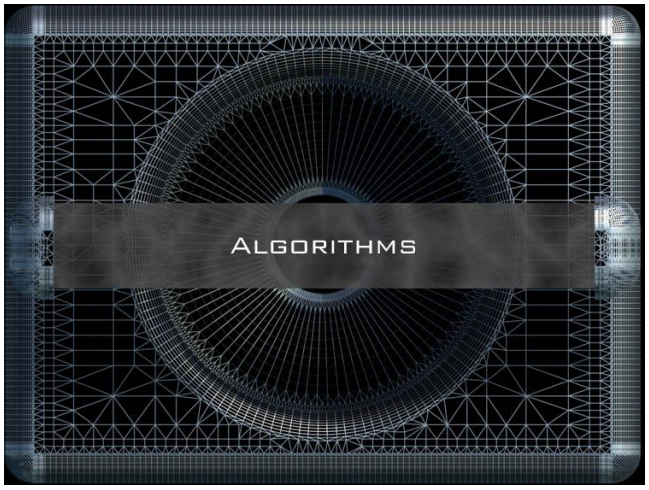
ΑΛΕΞΑΝΔΡΕΙΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ (ΤΕ)

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«Ανασκόπηση βασικών εννοιών της θεωρίας πολυπλοκότητας αλγορίθμων»



Της φοιτήτριας
Αθανασιάδου Ειρήνης
ΑΜ: 083373

Επιβλέπων καθηγητής:
Αντωνίου Ευστάθιος

Θεσσαλονίκη 2014

ΠΕΡΙΛΗΨΗ

Στην παρούσα πτυχιακή εργασία γίνεται ανάλυση των βασικών εννοιών της θεωρίας υπολογιστικής πολυπλοκότητας. Η θεωρία της πολυπλοκότητας είναι το μέρος εκείνο της θεωρίας υπολογισμού, το οποίο ασχολείται με την κοστολόγηση των πόρων που απαιτούνται για την αλγοριθμική επίλυση ενός προβλήματος. Επομένως η θεωρία πολυπλοκότητας αποτελεί βασικό δομικό λίθο της ανάλυσης αλγορίθμων και κεντρικό γνωστικό πεδίο της επιστήμης υπολογιστών. Στο πρώτο κεφάλαιο αναφέρονται οι βασικές έννοιες του υπολογιστικού προβλήματος, του στιγμιότυπου ενός προβλήματος και του αλγόριθμου. Στο δεύτερο κεφάλαιο περιγράφονται δύο βασικοί πόροι για την ανάλυση της πολυπλοκότητας που είναι η χρονική και η χωρική πολυπλοκότητα των αλγορίθμων. Το τρίτο κεφάλαιο αναφέρεται στην ασυμπτωτική εκτίμηση των αλγορίθμων, η οποία προσδιορίζει την τάξη μεγέθους του χρόνου εκτέλεσής τους. Επίσης αναλύεται ο ασυμπτωτικός συμβολισμός και συγκεκριμένα οι συμβολισμοί Θ, Ο, Ω, ω, ο. Στο τέταρτο κεφάλαιο περιγράφονται οι αναδρομικοί αλγόριθμοι, οι αλγόριθμοι τύπου «διαίρει και βασίλευε» καθώς και οι «άπληστοι» αλγόριθμοι. Το πέμπτο κεφάλαιο παρουσιάζει τον δυναμικό προγραμματισμό και τη θεωρία των γραφημάτων. Το έκτο και τελευταίο κεφάλαιο περιγράφει τους αποδοτικούς αλγόριθμους και αναλύει τις κλάσεις πολυπλοκότητας P και NP.

ABSTRACT

In the present thesis we review the basic concepts of the theory of computational complexity. This theory is that part of the computation theory, which is concerned with the evaluation of the resources required for the algorithmic solution of a given problem. In this respect, complexity theory is a basic tool for the analysis of algorithms and hence a very important topic in the field of computer science. In the first chapter we review notions of a computational problem, an instance of the problem and the algorithms used for its solution. The second chapter describes two central quantities used in the analysis of the complexity of a computation that is time and space complexity of algorithms. The third chapter refers to the asymptotic estimation of the complexity of an algorithm, which determines the order of magnitude of its run time. We also present the asymptotic notations and particularly the notations Θ, O, Ω, ω, o. The fourth chapter describes the structure of recursive, “divide and conquer” and greedy algorithms. The fifth chapter presents the principles of dynamic programming and some elements of graph theory. The sixth and last chapter describes the notion of efficient algorithms and analyzes the complexity classes P and NP.

Περιεχόμενα

Κεφάλαιο 1: Βασικές έννοιες

1.1 Υπολογιστικό πρόβλημα. Ο μετασχηματισμός ενός συνόλου δεδομένων εισόδου σε ένα σύνολο δεδομένων εξόδου ορίζεται ως υπολογιστικό πρόβλημα. Τα δεδομένα εισόδου έχουν τη μορφή ενός έγκυρου στιγμιότυπου, ενώ τα δεδομένα εξόδου έχουν τη μορφή και τις ιδιότητες μιας απάντησης ή μιας λύσης. Ένα υπολογιστικό πρόβλημα ορίζει τη μορφή και τους περιορισμούς που πρέπει να ικανοποιούν τα δεδομένα εισόδου και τα δεδομένα εξόδου. Για παράδειγμα σε ένα πρόβλημα πολλαπλασιασμού δύο αριθμών τα δεδομένα εισόδου είναι οι δύο αριθμοί (x,y), και τα δεδομένα εξόδου είναι το γινόμενό τους x*y.

1.1.1 Τύποι προβλημάτων

Με κριτήριο τη δυνατότητα επίλυσης ενός προβλήματος διακρίνονται τρεις κατηγορίες προβλημάτων:

- **Επιλύσιμα,** χαρακτηρίζονται τα προβλήματα για τα οποία η λύση τους είναι ήδη γνωστή και έχει διατυπωθεί.
- **Ανοικτά,** ονομάζονται τα προβλήματα για τα οποία η λύση τους δεν έχει μεν ακόμα βρεθεί, αλλά παράλληλα δεν έχει αποδειχθεί, ότι δεν επιδέχονται λύση.
- **Άλυτα,** χαρακτηρίζονται τα προβλήματα για τα οποία έχουμε φτάσει στη παραδοχή, ότι δεν επιδέχονται λύση.

Με κριτήριο το βαθμό δόμησης των λύσεων τους, τα επιλύσιμα προβλήματα μπορούν να διακριθούν σε τρεις βασικές κατηγορίες:

- **Δομημένα,** χαρακτηρίζονται τα προβλήματα των οποίων η επίλυση προέρχεται από μια αυτοματοποιημένη διαδικασία.
- **Ημιδομημένα**, ονομάζονται τα προβλήματα των οποίων η λύση επιδιώκεται στα πλαίσια ενός εύρους πιθανών λύσεων, αφήνοντας στον ανθρώπινο παράγοντα τα περιθώρια επιλογής.
- **Αδόμητα,** χαρακτηρίζονται τα προβλήματα των οποίων οι λύσεις δεν μπορούν να δομηθούν ή δεν έχει διερευνηθεί σε βάθος η δυνατότητα δόμησης τους.

Με κριτήριο το είδος επίλυσης που επιζητούν τα προβλήματα διακρίνονται σε τρεις κατηγορίες:

- **Απόφασης,** όπου η απόφαση που πρόκειται να ληφθεί ως λύση του προβλήματος που τίθεται απαντά σε ένα ερώτημα και πιθανόν αυτή η απάντηση να είναι ένα "Ναι" ή ένα "Όχι". Αυτό που θέλουμε να διαπιστώσουμε σε ένα πρόβλημα απόφασης είναι αν υπάρχει απάντηση που ικανοποιεί τα δεδομένα που θέτονται από το πρόβλημα.
- **Υπολογιστικά**, όπου το πρόβλημα που τίθεται απαιτεί τη διενέργεια υπολογισμών, για να μπορεί να δοθεί μια απάντηση στο πρόβλημα. Σε ένα υπολογιστικό πρόβλημα ζητάμε να βρούμε τη τιμή της απάντησης που ικανοποιεί τα δεδομένα του προβλήματος.
- **Βελτιστοποίησης,** όπου το πρόβλημα που τίθεται επιζητά το βέλτιστο αποτέλεσμα για τα συγκεκριμένα δεδομένα που διαθέτει. Σε ένα πρόβλημα βελτιστοποίησης αναζητούμε την απάντηση που ικανοποιεί κατά τον καλύτερο τρόπο τα δεδομένα που παρέχει το πρόβλημα.

1.2 Στιγμιότυπο προβλήματος.

Ως στιγμιότυπο προβλήματος χαρακτηρίζεται κάθε σύνολο δεδομένων εισόδου που πληροί τους περιορισμούς που τίθενται στον ορισμό ενός υπολογιστικού προβλήματος. Για παράδειγμα στο πρόβλημα του πολλαπλασιασμού δύο αριθμών, οι αριθμοί (2,3) αποτελούν ένα στιγμιότυπο. Κάθε στιγμιότυπο ενός προβλήματος μπορεί να έχει καμία, μία ή και περισσότερες λύσεις. Οποιοδήποτε σύνολο δεδομένων εξόδου, το οποίο σε συνδυασμό με το στιγμιότυπο ικανοποιεί τους περιορισμούς που τίθενται στον ορισμό του προβλήματος, αποτελεί τη λύση ενός στιγμιότυπου για το πρόβλημα. Για παράδειγμα το 6 αποτελεί λύση του στιγμιότυπου (2,3) για το πρόβλημα του πολλαπλασιασμού. Το σύνολο των στιγμιοτύπων ενός προβλήματος είναι άπειρο.

1.3 Αλγόριθμος: ως αλγόριθμος ορίζεται μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος. Ο όρος αλγόριθμος χρησιμοποιείται για να δηλώσει μεθόδους που εφαρμόζονται για την επίλυση προβλημάτων. Είναι μια διαδικασία ή ένα σύνολο κανόνων με σκοπό τον υπολογισμό.

Κάθε αλγόριθμος πρέπει να πληροί κάποια κριτήρια:

1. Να δέχεται σαν **είσοδο** ένα σύνολο δεδομένων τα οποία μετασχηματίζει σε ένα σύνολο δεδομένων εξόδου.
2. Ως **έξοδο** πρέπει να δημιουργεί τουλάχιστον μία τιμή δεδομένων σαν αποτέλεσμα. Επομένως ένας αλγόριθμος θεωρείται εργαλείο επίλυσης ενός υπολογιστικού προβλήματος, όπου ο ορισμός του προβλήματος καθορίζει την επιθυμητή σχέση μεταξύ των δεδομένων εισόδου και εξόδου.
3. Ένα άλλο κριτήριο είναι η **καθοριστικότητα** του αλγορίθμου, όπου κάθε εντολή πρέπει να καθορίζεται χωρίς καμία αμφιβολία για τον τρόπο εκτέλεσής της.
4. Ένας αλγόριθμος πρέπει να είναι **πεπερασμένος**, δηλαδή να τελειώνει σε πεπερασμένο αριθμό βημάτων.
5. Ως τελευταίο κριτήριο ορίζεται η **αποτελεσματικότητα**, όπου κάθε μεμονωμένη εντολή του αλγορίθμου πρέπει να είναι απλή. Δηλαδή μια εντολή δεν αρκεί να έχει ορισθεί αλλά πρέπει να είναι και εκτελέσιμη.

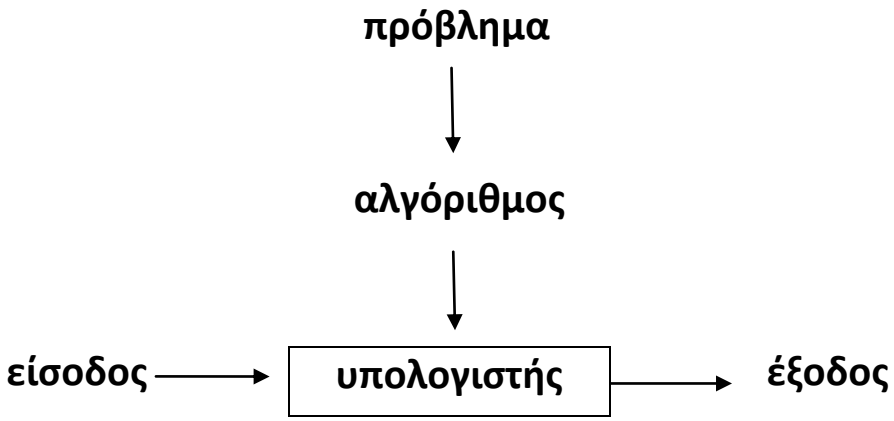
Οι αλγόριθμοι διακρίνονται ανάλογα με την τεχνική επίλυσης ενός προβλήματος σε:

- **Αναδρομικοί:** χρησιμοποιούν αναδρομικές λύσεις προβλημάτων και επιλύουν ένα πρόβλημα, λύνοντας ένα ή περισσότερα στιγμιότυπα του ίδιου προβλήματος.
- **Διαίρει και βασίλευε:** διαιρούν το πρόβλημα σε μικρότερα υποπροβλήματα και στη συνέχεια οι λύσεις τους συνδυάζονται για να προκύψει η λύση του αρχικού.
- **Άπληστοι:** επιλύουν προβλήματα επιλέγοντας κάθε φορά την τοπικά βέλτιστη λύση προσδοκώντας την συνολικά βέλτιστη.
- **Δυναμικού προγραμματισμού:** επιλύουν προβλήματα χωρίζοντάς τα σε ανεξάρτητα υποπροβλήματα και αποθηκεύουν τη λύση υποπροβλημάτων σε πίνακα για την επαναχρησιμοποίησή της.
- **Παράλληλοι:** μπορούν να διαχειριστούν ένα μεγάλο αριθμό επεξεργαστών που να δουλεύουν παράλληλα για την ολοκλήρωση ενός συνολικού υπολογισμού.

Ανάλογα με τη λύση που επιτυγχάνουν οι αλγόριθμοι μπορούν να διακριθούν σε:

- **Βέλτιστοι ή Άριστοι:** βρίσκουν τη βέλτιστη λύση ενός προβλήματος.
- **Προσεγγιστικοί ή ευριστικοί:** βρίσκουν όσο το δυνατόν καλύτερες λύσεις σε άλυτα ή πολύ δύσκολα προβλήματα.

Η έννοια του αλγορίθμου



Σχήμα 1: Η έννοια του αλγορίθμου

1.3.1 Ορθότητα αλγορίθμου.

Όπως αναφέρθηκε παραπάνω, ένα πρόβλημα έχει άπειρο αριθμό στιγμιότυπων. Για να θεωρηθεί ορθός ένας αλγόριθμος, πρέπει οι λύσεις που υπολογίζει να είναι σωστές για όλα τα στιγμιότυπα του προβλήματος που λύνει. Όταν θέλουμε να επιβεβαιώσουμε την ορθότητα ενός αλγορίθμου για κάποιο πρόβλημα, αρκεί να αποδείξουμε ότι ο αλγόριθμος υπολογίζει μία λύση για κάθε στιγμιότυπο του προβλήματος. Στην αντίθετη περίπτωση όπου θέλουμε να επιβεβαιώσουμε ότι ενός αλγόριθμος για κάποιο πρόβλημα δεν είναι ορθός, μπορούμε να παρουσιάσουμε ένα στιγμιότυπο του προβλήματος για το οποίο ο αλγόριθμος δεν μπορεί να υπολογίσει κάποια λύση ή η λύση που υπολογίζει δεν είναι σωστή.

Κεφάλαιο 2: Χρονική και χωρική πολυπλοκότητα αλγορίθμων

2.1 Πολυπλοκότητα

Για να χαρακτηριστεί ένας αλγόριθμος «καλός» ή όχι, πρέπει να οριστούν κάποια κριτήρια που είναι απαραίτητο να πληροί. Αυτό γίνεται πριν από τη σχεδίασή του. Συνεπώς το βασικό χαρακτηριστικό ενός αλγορίθμου είναι η αποδοτικότητά του, δηλαδή το κατά πόσο είναι ή δεν είναι αποδοτικός. Η έννοια της απόδοσης περιγράφει τη χρήση των υπολογιστικών πόρων που είναι απαραίτητοι για την επίλυση ενός προβλήματος. Αναλόγως το είδος του αλγορίθμου, οι υπολογιστικοί πόροι μπορεί να είναι η μνήμη, η κεντρική μονάδα επεξεργασίας, ή ακόμα και κάποιοι πιθανοί δικτυακοί πόροι που χρησιμοποιούνται. Μεγαλύτερη ωστόσο σημασία για το περισσότερο πλήθος προβλημάτων, έχει η χρήση της κεντρικής μονάδας επεξεργασίας. Με τον έλεγχο της χρήσης της κεντρικής μονάδας επεξεργασίας, αντιλαμβανόμαστε το χρόνο που απαιτείται για την εκτέλεση του αλγορίθμου.

Είναι συχνές οι περιπτώσεις που για την επίλυση ενός προβλήματος μπορεί να έχουμε περισσότερους από έναν αλγορίθμους, που ο καθένας τους να μπορεί να υλοποιηθεί σε ηλεκτρονικό υπολογιστή. Στις περιπτώσεις αυτές διακρίνουμε ως καλύτερο τον αλγόριθμο που χρησιμοποιεί τη μικρότερη υπολογιστική ισχύ από την κεντρική μονάδα επεξεργασίας και τις λιγότερες θέσεις μνήμης. Για να τεκμηριώσουμε ότι ένας αλγόριθμος είναι βέλτιστος, χρησιμοποιούμε σαν μέτρο την πολυπλοκότητά του.

Η θεωρία πολυπλοκότητας ανήκει στο κομμάτι της θεωρίας υπολογισμού, που ασχολείται με το κόστος που έχουν οι πόροι που απαιτούνται για την αλγοριθμική επίλυση ενός προβλήματος. Επομένως η θεωρία πολυπλοκότητας αποτελεί βασικό δομικό λίθο της ανάλυσης αλγορίθμων και κεντρικό γνωστικό πεδίο της επιστήμης υπολογιστών.

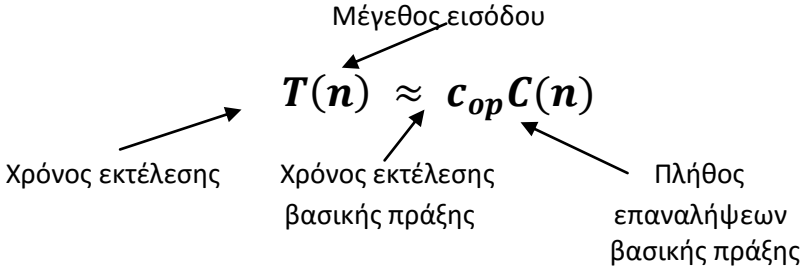
Όταν εξετάζεται η πολυπλοκότητα αλγορίθμων, γίνεται σύγκριση μεταξύ δύο αλγορίθμων στο επίπεδο της ιδέας και αγνοούνται οι λεπτομέρειες χαμηλού επιπέδου όπως για παράδειγμα το σύνολο των εντολών του επεξεργαστή, η γλώσσα προγραμματισμού που χρησιμοποιείται και το υλικό που χρησιμοποιεί ο αλγόριθμος για να υλοποιηθεί. Οι συνηθέστεροι πόροι για τους οποίους ενδιαφερόμαστε είναι ο χρόνος, οπότε μιλάμε για τη χρονική πολυπλοκότητα του αλγορίθμου, δηλαδή πόσα βήματα χρειάζεται να εκτελέσει ο αλγόριθμος συναρτήσει της εισόδου του, και ο χώρος, οπότε μιλάμε για τη χωρική πολυπλοκότητα, δηλαδή πόσο χώρο (μνήμη) χρειάζεται ο

αλγόριθμος συναρτήσει της εισόδου του. Εκτός από αυτούς τους πόρους, κατά περίπτωση, μπορεί να ενδιαφερόμαστε και για άλλους, όπως για παράδειγμα πόσοι παράλληλοι επεξεργαστές χρειάζονται για να λυθεί ένα πρόβλημα παράλληλα.

Η ανάλυση πολυπλοκότητας είναι επίσης ένα εργαλείο που μας επιτρέπει να εξηγήσουμε πώς ένας αλγόριθμος συμπεριφέρεται καθώς η είσοδος μεγαλώνει. Επίσης, μας επιτρέπει να εκτιμήσουμε πώς θα συμπεριφερθεί ο αλγόριθμος, αν του δώσουμε διαφορετική είσοδο. Για παράδειγμα, αν ο αλγόριθμός μας χρειάζεται 1 δευτερόλεπτο για να τρέξει σε είσοδο με μέγεθος 1000, θέλουμε να εξετάσουμε πώς θα συμπεριφερθεί αν διπλασιάσουμε το μέγεθος της εισόδου, αν θα τρέξει το ίδιο γρήγορα, με τη μισή ταχύτητα, ή τέσσερις φορές πιο αργά. Στον προγραμματισμό στην πράξη, αυτό είναι σημαντικό καθώς μας επιτρέπει να προβλέψουμε πώς ο αλγόριθμός μας θα συμπεριφερθεί καθώς η είσοδος μεγαλώνει. Για παράδειγμα, αν έχουμε φτιάξει έναν αλγόριθμο για μία εφαρμογή που δουλεύει καλά με 1000 χρήστες και μετρήσουμε το χρόνο εκτέλεσής του, χρησιμοποιώντας ανάλυση πολυπλοκότητας αλγορίθμων μπορούμε να έχουμε μια αρκετά καλή ιδέα για το τι θα συμβεί μόλις φτάσουμε τους 2000 χρήστες. Οπότε αν έχουμε μετρήσει τη συμπεριφορά του προγράμματός μας για μικρή είσοδο, μπορούμε να έχουμε μία καλή ιδέα για το πώς θα συμπεριφερθεί για μεγαλύτερη είσοδο.

2.1.1 Χρονική πολυπλοκότητα

Μια από τις πιο κοινές μετρικές πολυπλοκότητας είναι η χρονική πολυπλοκότητα που εκφράζει το χρόνο που απαιτείται για την εκτέλεση ενός προγράμματος. Η χρονική πολυπλοκότητα αναλύεται προσδιορίζοντας τον αριθμό των επαναλήψεων της βασικής πράξης ως συνάρτηση του μεγέθους εισόδου. Η βασική πράξη είναι αυτή που συνεισφέρει περισσότερο από τις άλλες στο χρόνο εκτέλεσης του αλγορίθμου.



Σχήμα 2: Ανάλυση χρονικής πολυπλοκότητας

Πίνακας 1: Βασικά παραδείγματα προβλημάτων - Ανάλυση μεγέθους εισόδου και βασικής πράξης

Πρόβλημα	Μέγεθος εισόδου	Βασική πράξη
Αναζήτηση κλειδιού σε λίστα με n αντικείμενα	Το πλήθος n των αντικειμένων	Συγκρίσεις κλειδιών
Πολλαπλασιασμός πινάκων με πραγματικούς αριθμούς	Διαστάσεις των πινάκων	Πολλαπλασιασμός πραγματικών αριθμών
Υπολογισμός α^n	n	Πολλαπλασιασμός πραγματικών αριθμών
Προβλήματα με γράφους	Πλήθος κορυφών ή και ακμών	Η επίσκεψη ενός κόμβου ή η διάσχιση μιας ακμής

Παράδειγμα αλγορίθμου 1

- Πρόβλημα: Δίνεται μια ακολουθία $\{\alpha_i\}=\alpha_1,\ldots..,\alpha_n$, $\alpha_i \in \mathbb{N}$, όπου ζητείται το μεγαλύτερο στοιχείο της.
- Αλγόριθμος **MAX1**:
    - Θέσε την τιμή μιας *προσωρινής μεταβλητής v* (που θα αποθηκεύει το μεγαλύτερο στοιχείο που έχουμε δει μέχρι τώρα) στην τιμή a_1
    - Για i=2,3,…,n επανέλαβε τα εξής:
        - Πάρε το επόμενο στοιχείο a_i στην ακολουθία. Αν $a_i>v$, τότε ανάθεσε στην v τον αριθμό a_i. Αλλιώς μην κάνεις τίποτα.
    - Επέστρεψε την τιμή της μεταβλητής v.

- **Απόδειξη Ορθότητας**: Με επαγωγή στον αριθμό των βημάτων (επαναλήψεων) του αλγόριθμου
    - ***Βάση***: Για i=2, η μεταβλητή v έχει την μεγαλύτερη τιμή ανάμεσα στους a_1 και a_2

- ***Υπόθεση***: Για $i=k$, η μεταβλητή *v* έχει την μεγαλύτερη τιμή ανάμεσα στους $a_1, \ldots, a_k$
- ***Επαγωγικό Βήμα***: Η εκτέλεση του βήματος $k+1$ συγκρίνει τον αριθμό a_{k+1} με τον μέγιστο ανάμεσα στους αριθμούς $a_1, \ldots, a_k$ και αναθέτει τον μεγαλύτερο από τους δύο στη μεταβλητή *v*. Άρα μετά την εκτέλεση του βήματος, η μεταβλητή *v* θα έχει αποθηκευμένο τον μέγιστο ανάμεσα στους αριθμούς $a_1, \ldots, a_{k+1}$

- **Χρονική Πολυπλοκότητα**
  - Καθένας από τους αριθμούς $a_1, \ldots, a_n$ συγκρίνεται με τη μεταβλητή *v* μια φορά. Αν *c* είναι ο χρόνος που απαιτείται για να συγκριθούν δυο αριθμοί, τότε ο συνολικός χρόνος που απαιτείται για τις συγκρίσεις είναι $c(n-1)$.
    - αν λάβουμε υπ' όψιν μας και τον χρόνο που απαιτείται για την αντιγραφή ενός αριθμού στη μεταβλητή *v* τότε η ανάλυση γίνεται πολύ πιο περίπλοκη αφού πρέπει να κάνουμε υποθέσεις για την κατανομή των αριθμών στην ακολουθία. Απλά υποθέτουμε ότι ο χρόνος αυτός συμπεριλαμβάνεται στην ποσότητα *c*.
    - Το *c* εξαρτάται από τον συγκεκριμένο Η/Υ που χρησιμοποιούμε. Για αυτό λέμε ότι η χρονική πολυπλοκότητα του αλγόριθμου είναι ανάλογη προς το $n-1$.

Παράδειγμα αλγορίθμου 2

- Πρόβλημα: Δίνεται μια ακολουθία $\{\alpha_i\}=\alpha_1,\ldots..,\alpha_n$, $\alpha_i \in \mathbb{N}$, όπου ζητείται το μεγαλύτερο στοιχείο της.
- Αλγόριθμος **MAX2**:
  - Για $i=1,3,\ldots,n-1$ επανέλαβε τα εξής:
    - Σύγκρινε τα στοιχεία a_i και a_{i+1} της ακολουθίας. Αν $a_i > a_{i+1}$, τότε άλλαξε θέση στους δύο αριθμούς. Αλλιώς μην κάνεις τίποτα.
  - Επέστρεψε την τιμή του στοιχείου a_n.
- **Απόδειξη Ορθότητας**
  - Με επαγωγή

- **Χρονική Πολυπλοκότητα**
    - Ο αλγόριθμος MAX2 εκτελεί *n*-1 συγκρίσεις αριθμών. Άρα η χρονική του πολυπλοκότητα είναι ανάλογη προς το *n*-1

Η διαφορά που έχει ο MAX2 με τον MAX1, είναι ότι μεταβάλει τα δεδομένα στη μνήμη αλλάζοντας θέση στους αριθμούς, ενώ ο MAX1 απλά γράφει το αποτέλεσμα σε μια άλλη θέση μνήμης. Για την επίλυση ενός προβλήματος είναι πιθανό να υπάρχει πλήθος διαφορετικών αλγορίθμων που μπορεί να έχουν ίδιες πολυπλοκότητες ή και διαφορετικές. Οι περισσότεροι αλγόριθμοι έχουν διαφορετικούς χρόνους εκτέλεσης για εισόδους διαφορετικού μεγέθους. Παραδείγματος χάρη η αναζήτηση σε μια μεγάλη λίστα συνήθως πραγματοποιείται σε περισσότερο χρόνο από την αναζήτηση σε μια μικρή λίστα. Για αυτό τον λόγο, η χρονική πολυπλοκότητα συνήθως εκφράζεται ως συνάρτηση του μεγέθους της εισόδου. Αυτή η συνάρτηση συνήθως δίνει την πολυπλοκότητα για την ***χειρότερη περίπτωση*** εισόδου κάθε συγκεκριμένου μεγέθους. Πιο σπάνια χρησιμοποιούνται και συναρτήσεις που εκφράζουν τις πολυπλοκότητες ***καλύτερης*** και ***μέσης περίπτωσης.***

Η **χειρότερη περίπτωση** ενός αλγορίθμου αφορά το μέγιστο κόστος εκτέλεσης του αλγορίθμου, το οποίο μετράται σε υπολογιστικούς πόρους. Το κόστος αυτό πολλές φορές κρίνει την επιλογή και το σχεδιασμό ενός αλγορίθμου. Για να εκφρασθεί αυτή η χειρότερη περίπτωση χρειάζεται κάποιο μέγεθος σύγκρισης και αναφοράς που να χαρακτηρίζει τον αλγόριθμο. Η πιο συνηθισμένη πρακτική μέτρηση είναι η μέτρηση του αριθμού των βασικών πράξεων που θα πρέπει να εκτελέσει ο αλγόριθμος στη χειρότερη περίπτωση. Για παράδειγμα μία βασική πράξη μπορεί να είναι η ανάθεση τιμής, η σύγκριση μεταξύ δυο μεταβλητών, ή οποιαδήποτε αριθμητική πράξη μεταξύ δυο μεταβλητών. Η χειρότερη περίπτωση αντιπροσωπεύει τις τιμές εκείνες, που όταν δίνονται ως είσοδος στον αλγόριθμο, οδηγούν στην εκτέλεση του μέγιστου αριθμού πράξεων.

Η **μέση περίπτωση** είναι η πιο αντιπροσωπευτική περίπτωση και είναι συχνά όσο κακή είναι και η χειρότερη. Στη μέση περίπτωση λαμβάνονται υπόψη όλες οι πιθανές είσοδοι και υπολογίζεται ο υπολογιστικός χρόνος για όλες τις εισόδους. Αθροίζονται όλες οι

τιμές και διαιρούνται με το πλήθος τους. Είναι δύσκολο να υπολογιστεί, αφού χρειάζεται να γίνουν υποθέσεις για την κατανομή των δεδομένων εισόδου.

Στην **καλύτερη περίπτωση** υπολογίζεται ένα κάτω φράγμα για το χρόνο εκτέλεσης ενός αλγορίθμου. Πρέπει να είναι γνωστή η υπόθεση που προκαλεί τον ελάχιστο αριθμό των λειτουργιών που θα εκτελεστούν. Περιγράφει το σενάριο στο οποίο ο αλγόριθμος παρουσιάζει την καλύτερη απόδοση. Αυτό το σενάριο δεν υλοποιείται συχνά.

2.1.2 Χωρική πολυπλοκότητα

Η χωρική πολυπλοκότητα ενός αλγορίθμου μελετάει πόση μνήμη χρειάζεται για να ολοκληρωθεί ο αλγόριθμος. Όπως και η χρονική πολυπλοκότητα, προσδιορίζεται ως συνάρτηση του μεγέθους των δεδομένων. Συμπεριλαμβάνει το χώρο των εντολών και το χώρο των δεδομένων. Ο χώρος των εντολών είναι το ποσό της μνήμης που απαιτούν οι εντολές του αλγορίθμου και εξαρτάται από το μεταγλωττιστή και το υλικό. Ο χώρος των δεδομένων είναι το ποσό της μνήμης που απαιτείται για την αποθήκευση των τιμών όλων των σταθερών και των μεταβλητών και για τη στοίβα περιβάλλοντος. Κάθε φορά που καλείται μια συνάρτηση στη στοίβα περιβάλλοντος αποθηκεύονται η διεύθυνση επιστροφής, οι τιμές των τοπικών μεταβλητών και των τυπικών παραμέτρων τιμών και η δέσμευση όλων των παραμέτρων αναφοράς. Ο χώρος που καταλαμβάνει ένα πρόγραμμα P συναρτήσει του μεγέθους του προβλήματος (n) είναι: $\mathbf{S(n) = c + Sp(n)}$

Όπου το c είναι το σταθερό μέρος και το Sp(n) είναι το μεταβλητό μέρος.

Η **χωρική πολυπλοκότητα χειρότερης περίπτωσης** ενός αλγόριθμου είναι η συνάρτηση S(n), η οποία είναι η μέγιστη για όλες τις εισόδους μεγέθους n, των αθροισμάτων χώρου μνήμης από κάθε βασική πράξη. Αν η συνάρτηση αυτή είναι εκθετική, τότε υπάρχει σοβαρό πρόβλημα ως προς την επίλυση.

Κεφάλαιο 3: Ασυμπτωτικοί συμβολισμοί

3.1 Ασυμπτωτική εκτίμηση

Η αποδοτικότητα ενός αλγορίθμου, χαρακτηρίζεται από τη συμπεριφορά του αλγορίθμου σε μεγάλα στιγμιότυπα, όπως προκύπτει από την ασυμπτωτική του εκτίμηση. Είναι σημαντικό όταν γίνεται ανάλυση ενός αλγορίθμου να προσδιορίζεται η τάξης μεγέθους του χρόνου εκτέλεσής του, σαν συνάρτηση του μεγέθους του στιγμιότυπου. Στην περίπτωση που το μέγεθος ενός στιγμιότυπου n γίνει αρκετά μεγάλο, οι τιμές της συνάρτησης από την οποία καθορίζεται η τάξη μεγέθους είναι αρκετά μεγαλύτερες από τους υπόλοιπους όρους. Για παράδειγμα, όταν ο όρος n γίνει αρκετά μεγάλος, τότε ο όρος 2^n είναι σημαντικά μεγαλύτερος από τον $1000n^2$ και ο όρος n^2 είναι σημαντικά μεγαλύτερος από τον $100n\log n$. Συνεπώς θεωρητικά, ένας αλγόριθμος ο οποίος έχει χρόνο εκτέλεσης μικρής τάξης μεγέθους προτιμάται από έναν αλγόριθμο ο οποίος έχει χρόνο εκτέλεσης μεγαλύτερης τάξης μεγέθους. Στην πράξη, ο πρώτος αλγόριθμος είναι πιο γρήγορος από τον δεύτερο μόνο στην περίπτωση που τα στιγμιότυπα που δίνονται προς επίλυση είναι αρκετά μεγάλα.

Κάποιες τάξεις μεγέθους συναντώνται αρκετά συχνά. Όταν το T(n) είναι ανεξάρτητο του n ονομάζουμε το χρόνο εκτέλεσης σταθερό. Όταν $T(n) = n$, n^2, ή n^3 ονομάζουμε το χρόνο εκτέλεσης γραμμικό, τετραγωνικό και κυβικό αντίστοιχα. Στην περίπτωση που το $T(n) = n^k$ για μια συγκεκριμένη σταθερά k, ονομάζουμε το χρόνο εκτέλεσης πολυωνυμικό. Όταν $T(n) = d^n$ για μια συγκεκριμένη σταθερά όπου $d > 1$, ονομάζουμε το χρόνο εκτέλεσης εκθετικό.

Εκτός από το χρόνο εκτέλεσης που είναι ο συνηθέστερος υπολογιστικός πόρος, η αποδοτικότητα ενός αλγορίθμου μπορεί να υπολογιστεί και από άλλους υπολογιστικούς πόρους. Τέτοιοι πόροι είναι τα μηνύματα δικτύου, η μνήμη, ο αριθμός επεξεργαστών κ.α.

Η ασυμπτωτική εκτίμηση χρησιμοποιείται γενικά για τις απαιτήσεις των αλγορίθμων όσον αφορά τους διάφορους υπολογιστικούς πόρους, καθώς και για να εξάγει συμπεράσματα σχετικά με το χρόνο εκτέλεσης τους. Η περίπτωση στην οποία ο χρόνος εκτέλεσης ενός αλγορίθμου θα μπορούσε να δίνεται από μια συνάρτηση του μεγέθους του στιγμιότυπου εισόδου n, θεωρείται ιδανική.

Συχνά ένας αλγόριθμος μπορεί να υπολογίζει διαφορετικούς χρόνους εκτέλεσης για διαφορετικά στιγμιότυπα του ίδιου μεγέθους. Υπάρχουν δηλαδή σημαντικές αποκλίσεις

στο χρόνο εκτέλεσης, οι οποίες εξαρτώνται εκτός από το μέγεθος και από τη δομή του στιγμιότυπου εισόδου.

Για παράδειγμα, όταν έχουμε έναν αλγόριθμο που εκτελεί γραμμική αναζήτηση σε έναν πίνακα με n στοιχεία, ο χρόνος εκτέλεσής του εξαρτάται από τη θέση στην οποία βρίσκεται το στοιχείο που αναζητούμε. Αν το στοιχείο που αναζητούμε βρίσκεται στις πρώτες θέσεις, ο χρόνος εκτέλεσης του αλγορίθμου είναι σταθερός, ενώ αν το στοιχείο βρίσκεται στις τελευταίες θέσεις ο χρόνος εκτέλεσης είναι γραμμικός. Αν το στοιχείο δεν υπάρχει στον πίνακα, ο χρόνος εκτέλεσης είναι επίσης γραμμικός. Συνεπώς, ο χρόνος εκτέλεσης του αλγόριθμου γραμμικής αναζήτησης είναι σταθερός στην καλύτερη περίπτωση και γραμμικός στη χειρότερη περίπτωση.

Σε αυτές τις περιπτώσεις, θέλουμε να διαπιστώσουμε ποια συμπεριφορά από τις δύο είναι πιο αντιπροσωπευτική για τον αλγόριθμο. Η πιο ασφαλής και απλή διεξαγωγή συμπερασμάτων γίνεται από τα στιγμιότυπα που μεγιστοποιούν το χρόνο εκτέλεσης του αλγορίθμου. Η εφαρμογή αυτής της μεθόδου, ονομάζεται ανάλυση χειρότερης περίπτωσης.

Η ανάλυση χειρότερης περίπτωσης παρέχει ένα άνω φράγμα στο χρόνο εκτέλεσης που ισχύει για κάθε στιγμιότυπο εισόδου και για το λόγο αυτό είναι ιδιαίτερα επιθυμητή. Όπως διαπιστώνεται και στην πράξη, οι χρόνοι εκτέλεσης των περισσότερων αριθμών δεν διαφοροποιούνται ιδιαίτερα από τα αποτελέσματα της ανάλυσης χειρότερης περίπτωσης. Υπάρχουν κάποιοι αλγόριθμοι στους οποίους σπανίζουν τα στιγμιότυπα εισόδου χειρότερης περίπτωσης, όπως ο αλγόριθμος αναζήτησης με παρεμβολή. Σε τέτοιους αλγόριθμους, προκειμένου να έχουμε μια ασφαλή και σαφή εικόνα για το χρόνο εκτέλεσής τους, εφαρμόζεται η ανάλυση της μέσης περίπτωσης.

3.2 Ασυμπτωτικός συμβολισμός

Στην ασυμπτωτική εκτίμηση λαμβάνεται υπόψη μόνο η τάξη μεγέθους του χρόνου εκτέλεσης του εκάστοτε αλγορίθμου και αγνοούνται οι σταθερές. Τα αποτελέσματα της ασυμπτωτικής εκτίμησης, στην ουσία εκφράζονται από τον ασυμπτωτικό συμβολισμό. Η ασυμπτωτική εκτίμηση του χρόνου εκτέλεσης ενός αλγορίθμου, κατηγοριοποιεί τους αλγόριθμους και ανάλογα με την αποτελεσματικότητά τους συγκρίνει μεταξύ τους. Η ασυμπτωτική εκτίμηση εξετάζει μόνο την τάξη μεγέθους, γιατί το μέγεθος της εισόδου n

μπορεί να γίνει αρκετά μεγάλο και τότε οι τιμές μιας συνάρτησης με μεγαλύτερη τάξη μεγέθους, όπως είναι η n^2, είναι σημαντικά μεγαλύτερες από τις τιμές μιας συνάρτησης μικρότερης τάξης μεγέθους, όπως είναι η nlogn. Συνεπώς για τα περισσότερα στιγμιότυπα ισχύει ότι ένας αλγόριθμος που είναι ασυμπτωτικά αποδοτικότερος, είναι αποδοτικότερος και στην πράξη.

Στην ασυμπτωτική εκτίμηση του χρόνου εκτέλεσης των αλγορίθμων, χρησιμοποιούνται αρκετά συχνά κάποιοι καθιερωμένοι τύποι ασυμπτωτικού συμβολισμού. Ο ασυμπτωτικός συμβολισμός, περιγράφει το χρόνο εκτέλεσης των αλγορίθμων και ορίζεται σε συναρτήσεις, οι οποίες έχουν πεδίο ορισμού το σύνολο των φυσικών αριθμών $\mathbb{N}$ = {0, 1, 2, …}. Ο συμβολισμός αυτού του είδους είναι κατάλληλος για την περιγραφή μιας συνάρτησης T(n), η οποία ορίζεται για ακέραια μόνο μεγέθη της εισόδου και δίνει το χρόνο εκτέλεσης ενός αλγορίθμου. Παρακάτω αναλύονται κάποιοι καθιερωμένοι ασυμπτωτικοί συμβολισμοί.

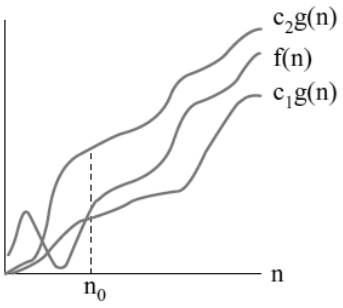
3.2.1 Συμβολισμός Θ

Χρησιμοποιούμε τον ασυμπτωτικό συμβολισμό Θ για να προσδιορίσουμε ακριβώς την τάξη μεγέθους μίας συνάρτησης f(n). Συγκεκριμένα, για κάθε συνάρτηση g(n), το σύνολο συναρτήσεων Θ(g(n)) περιλαμβάνει όλες τις συναρτήσεις που έχουν την ίδια ακριβώς τάξη μεγέθους με τη g(n). Με δεδομένη τη συνάρτηση g(n), συμβολίζουμε με Θ(g(n)) το σύνολο των συναρτήσεων:

Θ(g(n)) = {f(n) : υπάρχουν θετικές σταθερές c_1, c_2 και n_0 τέτοιες ώστε :

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ για κάθε $n \ge n_0$}

Με βάση αυτόν τον ορισμό, η συνάρτηση f(n) ανήκει στην κλάση συναρτήσεων Θ(g(n)), αν υπάρχουν θετικές σταθερές c_1, c_2 τέτοιες ώστε η f(n) να περιορίζεται μεταξύ $c_1 g(n)$ και $c_2 g(n)$ για αρκετά μεγάλες τιμές του n. Η γραφική αναπαράσταση φαίνεται στο σχήμα 3. Για όλες τις τιμές του n που είναι μεγαλύτερες του n_0, οι τιμές της f(n) βρίσκονται πάνω από το $c_1 g(n)$ και κάτω από το $c_2 g(n)$. Διαφορετικά, για κάθε $n \ge n_0$, η g(n) προσεγγίζει την f(n) κατά ένα σταθερό παράγοντα. Για το λόγο αυτό λέμε ότι η g(n) αποτελεί μια ασυμπτωτικά ακριβή εκτίμηση για την f(n).



f(n)=Θ(g(n))

Σχήμα 3: Γραφική παράσταση του ασυμπτωτικού συμβολισμού Θ

Ο ορισμός του Θ(g(n)) απαιτεί κάθε μέλος της κλάσης να είναι ασυμπτωτικά μη αρνητικό, δηλαδή η τιμή f(n) πρέπει να είναι μεγαλύτερη ή ίση του 0 για ένα αρκετά μεγάλο n. Συνεπώς, η ίδια συνάρτηση g(n) που χρησιμοποιείται για τον ορισμό της κλάσης πρέπει να είναι ασυμπτωτικά μη αρνητική. Στην αντίθετη περίπτωση το σύνολο Θ(g(n)) θα είναι κενό.

Κατά βάση ο συμβολισμός Θ(g(n)) δηλώνει κλάση συναρτήσεων, αλλά συνήθως γράφουμε f(n) = Θ(g(n)) για να δηλώσουμε ότι η f(n) είναι μέλος του συνόλου Θ(g(n)), δηλαδή f(n) $\in$ Θ(g(n)). Το ίδιο γίνεται και με τα υπόλοιπα είδη ασυμπτωτικού συμβολισμού.

Παραδείγματα

1. Αν $f(n) = n^6 + 3n$ τότε $n^6 + 3n \in \Theta(n^6)$ ή $f(n) = \Theta(n^6)$
2. Αν $f(n) = 2^n + 12$ τότε $2^n + 12 \in \Theta(2^n)$ ή $f(n) = \Theta(2^n)$
3. Αν $f(n) = 3^n + 2^n$ τότε $3^n + 2^n \in \Theta(3^n)$ ή $f(n) = \Theta(3^n)$
4. Αν $f(n) = n^n + n$ τότε $n^n + n \in \Theta(n^n)$ ή $f(n) = \Theta(n^n)$

Κατά τον προσδιορισμό μιας ασυμπτωτικά ακριβούς εκτίμησης για την f(n), μπορούν να αγνοηθούν οι όροι μικρότερης τάξης μεγέθους μιας ασυμπτωτικά μη αρνητικής συνάρτησης f(n). Όταν το n τείνει στο άπειρο, οι όροι μικρότερης τάξης μεγέθους θεωρούνται αμελητέοι σε σχέση με τον βασικό όρο. Έτσι μπορούμε να θέσουμε τη σταθερά c_1 σε τιμή μεγαλύτερη και τη σταθερά c_2 σε τιμή λίγο μικρότερη από το συντελεστή του βασικού όρου, ώστε οι ανισότητες του ορισμού του συμβολισμού Θ να

ικανοποιούνται για σημαντικά μεγάλες τιμές του n. Επομένως ο σταθερός συντελεστής του όρου που κυριαρχεί μπορεί να αγνοηθεί και μένει ο κυρίαρχος όρος, ο οποίος παρέχει μια ασυμπτωτικά ακριβή εκτίμηση για τη συνάρτηση.

Εφόσον κάθε σταθερά είναι ένα πολυώνυμο μηδενικού βαθμού, κάθε σταθερά μπορεί να εκφραστεί σαν $\Theta(n^0)$, ή απλούστερα σαν $\Theta(1)$. Ο συμβολισμός $\Theta(1)$ είναι καταχρηστικός γιατί δεν ξεκαθαρίζει τη μεταβλητή n. Τον χρησιμοποιούμε για να δηλώσουμε είτε μια σταθερά, είτε μια συνάρτηση η οποία έχει σταθερή τιμή.

3.2.2 Συμβολισμός O

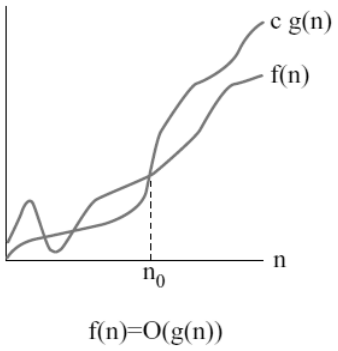
Ο συμβολισμός O χρησιμοποιείται στην περίπτωση που χρειαζόμαστε μόνο ένα ασυμπτωτικό άνω φράγμα στην τάξη μεγέθους μιας συνάρτησης. Δεδομένης μιας συνάρτησης g(n), συμβολίζουμε με O(g(n)) το σύνολο των συναρτήσεων:

$O(g(n)) = \{f(n) :$ υπάρχουν θετικές σταθερές c και n_0 τέτοιες ώστε :

$0 \leq f(n) \leq cg(n)$ για κάθε $n \geq n_0\}$

Η συνάρτηση g(n) πολλαπλασιασμένη με ένα σταθερό παράγοντα αποτελεί ένα ασυμπτωτικό άνω φράγμα για τη συνάρτηση f(n). Η γραφική αναπαράσταση φαίνεται στο σχήμα 4. Για όλα τα $n \geq n_0$ η συνάρτηση f(n) βρίσκεται κάτω από τη cg(n). Όπως και στην περίπτωση του συμβολισμού Θ, αν και το O(g(n)) ορίζει κλάση συναρτήσεων, γράφουμε f(n) = O(g(n)) για να δηλώσουμε ότι η f(n) είναι μέλος του συνόλου O(g(n)).

Από τη σύγκριση των ορισμών των συμβολισμών Θ και O, συμπεραίνουμε ότι κάθε συνάρτηση f(n) που ανήκει στο Θ(g(n)), ανήκει και στο O(g(n)) . Δηλαδή $\Theta(g(n)) \subseteq O(g(n))$.



Σχήμα 4: Γραφική παράσταση του ασυμπτωτικού συμβολισμού O

Όπως προκύπτει, το γεγονός ότι κάθε τετραγωνική συνάρτηση $f(n) = \alpha n^2 + \beta n + \gamma$, όπου $\alpha>0$, ανήκει στο $\Theta(n^2)$, σημαίνει επίσης ότι $f(n) = O(n^2)$. Επιπρόσθετα αν θεωρήσουμε μια οποιαδήποτε γραμμική συνάρτηση $f(n) = \alpha n + \beta$ όπου το $\alpha > 0$, θέτοντας τις σταθερές $c = \alpha + |\beta|$ και $n_0 = 1$, έχουμε ότι για κάθε $n \geq n_0$, $f(n) \leq c\, n^2$. Επομένως κάθε γραμμική συνάρτηση ανήκει στο $O(n^2)$.

Ο συμβολισμός Ο χρησιμοποιείται πολλές φορές για τη διατύπωση κατά προσέγγιση εκτιμήσεων σχετικά με το χρόνο εκτέλεσης ενός αλγορίθμου. Για παράδειγμα, ο παρακάτω αλγόριθμος περιέχει ένα διπλό φωλιασμένο βρόγχο:

```
for i ← 1 to n do
    for j ← i to n do
        <εντολή>
```

Αν το κόστος εκτέλεσης της <εντολής> είναι Ο(1), δηλαδή σταθερό, τότε το συνολικό κόστος για την εκτέλεση του βρόγχου είναι $O(n^2)$, γιατί οι δείκτες i και j παίρνουν τιμές από 1 μέχρι n και για κάθε τιμή του i η <εντολή> εκτελείται το πολύ n φορές.

Εφόσον ο ασυμπτωτικός συμβολισμός Ο παρέχει ένα άνω φράγμα, η περίπτωση που χρησιμοποιούμε τον Ο σαν φράγμα για το χρόνο εκτέλεσης της χειρότερης περίπτωσης ενός αλγορίθμου, μας δίνει ένα άνω φράγμα στο χρόνο εκτέλεσης για κάθε δεδομένη είσοδο. Όταν ο χρόνος εκτέλεσης χειρότερης περίπτωσης ενός αλγορίθμου είναι $O(n^2)$, σημαίνει ότι ο χρόνος εκτέλεσής του για κάθε είσοδο είναι $O(n^2)$. Αυτό δεν ισχύει για το συμβολισμό Θ, γιατί ακόμα και αν ο χρόνος εκτέλεσης χειρότερης περίπτωσης του αλγορίθμου είναι $\Theta(n^2)$, υπάρχουν στιγμιότυπα εισόδου για τα οποία ο χρόνος εκτέλεσής του είναι γραμμικός και επομένως δεν είναι $\Theta(n^2)$.

3.2.3 Συμβολισμός Ω

Σε αντίθεση με το συμβολισμό Ο, ο συμβολισμός Ω παρέχει ένα ασυμπτωτικό κάτω φράγμα στην τάξη μεγέθους μιας συνάρτησης. Δεδομένης μιας συνάρτησης g(n), συμβολίζουμε με Ω(g(n)) το σύνολο των συναρτήσεων

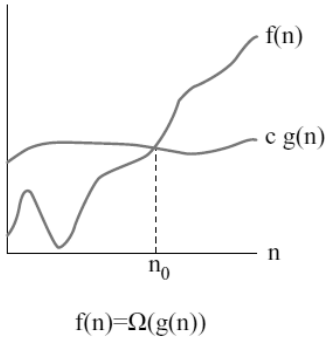
Ω(g(n)) = {f(n) : υπάρχουν θετικές σταθερές c και n_0 τέτοιες ώστε:

$0 \leq cg(n) \leq f(n)$ για κάθε $n \geq n_0$}

Η γραφική αναπαράσταση φαίνεται στο σχήμα 5. Χρησιμοποιώντας τους ορισμούς των ασυμπτωτικών συμβολισμών Θ, Ο και Ω μπορούμε να αποδείξουμε το ακόλουθο θεώρημα:

Για κάθε ζευγάρι συναρτήσεων f(n) και g(n), f(n) = Θ(g(n)) αν και μόνο αν

f(n) = O(g(n)) και f(n) = Ω(g(n))



Σχήμα 5: Γραφική παράσταση του ασυμπτωτικού συμβολισμού Ω

Για παράδειγμα εφαρμόζοντας το παραπάνω θεώρημα, στη σχέση $\alpha n^2 + \beta n + \gamma = \Theta(n^2)$ για κάθε τριάδα σταθερών α, β, γ και α > 0, συνάγουμε ότι $\alpha n^2 + \beta n + \gamma = O(n^2)$ και $\alpha n^2 + \beta n + \gamma = \Omega(n^2)$. Το θεώρημα αυτό χρησιμοποιείται συνήθως για να συνάγουμε ακριβείς εκτιμήσεις από τα άνω και κάτω φράγματα.

Δεδομένου ότι ο συμβολισμός Ω παρέχει ένα κάτω φράγμα για μια συνάρτηση, όταν τον χρησιμοποιούμε για να φράξουμε το χρόνο εκτέλεσης καλύτερης περίπτωσης ενός αλγορίθμου, το αποτέλεσμα αποτελεί ένα κάτω φράγμα στο χρόνο εκτέλεσης του αλγορίθμου για όλα τα στιγμιότυπα εισόδου. Δηλαδή το γεγονός ότι ο χρόνος εκτέλεσης καλύτερης περίπτωσης ενός αλγόριθμου είναι Ω(n) σημαίνει ότι ο χρόνος εκτέλεσης του αλγόριθμου είναι πάντα Ω(n) ή αλλιώς ότι ο αλγόριθμος δεν θα μπορέσει ποτέ να τερματίσει σε λιγότερο από το γραμμικό χρόνο.

3.2.4 Συμβολισμός ω

Ο ασυμπτωτικός συμβολισμός ω χρησιμοποιείται για τη δήλωση ενός κάτω φράγματος το οποίο δεν είναι ακριβές. Συγκεκριμένα δεδομένης μιας συνάρτησης g(n), συμβολίζουμε με ω(g(n)) το σύνολο των συναρτήσεων

ω(g(n)) = {f(n): για κάθε θετική σταθερά c, υπάρχει σταθερά $n_0 > 0$ τέτοια ώστε:

$0 \leq cg(n) < f(n)$ για κάθε $n \geq n_0$}

Ένας ισοδύναμος τρόπος να διατυπωθεί ο παραπάνω ορισμός είναι ότι η συνάρτηση f(n) = ω(g(n)) αν και μόνο αν g(n) = o(f(n)). Ο συμβολισμός f(n) = ω(g(n)) δηλώνει ότι $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, εφόσον το όριο υπάρχει. Δηλαδή η συνάρτηση f(n) γίνεται όσο μπορεί μεγαλύτερη σε σύγκριση με τη g(n), καθώς το n τείνει στο άπειρο. Για παράδειγμα $2n^2 =$ ω(n) αλλά $2n^2 \neq \omega(n^2)$.

3.2.5 Συμβολισμός ο

Ο ασυμπτωτικός συμβολισμός ο δηλώνει ένα άνω φράγμα το οποίο δεν είναι ακριβές. Το ασυμπτωτικό άνω φράγμα που παρέχεται από το συμβολισμό Ο κάποιες φορές είναι ακριβές και κάποιες άλλες όχι. Για παράδειγμα το φράγμα $2n^2 = O(n^2)$ είναι ακριβές, ενώ το φράγμα $2n = O(n^2)$ δεν είναι. Συγκεκριμένα δεδομένης μιας συνάρτησης g(n), συμβολίζουμε με ο(g(n)) το σύνολο των συναρτήσεων

ο(g(n)) = {f(n) : για κάθε θετική σταθερά c, υπάρχει $n_0 > 0$ τέτοια ώστε:

$0 \leq f(n) < cg(n)$ για κάθε $n \geq n_0$}

Υπάρχει ομοιότητα στους ορισμούς των συμβολισμών Ο και ο. Στην ουσία η διαφορά τους είναι ότι στον ορισμό του Ο ισχύει $0 \leq f(n) \leq cg(n)$ για κάθε σταθερά $c > 0$, ενώ στον ορισμό του ο ισχύει $0 \leq f(n) < cg(n)$ για κάθε σταθερά $c > 0$. Αυτό σημαίνει ότι στην περίπτωση του ο η συνάρτηση f(n) γίνεται μικρότερη από τη g(n) κατά μια οσοδήποτε μεγάλη σταθερά καθώς το n τείνει στο άπειρο. Σε διαφορετική μαθηματική διατύπωση αυτό γράφεται: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Από τα παραπάνω συνάγεται ότι $2n = o(n^2)$, αλλά $2n^2 \neq o(n^2)$.

3.2.6 Παραδείγματα ασυμπτωτικών συμβολισμών

Πίνακας 2: Ο παρακάτω πίνακας δείχνει τις περιπτώσεις που η συνάρτηση f(n) ανήκει στις κλάσεις Θ(g(n)), O(g(n)), Ω(g(n)), ω(g(n)), o(g(n)).

$f(n)$	g(n)	Θ(g(n))	O(g(n))	Ω(g(n))	ω(g(n))	o(g(n))
2^{n+5}	$2^n + 2^5 + n^{100}$	ναι	ναι	ναι	όχι	όχι
$n^4 - n^3$	$16^{\log n}$	ναι	ναι	ναι	όχι	όχι
5^{4n}	10^{2n}	όχι	όχι	ναι	ναι	όχι
$n^{1/\log \log n}$	$n^{0.001}$	όχι	ναι	όχι	όχι	ναι
n!	n^n	όχι	ναι	όχι	όχι	ναι
$n^{\log 20_n}$	2^n	όχι	ναι	όχι	όχι	ναι

Παραδείγματα

- Ένας Θ(n) αλγόριθμος είναι O(n). Αφού το αρχικό μας πρόγραμμα ήταν Θ(n) μπορούμε να πετύχουμε O(n) χωρίς καμία αλλαγή στο πρόγραμμά μας.
- Ένας Θ(n) αλγόριθμος είναι O(n^2), αφού το n^2 είναι χειρότερο από το n.
- Ένας Θ(n^2) αλγόριθμος είναι O(n^3), αφού το n^3 είναι χειρότερο από το n^2.
- Ένας Θ(n) αλγόριθμος δε μπορεί να είναι O(1), αφού το 1 δεν είναι χειρότερο από το n. Αν ένα πρόγραμμα τρέχει ασυμπτωτικά n εντολές (ένα γραμμικό πλήθος εντολών), δεν μπορούμε να το κάνουμε χειρότερο και μετά να τρέχει ασυμπτωτικά μόνο 1 εντολή (ένα σταθερό πλήθος από εντολές).
- Ένας O(1) αλγόριθμος είναι Θ(1), αφού οι δύο πολυπλοκότητες είναι ίδιες.
- Ένας O(n) αλγόριθμος είναι Θ(1). Αυτό μπορεί να ισχύει ή μπορεί και να μην ισχύει ανάλογα με τον αλγόριθμο. Στη γενική περίπτωση δεν ισχύει. Αν ένας αλγόριθμος είναι Θ(1), τότε είναι σίγουρα O(n). Όμως αν είναι O(n) τότε μπορεί να μην είναι Θ(1). Για παράδειγμα, ένας Θ(n) αλγόριθμος είναι O(n) αλλά όχι Θ(1).

Παραδείγματα σε ακριβή φράγματα

- Σε έναν αλγόριθμο με πολυπλοκότητα Θ(n) για τον οποίο βρήκαμε ένα άνω φράγμα O(n), η Θ πολυπλοκότητα και η O πολυπλοκότητα είναι οι ίδιες, οπότε το φράγμα είναι ακριβές.
- Σε έναν αλγόριθμο με πολυπλοκότητα $\Theta(n^2)$ για τον οποίο βρήκαμε ένα άνω φράγμα $O(n^3)$, η O πολυπλοκότητα είναι μεγαλύτερης κλίμακας από τη Θ πολυπλοκότητα οπότε αυτό το φράγμα δεν είναι ακριβές. Ένα φράγμα $O(n^2)$ θα ήταν ακριβές. Οπότε μπορούμε να γράψουμε ότι ο αλγόριθμος είναι $o(n^3)$.
- Σε έναν αλγόριθμο με πολυπλοκότητα Θ(1) για τον οποίο βρήκαμε ένα άνω φράγμα O(n), η O πολυπλοκότητα είναι μεγαλύτερης κλίμακας από τη Θ πολυπλοκότητα οπότε έχουμε ένα μη ακριβές φράγμα. Το φράγμα O(1) θα ήταν ακριβές. Οπότε μπορούμε να επισημάνουμε ότι το O(n) φράγμα δεν είναι ακριβές γράφοντάς το ως o(n).
- Σε έναν αλγόριθμο με πολυπλοκότητα Θ(n) για τον οποίο βρήκαμε ένα άνω φράγμα O(1), έχει γίνει σφάλμα στον υπολογισμό του φράγματος. Είναι αδύνατο ένας Θ(n) αλγόριθμος να έχει άνω φράγμα O(1), αφού το n είναι μεγαλύτερη πολυπλοκότητα από το 1. Το O δίνει άνω φράγμα.
- Σε έναν αλγόριθμο με πολυπλοκότητα Θ(n) για τον οποίο βρήκαμε ένα άνω φράγμα O(2n), ισχύει ότι O(2n) = O(n) και συνεπώς αυτό το φράγμα είναι ακριβές αφού η πολυπλοκότητα είναι η ίδια με το Θ. Η ασυμπτωτική συμπεριφορά των 2n και n είναι η ίδια.

Κεφάλαιο 4: Αναδρομικοί αλγόριθμοι, αλγόριθμοι διαίρει και βασίλευε και άπληστοι αλγόριθμοι

4.1 Αναδρομικοί αλγόριθμοι

Αναδρομικός αλγόριθμος είναι ένας αλγόριθμος που λύνει ένα πρόβλημα λύνοντας ένα ή περισσότερα στιγμιότυπα του ίδιου προβλήματος. Για την υλοποίηση αναδρομικών αλγορίθμων πολλές γλώσσες προγραμματισμού χρησιμοποιούν αναδρομικές συναρτήσεις ή αναδρομικές μεθόδους. Δηλαδή μέθοδοι ή συναρτήσεις που καλούν τον εαυτό τους. Οι αναδρομικές συναρτήσεις αντιστοιχούν σε αναδρομικούς ορισμούς των μαθηματικών συναρτήσεων.

Οι αναδρομικές σχέσεις είναι αναδρομικώς ορισμένες συναρτήσεις. Μια αναδρομική σχέση ορίζει μια συνάρτηση της οποίας το πεδίο ορισμού είναι οι μη αρνητικοί ακέραιοι, είτε με κάποιες αρχικές τιμές είτε ως συνάρτηση των δικών της τιμών για μικρότερους ακεραίους. Η γνωστότερη κατά πάσα πιθανότητα συνάρτηση αυτού του είδους είναι η παραγοντική συνάρτηση ή συνάρτηση παραγοντικού η οποία ορίζεται από την αναδρομική σχέση $N! = N * (N - 1)!$, για $N \geq 1$ με $0! = 1$

Συνάρτηση παραγοντικού σε java

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

Αυτή η αναδρομική μέθοδος υπολογίζει τη συνάρτηση N! χρησιμοποιώντας τον καθιερωμένο αναδρομικό ορισμό. Επιστρέφει τη σωστή τιμή όταν καλείται με θετικό N και αρκετά μικρό ώστε το N! να μπορεί να αναπαρασταθεί ως ακέραιος τύπου int.

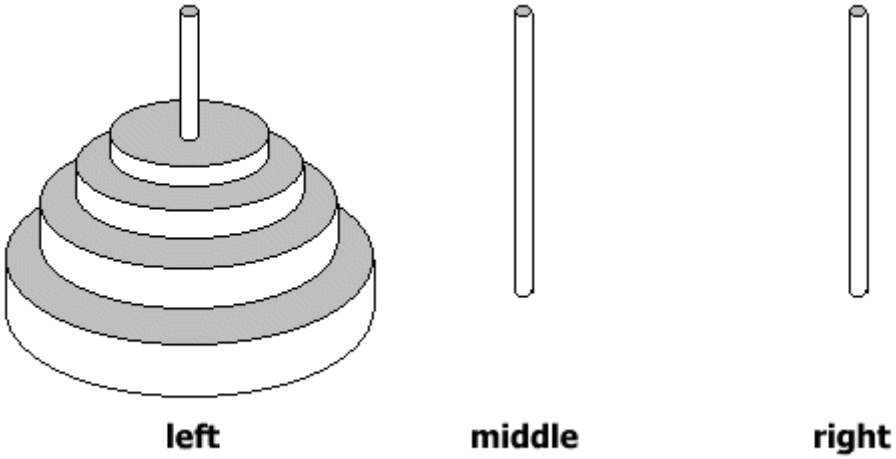
4.1.1 Ανάλυση πολυπλοκότητας της συνάρτησης παραγοντικού

Αυτή η συνάρτηση δεν έχει καθόλου βρόγχους, αλλά η πολυπλοκότητά της δεν είναι σταθερή. Αυτό που πρέπει να κάνουμε για να υπολογίσουμε την πολυπλοκότητά της είναι να αρχίσουμε να μετράμε εντολές. Είναι εμφανές ότι, αν δώσουμε κάποιο n σε αυτή τη συνάρτηση, θα εκτελέσει τον εαυτό της n φορές. Αυτό μπορούμε να το διαπιστώσουμε και αν τρέξουμε τη συνάρτηση για ένα συγκεκριμένο n. Για παράδειγμα, για n = 5, θα τρέξει 5 φορές, καθώς θα μειώνει συνεχώς το n κατά 1 σε κάθε κλήση. Μπορούμε λοιπόν να δούμε ότι αυτή η συνάρτηση είναι Θ(n). Επομένως η συνάρτηση έχει γραμμική πολυπλοκότητα O(n).

Χρησιμοποιούμε αναδρομή επειδή συχνά μας επιτρέπει να εκφράζουμε πολύπλοκους αλγορίθμους σε συμπαγή μορφή, χωρίς να χάνεται η αποδοτικότητα. Για παράδειγμα η αναδρομική υλοποίηση της συνάρτησης παραγοντικού κάνει φανερή την ανάγκη για τοπικές μεταβλητές. Το κόστος της αναδρομικής υλοποίησης προέρχεται από τους μηχανισμούς των συστημάτων προγραμματισμού που υποστηρίζουν κλήσεις μεθόδων, οι οποίες χρησιμοποιούν ένα ισοδύναμο ενσωματωμένης στοίβας ώθησης προς τα κάτω. Τα περισσότερα σύγχρονα συστήματα προγραμματισμού διαθέτουν προσεκτικά υλοποιημένους μηχανισμούς για την εκτέλεση αυτής της εργασίας.

4.1.2 Οι πύργοι του Hanoi

Ένα από τα πιο κλασικά παραδείγματα χρήσης αναδρομικών συναρτήσεων είναι οι πύργοι του Hanoi. Ο θρύλος λέει ότι σε έναν ναό στην Άπω Ανατολή, οι ιερείς προσπαθούν να μεταφέρουν μια στοίβα χρυσών δίσκων από έναν στύλο σε έναν άλλο. Η αρχική στοίβα έχει 64 δίσκους τοποθετημένους σε έναν στύλο και τακτοποιημένους από κάτω προς τα πάνω σε φθίνουσα σειρά μεγέθους. Οι ιερείς προσπαθούν να μεταφέρουν την στοίβα από τον ένα στύλο στον άλλο, υπό την προϋπόθεση ότι μόνο ένας δίσκος μεταφέρεται κάθε φορά και σε καμία περίπτωση δεν μπορεί ένας μεγαλύτερος δίσκος να τοποθετηθεί πάνω από έναν μικρότερο δίσκο. Είναι διαθέσιμοι τρεις στύλοι, ένας εκ των οποίων θα χρησιμοποιηθεί για την προσωρινή τοποθέτηση των δίσκων. Υποθετικά, ο κόσμος θα τελειώσει όταν οι ιερείς ολοκληρώσουν το έργο τους.



Σχήμα 6: Οι πύργοι του Hanoi

Οι ιερείς προσπαθούν να μεταφέρουν τους δίσκους από τον αριστερό στύλο στον δεξιό. Για την ανάπτυξη ενός αλγορίθμου που εμφανίζει την ακριβή ακολουθία κινήσεων της μεταφοράς των δίσκων από στύλο σε στύλο, χρησιμοποιείται αναδρομή. Η μεταφορά n δίσκων μπορεί να εξεταστεί ως μεταφορά μόνο n-1 δίσκων ως εξής:

- Μεταφέρονται n-1 δίσκοι από τον αριστερό στο μεσαίο στύλο, χρησιμοποιώντας τον δεξί στύλο ως προσωρινή περιοχή.
- Ο τελευταίος και μεγαλύτερος δίσκος μεταφέρεται από τον αριστερό στον δεξί στύλο.
- Οι n-1 δίσκοι μεταφέρονται από τον μεσαίο στύλο στον δεξιό, χρησιμοποιώντας τον αριστερό ως προσωρινή περιοχή.

Η διαδικασία τελειώνει όταν η τελευταία εργασία καλέσει τη μεταφορά του n=1 δίσκου. Αυτή η εργασία επιτυγχάνεται μεταφέροντας απλώς τον δίσκο, χωρίς την ανάγκη προσωρινής περιοχής.

Λύση της συνάρτησης σε java με αναδρομική μέθοδο

```
public class TowersOfHanoi
{
   // μεταφέρει με αναδρομή δίσκους μέσω των πύργων
   public static void solveTowers( int disks, int sourcePeg,
      int destinationPeg, int tempPeg )
```

```
{
   // βασική περίπτωση -- μόνο ένας δίσκος για μεταφορά
   if ( disks == 1 )
   {
      System.out.printf( "\n%d --> %d", sourcePeg,
        destinationPeg );
      return;
   } // τέλος της if

   // βήμα της αναδρομής–μεταφέρει (disk-1) δίσκους από το
   //sourcePeg στο tempPeg χρησιμοποιώντας το destinationPeg
   solveTowers( disks - 1, sourcePeg, tempPeg, destinationPeg
   );

   // μεταφέρει τον τελευταίο δίσκο από το sourcePeg
   // στο destinationPeg
   System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg
   );

   // μεταφέρει ( disks - 1 ) δίσκους από το tempPeg
   // στο destinationPeg
   solveTowers( disks - 1, tempPeg, destinationPeg, sourcePeg
   );
} // τέλος μεθόδου solveTowers

public static void main( String[] args )
{
   int startPeg = 1; // η τιμή 1 χρησιμοποιείται για να
   //υποδείξει το startPeg για έξοδο
   int endPeg = 3; // η τιμή 3 χρησιμοποιείται για να
   //υποδείξει το endPeg για έξοδο
   int tempPeg = 2; // η τιμή 2 χρησιμοποιείται για να
   //υποδείξει το tempPeg για έξοδο
   int totalDisks = 3; // πλήθος δίσκων

   // αρχική μη αναδρομική κλήση: μετακίνηση όλων των
   // δίσκων.
```

```
      solveTowers( totalDisks, startPeg, endPeg, tempPeg );
   } // τέλος της main
} // τέλος της κλάσης TowersOfHanoi
```

Η μέθοδος solveTowers λύνει το πρόβλημα των πύργων του Ανόι με δεδομένο το συνολικό πλήθος των δίσκων, τον αρχικό στύλο, τον τελικό στύλο και τον προσωρινό στύλο ως παραμέτρους. Η βασική περίπτωση συμβαίνει μόνο όταν ένας δίσκος χρειάζεται να μεταφερθεί από τον αρχικό στύλο στον τελικό στύλο. Στο βήμα αναδρομής μεταφέρονται disks-1 δίσκοι από τον πρώτο στύλο(sourcePeg) στον προσωρινό στύλο(tempPeg). Όταν μεταφερθούν όλοι οι δίσκοι εκτός από έναν στον προσωρινό στύλο, καταγράφεται το βήμα που μεταφέρει τον μεγαλύτερο δίσκο από τον αρχικό στύλο στον στύλο προορισμού. Με την κλήση της μεθόδου solveTowers που μεταφέρονται με αναδρομή disks-1 δίσκοι από τον προσωρινό στον στύλο προορισμού(destinationPeg), χρησιμοποιώντας τον πρώτο στύλο ως προσωρινό, τελειώνουν οι υπόλοιπες κινήσεις. Τέλος καλείται στη main η αναδρομική μέθοδος solveTowers, η οποία εξάγει τα βήματα στην προτροπή εντολών.

Η πολυπλοκότητα του αλγορίθμου υπολογίζεται σχετικά εύκολα λόγω της αναδρομικής δομής του. Ο αλγόριθμος επίλυσης του προβλήματος παράγει λύση η οποία δημιουργεί 2^N-1 κινήσεις.

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

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

Οπότε με επαγωγή έχουμε

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

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

$T_N = 2(2^{N-1} - 1) + 1 = 2 * 2^{N-1} - 2 + 1 = 2^N - 1$

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

4.2 Διαίρει και βασίλευε

Η μέθοδος σχεδιασμού αλγορίθμων διαίρει και βασίλευε εφαρμόζεται σε μεγάλο πλήθος προβλημάτων από διάφορα πεδία και είναι αρκετά απλή και πολύ αποτελεσματική. Η φύση της μεθόδου διαίρει και βασίλευε είναι αναδρομική. Η εφαρμογή της για την επίλυση ενός υπολογιστικού προβλήματος συνίσταται σε τρία βασικά βήματα:

- Στη διαίρεση του στιγμιότυπου εισόδου σε δύο ή και περισσότερα μικρότερα στιγμιότυπα.
- Στην επίλυση των μικρότερων στιγμιότυπων που δημιουργήθηκαν, με αναδρομική εφαρμογή του ίδιου αλγόριθμου.
- Στη σύνθεση της λύσης του αρχικού στιγμιότυπου από τις λύσεις των μικρότερων στιγμιότυπων.

Η εφαρμογή της μεθόδου διαίρει και βασίλευε γίνεται από την κορυφή προς τη βάση. Τα μεγάλα στιγμιότυπα διαιρούνται συνεχώς σε μικρότερα μέχρι να γίνουν στοιχειώδη. Όταν το μέγεθος των υπο-στιγμιότυπων γίνει αρκετά μικρό, η αναδρομή τερματίζεται τα υπο-στιγμιότυπα επιλύονται με τη χρήση κάποιου μη αναδρομικού αλγόριθμου.

Κάποια συνηθισμένα παραδείγματα εφαρμογής της μεθόδου διαίρει και βασίλευε είναι ο αλγόριθμος ταξινόμησης με συγχώνευση(merge sort), ο αλγόριθμος ταχείας ταξινόμησης(quicksort) και ο αλγόριθμος επιλογής σε γραμμικό χρόνο που βασίζεται στην ίδια διαδικασία διαίρεσης με την ταχεία ταξινόμηση.

Για την επιτυχημένη εφαρμογή της μεθόδου διαίρει και βασίλευε υπάρχουν κάποιες προϋποθέσεις:

- Τα στιγμιότυπα που διαιρούνται να επιλύονται ευκολότερα από το αρχικό στιγμιότυπο. Για παράδειγμα, στον αλγόριθμο της ταξινόμησης με συγχώνευση η διαίρεση είναι κοινότυπη, αφού ο αρχικός πίνακας διαιρείται σε δύο υποπίνακες ίσου μεγέθους. Στην ταχεία ταξινόμηση η διαίρεση γίνεται εύκολα εξετάζοντας κάθε στοιχείο του πίνακα μια φορά.
- Τα υπο-στιγμιότυπα που παράγονται από τη διαίρεση να είναι σημαντικά μικρότερα από το αρχικό στιγμιότυπο και συνεπώς να επιδέχονται και ευκολότερη επίλυση. Για παράδειγμα στους αλγόριθμους ταξινόμησης με

συγχώνευση και ταχείας ταξινόμησης, η ταξινόμηση δυο συνόλων n/2 στοιχείων είναι σημαντικά ευκολότερη από την ταξινόμηση ενός συνόλου n στοιχείων.

- Η διαίρεση να μην οδηγεί σε στιγμιότυπα που είναι ίδια μεταξύ τους ή που επικαλύπτονται. Επειδή κάθε στιγμιότυπο λύνεται από μια ανεξάρτητη αναδρομική κλήση, η επανάληψη υπο-στιγμιότυπων οδηγεί σε σπατάλη υπολογιστικών πόρων. Για παράδειγμα στους αλγόριθμους ταξινόμησης με συγχώνευση και ταχείας ταξινόμησης, από τη διαίρεση προκύπτουν ξένοι μεταξύ τους υποπίνακες που ταξινομούνται ανεξάρτητα.
- Η σύνθεση της λύσης για το αρχικό στιγμιότυπο από τις λύσεις των μικρότερων στιγμιότυπων να είναι σημαντικά ευκολότερη από την επίλυση του αρχικού στιγμιότυπου. Στην ταχεία ταξινόμηση για παράδειγμα, η διαδικασία της διαίρεσης καθιστά τη διαδικασία της σύνθεσης τετριμμένη. Στην ταξινόμηση με συγχώνευση, δύο ταξινομημένοι υποπίνακες μπορούν να συγχωνευτούν εύκολα σε έναν ταξινομημένο πίνακα σε γραμμικό χρόνο.

Η μέθοδος διαίρει και βασίλευε, εκτός του ότι είναι απλή στη σύλληψη και την εφαρμογή της, καταλήγει σε αναδρομικούς αλγόριθμους οι οποίοι μπορούν να αναλυθούν εύκολα. Για να γίνει ανάλυση ενός αλγόριθμου διαίρει και βασίλευε πρέπει να διατυπωθεί και στη συνέχεια να επιλυθεί η αναδρομική εξίσωση που χρησιμοποιείται για την εκτέλεσή του. Συνήθως η αναδρομική εξίσωση προκύπτει από την περιγραφή του αλγορίθμου. Μέσω των μαθηματικών εργαλείων επιτυγχάνεται εύκολα και η επίλυση της αναδρομικής εξίσωσης.

4.2.1 Αλγόριθμος Merge-sort

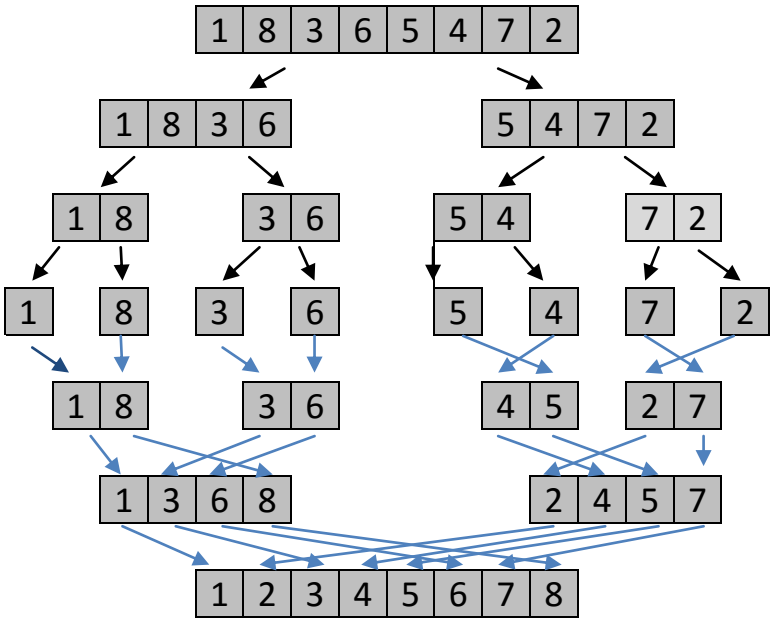
Ο αλγόριθμος merge sort είναι ένας αναδρομικός αλγόριθμος που αποσκοπεί στη διάταξη ενός συγκεκριμένου πλήθους αριθμών. Για να διατάξει έναν πίνακα με πλήθος αριθμών n, ο αλγόριθμος merge sort διαιρεί τον αρχικό πίνακα σε δύο υποπίνακες που ο καθένας έχει n/2 στοιχεία. Ο αλγόριθμος καλεί αναδρομικά τον εαυτό του και έτσι οι δύο υποπίνακες διατάσσονται και συνενώνονται σε έναν πίνακα του οποίου τα στοιχεία είναι διατεταγμένα σε αύξουσα σειρά.

Ψευδοκώδικας του αλγόριθμου:

```
mergeSort(int A[], int left, int right) {
      if (left >= right) return;
      mid = (left + right) / 2;
      mergeSort(A, left, mid);
      mergeSort(A, mid+1, right);
      merge(A, left, mid, right); }
```

Στον κώδικα της mergeSort, χρησιμοποιούμε τη `merge(A, left, mid, right)`, όπου A είναι ένας πίνακας και left ≤ mid ≤ right είναι οι δείκτες σε θέσεις του πίνακα A που ορίζουν τους υποπίνακες που θα συνενωθούν. Η merge πραγματοποιεί τη συνένωση των διατεταγμένων υποπινάκων A[left, …,mid] και A[mid + 1, …, right] σε έναν διατεταγμένο υποπίνακα A[left, …, right].

Ένα απλό παράδειγμα λειτουργίας του αλγορίθμου mergeSort είναι το παρακάτω:



Σχήμα 7: Παράδειγμα εφαρμογής αλγόριθμου merge sort

Ο πίνακας εισόδου είναι ο A = [1, 8, 3, 6, 5, 4, 7, 2] και διαιρείται από τις αναδρομικές κλήσεις της συνάρτησης mergeSort έως ότου οι υποπίνακες να περιέχουν μόνο ένα στοιχείο. Η ανάπτυξη της αναδρομής σταματά όταν κάθε υποπίνακας περιέχει μόνο ένα στοιχείο. Μετά από το σημείο αυτό, οι υποπίνακες συνενώνονται διαδοχικά και φτιάχνουν τον διατεταγμένο κατά αύξουσα σειρά πίνακα A = [1, 2, 3, 4, 5, 6, 7, 8]. Στο σχήμα 7 διακρίνονται οι πίνακες με τους οποίους γίνονται οι αναδρομικές κλήσεις της mergeSort και στο κάτω μέρος οι διατεταγμένοι πίνακες που προκύπτουν κατά την επιστροφή της αναδρομής από την εφαρμογή της merge.

Υπάρχει ορθότητα στη merge επειδή τα τμήματα είναι ταξινομημένα και όταν ένα στοιχείο μεταφέρεται στον A[ ], δεν υπάρχει μικρότερο διαθέσιμο στοιχείο στα δύο τμήματα. Η ορθότητα της mergeSort αποδεικνύεται επαγωγικά, καθώς η βάση είναι τετριμμένη και τα δύο τμήματα σωστά ταξινομημένα και συγχωνεύονται σωστά.

Έστω ότι ο χρόνος εκτέλεσης χειρότερης περίπτωσης που θα χρειαστεί ο αλγόριθμος mergeSort για να διατάξει n αριθμούς, συμβολίζεται με T(n). Αυτός ο χρόνος ισούται με το άθροισμα του χρόνου 2T(n/2), που χρειάζεται η mergeSort για τη διάταξη δύο υποπινάκων n/2 αριθμών, και του χρόνου που χρειάζεται η merge για να συνενώσει τους δύο διατεταγμένους υποπίνακες σε έναν διατεταγμένο πίνακα n στοιχείων. Δεδομένου ότι ο χρόνος εκτέλεσης χειρότερης περίπτωσης της διαδικασίας merge είναι Θ(n), προκύπτει ότι T(n) = 2T(n/2) + Θ(n). Στην περίπτωση που το n = 1, η mergeSort χρειάζεται σταθερό χρόνο, δηλαδή T(1) = Θ(1).

Συνεπώς, ο χρόνος εκτέλεσης χειρότερης περίπτωσης της mergeSort περιγράφεται από την αναδρομική εξίσωση:

$$\mathrm{T(n)} = \begin{cases} \Theta(1) & \alpha\nu\ n = 1 \\ 2T(n/2) + \Theta(n) & \alpha\nu\ n > 1 \end{cases}$$

Περιγράφοντας το χρόνο εκτέλεσης με μια αναδρομική εξίσωση, ολοκληρώνεται ο σχεδιασμός και η ανάλυση του αλγορίθμου, που είναι το σημαντικότερο κομμάτι της διαδικασίας, η οποία ολοκληρώνεται επιλύοντας την αναδρομική εξίσωση με μαθηματικό τρόπο.

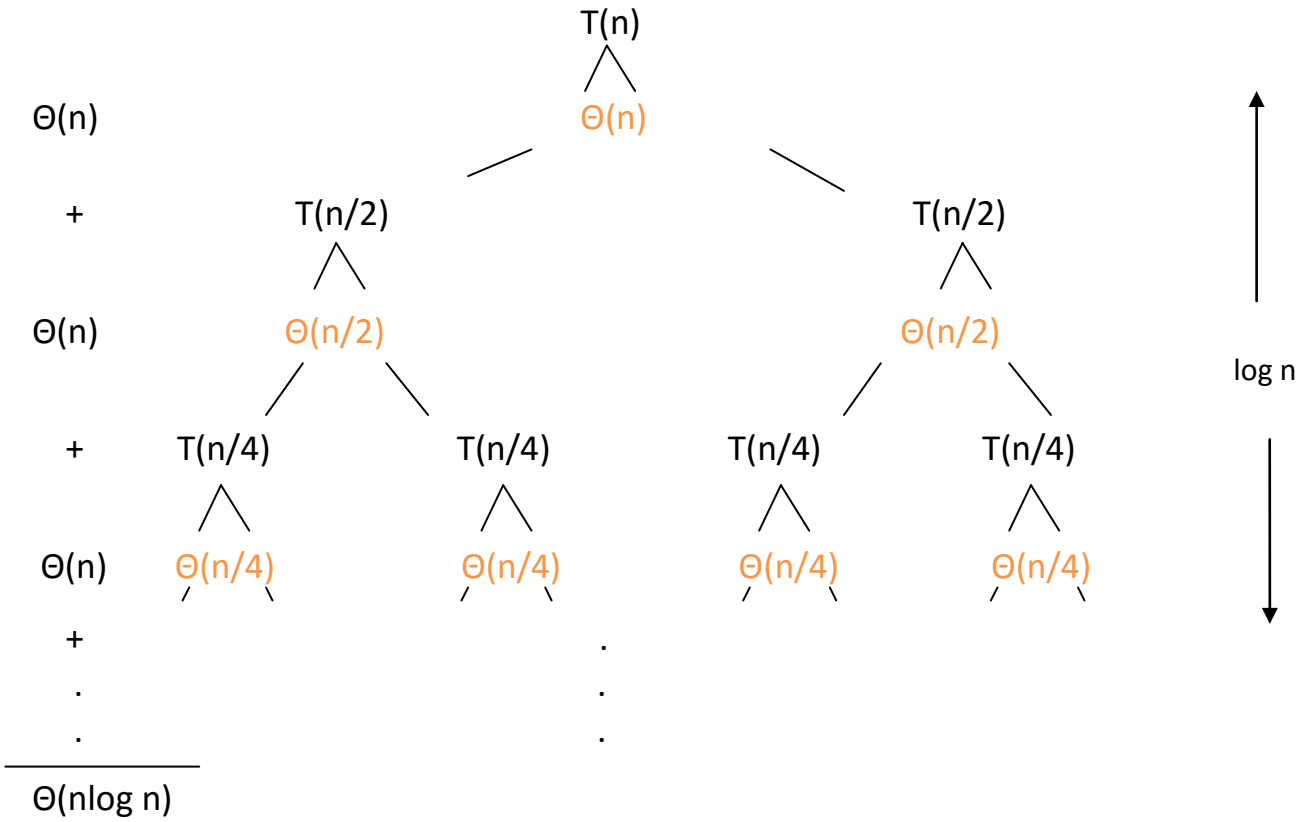
4.2.2 Μέθοδος επανάληψης

Η μέθοδος επανάληψης έχει σαν βασική ιδέα την ανάπτυξη της αναδρομικής εξίσωσης, παρουσιάζοντάς την σαν άθροισμα και τον υπολογισμό του κλειστού τύπου του αθροίσματος. Παρά την απλότητα της μεθόδου, η εφαρμογή της συνήθως οδηγεί σε πολύπλοκους μαθηματικούς υπολογισμούς.

Μία σημαντική παράμετρος για την εφαρμογή της μεθόδου είναι ο αριθμός των επαναλήψεων που κάνει η αναδρομή μέχρι να καταλήξει στην αρχική συνθήκη. Υπάρχει επίσης και μία ακόμη παράμετρος που αφορά τον υπολογισμό του αθροίσματος των όρων που προκύπτουν από κάθε επίπεδο στο οποίο αναπτύσσεται η αναδρομή. Όταν η αναδρομική εξίσωση ορίζεται με πάνω ή κάτω ακέραια μέρη κλασμάτων, η εφαρμογή της μεθόδου της επανάληψης μπορεί να οδηγήσει σε πολύπλοκες αλγεβρικές εκφράσεις. Σε αυτή την περίπτωση είναι ιδιαίτερα χρήσιμη η υπόθεση ότι το n έχει τέτοια μορφή ώστε το κλάσμα να δίνει πάντα ακέραιο αποτέλεσμα. Για παράδειγμα στην εξίσωση $T(n) = 2T(n/2) + n$, μπορούμε να υποθέσουμε ότι σε κάθε βήμα το n είναι δύναμη του 2, ώστε το $n/2$ να είναι ακέραιος.

4.2.2.1 Δέντρο αναδρομής

Για την αναπαράσταση της ανάπτυξης μιας αναδρομικής εξίσωσης, μπορούμε να χρησιμοποιήσουμε μία μέθοδο που είναι χρήσιμη και απλή στην εφαρμογή της. Η μέθοδος αυτή ονομάζεται δέντρο αναδρομής και οργανώνει καλύτερα τους αλγεβρικούς υπολογισμούς στην ανάπτυξη μιας αναδρομικής εξίσωσης. Για την αναδρομική εξίσωση $T(n) = 2T(n/2) + \Theta(n)$, το δέντρο της αναδρομής φαίνεται στο Σχήμα 8. Επειδή σε κάθε επίπεδο το n υποδιπλασιάζεται, το ύψος του δέντρου είναι $\Theta(\log n)$, δηλαδή το δέντρο έχει $\log n + 1$ επίπεδα. Ο συνολικός χρόνος εκτέλεσης θα είναι $\Theta(n \log n)$.



Σχήμα 8: Δέντρο αναδρομής

4.2.3 Μέθοδος αντικατάστασης

Η μέθοδος αντικατάστασης έχει σαν βασική ιδέα την πρόβλεψη της μορφής λύσης της αναδρομικής εξίσωσης και τη χρήση της μαθηματικής επαγωγής για τον υπολογισμό των σταθερών με ακρίβεια, έτσι ώστε να αποδειχτεί η ορθότητα της λύσης. Η μέθοδος αυτή βρίσκει εφαρμογή μόνο στις περιπτώσεις κατά τις οποίες η μορφή της λύσης μπορεί να προβλεφθεί. Επιπρόσθετα, η μέθοδος της αντικατάστασης μπορεί να εφαρμοστεί για την απόδειξη ασυμπτωτικών εκτιμήσεων όσον αφορά τη λύση αναδρομικών εξισώσεων.

Σε κάποιες περιπτώσεις που δεν είναι εύκολο να προβλέψουμε τη μορφή της λύσης, μπορούμε να εφαρμόσουμε τη μέθοδο της επανάληψης, υπολογίζοντας απλώς κάποια φράγματα στο άθροισμα που προκύπτει και να χρησιμοποιήσουμε αυτά τα φράγματα για την εφαρμογή της μεθόδου της αντικατάστασης.

Για να αποδείξουμε ότι T(n) = Θ(n log n) με τη μέθοδο της αντικατάστασης υποθέτουμε ότι T(n/2) = Θ(n/2 log(n/2)), όπου οι σταθερές c1, c2, του συμβολισμού Θ είναι ίδιες με αυτές του Θ(n) στον ορισμό της εξίσωσης. Αντικαθιστούμε στην αναδρομική εξίσωση αυτή την υπόθεση και έχουμε:

$$T(n) \le 2\left(c_2 \frac{n}{2}\log\frac{n}{2}\right) + c_2 n \qquad T(n) \ge 2\left(c_1 \frac{n}{2}\log\frac{n}{2}\right) + c_1 n$$
$$\le c_2 n\left(\log n - 1\right) + c_2 n \qquad \ge c_1 n(\log n - 1) + c_1 n$$
$$= c_2 n \log n \qquad = c_1 n \log n$$

Παρόλο που 1log1=0, όταν το n παίρνει μικρές τιμές μεγαλύτερες της μονάδας το T(n) = Θ(1). Άρα υπάρχουν θετικές σταθερές c1, c2, για κάθε n > 1, τέτοιες ώστε c1 n log n ≤ T(n) ≤ c2 n log n. Συνεπώς, ο χρόνος εκτέλεσης χειρότερης περίπτωσης της merge sort είναι Θ(n log n).

4.2.4 Θεώρημα Κυριαρχίας(Master Theorem)

Το Θεώρημα της Κυριαρχίας αποτελεί μια μέθοδο επίλυσης αναδρομικών εξισώσεων της μορφής T(n) = aT(n/b) + f(n) όπου α ≥ 1 και b > 1 είναι σταθερές, και f(n) είναι μία ασυμπτωτικά θετική συνάρτηση.

Η παραπάνω αναδρομική σχέση περιγράφει το χρόνο εκτέλεσης ενός αλγορίθμου διαίρει και βασίλευε, που διαιρεί το αρχικό πρόβλημα μεγέθους n σε α επιμέρους προβλήματα, καθένα μεγέθους n / b, τα οποία λύνονται αναδρομικά σε χρόνο T(n/b) και το συνολικό κόστος της διαίρεσης καθώς και της σύνθεσης των επιμέρους αποτελεσμάτων για την εύρεση της τελικής λύσης δίνεται από τη συνάρτηση f(n). Για παράδειγμα, στην περίπτωση της mergeSort a = 2, b = 2, και f(n) = Θ(n). Το Θεώρημα διατυπώνεται ως εξής:

Θεώρημα:

Έστω α ≥ 1 και b > 1 σταθερές, f(n) μια ασυμπτωτικά θετική συνάρτηση και T(n) μια συνάρτηση που ορίζεται επί των μη αρνητικών ακεραίων σύμφωνα με την αναδρομική σχέση T(n) = aT(n/b) + f(n)

Σε αυτήν την περίπτωση η συνάρτησηT(n) φράσσεται ασυμπτωτικά ως εξής:

1. Αν $f(n) = O(n^{\log_b a - \varepsilon})$ για κάποια σταθερά ε > 0, τότε $T(n) = \Theta(n^{\log_b a})$.
2. Αν $f(n) = \Theta(n^{\log_b a})$, τότε $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Αν $f(n) = \Omega(n^{\log_b a + \varepsilon})$ για κάποια σταθερά ε > 0, και αν $af(n/b) \le cf(n)$ για κάποια θετική σταθερά c < 1 για κάθε n από κάποια τιμή και πάνω, τότε $T(n) = \Theta(f(n))$.

Κατά την εφαρμογή του Θεωρήματος της Κυριαρχίας, συγκρίνουμε τη συνάρτηση f(n) με τη συνάρτηση $n^{\log_b a}$ και σε κάθε περίπτωση η ασυμπτωτικά μεγαλύτερη από τις δύο συναρτήσεις αποτελεί λύση της εξίσωσης. Δεν αρκεί όμως μία από τις δύο να είναι μεγαλύτερη, αλλά πολυωνυμικά μεγαλύτερη από την άλλη, δηλαδή μικρότερη κατά ένα παράγοντα n^{ε}, για κάποια σταθερά ε > 0. Στην πρώτη περίπτωση, η f(n) είναι πολυωνυμικά μικρότερη από την $n^{\log_b a}$, οπότε η λύση της αναδρομικής εξίσωσης είναι $T(n) = \Theta(n^{\log_b a})$ Στη δεύτερη περίπτωση, όπου οι συναρτήσεις f(n) και $n^{\log_b a}$ είναι της ίδιας τάξης μεγέθους, πολλαπλασιάζουμε με ένα λογαριθμικό παράγοντα και η λύση της αναδρομικής εξίσωσης είναι $T(n) = \Theta(n^{\log_b a} \log n)$. Στην τρίτη περίπτωση, η f(n) είναι πολυωνυμικά μεγαλύτερη από την $n^{\log_b a}$ και επιπλέον θα πρέπει να ικανοποιεί τη συνθήκη $af(n/b) \le cf(n)$, η οποία ικανοποιείται από τις περισσότερες πολυωνυμικά φραγμένες συναρτήσεις. Σε αυτή την περίπτωση η λύση της αναδρομικής εξίσωσης είναι $T(n) = \Theta(f(n))$. Παρόλα αυτά, είναι δυνατόν η f(n) να μην μπορεί να ενταχθεί σε καμία από τις παραπάνω περιπτώσεις και το θεώρημα να μην μπορεί να εφαρμοστεί.

Παραδείγματα:

1. Θεωρούμε την εξίσωση T(n) = 9T(n/3) + n, όπου α = 9, b = 3 και f(n) = n. Επομένως, $n^{\log_3 9} = n^2$ και $\mathrm{f(n)} = \mathrm{O(n^{2-\varepsilon})}$, όπου ε θετική μικρή σταθερά. Άρα εφαρμόζοντας την πρώτη περίπτωση του Θεωρήματος της Κυριαρχίας, καταλήγουμε στο συμπέρασμα ότι T(n) = Θ(n^2).
2. Θεωρούμε την εξίσωση T(n) = T(2n/3) + 1, όπου α = 1, b = 3/2 και f(n) = 1. Επομένως, έχουμε $f(n) = \Theta(1) = n^{\log_{3/2} 1} = n^0$. Συνεπώς εφαρμόζεται η δεύτερη περίπτωση του θεωρήματος και προκύπτει ότι T(n) = Θ(log n).

3. Θεωρούμε την εξίσωση T(n) = 3T(n/4) + nlog n, όπου α = 3, b = 4 και f(n) = n$\log$n. Επομένως $n^{\log_4 3} = n^{0,793}$ και $f(n) = \Omega(n^{0,793+\varepsilon})$, όπου ε μια μικρή θετική σταθερά. Επίσης ισχύει ότι α f(n/b) = 3 (n/4) log(n/4) = 3/4 n log (n/4) ≤ 3/4 nlogn = cf(n), όπου c=3/4 για αρκετά μεγάλες τιμές του n. Επομένως, T(n) = Θ(n log n).
4. Θεωρούμε την εξίσωση T(n) = 2T(n/2) + nlog n, όπου α = 2, b = 2 και $f(n) = n\log n$. Επομένως $n^{\log_2 2} = n$ και f(n) = Ω(n). Η f(n) δεν είναι πολυωνυμικά μεγαλύτερη από την $n^{\log_2 2}$. Άρα δεν μπορούμε να εφαρμόσουμε το Θεώρημα της Κυριαρχίας για αυτή την εξίσωση. Αυτή η αναδρομική εξίσωση δεν ανήκει στις περιπτώσεις 2 και 3 αλλά μπορεί να λυθεί με άλλες μεθόδους όπως για παράδειγμα τη μέθοδο της επανάληψης.

4.2.5 Πολλαπλασιασμός αριθμών και πινάκων

Θεωρώντας τις αριθμητικές πράξεις στοιχειώδεις λειτουργίες, ο χρόνος εκτέλεσής του είναι σταθερός. Όμως όταν έχουμε αριθμούς με πολλά ψηφία χρησιμοποιούνται διαφορετικοί αλγόριθμοι για να εκτελεστούν οι πράξεις και συνεπώς προκύπτουν διαφορετικοί χρόνοι απόκρισης για τον καθένα. Αν θεωρήσουμε τις πράξεις των μονοψήφιων αριθμών στοιχειώδεις λειτουργίες, τότε μπορούμε να προσδιορίσουμε ευκολότερα το χρόνο εκτέλεσης αλγορίθμων που εκτελούν αριθμητικές πράξεις μεταξύ πολυψήφιων αριθμών. Στις πράξεις αριθμών με πολλά ψηφία, το μέγεθος της εισόδου ισούται με το άθροισμα των ψηφίων των αριθμών, αφού όταν αυξάνονται τα ψηφία αυξάνεται και η δυσκολία στο πρόβλημα.

Όταν γίνεται πρόσθεση δύο αριθμών που ο καθένας τους έχει n δυαδικά ψηφία, το άθροισμά τους υπολογίζεται από τον αλγόριθμο πρόσθεσης που απαιτεί Θ(n) προσθέσεις στα δυαδικά ψηφία. Ενώ όταν γίνεται πολλαπλασιασμός μεταξύ των αριθμών αυτών, απαιτούνται Θ(n^2) στοιχειώδεις πράξεις, αφού οι αριθμοί αποτελούνται από n δυαδικά ψηφία και απαιτούνται n προσθέσεις. Ο κλασικός αλγόριθμος πολλαπλασιασμού a la russe απαιτεί Θ(n^2) στοιχειώδεις πράξεις, γιατί κάθε φορά που υποδιπλασιάζεται ο πολλαπλασιαστής, διπλασιάζεται ο πολλαπλασιαστέος. Ο κάθε διπλασιασμός έχει χρόνο εκτέλεσης Θ(n) και απαιτούνται συνολικά n υποδιπλασιασμοί.

Για να εξετάσουμε αν υπάρχει κάποιος αλγόριθμος πολλαπλασιασμού ο οποίος είναι ασυμπτωτικά πιο γρήγορος στο χρόνο εκτέλεσης χειρότερης περίπτωσης, εφαρμόζουμε τη μέθοδο διαίρει και βασίλευε.

Για παράδειγμα θεωρούμε τους δυαδικούς αριθμούς x, y όπου ο αριθμός των ψηφίων n είναι ζυγός. Έστω ότι $x = x_n x_{n-1} \ldots x_2 x_1, y = y_n y_{n-1} \ldots y_2 y_1$, όπου $x_i, y_i \in \{0,1\}$ είναι τα δυαδικά ψηφία των αριθμών x και y. Κατά τη διαίρεση ξεχωρίζουμε τα σημαντικότερα δυαδικά ψηφία από τα λιγότερο σημαντικά. Οπότε ο αριθμός x μπορεί να γραφεί σαν $x = X_L + 2^{n/2} X_H$, όπου ο X_H αποτελείται από τα n/2 πιο σημαντικά δυαδικά ψηφία και ο X_L αποτελείται από τα n/2 λιγότερο σημαντικά δυαδικά ψηφία. Το ίδιο γίνεται και για τον αριθμό y, όπου $y = Y_L + 2^{n/2} Y_H$. Το γινόμενο x*y είναι ίσο με $2^n X_H Y_H + 2^{n/2}(X_H Y_L + X_L Y_H) + X_L Y_L = 2^n Z_2 + 2^{n/2} Z_1 + Z_0$, όπου $Z_2 = X_H Y_H$, $Z_1 = X_H Y_L + X_L Y_H$ και $Z_0 = X_L Y_L$. Οι παραπάνω προσθέσεις εκτελούνται σε χρόνο Θ(n) και τα επιμέρους γινόμενα υπολογίζονται με τέσσερις αναδρομικές κλήσεις του αλγορίθμου για αριθμούς με n/2 δυαδικά ψηφία. Επομένως, ο χρόνος εκτέλεσης $T_1(n)$ αυτού του αλγορίθμου δίνεται από την αναδρομική εξίσωση $T_1(n) = 4T_1(n/2) + \Theta(n)$. Εφαρμόζοντας το Θεώρημα της Κυριαρχίας, βρίσκουμε ότι $T_1(n) = \Theta(n^2)$.

Συνεπώς, στο πρόβλημα του πολλαπλασιασμού, δεν αρκεί η απευθείας εφαρμογή της μεθόδους διαίρει και βασίλευε. Αντί να υπολογίσουμε το Z_1 απευθείας με πολλαπλασιασμούς δύο ζευγαριών (n/2)–ψηφίων αριθμών, μπορούμε να υπολογίσουμε πρώτα το $Z_T = (X_H + X_L)(Y_H + Y_L)$ με έναν πολλαπλασιασμό δύο (n/2 + 1)–ψηφίων αριθμών και στη συνέχεια να υπολογίσουμε το Z_1 από τον τύπο $Z_1 = Z_T - Z_0 - Z_2$. Αυτή η διαδικασία χρειάζεται συνολικά μόνο τρεις πολλαπλασιασμούς μεταξύ (n/2)–ψηφίων αριθμών. Οι προσθέσεις που χρειάζονται είναι τέσσερις περισσότερες, αλλά η καθεμία μπορεί να εκτελεστεί σε χρόνο Θ(n).

Ο χρόνος εκτέλεσης χειρότερης περίπτωσης T(n) για το νέο αλγόριθμο δίνεται από την αναδρομική εξίσωση $T(n) = 2T(n/2) + T(n/2 + 1) + \Theta(n)$. Για να απλοποιήσουμε την εξίσωση παρατηρούμε ότι $T(n/2 + 1) = T(n/2) + \Theta(n)$. Έτσι, καταλήγουμε στην εξίσωση $T(n) = 3T(n/2) + \Theta(n)$, η οποία μπορεί να λυθεί με εφαρμογή του Θεωρήματος της Κυριαρχίας και έχει λύση $T(n) = \Theta(n^{\log 3}) = \Theta(n^{1,59})$.

Όπως συνεπάγεται, η έμμεση εφαρμογή της μεθόδου διαίρει και βασίλευε βελτιώνει σημαντικά το χρόνο εκτέλεσης του αλγορίθμου. Υπάρχουν αλγόριθμοι πολλαπλασιασμού αριθμών με πολλά ψηφία που είναι γρηγορότεροι, όπως ο αλγόριθμος με χρόνο εκτέλεσης χειρότερης περίπτωσης $O(n \log^2 n)$. Ένας γνωστός αλγόριθμος με χρόνο εκτέλεσης χειρότερης περίπτωσης $O(n \log n \log\log n)$ είναι αυτός των Schonhage και Strassen, ο οποίος όμως δεν χρησιμοποιείται στην πράξη εξαιτίας της μεγάλης πολλαπλασιαστικής σταθεράς που κρύβεται από τον ασυμπτωτικό συμβολισμό.

4.2.6 Πολλαπλασιασμός πινάκων – Θεώρημα Strassen

Ο V.Strassen παρουσίασε στα τέλη της δεκαετίας του 1960 έναν διαίρει και βασίλευε αλγόριθμο πολλαπλασιασμού πινάκων με χρόνο εκτέλεσης $O(n^{\log 7}) = O(n^{2.81})$. Η βασική ιδέα του αλγόριθμου Strassen είναι παρόμοια με αυτή του αλγόριθμου πολλαπλασιασμού πολυψήφιων αριθμών.

Έστω A, B δύο πίνακες n x n των οποίων θέλουμε να υπολογίσουμε το γινόμενο C = A x B. Αρχικά εφαρμόζουμε τον ορισμό:

$$\forall_i = 1,\ldots,n,\ j = 1,\ldots,n \quad C[i,j] = \sum_{k=1}^{n} A[i,k]B[k,j]$$

Με βάση τον ορισμό κάθε στοιχείο του πίνακα C μπορεί να υπολογιστεί σε γραμμικό χρόνο $\Theta(n)$. Εφόσον ο C έχει n^2 στοιχεία, ο χρόνος εκτέλεσης του αλγορίθμου είναι $\Theta(n^3)$. Έστω ότι ο n είναι άρτιος. Τότε οι πίνακες A, B, C μπορούν να διαιρεθούν σε τέσσερις υποπίνακες μεγέθους $\frac{n}{2} \times \frac{n}{2}$ ως εξής:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad , \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad , \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

όπου

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Με αυτόν τον τρόπο ο υπολογισμός του γινομένου δύο πινάκων n x n ανάγεται στον υπολογισμό των γινομένων οκτώ ζευγαριών πινάκων $\frac{n}{2}\times\frac{n}{2}$ και σε τέσσερις προσθέσεις πινάκων $\frac{n}{2}\times\frac{n}{2}$. Το άθροισμα δύο πινάκων n x n μπορεί να υπολογισθεί σε χρόνο $\Theta(n^2)$. Επομένως ο χρόνος εκτέλεσης $T_1(n)$ του παραπάνω αλγόριθμου πολλαπλασιασμού πινάκων δίνεται από την αναδρομική εξίσωση $T_1(n) = 8T_1(n/2) + \Theta(n^2)$ με αρχική συνθήκη $T_1(1) = \Theta(1)$. Η λύση αυτής της εξίσωσης είναι $T_1(n) = \Theta(n^3)$. Όπως και στον πολλαπλασιασμό αριθμών, η προφανής εφαρμογή της μεθόδου διαίρει και βασίλευε δεν οδηγεί σε καμία βελτίωση. Μπορούμε όμως να βελτιώσουμε το χρόνο εκτέλεσης υπολογίζοντας τους υποπίνακες του C με διαφορετικό τρόπο. Αρχικά υπολογίζουμε τους παρακάτω πίνακες $\frac{n}{2}\times\frac{n}{2}$:

$$D_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11})$$

$$D_2 = A_{11}B_{11}$$

$$D_3 = A_{12}B_{21}$$

$$D_4 = (A_{11} - A_{21})(B_{22} - B_{12})$$

$$D_5 = (A_{21} + A_{22})(B_{12} - B_{11})$$

$$D_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22}$$

$$D_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21})$$

Στη συνέχεια υπολογίζουμε τους τέσσερις υποπίνακες του γινομένου C από τους τύπους:

$$C_{11} = D_2 + D_3$$

$$C_{12} = D_1 + D_2 + D_5 + D_6$$

$$C_{21} = D_1 + D_2 + D_4 - D_7$$

$$C_{22} = D_1 + D_2 + D_4 + D_5$$

Η παραπάνω διαδικασία υπολογισμού είναι γνωστή σαν αλγόριθμος του Strassen και απαιτεί την εκτέλεση μόνο επτά πολλαπλασιασμών πινάκων $\frac{n}{2}\times\frac{n}{2}$ και ενός σταθερού αριθμού προσθέσεων μεταξύ πινάκων $\frac{n}{2}\times\frac{n}{2}$. Το άθροισμα δύο πινάκων $\frac{n}{2}\times\frac{n}{2}$ μπορεί να υπολογισθεί σε χρόνο $\Theta(n^2)$.

Έτσι ο χρόνος εκτέλεσης του αλγόριθμου Strassen, έστω T(n), δίνεται από την αναδρομική εξίσωση $7T(n/2)+\Theta(n^2)$ με αρχική συνθήκη T(1)=Θ(1). Η λύση αυτής της εξίσωσης είναι $T(n)=\Theta(n^{\log 7})=\Theta(n^{2.81})$.

Η μεγάλη πολλαπλασιαστική σταθερά στο χρόνο εκτέλεσης του αλγορίθμου Strassen καθιστά τον αλγόριθμο πρακτικά εφαρμόσιμο μόνο για μεγάλες τιμές του n. Σήμερα είναι γνωστός αλγόριθμος πολλαπλασιασμού πινάκων με χρόνο εκτέλεσης $O(n^{2.376})$.

4.2.7 Υπολογισμός δύναμης

Η χρησιμότητα της μεθόδου υπολογισμού δύναμης φαίνεται σε μια απλή κρυπτογραφική εφαρμογή. Έστω ότι δύο άνθρωποι α και β επικοινωνούν για πρώτη φορά και για να προστατεύσουν την επικοινωνία τους θα πρέπει να χρησιμοποιήσουν ένα κοινό κρυπτογραφικό κλειδί. Η επικοινωνία τους είναι τηλεφωνική, οπότε θα πρέπει να εξασφαλίσουν ότι θα γνωρίζουν μόνο αυτοί οι δύο το κλειδί και δεν θα γνωστοποιηθεί σε κάποιον τρίτο που μπορεί να παρακολουθεί την επικοινωνία τους. Αρχικά αυτοί οι δύο συμφωνούν σε έναν πολυψήφιο πρώτο αριθμό p και σε έναν πολυψήφιο αριθμό q ο οποίος είναι μικρότερος του p. Οι αριθμοί αυτοί δεν μπορούν να θεωρηθούν κρυφοί, αφού κάποιο τρίτο πρόσωπο που παρακολουθεί τη συνομιλία μπορεί να τους μάθει.

Έπειτα ο α και ο β διαλέγουν τυχαία και ανεξάρτητα από έναν πολυψήφιο ακέραιο αριθμό μικρότερο του p. Έστω x και y οι αριθμοί αυτοί αντίστοιχα. Ο α υπολογίζει τον αριθμό $q_x = q^x \bmod p$ και τον κοινοποιεί στον β και αντίστοιχα ο β υπολογίζει τον αριθμό $q_y = q^y \bmod p$ και τον κοινοποιεί στον α. Όπως και τα p,q έτσι και οι αριθμοί q_x, q_y δεν μπορούν να θεωρηθούν μυστικοί από κάποιο τρίτο πρόσωπο.

Ο α γνωρίζει τους αριθμούς p, x, q_y και υπολογίζει το

$$K_x = q_y^x \bmod p = (q^y \bmod p)^x \bmod p = q^{xy} \bmod p$$

Ο β γνωρίζει τους αριθμούς p, y, q_x και υπολογίζει το

$$K_y = q_x^y \bmod p = (q^x \bmod p)^y \bmod p = q^{xy} \bmod p$$

Όπως προκύπτει $K_X = K_Y$. Αυτός ο αριθμός ο οποίος έχει υπολογισθεί από τον α και τον β ξεχωριστά δεν χρειάζεται να μεταδοθεί μέσω της τηλεφωνικής συνομιλίας και μπορεί να χρησιμοποιηθεί σαν κοινό μυστικό κλειδί ανάμεσά τους.

Η εφαρμογή του πρωτοκόλλου υπολογισμού δύναμης προϋποθέτει τον γρήγορο υπολογισμό της νιοστής δύναμης ενός ακεραίου αριθμού x στην αριθμητική υπολοίπου ως προς p. Επομένως χρειάζεται ένας αποδοτικός αλγόριθμος για τον υπολογισμό του $x^n \bmod p$ όπου x, n και p είναι πολυψήφιοι ακέραιοι αριθμοί. Ο αλγόριθμος αυτός είναι απαραίτητος για τον υπολογισμό των $q_x = q^x \bmod p$ και $q_y = q^y \bmod p$.

Αφού το n είναι ένας πολύ μεγάλος αριθμός, δεν μπορεί να εφαρμοστεί η προφανής προσέγγιση του υπολογισμού όλων των δυνάμεων x^i mod p, i=2,…,n, πολλαπλασιάζοντας το x^{i-1} mod p με το x. Η μέθοδος διαίρει και βασίλευε δίνει ένα απλό αναδρομικό αλγόριθμο για την επίλυση αυτού του προβλήματος.

Αν το n είναι άρτιος η νιοστή δύναμη του x προκύπτει ως το τετράγωνο της $\frac{n}{2}$-οστής δύναμης του x. Η $\frac{n}{2}$-οστή δύναμη του x υπολογίζεται αναδρομικά με τον ίδιο αλγόριθμο. Αν το n είναι περιττός , η νιοστή δύναμη του x προκύπτει πολλαπλασιάζοντας το x με το τετράγωνο της $\frac{n-1}{2}$-οστής δύναμης του x, που υπολογίζεται αναδρομικά από τον ίδιο αλγόριθμο. Μια αναδρομική υλοποίηση του αλγόριθμου είναι η παρακάτω:

```
ExponRec(x,n,p)
        if n=1 then return (x mod p);
        t ← ExponRec(x, [n/2],p);
        t ← t^2 mod p;
        if n is odd then return (t× x mod p);
        else return(t);
```

Υποθέτουμε ότι ο χρόνος εκτέλεσης του παραπάνω αλγόριθμου είναι T(n) για τον υπολογισμό της νιοστής δύναμης ενός αριθμού. Ο αλγόριθμος εκτελεί μια αναδρομική κλήση για τον υπολογισμό της $\frac{n}{2}$-οστής δύναμης και το πολύ δύο πολλαπλασιασμούς αριθμών που δεν ξεπερνούν το p δηλαδή έχουν το πολύ logp ψηφία. Η αναδρομική κλήση απαιτεί χρόνο T(n/2). Ο χρόνος για τον πολλαπλασιασμό δύο αριθμών μικρότερων του p είναι $O(\log^2 p)$.

Το T(n) δίνεται από την αναδρομική σχέση $T(n) = T(n/2) + O(\log^2 p)$ με αρχική συνθήκη T(1) = O(1). Η λύση αυτής της αναδρομικής εξίσωσης είναι $T(n) = O(\log n \log^2 p)$ αφού σε κάθε βήμα ο αλγόριθμος χρειάζεται χρόνο $O(\log^2 p)$ και το n υποδιπλασιάζεται. Η βελτίωση που προκύπτει είναι ότι αν το n έχει 512 δυαδικά ψηφία, ο αλγόριθμος εκτελεί το πολύ 2^{10}=1024 πολλαπλασιασμούς και όχι 2^{512} που θα χρειαζόταν αν ο υπολογισμός δυνάμεων γινόταν με τη σειρά.

4.3 Άπληστοι αλγόριθμοι

Οι αλγόριθμοι προβλημάτων βελτιστοποίησης συνήθως ακολουθούν μια σειρά από βήματα, κάνοντας σε κάθε βήμα επιλογές, που σχετίζονται με τη μορφή της βέλτιστης λύσης. Ένας άπληστος αλγόριθμος κάνει την επιλογή που δείχνει καλύτερη, τη δεδομένη χρονική στιγμή. Δηλαδή, ένας άπληστος αλγόριθμος κάνει σε κάθε βήμα την βέλτιστη επιλογή, με στόχο να επιτύχει μια συνολικά βέλτιστη επίλυση.

Η δομή των άπληστων αλγόριθμων είναι απλή και εφαρμόζονται σε προβλήματα βελτιστοποίησης. Ο αλγόριθμος λειτουργεί σε βήματα και κάνει μια επιλογή σε σχέση με τη μορφή της λύσης σε κάθε βήμα. Ο άπληστος αλγόριθμος έχει τα εξής χαρακτηριστικά:

- Ταξινομεί τις βασικές συνιστώσες του στιγμιότυπου εισόδου ως προς κάποιο κριτήριο που εξαρτάται από το πρόβλημα.
- Επιλέγει αμετάκλητα αν θα συμπεριλάβει την καλύτερη βασική συνιστώσα στη λύση.
- Εφαρμόζει τον εαυτό του στο υπο-στιγμιότυπο που προκύπτει με βάση την παραπάνω επιλογή.

Με βάση αυτήν την περιγραφή ένας άπληστος αλγόριθμος χαρακτηρίζεται από το κριτήριο ταξινόμησης των βασικών συνιστωσών και από το κριτήριο επιλογής της καλύτερης βασικής συνιστώσας στη λύση. Ονομάζουμε έναν άπληστο αλγόριθμο προσαρμοστικό, όταν μπορεί να διαφοροποιήσει την ταξινόμηση των βασικών συνιστωσών από βήμα σε βήμα και μη προσαρμοστικό όταν ακολουθεί την αρχική ταξινόμηση των βασικών συνιστωσών σε όλα τα βήματα.

Η επιλογή για κάθε βασική συνιστώσα είναι αμετάκλητη και η καλύτερη βασική συνιστώσα εξαιρείται από το υπο-στιγμιότυπο που θα εξεταστεί στο επόμενο βήμα.

Αν έχει επιλεγεί στη λύση, τότε οι υπόλοιπες παράμετροι του υπο-στιγμιότυπου αναπροσαρμόζονται με βάση αυτή την επιλογή. Διαφορετικά η καλύτερη βασική συνιστώσα αγνοείται. Όλες οι υπόλοιπες βασικές συνιστώσες συμπεριλαμβάνονται στο υπο-στιγμιότυπο για να εξετασθούν στα επόμενα βήματα.

Σε κάθε βήμα ο άπληστος αλγόριθμος εφαρμόζει την ίδια στρατηγική στο υπο-στιγμιότυπο που προκύπτει από τις επιλογές των προηγούμενων βημάτων. Όμως η εφαρμογή του άπληστου αλγόριθμου στο υποστιγμιότυπο δεν είναι ουσιαστικά αναδρομική, αφού σε κάθε βήμα έχουμε περιορισμό, εξειδίκευση και όχι διαίρεση του στιγμιότυπου. Έτσι οι άπληστοι αλγόριθμοι διατυπώνονται επαναληπτικά και δεν θεωρούνται αναδρομικοί αλγόριθμοι. Το γεγονός ότι εφαρμόζουν την ίδια στρατηγική σε όλο και μικρότερα στιγμιότυπα βρίσκει εφαρμογή στην απόδειξη ορθότητας η οποία γίνεται συνήθως με μαθηματική επαγωγή.

Η μέθοδος των τοπικά βέλτιστων επιλογών μπορεί να είναι επιτυχημένη σε κάποια προβλήματα, αλλά σε κάποια άλλα δεν επιτυγχάνει βέλτιστες λύσεις. Οι άπληστοι αλγόριθμοι σχεδιάζονται εύκολα και συνήθως είναι απλοί στη σύλληψή τους. Επίσης είναι ιδιαίτερα αποδοτικοί όσον αφορά τις υπολογιστικές απαιτήσεις, όπως είναι ο χρόνος εκτέλεσης και ο αριθμός των θέσεων μνήμης. Το πιο δύσκολο σημείο στην ανάλυση άπληστων αλγορίθμων είναι η απόδειξη της ορθότητάς τους, δηλαδή ότι καταλήγουν σε μία πραγματικά βέλτιστη λύση.

4.3.1 Επιλογή δραστηριοτήτων

Η μέθοδος της απληστίας εφαρμόζεται στο συχνό πρόβλημα της δρομολόγησης δραστηριοτήτων, οι οποίες επιδιώκουν πρόσβαση σε έναν κοινό πόρο. Σε μια απλή προσέγγιση του προβλήματος έχουμε ένα σύνολο δραστηριοτήτων $A = \{1, 2, \ldots, n\}$, όπου κάθε δραστηριότητα i αρχίζει τη χρονική στιγμή s_i και ολοκληρώνεται τη χρονική στιγμή f_i, όπου $f_i \geq s_i \geq 0$. Σε μία δεδομένη χρονική στιγμή υπάρχει η δυνατότητα εκτέλεσης μίας μόνο δραστηριότητας του συνόλου A, αφού χρησιμοποιούν όλες έναν κοινό πόρο. Κάθε δραστηριότητα δεσμεύει τον πόρο κατά το χρονικό διάστημα $[s_i, f_i)$. Ονομάζουμε συμβατές δύο δραστηριότητες των οποίων τα χρονικά διαστήματα δεν επικαλύπτονται.

Για να επιτύχουμε τη βέλτιστη λύση, θα πρέπει να βρούμε τις κατά το δυνατόν περισσότερες συμβατές δραστηριότητες. Οι δραστηριότητες αποτελούν τις βασικές συνιστώσες του προβλήματος.

Για να θεωρήσουμε μια δραστηριότητα κατάλληλη προς επιλογή, αρκεί να ορίσουμε κάποιο κριτήριο ταξινόμησης και επιλογής. Η δραστηριότητα που θα επιλέξουμε δεν θα πρέπει να χρησιμοποιεί καταχρηστικά τον κοινό πόρο, έτσι ώστε να μπορούν να εκτελεστούν και οι υπόλοιπες δραστηριότητες. Επομένως κάποια κριτήρια είναι η επιλογή της δραστηριότητας με το μικρότερο χρόνο έναρξης, η επιλογή της δραστηριότητας που απελευθερώνει νωρίτερα τον πόρο και η επιλογή της δραστηριότητας που έχει τη λιγότερη διάρκεια.

Το πρόβλημα επιλογής δραστηριοτήτων λύνεται από έναν άπληστο αλγόριθμο, δρομολογώντας σε κάθε βήμα μία δραστηριότητα. Οι δραστηριότητες που επιλέγονται θα διατηρούνται σε ένα σύνολο C. Σε κάθε βήμα, επιλέγεται μία δραστηριότητα από το σύνολο των δραστηριοτήτων A, η οποία θεωρείται βέλτιστη τη δεδομένη χρονική στιγμή και είναι συμβατή με τις δραστηριότητες του C. Η δραστηριότητα που επιλέγεται μπαίνει στο σύνολο C και αυτό γίνεται σε κάθε βήμα, έως ότου να μην υπάρχουν άλλες συμβατές δραστηριότητες.

Ο παρακάτω ψευδοκώδικας υλοποιεί τον αλγόριθμο για την περίπτωση που οι δραστηριότητες είναι διατεταγμένες σε αύξουσα σειρά ως προς τους χρόνους ολοκλήρωσης τους, δηλαδή $f_1 \leq f_2 \leq \ldots \leq f_n$ και σε κάθε βήμα επιλέγεται μια δραστηριότητα που είναι συμβατή με τις υπόλοιπες δραστηριότητες του C και έχει το μικρότερο χρόνο ολοκλήρωσης.

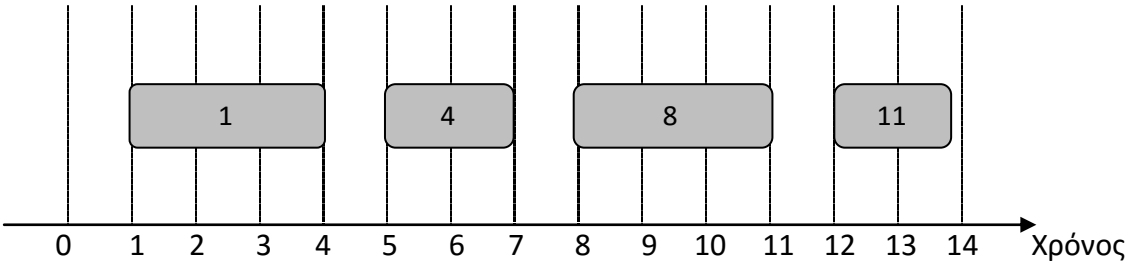
```
GREEDY - SELECTION(A = [(s1, f1),...,(sn, fn)])
{ θεωρούμε ότι f1 ≤ f2 ≤ ... ≤ fn }
C ← {1}; j ← 1;
for i ← 2 to n do
  if si ≥ fj then C ← C ∪{i}; j ← i;
return C;
```

Η μεταβλητή j περιέχει την τελευταία δραστηριότητα που επιλέχθηκε στο C, δηλαδή τη δραστηριότητα του C με το μεγαλύτερο χρόνο ολοκλήρωσης. Έπειτα γίνεται αναζήτηση στο σύνολο A για την επόμενη δραστηριότητα i, που μπορεί να προστεθεί στο C, δηλαδή να έχει $s_i \geq f_j$. Δεδομένου ότι οι δραστηριότητες του A είναι διατεταγμένες σε αύξουσα σειρά ως προς τους χρόνους ολοκλήρωσης, η δραστηριότητα i έχει το μικρότερο f_i από όλες αυτές που μπορούν να προστεθούν στο C. Όταν οι δραστηριότητες είναι διατεταγμένες σε αύξουσα σειρά των χρόνων ολοκλήρωσης, ο χρόνος εκτέλεσης του αλγορίθμου είναι Θ(n). Σε άλλη περίπτωση πρέπει να εκτελεστεί ακόμα ένα βήμα με χρόνο εκτέλεσης Θ(n log n) για τη διάταξη των δραστηριοτήτων.

Πίνακας 3: Στιγμιότυπο του αλγορίθμου GREEDY-SELECTION

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Στο παραπάνω στιγμιότυπο παρουσιάζεται η λειτουργία του αλγορίθμου επιλογής δραστηριοτήτων. Αρχικά δρομολογείται η δραστηριότητα 1, οπότε οι δραστηριότητες 2 και 3 αποκλείονται. Μετά δρομολογείται η δραστηριότητα 4, αφού $s_4 = 5 > f_1 = 4$, οπότε αποκλείονται οι δραστηριότητες 5, 6 και 7. Η επόμενη δραστηριότητα που δρομολογείται είναι η 8, αφού $s_8 = 8 > f_4 = 7$, η οποία με τη σειρά της αποκλείει τις δραστηριότητες 9 και 10. Τέλος η δραστηριότητα 11 επικαλύπτεται χρονικά μόνο με τη δραστηριότητα 10. Καταλήγουμε στη βέλτιστη λύση {1, 4, 8, 11} που αποτελεί ένα σύνολο από τέσσερις συμβατές δραστηριότητες. Η δρομολόγηση των δραστηριοτήτων φαίνεται στο παρακάτω σχήμα.



Σχήμα 9: Δρομολόγηση δραστηριοτήτων

Μπορούμε να βρούμε περισσότερες από μία βέλτιστες λύσεις για το συγκεκριμένο στιγμιότυπο, όπως γίνεται και σε άλλα παραδείγματα άπληστων αλγορίθμων. Για το λόγο αυτό πρέπει να τεκμηριώνουμε κάθε βέλτιστη λύση που βρίσκουμε με την απόδειξη ορθότητας. Σύμφωνα με την ιδιότητα της βέλτιστης επιλογής, αποδεικνύουμε ότι υπάρχει πάντα μία βέλτιστη λύση η οποία περιέχει το τοπικά βέλτιστο στοιχείο που επιλέγει ο άπληστος αλγόριθμος. Στη συνέχεια σύμφωνα με την ιδιότητα βέλτιστων επιμέρους δομών, δείχνουμε ότι δεδομένης της άπληστης επιλογής, ο αλγόριθμος απομένει να υπολογίσει τη βέλτιστη λύση ενός επιμέρους στιγμιότυπου του ίδιου προβλήματος.

Η απόδειξη της ορθότητας ολοκληρώνεται εφαρμόζοντας επαγωγικά την ιδιότητα της βέλτιστης επιλογής σε κάθε στιγμιότυπο που προκύπτει από κάθε βήμα του άπληστου αλγορίθμου. Σε κάποιες περιπτώσεις είναι πιο εύκολο να διατυπώσουμε τις αποδείξεις ορθότητας συγκρίνοντας τη λύση που υπολογίζει ο άπληστος αλγόριθμος με τα στοιχεία μιας βέλτιστης λύσης.

4.3.2 Το πρόβλημα του σακιδίου

Ένας ορειβάτης προετοιμάζει το σακίδιό του για την ανάβαση στο βουνό. Στη διάθεσή του έχει ένα σύνολο από n αντικείμενα $X = \{x_1, x_2, \ldots, x_n\}$ με βάρη $a_1, a_2, \ldots, a_n$. Κάθε αντικείμενο δίνει ένα κέρδος στον ορειβάτη c_i αν το επιλέξει για να το πάρει μαζί του στην ανάβαση. Ο ορειβάτης πρέπει να επιλέξει το υποσύνολο των αντικειμένων που μεγιστοποιούν το όφελός του, προσέχοντας να μην υπερβαίνεται το βάρος του σακιδίου b. Με άλλα λόγια, αν με $z_i \in \{0, 1\}$ συμβολίζουμε το αν ο ορειβάτης έχει επιλέξει το αντικείμενο i, ο στόχος είναι η μεγιστοποίηση του αθροίσματος:

$$\max \sum_{i=1}^{n} c_i z_i \quad \text{με τον περιορισμό} \quad \sum_{i=1}^{n} a_i z_i \leq b$$

Μια λογική προσέγγιση για την επίλυση του προβλήματος με άπληστο αλγόριθμο, είναι η ταξινόμηση των αντικειμένων σύμφωνα με τον όρο c_i/a_i που εκφράζει το κέρδος του ορειβάτη ανά μονάδα βάρους του αντικειμένου. Τότε οδηγούμαστε στον παρακάτω άπληστο αλγόριθμο:

```
Discrete-Knapsack (X : σύνολο αντικειμένων, a, c : αντίστοιχα
βάρη και αξίες, b : βάρος σακιδίου)
1. Φθινουσα-Ταξινομηση(ci/ai)
2. S = Ø
3. for i = 1 to n
4.      if b ≥ ai
5.           S = S U {xi}
6.           b = b – ai
7.      end if
8. end for
```

Ωστόσο η προσέγγιση αυτή δεν δουλεύει πάντα. Ας θεωρήσουμε την εξής εφαρμογή: Δίδονται 3 αντικείμενα $\{x_1, x_2, x_3\}$ με βάρη a = (k + 1, k, k) και αξίες c = (k+2, k, k). Το βάρος του σακιδίου είναι 2k. Η ταξινόμηση των αντικειμένων σε φθίνουσα σειρά του λόγου c_i/a_i είναι x_1, x_2, x_3 αφού

$$\frac{k+2}{k+1} > \frac{k}{k} = \frac{k}{k}$$

Έτσι ο αλγόριθμος θα επιλέξει το αντικείμενο x_1 με κέρδος k+2 αντί να επιλέξει τα αντικείμενα x_2, x_3 με κέρδος 2k. Βλέπουμε δηλαδή ότι για μεγάλο k το λάθος που κάνει ο άπληστος αλγόριθμος προσεγγίζει το 100%.

Το πρόβλημα του σακιδίου είναι NP–δύσκολο, οπότε η προσπάθεια εύρεσης μίας αποδοτικής σειράς εξερεύνησης των στοιχείων θα είναι αποτυχημένη. Ωστόσο μπορεί να επιλυθεί βέλτιστα από έναν αλγόριθμο της κατηγορίας δυναμικού προγραμματισμού που θα δούμε στο επόμενο κεφάλαιο.

4.3.3 Το πρόβλημα του σακιδίου με επανάληψη

Το συνεχές πρόβλημα σακιδίου, είναι η παραλλαγή του προβλήματος σακιδίου που επιτρέπεται στον ορειβάτη να πάρει μέρη των αντικειμένων. Με άλλα λόγια οι μεταβλητές απόφασης z_i του ακέραιου προβλήματος παίρνουν συνεχείς τιμές στο [0, 1]. Εδώ η φθίνουσα ταξινόμηση με βάση τον λόγο c_i/a_i δουλεύει αποδοτικά και βρίσκει την βέλτιστη λύση των αντικειμένων που πρέπει να επιλέξει ο ορειβάτης για να έχει μεγιστοποίηση του κέρδους του.

Ο άπληστος αλγόριθμος:

```
Continuous-Knapsack (X : σύνολο αντικειμένων, a, c :
αντίστοιχα βάρη και αξίες,
b : βάρος σακιδίου)
1. Φθινουσα-Ταξινομηση(c_i/a_i)
2. for i = 1 to n Θέσε z_i = 0
3. i = 1
3. while b ≥ a_i do
4.     z_i = 1
5.     b = b – a_i
6.     i = i + 1
7. end while
8. z_i = b * (c_i/a_i)
```

Κεφάλαιο 5: Δυναμικός προγραμματισμός και προβλήματα από τη θεωρία γραφημάτων

5.1 Δυναμικός προγραμματισμός

Η μέθοδος του δυναμικού προγραμματισμού όπως και η μέθοδος διαίρει και βασίλευε, συνδυάζει τις λύσεις επιμέρους προβλημάτων για να λύσει ένα συνολικό πρόβλημα. Με τη μέθοδο διαίρει και βασίλευε, το αρχικό πρόβλημα διαιρείται σε ανεξάρτητα μεταξύ τους υποπροβλήματα, που επιλύονται αναδρομικά και ο συνδυασμός των λύσεών τους αποτελεί τη λύση στο αρχικό πρόβλημα. Με τη μέθοδο του δυναμικού προγραμματισμού, το αρχικό πρόβλημα διαιρείται σε υποπροβλήματα που δεν είναι ανεξάρτητα μεταξύ τους αλλά αλληλεπικαλύπτονται. Ένας αλγόριθμος δυναμικού προγραμματισμού, επιλύει μία φορά κάθε υποπρόβλημα και αποθηκεύει αυτή τη λύση σε έναν πίνακα, στον οποίον θα καταφεύγει κάθε φορά που συναντά το συγκεκριμένο πρόβλημα. Λόγω της αποθήκευσης των λύσεων των υποπροβλημάτων τους, οι αλγόριθμοι δυναμικού προγραμματισμού απαιτούν αρκετές θέσεις μνήμης καθώς και αποθηκευτικό χώρο.

Ο δυναμικός προγραμματισμός αποτελεί μία υπολογιστική μέθοδο η οποία εφαρμόζεται σε προβλήματα που δεν είναι δυνατόν να λυθούν με άπληστες μεθόδους ή με τη μέθοδο διαίρει και βασίλευε. Ένας αλγόριθμος διαίρει και βασίλευε κάνει περισσότερη δουλειά από όση χρειάζεται, αφού επιλύει πολλές φορές το ίδιο επιμέρους πρόβλημα. Άρα ένας δυναμικός αλγόριθμος είναι περισσότερο αποδοτικός.

Συνήθως η μέθοδος του δυναμικού προγραμματισμού εφαρμόζεται σε προβλήματα βελτιστοποίησης, καθώς η αρχή της βελτιστοποίησης αποτελεί θεμέλιό της. Στα προβλήματα βελτιστοποίησης αναζητούμε τη λύση με τη βέλτιστη τιμή, καθώς υπάρχουν πολλές λύσεις που μπορούν να είναι αποδεκτές. Ο δυναμικός προγραμματισμός εφαρμόζεται σε προβλήματα βελτιστοποίησης των οποίων οι βέλτιστες λύσεις αποτελούνται από τις βέλτιστες λύσεις συγκεκριμένων επιμέρους δομών.

Η ανάπτυξη ενός αλγορίθμου δυναμικού προγραμματισμού μπορεί να αναλυθεί σε μία σειρά από τέσσερα βήματα:

- Χαρακτηρισμός της δομής μιας βέλτιστης λύσης.
- Αναδρομικός ορισμός της τιμής μιας βέλτιστης λύσης.
- Υπολογισμός της τιμής μιας βέλτιστης λύσης, προχωρώντας από τα μικρότερα στα μεγαλύτερα επιμέρους προβλήματα.
- Κατασκευή μιας βέλτιστης λύσης από τα δεδομένα που υπολογίστηκαν.

Χαρακτηριστικό των προβλημάτων που επιλύονται με τη μέθοδο του δυναμικού προγραμματισμού είναι ότι οι αποφάσεις λαμβάνονται διαδοχικά. Επίσης το πρόβλημα μπορεί να διαιρεθεί σε βήματα και σε κάθε βήμα απαιτείται να ληφθεί μία στρατηγική απόφαση. Κάθε βήμα έχει ένα ορισμένο αριθμό καταστάσεων που συνδέονται με αυτό. Το αποτέλεσμα μίας στρατηγικής απόφασης που λαμβάνεται σε κάθε βήμα είναι να μετατρέπει την παρούσα κατάσταση σε μία κατάσταση που συνδέεται με το επόμενο βήμα. Με κάθε απόφαση συνδέεται ένα κέρδος ή μία ζημία (κόστος). Ο αντικειμενικός σκοπός είναι να μεγιστοποιηθεί το συνολικό κέρδος ή να ελαχιστοποιηθεί η συνολική ζημία ή γενικότερα να επιτευχθεί το καλύτερο δυνατό αποτέλεσμα. Οι αποφάσεις που παίρνονται σε κάθε βήμα εξαρτώνται μόνον από την κατάσταση του προηγούμενου βήματος και όχι από τον τρόπο με τον οποίο πραγματοποιήθηκε το βήμα αυτό.
Κύριο χαρακτηριστικό του δυναμικού προγραμματισμού είναι ότι δεν υπάρχει γενικευμένη διατύπωση της μεθόδου για την επίλυση όλων των προβλημάτων με τα παραπάνω χαρακτηριστικά. Η μεθοδολογία του δυναμικού προγραμματισμού βασίζεται στην αρχή της βελτιστοποίησης, σύμφωνα με την οποία μία βέλτιστη πολιτική έχει την ιδιότητα, οποιαδήποτε και αν είναι η αρχική κατάσταση και η αρχική απόφαση, οι υπόλοιπες αποφάσεις πρέπει να αποτελούν μία βέλτιστη πολιτική, σε σχέση με την κατάσταση η οποία προκύπτει από την πρώτη απόφαση.

5.1.2 Παράδειγμα 1 - το πρόβλημα του σακιδίου με δυναμικό προγραμματισμό

Στο πρόβλημα αυτό μας δίνονται, ένα σύνολο X = {x_1, x_2, . . . , x_n} από n αντικείμενα που το καθένα έχει βάρος a_i και ένα κέρδος c_i, i = 1, 2, . . . , n. Επίσης μας δίνεται ένας ακέραιος W. Ζητάμε να επιλέξουμε ένα υποσύνολο Y ⊆ X τέτοιο ώστε $\sum_{x_i \in Y} c_i = \max$ και επιπλέον $\sum_{x_i \in Y} a_i \le W$.

Δηλαδή θέλουμε να επιλέξουμε τόσα αντικείμενα ώστε να μεγιστοποιήσουμε το κέρδος μας, χωρίς όμως να ξεπεράσουμε το συνολικό βάρος W. Για να υποδηλώσουμε ότι ένα αντικείμενο x_i έχει επιλεγεί, θέτουμε $x_i = 1$ ενώ σε αντίθετη περίπτωση θέτουμε $x_i = 0$.

Είναι γνωστό ότι το πρόβλημα αυτό είναι NP − hard. Παρόλα αυτά ο παρακάτω αλγόριθμος δυναμικού προγραμματισμού, επιλύει το πρόβλημα αυτό σε ψευδοπολυωνικό χρόνο.

- Βήμα 1: Βέλτιστα διασπώμενη δομή

Θεωρούμε την οικογένεια προβλημάτων:

$$P_k(y) = \{\max \sum_{j=1}^{k} c_j x_j, \sum_{j=1}^{k} a_j x_j \le y, x_j = 0 \acute{\eta} 1\} \quad (1)$$

όπου το y παίρνει τιμές από 0 έως το W και αντίστοιχα το k μεταβάλλεται από 1 έως n.

Η σχέση (1) υποδηλώνει μια οικογένεια πλήθους n(W + 1) υποπροβλημάτων αυξανόμενου μεγέθους, που το καθένα προκύπτει από το προηγούμενο, αυξάνοντας τον χώρο των δεδομένων εισόδου κατά ένα στοιχείο και αυξάνοντας διαδοχικά το συνολικό επιτρεπτό βάρος μέχρι το W. Για παράδειγμα, το πρόβλημα $P_2(y)$ ορίζει το πρόβλημα του σακιδίου όπου το σύνολο Q = {x_1, x_2} για κάποιο 0 ≤ y ≤ W. Το υποπρόβλημα $P_3(y)$ προκύπτει αν θέσουμε όπου X = X ∪ {x_3} για 0 ≤ y ≤ W.

- Βήμα 2: Αναδρομικός ορισμός της τιμής της βέλτιστης λύσης

Έστω $f_k(y)$ η τιμή της βέλτιστης λύσης του υποπροβλήματος $P_k(y)$. Τότε είναι αρκετά εύκολο να εξάγουμε από την σχέση (1), ότι

$$f_{k+1}(y) = \begin{cases} f_k(y) \\ \max\{f_k(y), f_k(y - a_{k+1}) + c_{k+1}\} \end{cases}, \quad (2)$$

αν $\alpha_{k+1} > y$ διαφορετικά για 1 ≤ k ≤ n και 0 ≤ y ≤ W.

Η αναδρομική εξίσωση (2), δηλώνει ότι η βέλτιστη λύση του υποπροβλήματος $P_{k+1}(y)$ είναι είτε η ίδια με αυτήν του $P_k(y)$ ή αυτή που προκύπτει αν μπορούμε να εισάγουμε το στοιχείο x_{k+1} αυξάνοντας το κέρδος κατά c_{k+1} και ελαττώνοντας το διαθέσιμο βάρος y κατά a_{k+1}. Η βέλτιστη λύση του προβλήματος $P_n(W)$ έχει την τιμή $f_n(W)$.

Για k = 1 έχουμε $f_1(y) = 0$ αν $a_1 > y$ και $f_1(y) = c_1$ αν $a_1 \leq y$, για όλα τα y, μικρότερα ή ίσα του W. Επίσης έχουμε $f_i(0) = 0$, ∀ i = 1, 2, . . . , n.

- Βήμα 3: Υπολογισμός της βέλτιστης λύσης

Παρατηρούμε ότι στην αναδρομική εξίσωση (2), η τιμή του $f_{k+1}(y)$ εξαρτάται μόνο από τις τιμές $f_k(y)$ για $0 \leq y \leq W$. Επίσης για να υπολογίσουμε το $f_{k+1}(y)$ είναι απαραίτητο να γνωρίζουμε την τιμή του $f_k(y)$ για όλα τα y με $0 \leq y \leq W$. Τέλος για κάθε y με $0 \leq y \leq W$, ορίζουμε και το βοηθητικό διάνυσμα x_k^y ως εξής:

$$x_{k+1}^y = \begin{cases} 1 \\ 0 \quad f_{k+1}(y) = f_k(y) \end{cases} \quad (3)$$

Κάθε ένα από τα διανύσματα αυτά μας δίνει, τα αντικείμενα που συμμετέχουν στην λύση του υποπροβλήματος $P_k(y)$. Με βάση αυτές τις παρατηρήσεις προκύπτει ο αλγόριθμος για την επίλυση του προβλήματος του σακιδίου.

```
Dynamic-Knapsack(A,C,W)
1. n = length(C)
2. for y = 0 to W do
3.     if a1 > y then
4.          f1(y) = 0
5.     else
6.          f1(y) = c1
7.     end if
8. end for
9. for k = 1 to n – 1 do
10.    for y = 0 to W do
11.        if ak+1 > y then
12.            fk+1(y) = fk(y)
13.            x^y_{k+1} = 0
14.        else if fk(y) > fk(y – ak+1) + ck+1
```

```
15.            f_{k+1(}y) = f_k(y)
16.            x_{k+1}^y = 0
17.        else
18.            f_{k+1(}y) = f_k(y − a_{k+1}) + c_{k+1}
19.            x_{k+1}^y = 1
20.        end if
21.    end for
22. end for
23. return f, x
```

Μπορούμε εύκολα να παρατηρήσουμε ότι η πολυπλοκότητα του αλγορίθμου είναι της τάξης O(nW), η οποία εξαρτάται πολυωνυμικά από το n αλλά εξαρτάται και από το μέγιστο δυνατό βάρος W το οποίο μπορεί να είναι εκθετική συνάρτηση του n. Από αυτό προκύπτει ότι η πολυπλοκότητα του αλγορίθμου δεν είναι αυστηρά πολυωνυμική αλλά ψευδοπολυωνυμική.

Στον πίνακα φαίνεται η εκτέλεση του αλγορίθμου όταν μας δίδονται 6 αντικείμενα με βάρη $a_1 = 9$, $a_2 = 8$, $a_3 = 6$, $a_4 = 5$, $a_5 = 4$, $a_6 = 1$ και αξίες $c_1 = 20$, $c_2 = 16$, $c_3 = 11$, $c_4 = 9$, $c_5 = 7$, $c_6 = 1$ και το βάρος του σακιδίου είναι W = 12. Στο κελί (k, y) έχει σημειωθεί το βέλτιστο κέρδος από τον συνδυασμό των αντικειμένων 1, . . . , k με βάρος το πολύ y ($f_k(y)$) και σε παρένθεση αν χρησιμοποιείται το αντικείμενο k(x_k^y).

Πίνακας 4: Παράδειγμα προβλήματος σακιδίου. Στο κάθε κελί σημειώνεται το βέλτιστο κέρδος και στην παρένθεση το αντικείμενο κ

	1	2	3	4	5	6	7	8	9	10	11	12
1	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	20(1)	20(1)	20(1)	20(1)
2	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	16(1)	20(0)	20(0)	20(0)	20(0)
3	0(0)	0(0)	0(0)	0(0)	0(0)	11(1)	11(1)	16(0)	20(0)	20(0)	20(0)	20(0)
4	0(0)	0(0)	0(0)	0(0)	9(1)	11(0)	11(0)	16(0)	20(0)	20(0)	20(0)	20(0)
5	0(0)	0(0)	0(0)	7(1)	9(0)	11(0)	11(0)	16(0)	20(0)	20(0)	20(0)	23(1)
6	1(1)	1(1)	1(1)	7(0)	9(0)	11(0)	12(1)	16(0)	20(0)	21(1)	21(1)	23(0)

- Βήμα 4: Κατασκευή της βέλτιστης λύσης

Ολοκληρώνουμε την μελέτη μας πάνω στο πρόβλημα του σακιδίου με την κατασκευή της βέλτιστης λύσης. Καθένα από τα διανύσματα x_k^y έχει τιμές 1 ή 0 ανάλογα με το αν το στοιχείο x_k συμμετέχει ή όχι, στη λύση του υποπροβλήματος $P_k(y)$.

Μια γρήγορη μέθοδος για την εξαγωγή της λύσης είναι η παρακάτω:

```
Construct-Knapsack(x)
1. τύπωσε: αξία βέλτιστης λύσης: f_n(W)
2. τύπωσε 'Αντικείμενα στην λύση: '
3. while W > 0 do
4.     if x_k^W = 1 then
5.         τύπωσε: k
6.         W = W – a_k
7.     end if
8.     k = k – 1
9. end while
```

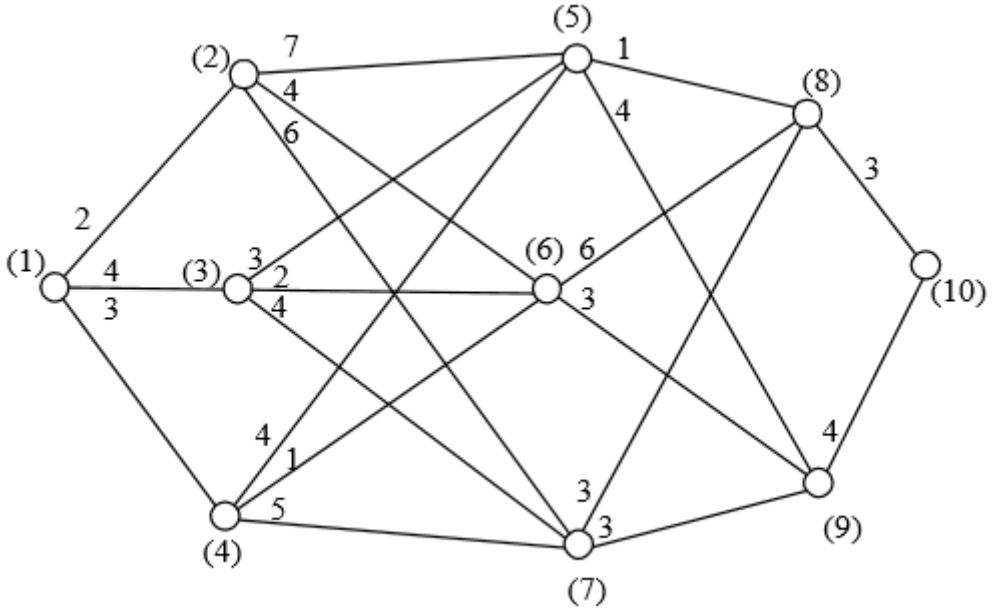
Ο χρόνος εκτέλεσης του αλγορίθμου είναι O(n) και η εκτέλεσή του φαίνεται στον πίνακα 5. Με έντονα γράμματα έχουν σημειωθεί οι θέσεις του πίνακα που περνάει ο αλγόριθμος ώστε να τυπώσει την βέλτιστη λύση, που είναι η επιλογή των αντικειμένων 2, 5 με συνολικό κέρδος 23.

Πίνακας 5: Παράδειγμα υπολογισμού της βέλτιστης λύσης

	1	2	3	4	5	6	7	8	9	10	11	12
1	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	20(1)	20(1)	20(1)	20(1)
2	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	**16(1)**	20(0)	20(0)	20(0)	20(0)
3	0(0)	0(0)	0(0)	0(0)	0(0)	11(1)	11(1)	**16(0)**	20(0)	20(0)	20(0)	20(0)
4	0(0)	0(0)	0(0)	0(0)	9(1)	11(0)	11(0)	**16(0)**	20(0)	20(0)	20(0)	20(0)
5	0(0)	0(0)	0(0)	7(1)	9(0)	11(0)	11(0)	16(0)	20(0)	20(0)	20(0)	**23(1)**
6	1(1)	1(1)	1(1)	7(0)	9(0)	11(0)	12(1)	16(0)	20(0)	21(1)	21(1)	**23(0)**

5.1.3 Παράδειγμα δυναμικού προγραμματισμού

Στο δίκτυο του Σχήματος 10 ζητείται να βρεθεί ο συντομότερος δρόμος από τον κόμβο (1) στον κόμβο (10) του δικτύου. Οι αριθμοί στην αρχή κάθε κλάδου παριστάνουν τις επιμέρους αποστάσεις μεταξύ των κόμβων. Αντί να υπολογίσουμε το συνολικό μήκος των 3x3x2=18 δυνατών διαδρομών από το (1) στο (10), ένας πιο αποτελεσματικός τρόπος να λύσουμε το πρόβλημα είναι να το σπάσουμε σε μικρότερα προβλήματα τα οποία λύνουμε διαδοχικά και συνδέουμε τις λύσεις τους. Δηλαδή αντιμετωπίζουμε το πρόβλημα σε χωριστά βήματα, όπου καθένα αποτελεί την επίλυση ενός επιμέρους προβλήματος που η λύση του δίνει πληροφορίες για την επίλυση του επόμενου προβλήματος, μέχρις ότου φτάσουμε στο αρχικό πρόβλημα.



Σχήμα 10

Έτσι αρχίζοντας από το τέλος, βρίσκουμε ότι η ελάχιστη διαδρομή από το (8) στο (10) είναι 3 και η ελάχιστη διαδρομή από το (9) στο (10) είναι 4. Οπισθοχωρώντας ένα βήμα ακόμη, βρίσκουμε την ελάχιστη απόσταση από το (5) στο (10) χρησιμοποιώντας τα προηγούμενα αποτελέσματα.

Η ελάχιστη απόσταση από το (5) στο (10) είναι: min {1+3, 4+4} = 4 και η διαδρομή είναι μέσω του (8).

Μπορούμε να επεκταθούμε προς τα πίσω, **βήμα-βήμα**. Έτσι έχουμε:

- Η ελάχιστη απόσταση από το (6) στο (10) είναι 7 (μέσω (9)).
- Η ελάχιστη απόσταση από το (7) στο (10) είναι 6 (μέσω (8)).

Βήμα 3 από το τέλος:

- Η ελάχιστη απόσταση από το (2) στο (10) είναι 11 (μέσω (5) ή (6)).
- Η ελάχιστη απόσταση από το (3) στο (1) είναι 7 (μέσω (5)).
- Η ελάχιστη απόσταση από το (4) στο (10) είναι 8 (μέσω (5) ή (6)).

Βήμα 4 από το τέλος:

- Η ελάχιστη απόσταση από το (1) στο (10) είναι 11 (μέσω (3) ή (4)).

Από τα αποτελέσματα αυτά και μόνο μπορούμε να προσδιορίσουμε τη βέλτιστη διαδρομή: Από το (1) η πρώτη απόφαση μας φέρνει στο (3) ή στο (4). Εάν πάμε στο (3), τότε το επόμενο βήμα είναι στο (5), ύστερα στο (8) και τέλος στο (10). Εάν πάμε στο (4), τότε το επόμενο βήμα είναι στο (5) ή στο (6). Εάν πάμε στο (5), τότε τα επόμενα βήματα είναι (8), (10). Εάν πάμε στο (6), τότε προχωρούμε μέσω (9) στο (10). Έτσι υπάρχουν τρεις βέλτιστες διαδρομές:

(1) - (3) - (5) - (8) - (10)

(1) - (4) - (5) - (8) - (10)

(1) - (4) - (6) - (9) - (10)

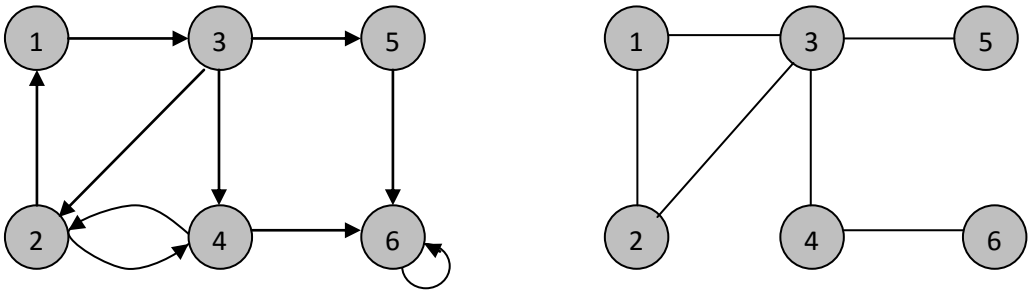
Και οι τρεις έχουν φυσικά το ίδιο συνολικό μήκος 11.

5.2 Προβλήματα από τη θεωρία γραφημάτων

Η έννοια του γράφου είναι μια συνδυαστική δομή, η οποία μας επιτρέπει να αναπαραστήσουμε πολυάριθμες καταστάσεις, που συναντάμε σε εφαρμογές, οι οποίες εμπλέκουν τα διακριτά μαθηματικά και απαιτούν μια λύση πληροφορικής. Ένας γράφος είναι μια μαθηματική οντότητα και αποτελεί μια δομή δεδομένων ισχυρή για την πληροφορική. Οι γράφοι αποτελούν ένα ισχυρό εργαλείο για την αναπαράσταση προβλημάτων της καθημερινής ζωής, που θα ήταν δύσκολα προσεγγίσιμα με τις κλασικές μεθόδους των μαθηματικών. Οι γράφοι χρησιμοποιούνται ιδιαίτερα για την περιγραφή της δομής ενός πολύπλοκου συνόλου και για τις σχέσεις και τις εξαρτήσεις ανάμεσα στα στοιχεία του συνόλου.

Ένα γράφημα G = (V, E) αποτελείται από δύο σύνολα. Ένα πεπερασμένο σύνολο V από στοιχεία που ονομάζονται κορυφές ή κόμβοι (nodes ή vertices) και ένα πεπερασμένο επίσης σύνολο E ακμών(edges).

Οι ακμές ενός γραφήματος μπορεί να είναι κατευθυνόμενες, οπότε το γράφημα ονομάζεται κατευθυντικό ή μη κατευθυνόμενες, οπότε το γράφημα ονομάζεται μη κατευθυντικό.



(α) (β)

Σχήμα 11: Κατευθυντικό γράφημα (α) και μη κατευθυντικό γράφημα (β)

Για δύο κορυφές Α, Β λέμε ότι είναι γειτονικές, αν υπάρχει ακμή e που τις συνδέει. Στην περίπτωση αυτή τα Α, Β ονομάζονται άκρα της e και η e προσπίπτει σε αυτά. Στις ακμές ενός γραφήματος *G*(*V*, *E*) μπορούμε να αντιστοιχίσουμε βάρη w(e). Σε αυτή την περίπτωση, το γράφημα ονομάζεται βεβαρημένο γράφημα και μερικές φορές συμβολίζεται με *G*(*V*, *E*, *w*). Επίσης, σε μερικές περιπτώσεις τα βάρη ερμηνεύονται σαν μήκη των ακμών. Το βάρος του γράφου είναι το άθροισμα των βαρών των ακμών του. Αραιός ονομάζεται ο γράφος που ο αριθμός των ακμών του είναι της τάξης O(n), όπου n είναι ο αριθμός κορυφών του, διαφορετικά λέγεται πυκνός.

Συνήθως οι κορυφές συμβολίζονται σαν $V=\{v_1,v_2,\ldots,v_n\}$ και οι ακμές σαν $E=\{e_1,e_2,\ldots,e_m\}$. Εάν οι κορυφές v_i και v_j συσχετίζονται με την ακμή e_k τότε ονομάζονται καταληκτικές κορυφές αυτής της ακμής και συμβολίζεται σαν $e_k=(v_i,v_j)$. Όλες οι ακμές που έχουν τις ίδιες καταληκτικές κορυφές ονομάζονται παράλληλες ακμές. Επιπλέον, εάν $e_k=(v_i,v_i)$τότε η ακμή e_k αποτελεί έναν αυτοβρόγχο της κορυφής v_i .Ένας γράφος που δεν έχει παράλληλες ακμές αλλά ούτε και αυτο-βρόγχους ονομάζεται απλός γράφος. Ένας γράφος χωρίς ακμές ονομάζεται άδειος γράφος ενώ χωρίς κορυφές (οπότε και χωρίς ακμές) ονομάζεται κενός γράφος. Ένα γράφημα n κορυφών ονομάζεται πλήρες όταν όλα τα ζεύγη κορυφών συνδέονται με μία ακμή, οπότε έχουμε n(n-1)/2 ακμές. Ένα γράφημα $\overline{G}$ είναι συμπληρωματικό του γραφήματος G, όταν έχει το ίδιο σύνολο κορυφών με το G και όσες ακμές δεν υπάρχουν σε αυτό.

Ο αριθμός των ακμών που έχει μία κορυφή ονομάζεται *βαθμός της κορυφής* και συμβολίζεται σαν $d(v_i)$. Εάν ισχύει $d(v_i)=0$ τότε αυτή η κορυφή ονομάζεται *απομονωμένη κορυφή*. Στα κατευθυντικά γραφήματα διακρίνουμε τον αριθμό των ακμών που ξεκινούν από μία κορυφή *v*, ο οποίος ονομάζεται *βαθμός εξόδου* της *v*, και τον αριθμό των ακμών που καταλήγουν σε μία κορυφή v, ο οποίος ονομάζεται *βαθμός εισόδου* της v.

Μια ακολουθία κορυφών $(v_1,v_2,\ldots,v_n)$, όπου η ακμή $(v_{i-1},v_i) \in E$ για κάθε i=1,…k, αποτελεί μια διαδρομή μήκους k. Η διαδρομή είναι μια ακολουθία κορυφών που συνδέονται με συνεχόμενες ακμές. Τα μήκος της διαδρομής είναι ίσο με τον αριθμό των ακμών της. Μια διαδρομή ονομάζεται μονοκονδυλιά όταν όλες οι ακμές της είναι διαφορετικές και μονοπάτι όταν όλες οι κορυφές της είναι διαφορετικές. Χρησιμοποιούμε τον όρο απλό μονοπάτι για μια διαδρομή με διαφορετικές κορυφές και άρα ακμές. Μια κορυφή α είναι προσπελάσιμη από μια κορυφή b αν υπάρχει μονοπάτι από την α στη b.

Όταν η αρχική και η τελική κορυφή μιας διαδρομής συμπίπτουν, έχουμε μια κλειστή διαδρομή. Μια κλειστή διαδρομή ονομάζεται κύκλωμα, όταν όλες οι ακμές της είναι διαφορετικές και κύκλος όταν όλες οι κορυφές της είναι διαφορετικές. Δηλαδή το κύκλωμα είναι μία κλειστή μονοκονδυλιά και ο κύκλος είναι ένα κλειστό μονοπάτι. Χρησιμοποιούμε τον όρο απλός κύκλος για μια κλειστή διαδρομή με διαφορετικές κορυφές.

Ο αριθμός των ακμών σε ένα μονοπάτι ορίζουν το μήκος του μονοπατιού. *Απόσταση* μεταξύ δύο κορυφών v_1 και v_2, συμβολίζεται σαν $d(v_1, v_2)$ και ορίζεται σαν το μήκος του συντομότερου μονοπατιού μεταξύ των δύο αυτών κορυφών. Εάν δεν υπάρχει τέτοιο μονοπάτι, αυτή η απόσταση ορίζεται σαν άπειρη. *Διάμετρος* ενός γράφου *G*, ονομάζεται η μέγιστη απόσταση μεταξύ δύο οποιονδήποτε κορυφών αυτού του γράφου και συμβολίζεται σαν *diam*(*G*) .

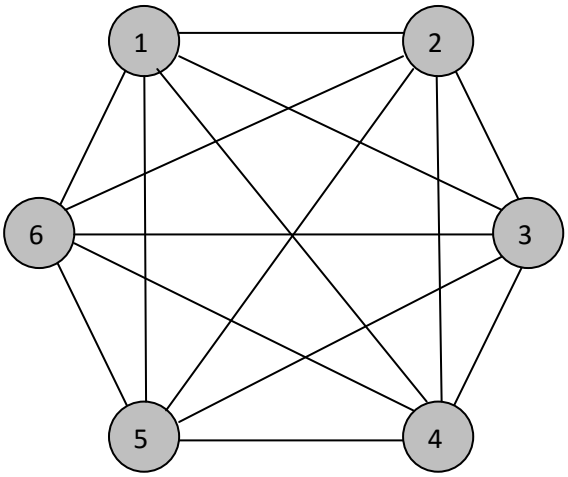
Ένα γράφημα G’(V΄,E΄), ονομάζεται *υπογράφημα* του γραφήματος G(V,E), αν οι κορυφές και οι ακμές του G’ περιέχονται στις κορυφές και στις ακμές του G, Δηλαδή αν $V' \subseteq V$ και $E' \subseteq E$.

Ένα μη κατευθυντικό γράφημα ονομάζεται *συνεκτικό*, όταν για κάθε ζευγάρι κορυφών, υπάρχει ένα μονοπάτι μεταξύ τους. Όταν ένα γράφημα G(V, E) δεν είναι συνεκτικό, μπορεί να διαιρεθεί σε *συνεκτικές συνιστώσες*, οι οποίες είναι συνεκτικά υπογραφήματα του G, που ορίζονται από μία διαμέριση των κορυφών του V. Για παράδειγμα, το γράφημα του Σχήματος 11(β) είναι συνεκτικό.

Οι συνεκτικές συνιστώσες ενός γραφήματος είναι οι κλάσεις ισοδυναμίας στις οποίες διαιρείται το σύνολο των κορυφών από τη σχέση ισοδυναμίας «υπάρχει μονοπάτι μεταξύ των κορυφών v_1 και v_2».

Ένα κατευθυντικό γράφημα ονομάζεται *ισχυρά συνεκτικό*, όταν οποιεσδήποτε δύο κορυφές v_1 και v_2 είναι αμοιβαία προσπελάσιμες, δηλαδή τόσο η v_1 είναι προσπελάσιμη από τη v_2, όσο και η v_2 είναι προσπελάσιμη από τη v_1. Όταν ένα κατευθυντικό γράφημα *G*(*V*, *E*) δεν είναι ισχυρά συνεκτικό, τότε μπορεί να διαιρεθεί σε *ισχυρά συνεκτικές συνιστώσες*, οι οποίες είναι ισχυρά συνεκτικά υπογραφήματα του *G*, που ορίζονται από μία διαμέριση των κορυφών του V. Για παράδειγμα, το γράφημα του Σχήματος 11(α) αποτελείται από τρεις ισχυρά συνεκτικές συνιστώσες που ορίζονται από τα σύνολα κορυφών {1, 2, 4, 5}, {3}, και {6}. Οι ισχυρά συνεκτικές συνιστώσες ενός γραφήματος είναι οι κλάσεις ισοδυναμίας στις οποίες διαιρείται το σύνολο των κορυφών από τη σχέση «οι κορυφές v_1 και v_2 είναι αμοιβαία προσπελάσιμες».

Ο κύκλος Euler, είναι μια κλειστή μονοκονδυλιά που διέρχεται από κάθε ακμή μία φορά και από κάθε κορυφή τουλάχιστον μία φορά. Ένα συνεκτικό, μη κατευθυντικό γράφημα έχει κύκλο Euler, αν όλες οι κορυφές έχουν άρτιο βαθμό.



Σχήμα 12: κύκλος Euler

Ένα γράφημα G =(V,E) ονομάζεται κανονικό όταν όλοι οι κόμβοι έχουν τον ίδιο βαθμό κ, δηλ. $\forall \ v \in V \ d(v) = κ$. Ένας γράφος ονομάζεται επίπεδος αν μπορούμε να τον σχεδιάσουμε στο επίπεδο, χωρίς τμήση των πλευρών του.

Όταν δίνεται κάποιο γράφημα G, μπορούμε να πάρουμε ένα νέο γράφημα αν διαιρέσουμε μία ακμή του G χωρίς επιπλέον κορυφές. Λέμε ότι δύο γραφήματα G και *G'* είναι *ομοιόμορφα* αν μπορούν να ληφθούν από το ίδιο γράφημα ή ισόμορφο γράφημα με την μέθοδο αυτή.

Δύο γράφοι $G_1 = (V_1,E_1)$ και $G_2 = (V_2,E_2)$ λέγονται ισομορφικοί αν υπάρχει μια αμφιμονοσήμαντη αντιστοιχία f : V1 → V2 έτσι ώστε $[f(v_1), f(v_2)] \in E_2 \Leftrightarrow (v_1, v_2) \in E_1$. Δηλαδή αν οι κορυφές v_1, v_2 ενώνονται με μια ακμή στον γράφο G_1 τότε οι αντίστοιχες κορυφές $f(v_1)$, $f(v_2)$ θα είναι γειτονικές στον γράφο G_2. Δύο ισομορφικοί γράφοι λοιπόν μπορεί να έχουν διαφορετική μορφή, αλλά έχουν τις ίδιες βασικές δομικές ιδιότητες.

Ένα γράφημα G είναι *διμερές* αν οι κορυφές του V μπορούν να επιμεριστούν σε δύο υποσύνολα M και N έτσι ώστε κάθε ακμή του G να συνδέει μια κορυφή του M με μία κορυφή του N. Αυτό το γράφημα συμβολίζεται με $K_{m,n}$ όπου m είναι το πλήθος των κορυφών του M και n είναι το πλήθος των κορυφών του N, και για τυποποίηση υποθέτουμε ότι $m \leq n$. Είναι φανερό ότι το γράφημα $K_{m,n}$ έχει m*n ακμές .

Μία από τις σημαντικότερες κατηγορίες γραφημάτων είναι τα δέντρα. Ένα γράφημα που δεν περιέχει κύκλους ονομάζεται δάσος, ενώ ένα συνεκτικό γράφημα που δεν περιέχει κύκλους ονομάζεται δέντρο. Οι σημαντικότερες ιδιότητες των δέντρων συνοψίζονται στο παρακάτω θεώρημα:

Θεώρημα: Έστω G(V, E) ένα μη κατευθυντικό γράφημα. Οι ακόλουθες προτάσεις είναι ισοδύναμες:

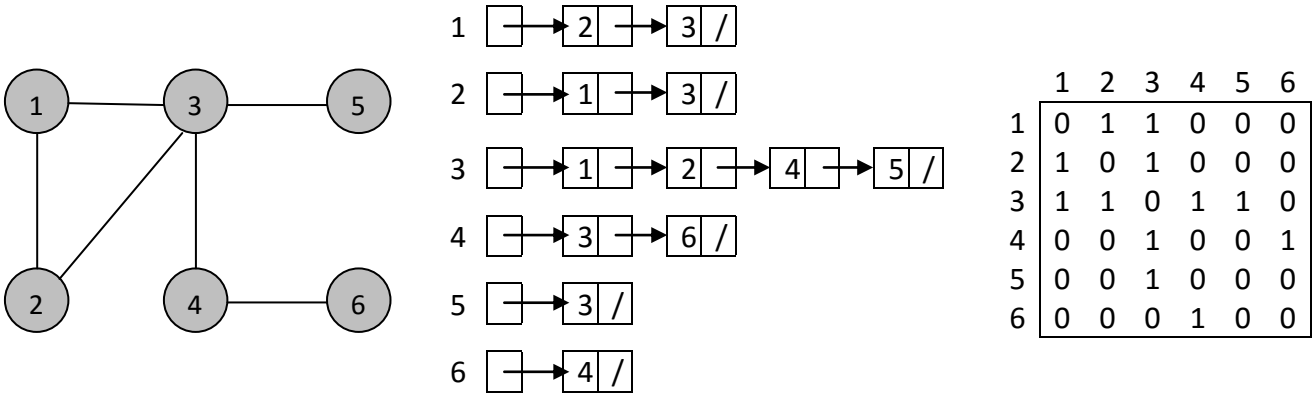
- Το γράφημα *G* είναι δέντρο.
- Οποιεσδήποτε δύο κορυφές του G ενώνονται μέσω ενός μοναδικού απλού μονοπατιού.
- Το γράφημα G είναι συνεκτικό, αλλά αν μία οποιαδήποτε ακμή του *E* αφαιρεθεί, το γράφημα που θα προκύψει δεν θα είναι συνεκτικό.
- Το γράφημα G είναι συνεκτικό και έχει ακριβώς |V| – 1 ακμές.
- Το γράφημα G δεν περιέχει κύκλους και έχει ακριβώς |V| – 1 ακμές.
- Το γράφημα G δεν περιέχει κύκλους, αλλά αν μία οποιαδήποτε ακμή προστεθεί στο E, το γράφημα που προκύπτει περιέχει κύκλο.

5.2.1 Αναπαράσταση γραφημάτων

Για την επεξεργασία ενός γραφήματος από έναν αλγόριθμο είναι απαραίτητη η αναπαράσταση του γραφήματος μέσω μιας κατάλληλης δομής δεδομένων. Οι συνηθέστεροι τρόποι αναπαράστασης ενός γραφήματος είναι μέσω του ονομαζόμενου *πίνακα γειτνίασης* και της λίστας γειτνίασης.

Η λίστα γειτνίασης ενός γραφήματος G(V, E) με n κορυφές και m ακμές, αποτελείται από τη λίστα των γειτονικών κορυφών για κάθε κορυφή $v \in V$. Οι δείκτες στα πρώτα στοιχεία αυτών των λιστών αποθηκεύονται σε έναν πίνακα L με n θέσεις. Συγκεκριμένα ο δείκτης στο πρώτο στοιχείο της λίστας των γειτονικών κορυφών της $v \in V$ βρίσκεται στη θέση L[v]. Το μέγεθος της λίστας γειτόνων της v είναι ίσο με το βαθμό της. Επομένως η αναπαράσταση του G με λίστα γειτνίασης απαιτεί Θ(n+m) θέσεις μνήμης.

Συμβολίζουμε με A τον πίνακα γειτνίασης του γραφήματος G(V, E) που αποτελείται από n x n = n^2 στοιχεία, εάν υποθέσουμε ότι έχουμε ένα γράφο n κορυφών. Το στοιχείο A[i, j] = 1, εάν υπάρχει η ακμή (v_i, v_j) ενώ αλλιώς είναι 0. Προφανώς αυτός ο πίνακας είναι συμμετρικός. Το μόνο μειονέκτημα αυτής της αναπαράστασης είναι ότι απαιτεί $\Omega(n^2)$ αποθηκευτικό χώρο ενώ συνήθως ο αριθμός των ακμών είναι μικρότερος του n^2 και φυσικά η προσπέλαση του πίνακα απαιτεί $O(n^2)$ χρόνο. Όταν το γράφημα είναι αραιό, δηλαδή έχει αριθμό ακμών $o(n^2)$, η αναπαράσταση με λίστα γειτνίασης απαιτεί ασυμπτωτικά λιγότερες θέσεις μνήμης από την αναπαράσταση με πίνακα γειτνίασης. Ενώ όταν το γράφημα είναι πυκνό, δηλαδή έχει αριθμό $\Theta(n^2)$, οι απαιτήσεις των δύο αναπαραστάσεων σε θέσεις μνήμης ταυτίζονται ως προς την ασυμπτωτική τους συμπεριφορά. Όταν το γράφημα έχει βάρη το στοιχείο A[i, j] ισούται με το βάρος w(v_i, v_j) της αντίστοιχης ακμής.



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	0	0	0
3	1	1	0	1	1	0
4	0	0	1	0	0	1
5	0	0	1	0	0	0
6	0	0	0	1	0	0

Σχήμα 13: Αναπαράσταση μη κατευθυντικού γραφήματος με λίστα και πίνακα γειτνίασης

5.2.2 Αναζήτηση κατά πλάτος (Breadth-First Search (BFS))

Στην αναζήτηση κατά πλάτος, από κάθε κόμβο v που επισκεπτόμαστε, αναζητούμε κόμβους όσο το δυνατόν κατά πλάτος. Δηλαδή αμέσως μετά την επίσκεψη του κόμβου v επισκεπτόμαστε όλους τους γειτονικούς του κόμβους που βρίσκονται σε απόσταση k, έπειτα αυτούς που βρίσκονται σε απόσταση k+1 κοκ. Έτσι λοιπόν κατά τη διάρκεια της διαδικασίας αυτής μπορούμε να ξεχωρίσουμε τρεις κατηγορίες κορυφών.

Τις ανεξερεύνητες κορυφές που δεν έχουμε επισκεφτεί ακόμα, τις υπο-εξέταση κορυφές που έχουμε επισκεφτεί αλλά απομένει η εξερεύνηση των γειτονικών τους και τις εξερευνημένες κορυφές, τις οποίες έχουμε επισκεφτεί και έχουμε ολοκληρώσει την εξερεύνηση των γειτονικών τους.

Για να εξασφαλιστεί η κατά πλάτος εξερεύνηση των κορυφών, η σειρά με την οποία η υπο-εξέταση κορυφές γίνονται εξερευνημένες, πρέπει να είναι ίδια με τη σειρά που οι ανεξερεύνητες κορυφές γίνονται υπο-εξέταση. Για το λόγο αυτό ο αλγόριθμος διατηρεί μια First-In-First-Out (FIFO) ουρά από υπο-εξέταση κορυφές. Αρχικά η μόνη υπο-εξέταση κορυφή είναι η s. Η αναζήτηση κατά πλάτος εξάγει την πρώτη κορυφή της ουράς, εξερευνά τις γειτονικές κορυφές και θεωρεί την κορυφή εξερευνημένη. Σαν αποτέλεσμα αυτής της διαδικασίας, κάποιες κορυφές γίνονται υπο-εξέταση και προστίθενται στο τέλος της ουράς. Χρησιμοποιούνται δύο βοηθητικοί πίνακες, ο m και ο p. Για κάθε κορυφή v το m[v] περιέχει την τρέχουσα κατάσταση της v σε σχέση με την αναζήτηση κατά πλάτος και το p[v] την κορυφή από την οποία επισκέφτηκε το v για πρώτη φορά. Όσο η v παραμένει ανεξερεύνητη το p[v]=NULL. Οι ανεξερεύνητες κορυφές χαρακτηρίζονται με A, οι υπο-εξέταση με Y και οι εξερευνημένες με E.

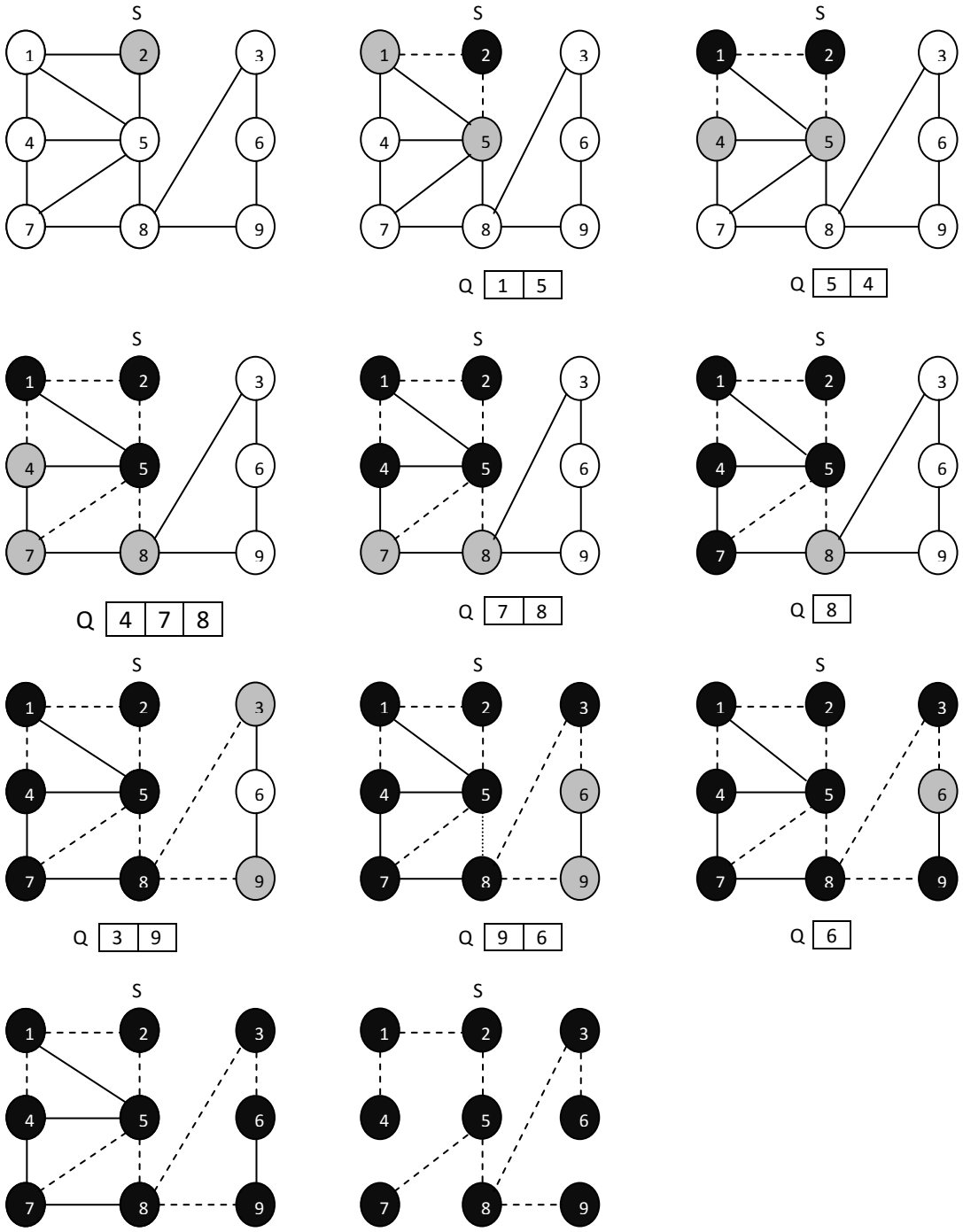
Παρακάτω παρουσιάζεται ένας αλγόριθμος για την εφαρμογή.

```
BFS(G(V,E),s)
        addToQueue(s); m[s] ← Y; p[s] ← NULL;
        for all v ∈ V \ {s} do
                m[v] ← A; p[v] ← NULL;
        while not emptyQueue() do
                u← extractFromQueue(); m[u] ← E;
                 for all v ∈ L[u] do
                        if m[v] = A then
                           addToQueue(v); m[v] ← Y; p[v] ← u;
```

Στο σχήμα 14 παρουσιάζεται ένα παράδειγμα εφαρμογής της αναζήτησης κατά πλάτος, όπου οι ανεξερεύνητες κορυφές συμβολίζονται με λευκό χρώμα, οι υπο-εξέταση με γκρι και οι εξερευνημένες με μαύρο. Παρατηρούμε ότι η κάθε κορυφή αλλάζει δύο φορές χαρακτηρισμό, από A σε Y όταν προστίθεται στο τέλος της ουράς και από Y σε E όταν εξάγεται από την αρχή της ουράς.

Η εισαγωγή στο τέλος και η εξαγωγή από την αρχή μιας FIFO ουράς υλοποιείται εύκολα σε σταθερό χρόνο. Ο συνολικός χρόνος για αυτές τις λειτουργίες είναι O(n). Η λίστα των γειτόνων μιας κορυφής εξετάζεται όταν η κορυφή γίνεται από υπο-εξέταση εξερευνημένη. Το άθροισμα των γειτόνων όλων των κορυφών είναι Θ(m) και ο αλγόριθμος αφιερώνει σταθερό χρόνο για κάθε γειτονική κορυφή που δεν χαρακτηρίζεται ανεξερεύνητη. Ο αντίστοιχος χρόνος είναι O(m).

Επομένως ο συνολικός χρόνος εκτέλεσης είναι O(n+m), δηλαδή γραμμικός στο μέγεθος της λίστας γειτνίασης. Εκτός από την αρχική κορυφή s, για κάθε κορυφή v που είναι εξερευνημένη ή υπο-εξέταση ισχύει p[v] ≠ NULL. Στο τέλος του αλγορίθμου όλες οι κορυφές είναι είτε εξερευνημένες είτε ανεξερεύνητες, γιατί κάθε υπο-εξέταση κορυφή εξάγεται από την ουρά πριν την ολοκλήρωση του αλγορίθμου και γίνεται εξερευνημένη. Αν όλες οι κορυφές είναι προσπελάσιμες από την s και το γράφημα είναι μη κατευθυντικό και συνεκτικό, ο αλγόριθμος τερματίζει με όλες τις κορυφές εξερευνημένες.



Σχήμα 14: Αναζήτηση κατά πλάτος

Η αναζήτηση κατά πλάτος παράγει ένα δέντρο το οποίο αποτελείται από όλους τους προσπελάσιμους από τον s κόμβους, έτσι ώστε για κάθε κόμβο v του δέντρου ισχύει ότι το απλό μονοπάτι από τον s στον v είναι επίσης ένα ελάχιστο μονοπάτι. Η αναζήτηση αυτή μπορεί να εφαρμοστεί και σε κατευθυνόμενα και σε μη κατευθυνόμενα γραφήματα.

5.2.3 Αναζήτηση κατά βάθος (Depth-First Search (DFS))

Η αναζήτησης κατά βάθος προχωράει όσο το δυνατόν μακρύτερα από την αρχική κορυφή s. Χρησιμοποιεί όλες τις εξερχόμενες ακμές της κορυφής v που επισκέφτηκε τελευταία. Όταν η τελευταία κορυφή v δεν έχει άλλες αχρησιμοποίητες ακμές, επιστρέφει στην κορυφή που επισκέφθηκε αμέσως προηγουμένως, για να χρησιμοποιήσει τις ακμές της. Η διαδικασία αυτή συνεχίζεται μέχρι να ανακαλυφθούν όλες οι κορυφές που είναι προσπελάσιμες από την αρχική κορυφή s. Αν σε αυτό το σημείο υπάρχουν κορυφές οι οποίες παραμένουν ανεξερεύνητες, μία από αυτές επιλέγεται σαν αρχική, και η αναζήτηση συνεχίζεται. Η αναζήτηση κατά βάθος ολοκληρώνεται όταν έχει επισκεφθεί όλες τις κορυφές του γραφήματος.

Διακρίνονται τρία είδη κορυφών, όπως και στην αναζήτηση κατά πλάτος. Στην αρχή όλες οι κορυφές είναι ανεξερεύνητες. Την πρώτη φορά που γίνεται επίσκεψη σε μια ανεξερεύνητη κορυφή v, αυτή γίνεται υπο-εξέταση και η αναζήτηση κατά βάθος εφαρμόζεται αναδρομικά με αρχική κορυφή την v. Η κορυφή v θεωρείται εξερευνημένη όταν ολοκληρωθεί η αναδρομική κλήση.

Για την υλοποίηση του αλγόριθμου χρησιμοποιείται μια βοηθητική μεταβλητή t και τρεις βοηθητικοί πίνακες m, p, d. Η τρέχουσα τιμή της μεταβλητής t αντιστοιχεί στον αριθμό των κορυφών που έχουν επισκεφτεί. Όταν η κορυφή v αλλάζει χαρακτηρισμό από ανεξερεύνητη σε υπο-εξέταση, το t αυξάνεται κατά 1 και το d[v] τίθεται στην τρέχουσα τιμή της t. Έτσι το d[v] δείχνει τη σειρά με την οποία η αναζήτηση κατά βάθος επισκέφτηκε την κορυφή v.

Το m[v] περιέχει την τρέχουσα κατάσταση της v σε σχέση με την αναζήτηση κατά βάθος και το p[v] την κορυφή από την οποία η αναζήτηση κατά βάθος επισκέφτηκε τη v για πρώτη φορά.

```
DFS_Init(G(V,E))
     t← 0;
     for all v ∈ V do
          m[v]← A; p[v]← NULL;
     for all v ∈ V do
          if m[v]=A then DFS(v);
```

```
DFS(v)
   m[v] ← Y; d[v] ← ++ t;
   for all u ∈ L[v] do
       if m[u] = A then
            p[u]← v; DFS(u);
   m[v]← E; f[v] ← ++ t;
```

Στον παραπάνω αλγόριθμο η διαδικασία DFS_Init αρχικοποιεί τις μεταβλητές που χρησιμοποιούνται στον αλγόριθμο και ξεκινάει την αναδρομική διαδικασία DFS. Η διαδικασία DFS καλείται μία φορά για κάθε κορυφή, αφού η κλήση DFS(v) αλλάζει το χαρακτηρισμό της v από A σε Y και στη συνέχεια σε E. Η κλήση DFS(v) ελέγχει όλους τους γείτονες της v για κορυφές που παραμένουν ανεξερεύνητες. Επομένως κάθε ακμή ελέγχεται μία φορά και όταν πρόκειται για μη κατευθυντικό γράφημα δύο. Ο συνολικός χρόνος εκτέλεσης της αναζήτησης κατά βάθος είναι O(n+m), δηλαδή γραμμικός στο μέγεθος της λίστας γειτνίασης.

Η DFS παράγει ένα δάσος από δέντρα το οποίο αποτελείται από ένα ή περισσότερα δέντρα κατά βάθος αναζήτησης. Αυτό συμβαίνει γιατί η πρώτη επίσκεψη σε κάθε κορυφή v γίνεται από μοναδικό μονοπάτι από την αντίστοιχη αρχική κορυφή προς τη v.

5.2.4 Ελάχιστο συνδετικό δέντρο(Minimum Spanning Tree-MST) και Συντομότερα μονοπάτια

Δύο ακόμη προβλήματα από τη θεωρία γραφημάτων είναι αυτά του ελάχιστου συνδετικού δέντρου και του συντομότερου μονοπατιού. Το ελάχιστο συνδετικό δέντρο ενός γραφήματος G(V,E,w) με βάρη στις ακμές, είναι το συνδετικό δέντρο του G με το ελάχιστο συνολικό βάρος. Το πρόβλημα του υπολογισμού ενός τέτοιου δέντρου αποτελεί ένα τυπικό παράδειγμα προβλήματος συνδυαστικής βελτιστοποίησης. Το πρόβλημα αυτό εφαρμόζεται στο σχεδιασμό δικτύων όταν επιδιώκουμε να ελαχιστοποιήσουμε το κόστος σύνδεσης κάποιων σημείων. Για τον υπολογισμό ενός ελαχίστου συνδετικού δέντρου χρησιμοποιείται ο αλγόριθμος του Kruskal και ο αλγόριθμος του Prim.

Στο πρόβλημα των συντομότερων μονοπατιών, δίνεται ένα γράφημα G(V,E,w) με μήκη στις ακμές και ζητούνται τα συντομότερα μονοπάτια μεταξύ κάποιων κορυφών. Σε μερικές περιπτώσεις το ζητούμενο είναι ο υπολογισμός των αντίστοιχων αποστάσεων και όχι τα ίδια τα μονοπάτια. Το πρόβλημα αυτό έχει πολλές και σημαντικές πρακτικές εφαρμογές και επιλύεται με τον αλγόριθμο του Dijkstra και τον αλγόριθμο των Bellman-Ford.

5.2.5 Αλγόριθμοι γραφημάτων

- Ο αλγόριθμος του **Kruskal** επιλέγει ακμές, απλά προσπαθώντας να αποφύγει τη δημιουργία κύκλων. Αυτός ο τρόπος λειτουργίας έχει ως συνέπεια τη δημιουργία ενός δάσους από δέντρα, τα οποία στη συνέχεια αναπτύσσονται με ακανόνιστο τρόπο, ανάλογα με την εξέλιξη του αλγορίθμου. Ο Kruskal επιστρέφει το ελάχιστο συνδετικό δέντρο που ενώνει όλες τις κορυφές ενός γράφου.

- Ο αλγόριθμος του **Prim** αναπτύσσει σταδιακά το ίδιο το ελάχιστο συνδεδεμένο δέντρο ξεκινώντας από μια τυχαία επιλεγείσα ρίζα. Σε κάθε βήμα προστίθεται ένα νέο κλαδί του δέντρου και ο αλγόριθμος ολοκληρώνεται όταν έχουν επιλεγεί όλες οι κορυφές του γράφου.
- Ο αλγόριθμος του **Dijkstra**, βρίσκει την ελάχιστη διαδρομή μεταξύ δύο κορυφών.
- Ο αλγόριθμος των **Bellman-Ford**, επιλύει το πρόβλημα των ομοαφετηριακών ελαφρύτατων διαδρομών στη γενική περίπτωση όπου οι ακμές ενδέχεται να έχουν αρνητικά βάρη. Εάν υπάρχουν αρνητικά βάρη, ο αλγόριθμος υποδεικνύει ότι το πρόβλημα δεν έχει λύση. Εάν δεν υπάρχουν, υπολογίζει τις ελαφρύτατες διαδρομές και τα βάρη τους.
- Ο αλγόριθμος του **Boruvka,** είναι ένας πολύ απαιτητικός αλγόριθμος διότι βρίσκει το ελάχιστο συνδετικό δέντρο σε ένα γράφημα του οποίου όλες οι ακμές έχουν διαφορετικά βάρη.

Κεφάλαιο 6 - Αποδοτικοί αλγόριθμοι, Αξίωμα Cook-Karp, Κλάσεις P και NP

6.1 Αποδοτικοί αλγόριθμοι

Υπάρχουν προβλήματα τα οποία είναι δύσκολο να επιλυθούν από υπολογιστές ή ακόμα και αδύνατο. Με την μελέτη των προβλημάτων αυτών ασχολείται η θεωρία της υπολογιστικής πολυπλοκότητας. Όταν τα προβλήματα μπορούν να επιλυθούν από τους υπολογιστές, χρησιμοποιούνται υπολογιστικοί πόροι που καθορίζονται από τη θεωρία πολυπλοκότητας. Ο καθορισμός γίνεται εντάσσοντας τα προβλήματα που έχουν παρόμοιες συμπεριφορές σε κοινές ομάδες που ονομάζονται κλάσεις πολυπλοκότητας. Κάθε κλάση περιέχει έναν μικρό σχετικά αριθμό προβλημάτων που αντιπροσωπεύουν όλη την κλάση και ονομάζονται πλήρη. Αν ένα πλήρες πρόβλημα επιλυθεί με καλύτερο τρόπο, τότε η επίλυση αυτή εφαρμόζεται στα προβλήματα όλης της κλάσης στην οποία ανήκει.

Σύμφωνα με τη θεωρία των αλγορίθμων, τα υπολογιστικά προβλήματα που πρέπει να επιλυθούν με κάποιον αλγόριθμο είναι μια κατηγορία πραγματικών προβλημάτων. Ένα πραγματικό πρόβλημα, περιγράφεται μαθηματικά και υλοποιείται από κάποιον αλγόριθμο. Εφόσον οι αλγόριθμοι χρησιμοποιούνται σε πραγματικά προβλήματα, είναι απαραίτητο να υπολογίζεται η τάξη μεγέθους των συναρτήσεων που προσδιορίζουν την ποσότητα των υπολογιστικών πόρων τους. Η θεωρία αλγορίθμων εξετάζει περισσότερο τα προβλήματα βελτιστοποίησης.

Σύμφωνα με τη θεωρία της πολυπλοκότητας, τα υπολογιστικά προβλήματα που μελετώνται είναι μαθηματικές οντότητες. Με βάση αυτή την ανάλυση δεν είναι απαραίτητο να υπολογίζεται η ακριβής τάξη μεγέθους των συναρτήσεων που προσδιορίζουν την ποσότητα των υπολογιστικών πόρων, αλλά η κατηγορία των συναρτήσεων.

Στη θεωρία της πολυπλοκότητας, ένα πρόβλημα αποτελεί μία δυαδική σχέση η οποία περιλαμβάνει όλα τα διαφορετικά ζεύγη των στιγμιότυπων του προβλήματος και των λύσεών τους. Για παράδειγμα, θεωρούμε το πρόβλημα του Συντομότερου Μονοπατιού μεταξύ δύο κορυφών, με στιγμιότυπο το γράφημα $G(V, E)$ και δύο κορυφές $v_1, v_2 \in V$. Κάθε έγκυρο στιγμιότυπο του προβλήματος αυτού, αποτελείται από ένα γράφημα και δύο κορυφές.

Λύση του προβλήματος αποτελεί το συντομότερο μονοπάτι από την κορυφή v_1 στην κορυφή v_2. Ένα στιγμιότυπο προβλήματος μπορεί να έχει περισσότερες από μία λύσεις. Αν αντιμετωπίσουμε το πρόβλημα σαν πρόβλημα βελτιστοποίησης, η βέλτιστη λύση θα είναι το μονοπάτι που έχει το ελάχιστο μήκος από την κορυφή v_1 στην κορυφή v_2.

Μια διαφορετική προσέγγιση του προβλήματος του Συντομότερου Μονοπατιού, είναι η αντιμετώπισή του σαν πρόβλημα απόφασης. Στην περίπτωση αυτή ένα στιγμιότυπο του προβλήματος αποτελείται από ένα γράφημα $G(V, E)$, δύο κορυφές $v_1, v_2 \in V$ και έναν φυσικό αριθμό $x \geq 0$, που εκφράζει το μήκος ανάμεσα στις δύο κορυφές. Το ζητούμενο είναι να βρούμε αν υπάρχει κάποιο μονοπάτι ανάμεσα στην κορυφή v_1 στην κορυφή v_2 που να έχει το πολύ μήκος x. Η λύση-απάντηση στο πρόβλημα αυτό μπορεί να είναι μόνο ένα «ναι» ή ένα «όχι». Τα προβλήματα αυτής της μορφής ονομάζονται προβλήματα απόφασης.

Μπορούμε να περιγράψουμε οποιοδήποτε πρόβλημα βελτιστοποίησης σαν πρόβλημα απόφασης, συμπεριλαμβάνοντας στη λύση του έναν φυσικό αριθμό x που λειτουργεί σαν φράγμα. Όταν έχουμε προβλήματα ελαχιστοποίησης το ερώτημα που τίθεται από το πρόβλημα απόφασης είναι αν υπάρχει αποδεκτή λύση με κόστος το πολύ x. Ενώ όταν έχουμε προβλήματα μεγιστοποίησης εξετάζουμε αν το κόστος είναι τουλάχιστον x. Συνήθως όταν ένα πρόβλημα απόφασης λύνεται αποδοτικά, τότε μπορεί να λυθεί αποδοτικά και το αντίστοιχο πρόβλημα βελτιστοποίησης. Αυτό ισχύει και αντιστρόφως.

6.2 Αξίωμα Cook-Karp

Τα προβλήματα που λύνονται σε υπολογιστή διακρίνονται σε ευεπίλυτα και σε δυσεπίλυτα. Ευεπίλυτα καλούνται τα προβλήματα των οποίων η λύση επιτυγχάνεται με αποδοτικό τρόπο και δυσεπίλυτα αυτά που δεν λύνονται με αποδοτικό τρόπο. Η αποδοτικότητα των αλγορίθμων υπολογίζεται από τους υπολογιστικούς πόρους που απαιτούν. Ευεπίλυτα είναι τα προβλήματα για τα οποία υπάρχει αποδοτικός αλγόριθμος που απαιτεί πολλούς υπολογιστικούς πόρους. Δυσεπίλυτα είναι τα προβλήματα για τα οποία έχει αποδειχτεί ότι δεν υπάρχει αποδοτικός αλγόριθμος.

Σύμφωνα με την αντίληψη που επικρατεί, ευεπίλυτα είναι μόνο τα προβλήματα που η επίλυση τους πραγματοποιείται σε πολυωνυμικό χρόνο. Για να λυθεί ένα πρόβλημα σε πολυωνυμικό χρόνο πρέπει να υπάρχει κάποιος αλγόριθμος με χρόνο εκτέλεσης χειρότερης περίπτωσης $O(n^k)$, όπου n είναι το μέγεθος του στιγμιότυπου εισόδου και k μία οποιαδήποτε σταθερά. Τα προβλήματα που λύνονται σε πολυωνυμικό χρόνο αποτελούν την κλάση πολυπλοκότητας P. Η πεποίθηση ότι η κλάση P ταυτίζεται με την κλάση των ευεπίλυτων προβλημάτων αποδίδεται στους *S.* Cook και R. Karp και αναφέρεται σαν αξίωμα των Cook – Karp.

Κάποια βασικά επιχειρήματα που υποστηρίζουν το αξίωμα των Cook –Karp είναι:

- Η ταχύτητα με την οποία αυξάνεται ένα πολυώνυμο μικρού βαθμού επιτρέπει την επίλυση αρκετά μεγάλων στιγμιότυπων ενός προβλήματος σε εύλογο χρονικό διάστημα.
- Η ύπαρξη ενός αλγόριθμου πολυωνυμικού χρόνου, επιτρέπει τη σημαντική αύξηση του μεγέθους των στιγμιότυπων που λύνει ο αλγόριθμος σε δεδομένο χρονικό διάστημα, σαν αποτέλεσμα της συνεχούς αύξησης της ταχύτητας των υπολογιστών.

6.3 Κλάση P

Η κλάση P περιλαμβάνει όλα τα προβλήματα απόφασης, δηλαδή προβλήματα στα οποία καλούμαστε να απαντήσουμε μια συγκεκριμένη ερώτηση με ναι ή όχι. Είναι μια από τις σημαντικότερες κλάσεις πολυπλοκότητας και περιλαμβάνει τα προβλήματα απόφασης που επιλύονται από κάποιον αλγόριθμο σε πολυωνυμικό χρόνο. Δηλαδή για έναν αλγόριθμο θεωρούμε ότι έχει υπολογιστική πολυπλοκότητα τάξης P, όταν υπάρχει κάποιο πολυώνυμο p(), τέτοιο ώστε η πολυπλοκότητα του αλγόριθμου να μπορεί να εκφρασθεί σαν O(p(n)) για οποιοδήποτε πλήθος δεδομένων n.

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

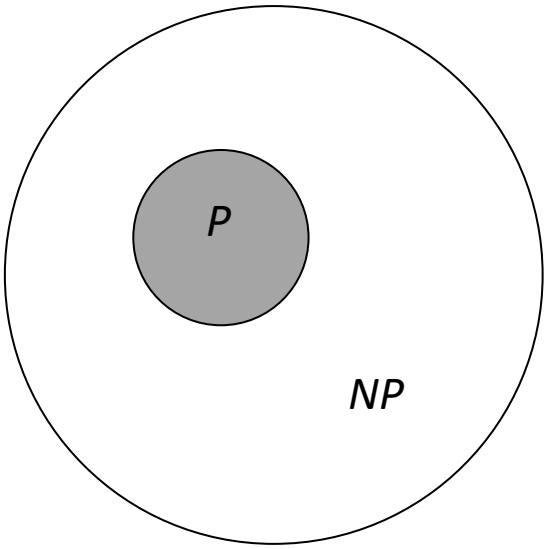
6.4 Κλάση NP

Η κλάση NP περιλαμβάνει όλα τα προβλήματα απόφασης για τα οποία μπορούμε να επαληθεύσουμε μια λύση τους σε πολυωνυμικό χρόνο.

Παράδειγμα 1: Θεωρούμε το παρακάτω πρόβλημα απόφασης που έχει τη μορφή: «Τα δεδομένα *A*=(*A*(1), *A*(2), …, *A*(*n*)) ικανοποιούν την συνθήκη *C*».
Δίνεται ένας φυσικός αριθμός a με δυαδική αναπαράσταση a = $a_n a_{n-1} \ldots a_2 a_1$= , όπου $a_i \in$ {0,1}. Θέλουμε να ελέγξουμε αν αυτός ο αριθμός έχει ακέραιους διαιρέτες, δηλαδή αν είναι σύνθετος. Υποθέτουμε ότι αυτή η συνθήκη ισχύει έστω και αν βρεθεί ένας μόνος διαιρέτης, εκτός του 1 και του a. Όταν βρίσκουμε έναν αριθμό x σαν αξιόπιστη λύση, δηλαδή όταν θεωρούμε ότι ο x διαιρεί τον a, το πρόβλημα παίρνει τη μορφή του ελέγχου στο εάν ο x διαιρεί τον a. Εάν το νέο αυτό πρόβλημα ανήκει στην κλάση P, τότε ορίζεται πως το αρχικό πρόβλημα ανήκει στην κλάση NP.

Παράδειγμα 2: Θεωρούμε το πρόβλημα απόφασης: Αν ένα γράφημα G(V,E) έχει κύκλο Hamilton ή όχι. Όταν η απάντηση είναι θετική, δίνεται μια σειρά κόμβων v_i, i=0,1,…,|V|-1 του γραφήματος. Ελέγχουμε σε χρόνο O(|V|) αν οι κόμβοι που δόθηκαν ως απάντηση σχηματίζουν κύκλο, εξετάζοντας αν ισχύει (v_i, $v_{(i+1) \bmod |V|}$) $\in$ E για i=0,1,…,|V|-1. Προφανώς μια θετική απάντηση μπορεί να βρεθεί σε πολυωνυμικό χρόνο για όλα τα προβλήματα της P, αφού αυτά επιλύονται σε πολυωνυμικό χρόνο.

Από τα παραπάνω παραδείγματα καταλήγουμε στην σχέση P $\subseteq$ NP . Δηλαδή ένα πρόβλημα που ανήκει στην κλάση P, ανήκει σίγουρα και στην κλάση NP. Αν μπορούμε να λύσουμε ένα πρόβλημα σε πολυωνυμικό χρόνο, μπορούμε να ελέγξουμε και τη λύση του σε πολυωνυμικό χρόνο.
Ορισμός κλάσης NP: Το σύνολο των αλγόριθμων απόφασης που επιλύονται σε πολυωνυμικό χρόνο από μη προσδιοριστικούς αλγόριθμους, ανήκουν στην κλάση NP.



Σχήμα 15: Σχηματική αναπαράσταση P ⊆ NP

6.4.1 NP-complete κλάση

Για κάθε κλάση πολυπλοκότητας υπάρχουν κάποια προβλήματα, τα οποία είναι πλήρη για την κλάση αυτή, δηλαδή την περιγράφουν πλήρως υπό την έννοια ότι η αποδοτική επίλυση ενός πλήρους προβλήματος για μια κλάση πολυπλοκότητας θα επιφέρει την επίλυση όλων των ισοδύναμων προβλημάτων που ανήκουν στην κλάση αυτή. Έτσι ορίζεται η κλάση προβλημάτων NP-complete.

Τα προβλήματα που ανήκουν στην NP-complete(NPC) κλάση, έχουν την ιδιότητα να ανήκουν και στη NP κλάση. Επίσης αν έστω και για ένα από τα προβλήματα της NP-complete κλάσης αποδειχθεί ότι ανήκει στην κλάση P τότε αυτόματα έχει αποδειχθεί ότι NPC ⊂ P , δηλαδή όλα τα NP-complete προβλήματα θα ήταν επιλύσιμα σε πολυωνυμικό χρόνο.

Σύμφωνα με τα παραπάνω γίνεται σαφές ότι από τα προβλήματα της κλάσης NP, τα προβλήματα που ανήκουν στην P είναι εύκολα όσον αφορά την υπολογιστική τους πολυπλοκότητα, ενώ τα προβλήματα της κλάσης NP-complete είναι δύσκολα. Από την άλλη μεριά όμως μπορεί να αποδειχθεί ότι όλα τα NP προβλήματα, συμπεριλαμβανόμενων και των NP-complete επιδέχονται αλγόριθμους με πολυπλοκότητα $f(n) \in O(2^n)$. Είναι δηλαδή λιγότερο ή το πολύ ισοδύναμοι με αλγόριθμους εκθετικής πολυπλοκότητας.

Κάποια χαρακτηριστικά προβλήματα της κλάσης NP-complete είναι το πρόβλημα ισομορφισμού γραφημάτων(graph isomorphism problem), το πρόβλημα του σακιδίου(Knapsack problem), το πρόβλημα του μονοπατιού Hamilton(Hamiltonian path problem), το πρόβλημα του πλανόδιου πωλητή(Travelling salesman problem), το πρόβλημα αθροίσματος υποσυνόλων(Subset sum problem), το πρόβλημα της κλίκας(Clique problem), το πρόβλημα κάλυψης κορυφών(Vertex cover problem), το πρόβλημα ανεξάρτητου συνόλου(Independent set problem), το πρόβλημα κυριαρχίας συνόλου(Dominating set problem) και το πρόβλημα χρωματισμού γραφημάτων(Graph coloring problem).

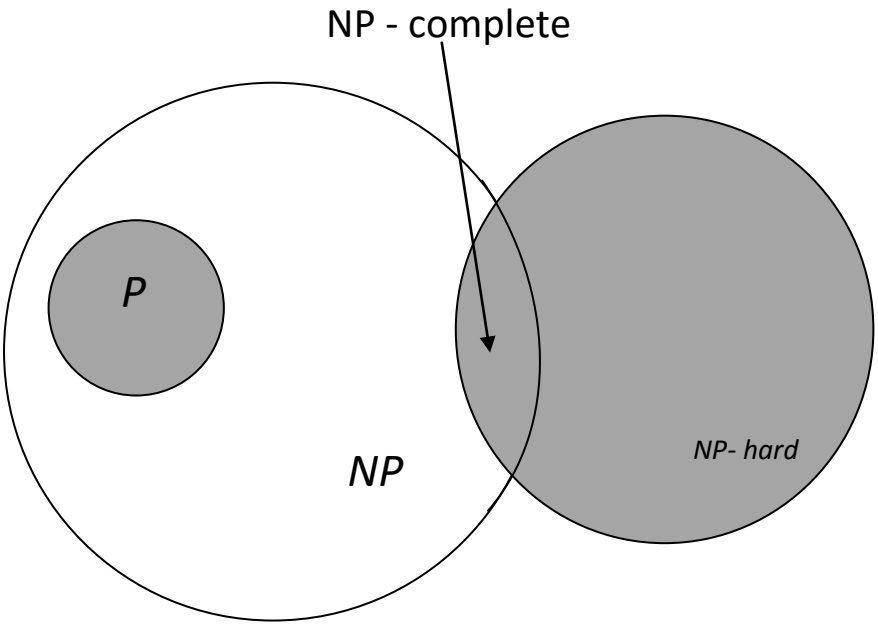
6.4.2 NP-hard κλάση

Μία άλλη κλάση είναι η NP-hard η οποία περιλαμβάνει προβλήματα που είναι τουλάχιστον τόσο δύσκολα, όσο οποιοδήποτε πρόβλημα της κλάσης NP. Η σημασία της κλάσης αυτής έγκειται στο γεγονός ότι αν ανακαλυφθεί ένας αλγόριθμος πολυωνυμικής πολυπλοκότητας $O(n^k)$ που να επιλύει ένα πρόβλημα NP-hard, τότε όλα τα προβλήματα NP θα επιλύονται σε πολυωνυμικό χρόνο.

Αν έχουμε δύο προβλήματα απόφασης Α και Β, το πρόβλημα Α περιορίζει το πρόβλημα Β αν και μόνο αν υπάρχει ένας τρόπος ώστε να επιλυθεί το Α από προσδιοριστικό αλγόριθμο σε πολυωνυμικό χρόνο χρησιμοποιώντας έναν προσδιοριστικό αλγόριθμο που επιλύει το Β.

Αν ένα πρόβλημα ανήκει στην κλάση NP-hard και ταυτόχρονα και στην κλάση NP, τότε θεωρείται ότι ανήκει στην κλάση NP-complete. Οι σχέσεις που υπάρχουν μεταξύ των διαφόρων κλάσεων προβλημάτων φαίνονται στο Σχήμα 16.

Κάποια χαρακτηριστικά προβλήματα της κλάσης NP-hard είναι το πρόβλημα αθροίσματος υποσυνόλων(Subset sum problem) και το πρόβλημα του πλανόδιου πωλητή(Travelling salesman problem) που ανήκουν επίσης και στην κλάση NP-complete. Ένα ακόμη γνωστό πρόβλημα της NP-hard είναι το πρόβλημα τερματισμού(halting problem).



Σχήμα 16: Οι Κλάσεις NP-complete και NP-hard σχηματικά

6.4.2 Το ερώτημα P=NP?

Ένα βασικό ερώτημα που απασχολεί την επιστήμη των υπολογιστών είναι αν το P ίσο με το NP, δηλαδή αν P ⊂ NP. Το ζήτημα αυτό δεν έχει αποδειχτεί μέχρι σήμερα. Ενώ είναι γνωστά τα προβλήματα της κλάσης NP δεν μπορεί να αποδειχθεί ότι δεν μπορεί να βρεθούν κάποιοι καλύτεροι αλγόριθμοι για αυτά τα προβλήματα ώστε να ανήκουν στην κλάση P. Το ερώτημα λοιπόν είναι αν όντως υπάρχουν προβλήματα τα οποία είναι εύκολο να ελεγχθούν αλλά πρακτικά αδύνατο να λυθούν με άμεσες αλγοριθμικές διαδικασίες.

Το πρόβλημα «P versus NP» όπως είναι γνωστό στην επιστημονική κοινότητα, είναι θεμελιώδες για την ασφάλεια των υπολογιστών, καθώς η απόδειξη του ζητήματος P=NP θα καταρρίψει την έννοια της κρυπτογραφίας.

ΒΙΒΛΙΟΓΡΑΦΙΑ

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

2. Θ. Παπαθεοδώρου, «Αλγόριθμοι: Εισαγωγικά Θέματα και Παραδείγματα», Εκδόσεις Πανεπιστημίου Πατρών, 1999.

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

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

5. H. Deitel, P. Deitel, «Java ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ», 8η έκδοση, Εκδόσεις Μ. Γκιούρδας, Αθήνα 2010.

6. Δρ. Ηλίας Κ. Σάββας, «Αλγόριθμοι και Πολυπλοκότητα», http://www.teilar.gr/dbData/ProfAnn/profann-2a82b2bb.pdf

7. Β. Χαρμανδάρης, «Ανάλυση-Απόδοση Αλγορίθμων», http://www.tem.uoc.gr/~vagelis/Courses/EM240/Ch2_Analysh-Apodosh-Algorithmvn.pdf

8. Γ. Φρ. Γεωργακόπουλος, «Αλγόριθμοι και Πολυπλοκότητα», http://www.csd.uoc.gr/~hy380/documents2009/HY380-2-divide-n-conquer.pdf

9. Β. Ζησιμόπουλος, «Αλγόριθμοι και Πολυπλοκότητα», http://cgi.di.uoa.gr/~vassilis/ac/VZalgorithms08.pdf

10. Γ. Βασιλειάδης, «Σημειώσεις Δυναμικού Προγραμματισμού», http://users.auth.gr/gvasil/dynamikos_new.pdf

11. Π. Α. Μηλιώτης, «Δυναμικός Προγραμματισμός», http://eduportal.dmst.aueb.gr/html/det/DYNAMIKOS_PROGRAMMATISMOS__661.pdf

12. Δ. Φωτάκης, Π.Σπυράκης, «Αλγόριθμοι και Πολυπλοκότητα», http://evripides.mysch.gr/Algorithmoi_kai_Polyplokotita.pdf

13. Β. Δούρος, «Κλάση NP, NP-Complete Προβλήματα», http://www.aueb.gr/users/douros/algorithms/tutorials_2012/14_frontistirio_complete.pdf

14. Μ. Μαυρονικόλας, Α. Φιλίππου, «Αλγόριθμοι και Πολυπλοκότητα», https://www.cs.ucy.ac.cy/~mavronic/Classes/cs232/Notes/notes1.pdf

15. http://discrete.gr/complexity/?el

16. http://lca.ceid.upatras.gr/courses/sttcc/Section3.pdf

17. http://cgi.di.uoa.gr/~vassilis/ac/10L13-ElementaryGraphsAlgorithms.pdf

18. http://www.corelab.ntua.gr/courses/algorithms/

19. http://infoman.teikav.edu.gr/e_education/67/files/alg%20eis.pdf

20. http://en.wikipedia.org/wiki/Complexity_theory

21. http://en.wikipedia.org/wiki/NP-complete

22. http://en.wikipedia.org/wiki/NP-hard