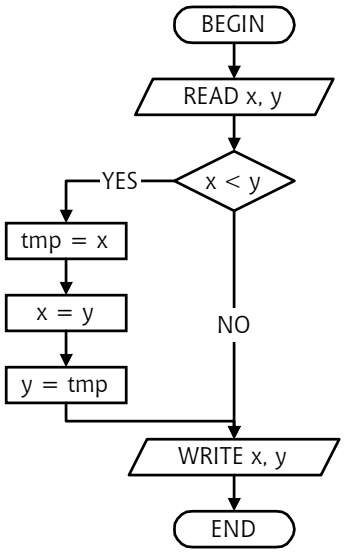
Παράδειγμα 2. Ταξινόμηση δύο αριθμών

Έστω ότι ο χρήστης δίνει δύο αριθμούς x και y. Ζητείται ένα πρόγραμμα που να τοποθετεί τον μεγαλύτερο από αυτούς στη μεταβλητή x και το μικρότερο στην y.

Στην περίπτωση αυτή πρέπει να συγκριθούν οι αριθμοί, και αν δεν βρίσκονται στις κατάλληλες θέσεις να εναλλαχθούν· στην αντίθετη περίπτωση δεν χρειάζεται να γίνει τίποτε και γιαυτό δεν υπάρχει τμήμα ELSE στην εντολή IF στον ψευδοκώδικα που ακολουθεί.

Στον αλγόριθμο που ακολουθεί έχει ενδιαφέρον η διαδικασία εναλλαγής των τιμών των δύο μεταβλητών x και y (οι τρεις γραμμές που εκτελούνται μέσα στο IF): η τιμή του x καταχωρίζεται στη βοηθητική μεταβλητή tmp και κατόπιν «κυκλικά» δίνονται στο x η τιμή του y και στο y η τιμή της tmp, δηλαδή η αρχική τιμή της x.

```
DATA  x, y, tmp

READ  x, y
IF  x < y
      tmp = x
      x = y
      y = tmp
END IF
WRITE  x, y
```



Παράδειγμα 3. Μέγιστος τριών αριθμών

Υποθέτουμε ότι δίνονται τρεις αριθμοί και θέλουμε τον μεγαλύτερο από αυτούς. Αυτός θα προκύψει με διαδοχικές συγκρίσεις των αριθμών ανά δύο. Στον παρακάτω ψευδοκώδικα αξίζει να προσεχθεί πώς οι εσοχές τονίζουν τη δομή του αλγορίθμου: οι εντολές που «ανήκουν» σε κάποιο IF ή ELSE ομαδοποιούνται με το να γράφονται πιο μέσα· επίσης τα ELSE και END IF στοιχίζονται στην ίδια στήλη με το IF στο οποίο αναφέρονται.

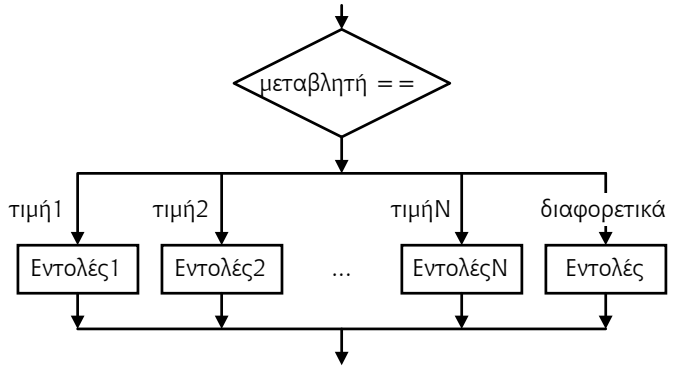
```
DATA  a, b, c, max
```

```
READ  a, b, c
IF a > b
      IF a > c
            max = a
      ELSE
            max = c
      END IF
ELSE
      IF c > b
            max = c
      ELSE
            max = b
      END IF
END IF
WRITE  max
```

2.2.2 Πολλαπλή επιλογή

Η εντολή πολλαπλής επιλογής είναι μια επέκταση της απλής επιλογής που παρέχεται από πολλές σύγχρονες γλώσσες προγραμματισμού και γιαυτό θα την παρουσιάσουμε εδώ. Η εντολή έχει τη μορφή:

```
SWITCH  μεταβλητή
      CASE τιμή1 :
            Εντολές1
      CASE τιμή2 :
            Εντολές2
      ...
      CASE τιμήN :
            ΕντολέςN
      DEFAULT :
            Εντολές
END SWITCH
```



Με την εντολή πολλαπλής επιλογής εξετάζεται η τιμή της «μεταβλητής» και αν είναι μία από τις «τιμή1», «τιμή2», …, «τιμήN» εκτελούνται οι αντίστοιχες εντολές, διαφορετικά εκτελούνται οι εντολές που αντιστοιχούν στο τμήμα DEFAULT.

Το τμήμα DEFAULT είναι ανάλογο του ELSE της εντολής IF και, όπως και εκείνο, είναι προαιρετικό. Συχνά όμως το γεγονός ότι η μεταβλητή παίρνει τιμή διαφορετική από αυτές που έχουν προβλεφθεί είναι ένδειξη σφάλματος, και τότε το τμήμα DEFAULT περιλαμβάνει εντολές για την αντιμετώπισή του.

Τονίζεται ότι η μεταβλητή που εξετάζεται σε μια εντολή πολλαπλής επιλογής πρέπει να παίρνει *διακριτές* τιμές, δηλαδή ακέραιους αριθμούς ή γράμματα, ενώ δεν μπορεί να παίρνει τιμές πραγματικούς αριθμούς ή λέξεις, καθώς με αυτούς δεν μπορεί να γίνει επακριβής ή άμεση σύγκριση.

Η παραπάνω εντολή πολλαπλής επιλογής είναι ισοδύναμη με τη σειρά εντολών IF :

```
IF  μεταβλητή == τιμή1
      Εντολές1
ELSE
      IF  μεταβλητή == τιμή2
            Εντολές2
      ELSE
            ...
                  IF  μεταβλητή == τιμήN
                        ΕντολέςN
                  ELSE
                        Εντολές
                  END IF
      END IF
END IF
```

Μάλιστα μια τέτοια ακολουθία εντολών IF μπορεί να χρησιμοποιηθεί στην περίπτωση που η μεταβλητή επιλογής δεν επιτρέπεται να χρησιμοποιηθεί σε εντολή SWITCH γιατί παίρνει τιμές πραγματικούς αριθμούς ή γενικότερα μη διακριτού τύπου.

Παράδειγμα 1. Εκτύπωση αριθμών

Σε ένα πρόγραμμα για την εκμάθηση των αριθμών, το παιδί δίνει ένα αριθμό από 1 ως 5 και το ζητούμε ένα πρόγραμμα που να εμφανίζει την ανάλογη λέξη.

Αυτή είναι μια χαρακτηριστική περίπτωση εφαρμογής της εντολής πολλαπλής επιλογής, καθώς έχουμε μια μεταβλητή (τον αριθμό που εισάγεται) που παίρνει ακέραιες τιμές και ανάλογα με την τιμή του πρέπει να γίνει και διαφορετική ενέργεια.

```
DATA  n

READ  n
SWITCH  n
      CASE 1 :
            WRITE "One"
      CASE 2 :
            WRITE "Two"
      CASE 3 :
            WRITE "Three"
      CASE 4 :
            WRITE "Four"
      CASE 5 :
            WRITE "Five"
      DEFAULT :
            WRITE "Λάθος αριθμός"
END SWITCH
```

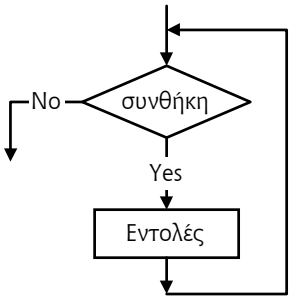
2.3 Εντολές επανάληψης

Οι εντολές επανάληψης προκαλούν την εκτέλεση μιας σειράς εντολών επαναληπτικά, για όσο διάστημα ισχύει κάποια συνθήκη.

2.3.1 Επανάληψη με συνθήκη στην αρχή

Η βασική εντολή επανάληψης είναι η εντολή WHILE. Αυτή έχει την εξής μορφή:

```
WHILE  συνθήκη
      Εντολές
END WHILE
```



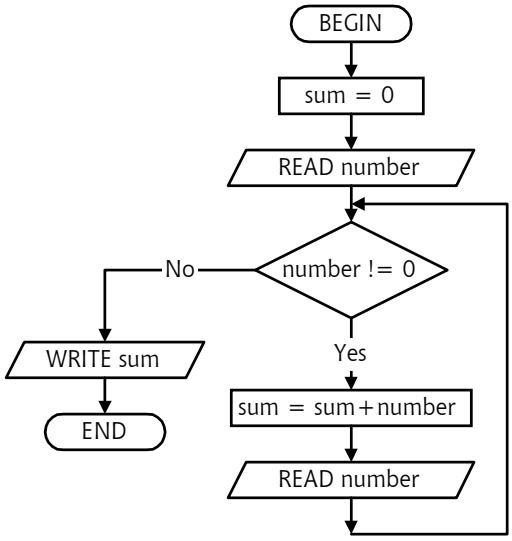
Η «συνθήκη» εξετάζεται στην αρχή, και αν είναι αληθής εκτελούνται οι «Εντολές». Αφού εκτελεστούν, η «συνθήκη» υπολογίζεται ξανά και αν ισχύει πάλι οι «Εντολές» επαναλαμβάνονται. Ο βρόχος εκτελείται μέχρι η συνθήκη να γίνει ψευδής, και κατόπιν η ροή εκτέλεσης του προγράμματος συνεχίζεται με τις επόμενες εντολές (βέλος «No» στο λογικό διάγραμμα). Είναι προφανές ότι αν η συνθήκη δεν αληθεύει την πρώτη φορά που συναντάται κατά την εκτέλεση του προγράμματος, τότε οι «Εντολές» δεν θα εκτελεστούν καθόλου.

Καθώς μεταξύ των διαδοχικών υπολογισμών της «συνθήκης» εκτελούνται μόνο οι «Εντολές», πρέπει αυτές με κάποιο τρόπο να οδηγούν προς την έξοδο από το βρόχο. Για παράδειγμα, αν η συνθήκη εξετάζει αριθμούς που εισάγονται από το χρήστη θα πρέπει μέσα στις «Εντολές» να εισάγεται κάθε φορά ο επόμενος αριθμός· αν η συνθήκη εξετάζει την τιμή μιας παράστασης πρέπει να βεβαιωνόμαστε ότι οι υπολογισμοί που γίνονται μέσα στις «Εντολές» δίνουν στην παράσταση τέτοιες τιμές ώστε μετά από κάποιο αριθμό βημάτων να σταματήσουν οι επαναλήψεις.

Παράδειγμα 1. Άθροισμα αριθμών

Το επόμενο πρόγραμμα διαβάζει αριθμούς που εισάγει ο χρήστης και υπολογίζει το άθροισμά τους. Το τέλος των αριθμών δηλώνεται με την εισαγωγή του 0.

```
DATA  number, sum

sum = 0
READ  number
WHILE  number != 0
      sum = sum+number
      READ  number
END WHILE
WRITE  sum
```



Ο υπολογισμός του αθροίσματος γίνεται καταχωρίζοντας στη μεταβλητή sum το «τρέχον άθροισμα»: κάθε αριθμός που διαβάζεται προστίθεται στην τιμή της sum και έτσι κάθε φορά αυτή πε-

ριέχει το άθροισμα των αριθμών που έχουν εισαχθεί μέχρι εκείνο το σημείο· με τον τρόπο αυτό στο τέλος του προγράμματος θα περιέχει το συνολικό άθροισμα.

Η εντολή απόδοσης τιμής « sum = sum+number » παρουσιάζει ιδιαίτερο ενδιαφέρον καθώς η μεταβλητή sum εμφανίζεται τόσο στα αριστερά όσο και στα δεξιά του «=» και είναι εύλογο να δημιουργηθούν απορίες ως προς την εκτέλεσή της. Όμως αυτό που συμβαίνει κατά την εκτέλεση μιας οποιασδήποτε εντολής απόδοσης τιμής είναι να υπολογίζεται πρώτα η τιμή του δεξιού μέρους της με τις τρέχουσες τιμές των μεταβλητών και κατόπιν, σε ένα δεύτερο στάδιο, αυτή η τιμή να αποδίδεται στη μεταβλητή της οποίας το όνομα είναι στο αριστερό μέρος. Έτσι στην περίπτωση αυτού του παραδείγματος δικαιολογούνται όσα γράψαμε πιο πάνω: σε κάθε βήμα προστίθεται ο αριθμός που διαβάζεται στην τρέχουσα τιμή της sum και το αποτέλεσμα καταχωρίζεται και πάλι στην sum ώστε να χρησιμοποιηθεί στο επόμενο βήμα της επανάληψης.

Σημειώνουμε επίσης ότι είναι απαραίτητο να δώσουμε στη μεταβλητή sum *αρχική τιμή* (ίση με 0, όση το άθροισμα των 0 αριθμών που έχουν εισαχθεί στην αρχή του προγράμματος). Αν δεν το κάνουμε, τότε αυτή θα χρησιμοποιηθεί για πρώτη φορά μέσα στο βρόχο WHILE, αφού διαβαστεί ο πρώτος αριθμός, στη σχέση sum=sum+number. Θα προσπαθήσουμε δηλαδή να προσθέσουμε τον αριθμό number στην τρέχουσα τιμή της sum, η οποία όμως δεν υπάρχει! Στην πράξη, αν τρέχαμε αυτό το πρόγραμμα σε υπολογιστή και παραλείπαμε την απόδοση αρχικής τιμής στη sum, ο υπολογιστής θα της έδινε μια αυθαίρετη τιμή και έτσι το τελικό άθροισμα θα ήταν εντελώς λανθασμένο.

Παράδειγμα 2. Δυνάμεις

Να δοθεί αλγόριθμος που υπολογίζει τη σειρά των αριθμών

$$x, x^2, x^3, \ldots, x^n$$

όπου x και n είναι ακέραιοι αριθμοί που δίνονται από το χρήστη.

Οι ζητούμενες δυνάμεις μπορούν να υπολογιστούν με μια επαναληπτική διαδικασία που θα εκτελεστεί n φορές: σε κάθε βήμα μπορούμε να υπολογίσουμε μία δύναμη του x απλώς πολλαπλασιάζοντας την προηγούμενη δύναμη επί x. Ονομάζουμε number τον αριθμό (δύναμη του x) που υπολογίζεται σε κάθε βήμα. Θα χρειαστούμε επίσης μια μεταβλητή, έστω power, με τη βοήθεια της οποίας θα μετρούμε τις δυνάμεις που υπολογίζουμε.

```
DATA  x, n, number, power

READ  x, n
number = x
power = 1
WHILE  power <= n
      WRITE  number
      number = number * x
      power = power + 1
END WHILE
```

Παράδειγμα 3. Άθροισμα ψηφίων αριθμού

Ζητείται αλγόριθμος που υπολογίζει το άθροισμα των ψηφίων ενός τυχαίου ακεραίου αριθμού. Για παράδειγμα, αν δίνεται ο αριθμός 175 το αποτέλεσμα πρέπει να είναι 1+7+5=13.

Παρατηρούμε ότι τα ψηφία του αριθμού μπορούν να προκύψουν ως υπόλοιπα διαδοχικών διαιρέσεών του με το 10 (σε αντίστροφη σειρά, αλλά κάτι τέτοιο δεν μας επηρεάζει). Για τον παραπάνω αριθμό θα έχουμε: (Ο τελεστής % συμβολίζει το υπόλοιπο της ακέραιας διαίρεσης, και στη δεξιά στήλη γράφουμε τα ακέραια πηλίκα των διαιρέσεων με το 10)

175 % 10 = **5**	175 / 10 = 17
17 % 10 = **7**	17 / 10 = 1
1 % 10 = **1**	1 / 10 = 0

Είναι φανερό ότι η αλληλουχία υπολογισμού του υπολοίπου και διαίρεσης με το 10 πρέπει να τεθεί σε μια επαναληπτική διαδικασία προκειμένου να λάβουμε όλα τα ψηφία του αριθμού. Επιπλέον μόλις προκύπτει ένα νέο ψηφίο θα πρέπει να προστίθεται στο άθροισμα των ψηφίων. Η επαναληπτική διαδικασία μπορεί να σταματήσει μόλις το πηλίκο της διαίρεσης γίνει 0.

```
DATA  a, sum, digit

READ  a
sum = 0
WHILE  a > 0
      digit = a % 10
      sum = sum+digit
      a = a / 10
END WHILE
WRITE  sum
```

Παράδειγμα 4. Αριθμητικές σειρές

Σε δοκιμασίες αξιολόγησης ευφυΐας συχνά δίνονται σειρές αριθμών όπως η

2, 5, 11, 20, 32,...

και ζητείται να συνεχιστούν. Παρακάτω θα παρουσιάσουμε έναν αλγόριθμο που κατασκευάζει την παραπάνω σειρά, τυπώνοντας τους αριθμούς της μέχρι να φτάσουν ένα όριο που καθορίζει ο χρήστης.

Για να κατασκευαστεί αυτή η σειρά (και αντίστοιχα για να μπορέσει να τη συνεχίσει ο εξεταζόμενος) πρέπει να γίνει κατανοητή η σχέση μεταξύ των όρων της. Στη συγκεκριμένη περίπτωση κάθε όρος προκύπτει αν προστεθεί 3, 6, 9, 12,... στον προηγούμενο, με πρώτο όρο το 2.

Για την υλοποίηση αυτής της διαδικασίας είναι απαραίτητος ένας βρόχος επανάληψης, ο οποίος σε κάθε βήμα θα υπολογίζει το ποσό που πρέπει να προστεθεί στον τρέχοντα αριθμό, θα κάνει την πρόσθεση και θα τυπώνει τον νέο αριθμό. Οι επαναλήψεις θα συνεχίζονται εφόσον ο τρέχων αριθμός είναι μικρότερος ή ίσος με το όριο που έχει θέσει ο χρήστης. Οι μεταβλητές που χρειάζονται έγιναν φανερές από την περιγραφή: μία μεταβλητή num θα αποθηκεύει τον τρέχοντα αριθμό και μία add το ποσό που θα προστεθεί σε αυτόν για να προκύψει ο επόμενος. Το άνω όριο των αριθμών θα συμβολιστεί maxnum και ο πρώτος όρος firstnum.

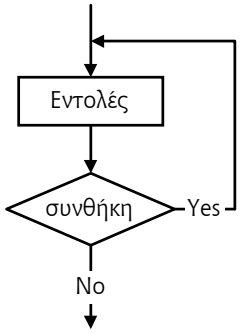
```
DATA  firstnum, maxnum, add

firstnum = 2
add = 3
READ  maxnum
num = firstnum
WHILE  num <= maxnum
      WRITE  num
      num = num+add
      add = add+3
END WHILE
```

Σε αυτό το πρόγραμμα ο βρόχος WHILE οδηγείται προς το τέλος του με την εντολή num=num+add. Επίσης αν το όριο maxnum που εισάγει ο χρήστης είναι μικρότερο από 2 ο βρόχος δεν θα εκτελεστεί καμία φορά και δεν θα εκτυπωθεί κανένας αριθμός, όπως άλλωστε είναι και το σωστό. Παρατηρούμε, τέλος, ότι αυτό το πρόγραμμα είναι αρκετά *γενικό*: με αλλαγές στις αριθμητικές τιμές των firstnum και add καθώς και στην εντολή add=add+3 μπορούν να παραχθούν διαφορετικές αριθμητικές σειρές του ίδιου τύπου. Για σειρές με διαφορετικές προδιαγραφές (για παράδειγμα, σειρές που κάθε όρος εξαρτάται από δύο προηγούμενους) απαιτείται βέβαια διαφορετικός αλγόριθμος.

2.3.2 Επανάληψη με συνθήκη στο τέλος

Μια μικρή παραλλαγή της προηγούμενης εντολής επανάληψης είναι η εντολή DO ... WHILE. Αυτή έχει την παρακάτω μορφή:

```
DO
    Εντολές
WHILE συνθήκη
```



Όπως φαίνεται, η εντολή DO ... WHILE διαφέρει από την απλή εντολή WHILE στο ότι οι «Εντολές» εκτελούνται τουλάχιστον μία φορά πριν εξεταστεί η συνθήκη και αποφασιστεί αν θα επαναληφθούν ή αν θα προχωρήσει παρακάτω η εκτέλεση του αλγορίθμου. Παρατηρούμε ότι σε αυτή την εντολή δεν χρειάζονται λέξεις τερματισμού (όπως τα END IF, END WHILE παραπάνω) καθώς οι επαναλαμβανόμενες εντολές διαχωρίζονται από τις επόμενες με το τελικό WHILE.

Η εντολή DO ... WHILE παρέχει διευκόλυνση στον προγραμματιστή σε αρκετές περιπτώσεις και έτσι είναι χρήσιμη αντί της απλής εντολής WHILE. Πρέπει να επισημάνουμε πάντως ότι οι δυο εντολές είναι ισοδύναμες και οποιοσδήποτε βρόχος χρησιμοποιεί τη μία μπορεί με λίγη προσοχή να μετατραπεί ώστε να χρησιμοποιεί την άλλη.

Παράδειγμα 1. Εισαγωγή στοιχείων

Στον τελικό βαθμό ενός μαθήματος υπολογίζονται με 75% ο βαθμός της εξέτασης και με 25% ο μέσος όρος δύο ασκήσεων. Αφού έχει συγκεντρώσει τους βαθμούς κάθε φοιτητή, ο καθηγητής θέλει ένα πρόγραμμα για να κάνει τις απαραίτητες πράξεις. Το πρόγραμμα θα διαβάζει το βαθμό της εξέτασης και τους δύο βαθμούς των ασκήσεων, θα κάνει τους σχετικούς υπολογισμούς και θα τυπώνει τον τελικό βαθμό. Αν υπάρχουν και άλλοι φοιτητές θα πρέπει να επαναλαμβάνει τη διαδικασία.

```
DATA exam, exer1, exer2, final, answer

DO
    WRITE "Δώστε βαθμό εξέτασης, 1ης & 2ης άσκησης"
    READ exam, exer1, exer2
    final = 0.75*exam + 0.25*(exer1+exer2)/2.0
    WRITE "Τελικός βαθμός: ", final
    WRITE "Να συνεχίσω;"
    READ answer
WHILE answer == 'y'
```

Στο πρόβλημα αυτό είναι φανερό ότι ταιριάζει καλύτερα μια εντολή επανάληψης DO … WHILE: θα εισαχθούν τα στοιχεία τουλάχιστον ενός φοιτητή πριν ερωτηθεί ο χρήστης αν θέλει να συνεχίσει.

Παράδειγμα 2. Μάντεψε τον αριθμό!

Ο ακόλουθος αλγόριθμος είναι ένα παιχνίδι μαντείας αριθμών. Ο υπολογιστής έχει ένα μυστικό αριθμό και ο χρήστης καλείται να τον μαντέψει, με την εξής διαδικασία: εισάγει αριθμούς και κάθε φορά ο υπολογιστής τον πληροφορεί αν ο αριθμός που έδωσε είναι μεγαλύτερος ή μικρότερος από το μυστικό αριθμό. Όταν ο χρήστης βρει τον αριθμό, ο υπολογιστής τον πληροφορεί πόσες προσπάθειες χρειάστηκε.

Παρουσιάζουμε δύο υλοποιήσεις του αλγορίθμου, με χρήση τόσο της εντολής DO … WHILE όσο και της απλής WHILE, ώστε να γίνει εμφανής η ισοδυναμία τους αλλά και οι διαφορές τους.

```
DATA  secret, guess, tries

secret = 135

tries = 0
DO
      READ  guess
      tries = tries+1
      IF  guess > secret
            WRITE  "Πολύ μεγάλος!"
      ELSE
            IF  guess < secret
                  WRITE  "Πολύ μικρός!"
            END IF
      END IF
WHILE  guess != secret

WRITE  "Βρέθηκε μετά από ", tries,
          " προσπάθειες"
```

```
DATA  secret, guess, tries

secret = 135

READ  guess
tries = 1
WHILE  guess != secret
      IF  guess > secret
            WRITE  "Πολύ μεγάλος!"
      ELSE
            WRITE  "Πολύ μικρός!"
      END IF
      READ  guess
      tries = tries+1
END WHILE

WRITE  "Βρέθηκε μετά από ", tries,
          " προσπάθειες"
```

Οι δύο υλοποιήσεις διαφέρουν ελαφρά στην τοποθέτηση των εντολών READ με τις οποίες ο χρήστης εισάγει τον αριθμό του σε κάθε βήμα· αυτή η διαφοροποίηση είναι απαραίτητη ώστε να μπορούν κάθε φορά να πραγματοποιηθούν οι έλεγχοι των WHILE στις θέσεις που βρίσκονται (στο τέλος του βρόχου στην πρώτη περίπτωση, στην αρχή του στη δεύτερη). Όμως η κυριότερη διαφορά μεταξύ των δύο υλοποιήσεων είναι ότι στη δεύτερη δεν υπάρχει η ανάγκη για το IF guess<secret που εμφανίζεται στη πρώτη: αυτός ο έλεγχος στην πρώτη υλοποίηση είναι απαραίτητος ώστε να αποκλειστεί η περίπτωση να βρήκε ο χρήστης το μυστικό αριθμό· αντίθετα στη δεύτερη περίπτωση αυτό έχει ήδη ελεγχθεί, από το WHILE στην αρχή του βρόχου, οπότε πλέον ο αριθμός θα είναι είτε μικρότερος είτε μεγαλύτερος του μυστικού.

Παράδειγμα 3. Ημέρες και ημερομηνίες

Αν είναι γνωστή η πρώτη ημέρα ενός μήνα και το πλήθος των ημερών του, να δοθεί ένας αλγόριθμος που βρίσκει όλες τις ημερομηνίες του μήνα που αντιστοιχούν σε μια συγκεκριμένη ημέρα της εβδομάδας. Για παράδειγμα αν ο μήνας έχει 31 μέρες και ξεκινά Τρίτη, τότε Πέμπτη είναι στις 3, 10, 17, 24, 31.

Για τον αλγόριθμο αυτό είναι απαραίτητο να κωδικοποιήσουμε τις ημέρες τις εβδομάδας, ώστε να μπορούμε να τις χειριστούμε εύκολα· κάνουμε λοιπόν την αντιστοιχία: Δευτέρα = 1, Τρίτη = 2,..., Κυριακή = 7.

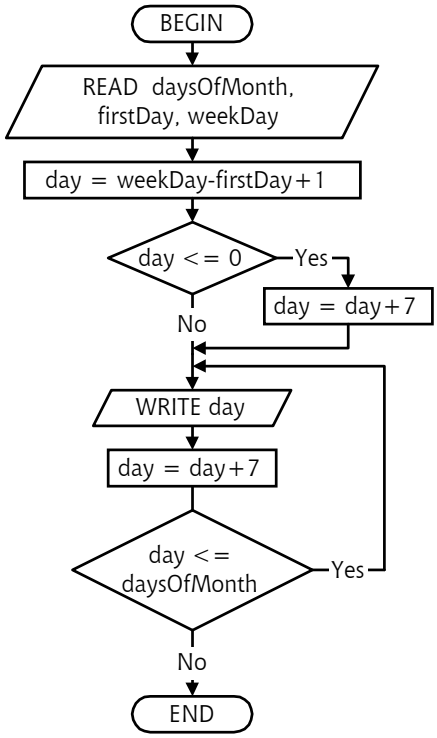
Το πρώτο που πρέπει να υπολογίσουμε είναι ποια είναι η πρώτη ημερομηνία που αντιστοιχεί στη ζητούμενη ημέρα. Αν καλέσουμε firstDay την πρώτη ημέρα του μήνα και weekDay την ημέρα της εβδομάδας που μας ενδιαφέρει, τότε μπορούμε να δούμε ότι αυτή η ημερομηνία είναι η

weekDay – firstDay + 1

Πράγματι, για το παράδειγμα που δίνουμε είναι firstDay = 2, weekDay = 4 και η σχέση αυτή δίνει 3. Παρατηρούμε όμως ότι αν weekDay < firstDay τότε η σχέση αυτή δίνει αρνητικό αποτέλεσμα, με άλλα λόγια μας μεταφέρει στον προηγούμενο μήνα. Για να διορθώσουμε αυτό πρέπει σε μια τέτοια περίπτωση να προσθέσουμε μία εβδομάδα (7) στο αποτέλεσμα ώστε να βρεθούμε στην πρώτη εβδομάδα του μήνα που μας ενδιαφέρει.

Αφού βρήκαμε την πρώτη ημερομηνία, ο υπολογισμός των υπολοίπων είναι απλός: αρκεί να προσθέτουμε 7 στην ημερομηνία αυτή μέχρι να φτάσουμε στο τέλος του μήνα.

```
DATA  daysOfMonth, firstDay, weekDay, day

READ  daysOfMonth, firstDay, weekDay

day = weekDay – firstDay + 1
IF  day <= 0
      day = day + 7
END IF

DO
      WRITE  day
      day = day + 7
WHILE  day <= daysOfMonth
```



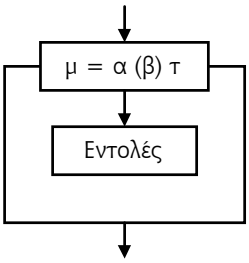
2.3.3 Επανάληψη σταθερού αριθμού φορών

Στις εντολές επανάληψης που εξετάσαμε προηγουμένως, η απόφαση αν θα επαναληφθεί ένα συγκεκριμένο τμήμα του αλγορίθμου εξαρτάται από την εκπλήρωση κάποιας συνθήκης. Μια ειδική μορφή επανάληψης είναι αυτή κατά την οποία ένα τμήμα του αλγορίθμου εκτελείται ένα *συγκεκριμένο αριθμό φορών*. Με χρήση μιας εντολής WHILE αυτή η μορφή επανάληψης μπορεί να υλοποιηθεί με τη βοήθεια ενός *μετρητή*, δηλαδή μιας μεταβλητής που απαριθμεί τις επαναλήψεις. Ο μετρητής λαμβάνει μια *αρχική τιμή*, στο τέλος κάθε επανάληψης αυξάνεται η τιμή του κατά ένα *βήμα επανάληψης* και οι επαναλήψεις συνεχίζονται εφόσον η τιμή του είναι μικρότερη από μια *τελική τιμή*. Σχηματικά:

```
count = αρχική τιμή
WHILE  count <= τελική τιμή
      Εντολές
      count = count + βήμα
END WHILE
```

Αυτή η μορφή επανάληψης συναντάται πολύ συχνά στην πράξη, και έτσι παρέχεται μια ειδική εντολή ώστε να είναι πιο εύκολη η χρήση της:

```
FOR  μ = α (β) τ
      Εντολές
END FOR
```



Εδώ «μ» είναι ο μετρητής, «α» η αρχική τιμή, «τ» η τελική, και «β» το βήμα. Ο μετρητής είναι μια μεταβλητή όπως όλες οι υπόλοιπες που έχουμε δει. Όταν ξεκινά ο βρόχος FOR λαμβάνει τιμή ίση με την αρχική. Μέσα στις «Εντολές» μπορεί να χρησιμοποιηθεί όπως οποιαδήποτε άλλη μεταβλητή. Μόλις εκτελεστούν οι «Εντολές» η τιμή του μετρητή αυξάνεται κατά το βήμα και εξετάζεται αν είναι ακόμα μικρότερος ή ίσος της τελικής τιμής. Αν είναι, οι «Εντολές» επαναλαμβάνονται, ο μετρητής αυξάνεται πάλι, κ.ο.κ., ενώ διαφορετικά το πρόγραμμα συνεχίζεται παρακάτω. Για παράδειγμα, ο επόμενος βρόχος τυπώνει όλους τους αριθμούς από το 1 μέχρι το 20:

```
DATA  count
FOR  count = 1 (1) 20
      WRITE count
END FOR
```

Το βήμα μπορεί να είναι θετικό, οπότε η τιμή του μετρητή αυξάνεται σε κάθε επανάληψη, ή αρνητικό, οπότε η τιμή του μειώνεται. Αν το βήμα είναι θετικό και η αρχική τιμή είναι μικρότερη από την τελική, τότε ο βρόχος δεν θα εκτελεστεί καθόλου· αντίστοιχα δεν θα εκτελεστεί αν το βήμα είναι αρνητικό και η αρχική τιμή μεγαλύτερη από την αρχική.

Παράδειγμα 1. Άθροισμα ακεραίων μεταξύ ορίων

Ο επόμενος αλγόριθμος αθροίζει όλους τους ακέραιους μεταξύ δύο αριθμών που δίνει ο χρήστης (συμπεριλαμβάνονται και αυτοί). Εξετάζεται ποιος από τους δύο αριθμούς είναι μεγαλύτερος ώστε να χρησιμοποιηθεί κατάλληλη εντολή FOR:

```
DATA  a, b, sum, i

READ  a, b
sum = 0
IF  a < b
      FOR  i = a(1)b
            sum = sum+i
      END FOR
ELSE
      FOR  i = a(-1)b
```

```
        sum = sum+i
    END FOR
END IF
WRITE  sum
```

Παράδειγμα 2. Άθροισμα περιττών αριθμών

Ο αλγόριθμος που ακολουθεί αθροίζει τους περιττούς αριθμούς μεταξύ δύο αριθμών που δίνει ο χρήστης (εδώ υποθέτουμε ότι δίνονται σε αύξουσα σειρά, για ευκολία). Εδώ ενδιαφερόμαστε να «επισκεφτούμε» τους αριθμούς ανά δύο, ξεκινώντας βέβαια από περιττό αριθμό. Έτσι το βήμα επανάληψης πρέπει να είναι 2. Επίσης πρέπει να εξεταστεί αν ο πρώτος αριθμός είναι περιττός, αλλιώς να του προστεθεί 1 πριν την αρχή των επαναλήψεων ώστε να γίνει περιττός.

```
DATA  a, b, sum, i

READ  a, b
sum = 0
IF  a%2 == 0
    a = a+1
END IF
FOR  i = a(2)b
    sum = sum+i
END FOR
WRITE sum
```

Παράδειγμα 3. Πρώτοι αριθμοί

Να γραφτεί ένα πρόγραμμα που να διαβάζει έναν ακέραιο αριθμό n και στη συνέχεια να βρίσκει όλους του πρώτους αριθμούς από το 1 μέχρι το n και να τους τυπώνει. (Πρώτοι είναι οι αριθμοί που διαιρούνται μόνο με τον εαυτό τους και τη μονάδα.)

Το κύριο μέρος του αλγορίθμου είναι αυτό που θα αποφαίνεται αν ένας ακέραιος, έστω k, είναι πρώτος ή όχι. Για να γίνει αυτό θα πρέπει να δοκιμάζονται όλοι οι ακέραιοι μέχρι τον k (μέχρι και τον k–1), αν διαιρούνται με αυτόν. (Μπορεί να δειχτεί ότι αυτό αρκεί να γίνει για τους ακεραίους μέχρι και τη $\sqrt{k}$.) Μόλις βρεθεί κάποιος που διαιρείται οι έλεγχοι πρέπει να σταματούν καθώς είναι σίγουρο ότι ο k δεν είναι πρώτος. Η διαδικασία φυσικά θα εκτελεστεί για όλους τους ακεραίους k από το 1 μέχρι το n ώστε να ολοκληρωθεί το ζητούμενο πρόγραμμα.

```
DATA  n, k, p, isPrime

READ  n
FOR  k = 1(1)n
    isPrime = true
    p = 2
    WHILE  (isPrime == true  &&  p <= √k )
        IF  k%p == 0
            isPrime = false
        ELSE
            p = p+1
        END IF
    END WHILE
    IF  isPrime == true
        WRITE  k
```

```
    END IF
END FOR
```

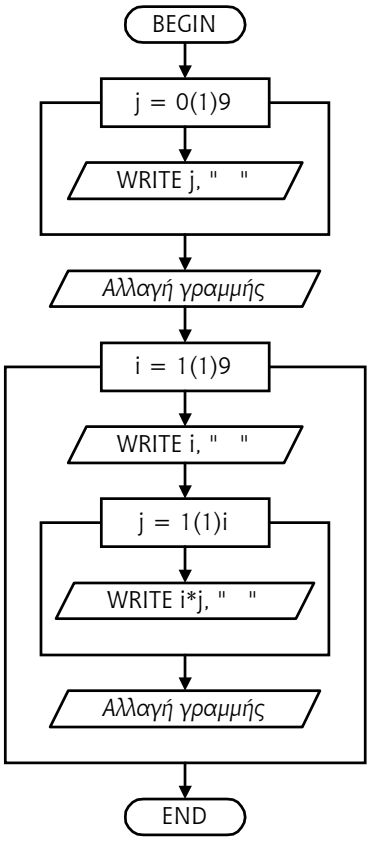
Παράδειγμα 4. Πίνακας πολλαπλασιασμού

Να γραφτεί ένα πρόγραμμα που να τυπώνει τον παρακάτω πίνακα πολλαπλασιασμού:

0	1	2	3	4	5	6	7	8	9
1	1								
2	2	4							
3	3	6	9						
4	4	8	12	16					
5	5	10	15	20	25				
6	6	12	18	24	30	36			
7	7	14	21	28	35	42	49		
8	8	16	24	32	40	48	56	64	
9	9	18	27	36	45	54	63	72	81

Ας θεωρήσουμε καταρχήν ένα τυχαίο στοιχείο του πίνακα, έστω για παράδειγμα το 21. Αυτό βρίσκεται στη γραμμή i=7 και στη στήλη j=3, και είναι ακριβώς το γινόμενο i*j. Για να παραγάγουμε όλη τη γραμμή στην οποία ανήκει, δηλαδή τη γραμμή με αριθμό i=7 θα πρέπει να τυπώσουμε όλους τους αριθμούς 7*j για j=1(1)7 ή, αντικαθιστώντας το i, i*j για j=1(1)i. Για να εκτυπωθεί όλος ο πίνακας πρέπει με αυτό τον τρόπο να εκτυπωθούν όλες οι γραμμές του, δηλαδή η διαδικασία που περιγράψαμε να γίνει για i=1(1)9. Ο αλγόριθμος που ακολουθεί υλοποιεί όσα περιγράψαμε:

```
DATA  i, j

(Εκτύπωση της πρώτης γραμμής)
FOR  j = 0(1)9
     WRITE  j, "   "
END FOR
Αλλαγή γραμμής

(Εκτύπωση υπόλοιπου πίνακα)
FOR  i = 1(1)9
     WRITE  i, "   "  (Πρώτη στήλη)
     FOR  j = 1(1)i
          WRITE  i*j, "   "
     END FOR
     Αλλαγή γραμμής
END FOR
```



2.4 Πίνακες

Ας θεωρήσουμε ένα πρόβλημα παρόμοιο με κάποια που έχουμε δει προηγουμένως: *Δίνονται* κάποιοι αριθμοί και ζητούμε το άθροισμά τους. Σε σχέση με τα προηγούμενα διαφοροποιούμαστε στο ότι οι αριθμοί δίνονται εξαρχής και δεν εισάγονται ένας-ένας από τον χρήστη (τέτοιες περιπτώσεις είναι κοινές σε πολύπλοκα προγράμματα όπου τα δεδομένα δίνονται μια φορά αλλά υφίστανται επεξεργασία σε πολλά σημεία του προγράμματος). Επομένως θα πρέπει να αποθηκευτούν όλοι οι αριθμοί σε κατάλληλες μεταβλητές και κατόπιν να υπολογιστεί το άθροισμά τους. Αν υποθέσουμε ότι έχουμε τρεις αριθμούς, έστω 3, 5, και 10, μια υλοποίηση θα έμοιαζε ως εξής:

```
num1 = 3
num2 = 5
num3 = 10
sum = num1+num2+num3
```

Η υλοποίηση αυτή παρουσιάζει σημαντικά μειονεκτήματα. Καταρχήν είναι περιορισμένη αποκλειστικά σε τρεις αριθμούς· αν θέλαμε να κάνουμε το ίδιο για τέσσερις αριθμούς θα έπρεπε να χρησιμοποιήσουμε μια νέα μεταβλητή num4 και να θυμηθούμε να την προσθέσουμε στο άθροισμα και πιθανώς όπου αλλού χρειαζόταν αντίστοιχα σε ένα μεγαλύτερο πρόγραμμα. Επιπλέον με τον τρόπο αυτό οι αριθμοί δεν αντιμετωπίζονται ως μια ομάδα, όπως είναι στην πραγματικότητα, αλλά ως ανεξάρτητες ποσότητες που απλά προστίθενται.

Οι γλώσσες προγραμματισμού παρέχουν τη δυνατότητα να αντιμετωπίσουμε ομάδες όμοιων δεδομένων με ενιαίο τρόπο, με δομές που καλούνται *πίνακες*. Ένας πίνακας είναι ακριβώς μια συλλογή από αντικείμενα του ίδιου τύπου, αποθηκευμένα σε σειρά, και τα οποία μπορούν να προσπελαστούν ενιαία, με ένα κοινό όνομα και ένα *δείκτη* που οδηγεί στη θέση του κάθε αντικειμένου στη σειρά.

Στην §2.1.1 είδαμε ότι μια μεταβλητή αντιπροσωπεύει μία θέση στη μνήμη του υπολογιστή. Με αντίστοιχους όρους, ένας πίνακας αντιπροσωπεύει μία σειρά (ίδιων) θέσεων στη μνήμη του υπολογιστή και ο δείκτης μας δίνει τη δυνατότητα να αναφερθούμε σε κάθε μία από αυτές. Έτσι ένας πίνακας με όνομα num στον οποίο αποθηκεύονται οι παραπάνω τρεις αριθμοί μπορεί να παρασταθεί ως εξής:

num :	3	5	10
	1	2	3

Ένα οποιοδήποτε στοιχείο του πίνακα συμβολίζεται με το όνομα του πίνακα ακολουθούμενο από το δείκτη μέσα σε αγκύλες. Για παράδειγμα, το προηγούμενο τμήμα ψευδοκώδικα γίνεται:

```
num[1] = 3
num[2] = 5
num[3] = 10
sum = num[1]+num[2]+num[3]
```

Όπως είπαμε, υποθέτουμε ότι έχουμε ορίσει ένα πίνακα num, αποθηκεύουμε τρία στοιχεία στις θέσεις του με δείκτες 1, 2, 3 και μετά χρησιμοποιούμε τα στοιχεία αυτά για τον υπολογισμό του αθροίσματος. Με τον τρόπο αυτό το άθροισμα μπορεί πλέον να υπολογιστεί με τρόπο παρόμοιο με αυτόν που έχουμε δει,

```
sum = 0
FOR i=1(1)3
    sum = sum+num[i]
END FOR
```

καθώς τίποτα δεν απαγορεύει στον δείκτη μέσα στις αγκύλες να είναι και αυτός μια μεταβλητή. Πλέον τα στοιχεία αντιμετωπίζονται ως μια ομάδα, μια ακολουθία, και επίσης η τροποποίηση του κώδικα είναι πολύ ευκολότερη καθώς αρκεί να αλλάξει το άνω όριο του βρόχου FOR.

Εδώ θα περιοριστούμε σε πίνακες που το μέγεθός τους είναι γνωστό στην αρχή του προγράμματος και παραμένει σταθερό κατά την εκτέλεσή του· πολλές γλώσσες προγραμματισμού δίνουν περισσότερη ελευθερία στο θέμα αυτό, αλλά κάτι τέτοιο δεν θα μας απασχολήσει. Οι πίνακες πρέπει να δηλώνονται στο τμήμα DATA του ψευδοκώδικα όπως και οι υπόλοιπες μεταβλητές. Μάλιστα το όνομα ενός πίνακα θα συνοδεύεται από δύο αγκύλες μέσα στις οποίες θα γράφεται το μέγεθός του. Έτσι για παράδειγμα στον παραπάνω ψευδοκώδικα υπολογισμού του αθροίσματος αριθμών χρειάζεται η δήλωση

```
DATA num[3], sum, i
```

Το μέγεθος του πίνακα στη δήλωση μπορεί να δίνεται είτε με έναν αριθμό, όπως εδώ, είτε με χρήση κάποιας σταθεράς που έχει πάρει τιμή πριν τη δήλωση του πίνακα. Για το σκοπό αυτό όμως δεν μπορεί να χρησιμοποιηθεί κάποια μεταβλητή, καθώς, όπως αναφέραμε, το μέγεθος των πινάκων δεν μπορεί να μεταβληθεί κατά την εκτέλεση του προγράμματος. Για παράδειγμα, μία εναλλακτική δήλωση για τον παραπάνω ψευδοκώδικα είναι

```
CONST size = 3
DATA num[size], sum, i
```

Στα επόμενα θα ακολουθήσουμε τη σύμβαση της γλώσσας C++ για τους δείκτες των πινάκων: θα ξεκινούν πάντα από το 0 και έτσι ένας πίνακας με n στοιχεία θα έχει θέσεις με δείκτες 0, 1,..., n-1. Αυτή η σύμβαση σε αρκετές περιπτώσεις προκαλεί δυσκολίες (θα ήταν πολύ πιο φυσιολογικό οι δείκτες να είναι 1, 2,..., n ή πιο βολικό να μπορούμε να ορίσουμε εμείς τα όρια) αλλά θα βοηθήσει στην πιο άμεση μεταφορά των αλγορίθμων που θα παρουσιαστούν σε προγράμματα C++.

Παράδειγμα 1. Ελάχιστο στοιχείο πίνακα και θέση του

Σε ένα πίνακα καταχωρίζουμε 10 στοιχεία. Ζητείται να εμφανίσουμε το ελάχιστο στοιχείο του πίνακα και τη θέση που βρίσκεται.

Ο ζητούμενος υπολογισμός μπορεί να γίνει σταδιακά. Ξεκινώντας υποθέτουμε ότι το πρώτο στοιχείο του πίνακα είναι το ελάχιστο· κατόπιν εξετάζουμε διαδοχικά ένα-ένα τα στοιχεία του, και αν κάποιο βρεθεί μικρότερο από το θεωρούμενο ως ελάχιστο θέτουμε αυτό ως ελάχιστο και συνεχίζουμε με τα υπόλοιπα στοιχεία του πίνακα. Ο ψευδοκώδικας που ακολουθεί υλοποιεί αυτή την ιδέα. Επιπλέον, καθώς ζητείται και η θέση του ελάχιστου στοιχείου στον πίνακα, κάθε φορά που ενημερώνεται το ελάχιστο στοιχείο αποθηκεύεται σε μία μεταβλητή η θέση του.

```
DATA numbers[10], i, min, minpos

(Εισαγωγή αριθμών)
FOR i = 0(1)9
    READ numbers[i]
END FOR
```

```
(Υπολογισμός ελαχίστου και θέσης του)
min = numbers[0]
minpos = 0
FOR  i = 1(1)9
      IF  numbers[i] < min
            min = numbers[i]
            minpos = i
      END IF
END FOR
WRITE  min, minpos
```

Παράδειγμα 2. Σύγκριση πινάκων

Ζητείται ένα πρόγραμμα που συγκρίνει δύο πίνακες (του ίδιου πλήθους στοιχείων). Το πρόγραμμα θα πρέπει να συγκρίνει τα στοιχεία των πινάκων που βρίσκονται στις αντίστοιχες θέσεις· αν όλα τα ζεύγη αντίστοιχων στοιχείων είναι όμοια τότε οι πίνακες είναι προφανώς ίδιοι, αλλιώς διαφέρουν. Θα πρέπει να εκτυπώνεται κατάλληλο μήνυμα και στη δεύτερη περίπτωση το πλήθος των πρώτων όμοιων στοιχείων.

```
CONST  n = 5  (Ενδεικτική τιμή για το μέγεθος των πινάκων)
DATA  a[n], b[n], i, equal

(Εισαγωγή πινάκων από το χρήστη)
FOR  i = 0(1)n-1
      READ  a[i]
END FOR
FOR  i = 0(1)n-1
      READ  b[i]
END FOR

(Έλεγχος ισότητας)
equal = true
i = 0
WHILE  (i < n  &&  equal == true)
      IF  a[i] != b[i]
            equal = false
      ELSE
            i = i+1
      END IF
END WHILE

IF  equal == true
      WRITE  "Οι πίνακες είναι ίδιοι"
ELSE
      WRITE "Μόνο τα ", i, " πρώτα στοιχεία είναι ίσα"
END IF
```

Στον παραπάνω ψευδοκώδικα καταρχήν οι δύο πίνακες ζητούνται από το χρήστη χωριστά, με χρήση βρόχων FOR για να εισαχθούν κατά σειρά οι τιμές των στοιχείων τους. Κατόπιν για την εξέταση της ισότητας των πινάκων χρησιμοποιείται ένας βρόχος WHILE, και αυτό γιατί οι συγκρίσεις μπορούν να διακοπούν μόλις βρεθεί κάποιο ζεύγος αντίστοιχων στοιχείων που δεν είναι όμοια. Χρησιμο-

ποιώντας βρόχους WHILE πρέπει να προσέξουμε να κάνουμε εμείς όλη τη δουλειά που ο βρόχος FOR κάνει αυτόματα, δηλαδή να δώσουμε αρχική τιμή στο δείκτη και να αυξάνουμε την τιμή του μετά από κάθε επανάληψη· αν παραλείψουμε το τελευταίο οι δείκτες θα παραμένουν σταθεροί και το πρόγραμμα δεν θα τερματίζει καθώς θα εξετάζεται συνεχώς το ίδιο στοιχείο του πίνακα! Η λογική μεταβλητή equal τίθεται αρχικά ίση με true γιατί υποθέτουμε ότι οι πίνακες είναι ίσοι· γίνεται false μόλις βρεθεί ζεύγος ανόμοιων στοιχείων και χρησιμεύει για τη διακοπή του WHILE.

Τέλος αξίζει να παρατηρήσουμε ότι καταχωρίζοντας το μέγεθος του πίνακα σε μια σταθερά, εδώ στην n, μπορούμε να μεταβάλουμε το μέγεθος που επεξεργάζεται το πρόγραμμα αλλάζοντας μόνο την τιμή της σταθεράς αυτής.

Παράδειγμα 3. Επόμενη ημερομηνία

Ζητείται ένα πρόγραμμα που να τυπώνει την επόμενη μιας ημερομηνίας που εισάγει ο χρήστης. Για παράδειγμα αν εισαχθεί η ημερομηνία «28 Μαΐου 2001» το πρόγραμμα θα πρέπει να τυπώσει «29 Μαΐου 2001», αν εισαχθεί «31 Μαρτίου 1950» θα πρέπει να τυπώσει «1 Απριλίου 1950» κ.ο.κ. Για απλοποίηση του προβλήματος δεν λαμβάνουμε υπόψη τα δίσεκτα έτη. Επίσης υποθέτουμε ότι ο χρήστης εισάγει την ημερομηνία στη μορφή τριών αριθμών, κατά σειρά ημέρα, μήνας, έτος, και τα δεδομένα που δίνει είναι έγκυρα.

Για την επίλυση αυτού του προβλήματος παρατηρούμε ότι στην απλούστερη περίπτωση η ζητούμενη ημερομηνία προκύπτει προσθέτοντας 1 στις ημέρες που δίνονται και αφήνοντας τα υπόλοιπα στοιχεία αμετάβλητα. Διαφορετική αντιμετώπιση απαιτείται, όπως φαίνεται και από το δεύτερο παράδειγμα που δόθηκε παραπάνω, αν ο χρήστης ζητήσει την επόμενη της τελευταίας ημέρας κάποιου μήνα· τότε η επόμενη ημέρα είναι η 1η του επόμενου μήνα. Επίσης ιδιαίτερη περίπτωση είναι η 31η Δεκεμβρίου, της οποίας επόμενη είναι η 1η Ιανουαρίου του επόμενου έτους.

Το σκεπτικό αυτό μπορεί σχετικά άμεσα να διαμορφωθεί σε πρόγραμμα. Χρειαζόμαστε τρεις μεταβλητές d1, m1, y1 στις οποίες θα καταχωρισθούν τα δεδομένα της ημερομηνίας που δίνει ο χρήστης και άλλες τρεις d2, m2, y2 στις οποίες θα καταχωρισθεί η επόμενη ημερομηνία. Ο ψευδοκώδικας που ακολουθεί ελέγχει τις δύο ειδικές περιπτώσεις που αναφέραμε ξεκινώντας από την πιο «ειδική» (τέλος του έτους), προχωρώντας στην λιγότερο «ειδική» (τέλος κάποιου μήνα) και αφήνοντας τελευταία τη γενικότερη περίπτωση.

```
DATA  d1, m1, y1, d2, m2, y2
CONST  monthDays[13] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }

READ  d1, m1, y1
IF  (d1 == 31  &&  m1 == 12)
     d2 = 1
     m2 = 1
     y2 = y1+1
ELSE
     IF  d1 == monthDays[m1]
          d2 = 1
          m2 = m1+1
     ELSE
          d2 = d1+1
          m2 = m1
     END IF
     y2 = y1
END IF
WRITE  "Επόμενη ημερομηνία: ", d2, m2, y2
```

Παράδειγμα 4. Μετατόπιση στοιχείων πίνακα

Δίνεται ένας πίνακας a με n στοιχεία και ένας αριθμός k, $0 \le k \le n$. Να δοθεί αλγόριθμος ο οποίος μετατοπίζει τα στοιχεία του πίνακα a κατά k θέσεις προς τα δεξιά. Στις κενές θέσεις του a που προκύπτουν από τη μετατόπιση τοποθετούνται μηδενικά, ενώ τα στοιχεία που φεύγουν από τη δεξιά πλευρά του εξαφανίζονται. Στον αλγόριθμο δεν πρέπει να χρησιμοποιηθεί βοηθητικός πίνακας για την προσωρινή καταχώριση των στοιχείων. Για παράδειγμα, αν

$$n = 5,\ a = \{12, 6, 4, 15, 20\},\ k = 2$$

θα προκύψει ο πίνακας

$$a = \{0, 0, 12, 6, 4\}$$

Το ουσιαστικό ενδιαφέρον του αλγορίθμου είναι το γέμισμα των τελευταίων $n-k$ θέσεων του πίνακα, καθώς η εισαγωγή μηδενικών στις πρώτες k θέσεις είναι τετριμμένη. Παρατηρούμε ότι τα στοιχεία του τελικού πίνακα στις θέσεις αυτές προκύπτουν από τη σχέση

$$a[i] = a[i-k]$$

που πρέπει να εφαρμοστεί για τα στοιχεία με δείκτες i από k μέχρι $n-1$ (ας θυμηθούμε ότι η αρίθμηση των δεικτών ξεκινά από 0). Χρειάζεται προσοχή όμως στη σειρά που θα γεμίσουμε τις θέσεις αυτές. Αν κινηθούμε από τα αριστερά προς τα δεξιά τότε για το παραπάνω παράδειγμα θα έχουμε:

$$a[2] = a[0] = 12$$
$$a[3] = a[1] = 6$$
$$a[4] = a[2] = 12 \quad \text{(Λάθος!)}$$

Παρατηρούμε ότι κινούμενοι από αριστερά προς τα δεξιά στον πίνακα καλύπτουμε στοιχεία που θα μας χρειαστούν στη συνέχεια. Για να αποφύγουμε αυτή την επιπλοκή πρέπει να γεμίσουμε τις θέσεις από δεξιά προς τα αριστερά, από το τέλος προς την αρχή, καλύπτοντας πρώτα τις θέσεις των στοιχείων που θα εξαφανιστούν. Για τον ίδιο λόγο η εισαγωγή των μηδενικών στα αριστερότερα στοιχεία του πίνακα πρέπει να γίνει *αφού* τακτοποιήσουμε τα υπόλοιπα· σε αντίθετη περίπτωση θα καλύψουμε τα στοιχεία που χρειαζόμαστε.

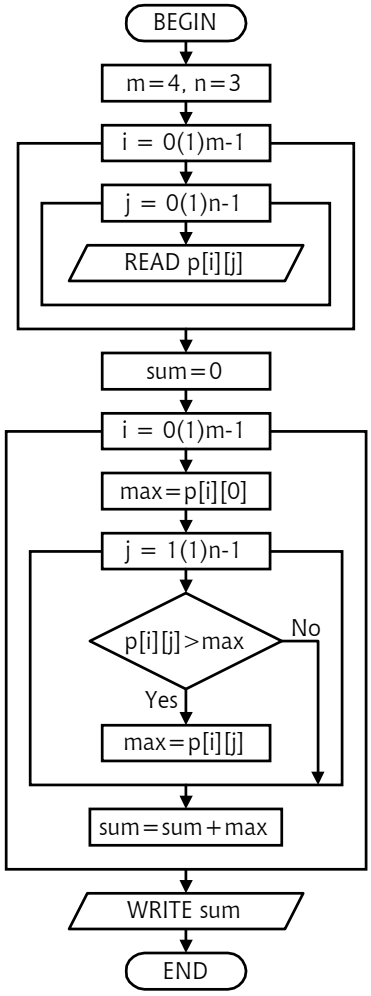
```
CONST  n = 5
DATA  a[n], k, i

(Εισαγωγή στοιχείων)
FOR  i = 0(1)n-1
     READ  a[i]
END FOR
READ  k
(Μετακίνηση στοιχείων)
FOR  i = n-1(-1)k
     a[i] = a[i-k]
END FOR
(Εισαγωγή μηδενικών)
FOR  i = 0(1)k-1
     a[i] = 0
END FOR
(Εκτύπωση αποτελέσματος)
FOR  i = 0(1)n-1
     WRITE  a[i]
END FOR
```

Παράδειγμα 5. Πίνακες δύο διαστάσεων

Οι πίνακες που παρουσιάσαμε μέχρι τώρα είναι *μονοδιάστατοι*. Εύκολα όμως μπορούμε να τους επεκτείνουμε σε δύο ή και περισσότερες διαστάσεις: ένας πίνακας δύο διαστάσεων είναι μια συλλογή στοιχείων οργανωμένων σε γραμμές και στήλες, που μπορούν να προσπελαστούν με χρήση δύο δεικτών, για τις γραμμές και τις στήλες αντίστοιχα· μια πρώτη μορφή είχαμε παρουσιάσει στο Παράδειγμα 4 της §2.3.3, όπου όμως δεν αποθηκεύαμε τα στοιχεία σε κάποιο πίνακα αλλά τα επεξεργαζόμασταν ένα-ένα. Σε ένα πίνακα δύο διαστάσεων, έστω p, το στοιχείο που βρίσκεται στην i γραμμή και j στήλη θα συμβολίζεται p[i][j].

Ο επόμενος αλγόριθμος διαβάζει έναν πίνακα διαστάσεων m×n (m γραμμών και n στηλών, όπου m και n είναι σταθερές που καθορίζονται στην αρχή) και τυπώνει το άθροισμα των μέγιστων στοιχείων των γραμμών του.

```
CONST  m = 3
CONST  n = 4
DATA  p[m][n], i, j, max, sum

FOR  i = 0(1)m-1
     FOR  j = 0(1)n-1
          READ  p[i][j]
     END FOR
END FOR

sum = 0
FOR  i = 0(1)m-1
     (Υπολογισμός μεγίστου γραμμής)
     max = p[i][0]
     FOR  j = 1(1)n-1
          IF  p[i][j] > max
               max = p[i][j]
          END IF
     END FOR

     (Υπολογισμός αθροίσματος)
     sum = sum+max
END FOR

WRITE  sum
```



Για καλύτερη εποπτεία δίνουμε έναν πίνακα 3×4:

3	5	1	0	**5**
7	12	-1	7	**12**
5	9	8	7	**9**
		Άθροισμα μεγίστων:		**26**

Οι διαδικασίες που απαιτούνται για αυτό τον αλγόριθμο μας είναι γνωστές από προηγούμενες εφαρμογές τους: εύρεση μεγίστου κάποιων αριθμών (ανάλογα με το Παράδειγμα 1) και άθροιση α-

ριθμών. Στην περίπτωση που εξετάζουμε συνδυάζονται και χρησιμοποιούνται κατάλληλα τα στοιχεία του πίνακα. Κινούμαστε στον πίνακα κατά γραμμές (εξωτερικό FOR με το δείκτη i), για κάθε γραμμή υπολογίζουμε το μέγιστο στοιχείο διατρέχοντας τις στήλες της (εσωτερικό FOR με το δείκτη j), και το προσθέτουμε στο άθροισμα των μεγίστων που διατηρούμε στη μεταβλητή sum.

Παράδειγμα 6. Συμμετρικοί πίνακες

Ο ακόλουθος αλγόριθμος διαβάζει έναν πίνακα matrix διαστάσεων n×n (n κάποια σταθερά που ορίζεται στην αρχή) και εξετάζει αν είναι συμμετρικός, δηλαδή αν τα στοιχεία που είναι σε συμμετρικές θέσεις ως προς τη διαγώνιο είναι ίδια. Σημειώνουμε ότι το συμμετρικό του στοιχείου matrix[i][j] είναι το στοιχείο matrix[j][i].

```
CONST  n = 4  (Ενδεικτική τιμή)
DATA  matrix[n][n], i, j, symmetric

FOR  i = 0(1)n-1
     FOR  j = 0(1)n-1
          READ  matrix[i][j]
     END FOR
END FOR

symmetric = true
i = 0
WHILE  (i < n  &&  symmetric == true)
     j = 0
     WHILE  (j < n  &&  symmetric == true)
          IF  matrix[i][j] != matrix[j][i]
               symmetric = false
          ELSE
               j = j+1
          END IF
     END WHILE
     i = i+1
END WHILE

IF  symmetric == true
     WRITE  "Συμμετρικός"
ELSE
     WRITE  "Μη συμμετρικός"
END IF
```

Ο αλγόριθμος μοιάζει πολύ με αυτόν που δώσαμε στο Παράδειγμα 2, αν και είναι πιο πολύπλοκος καθώς χειρίζεται πίνακα δύο διαστάσεων. Και εδώ για την εισαγωγή του πίνακα χρησιμοποιούμε βρόχους FOR, δύο ένθετους βρόχους ώστε να διατρέξουμε τον πίνακα κατά γραμμές, ενώ για την εξέταση της συμμετρίας χρησιμοποιούμε (δύο ένθετους) βρόχους WHILE καθώς η διαδικασία μπορεί να διακοπεί μόλις βρεθεί κάποιο στοιχείο να μην είναι ίσο με το συμμετρικό του. Και πάλι προσέχουμε να αυξάνουμε τους δείκτες των βρόχων WHILE σε κάθε βήμα.

3. Αρθρωτός προγραμματισμός

Οι αλγόριθμοι που παρουσιάσαμε μέχρι εδώ αποτελούσαν ένα ολοκληρωμένο πρόγραμμα ο καθένας· όλα αυτά τα προγράμματα ήταν καλά παραδείγματα του μονολιθικού προγραμματισμού. Όπως όμως αναφέραμε και στην §1.3.2, για την ανάπτυξη πιο πολύπλοκων προγραμμάτων η χρήση του αρθρωτού προγραμματισμού είναι απαραίτητη. Σε αυτή την παράγραφο θα παρουσιάσουμε τις δυνατότητες που παρέχουν οι σύγχρονες γλώσσες προγραμματισμού για την υποστήριξη του αρθρωτού προγραμματισμού.

Στον αρθρωτό προγραμματισμό ένα πρόγραμμα διασπάται σε *υποπρογράμματα*, δηλαδή σε ανεξάρτητες ενότητες που επιτελούν συγκεκριμένες διαδικασίες και που συνδυαζόμενες κατάλληλα δίνουν τη συνολική λύση του προβλήματος. Το τμήμα του προγράμματος που καλεί τα επιμέρους υποπρογράμματα και γενικά οργανώνει τη λύση ονομάζεται πλέον *κύριο πρόγραμμα*.

3.1 Υποπρογράμματα

Ας υποθέσουμε ότι για κάποια εφαρμογή χρειάζεται η εκτύπωση των αριθμών από το 1 ως το 10 σε διάφορα σημεία της και για το λόγο αυτό αποφασίζουμε να γράψουμε ένα υποπρόγραμμα που να κάνει αυτή ακριβώς την εργασία. Ένα τέτοιο πρόγραμμά σε ψευδοκώδικα είναι το επόμενο:

```
PROCEDURE  PRINT10 ( )
     DATA  num
     FOR  num = 1(1)10
          WRITE  num
     END FOR
END PROCEDURE

MAIN
     WRITE  "Οι αριθμοί από το 1 μέχρι το 10"
     PRINT10  ( )
     WRITE  "Πάλι οι ίδιοι αριθμοί"
     PRINT10  ( )
     WRITE  "Οι αριθμοί τυπώθηκαν δύο φορές"
```

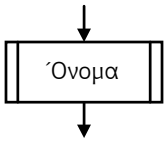
Το υποπρόγραμμα περιέχεται ανάμεσα στα PROCEDURE («*διαδικασία*») και END PROCEDURE. Μετά τη χαρακτηριστική λέξη PROCEDURE ακολουθεί το *όνομα* του υποπρογράμματος, εδώ ***PRINT10***, και δύο κενές παρενθέσεις (στα επόμενα θα δούμε τι μπορεί να περιέχουν αυτές οι παρενθέσεις σε πιο σύνθετα υποπρογράμματα). Το κύριο πρόγραμμα γράφεται πλέον μετά τη χαρακτηριστική λέξη ***MAIN***.

Το όνομα του υποπρογράμματος χρησιμεύει για να *καλείται* το υποπρόγραμμα από άλλα σημεία του προγράμματος, δηλαδή για να δοθεί η εντολή να εκτελεστεί ο κώδικας του υποπρογράμματος. Ο κώδικας των υποπρογραμμάτων γράφεται πριν από τον κώδικα του κύριου προγράμματος ώστε να είναι γνωστός όταν χρειαστεί να εκτελεστεί.

Η εκτέλεση ενός προγράμματος που περιέχει υποπρογράμματα αρχίζει από την πρώτη γραμμή του κύριου προγράμματος. Έτσι στο παραπάνω πρόγραμμα αρχικά εκτυπώνεται «Οι αριθμοί από το

1 μέχρι το 10» με την πρώτη εντολή WRITE. Στην επόμενη γραμμή γίνεται μία *κλήση* του υποπρογράμματος *PRINT10*, οπότε εκτελείται ο κώδικάς του και έτσι εκτυπώνονται οι αριθμοί από το 1 μέχρι το 10. Αφού τελειώσει το υποπρόγραμμα η εκτέλεση συνεχίζεται με την επόμενη γραμμή και έτσι εκτελείται η δεύτερη εντολή WRITE. Μετά υπάρχει άλλη μία κλήση του υποπρογράμματος οπότε εκτυπώνονται άλλη μία φορά οι αριθμοί, και τέλος γίνεται μία ακόμη εκτύπωση με την τελευταία εντολή WRITE.

Σημειώνουμε ότι στα λογικά διαγράμματα κάθε υποπρόγραμμα γράφεται ξεχωριστά, με το δικό του BEGIN και END και κατάλληλη σημείωση του ονόματος και των παραμέτρων του. Η κλήση ενός υποπρογράμματος συμβολίζεται ως εξής:



Παρατηρούμε ότι το υποπρόγραμμα είναι μια μικρογραφία προγράμματος, καθώς περιέχει τις δικές του δηλώσεις μεταβλητών (με τη χαρακτηριστική λέξη DATA) και εκτελεί κάποια συγκεκριμένη εργασία. Οι μεταβλητές αυτές που δηλώνονται μέσα στο υποπρόγραμμα ονομάζονται *τοπικές μεταβλητές* (local variables) του υποπρογράμματος. Οι τοπικές μεταβλητές δημιουργούνται μόλις αρχίσει η εκτέλεση του υποπρογράμματος, έχουν νόημα μόνο μέσα σε αυτό και παύουν να υφίστανται μόλις τελειώσει η εκτέλεσή του. Λέμε ότι οι τοπικές μεταβλητές έχουν *περιοχή ισχύος* (scope) το υποπρόγραμμα.

3.1.1 Παράμετροι υποπρογραμμάτων

Ας φανταστούμε ένα λίγο πιο πολύπλοκο πρόβλημα από το προηγούμενο, για το οποίο απαιτούνται παρόμοιες εκτυπώσεις ακεραίων αριθμών αλλά μεταξύ ορίων που είναι διαφορετικά κάθε φορά. Ο κώδικας θα πρέπει να είναι ανάλογος με τον κώδικα του προηγούμενου παραδείγματος, μόνο που τα όρια της εντολής FOR του υποπρογράμματος χρειάζεται να αλλάζουν σε κάθε κλήση. Επομένως είναι απαραίτητος ένας μηχανισμός με τον οποίο να περνούμε στο υποπρόγραμμα δεδομένα που να ρυθμίζουν τη λειτουργία του. Ο μηχανισμός αυτός είναι οι *παράμετροι* των υποπρογραμμάτων.

Παράδειγμα 1. Εκτύπωση ακεραίων μεταξύ ορίων

Το επόμενο πρόγραμμα διαβάζει από το χρήστη δύο (ακέραιους) αριθμούς min και max και τυπώνει τους ακέραιους από το min μέχρι το max με τη βοήθεια ενός υποπρογράμματος.

```
PROCEDURE PRINTNUMBERS (a, b)
    DATA  num
    FOR  num = a(1)b
        WRITE  num
    END FOR
END PROCEDURE

MAIN
    DATA  min, max
    READ  min, max
    PRINTNUMBERS  (min, max)
```

Το υποπρόγραμμα ***PRINTNUMBERS*** έχει δύο παραμέτρους, a και b, που σημειώνονται μέσα σε παρενθέσεις μετά το όνομά του. Οι παράμετροι μπορούν να χρησιμοποιηθούν μέσα στο υποπρό-

γραμμα όπως οι κοινές μεταβλητές, και έτσι στο παράδειγμα αυτό μπαίνουν ως όρια της εντολής FOR.

Κατά την κλήση του υποπρογράμματος πρέπει να δώσουμε στις παραμέτρους του τις συγκεκριμένες τιμές με τις οποίες θα τρέξει. Αυτό γίνεται συνοδεύοντας το όνομά του με τις τιμές που θέλουμε να αποδώσουμε στις παραμέτρους του μέσα σε παρενθέσεις, όπως φαίνεται στην τελευταία γραμμή του κύριου προγράμματος. Στο παράδειγμά μας, όταν καλείται το υποπρόγραμμα ***PRINT-NUMBERS*** αποδίδονται στις παραμέτρους a και b αντίστοιχα οι τιμές των min και max που έχει δώσει ο χρήστης και έτσι κάθε φορά εκτυπώνονται οι αριθμοί από min ως max.

Σημειώνουμε ότι οι μεταβλητές του κύριου προγράμματος, min και max, δηλώνονται μέσα στο κύριο πρόγραμμα και είναι τοπικές μεταβλητές του. Αντίστοιχα με τις τοπικές μεταβλητές των υποπρογραμμάτων, αυτές ισχύουν μόνο στον κώδικα του κύριου προγράμματος.

3.1.2 Επιστρεφόμενη τιμή

Τα δύο υποπρογράμματα που δώσαμε παραπάνω εκτελούσαν κάποια εργασία και η επικοινωνία τους με το υπόλοιπο πρόγραμμα περιοριζόταν στην εισαγωγή κάποιων δεδομένων μέσω των παραμέτρων. Συχνά όμως ενσωματώνουμε σε υποπρογράμματα κάποιους υπολογισμούς και στην περίπτωση αυτή το αποτέλεσμα πρέπει να μεταφερθεί πίσω σε αυτόν που κάλεσε το υποπρόγραμμα (στο κύριο πρόγραμμα ή σε κάποιο άλλο υποπρόγραμμα).

Σε περιπτώσεις που επιθυμούμε να μεταφέρουμε μία μοναδική τιμή από το υποπρόγραμμα στον καλούντα μπορούμε να χρησιμοποιήσουμε την *επιστρεφόμενη τιμή* (return value) του υποπρογράμματος.

Παράδειγμα 1. Μέγιστο στοιχείο πίνακα

Το επόμενο πρόγραμμα περιέχει ένα υποπρόγραμμα που δέχεται ως παραμέτρους έναν πίνακα και το πλήθος των στοιχείων του και υπολογίζει το μέγιστο στοιχείο του.

Σημειώνουμε τα υποπρογράμματα που επιστρέφουν κάποια τιμή με τη χαρακτηριστική λέξη FUNCTION («*συνάρτηση*») και τα τερματίζουμε αντίστοιχα με το END FUNCTION, ώστε να τα διαφοροποιούμε από εκείνα που δεν έχουν επιστρεφόμενη τιμή. Η νέα εντολή RETURN που εμφανίζεται στην τελευταία γραμμή του υποπρογράμματος καθορίζει την τιμή που θα επιστραφεί από το υποπρόγραμμα· εδώ αυτή είναι το μέγιστο στοιχείο max που υπολογίζεται.

Η χρήση της επιστρεφόμενης τιμής των υποπρογραμμάτων-συναρτήσεων είναι απλή. Συγκεκριμένα είναι σαν η κλήση τους να αντικαθίσταται από την επιστρεφόμενη τιμή. Παρατηρούμε ότι το αποτέλεσμα της συνάρτησης μπορεί να χρησιμοποιηθεί στο δεξί μέρος μιας εντολής απόδοσης τιμής (προτελευταία γραμμή). Γενικότερα μπορεί να χρησιμοποιηθεί ως όρος παραστάσεων (§2.1.3), να εκτυπωθεί απευθείας με χρήση της εντολής WRITE, κ.λπ. Αντίθετα δεν έχει νόημα να διαβαστεί με μια εντολή READ ούτε να βρίσκεται στο αριστερό μέρος μιας εντολής απόδοσης τιμής.

```
FUNCTION ARRAYMAX (array[5], size)
    DATA  max, i
    max = array[0]
    FOR  i = 1 (1) size-1
        IF  array[i] > max
            max = array[i]
        END IF
    END FOR
    RETURN  max
END FUNCTION
```

```
MAIN
    DATA  array[5], i, m
    FOR  i = 0 (1) 4
        READ  array[i]
    END FOR
    m = ARRAYMAX (array, 5)
    WRITE  m
```

Στον παραπάνω ψευδοκώδικα τα ονόματα i και array χρησιμοποιούνται τόσο στο κύριο πρόγραμμα όσο και μέσα στο υποπρόγραμμα. Αυτό στην αρχή μπορεί να προκαλεί σύγχυση, όμως πρέπει πάντα να έχουμε υπόψη ότι τα υποπρογράμματα είναι ανεξάρτητα τμήματα του προγράμματος: όπως ήδη αναφέραμε, οι τοπικές μεταβλητές έχουν περιορισμένη περιοχή ισχύος, και επομένως τα ονόματα που εμφανίζονται σε ένα υποπρόγραμμα είναι ανεξάρτητα από αυτά που εμφανίζονται σε άλλα υποπρογράμματα και στο κύριο πρόγραμμα. Έτσι η μεταβλητή i του υποπρογράμματος είναι εντελώς διαφορετική (αναφέρεται σε άλλη θέση μνήμης) από τη μεταβλητή i του κύριου προγράμματος – απλώς συμβαίνει να έχουν το ίδιο όνομα. Επίσης το όνομα array στο κύριο πρόγραμμα αναφέρεται στον πίνακα που εισάγει ο χρήστης· το όνομα array στο υποπρόγραμμα αναφέρεται σε έναν άλλο πίνακα, στον οποίο όταν καλείται το υποπρόγραμμα δίνεται η τιμή του πίνακα (συμπτωματικά με το ίδιο όνομα) του κύριου προγράμματος. Γενικά η κατάσταση ξεκαθαρίζει αν διαβάζουμε κάθε τμήμα του κώδικα χωρίς να αναφερόμαστε στα υπόλοιπα, και λαμβάνουμε υπόψη κάθε φορά μόνο τις τοπικές μεταβλητές που ορίζονται σε αυτό, και, στην περίπτωση των υποπρογραμμάτων, τις παραμέτρους.

3.1.3 Παράμετροι εξόδου, εισόδου-εξόδου

Η επιστρεφόμενη τιμή των υποπρογραμμάτων μας επιτρέπει να επιστρέφουμε κάποιο αποτέλεσμα από το υποπρόγραμμα προς το σημείο του προγράμματος που το κάλεσε. Υπάρχουν όμως περιπτώσεις στις οποίες θέλουμε να επιστρέψουμε περισσότερα από ένα αποτελέσματα από ένα υποπρόγραμμα, και τότε ο μηχανισμός αυτός δεν επαρκεί.

Σε τέτοιες περιπτώσεις χρησιμοποιείται ο μηχανισμός των *παραμέτρων εξόδου* ενός υποπρογράμματος. Πρόκειται για παραμέτρους όπως αυτές που χρησιμοποιούσαμε μέχρι τώρα, οι οποίες όμως χρησιμεύουν όχι για να εισέρχονται δεδομένα στο υποπρόγραμμα αλλά για να επιστρέφονται δεδομένα στο (κύριο) πρόγραμμα. Γράφονται και αυτές στην παρένθεση μαζί με τις απλές παραμέτρους (που θα καλούνται σε αντιδιαστολή *παράμετροι εισόδου*), με τη χαρακτηριστική λέξη OUT για να διακρίνονται από τις παραμέτρους εισόδου, και μπορούν να είναι σε πλήθος όσες χρειάζεται ο αλγόριθμος.

Παράδειγμα 1. Μέγιστο στοιχείο πίνακα και θέση του

Έστω για παράδειγμα ότι θέλουμε να επεκτείνουμε το προηγούμενο υποπρόγραμμα ώστε να μας επιστρέφει και τη θέση στην οποία βρίσκεται το μέγιστο στοιχείο μέσα στον πίνακα (αντίστοιχο ενιαίο πρόγραμμα για το ελάχιστο στοιχείο είχαμε δει στο Παράδειγμα 1 της §2.4). Ο ψευδοκώδικας που ακολουθεί υλοποιεί τη λύση αυτού του προβλήματος.

Στην περίπτωση αυτή το υποπρόγραμμα επιστρέφει τα αποτελέσματά του μόνο μέσω παραμέτρων εξόδου, δεν έχει επιστρεφόμενη τιμή και έτσι είναι υποπρόγραμμα-διαδικασία (φυσικά είναι δυνατό να κατασκευάσουμε υποπρογράμματα-συναρτήσεις που να έχουν και παραμέτρους εξόδου). Οι υπολογιζόμενες τιμές για το μέγιστο και τη θέση του επιστρέφονται μέσω των παραμέτρων max και pos αντίστοιχα στις μεταβλητές m και p του κύριου προγράμματος, όπως φαίνεται από την εντολή κλήσης του. Σημειώνουμε ότι οι m και p στο κύριο πρόγραμμα δεν λαμβάνουν κάποια τιμή πριν κληθεί η συνάρτηση· αυτό δεν αποτελεί πρόβλημα καθώς πριν από το σημείο αυτό δεν χρησιμοποιούνται

στο κύριο πρόγραμμα, ο σκοπός τους είναι ακριβώς να λάβουν τιμή μόλις τελειώσει το υποπρόγραμμα και με αυτή να συνεχίσουν στο κύριο πρόγραμμα (εδώ εκτυπώνονται).

```
PROCEDURE ARRAYMAXPOS (array[5], size, OUT max, OUT pos)
    DATA  i
    pos = 0
    FOR  i = 1 (1) size
        IF  array[i] > array[pos]
            pos = i
        END IF
    END FOR
    max = array[pos]
END PROCEDURE

MAIN
    DATA  array[5], i, m, p
    FOR  i = 0 (1) 4
        READ  array[i]
    END FOR
    ARRAYMAXPOS (array, 5, m, p)
    WRITE  m, p
```

Όπως πρέπει να έχει διαφανεί, η διαδικασία αντιστοίχησης των τιμών των παραμέτρων διαφέρει όταν αυτές είναι παράμετροι εισόδου και όταν είναι παράμετροι εξόδου· αξίζει λοιπόν να εξετάσουμε τι συμβαίνει στην πράξη στις δύο περιπτώσεις. Μία παράμετρος εισόδου, όπως είχαμε πει, είναι στην ουσία μια νέα μεταβλητή στην οποία *αντιγράφεται* η αποδιδόμενη τιμή· αυτή η νέα «μεταβλητή» χρησιμοποιείται μέσα στο υποπρόγραμμα. Αντίθετα, μία παράμετρος εξόδου στην πραγματικότητα *αναφέρεται* στην ίδια θέση μνήμης με τη μεταβλητή στην οποία τελικά θα μεταφερθεί η τιμή της· για το λόγο αυτό όποιες αλλαγές γίνονται στην παράμετρο μέσα στο υποπρόγραμμα γίνονται στην ίδια τη μεταβλητή, και με τον τρόπο αυτό επιτυγχάνεται η μεταφορά της τιμής που αναφέραμε. Επακόλουθο αυτού του μηχανισμού είναι ότι σε μια παράμετρο εξόδου δεν είναι δυνατό να αποδοθεί κάποια αριθμητική ή γενικότερα σταθερή τιμή (όπως είχαμε παραπάνω για τη μεταβλητή εισόδου size στην οποία δίναμε την τιμή 5). Κάτι τέτοιο δεν έχει άλλωστε νόημα καθώς ο ρόλος μιας τέτοιας παραμέτρου είναι να μεταφέρει μια τιμή σε κάποια μεταβλητή του προγράμματος. Όσον αφορά τις παραμέτρους εισόδου, η διαδικασία της αντιγραφής που περιγράψαμε έχει ως αποτέλεσμα αν τυχόν αλλάξει η τιμή κάποιας παραμέτρου μέσα στο υποπρόγραμμα (κάτι που δεν είδαμε στα παραδείγματα, αλλά επιτρέπεται) αυτή η αλλαγή να μην μεταφέρεται πίσω στην αντίστοιχη μεταβλητή. Εξάλλου κάτι τέτοιο γενικά δεν συνιστάται καθώς οι παράμετροι εισόδου έχουν αποστολή να μεταφέρουν τα δεδομένα που θα χρησιμοποιήσει ο αλγόριθμος· αν θέλουμε να μεταβάλλουμε στο εσωτερικό του υποπρογράμματος κάποια από αυτές τις τιμές είναι προτιμότερο να την καταχωρίσουμε σε μια τοπική μεταβλητή.

Εκτός από τις παραμέτρους εισόδου και τις παραμέτρους εξόδου των υποπρογραμμάτων υπάρχουν και οι *παράμετροι εισόδου-εξόδου*. Όπως πρέπει να είναι φανερό από το όνομά τους, αυτές οι παράμετροι συνδυάζουν τη λειτουργία των δύο άλλων ειδών: περνούν κάποια δεδομένα στο υποπρόγραμμα, αυτά τα δεδομένα υφίστανται κάποια επεξεργασία και μεταβάλλονται, και οι νέες τιμές τους επιστρέφονται πάλι στο πρόγραμμα στην ίδια μεταβλητή. Στον ψευδοκώδικα σημειώνουμε τις παραμέτρους εισόδου-εξόδου με τη χαρακτηριστική λέξη INOUT. Σχετικά με τη διαδικασία αντιστοίχησης μιας παραμέτρου εισόδου-εξόδου με την αντίστοιχη μεταβλητή του προγράμματος ισχύει ό,τι

αναφέραμε παραπάνω για τις μεταβλητές εξόδου. Το επόμενο παράδειγμα είναι χαρακτηριστικό της χρήσης παραμέτρων εισόδου-εξόδου.

Παράδειγμα 2. Ανταλλαγή τιμών μεταβλητών

Το υποπρόγραμμα *SWAP* που ακολουθεί ανταλλάσσει τις τιμές των δύο παραμέτρων του. Η διαδικασία είναι γνωστή από το Παράδειγμα 2 της §2.2.1, αλλά τοποθετώντας τη μέσα σε ένα υποπρόγραμμα γίνεται πολύ πιο εύχρηστη:

```
PROCEDURE SWAP (INOUT x, INOUT y)
    DATA  tmp
    tmp = x
    x = y
    y = tmp
END PROCEDURE

MAIN
    DATA  a, b
    READ  a, b
    SWAP (a, b)
    WRITE  a, b  (Τυπώνονται τα a, b σε αντίστροφη σειρά)
```

3.1.4 Γενικές μεταβλητές

Στα παραδείγματα που παρουσιάσαμε φροντίσαμε τόσο το κύριο πρόγραμμα όσο και τα υποπρογράμματα να είναι απολύτως ανεξάρτητα και το κάθε τμήμα κώδικα να χρησιμοποιεί μόνο τοπικές μεταβλητές.

Σε αντιδιαστολή με τις τοπικές μεταβλητές, ένα πρόγραμμα μπορεί να έχει και *γενικές μεταβλητές* (global variables), οι οποίες ισχύουν τόσο στο κύριο πρόγραμμα όσο και μέσα στα υποπρογράμματα και για το σκοπό αυτό δηλώνονται στην αρχή του κώδικα, έξω από τα υποπρογράμματα και από το κύριο πρόγραμμα. Γενικές μεταβλητές μπορούν να χρησιμοποιηθούν για ποσότητες που αφορούν όλο το πρόγραμμα, αλλάζουν τιμές σε διάφορα σημεία του και για το λόγο αυτό πρέπει να είναι εξίσου διαθέσιμες σε όλο τον κώδικα.

Θα μπορούσε κανείς να θεωρήσει ότι αυτές οι μεταβλητές μπορούν να χρησιμεύσουν για την ανταλλαγή δεδομένων μεταξύ του κύριου προγράμματος και των υποπρογραμμάτων, αντικαθιστώντας τις παραμέτρους των υποπρογραμμάτων. Έτσι αν στο παράδειγμα που δώσαμε παραπάνω οι μεταβλητές a, b ήταν γενικές, το πρόγραμμα θα είχε την εξής μορφή:

```
DATA  a, b

PROCEDURE PRINTNUMBERS ( )
    DATA  num
    FOR  num = a(1)b
        WRITE  num
    END FOR
END PROCEDURE

MAIN
    READ  a, b
    PRINTNUMBERS  (a, b)
```

Όσο κι αν κάτι τέτοιο φαίνεται εξυπηρετικό, δεν πρέπει να προτιμάται γιατί έτσι τα υποπρογράμματα παύουν να είναι αυτόνομα τμήματα του προγράμματος και συνδέονται με το υπόλοιπο πρόγραμμα μέσω των γενικών μεταβλητών. Σε μια τέτοια περίπτωση, αν προσπαθήσουμε να χρησιμοποιήσουμε το υποπρόγραμμα σε ένα διαφορετικό πρόγραμμα θα πρέπει να «μεταφέρουμε» μαζί και τις γενικές μεταβλητές που χρησιμοποιούνται σε αυτό (τις a, b παραπάνω), και φυσικά να εξασφαλίσουμε ότι αυτά τα ονόματα μεταβλητών δεν χρησιμοποιούνται για κάποιο σκοπό στο νέο πρόγραμμα. Ομοίως για να διαβάσουμε το παραπάνω πρόγραμμα πρέπει πάντα να θυμόμαστε και τις γενικές μεταβλητές, κάτι όχι εύκολο καθώς είναι δηλωμένες μακριά από το σημείο που χρησιμοποιούνται. Στην πράξη η χρήση των γενικών μεταβλητών πρέπει να είναι εξαιρετικά περιορισμένη καθώς δυσκολεύει τη δομή και την κατανόηση ενός προγράμματος· ευτυχώς σχεδόν πάντα μπορεί να αποφευχθεί με τη χρήση κατάλληλων παραμέτρων στα υποπρογράμματα.

Πιο συνηθισμένη είναι η χρήση *γενικών σταθερών*, δηλαδή σταθερών που δηλώνονται (και συγχρόνως παίρνουν τιμή) στην αρχή του κώδικα· τέτοιες σταθερές μπορεί να αφορούν κάποια χαρακτηριστικά μεγέθη του προβλήματος, κάποιες αριθμητικές σταθερές όπως ο αριθμός *π* που πραγματικά δεν θα είχε νόημα να δηλώνονται χωριστά σε κάθε τμήμα του προγράμματος.

Παράδειγμα 1. Πρόσθεση πινάκων

Ζητούμε ένα πρόγραμμα για την πρόσθεση δύο μονοδιάστατων πινάκων n στοιχείων (Τα στοιχεία του πίνακα του αθροίσματος προκύπτουν προσθέτοντας τα αντίστοιχα στοιχεία των δύο πινάκων). Ο ψευδοκώδικας που ακολουθεί περιλαμβάνει ένα υποπρόγραμμα για την εισαγωγή των στοιχείων των πινάκων από το χρήστη, ένα άλλο για την πρόσθεση δύο πινάκων και ένα τελευταίο για την εκτύπωση ενός μονοδιάστατου πίνακα. Στον ψευδοκώδικα χρησιμοποιείται μία γενική σταθερά maxsize που είναι το μέγιστο μέγεθος πινάκων που μπορεί να χειριστεί το πρόγραμμα (όλοι οι πίνακες δηλώνονται να είναι μεγέθους maxsize στοιχείων). Το πραγματικό μέγεθος των πινάκων n ζητείται από το χρήστη και δίνεται στα υποπρογράμματα μέσω της παραμέτρου size.

```
CONST  maxsize = 10

PROCEDURE READARRAY (OUT array[maxsize], size)
    DATA  i
    FOR  i = 0(1)size-1
        READ  array[i]
    END FOR
END PROCEDURE

PROCEDURE ADDARRAYS (array1[maxsize], array2[maxsize], OUT sum[maxsize], size)
    DATA  i
    FOR  i = 0(1)size-1
        sum[i]  = array1[i]+array2[i]
    END FOR
END PROCEDURE

PROCEDURE WRITEARRAY (array[maxsize], size)
    DATA  i
    FOR  i = 0(1)size-1
        WRITE  array[i]
    END FOR
END PROCEDURE
```

```
MAIN
    DATA  a[maxsize], b[maxsize], c[maxsize]

    READ  n
    READARRAY (a, n)
    READARRAY (b, n)

    ADDARRAYS (a, b, c, n)

    WRITEARRAY (a, n)
    WRITE  "+"
    WRITEARRAY (b, n)
    WRITE  "="
    WRITEARRAY (c, n)
```

Σε αυτό το παράδειγμα φαίνεται καθαρά η διευκόλυνση που παρέχει η χρήση υποπρογραμμάτων. Τα υποπρογράμματα για την εισαγωγή και την εκτύπωση πίνακα καλούνται πολλαπλές φορές από το κύριο πρόγραμμα ενώ έχουν γραφτεί μόνο μία. Συγχρόνως η λειτουργία του προγράμματος φαίνεται εύκολα, απλά διαβάζοντας το κύριο πρόγραμμα.

3.2 Αναδρομή

Το παραγοντικό ενός ακέραιου αριθμού n (συμβολίζεται n!) είναι το γινόμενο

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Μια ισοδύναμη μορφή αυτού του ορισμού είναι η

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

καθώς για παράδειγμα το παραγοντικό του 4 είναι, σύμφωνα με την τελευταία σχέση,

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

Παρατηρούμε ότι στη δεύτερη αυτή μορφή το παραγοντικό του n ορίζεται με τη βοήθεια του παραγοντικού του $n-1$. Μία σχέση όπως αυτή που περιέχει, που περιέχει ως μέρος της τον εαυτό της, ονομάζεται *αναδρομική*.

Γενικότερα ένας αλγόριθμος καλείται *αναδρομικός* (ή *αναγωγικός*) αν ορίζεται με τη βοήθεια του εαυτού του. Με άλλα λόγια, ένας αναδρομικός αλγόριθμος ανάγει την επίλυση του προβλήματος στην επίλυση του ίδιου προβλήματος με «πιο απλά» δεδομένα. Πολλά προβλήματα, κυρίως μαθηματικής φύσης, βρίσκουν εύκολη και κομψή λύση με χρήση αναδρομής, και η υποστήριξη αναδρομικών τεχνικών από τις σύγχρονες γλώσσες προγραμματισμού δίνει τη δυνατότητα για αντίστοιχα απλές υλοποιήσεις.

Στην εφαρμογή του αναδρομικού αλγορίθμου του παραγοντικού για τον υπολογισμό του 4! παρατηρούμε ότι τα αναδρομικά βήματα σταματούν όταν φθάνουμε στο 1! που γνωρίζουμε ότι ισούται με 1 από τον πρώτο ορισμό. Αυτό είναι κοινό στοιχείο όλων των αναδρομικών αλγορίθμων: μαζί με την αναδρομική σχέση πρέπει να ορίζεται και μία *συνθήκη διακοπής* που καθορίζει πώς θα τερματιστούν οι αναδρομικές κλήσεις. Η συνθήκη διακοπής αφορά πάντα την εκτέλεση του αλγορίθμου με πολύ απλές τιμές των παραμέτρων, για τις οποίες το αποτέλεσμά του είναι γνωστό. Έτσι ο πλήρης αναδρομικός ορισμός του παραγοντικού είναι

$$n! = n \cdot (n-1)! \ , \ \text{με } 1! = 1.$$

Παράδειγμα 1. Υπολογισμός παραγοντικού

Θα δώσουμε τώρα τους αλγορίθμους για τον υπολογισμό του παραγοντικού σύμφωνα και με τους δύο ορισμούς, αριστερά τον επαναληπτικό και δεξιά τον αναδρομικό:

```
FUNCTION FACTORIAL (n)
    DATA  fact, i
    fact = 1
    FOR  i=2(1)n
        fact = fact * i
    END FOR
    RETURN  fact
END FUNCTION
```

```
FUNCTION FACTORIAL (n)
    DATA  fact
    IF  n==1
        fact = 1
    ELSE
        fact = n * FACTORIAL (n-1)
    END IF
    RETURN  fact
END FUNCTION
```

Σχετικά με τον επαναληπτικό αλγόριθμο σημειώνουμε ότι ο υπολογισμός του γινομένου μοιάζει αρκετά με τον υπολογισμό των αθροισμάτων που έχουμε δει σε προηγούμενα παραδείγματα. Το μόνο σημείο που πρέπει να προσεχθεί είναι η αρχική τιμή του γινομένου, που είναι 1 καθώς πρόκειται για γινόμενο αριθμών – αν δίναμε αρχική τιμή 0 όπως στα αθροίσματα το αποτέλεσμα δεν θα ήταν πολύ αξιόλογο!

Σχετικά με τον αναδρομικό αλγόριθμο παρατηρούμε ότι η (αναδρομική) κλήση της συνάρτησης *FACTORIAL* στην τελευταία γραμμή υλοποιεί ακριβώς τον αναδρομικό ορισμό του παραγοντικού, ενώ ο έλεγχος στις γραμμές 2-3 υλοποιεί τη συνθήκη διακοπής.

Παράδειγμα 2. Αριθμοί Fibonacci

Οι αριθμοί Fibonacci είναι μια πολύ γνωστή ακολουθία στα Μαθηματικά. Είναι οι αριθμοί

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

όπου ο καθένας είναι το άθροισμα των δύο προηγούμενών του στην ακολουθία ξεκινώντας με τους αριθμούς 1, 1. Αν συμβολίσουμε F(n) τον n-οστό αριθμό Fibonacci μπορούμε να γράψουμε την ακολουθία ως

$$F(n)=F(n-1)+F(n-2), \text{ με } F(1)=1,\ F(2)=1$$

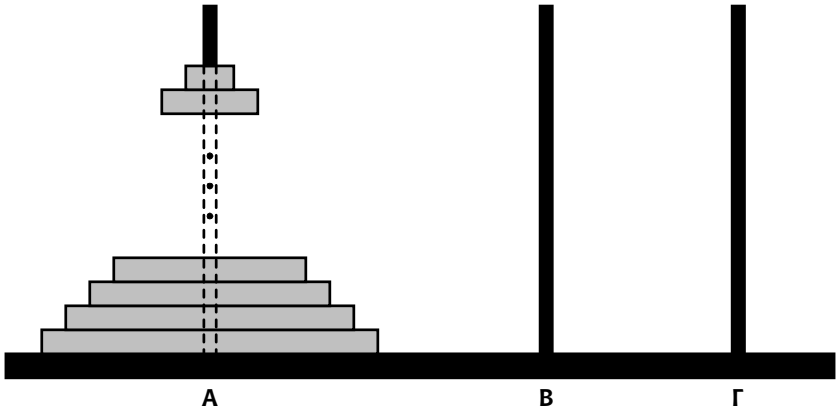
Ο ορισμός αυτός είναι καθαρά αναδρομικός και επομένως μπορεί να υλοποιηθεί άμεσα με έναν αναδρομικό αλγόριθμο:

```
FUNCTION FIBONACCI (n)
    DATA  fib
    IF  (n==1  ||  n==2)
        fib = 1
    ELSE
        fib = FIBONACCI (n-1) + FIBONACCI (n-2)
    END IF
    RETURN  fib
END FUNCTION
```

Παρατηρούμε ότι στην περίπτωση αυτή χρειάζονται δύο αναδρομικές κλήσεις του αλγορίθμου σε κάθε βήμα.

Παράδειγμα 3. Πύργοι του Ανόι

Σύμφωνα με την παράδοση μια ομάδα βουδιστών μοναχών από το Ανόι ασχολήθηκε με το εξής πρόβλημα: Οι μοναχοί είχαν τρεις κατακόρυφες ράβδους Α, Β, Γ και 64 κυκλικούς δίσκους διαφορετικών διαμέτρων περασμένους στη ράβδο Α, ο ένας πάνω στον άλλο με φθίνουσες διαμέτρους από κάτω προς τα πάνω, έτσι ώστε να σχηματίζουν ένα πύργο.



Οι μοναχοί ήθελαν να μετακινήσουν τον πύργο των δίσκων από τη ράβδο Α στη Β διατηρώντας την αρχική διάταξη των δίσκων και χρησιμοποιώντας τη ράβδο Γ ως βοηθητική. Στην προσπάθειά τους αυτή έπρεπε να τηρούν τους εξής κανόνες: Σε κάθε βήμα μπορεί να μετακινηθεί ένας μόνο δίσκος (αυτός που βρίσκεται στην κορυφή) από μια ράβδο σε άλλη, και επίσης ένας δίσκος δεν επιτρέπεται να βρεθεί ποτέ πάνω από μικρότερό του.

Ένας επαναληπτικός αλγόριθμος για την αντιμετώπιση αυτού του προβλήματος είναι αρκετά περίπλοκος. Αντίθετα, ένας αναδρομικός αλγόριθμος μπορεί να προκύψει εύκολα από την παρατήρηση ότι το πρόβλημα των 64 δίσκων μπορεί να αναχθεί στο αντίστοιχο των 63 δίσκων κ.ο.κ. Πράγματι, αν υποθέσουμε ότι γνωρίζουμε πώς να μεταφέρουμε 63 δίσκους από μία ράβδο σε μία άλλη, τότε μπορούμε να μεταφέρουμε τους 63 από τους δίσκους της ράβδου Α στη Γ, στη συνέχεια να μεταφέρουμε τον τελευταίο δίσκο της Α στη Β και τέλος να μεταφέρουμε τους 63 δίσκους από τη Γ στη Β.

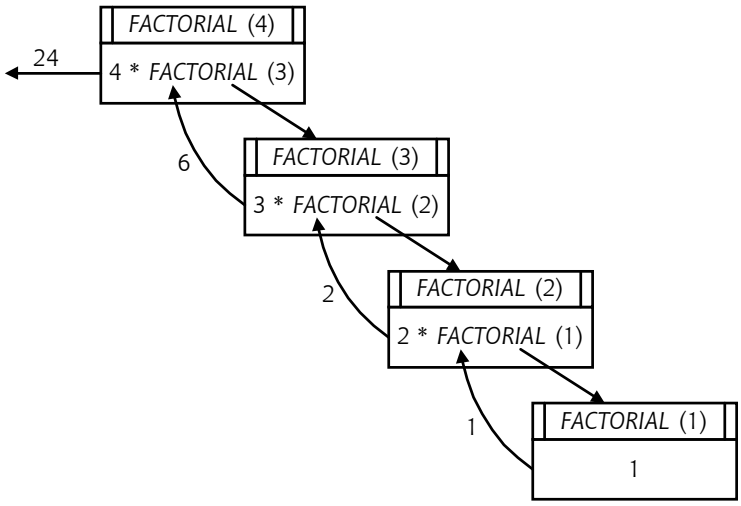
Ο επόμενος ψευδοκώδικας υλοποιεί αυτό τον αλγόριθμο. Στις παραμέτρους του h είναι το ύψος του πύργου που μεταφέρουμε, a η αρχική ράβδος, b η τελική και c η βοηθητική. Επίσης χρησιμοποιείται μια βοηθητική διαδικασία για τη μεταφορά κάθε δίσκου, που εδώ απλώς τυπώνει τη ράβδο προέλευσης και τη ράβδο προορισμού.

```
PROCEDURE MOVEDISK (a, b)
    WRITE  "Μεταφορά από ", a, " προς ", b
END PROCEDURE

PROCEDURE MOVETOWER (h, a, b, c)
    IF  h==1
        MOVEDISK (a, b)
    ELSE
        MOVETOWER (h-1, a, c, b)
        MOVEDISK (a, b)
        MOVETOWER (h-1, c, b, a)
    END IF
END PROCEDURE

MAIN
    DATA  h
    h = 5  (Ύψος του πύργου)
    MOVETOWER  (h, 1, 2, 3)
```

Καλώντας τη διαδικασία *MOVETOWER* με πρώτη παράμετρο το συνολικό ύψος του αρχικού πύργου θα λάβουμε μια λίστα με όλες τις απαιτούμενες κινήσεις ώστε να μεταφερθεί ο πύργος από την αρχική στην τελική ράβδο.

Αφού είδαμε μερικά παραδείγματα αναδρομικών αλγορίθμων, είναι σκόπιμο να παρουσιάσουμε τον τρόπο με τον οποίο υλοποιείται η διαδικασία της αναδρομής σε μια γλώσσα προγραμματισμού.

Σε κάθε κλήση της αναδρομικής σχέσης γίνεται και μια νέα κλήση του αναδρομικού υποπρογράμματος. Τότε η τρέχουσα κλήση διακόπτεται και όλα τα στοιχεία που την αφορούν (τιμές τοπικών μεταβλητών και παραμέτρων) αποθηκεύονται στη μνήμη του υπολογιστή· για τη νέα κλήση δημιουργούνται νέες μεταβλητές και παράμετροι που παίρνουν κατάλληλες τιμές. Το ίδιο γίνεται για όλα τα αναδρομικά βήματα και έτσι φθάνοντας στη συνθήκη διακοπής θα έχουν δημιουργηθεί πολλά αντίγραφα των αντικειμένων του υποπρογράμματος. Με τη συνθήκη διακοπής μπορεί πλέον να γίνει κάποιος υπολογισμός, οπότε το τελευταίο αντίγραφο του υποπρογράμματος ολοκληρώνεται. Η διαδικασία συνεχίζεται προς τα πίσω, με την ανάκληση από τη μνήμη σε αντίστροφη σειρά των αντιγράφων του υποπρογράμματος που είχαν αποθηκευτεί και με την ολοκλήρωσή τους. Έτσι φτάνουμε στο αρχικό αντίγραφο του υποπρογράμματος, του οποίου η ολοκλήρωση παρέχει το τελικό αποτέλεσμα. Το επόμενο σχήμα παρουσιάζει τον υπολογισμό του 4!.



Όπως πρέπει να έγινε φανερό από αυτή την περιγραφή, η αναδρομή στην πράξη έχει αρκετά μεγάλο κόστος κατά την εκτέλεση του προγράμματος, τόσο σε χώρο μνήμης (που καταλαμβάνεται από τα διαδοχικά αντίγραφα των μεταβλητών) όσο και σε χρόνο εκτέλεσης (που χρειάζεται για να γίνουν οι αντιγραφές των μεταβλητών). Το κόστος μάλιστα γίνεται ακόμη μεγαλύτερο όταν ο αναδρομικός αλγόριθμος περιλαμβάνει πολλαπλές κλήσεις του εαυτού του σε κάθε βήμα (π.χ. οι αριθμοί Fibonacci). Επιπλέον σε τέτοιες περιπτώσεις οι υπολογισμοί πολλών βημάτων επαναλαμβάνονται καθώς συναντώνται στις διάφορες κλήσεις, με ακόμα μεγαλύτερη και άσκοπη επιβάρυνση του προγράμματος.

Για τους λόγους αυτούς πρέπει, όπου είναι δυνατό, να προτιμώνται ισοδύναμοι μη αναδρομικοί αλγόριθμοι. Αυτοί έχουν τη μορφή κάποιας επαναληπτικής διαδικασίας που με κατάλληλους υπολογισμούς καταλήγει στο ίδιο αποτέλεσμα· τέτοια ήταν η περίπτωση του επαναληπτικού ορισμού του παραγοντικού που δώσαμε παραπάνω. Γενικά βέβαια η κατασκευή ισοδύναμου επαναληπτικού αλγορίθμου δεν είναι απλή και εξαρτάται από τη συγκεκριμένη μορφή του αναδρομικού αλγορίθμου· συγκεκριμένες μέθοδοι υπάρχουν μόνο για κάποιες απλές κατηγορίες αναδρομικών αλγορίθμων.

4. Αλγόριθμοι Ταξινόμησης και Αναζήτησης

4.1 Αλγόριθμοι Ταξινόμησης

Με τον όρο *ταξινόμηση* εννοούμε γενικά τη διαδικασία της διάταξης ενός δεδομένου συνόλου ομοειδών αντικειμένων με βάση ένα συγκεκριμένο *κριτήριο διάταξης*. Η ταξινόμηση είναι απαραίτητη σε πολλές εφαρμογές (τηλεφωνικοί κατάλογοι, λεξικά, κ.λπ.), ιδιαίτερα μάλιστα σε περιπτώσεις όπου υπάρχει μεγάλος αριθμός καταχωρισμένων αντικειμένων στα οποία πραγματοποιούμε αναζητήσεις με σκοπό την ανάκτηση πληροφοριών (βλ. και την §4.2).

Οι αλγόριθμοι ταξινόμησης διακρίνονται σε δύο μεγάλες κατηγορίες, στην *εσωτερική* και στην *εξωτερική* ταξινόμηση. Στην πρώτη περίπτωση τα δεδομένα που ταξινομούνται έχουν δομή πίνακα και χωρούν όλα στην κύρια μνήμη του υπολογιστή· η ταξινόμηση πραγματοποιείται χωρίς να χρησιμοποιείται κάποιος βοηθητικός χώρος αποθήκευσης των δεδομένων, με κατάλληλες αντιμεταθέσεις των στοιχείων του πίνακα μέχρι αυτά να ταξινομηθούν.

Στη δεύτερη περίπτωση τα δεδομένα είναι αποθηκευμένα σε εξωτερικές μνήμες (δίσκους, ταινίες) σε μορφή σειραϊκού αρχείου· σε αυτή την περίπτωση δεν υπάρχει άμεση πρόσβαση στα δεδομένα και συνεπώς δεν είναι δυνατό να εφαρμοστεί κάποια μέθοδος εσωτερικής ταξινόμησης. Για την ταξινόμηση τέτοιων δεδομένων συνήθως χρησιμοποιείται κάποιο *ευρετήριο*, δηλαδή ένας πίνακας στον οποίο για κάθε εγγραφή των δεδομένων καταχωρίζονται το *κλειδί* με βάση το οποίο γίνεται η ταξινόμηση (π.χ. Ονοματεπώνυμο, Αριθμός Μητρώου) και η θέση της εγγραφής στο αρχείο· το ευρετήριο έχει μικρό μέγεθος και μπορεί να αποθηκευτεί στην κύρια μνήμη του υπολογιστή, και επομένως μπορεί να ταξινομηθεί με κάποια μέθοδο εσωτερικής ταξινόμησης. Εναλλακτικά, ή και σε συνδυασμό με την προηγούμενη τακτική (όταν για παράδειγμα και το ευρετήριο είναι αποθηκευμένο σε εξωτερική μνήμη), ανακτώνται τμήματα των δεδομένων στην κύρια μνήμη, ταξινομούνται, αποθηκεύονται πάλι στην εξωτερική μνήμη, και τελικά οι επιμέρους ταξινομημένες ακολουθίες συγχωνεύονται.

Στα επόμενα θα παρουσιάσουμε μεθόδους εσωτερικής ταξινόμησης. Σε κάθε περίπτωση, δίνεται ένας πίνακας με n ακέραιους αριθμούς $k_0, k_1, k_2, \ldots, k_{n-1}$ (αριθμούμε τους δείκτες από το 0 και όχι από το 1 για ευκολότερη προσαρμογή του ψευδοκώδικα σε C++) και ζητούμε να ταξινομηθούν σε αύξουσα σειρά. (Η §4.1.4 αντιμετωπίζει ένα διαφορετικό πρόβλημα.) Αυτή είναι η απλούστερη περίπτωση ταξινόμησης, καθώς ταξινομούνται τα ίδια τα στοιχεία που δίνονται. Οι αλγόριθμοι που θα αναπτύξουμε προσαρμόζονται εύκολα και σε πιο σύνθετες περιπτώσεις όπου δίνονται κάποιες εγγραφές (για παράδειγμα με προσωπικά στοιχεία ατόμων) οι οποίες πρέπει να ταξινομηθούν με βάση κάποιο *κλειδί* (για παράδειγμα το ονοματεπώνυμο).

4.1.1 Μέθοδος της Φυσαλλίδας

Η μέθοδος της *φυσαλλίδας* (bubble sort) ή αλλιώς μέθοδος της *ανταλλαγής* (exchange sort) έχει την ιδιότητα να ανεβάζει σταδιακά τα μικρότερα στοιχεία στην κορυφή του πίνακα, ακριβώς όπως μια φυσαλλίδα ανεβαίνει στην επιφάνεια του νερού. Αρχικά διατρέχεται ο πίνακας από το τέλος προς την αρχή και συγκρίνονται τα διαδοχικά στοιχεία k_i και k_{i-1}. Αν $k_i < k_{i-1}$ τα στοιχεία αντιμετατίθενται, και τελικά με τον τρόπο αυτό το ελάχιστο στοιχείο του πίνακα έρχεται στην πρώτη θέση. Η ίδια διαδικασία επαναλαμβάνεται για τα $n-1$ τελευταία στοιχεία του πίνακα, οπότε το αμέσως μεγαλύτερο

στοιχείο έρχεται στη δεύτερη θέση. Συνεχίζοντας έτσι μετά από συνολικά $n-1$ βήματα θα ταξινομηθεί ολόκληρος ο πίνακας. Σημειώνουμε ότι αν σε κάποιο βήμα δεν γίνει καμία αντιμετάθεση στοιχείων, αυτό σημαίνει ότι ο πίνακας είναι ήδη ταξινομημένος και επομένως η διαδικασία μπορεί να διακοπεί.

Ένα παράδειγμα εφαρμογής της μεθόδου της φυσαλλίδας παρουσιάζεται στο επόμενο διάγραμμα. Το πρώτο βήμα παρουσιάζεται αναλυτικά, και τα στοιχεία που αντιμετατίθενται έχουν υπογραμμιστεί. Τα υπόλοιπα βήματα παρουσιάζονται συνοπτικά, και φαίνεται ο πίνακας μετά το τέλος τους. Με έντονα στοιχεία σημειώνουμε το ήδη ταξινομημένο τμήμα του πίνακα.

Αρχικός πίνακας	Βήμα 1			Βήμα 2	Βήμα 3	Βήμα 4
38	38	38	**8**	**8**	**8**	**8**
19	19	8	38	**19**	**19**	**19**
25	8	19	19	38	**21**	**21**
8	25	25	25	21	38	**25**
21	21	21	21	25	25	38

Για την υλοποίηση της μεθόδου της φυσαλλίδας είναι απαραίτητος ένας δείκτης t που κρατάει την τρέχουσα θέση της κορυφής του μη ταξινομημένου τμήματος του πίνακα καθώς και ένας δείκτης i που διατρέχει τα (μη ταξινομημένα) στοιχεία του πίνακα. Επίσης χρήσιμη είναι μία λογική μεταβλητή sorted που γίνεται true αν σε κάποιο βήμα δεν έγινε καμία αντιμετάθεση, ώστε να διακοπεί η διαδικασία. Ο ακόλουθος ψευδοκώδικας υλοποιεί τη μέθοδο της φυσαλλίδας: (Σε αυτό όπως και στους επόμενους ψευδοκώδικες υποθέτουμε την ύπαρξη μιας διαδικασίας *SWAP* (x, y) που ανταλλάσσει τα περιεχόμενα των μεταβλητών x και y, βλ. το Παράδειγμα 2 της §**Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**)

```
PROCEDURE BUBBLESORT (INOUT k[maxsize], n)
    DATA  sorted, t, i
    sorted = false;
    t = 0;
    DO
        sorted = true;
        FOR  i = n-1 (-1) t+1
            IF  k[i] < k[i-1]
                SWAP (k[i], k[i-1])
                sorted = false
            END IF
        END FOR
        t = t+1
    WHILE  ( sorted == false  &&  t <= n-2 )
END PROCEDURE
```

Πρέπει να παρατηρήσουμε ότι η μέθοδος της φυσαλλίδας δεν είναι ιδιαίτερα αποτελεσματική, καθώς απαιτεί πολλές εναλλαγές μεταξύ στοιχείων του πίνακα (n^2 στη χειρότερη περίπτωση). Οι επόμενες μέθοδοι ταξινόμησης που θα παρουσιάσουμε είναι γενικά πιο αποτελεσματικές.

4.1.2 Μέθοδος της Επιλογής

Η μέθοδος της *επιλογής* (selection sort) στηρίζεται στην εξής διαδικασία: Στο πρώτο βήμα βρίσκεται το ελάχιστο στοιχείο του πίνακα και τοποθετείται στην πρώτη θέση (ανταλλάσσεται με το πρώτο στοιχείο). Σε κάθε επόμενο βήμα βρίσκεται το ελάχιστο στοιχείο μεταξύ των υπολοίπων και

τοποθετείται στην επόμενη θέση στον πίνακα. Είναι προφανές ότι μετά από $n-1$ βήματα ολόκληρος ο πίνακας θα έχει ταξινομηθεί.

Το επόμενο διάγραμμα παρουσιάζει ένα παράδειγμα εφαρμογής της μεθόδου της επιλογής:

Αρχικός πίνακας	Βήμα 1	Βήμα 2	Βήμα 3	Βήμα 4
38	**8**	**8**	**8**	**8**
19	19	**19**	**19**	**19**
25	25	25	**21**	**21**
8	38	38	38	**25**
21	21	21	25	38

Για την υλοποίηση της μεθόδου χρειάζεται ένας δείκτης t που κρατάει τη θέση στην οποία τοποθετείται σε κάθε βήμα το ελάχιστο στοιχείο του μη ταξινομημένου τμήματος του πίνακα, ένας δεύτερος δείκτης i που διατρέχει τα στοιχεία του πίνακα, καθώς και ένας δείκτης m στον οποίο αποθηκεύεται η θέση του ελάχιστου στοιχείου σε κάθε βήμα.

```
PROCEDURE SELECTIONSORT (INOUT k[maxsize], n)
    DATA  t, m, i
    FOR  t = 0 (1) n-2
        m = t
        FOR  i = t+1 (1) n-1
            IF  k[i] < k[m]
                m = i
            END IF
        END FOR
        SWAP (k[t], k[m])
    END FOR
END PROCEDURE
```

Είναι εύκολο να συμπεράνουμε ότι η μέθοδος της επιλογής είναι αρκετά πιο αποτελεσματική από τη μέθοδο της φυσαλλίδας, καθώς απαιτεί μόνο $n-1$ αντιμεταθέσεις στοιχείων στη χειρότερη περίπτωση.

4.1.3 Γρήγορη Ταξινόμηση

Η μέθοδος της *γρήγορης ταξινόμησης* (quicksort) είναι μία χαρακτηριστική αναδρομική μέθοδος ταξινόμησης. Συγκεκριμένα, ο συνολικός πίνακας διαμερίζεται (πιθανώς με κατάλληλη αναδιάταξη των στοιχείων του) σε δύο τμήματα, έτσι ώστε κάθε στοιχείο του ενός (αριστερού) τμήματος να είναι μικρότερο ή ίσο κάθε στοιχείου του άλλου (δεξιού) τμήματος. Τα δύο αυτά τμήματα μπορούν να ταξινομηθούν εφαρμόζοντας (αναδρομικά) την ίδια διαδικασία. Ο αλγόριθμος σταματά όταν κάποιο τμήμα που πρόκειται να ταξινομηθεί έχει 0 ή 1 στοιχείο. Τελικά με τον τρόπο αυτό λαμβάνεται ο αρχικός πίνακας ταξινομημένος.

Κεντρικό σημείο του αλγορίθμου της γρήγορης ταξινόμησης είναι η διαδικασία της διαμέρισης. Αυτή επιτυγχάνεται ως εξής: Επιλέγεται ένα *οδηγό* στοιχείο του πίνακα και αναζητείται η τελική του θέση στον ταξινομημένο πίνακα. Κατά τη διάρκεια της εύρεσης της θέσης του οδηγού στοιχείου τα υπόλοιπα στοιχεία μετακινούνται έτσι ώστε όλα τα στοιχεία προς τα αριστερά του οδηγού να είναι μικρότερα ή ίσα αυτού και όλα τα στοιχεία προς τα δεξιά του να είναι μεγαλύτερα.

Στην πράξη, ορίζεται ως οδηγό στοιχείο το πρώτο στοιχείο του πίνακα (υπάρχουν και άλλοι τρόποι επιλογής του, αλλά μπορεί να οδηγήσουν σε ειδικές περιπτώσεις). Επίσης ορίζονται δύο δείκτες i και j: ο i κινείται από αριστερά προς τα δεξιά ξεκινώντας από την αρχή του πίνακα, και ο j κινείται αντίστροφα, από δεξιά προς τα αριστερά ξεκινώντας από το τέλος του πίνακα. Σε κάθε χρονική στιγμή πρέπει τα στοιχεία που βρίσκονται αριστερά του i να είναι μικρότερα ή ίσα του οδηγού στοιχείου ενώ τα στοιχεία στα δεξιά του j να είναι μεγαλύτερα του οδηγού στοιχείου. Για να επιτευχθεί αυτό ο δείκτης j προσπερνά κάθε στοιχείο που είναι μεγαλύτερο από το οδηγό και σταματά μόλις φτάσει σε ένα στοιχείο μικρότερο από το οδηγό στοιχείο. Αν ο j φτάσει να δείχνει το ίδιο στοιχείο με τον i, τότε η διαδικασία σταματά και ανταλλάσσεται το οδηγό στοιχείου με εκείνο που δείχνουν οι i και j (και που είναι η τελική θέση του οδηγού). Αν ο j σταματήσει πριν τον i, τότε αρχίζει να κινείται ο i. Αυτός αντίστοιχα προσπερνά κάθε στοιχείο που είναι μικρότερο ή ίσο από το οδηγό και σταματά μόλις φτάσει σε ένα στοιχείο μεγαλύτερο από το οδηγό στοιχείο ή φτάσει στο j. Όταν σταματήσει ο i, το στοιχείο που δείχνει ανταλλάσσεται με το στοιχείο που δείχνει ο j (αν βέβαια δεν δείχνουν στο ίδιο στοιχείο) και ο j αρχίζει πάλι την κίνησή του.

Στο διάγραμμα που ακολουθεί παρουσιάζεται η διαδικασία της διάσπασης ενός απλού πίνακα:

Αρχικός πίνακας	**4** i	3	1	9	2	6	5 j
Κίνηση j	4 i	3	1	9	2 j	6	5
Κίνηση i	4	3	1	9 i	2 j	6	5
Ανταλλαγή	4	3	1	2 i	9 j	6	5
Κίνηση j	4	3	1	2 i, j	9	6	5
Τελική θέση οδηγού στοιχείου	2	3	1	**4**	9	6	5

Στον επόμενο ψευδοκώδικα παρουσιάζεται η υλοποίηση του αλγορίθμου της γρήγορης ταξινόμησης. Παρατηρούμε μόνο ότι στην περίπτωση αυτή περνούν ως παράμετροι στον αλγόριθμο οι δείκτες του πρώτου και του τελευταίου στοιχείου του πίνακα που θα ταξινομηθεί· αυτό είναι απαραίτητο ώστε να είναι δυνατή η (αναδρομική) εφαρμογή του αλγορίθμου στα επιμέρους τμήματα του πίνακα όπως περιγράφηκε παραπάνω.

```
PROCEDURE QUICKSORT (INOUT k[maxsize], start, end)
    DATA  i, j
    IF  start < end
        i = start
        j = end
        WHILE  i < j
            WHILE  ( k[j] > k[start] )
                j = j-1
            END WHILE
            WHILE  ( k[i] <= k[start]  &&  i < j )
                i = i+1
            END WHILE
            IF  i < j
                SWAP (k[i], k[j])
            END IF
        END WHILE
```

```
            SWAP (k[start], k[j])
            QUICKSORT (k, start, j-1)
            QUICKSORT (k, j+1, end)
        END IF
END PROCEDURE
```

Η μέθοδος της γρήγορης ταξινόμησης δικαιολογεί το όνομά της, καθώς είναι η πιο αποτελεσματική μέθοδος ταξινόμησης μεγάλου πλήθους τυχαίων δεδομένων. Το μόνο μειονέκτημα του αλγορίθμου που παρουσιάσαμε είναι η αναδρομικότητά του, η οποία μπορεί να επηρεάσει αρνητικά την απόδοσή του για μεγάλο βάθος αναδρομικών κλήσεων, όπως αναφέρθηκε και στην §3.2. Υπάρχει όμως δυνατότητα αφαίρεσης της αναδρομής από τον αλγόριθμο της γρήγορης ταξινόμησης με χρήση κατάλληλων τεχνικών.

4.1.4 Ταξινόμηση με Μείξη

Θα εξετάσουμε τώρα ένα διαφορετικό πρόβλημα, αυτό της μείξης δύο ήδη ταξινομημένων πινάκων σε ένα νέο ταξινομημένο πίνακα. Υποθέτουμε δηλαδή ότι δίνονται δύο πίνακες με *n* και *m* στοιχεία αντίστοιχα $a_0, a_1, a_2, \ldots, a_{n-1}$ και $b_0, b_1, b_2, \ldots, b_{m-1}$, για τα οποία ισχύει $a_i \le a_{i+1}$ και $b_j \le b_{j+1}$, και ζητείται να συγχωνευθούν σε ένα νέο πίνακα με $n+m$ στοιχεία $c_0, c_1, c_2, \ldots, c_{n+m-1}$ που να είναι επίσης ταξινομημένος.

Για την επίλυση αυτού του προβλήματος θα μπορούσαμε απλώς να αντιγράψουμε τα στοιχεία a_i και b_i στον νέο πίνακα και να χρησιμοποιηθεί μία από τις μεθόδους ταξινόμησης που έχουμε ήδη αναπτύξει· είναι όμως πολύ πιο αποτελεσματικό να εκμεταλλευτούμε το γεγονός ότι οι αρχικοί πίνακες είναι ήδη ταξινομημένοι.

Η μέθοδος της *ταξινόμησης με μείξη* (merge sort) διαβάζει ένα στοιχείο από κάθε πίνακα, έστω a_i και b_j, και καταχωρίζει το μικρότερο από τα δύο στο νέο πίνακα. Αν αυτό είναι το στοιχείο του πρώτου πίνακα (το a_i), τότε η διαδικασία συνεχίζεται λαμβάνοντας το επόμενο στοιχείο του πίνακα αυτού, a_{i+1}, και συγκρίνοντάς το με το b_j· αντίστοιχα αν μικρότερο είναι το b_j καταχωρίζεται αυτό στον πίνακα και λαμβάνεται το b_{j+1}. Οι συγκρίσεις συνεχίζονται μέχρι να τελειώσουν τα στοιχεία ενός από τους δύο πίνακες, οπότε τα υπόλοιπα στοιχεία του άλλου πίνακα απλώς αντιγράφονται στον τελικό.

Ένα παράδειγμα εφαρμογής της ταξινόμησης με μείξη φαίνεται στο επόμενο διάγραμμα. Σε κάθε βήμα συγκρίνονται τα στοιχεία που είναι στην «κορυφή» των πινάκων· το μικρότερο διαγράφεται και εισάγεται στον τελικό πίνακα, ο οποίος φαίνεται στην τελευταία γραμμή.

Αρχικοί πίνακες		Βήμα 1		Βήμα 2		Βήμα 3		Βήμα 4		Βήμα 5		Βήμα 6	
17	11	17	~~11~~	~~17~~									
23	37	23	37	23	37	~~23~~	37		~~37~~				
68	45	68	45	68	45	68	45	68	45	68	~~45~~	~~68~~	
78		78		78		78		78		78		~~78~~	
Τελικός πίνακας:		11		17		23		37		45		68	78

Για την υλοποίηση της μείξης δύο ταξινομημένων πινάκων απαιτούνται απλώς τρεις δείκτες i, j, k που δείχνουν τα τρέχοντα στοιχεία στους τρεις πίνακες a_i, b_j, c_k αντίστοιχα. Ο επόμενος ψευδοκώδικας υλοποιεί αυτό τον αλγόριθμο:

```
PROCEDURE MERGESORT (a[maxsize], n, b[maxsize], m, OUT c[maxsize])
    DATA i, j, k
    i = 0, j = 0, k = 0
    WHILE (i <= n-1 && j <= m-1)
        IF a[i] < b[j]
            c[k] = a[i]
            i = i+1
        ELSE
            c[k] = b[j]
            j = j+1
        END IF
        k = k+1
    END WHILE

    WHILE i <= n-1
        c[k] = a[i]
        i = i+1
        k = k+1
    END WHILE

    WHILE j <= m-1
        c[k] = b[j]
        j = j+1
        k = k+1
    END WHILE
END PROCEDURE
```

4.2 Αλγόριθμοι Αναζήτησης

Ένα άλλο πολύ σημαντικό πρόβλημα είναι αυτό της *αναζήτησης*. Με τον όρο αυτό εννοούμε την εξέταση μιας ακολουθίας στοιχείων καταχωρισμένων στη μνήμη του υπολογιστή με σκοπό την εύρεση ενός συγκεκριμένου στοιχείου, αν βέβαια αυτό υπάρχει στην ακολουθία.

Είναι προφανές ότι αν τα στοιχεία της ακολουθίας είναι τυχαία, η αναζήτηση δεν μπορεί παρά να είναι *σειραϊκή*: εξετάζονται τα στοιχεία της ένα προς ένα και η διαδικασία σταματά όταν βρεθεί το ζητούμενο στοιχείο· αν φτάσουμε στο τέλος της ακολουθίας τότε το ζητούμενο στοιχείο δεν υπάρχει σε αυτή. Αντίθετα, αν τα στοιχεία της ακολουθίας είναι εξαρχής ταξινομημένα, τότε μπορούμε να εκμεταλλευτούμε τη διάταξή τους για να επιταχύνουμε τη διαδικασία. Μία χαρακτηριστική τέτοια μέθοδος είναι η *δυαδική αναζήτηση* που παρουσιάζουμε παρακάτω.

Το πρόβλημα που επιλύουμε έχει ως εξής: Δίνεται μία ταξινομημένη ακολουθία *n* ακεραίων αριθμών $k_0, k_1, k_2, \ldots, k_{n-1}$ με $k_i < k_{i+1}$ για $i = 0\,(1)\,n-2$ καθώς και ένας ακέραιος αριθμός x. Ζητείται να βρεθεί η θέση m του x στην ακολουθία, αν βέβαια ο x υπάρχει σε αυτή, ή στην αντίθετη περίπτωση να δίνεται κατάλληλο μήνυμα.

4.2.1 Δυαδική Αναζήτηση

Όπως ήδη αναφέραμε, η δυαδική αναζήτηση είναι μία μέθοδος αναζήτησης που εφαρμόζεται σε ταξινομημένες ακολουθίες στοιχείων. Ο αλγόριθμος βασίζεται στην παρακάτω ιδέα: Αρχικά εξετάζουμε αν το μεσαίο στοιχείο της ακολουθίας (ακριβέστερα το στοιχείο της θέσης n div 2) ισούται με το δεδομένο στοιχείο. Αν αυτό ισχύει, τότε βέβαια ο αλγόριθμος τερματίζεται και η ζητούμενη θέση είναι η «μεσαία». Διαφορετικά εξετάζουμε αν το δεδομένο στοιχείο είναι μικρότερο ή μεγαλύτερο του μεσαίου. Στην πρώτη περίπτωση είναι φανερό ότι η αναζήτηση μπορεί να περιοριστεί στο πρώτο

μισό της αρχικής ακολουθίας, ενώ στη δεύτερη περίπτωση μπορεί να περιοριστεί στο δεύτερο μισό της. Η διαδικασία συνεχίζεται βρίσκοντας και πάλι το «μεσαίο» στοιχείο της ακολουθίας στην οποία έχουμε περιοριστεί, συγκρίνοντάς το με το δεδομένο στοιχείο, κ.ο.κ. μέχρι να βρεθεί η θέση του στοιχείου ή να διαπιστωθεί ότι δεν υπάρχει στην ακολουθία.

Για παράδειγμα, έστω ότι αναζητούμε το 63 στην ακολουθία 5, 16, 24, 28, 47, 63, 92, 98, 105, 200. Στον επόμενο πίνακα φαίνεται η ακολουθία που εξετάζεται σε κάθε βήμα του αλγορίθμου και με έντονα στοιχεία το μεσαίο στοιχείο της:

Θέση:	0	1	2	3	4	5	6	7	8	9
Βήμα 1:	5	16	24	28	**47**	63	92	98	105	200
Βήμα 2:						63	92	**98**	105	200
Βήμα 3:						**63**	92			

Το κέρδος της δυαδικής αναζήτησης σε σχέση με τη σειραϊκή αναζήτηση είναι φανερό, καθώς σε κάθε βήμα απαλλασσόμαστε από τα μισά στοιχεία της ακολουθίας. Η δυαδική αναζήτηση είναι μάλιστα η πιο αποτελεσματική μέθοδος αναζήτησης όταν το πλήθος το στοιχείων είναι σχετικά μικρό. Για μεγάλο πλήθος στοιχείων χρησιμοποιούνται πιο πολύπλοκες τεχνικές διαμέρισης της ακολουθίας, ώστε να είναι δυνατός ο αποκλεισμός όσο το δυνατό περισσότερων στοιχείων σε κάθε βήμα και η γρηγορότερη «προσέγγιση» της θέσης του δεδομένου στοιχείου.

Για την υλοποίηση του αλγορίθμου της δυαδικής αναζήτησης είναι απαραίτητοι δύο δείκτες lb και rb που αποθηκεύουν αντίστοιχα τη θέση του αριστερού και δεξιού ακραίου στοιχείου της εξεταζόμενης ακολουθίας, ο δείκτης m=(lb+rb) div 2 του μεσαίου στοιχείου, καθώς και μία λογική μεταβλητή found που παίρνει τιμή true ή false ανάλογα με το αν το στοιχείο x υπάρχει στην ακολουθία k. Ο ψευδοκώδικας που ακολουθεί παρουσιάζει τη διαδικασία της δυαδικής αναζήτησης:

```
PROCEDURE BINARYSEARCH (k[maxsize], n, x, OUT found, OUT m)
    DATA lb, rb
    lb = 0
    rb = n-1
    found = false
    DO
        m = (lb+rb) / 2
        IF  x == k[m]
            found = true
        ELSE
            IF  x < k[m]
                rb = m-1
            ELSE
                lb = m+1
            END IF
        END IF
    WHILE  ( found == false  &&  lb <= rb )
END PROCEDURE
```

Παράρτημα: Προγράμματα σε C++

Σε αυτό το παράρτημα δίνονται προγράμματα σε C++ για τους αλγορίθμους που παρουσιάστηκαν με ψευδοκώδικα στο κείμενο.

§2.1.4 - Παράδειγμα 1 Εμβαδό παραλληλογράμμου

```
#include <iostream.h>

int main()
{
    float width, height, surface;      // Μεταβλητές του προγράμματος

    cout << "Give width, height: ";
    cin >> width >> height;

    surface = width * height;

    cout << "Surface: " << surface << endl;

    return 0;
}
```

§2.2.1 - Παράδειγμα 1 Μέγιστος μεταξύ δύο αριθμών

```
#include <iostream.h>

int main()
{
    int x, y, max;

    cin >> x >> y;
    if (x > y)
        max = x;
    else
        max = y;
    cout << max << endl;

    return 0;
}
```

§2.2.1 - Παράδειγμα 2 Ταξινόμηση δύο αριθμών

```
#include <iostream.h>

int main()
{
    int x, y, tmp;

    cin >> x >> y;
    if (x < y)  {   // Αν χρειάζεται, οι αριθμοί εναλλάσσονται
        tmp = x;
        x = y;
        y = tmp;
    }
    cout << x << " " << y << endl;

    return 0;
}
```

§2.2.1 - Παράδειγμα 3 Μέγιστος τριών αριθμών

```
#include <iostream.h>

int main()
{
    int a, b, c, max;

    cin >> a >> b >> c;
    if (a > b)
        if (a > c)
            max = a;
        else
            max = c;
    else
        if (c > b)
            max = c;
        else
            max = b;
    cout << max << endl;

    return 0;
}
```

§2.2.2 - Παράδειγμα 1 Εκτύπωση αριθμών

```
#include <iostream.h>

int main()
{
    int n;

    cin >> n;
    switch (n)  {
        case 1:  {
            cout << "One" << endl;
            break;
        }
        case 2:  {
            cout << "Two" << endl;
            break;
        }
        case 3:  {
            cout << "Three" << endl;
            break;
        }
        case 4:  {
            cout << "Four" << endl;
            break;
        }
        case 5:  {
            cout << "Five" << endl;
            break;
        }
        default: {
            cout << "Wrong number." << endl;
            break;
        }
    }

    return 0;
}
```

§2.3.1 - Παράδειγμα 1 Άθροισμα αριθμών

```
#include <iostream.h>

int main()
```

```
{
    float number, sum;

    sum = 0;   // Αρχικοποίηση του αθροίσματος
    cin >> number;
    while (number != 0.0)  {
        sum += number;
        cin >> number;
    }
    cout << sum << endl;

    return 0;
}
```

§2.3.1 - Παράδειγμα 2 Δυνάμεις

```
#include <iostream.h>

int main()
{
    int x, n, number, power;

    cin >> x >> n;
    number = x;
    power = 1;
    while (power <= n)  {
        cout << number << endl;
        number *= x;
        power += 1;
    }

    return 0;
}
```

§2.3.1 - Παράδειγμα 3 Άθροισμα ψηφίων αριθμού

```
#include <iostream.h>

int main()
{
    int a, digit, sum;

    cin >> a;
    sum = 0;
    while (a > 0)  {
        digit = a % 10;
        sum += digit;
        a /= 10;
    }
    cout << sum << endl;

    return 0;
}
```

§2.3.1 - Παράδειγμα 4 Αριθμητικές σειρές

```
#include <iostream.h>

int main()
{
    int firstnum, num, maxnum, add;

    firstnum = 2;
    add = 3;

    cin >> maxnum;

    num = firstnum;
```

```
    while (num <= maxnum)  {
        cout << num << "  ";
        num += add;
        add += 3;
    }

    return 0;
}
```

§2.3.2 - Παράδειγμα 1 Εισαγωγή στοιχείων

```
#include <iostream.h>

int main()
{
    float final, exam, exer1, exer2;
    char answer;

    do  {
        cout << "Give grades for exam, 1st & 2nd exercise: ";
        cin >> exam >> exer1 >> exer2;

        final = 0.75*exam + 0.25*(exer1+exer2)/2.0;
        cout << "Final grade: " << final << endl;

        cout << "Continue? (y/n)  ";
        cin >> answer;
    }  while (answer == 'y');

    return 0;
}
```

§2.3.2 - Παράδειγμα 2 Μάντεψε τον αριθμό!

```
#include <iostream.h>

int main()
{
    const int secret = 135;
    int tries, guess;

    tries = 0;
    do  {
        cout << "Give a number: ";
        cin >> guess;
        tries++;

        if (guess > secret)
            cout << "Too high!" << endl;
        else if (guess < secret)
            cout << "Too low!"  << endl;
    }  while (guess != secret);

    cout << "You found it after "
         << tries << " tries" << endl;

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    const int secret = 135;
    int tries, guess;

    cout << "Give a number: ";
    cin >> guess;
    tries = 1;
    while (guess != secret)  {
        if (guess > secret)
            cout << "Too high!" << endl;
        else
            cout << "Too low!"  << endl;

        cout << "Give a number: ";
        cin >> guess;
        tries++;
    }

    cout << "You found it after "
         << tries << " tries" << endl;

    return 0;
}
```

§2.3.2 - Παράδειγμα 3 Ημέρες και ημερομηνίες

```
#include <iostream.h>

int main()
{
    int daysOfMonth, firstDay, weekDay, day;
```

```
    cout << "Number of days in month: ";
    cin  >> daysOfMonth;
    cout << "First day of the month (1=Monday, 7=Sunday): ";
    cin  >> firstDay;
    cout << "Weekday to view dates (1=Monday, 7=Sunday): ";
    cin  >> weekDay;

    day = weekDay - firstDay + 1;
    if (day <= 0)
        day += 7;
    do  {
        cout << day << "  ";
        day += 7;
    }  while (day <= daysOfMonth);

    return 0;
}
```

§2.3.3 - Παράδειγμα 1 Άθροισμα ακεραίων μεταξύ ορίων

```
#include <iostream.h>

int main()
{
    int a, b, sum;

    sum = 0;
    cin >> a >> b;

    if (a < b)  {
        for (int i = a; i <= b; i++)
            sum += i;
    }
    else  {
        for (int i = a; i >= b; i--)
            sum += i;
    }

    cout << sum << endl;

    return 0;
}
```

§2.3.3 - Παράδειγμα 2 Άθροισμα περιττών αριθμών

```
#include <iostream.h>

int main()
{
    int a, b, sum;

    sum = 0;
    cin >> a >> b;

    if (a % 2 == 0)
        a +=1;

    for (int i = a; i <= b; i+=2)
        sum += i;

    cout << sum << endl;

    return 0;
}
```

§2.3.3 - Παράδειγμα 3 Πρώτοι αριθμοί

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

int main()
{
    int n, p;
    bool isPrime;

    cin >> n;
    for (int k = 1; k <= n; k++)  {
        isPrime = true;
        p = 2;
        while ((isPrime == true)  &&  (p <= sqrt(k)))  {
            if (k % p == 0)
                isPrime = false;
            else
                p++;
        }
        if (isPrime == true)
            cout << k << "  ";
    }

    return 0;
}
```

§2.3.3 - Παράδειγμα 4 Πίνακας πολλαπλασιασμού

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
    int i, j;

    // Εκτύπωση της πρώτης γραμμής
    for (j = 0; j <= 9; j++)
        cout << setw(4) << j;
    cout << endl;

    // Εκτύπωση υπόλοιπων γραμμών
    for (i = 1; i <= 9; i++)  {
        // Εκτύπωση πρώτης στήλης
        cout << setw(4) << i;

        // Εκτύπωση υπόλοιπων στηλών
        for (j = 1; j <= i; j++)
            cout << setw(4) << i*j;

        cout << endl;
    }
    return 0;
}
```

§2.4 - Παράδειγμα 1 Ελάχιστο στοιχείο πίνακα και θέση του

```
#include <iostream.h>

int main()
{
    float numbers[10];
    float min;
    int i, minpos;

    // Εισαγωγή αριθμών
    for (i = 0; i < 10; i++)
        cin >> numbers[i];
```

```
    // Υπολογισμός ελαχίστου και θέσης του
    min = numbers[0];
    minpos = 0;
    for (i = 1; i < 10; i++)  {
        if (numbers[i] < min)  {
            min = numbers[i];
            minpos = i;
        }
    }

    cout << "Min: " << min << endl
         << "Pos: " << minpos << endl;

    return 0;
}
```

§2.4 - Παράδειγμα 2 Σύγκριση πινάκων

```
#include <iostream.h>

int main()
{
    const int n = 5;
    float a[n];
    float b[n];
    int i;

    for (i = 0; i < n; i++)
        cin >> a[i];
    for (i = 0; i < n; i++)
        cin >> b[i];

    bool equal = true;
    i = 0;
    while ((i < n)  &&  (equal == true))  {
        if (a[i] != b[i])
            equal = false;
        else
            i++;
    }

    if (equal == true)
        cout << "The arrays are equal" << endl;
    else
        cout << "The first " << i << " elements of the arrays are equal" << endl;

    return 0;
}
```

§2.4 - Παράδειγμα 3 Επόμενη ημερομηνία

```
#include <iostream.h>

int main()
{
    // Οι ημέρες κάθε μήνα. Στη θέση με δείκτη 0 βάζουμε 0 ώστε να μπορούμε
    // να αναφερθούμε στους μήνες με τον κανονικό τους αριθμό,
    // π.χ. για τον Οκτώβριο monthDays[10] (=31).
    int monthDays[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    int d1, m1, y1,   // Ημέρα, μήνας, έτος της ημερομηνίας που δίνεται
        d2, m2, y2;   // Ημέρα, μήνας, έτος της επόμενης ημερομηνίας

    cout << "Give a date (Day Month Year): ";
    cin >> d1 >> m1 >> y1;

    if ((d1 == 31)  &&  (m1 == 12))  {
```

```
        // Τέλος του έτους
        d2 = 1;
        m2 = 1;
        y2 = y1 + 1;
    }
    else  {
        if (d1 == monthDays[m1])  {
            // Τέλος του μήνα
            d2 = 1;
            m2 = m1 + 1;
        }
        else  {
            d2 = d1 + 1;
            m2 = m1;
        }
        y2 = y1
    }

    cout << "Next date: " << d2 << " / " << m2 << " / " << y2 << endl;

    return 0;
}
```

§2.4 - Παράδειγμα 4 Μετατόπιση στοιχείων πίνακα

```
#include <iostream.h>

int main()
{
    const int n = 5;
    int a[n];
    int i, k;

    // Εισαγωγή στοιχείων
    cout << "Give the elements of the array:" << endl;
    for (i = 0; i < n; i++)
        cin >> a[i];

    cout << "Give offset (0-" << n << "): ";
    cin >> k;

    // Μετακίνηση στοιχείων
    for (i = n-1; i >= k; i--)
        a[i] = a[i-k];

    // Προσθήκη μηδενικών στην αρχή
    for (i = 0; i <= k-1; i++)
        a[i] = 0;

    // Εκτύπωση τελικού πίνακα
    for (i = 0; i < n; i++)
        cout << a[i] << "  ";

    return 0;
}
```

§2.4 - Παράδειγμα 5 Πίνακες δύο διαστάσεων

```
#include <iostream.h>

int main()
{
    const int m = 3;   // Αριθμός γραμμών
    const int n = 4;   // Αριθμός στηλών
    float p[m][n];
    int i, j;
    float max, sum;
```

```
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            cin >> p[i][j];

    sum = 0;
    for (i = 0; i < m; i++)  {
        // Υπολογισμός μεγίστου γραμμής
        max = p[i][0];
        for (j = 1; j < n; j++)  {
            if (p[i][j] > max)
                max = p[i][j];
        }

        // Υπολογισμός αθροίσματος
        sum += max;
    }

    cout << sum << endl;

    return 0;
}
```

§2.4 - Παράδειγμα 6 Συμμετρικοί πίνακες

```
#include <iostream.h>

int main()
{
    const int n = 3;

    int matrix[n][n];
    int i, j;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            cin >> matrix[i][j];

    bool symmetric = true;
    i = 0;
    while ((i < n)  &&  (symmetric == true))  {
        j = 0;
        while ((j < n)  &&  (symmetric == true))  {
            if (matrix[i][j] != matrix[j][i])
                symmetric = false;
            j++;
        }
        i++;
    }

    if (symmetric == true)
        cout << "Symmetric" << endl;
    else
        cout << "Non-symmetric" << endl;

    return 0;
}
```

§3.1.1 - Παράδειγμα 1 Εκτύπωση ακεραίων μεταξύ ορίων

```
#include <iostream.h>

void PrintNumbers(int a, int b)
{
    for (int num = a; num <= b; num++)
        cout << num << endl;
}

int main()
```

```
{
    int min, max;

    cin >> min >> max;
    PrintNumbers(min, max);

    return 0;
}
```

§3.1.2 - Παράδειγμα 1 Μέγιστο στοιχείο πίνακα

```
#include <iostream.h>

float ArrayMax(float array[], int size)
{
    float max = array[0];

    for (int i = 1; i < size; i++)  {
        if (array[i] > max)  {
            max = array[i];
        }
    }

    return max;
}

int main()
{
    float array[5];

    for (int i = 0; i < 5; i++)
        cin >> array[i];

    float m = ArrayMax(array, 5);
    cout << m << endl;

    return 0;
}
```

§3.1.3 - Παράδειγμα 1 Μέγιστο στοιχείο πίνακα και θέση του

```
#include <iostream.h>

void ArrayMaxPos(float array[], int size, float& max, int& pos)
{
    pos = 0;

    for (int i = 1; i < size; i++)  {
        if (array[i] > array[pos])  {
            pos = i;
        }
    }
    max = array[pos];
}

int main()
{
    float array[5];
    float m;
    int p;

    for (int i = 0; i < 5; i++)
        cin >> array[i];

    ArrayMaxPos(array, 5, m, p);
    cout << m << " " << p << endl;

    return 0;
```

```
}
```

§3.1.3 - Παράδειγμα 2 Ανταλλαγή τιμών μεταβλητών

```
#include <iostream.h>

void Swap(int& x, int& y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int main()
{
    int a, b;
    cin >> a >> b;
    Swap(a, b);
    cout << a << "  " << b << endl;

    return 0;
}
```

§3.1.4 - Παράδειγμα 1 Πρόσθεση πινάκων

```
#include <iostream.h>

void ReadArray(int array[], int size)
{
    for (int i = 0; i < size; i++)
        cin >> array[i];
}

void AddArrays(int array1[], int array2[], int sum[], int size)
{
    for (int i = 0; i < size; i++)
        sum[i] = array1[i] + array2[i];
}

void WriteArray(int array[], int size)
{
    cout << " [ ";
    for (int i = 0; i < size; i++)
        cout << array[i] << " ";
    cout << "] ";
}

int main()
{
    const int n = 3;
    int a[n];
    int b[n];
    int c[n];

    ReadArray(a, n);
    ReadArray(b, n);

    AddArrays(a, b, c, n);

    WriteArray(a, n);
    cout << " + ";
    WriteArray(b, n);
    cout << " = ";
    WriteArray(c, n);

    return 0;
}
```

§3.2 - Παράδειγμα 1 Υπολογισμός παραγοντικού

```
int factorial(int n)
{
    int fact = 1;
    for (int i = 1; i <= n; i++)
        fact *= i;
    return fact;
}
```

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

§3.2 - Παράδειγμα 2 Αριθμοί Fibonacci

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

§3.2 - Παράδειγμα 3 Πύργοι του Ανόι

```
void MoveDisk(int a, int b)
{
    cout << "Moving from " << a << " to " << b << endl;
}

void MoveTower(int h, int a, int b, int c)
{
    if (h == 1)
        MoveDisk(a, b);
    else  {
        MoveTower(h-1, a, c, b);
        MoveDisk(a, b);
        MoveTower(h-1, c, b, a);
    }
}
```

§4.1.1 Μέθοδος της Φυσαλλίδας

```
void BubbleSort(int k[], int n)
{
    // Ταξινόμηση σε αύξουσα σειρά με τη μέθοδο της Φυσαλλίδας
    bool sorted = false;
    int t;

    t = 0;
    do {
        sorted = true;
        for (int i = n-1; i >= t+1; i--)  {
            if (k[i] < k[i-1])  {   // Τα μικρότερα στοιχεία ανεβαίνουν επάνω
                SwapInt(k[i], k[i-1]);
                sorted = false;
            }
        }

        t++;
    }  while ((!sorted)  &&  (t <= n-2));
}
```

§4.1.2 Μέθοδος της Επιλογής

```
void SelectionSort(int k[], int n)
{
    // Ταξινόμηση σε αύξουσα σειρά με τη μέθοδο της Επιλογής
    for (int t = 0; t <= n-2; t++)  {
        int m = t;
```

```
        for (int i = t; i <= n-1; i++)  {
            // Εύρεση του ελάχιστου στοιχείου από όσα δεν έχουν ταξινομηθεί
            if (k[i] < k[m])
                m = i;
        }
        SwapInt(k[t], k[m]);
    }
}
```

§4.1.3 Γρήγορη Ταξινόμηση

```
void QuickSort(int k[], int start, int end)
{
    // Ταξινόμηση σε αύξουσα σειρά με τη μέθοδο της Γρήγορης Ταξινόμησης
   if (start < end)  {
        int i = start,
            j = end;

        // Εύρεση τελικής θέσης οδηγού στοιχείου
        while (i < j)  {
            while (k[j] > k[start])
                j--;
            while ((k[i] <= k[start])  &&  (i < j))
                i++;
            if (i < j)
                SwapInt(k[i], k[j]);
        }
        SwapInt(k[start], k[j]);

        // Ταξινόμηση των τμημάτων αριστερά και δεξιά του οδηγού στοιχείου
        QuickSort(k, start, j-1);
        QuickSort(k, j+1, end);
    }
}
```

§4.1.4 Ταξινόμηση με Μείξη

```
void MergeSort(int a[], int n, int b[], int m, int c[])
{
    // Μείξη δύο ταξινομημένων πινάκων
    int i = 0, j = 0, k = 0;

    // Όσο υπάρχουν στοιχεία και στους δύο πίνακες, τα τοποθετούμε
    // στη σωστή σειρά στον τελικό πίνακα
    while ((i <= n-1)  &&  (j <= m-1))  {
        if (a[i] < b[j])  {
            c[k] = a[i];
            i = i+1;
        }
        else  {
            c[k] = b[j];
            j = j+1;
        }
        k = k+1;
    }

    // Αν έχουν μείνει στοιχεία στον πίνακα a, τα αντιγράφουμε στον τελικό
    while (i <= n-1)  {
        c[k] = a[i];
        i = i+1;
        k = k+1;
    }

    // Αν έχουν μείνει στοιχεία στον πίνακα b, τα αντιγράφουμε στον τελικό
    while (j <= m-1)  {
        c[k] = b[j];
        j = j+1;
        k = k+1;
```

```
    }
}
```

§4.2.1 Δυαδική Αναζήτηση

```
void BinarySearch(int k[], int n, int x, bool& found, int& m)
{
    int lb = 0,
        rb = n-1;

    found = false;

    do  {
        m = (lb + rb) / 2;   // Θέση του μεσαίου στοιχείου

        if (x == k[m])  {
            found = true;    // Το ζητούμενο βρέθηκε, θα βγούμε από το βρόχο
        }
        else  {
            if (x < k[m])
                rb = m-1;    // Περιοριζόμαστε αριστερά του μεσαίου
            else
                lb = m+1;    // Περιοριζόμαστε δεξιά του μεσαίου
        }
    }  while ((!found)  &&  (lb <= rb));
}
```