ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
Τμήμα Πληροφορικής

ΑΛΓΟΡΙΘΜΟΙ

(Πανεπιστημιακές Παραδόσεις)

Θ. Καλαμπούκης - Ε. Μαγείρου - Μ. Σιδέρη

ΑΘΗΝΑ 2002

ΔΩΡΕΑ ΣΤΗ ΜΝΗΜΗ
ΚΩΝΣΤΑΝΤΙΝΟΥ Η.
ΠΑΠΑΚΩΝΣΤΑΝΤΙΝΟΥ

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
Τμήμα Πληροφορικής

ΑΛΓΟΡΙΘΜΟΙ

(Πανεπιστημιακές Παραδόσεις)

Θ. Καλαμπούκης - Ε. Μαγείρου - Μ. Σιδέρη

Αθήνα,

Πρόλογος

Οι σημειώσεις αυτές γράφτηκαν για το μάθημα "Αλγόριθμοι" που διδάσκεται στο τρίτο έτος του Τμήματος Πληροφορικής του Οικονομικού Πανεπιστημίου Αθηνών και ασφαλώς δεν αντικαθιστούν ένα βιβλίο ανάλυσης αλγορίθμων.

Σκοπός του μαθήματος είναι να εισαγάγει τους φοιτητές στις βασικές μεθοδολογίες ανάλυσης της πολυπλοκότητας αλγορίθμων και σε μερικούς βασικούς αλγόριθμους της επιστήμης των υπολογιστών. Ορισμένα θέματα, όπως για παράδειγμα η αναζήτηση (εσωτερική και εξωτερική), δεν περιλαμβάνονται στις σημειώσεις αυτές επειδή έχουν διδαχθεί στα μαθήματα "Δομές Δεδομένων" και "Δομές Αρχείων".

Οπωσδήποτε δεν περιλαμβάνονται σημαντικοί αλγόριθμοι Υπολογιστικών Μαθηματικών, Υπολογιστικής Γεωμετρίας κτλ. Ο περιορισμός αυτός της ύλης οφείλεται κυρίως σε εκπαιδευτικούς λόγους και συγκεκριμένα στην αδυναμία να καλυφθούν όλα τα θέματα με αποτελεσματικό τρόπο σε ένα εξαμηνιαίο μάθημα.

Στη νέα αυτή έκδοση προστέθηκαν:

- *Περισσότεροι αλγόριθμοι δυναμικού προγραμματισμού με σημαντικές εφαρμογές όπως Δρομολόγηση Πακέτων, Σχεδιασμό Δένδρων Αναζήτησης*
- *Αλγόριθμοι Θεωρίας Αριθμών και εφαρμογή τους σε θέματα Κρυπτογραφίας με Δημόσιο Κλειδί.*

Ευχαριστούμε θερμά το Στέλιο Μιχαηλίδη για τη βοήθειά του στη στοιχειοθεσία του βιβλίου σε LaTeX.

Θεόδωρος Καλαμπούκης, Ευάγγελος Μαγείρου, Μάρθα Σιδέρη
Αθήνα 1998

Πρόλογος

Περιεχόμενα

1 ΑΞΙΟΛΟΓΗΣΗ ΑΛΓΟΡΙΘΜΩΝ **1**

1.1. Γενικά 1

1.2. Πολυπλοκότητα 2

1.3. Ανάλυση αλγορίθμων - RAM πρότυπο 4

1.3.1. Συμβολισμός O (Big oh notation) 7

1.3.2. Συμβολισμός Θ (theta notation) 8

1.3.3. Συμβολισμός Ω (Big omega notation) 9

1.3.4. Συμβολισμός ≈ (asymptotically equal notation) 11

1.3.5. Συμβολισμός o (little o notation) 11

1.3.6. Συμβολισμός ω (little omega notation) 12

1.3.7. Ασκήσεις 13

1.4. Ασυμπτωτική συμπεριφορά αθροισμάτων 15

1.4.1. Με επαγωγή 15

1.4.2. Με αντικατάσταση 15

1.4.3. Με σπάσιμο του αθροίσματος 16

1.4.4. Προσέγγιση αθροίσματος με ολοκλήρωμα 17

2 ΑΝΑΔΡΟΜΙΚΑ ΠΡΟΒΛΗΜΑΤΑ **21**

2.1. Μέθοδος "Διαίρει και Βασίλευε" 21

2.1.1. Επαναληπτική μέθοδος 22

2.1.2. Αλλες τεχνικές για τη λύση επαν. σχημάτων (Telescoping) 25

2.2. Εφαρμογές 27

2.3. Μέθοδος αντικατάστασης 30

3 ΤΑΞΙΝΟΜΗΣΗ **33**

3.1. Γενικά 33

3.2. Mergesort 35
3.3. Απ' ευθείας εισαγωγή (Insertion sort) 35
3.4. Quicksort 38
3.5. Πολυπλοκότητα του quicksort 41
3.6. Ταξινόμηση με επιλογή (Selection sort) 47
3.7. Heapsort 48
3.8. Bin Sort 51
3.9. Γενίκευση του Radix-sort 54
3.9.1. Λεξικογραφική ταξινόμηση στοιχείων (Ορισμός) 54
3.10. Το πρόβλημα της επιλογής 55

4 ΑΝΑΖΗΤΗΣΗ ΣΥΜΒΟΛΟΣΕΙΡΩΝ 57
4.1. Γενικά 57
4.2. Αλγόριθμος Brute-Force 57
4.2.1. Αλγόριθμος των Knuth-Morris-Pratt 59
4.2.2. Αλγόριθμος των Boyer-Moore 62

5 ΟΡΟΛΟΓΙΑ ΓΡΑΦΗΜΑΤΩΝ 65
5.1. Γραφήματα 65
5.2. Παράσταση γραφημάτων 67
5.3. Γραφήματα με κατεύθυνση 72
5.4. Γραφήματα με βάρη 72

6 ΣΤΟΙΧΕΙΩΔΕΙΣ ΑΛΓΟΡΙΘΜΟΙ ΓΡΑΦΗΜΑΤΩΝ 75
6.1. Εξερεύνηση σε βάθος 75
6.2. Δισυνεκτικές συνιστώσες, κόμβοι άρθρωσης 81
6.3. Ακυκλικά γραφήματα με κατεύθυνση, τοπολογική ταξινόμηση . 87
6.4. Συνεκτικές συνιστώσες γραφημάτων με κατεύθυνση 91
6.5. Μη αναδρομική εξερεύνηση σε βάθος, εξερεύνηση κατά πλάτος 102

7 ΠΡΟΒΛΗΜΑΤΑ ΣΕ ΓΡΑΦΗΜΑΤΑ ΜΕ ΒΑΡΗ 107
7.1. Ελάχιστα μονοπάτια 107
7.2. Ελάχιστα δένδρα 115

8 ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ 131
8.1. Εισαγωγή 131
8.2. Μεταβατική κλειστότητα γραφήματος 132

8.3. Ελάχιστα μονοπάτια μεταξύ κάθε ζεύγους σημείων 135
8.4. Το πρόβλημα του περιοδεύοντος πωλητή 138

9 NP–ΠΛΗΡΗ ΠΡΟΒΛΗΜΑΤΑ 141
9.1. Εισαγωγή 141
9.2. Προβλήματα απόφασης 143
9.3. Οι κλάσεις $\mathcal{P}$ και $\mathcal{NP}$ 145
9.4. Πολυωνυμικές αναγωγές και πλήρη προβλήματα 146
9.5. Μερικά $\mathcal{NP}$-πλήρη προβλήματα. 150

10 Αλγόριθμοι Θεωρίας Αριθμών 157
10.1. Κρυπτογράφηση με δημόσιο κλειδί κρυπτογράφησης 157
10.2. Το σύστημα RSA (Rivest, Shamir, Adleman) 158
10.3. Δικαιολόγηση συστήματος RSA 160

11 Αλλες Εφαρμογές του Δυναμικού Προγραμματισμού 167
11.1. Το πρόβλημα του σακκιδίου 167
11.1.1. Πρώτη περίπτωση - (Integer Knapsack) 168
11.1.2. Δεύτερη περίπτωση - $(0-1$ kanpsack$)$ 168
11.2. Βέλτιστο Δυαδικό Δένδρο Αναζήτησης 170
11.3. Πολλαπλασιασμός Πινάκων 173
11.4. Πρόβλημα Ελάχιστης Διαδρομής Αλγόριθμος Bellman-Ford . . 176

Κεφάλαιο 1

ΑΞΙΟΛΟΓΗΣΗ ΑΛΓΟΡΙΘΜΩΝ

1.1. Γενικά

Η λέξη *algorithm* εμφανίστηκε για πρώτη φορά στο λεξικό Webster's New World Dictionary το 1957. Προέρχεται από την αρχαία λέξη algorism που σημαίνει κάνω πράξεις με Αραβικούς αριθμούς. Για την προέλευση της λέξης υπάρχουν διάφορες εκδοχές όπως π.χ., άλγος+αριθμός. Τελικά οι ιστορικοί των Μαθηματικών βρήκαν ότι η σωστή προέλευση της λέξης ειναι από τ'όνομα του Αραβα Abu Ja'far Mohammed ibn Musa al-Khowarizmi. Μέχρι το 1950 η λέξη ήταν συνιφασμένη με τον αλγόριθμο του Ευκλείδη. Η μοντέρνα έννοια της λέξης είναι: συνταγή,διεργασία, μέθοδος, διαδικασία, ρουτίνα, με μια μικρή οπωσδήποτε διαφορά.
Ορισμός. *Αλγόριθμος* είναι ένα πεπερασμένο σύνολο από κανόνες οι οποίοι δίνουν μια ακολουθία από πράξεις (βήματα) για την λύση ενός προβλήματος. Οι κανόνες που διέπουν κάθε αλγόριθμο είναι:

1. Κάθε αλγόριθμος είναι *πεπερασμένος (Finite)*

2. *Αυστηρά ορισμένος (Definite).* Κάθε βήμα του αλγόριθμου πρέπει να είναι αυστηρά ορισμένο. Για τον λόγο αυτό η χρήση μιας γλώσσας όπως πχ η Ελληνική μπορεί να δημιουργήσει προβλήματα καθόσον ο αναγνώστης μπορεί να μη καταλάβει τι ακριβώς εννοούσε ο συγγραφέας. Ετσι λοιπόν η περιγραφή του αλγόριθμου συνηθίζεται να δίνεται σε μία γλώσσα προγραμματισμού ή σε μία "Μαθηματική" γλώσσα.

3. *Είσοδος δεδομένων (input).* Κάθε αλγόριθμος έχει κανένα ή περισσότερα δεδομένα εισόδου τα οποία χρειάζονται για να ξεκινήσει.

4. *Εξοδος δεδομένων (Output).* Κάθε αλγόριθμος έχει ένα ή περισσότερα αποτελέσματα.

5. *Αποτελεσματικός (Effective).* Κάθε αλγόριθμος γενικά αναμένεται να είναι αποτελεσματικός. Αυτό σημαίνει ότι όλες οι πράξεις που εκτελούνται από τον αλγόριθμο είναι απλές (βασικές), ακριβείς και μπορούν να εκτελεστούν σε πεπερασμένο χρόνο με "μολύβι καί χαρτί".

Η συνταγή ενός φαγητού δεν αποτελεί αλγόριθμο και τούτο γιατί ορισμένα βήματα-διεργασίες μιας συνταγής δεν είναι σαφώς ορισμένα.

Ο περιορισμός του αλγορίθμου να είναι πεπερασμένος δεν είναι τόσο πολύ "αυστηρός" στη πράξη. Δηλ. ένας αλγόριθμος μπορεί να είναι πεπερασμένος, αλλά να θέλει πολύ χρόνο για να τελειώσει. Στη πράξη δεν θέλ' με απλά ένα αλγόριθμο αλλά ένα καλό αλγόριθμο. Τι είναι όμως καλός αλγόριθμος; Ενα κριτήριο είναι το πόσο χρόνο θέλει για να τελειώσει. Αλλα κριτήρια είναι ο χώρος που χρειάζεται, η απλότητά του κλπ. Πολλές φορές έχουμε για το ίδιο πρόβλημα πολλούς αλγορίθμους και θα πρέπει ν'αποφασίσουμε ποιός είναι ο καλύτερος. Αυτό μας οδηγεί στο εξαιρετικά ενδιαφέρον πεδίο της ανάλυσης αλγορίθμων. Δηλ. δοθέντος ενός αλγορίθμου να υπολογιστούν τα χαρακτηριστικά της απόδοσής του. Η γενική ιδέα είναι ο υπολογισμός της μέσης συμπεριφοράς του αλγόριθμου. Σε ορισμένες όμως περιπτώσεις μας ενδιαφέρει κατά πόσο ο αλγόριθμος είναι βέλτιστος κατά κάποια έννοια. Η θεωρία αλγορίθμων είναι ένα τελείως διαφορετικό πεδίο το οποίο εξετάζει την ύπαρξη ή μη αποτελεσματικών αλγορίθμων. Για την ανάλυση αλγορίθμων απαιτούνται γνώσεις στοιχειώδους Αλγέβρας, πιθανοτήτων και συνδιαστικής.

1.2. Πολυπλοκότητα

Επειδή είναι δυνατόν για την λύση ενός προβλήματος να χρησιμόποιηθούν περισσότεροι του ενός αλγόριθμοι, είναι αναγκαίο να έχουμε κάποια κριτήρια επιλογής του καλύτερου. Τα κριτήρια αυτά θα πρέπει να είναι τέτοια, ώστε να μπορούμε να κάνουμε μια ποιοτική ανάλυση του αλγόριθμου. Η απόδοση του αλγόριθμου αναλύεται με βάση τις διάφορες παραμέτρους του προβλήματος.

Συνήθως τα σπουδαιότερα κριτήρια, μετά φυσικά την ορθότητα του αλγορίθμου, αναφέρονται στην αποδοτικότητά του. Η αποδοτικότητα συνήθως καθορίζεται από το χώρο της μνήμης που απαιτεί ο αλγόριθμος και το χρόνο εκτέλεσής του. Οταν ο χρόνος εκτέλεσης ενός προγράμματος μετριέται σε δευτερόλεπτα και ο χώρος με το πλήθος των ψηφιοσυλλαβών που καταλαμβάνει το πρόγραμμα στη μνήμη του υπολογιστή, τότε τα αποτελέσματα της ανάλυσης θα εξαρτιούνται άμεσα από τον υπολογιστή που χρησιμοποιούμε. Υπολογιστές με διαφορετική αρχιτεκτονική, και διαφορετικό σύνολο εντολών θα μας δώσουν διαφορετικά αποτελέσματα για κάποιο δεδομένο αλγόριθμο. Για το λόγο αυτό χρησιμοποιούμε για την αξιολόγιση των αλγορίθμων μεθόδους που είναι ανεξάρτητες από τον υπολογιστή. Ετσι, τα αποτελέσματα της απόδοσης των αλγορίθμων εκφράζονται συνήθως ως μια συνάρτηση του μεγέθους των δεδομένων εισόδου του προβλήματος n. Είναι προφανές ότι το πλήθος των βασικών πράξεων που απαιτούνται για ένα αλγόριθμο είναι ανάλογο του μεγέθους των δεδομένων εισόδου (μεγέθους του προβλήματος). Ετσι η πολυπλοκότητα χρόνου μετράτε εκφράζοντας τον χρόνο εκτέλεσης ενός αλγορίθμου ως συνάρτηση του μεγέθους των δεδομένων του προβλήματος. Ο χρόνος που απαιτείται για να εκτελεστεί ένας αλγόριθμος συμβολίζεται με $T(n)$ και πολλές φορές αναφέρεται στη βιβλιογραφία ως *πολυπλοκότητα χρόνου (time complexity)*. Ετσι λέμε οτι ο πολλαπλασιασμός δύο πινάκων $n \times n$ απαιτεί $O(n^3)$ χρόνο. Αν κάποιος μπορούσε να ελαττώσει το χρόνο αυτό ώστε να είναι $O(n^k)$, όπου $2 < k < 3$, θα ήταν μεγάλη βελτίωση. Ενας αλγόριθμος που έχει χρόνο εκτέλεσης ανάλογο του cn^3 ή cn^5 θεωρείται εύκολο πρόβλημα σε αντίθεση μ'ενα αλγόριθμο που απαιτεί χρόνο ανάλογο του 2^n το οποίο θεωρείται δύσκολο (hard) πρόβλημα. Ο γενικός κανόνας είναι ότι αν ο χρόνος εκτέλεσης ενός αλγορίθμου είναι πολυωνυμική συνάρτηση του πλήθους των δεδομένων τότε το πρόβλημα θεωρείται απλό διαφορετικά δύσκολο (σκληρό). Για να πείσει κανείς κάποιον ότι ένα πρόβλημα είναι δύσκολο θα πρέπει να δείξει ότι είναι αδύνατον να βρεθεί ένας γρήγορος αλγόριθμος για την λύση του. Το γεγονός ότι ένα πρόβλημα είναι δύσκολο δέν σημαίνει ότι σε ειδικές περιπτώσεις δεν μπορούμε να το λύσουμε εύκολα. Ενα πρόβλημα είναι hard (δύσκολο) όταν δεν μπορούμε να έχουμε ένα αλγόριθμο που να μας εξασφαλίζει γρήγορη απόδοση για όλες τις περιπτώσεις (περιστατικά, instances) του προβλήματος. Ενας αλγόριθμος θεωρείται γρήγορος (fast) όταν εξασφαλίζει γρήγορη απόδοση π.χ. όταν ξέρουμε με βεβαιότητα ότι ο αλγόριθμος θα τελειώσει μετά $5n^3$ msecs. Σε μια τέτοια περίπτωση μιλάμε για πολυπλοκότητα στη *χειρότερη περίπτωση (worst case performahce)*

$$T(n) = \max_{(\alpha_1,\alpha_2,...,\alpha_n)\in D} T(\alpha_1,\alpha_2,...\alpha_n)$$

όπου $(\alpha_1,\alpha_2,...,\alpha_n)$ είναι μία n-ιάδα δεδομένων εισόδου μέσα από το σύνολο D όλων των δυνατών δεδομένων μήκους n. π.χ. όταν έχουμε να ταξινομήσουμε n αριθμούς που είναι σχεδόν ταξινομημένοι με bubblesort τότε το πλήθος των πράξεων είναι κατά πολύ μικρότερο του n^2. Ωστόσο όμως τα προς ταξινόμηση στοιχεία δεν είναι πάντα σχεδόν ταξινομημένα. Επίσης μπορεί να αξιολογήσουμε την πολυπλοκότητα ενός αλγορίθμου σε μια *μέση περίπτωση (average case complexity)*. Αυτό δηλώνει την μέση απόδοση του αλγορίθμου για όλα τα δυνατά δεδομένα εισόδου. Δηλ.

$$TM(n) = (1/|D|) \sum_{(\alpha_1,\alpha_2,...,\alpha_n)\in D} T(\alpha_1,\alpha_2,...\alpha_n)$$

όπου $|D|$ είναι ο πληθάριθμος του D.

Οταν λέμε ότι ένας αλγόριθμος είναι αργός (slow) εννοούμε ότι η πολυπλοκότητά του εκφράζεται με μία συνάρτηση που "αυξάνει γρηγορότερα" από οποιοδήποτε πολυώνυμο π.χ. είναι ανάλογος του e^n ή 2^n. Ενας αλγόριθμος είναι αργός όταν για οποιοδήποτε πολυώνυμο p απαιτεί περισσότερες από p(n) πράξεις.

1.3. Ανάλυση αλγορίθμων - RAM πρότυπο

Ανάλυση ή αξιολόγηση ενός αλγόριθμου όπως είπαμε σημαίνει τον υπολογισμό του χώρου και του χρόνου εκτέλεσής του. Για τον υπολογισμό της πολυπλοκότητας του χρόνου μετράμε συνήθως το πλήθος των αριθμητικών πράξεων που εκτελούνται από τον αλγόριθμο. Θα μπορούσαμε ίσως να συμπεριλάβουμε και άλλες πράξεις όπως για παράδειγμα λογικές πράξεις, το χρόνο προσπέλασης στα στοιχεία πινάκων, κ.λπ., αλλά εκείνο που μας ενδιαφέρει είναι η συμπεριφορά του αλγορίθμου όταν το μέγεθος των δεδομένων εισόδου μεγαλώνει. Πρίν όμως προχωρήσουμε στην αξιολόγηση ενός αλγορίθμου πρέπει να έχουμε κατά νου το πρότυπο του υπολογιστή (μοντέλο υλοποίησης) που θα χρησιμοποιειθεί. Εδώ θα περιγράψουμε πολύ συνοπτικά το πρότυπο RAM (Random Access Machine) το οποίο αποτελεί μία λογική αφαίρεση (abstraction) ενός υπολογιστή γενικού σκοπού. Το μοντέλο RAM περιγράφεται ως εξής:

Η μνήμη της μηχανής αυτής είναι ένας πίνακας από n λέξεις (words) κάθε μία των οποίων μπορεί να χωρέσει ένα ακέραιο. Οι λέξεις αυτές είναι αριθμημένες (έχουν διευθύνσεις) από 1 έως n. Η μηχανή έχει επίσης ένα σταθερό (πεπερασμένο) πλήθος από καταχωρητές $(r_0, r_1, ..., r_m)$ καθένας των οποίων μπορεί να χωρέσει ένα ακέραιο. Σ'ενα βήμα η μηχανή μπορεί να μεταφέρει το περιεχόμενο ενός καταχωρητή σε μιά θέση στη μνήμη της οποίας η διεύθυνση είναι στον καταχωρητή, ή να μεταφέρει σ'ενα καταχωρητή το περιεχόμενο μιας θέσης μνήμης ή να εκτελέσει μια αριθμητική πράξη πάνω στα περιεχόμενα δύο καταχωρητών ή να συγκρίνει τα περιεχόμενα δύο καταχωρητών. Το πρόγραμμα είναι μία ακολουθία απο λειτουργίες (operations) προς εκτέλεση και είναι πεπερασμένου μεγέθους. Υπάρχουν αριθμητικές εντολές, I/O εντολές, εντολές έμμεσης διευθυνσιοδότησης και εντολές μεταφοράς ελέγχου. Ολοι οι υπολογισμοί γίνονται στον πρώτο καταχωρητή r_0. Κάθε εντολή αποτελείται από δύο μέρη, τον κώδικα της εντολής (instruction code) και την διεύθυνση. Ολες οι εντολές εκτελούνται ακολουθιακά εκτός μετά από εντολή μεταφοράς ελέγχου (jump).

Ενα RAM πρόγραμμα ορίζει μία απεικόνηση από το input στο output. Η απεικόνηση αυτή είναι μία "μερική" απεικόνηση καθόσον μπορεί να μην ορίζεται για ειδικά δεδομένα εισόδου π.χ. όταν υπάρχει διαίρεση με το μηδέν. Ετσι ένα πρόγραμμα υπολογίζει μία συνάρτηση

$$f(a_1, a_2, ..., a_n) = y$$

όπου a_i είναι τα δεδομένα εισόδου και y το αποτέλεσμα. Ο πιό συνηθισμένος τρόπος να ορίσουμε ένα πρόγραμμα είναι να χρησιμοποιήσουμε μία γλώσσα. Γλώσσα είναι ένα σύνολο από συμβολοσειρές που σχηματίζονται από ένα πεπερασμένο σύνολο συμβόλων (το αλφάβητο). Η πολυπλοκότητα ενός RAM προγράμματος είναι μία συνάρτηση που ισούται όπως είπαμε με το μέγιστο του συνολικού χρόνου εκτέλεσης όλων των εντολών του προγράμματος για όλα τα δεδομένα εισόδου μεγέθους n. Για να υπολογίσουμε την πολυπλοκότητα χρόνου και χώρου ακριβώς πρέπει να γνωρίζουμε το χρόνο εκτέλεσης κάθε RAM εντολής και το χώρο που καταλαμβάνεται από τους καταχωρητές. Εδώ θα δούμε το κριτήριο ομοιόμορφου κόστους (uniform cost criterion). Σύμφωνα με το κριτήριο αυτό κάθε εντολή απαιτεί σταθερό χρόνο (έστω μία μονάδα χρόνου) και κάθε καταχωρητής καταλαμβάνει σταθερό χώρο (μία μονάδα χώρου).

Παράδειγμα. Εστω ότι έχουμε το πρόβλημα υπολογισμού της συνάρτησης

$$f(n) = \begin{cases} n^n & n \geq 1 \\ 0 & \text{διαφορετικά} \end{cases} \tag{1.1}$$

Η συνάρτηση αυτή θα μπορούσε να υπολογιστεί με το παρακάτω πρόγραμμα:

```
                              κόστος      επαναλήψεις
f:= 1;                          c1             1
for i:= 1 to n do               c2             n
  f:= f*n;                      c3             n
if n < 1 then f:= 0             c4             1
```

Ο χρόνος εκτέλεσης του προγράμματος ισούται με το άθροισμα των χρόνων εκτέλεσης όλων των εντολών. Συνεπώς ο χρόνος εκτέλεσης T(n) του προγράμματος είναι:

$$T(n) = c_1 + c_2 n + c_3 n + c_4 = (c_1 + c_3)n + c_1 + c_4 = an + b$$

δηλαδή ο χρόνος εκτέλεσης είναι μία γραμμική συνάρτηση του n. Στο παράδειγμα αυτό κάναμε τις εξής απλουστεύσεις. Πρώτα υποθέσαμε ότι κάθε εντολή απαιτεί κάποιο χρόνο (c_i) δηλαδή αγνοήσαμε τον πραγματικό χρόνο που απαιτεί κάθε εντολή. Μετά αντικαταστήσαμε τις σταθερές αυτές με νέες (a,b) δηλαδή αγνοήσαμε όχι μόνο τους πραγματικούς χρόνους αλλά και τους υποθετικούς χρόνους(c_i). Τέλος θα κάνουμε ακόμη μία απλούστευση. Εκείνο που μας ενδιαφέρει είναι η συχνότητα αύξησης (rate of growth) του χρόνου εκτέλεσης δηλ. από την συνάρτηση που εκφράζει την πολυπλοκότητα κρατάμε μόνο τον πιό σημαντικό όρο της (an) αφού οι υπόλοιποι όροι για μεγάλα n είναι ασήμαντοι. Επίσης αγνοούμε και την σταθερά του πλέον σημαντικού όρου καθόσον και η σταθερά είναι λιγότερο σημαντική για τον υπολογισμό της υπολογιστικής πολυπλοκότητας από την συχνότητα αύξησης για μεγάλα n. Η σταθερά είναι χαρακτηριστικό του υπολογιστή στον οποίο θα τρέξει το αλγόριθμος και εξαρτάται επίσης από τον κώδικα του προγράμματος. Ετσι λοιπόν θα λέμε ότι ο χρόνος εκτέλεσης του αλγόριθμου είναι ανάλογος του n (και θα γράφουμε όπως θα δούμε παρακάτω $T(n) = \Theta(n)$. Εστω για παράδειγμα το πρόβλημα της αντιστροφής ενός πίνακα. Ενας αλγόριθμος για το πρόβλημα αυτό λέμε ότι χρειάζεται χρόνο $15n^3$, όπου n είναι το μέγεθος του πίνακα. Αυτό σημαίνει ότι ξέρουμε ότι αν ένας 100×100 πίνακας θέλει 1 min

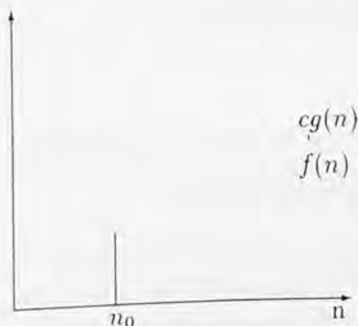
τότε ένας 200×200 θέλει 8 min. Η σταθερά 15 δεν χρησιμοποιήθηκε αλλά μόνο το γεγονός ότι η πολυπλοκότητα είναι ανάλογος του n^3.

Για να συγκρίνουμε τους αλγορίθμους για το ίδιο πρόβλημα ως προς την ταχύτητά τους, χρειαζόμαστε μία καλή και αυστηρή γλώσσα για να εκφράσουμε την πολυπλοκότητά τους. Ετσι λοιπόν θέλουμε μία γλώσσα που να μας επιτρέπει να εκφράσουμε, ότι η πολυπλοκότητα ως συνάρτηση του n αυξάνει ανάλογα του n^3, ή το πολύ όσο το n^3 για το παραπάνω παράδειγμα. Ετσι λοιπόν εισάγουμε τους παρακάτω συμβολισμούς. Οι συμβολισμοί αυτοί αναφέρονται στη βιβλιογραφία και ως υπεροπτικοί συμβολισμοί (hyperoptic) από την Ελληνική λέξη υπεροπτικός για προφανείς λόγους όπως θα δούμε παρακάτω.

1.3.1. Συμβολισμός O (Big oh notation)

Ορισμός. Λέμε ότι f(n) = O(g(n)) όταν υπάρχουν c, n_0 τέτοια ώστε $0 \leq f(n) \leq cg(n)$ για κάθε $n \geq n_0$.

Σιωπηρά στον ορισμό έχουμε υποθέσει ότι $n \longrightarrow \infty$.



Σχήμα 1.1. $f(n) = O(g(n))$

Πράγματι όταν συγκρίνουμε αλγορίθμους είναι λογικό να κάνουμε την υπόθεση ότι ενδιαφερόμαστε για την συμπεριφορά της πολυπλοκότητας όταν το μέγεθος του προβλήματος είναι σχετικά μεγάλο. Με άλλα λόγια ο ορισμός του O μας λέει ότι ο ρυθμός ανάπτυξης (rate of growth) της f δεν είναι

μεγαλύτερος του ρυθμού ανάπτυξης της g. Παραδείγματα:

$$f(n) = 3n^2 + 5n = O(n^2)$$

$$f(n) = n^3 + 4n^2 + 6n = O(n^3)$$

$$f(n) = 1/(1 + n^2) = O(1)$$

Στο σχήμα 1.1 δίνεται η γραφική παράσταση της σημασιολογίας του συμβολισμού $f(n) = O(g(n))$.

Από τον ορισμό φαίνεται ότι ο συμβολισμός $O(g(n))$ είναι ένα σύνολο που περιέχει όλες τις συναρτήσεις f που πληρούν τον ορισμό ,

$$O(g(n)) = \{f(n) : \exists c, n_0 : 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

Ετσι λοιπόν θα ήταν πιό σωστό να γράφουμε $f(n) \in O(g(n))$. Οστόσο όμως έχει επικρατήσει να δηλώνουμε το ίδιο πράγμα γράφοντας $f(n) = O(g(n))$.

Σύμφωνα με τον ορισμό παρατηρούμε ότι η συνάρτηση $f(n) = an + b$ είναι $O(n^2)$. Ο συμβολισμός O όμως χρησιμοποιείται για να δηλώσει ασυμπτωτικά αυστηρά φράγματα. Επειδή ο συμβολισμός O μας δίνει ένα πάνω φράγμα για μία συνάρτηση χρησιμοποιείται για τον υπολογισμό της πολυπλοκότητας ενός αλγόριθμου στη χειρότερη περίπτωση.

Ασκήσεις.

1. Δείξτε ότι $f(n) = n$ δεν μπορεί να είναι $O(\log n)$.
2. Δείξτε ότι $an^2 + bn + c = O(n^2)$.

Γενικά για κάθε πολυώνυμο $p(n) = a_0 + a_1 n + ... + a_k n^k$ βαθμού k ισχύει ότι: $p(n) = O(n^k)$. Επειδή μία σταθερά μπορεί να θεωρηθεί ως πολυώνυμο μηδενικού βαθμού λέμε ότι μία σταθερά είναι $O(n^0)$ ή $O(1)$.

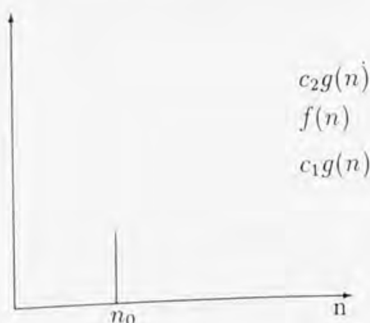
1.3.2. Συμβολισμός Θ (theta notation)

Ορισμός. Λέμε ότι $f(n) = \Theta(g(n))$ όταν υπάρχουν σταθερές c_1 και c_2 και n_0 διάφορες του μηδενός έτσι ώστε: για κάθε $n \geq n_0$ ισχύει $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$. (Τότε λέμε ότι οι f, g έχουν τον ίδιο ρυθμό ανάπτυξης). π.χ.

$$(n + 1)^2 = \Theta(3n^2)$$

$$\frac{(n^2+5n+6)}{(n^3+2n^2)} = \Theta(1/n)$$

Αν $f(n) = \Theta(n^2)$ τότε $c_1 \leq f(n)/n^2 \leq c_2$ όπου c_1, c_2 διάφορα του μηδέν για τιμές του n αρκετά μεγάλες. Στο σχήμα 1.2 δίνεται η γραφική παράσταση της σημασιολογίας του συμβολισμού $f(n) = \Theta(g(n))$.



Σχήμα 1.2. $f(n) = \Theta(g(n))$

Ο ορισμός του $\Theta(g(n))$ απαιτεί η $f(n)$ και η $g(n)$ να είναι ασυμπτωτικά μη αρνητικές.

Παραδείγματα

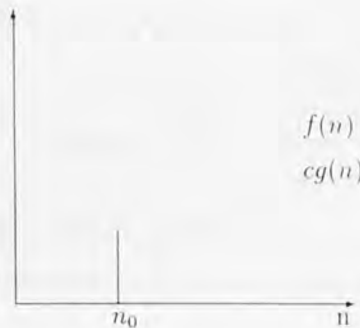
1. Δείξτε ότι $4n^3 + n^2 = \Theta(n^3)$
   Αρκεί να βρούμε c_1, c_2 και n_0 τέτοια ώστε $c_1n^3 \leq 4n^3 + n^2 \leq c_2n^3$ για κάθε $n \geq n_0$. Πράγματι για $c_1 = 3, c_2 = 5$ και $n_0 = max(0,4) = 4$ ισχύει: $3n^3 \leq 4n^3 + n^2 \leq 5n^3$ για κάθε $n \geq 4$.
2. Δείξτε αν ισχύει $f(n) = 4n^3 + n^2 = \Theta(n^2)$. Η απόδειξη γίνεται με εις άτοπο απαγωγή. Εστω ότι υπάρχει $c \geq 0$ και $n_0 : 4n^3 + n^2 \leq cn^2$ για κάθε $n \geq n_0$ ή $4n + 1 \leq c$ για κάθε $n \geq n_0$ ή $n \leq (c-1)/4$ άτοπο.
3. Δείξτε ότι $f(n) = \alpha n^2 + \beta n + \gamma = \Theta(n^2)$.

1.3.3. Συμβολισμός Ω (Big omega notation)

Ορισμός. Λέμε ότι $f(n) = \Omega(g(n))$ όταν υπάρχει $c > 0$ έτσι ώστε $f(n) \geq cg(n)$ για απείρως πολλά n, ή ισοδύναμα $f(n) = \Omega(g(n))$ όταν υπάρχει $c > 0$

και μία ακολουθία (x_j) έτσι ώστε για κάθε j ισχύει $|f(x_j)| > cg(x_j)$. Ο λόγος για τον οποίο ο συμβολισμός Ω ορίζεται για άπειρες τιμές του n και όχι για κάθε $n \geq n_0$ είναι ότι μερικοί αλγόριθμοι είναι πολύ γρήγοροι για ορισμένα δεδομένα και για άλλα δεν είναι π.χ.

$$f(n) = \begin{cases} 2n & n \text{ περιττό} \\ 2n^2 - 10 & \text{για } n \text{ άρτιο} \end{cases} \tag{1.2}$$



Σχήμα 1.3. $f(n) = \Omega(g(n))$

Συνεπώς $f(n) \geq n^2$ για $n = 4, 6, 8, ...$ και $f(n) = \Omega(n^2)$. Ο συμβολισμός Ω χρησιμοποιείται όταν θέλουμε να πούμε ότι ένας αλγόριθμος απαιτεί τουλάχιστον t1 χρόνο για να εκτελεστεί. Για παράδειγμα λέμε ότι ο πολλαπλασιασμός πινάκων έχει πολυπλοκότητα $O(n^3)$. Ωστόσο όμως υπάρχουν αλγόριθμοι γρηγορότεροι δηλ. που απαιτούν χρόνο $O(n^{2.81})$. Ισως υπάρχουν ακόμη γρηγορότεροι αλγόριθμοι. Ενα όμως είναι γεγονός ότι δεν μπορούμε να πολλαπλασιάσουμε δύο πίνακες με λιγότερες από $O(n^2)$ πράξεις αφού τα δεδομένα μας είναι $2n^2$. Συνεπώς το cn^2 είναι ένα κάτω φράγμα της πολυπλοκότητας του πολλαπλασιασμού πινάκων. Ετσι λέμε ότι ο αλγόριθμος απαιτεί χρόνο $\Omega(n^2)$. Στο σχήμα 1.3 δίνεται η γραφική παράσταση της σημασιολογίας του συμβολισμού $f(n) = \Omega(g(n))$.

1.3.4. Συμβολισμός $\approx$ (asymptotically equal notation)

Ορισμός. Λέμε ότι $f(n) \approx g(n)$ όταν

$$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = 1,$$

Ο ορισμός αυτός είναι ακόμη πιό ακριβής από τους προηγούμενους καθόσον μας λέει όχι μόνο ότι οι f και g έχουν την ίδια συχνότητα ανάπτυξης αλλά και ότι το f/g συγκλίνει στο 1 για μεγάλα n.

$$(3n+1)^4 \approx 81n^4$$

$$n^2 + n \approx n^2$$

$$\frac{(2n^3+5n-7)}{(n^2+4)} \approx 2n$$

$$2^n + 7\log n + \cos n \approx 2^n$$

1.3.5. Συμβολισμός o (little o notation)

Οπως είδαμε στο συμβολισμό O το πάνω φράγμα που υπολογίζουμε μπορεί να είναι αλλά και να μην είναι ασυμπτωτικά αυστηρό. Ο συμβολισμός $o(g(n))$ χρησιμοποιείται ακριβώς για να δηλώσει ένα πάνω φράγμα που δεν είναι ασυμπτωτικά αυστηρό.

Ορισμός. Λέμε ότι $f(n) = o(g(n))$ όταν για κάθε $c > 0$ υπάρχει $n_0 > 0$ τέτοιο ώστε $0 \leq f(n) < cg(n)$ για κάθε $n \geq n_0$ ή ισοδύναμα

$$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

Η διαφορά μεταξύ των συμβολισμών O και o είναι ότι για τον O ο ορισμός ισχύει για κάποιο $c > 0$ ενώ για τον o ισχύει για κάθε $c > 0$.

Παραδείγματα

$$n^2 = o(n^5)$$

$$5\sqrt{n} = o(n/2)$$

$$23\log n = o(n^{0.02})$$

Ο συμβολισμός αυτός δηλώνει ότι η $f(n)$ είναι αυστηρα μικρότερη από την $g(n)$. Οταν έχουμε δύο αλγορίθμους αντιστροφής ενός $n \times n$ πίνακα και ο ένας χρειάζεται, n^3 και ο άλλος $o(600n^{2.8})$ τότε για αρκετά μεγάλες τιμές του n ο δεύτερος θα είναι καλύτερος από τον πρώτο αν και ο πρώτος είναι καλύτερος για ορισμένες τιμές του n. Ομοια αν έχουμε δύο αλγορίθμους με πολυπλοκότητες $n^{2.03}$ και $o(n^2 \log n)$ τότε ο δεύτερος θα είναι καλύτερος. Από την $f(n) = 1/(1+n^2) = O(1)$ παρατηρούμε ότι ο συμβολισμός ο μπορεί να δώσει καλύτερες (sharper) πληροφορίες από τον O διότι η σχέση $1/(1+n^2) = o(1)$ μας λέει όχι μόνο ότι η συνάρτηση είναι αυστηρά φραγμένη για μεγάλα n αλλά και ότι συγκλίνει στο 0 όταν $n \longrightarrow \infty$.

1.3.6. Συμβολισμός ω (little omega notation)

Ορισμός Λέμε ότι η συνάρτηση $f(n) = \omega(g(n))$ όταν για κάθε $c > 0$ υπάρχει $n_0 > 0$ τέτοιο ώστε $0 \leq cg(n) < f(n)$ για κάθε $n \geq n_0$ ή ισοδύναμα

$$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = \infty,$$

Η σχέση αυτή μας λέει ότι η $f(n)$ είναι αυστηρά μεγαλύτερη από την $g(n)$ για παράδειγμα $2n^2 = \omega(n)$.

Ασκηση. Δείξτε ότι $\log n = o(n^{0.01})$.

Αρκεί να δείξω ότι το $n^{0.01}$ έχει μεγαλύτερη συχνότητα ανάπτυξης από το $\log n$.

$$\lim_{n \longrightarrow \infty} \log \frac{n}{n^{0.01}} = \frac{1/n}{0.01n^{-0.99}} = \frac{1}{0.01n^{0.01}} \longrightarrow 0.$$

Ασκηση. Δείξτε ότι η συχνότητα ανάπτυξης της $e^{\ln^2 n}$ είναι μεγαλύτερη της n^{1000}.

Ορισμός. Μία συνάρτηση f ονομάζεται εκθετική όταν υπάρχει $c > 1$ τέτοιο ώστε $f(n) = \Omega(c^n)$ και υπάρχει d τέτοιο ώστε $f(n) = O(d^n)$

Πέρα από τις εκθετικές συναρτήσεις έχουμε άλλες συναρτήσεις με συχνότητα ανάπτυξης ακόμη μεγαλύτερη πχ. $n!, n^n$. Οι πολυπλοκότητες όμως που έχουν κάποιο νόημα για τον προγραμματιστή είναι κατά σειρά:

$$O(1), O(\log \log n), O(\log n), O(n), O(n \log n), O(n^2), O(n^3).$$

Ο λόγος είναι απλός. Εστω ότι έχουμε ένα αλγόριθμο με πολυπλοκότητα $T(n) = 2^n$. Αν $n = 64$ και κάθε στοιχειώδης πράξη απαιτεί σ'ενα Η/Υ χρόνο 1μsec τότε ο χρόνος εκτέλεσης του αλγόριθμου θα είναι $T = 2^{64} \times 10^{-6} sec = 3.2 * 105$ χρόνια !!

1.3.7. Ασκήσεις

1. Δείξτε ότι
$$t(n) = an + b = O(n)$$
$$t(n) = 5n^3 + 2n^2 = O(n^3)$$

2. Δείξτε ότι η $t(n) = 4^n$ δεν είναι $O(2^n)$

3. Ποιές από τις παρακάτω σχέσεις είναι σωστές και ποιές είναι λάθος; (εξηγήστε γιατί)
$$(n^2 + 2n + 1) \approx n^6$$
$$(\sqrt{n} + 1)^3/(n^2 + 1) = o(1)$$
$$e^{1/n} = \Theta(1)$$
$$1/n \approx 0$$
$$n^3(\log\log n)^2 = o(n^3 \log n)$$
$$\sqrt{(logn + 1)} = \Omega(\log\log n)$$
$$\sin(n) = \Omega(1)$$
$$\cos n/n = O(1).$$

4. Ποιός από τους υπεροπτικούς συμβολισμούς που εξετάσαμε είναι μεταβατικός; δηλ. αν $f = O(g)$ και $g = O(h)$ ισχύει $f = O(h)$;

5. Δείξτε ότι αν $f = o(g)$ τότε υπάρχει μία συνάρτηση h τέτοια ώστε $f = o(h)$ και $h = o(g)$. Δώστε τον ορισμό της h συναρτήσει των f και g.

6. Δίνονται οι συναρτήσεις :
$$2^{\sqrt{n}}, e^{\log n^3}, n^{3.01}, 2^{n^2}, n^{1.6}, \log n^3 + 1, (\log\log n)^3, \sqrt{n}!$$
Γράψτε τις συναρτήσεις αυτές με τέτοια σειρά ώστε κάθε μια να είναι ίση με το o της επόμενής της.

7. Βρείτε μια συνάρτηση $f(x)$ τέτοια ώστε η σχέση $f(x) = O(x^{1+\varepsilon})$ να είναι ορθή για κάθε $\varepsilon > 0$ αλλά η $f(x) = O(x)$ να μην είναι ορθή.

8. Δείξτε ότι

$$t(n) = O(t(n))$$
$$c \times O(t(n)) = O(t(n))$$
$$O(f(n) + g(n)) = O(max(f(n), g(n)))$$
$$O(t(n) + t(n)) = O(t(n))$$
$$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$$

9. Δείξτε ότι:

   α) $f(n) = \Theta(g(n))$ αν και μόνο αν $f(n) = O(g(n))$ και $f(n) = \Omega(g(n))$

   β) $f(n) = \omega(g(n))$ αν και μόνο αν $g(n) = o(f(n))$

   γ) $f(n) = O(g(n))$ αν και μόνο αν $g(n) = \Omega(f(n))$

   δ) $max(f(n), g(n)) = \Theta(f(n) + g(n))$

10. Ποιά από τις σχέσεις ισχύει;

$$2^{n+1} = O(2^n), \quad 2^{2n} = O(2^n).$$

11. Δείξτε ότι ισχύει:

$$\log(n!) = \Theta(n \log n), \quad n! = o(n^n)$$

12. Δείξτε αν οι συναρτήσεις

$$\lceil \log n \rceil! \quad \textit{και} \quad \lceil \log \log n \rceil!$$

φράσσονται από κάποιο πολυώνυμο.

13. Αν f,g είναι ασυμπτωτικά θετικές συναρτήσεις δείξτε ποιές από τις σχέσεις είναι αληθείς.

   a) $f(n) + g(n) = \Theta(min(f(n), g(n))$

   b) $f(n) = O(g(n))$ συνεπάγεται $\log(f(n)) = O(\log(g(n)))$ όπου $\log(g(n)$ 0 και $f(n) > 1$ για n αρκετά μεγάλο.

   c) $f(n) = O(g(n))$ συνεπάγεται $2^{f(n)} = O(2^{g(n)})$.

14. Δείξτε ότι η συνάρτηση $e^{\sqrt{n}+2n}/(n^{49} + 37)$ είναι εκθετική.

1.4. Ασυμπτωτική συμπεριφορά αθροισμάτων

Πολύ συχνά στην αξιολόγηση ενός αλγόριθμου η πολυπλοκότητα χρόνου εκφράζεται υπό μορφή ενός αθροίσματος το οποίο συνήθως είναι δύσκολο να το υπολογίσουμε. Για το λόγο αυτό τις περισσότερες φορές προσεγγίζουμε τ' αθροίσματα αυτά υπολογίζοντας ένα φράγμα τους. Παρακάτω θα περιγράψουμε μερικές τεχνικές υπολογισμού τέτοιων φραγμάτων.

1.4.1. Με επαγωγή

Εστω οτι θέλουμε να δείξουμε οτι ισχύει: $\sum_{i=0}^{n} 2^i = O(2^n)$.

Αρκεί να δείξουμε ότι υπάρχει c και n_0:

$\sum 2^i \leq c2^n$, για κάθε $n \geq n_0$.

Πράγματι για $i = 0$ έχουμε: $\sum 2^i = 1 \leq c \times 1$ για κάποιο $c \geq 1$. Εστω οτι ισχύει για $i = 0, ..., n$. δηλ.

$$\sum_{i=0}^{n} 2^i \leq c2^n$$

Θα δείξουμε ότι ισχύει και για $i = n + 1$.

$$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} \leq c2^n + 2^{n+1} = c2^{n+1}(\frac{1}{2} + \frac{1}{c}) \leq c2^{n+1}$$

για $(1/2) + (1/c) \leq 1$, ή $c \geq 2$.

Προσοχή. Οταν χρησιμοποιούμε επαγωγή θα πρέπει να προσέξουμε η σχέση να ισχύει για κάθε n και για c=σταθερό.

1.4.2. Με αντικατάσταση

Πολλές φορές μπορούμε να βρούμε ένα καλό πάνω φράγμα αντικαθιστώντας κάθε όρο του αθροίσματος μ'ενα φράγμα, για παράδειγμα

$$\sum \alpha_i \leq \sum \alpha_{max}$$

όπου $\alpha_i \leq \alpha_{max}$ για όλα τα $i = 1, ..., n$.

Η μέθοδος αυτή όμως θα μπορούσε να μας δώσει ένα πολύ κακό φράγμα όταν το άθροισμα (ή σειρά) που δίνεται φράσσεται από μια φθίνουσα γεωμετρική σειρά. Για παράδειγμα έστω ότι έχουμε:

$\sum_{i=0}^{n} \alpha_i$ και ισχύει ότι: $\alpha_{i+1}/\alpha_i \leq r$ για κάθε i και $r < 1$.

Τότε $\alpha_i \leq \alpha_0 r^i$ και το άθροισμα φράσσεται από μία φθίνουσα γεωμετρική σειρά.

$$\sum \alpha_i \leq \sum \alpha_0 r^i = \alpha_0 \sum r^i = \frac{\alpha_0}{1-r}.$$

Ασκηση. Υπολογίστε ένα φράγμα για την σειρά $\sum_{i=1}^{\infty} \frac{i}{3^i}$.

Ισχύει οτι:

$$\frac{\alpha_{i+1}}{\alpha_i} = \frac{1}{3} \times \frac{i+1}{i} \leq \frac{2}{3} \text{ για } \text{ κάθε } i \geq 1$$

Συνεπώς $\alpha_{i+1} \leq 1/3(2/3)^i$ και $\sum i/3^i \leq 1$.

Προσοχή Για να δείξουμε οτι μία σειρά φράσσεται από μια φθίνουσα γεωμετρική σειρά θα πρέπει να δείξουμε ότι: $\alpha_{i+1}/\alpha_i \leq r < 1$ για όλα τα i.

Ασκηση. Δείξτε ότι:

$$\sum_{i=0}^{\infty} \frac{i^2}{2^i} = O(1)$$

$\alpha_{i+1}/\alpha_i = (i+1)^2/2i^2 \leq 8/9$ για $i \geq 3$

$$\alpha_{i+1} = \frac{8}{9}\alpha_i = \frac{8}{9} \times \frac{8}{9}\alpha_{i-1} = ... = \frac{9}{8} \times (\frac{8}{9})^{i-2}$$

Συνεπώς

$$\sum_{i=0}^{\infty} \frac{i^2}{2^i} = 1/2 + 1 + \sum_{i=3}^{\infty} \frac{i^2}{2^i} = O(1)$$

1.4.3. Με σπάσιμο του αθροίσματος

Ενας άλλος τρόπος να υπολογίσουμε ένα φράγμα για ένα αθροισμα είναι να σπάσουμε το αθροισμα σε δύο ή περισσότερα αθροίσματα και να υπολογίσουμε ξεχωριστά φράγματα για καθένα από τα αθροίσματα αυτά. Για παράδειγμα έστω ότι θέλουμε να υπολογίσουμε ένα κάτω φράγμα για το άθροισμα

$$\sum_{i=1}^{n} i$$

Ενας τρόπος είναι ν'αντικαταστήσουμε τους όρους του αθροίσματος με τον ελάχιστο όρο έχουμε $\sum i \geq \sum 1 = n$. Το φράγμα αυτό όμως είναι πολύ μακριά από το πάνω φράγμα που είναι n^2. Ενα καλύτερο φράγμα υπολογίζεται ως εξής:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n/2} i + \sum_{i=n/2+1}^{n} i \geq n/2 + (n/2)^2 = \Omega(n^2)$$

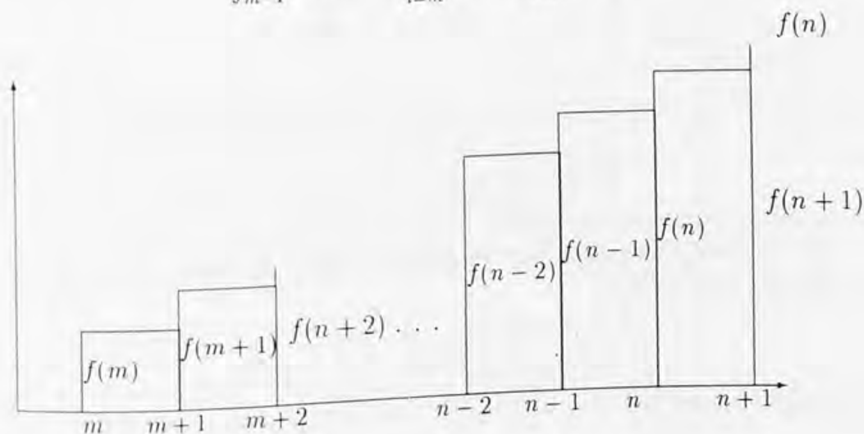
1.4.4. Προσέγγιση αθροίσματος με ολοκλήρωμα

Η γενική ιδέα είναι να συγκρίνουμε τ'αθροίσματα με ολοκληρώματα τα οποία κατά κανόνα υπολογίζονται ευκολότερα. Εστω για παράδειγμα ότι θέλουμε να υπολογίσουμε το άθροισμα:

$$\sum_{i=m}^{n} f(i)$$

όπου f είναι μία μή φθίνουσα συνάρτηση με πεδίο ορισμού το N^+. Τότε μπορούμε να προσεγγίσουμε το άθροισμα από τα ολοκληρώματα:

$$\int_{m-1}^{n} f(x)dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m}^{n+1} f(x)dx$$



Σχήμα 1.4

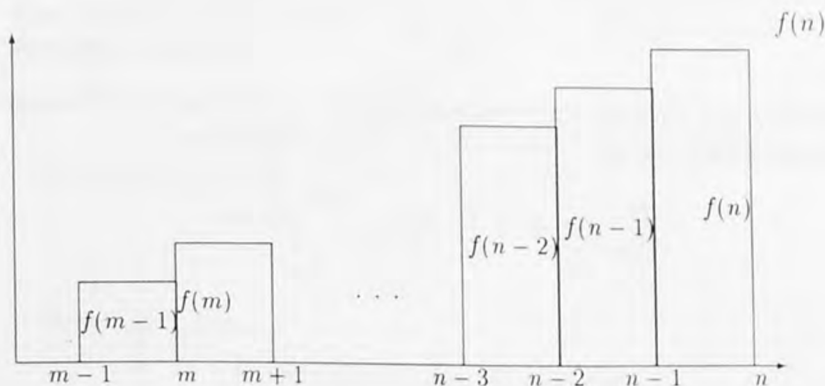
Η ισχύς της σχέσης αυτής φαίνεται στα σχήματα 1.4, 1.5.

Αν προσεγγίσουμε το ολοκλήρωμα

$$\int_{m-1}^{n} f(x)dx$$

με το άθροισμα των εμβαδών των ορθογωνίων του σχήματος 1.4 έχουμε δείξει το δεξιό σκέλος της ανισότητας.

Ομοια αν προσεγγίσουμε το ολοκλήρωμα με το άθροισμα των εμβαδών των ορθογωνίων, μέρος των οποίων τώρα βρίσκεται πάνω από την f(x) όπως φαίνεται στο σχήμα 1.5 τότε έχουμε δείξει το αριστερό σκέλος της ανισότητας.



Σχήμα 1.5

Με τον ίδιο τρόπο μπορούμε να δείξουμε ότι αν f είναι μία μη αύξουσα συνάρτηση τότε ισχύει:

$$\int_{m}^{n+1} f(x)dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)dx$$

Παραδείγματα.

1. Υπολογίστε την ασυμπτωτική συμπεριφορά του αθροίσματος:

$$\sum_{i=1}^{n} i^2$$

Η $f(i) = i^2$ είναι μία μονότονα αύξουσα συνάρτηση συνεπώς ισχύει ότι:

$$\int_{m-1}^{n} x^2 dx \leq \sum i^2 \leq \int_{m}^{n+1} x^2 dx$$

ή

$$n^3/3 \leq \sum i^2 \leq (n+1)^3/3 - 1/3.$$

2. Υπολογίστε την ασυμπτωτική συμπεριφορά του αθροίσματος:

$$\sum_{i=1}^{n} \log i$$

Απάντηση.

$$nlogn - n \leq \sum \log i \leq (n+1)\log(n+1) - n$$

ή

$$(n/e)^n \leq n! \leq (n+1)^{n+1}/e^n$$

Ασκήσεις

1. Εξετάστε επαγωγικά αν ισχύει $\sum_{i=1}^{n} i = O(n)$
2. Δείξτε ότι $\sum_{i=1}^{n} 1/i^2$ φράσσεται από μία σταθερά.
3. Μπορείτε να χρησιμοποιήσετε την μέθοδο προσέγγισης με ολοκληρώματα για τον υπολογισμό του n-οστού αρμονικού αριθμού

$$H_n = \sum_{i=1}^{n} 1/i;$$

4. Υπολογίστε την ασυμπτωτική συμπεριφορά των αθροίσματων για $r \geq 0$ και $s \geq 0$:
$\sum_{i=1}^{n} i^r$ $\quad \sum_{i=1}^{n} log^s i$ $\quad \sum_{i=1}^{n} i^r \log^s i.$

Κεφάλαιο 2

ΑΝΑΔΡΟΜΙΚΑ ΠΡΟΒΛΗΜΑΤΑ

2.1. Μέθοδος "Διαίρει και Βασίλευε"

Ενας αλγόριθμος λέγεται αναδρομικός όταν καλεί τον εαυτό του. Ενα αναδρομικό πρόβλημα μπορεί να εκφραστεί μ'ενα αναδρομικό αλγόριθμο όταν μπορεί ν'αναχθεί σ'ενα ή περισσότερα πανομοιότατα προβλήματα μικροτέρου όμως μεγέθους των οποίων γνωρίζουμε ή μπορούμε εύκολα να υπολογίσουμε την λύση π.χ. $n! = (n-1)! * n, \quad 0! = 1$. Οι αναδρόμικοί αλγόριθμοι ακολουθούν την τεχνική του διαίρει και βασίλευε (Divide and Conquer) . Σύμφωνα με την τεχνική αυτή μπορούμε να πούμε ότι ένας αναδρομικός αλγόριθμος περιλαμβάνει τρία βήματα.

Διαίρει : Διαίρεσε το πρόβλημα σ'ένα αριθμό υποπροβλημάτων.

Βασίλευε : Λύσε τα υποπροβλήματα αναδρομικά. Οταν το μέγεθος των υποπροβλημάτων είναι αρκετά μικρό τότε μπορούν να λυθούν εύκολα.

Συγχώνευσε : Συγχώνευσε τις λύσεις των υποπροβλημάτων για να βρείς τη λύση του αρχικού προβλήματος.

Τα βήματα αυτά, δοθέντος ενός προβλήματος P, μπορούμε να τα περιγράψουμε σε ψευδο-Pascal ως εξής:

```
if P διαιρείται σε μικρότερα προβλήματα then
   begin
      διαίρεσε το P σε δύο ή περισσότερα υποπροβλήματα P1, P2, ..., Pn;
      λύσε P1; λύσε P2; ... λύσε Pn;
```

συγχώνευσε τις n μερικές λύσεις σε μια λύση του P
end
else λύσε το P

Οταν η συγχώνευση των μερικών λύσεων είναι πολύ πιό απλή διαδικασία απο το να λύσουμε το P απ'ευθείας τότε η τεχνική του διαίρει και βασίλευε μας δίνει πολύ αποτελεσματικούς αλγορίθμους. Για να υπολογίσουμε την πολυπλοκότητα χρόνου του αναδρομικού προβλήματος P συνήθως εκφράζουμε την πολυπλοκότητά του ως συνάρτηση της πολυπλοκότητας των μικροτέρων υποπροβλημάτων $P_1, P_2, ..., P_n$. Ενας αναδρομικός αλγόριθμος μπορεί να περιγραφτεί μ'ένα επαναληπτικό σχήμα (recurrence relation) το οποίο μας δίνει το χρόνο εκτέλεσης ενός προβλήματος μεγέθους n συναρτήσει του χρόνου εκτέλεσης των μικροτέρων υποπροβλημάτων. Γενικά ένα επαναληπτικό σχήμα είναι μία εξίσωση ή ανισότητα που περιγράφει μία συνάρτηση συναρτήσει των τιμών της για ένα μικρότερο σύνολο δεδομένων εισόδου π.χ. αν $f(n) = n!$ τότε $f(n) = f(n-1) \times n$. Στο κεφάλαιο αυτό θα εξετάσουμε μεθόδους επίλυσης επαναληπτικών σχημάτων. Το πρώτο βήμα θα είναι να εκφράσουμε το αναδρομικό πρόβλημα μ'ενα επαναληπτικό σχήμα και μετά να λύσουμε το επαναληπτικό σχήμα εκφράζοντας την λύση του υπό την μορφή ενός ασυμπτωτικού συμβολισμού O, Ω, Θ κλπ. Τα βήματα που θ'ακολουθήσουμε είναι τα εξής:

Εστω ότι $T(n)$ είναι η πολυπλοκότητα χρόνου για ένα πρόβλημα μεγέθους n. Οταν το n είναι μικρό ($n \leq c$) τότε η λύση του προβλήματος απαιτεί σταθερό χρόνο δηλ. $\Theta(1)$. Εστω ότι διαιρούμε το πρόβλημα σε a υποπροβλήματα καθένα των οποίων ισούται με το $1/b$ του μεγέθους του αρχικού προβλήματος. Αν $f(n)$ είναι ο χρόνος συγχώνευσης των λύσεων των υποπροβλημάτων τότε έχουμε το επαναληπτικό σχήμα

$$T(n) = \begin{cases} \Theta(1) & \text{για } n \leq c \\ aT(n/b) + f(n) & \text{διαφορετικά} \end{cases} \tag{2.1}$$

2.1.1. Επαναληπτική μέθοδος

Η επαναληπτική μέθοδος βασίζεται στο ανάπτυγμα του επαναληπτικού σχήματος δηλαδή να το εκφράσουμε υπό μορφή ενός αθροίσματος που εξαρτάται μόνο από το n (μέγεθος του προβλήματος) και τις αρχικές συνθήκες (initial values ή boundary conditions). Αν έχουμε ένα επαναληπτικό σχήμα που

εκφράζεται μέσω μιας συνάρτησης f(n) τότε ως αρχικές συνθήκες εννοούμε την τιμή της f για ένα συγκεκριμμένο μέγεθος του n πχ. $f(n) = f(n-1) \times n$ με αρχική συνθήκη $f(0) = 0! = 1$.

Παράδειγμα 1. Εστω ότι θέλουμε να πολλαπλασιάσουμε δύο πίνακες μεγέθους nxn. Ο πολλαπλασιασμός αυτός μπορεί να γίνει πολλαπλασιάζοντας 4 πίνακες μεγέθους $(n/2) \times (n/2)$. Αν $M(n)$ δηλώνει το πλήθος των πολλαπλασιασμών (*) του γινομένου των πινάκων τότε έχουμε οτι: $M(n) = 4M(n/2)$ ή αν υποθέσουμε ότι $n = 2^s$ τότε $M(s+1) = 8M(s)$ και $M(1) = 1$. Το πρόβλημα τώρα είναι να βρούμε ένα ακριβή τύπο για την άγνωστη συνάρτηση $M(s)$. Αυτή μπορεί να βρεθεί εύκολα αν γράψουμε το $M(s)$ συναρτήσει του $M(s-1)$, μετά το $M(s-1)$ συναρτήσει του $M(s-2)$ κοκ.

$$M(s) = 8M(s-1) = 8^2 M(s-2) = \ldots = 8^s M(1) = 8^s$$

ή

$$M(s) = n^3.$$

(Η σχέση $M(n)$ είναι γνωστή ως κλειστή μορφή της $M(n)$).

Παράδειγμα 2. Εστω οτι έχουμε να πολλαπλασιάσουμε δύο ακεραίους από n ψηφία ο καθένας. Να υπολογιστεί η πολυπλοκότητα χρόνου $T(n)$.

Το $T(n)$ θα υπολογιστεί με βάση το πλήθος των πολλαπλασιασμών δύο αριθμών του ενός ψηφίου (στοιχειώδης πολλαπλασιασμός). Εστω δύο ακέραιοι x, y με n ψηφία ο καθένας όπου $n = 2m$. Τότε οι x,y γράφονται ως

$$x = x_0 + x_1 b^m \quad y = y_0 + y_1 b^m$$

όπου b δηλώνει τη βάση και x_0, x_1, y_0, y_1 είναι ακέραιοι με $m = n/2$ ψηφία. Το γινόμενο xy μπορεί να γραφτεί συναρτήσει των x_0, x_1, y_0, y_1 ως εξής:

$$z = xy = (x_0 + x_1 b^m)(y_0 + y_1 b^m) = x_0 y_0 + (x_0 y_1 + x_1 y_0) b^m + x_1 y_1 b^{2m}$$

και ο υπολογισμός του γίνεται ως εξής:

α. Υπολόγισε το $x_0 y_0$

β. Υπολόγισε το $x_0 y_1 + x_1 y_0$

γ. Υπολόγισε το $x_1 y_1$

δ. Εκτέλεσε $n = 2m$ προσθέσεις του ενός ψηφίου.

Πολλαπλασιασμοί με δυνάμεις του 2 αγνοούνται από την ανάλυση καθόσον αυτοί υλοποιούνται με μια μετακίνηση (shifting) ψηφίων. Συνεπώς για τον υπολογισμό του z θέλουμε 4 πολλαπλασιασμούς αριθμών των m ψηφίων και δύο προσθέσεις αριθμών με 2m ψηφία. Ετσι ο αριθμός των πολλαπλασιασμών μπορεί να δοθεί από το επαναληπτικό σχήμα.

$$T(n) = 4T(n/2) + c \quad \text{και} \quad T(1) = 1 + c = c_1$$

Από τη σχέση αυτή έχουμε:

$$T(n) = 4T(n/2) + c = 4\{4T(n/2^2) + c\} + c = 4^2T(n/2^2) + (4c + c) =$$

$$4^2\{4T(n/2^3) + c\} + 4c + c = 4^3T(n/2^3) + (4^2c + 4c + c) = ...$$

$$= 4^kT(n/2^k) + (4^{k-1} + ... + 4^2 + 4 + 1)c$$

Αν υποθέσουμε ότι $n = 2^k$ ή $k = log_2 n$ τότε έχουμε:

$$T(n) = 4^{\log n}T(1) + \frac{4^{\log n} - 1}{3} = n^2 + \frac{n^2 - 1}{3} = \Theta(n^2)$$

Το 1962 δύο Ρώσοι μαθηματικοί (Karatsuba-Ofman) [Karatsuba, Ofman, Multiplication of multidigit numbers on automata Dokl Acad Nauk SSSR 145(2) 293-294, 1962] επρότειναν ένα νέο τρόπο για τον υπολογισμό του γινομένου που βελτιώνει την πολυπλοκότητα χρόνου. Η μέθοδός τους βασίζεται στις παρακάτω ταυτότητες:

$$x_0y_0 = x_0y_0$$

$$x_0y_1 + x_1y_0 = (x_0 - x_1)(y_1 - y_0) + x_0y_0 + x_1y_1$$

$$x_1y_1 = x_1y_1$$

Από τις σχέσεις αυτές βλέπουμε ότι μπορούμε να υπολογίσουμε τις παραστάσεις στο αριστερό μέλος των ταυτοτήτων με τρείς πολλαπλασιασμούς αριθμών των m ψηφίων και με κάποιες επί πλέον προσθέσεις και αφαιρέσεις. Αν λάβουμε σαν μονάδα της πρόσθεσης ή αφαίρεσης την πρόσθεση δύο αριθμών του ενός ψηφίου (με ή χωρίς κρατούμενο) τότε το γινόμενο δύο αριθμών των n ψηφίων χρειάζεται 3 πολλαπλασιασμούς αριθμών των m ψηφίων συν 4n

μονάδες πρόσθεσης. Ετσι το επαναληπτικό σχήμα για τον υπολογισμό των πολλαπλασιασμών είναι:

$$T(n) = 3T(n/2) + b, \quad T(1) = 1 + b = b_1$$

και έχουμε:

$$T(n) = 3T(n/2) + b = 3^2T(3/2^2) + 3b + b = ... =$$

$$= 3^kT(n/2^k) + b(1 + 3 + ... + 3^{k-1}) = 3^kT(1) + (3^k - 1)/2$$

Αν $n = 2^k$ ή $k = logn$ Τότε

$$T(n) = b_1 3^{\log n} + \frac{3^{\log n} - 1}{2} = O(3^{\log n}) = O(n^{\log 3}) = O(n^{1.59}).$$

2.1.2. Αλλες τεχνικές για τη λύση επαν. σχημάτων (Telescoping)

Μερικές φορές ένα επαναληπτικό σχήμα μπορεί να λυθεί ευκολότερα μετά από κάποιο μετασχηματισμό. Εστω για παράδειγμα

$$\frac{t_n}{n} = \frac{t_{n-1}}{n-1} + c(\frac{1}{n} + \frac{1}{n-1}) \quad t_1 = 0, \quad n \geq 2$$

Γράφοντας την εξίσωση για $n = n, n-1, ..., 3, 2$ έχουμε:

$$\frac{t_n}{n} = \frac{t_{n-1}}{n-1} + c(\frac{1}{n} + \frac{1}{n-1})$$

$$\frac{t_{n-1}}{n-1} = \frac{t_{n-2}}{n-2} + c(\frac{1}{n-1} + \frac{1}{n-2})$$

$$\vdots$$

$$\frac{t_3}{3} = \frac{t_2}{2} + c(\frac{1}{3} + \frac{1}{2})$$

$$\frac{t_2}{2} = \frac{t_1}{1} + c(\frac{1}{2} + \frac{1}{1})$$

και προσθέτοντας κατά μέλη τις εξισώσεις λαμβάνουμε:

$$\frac{t_n}{n} = c(\frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}) + c(1 + \frac{1}{2} + ... + \frac{1}{n-1})$$

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$

$$\frac{t_n}{n} = c(H_n - 1) + cH_{n-1} = c(H_n - 1) + c(H_n - \frac{1}{n})$$

ή

$$\frac{t_n}{n} = 2cH_n - c(1 + \frac{1}{n})$$

ή

$$t_n = 2cnH_n - c(n+1)$$

Η λύση του παραπάνω επαναληπτικού σχήματος ήταν σχετικά εύκολη. Υπάρχουν όπως περιπτώσεις που δεν μπορούμε απ'ευθείας να εφαρμόσουμε την παραπάνω τεχνική. Εστω

$$\begin{cases} P_n s_n - q_n s_{n-1} = r_n \quad n \geq 1 \quad (1) \\ s_0 = r_0 \end{cases} \tag{2.2}$$

όπου τα p_i, q_i, r_i δίνονται.

Αν δεν ισχύει $P_{n-1} = q_n$ τότε αν προσθέσουμε δύο ή περισσότερες από τις (1) το s_{n-1} δεν απαλείφεται. Αυτό μπορούμε να το επιτύχουμε ως εξής. Πολλαπλασιάζουμε την (1) με a_n έτσι ώστε ν'απαλείψουμε το s_{n-1}.

$$a_n p_n s_n - a_n q_n s_{n-1} = a_n r_n$$

$$a_{n-1} p_{n-1} s_{n-1} - a_{n-1} q_{n-1} s_{n-2} = a_{n-1} r_{n-1}$$

Αν

$$a_{n-1} p_{n-1} = a_n q_n \quad \text{ή} \quad a_n = \frac{p_{n-1}}{q_n} a_{n-1}$$

τότε ο όρος s_{n-1} απαλείφεται.

$$a_n = \frac{p_{n-1}}{q_n} a_{n-1} = \frac{p_{n-1} p_{n-2}}{q_n q_{n-1}} a_{n-2} = ... = \frac{p_{n-1} p_{n-2} \cdots p_0}{q_n q_{n-1} \cdots q_1} a_0$$

Επομένως για

$$a_n = \frac{\Pi_{i=0}^{n-1} p_i}{\Pi_{i=1}^{n} q_i}$$

μπορούμε να εφαρμόσουμε την παραπάνω τεχνική.
Ασκηση. Να λυθεί η $s_n - 7s_{n-1} = 18(4^{n-1})$ $s_0 = 0$
(**Απάντηση**: $s_n = 6(7^n - 4^n)$, $a_n = 1/7^n$)

2.2. Εφαρμογές

1. **Δυαδικό ψάξιμο**

Εστω ότι έχουμε ένα πίνακα (P: array $[l...u]$ of τύπος_στοιχείων) και ψάχνουμε για ένα συγκεκριμένο στοιχείο.
Λύση: Αν $u - l + 1 = 1$ ή $u = l$ τότε με μια σύγκριση ευρέθη το στοιχείο που ψάχνουμε $T(1) = 1$.
Αν $u - l + 1 > 1$ ή $u > l$ τότε χωρίζουμε το διάστημα στη μέση και συνεχίζουμε με τον ίδιο τρόπο αναδρομικά το ψάξιμο στο μισό διάστημα.
Επομένως:

$$T(n) = T(n/2) + c \quad n > 1 \quad T(1) = 1$$

$$T(n) = T(n/2) + c = T(n/2^2) + 2c = ... = T(n/2^k) + kc$$

Αν $n = 2^k$ ή $k = \log_2 n$ τότε έχουμε

$$T(n) = 1 + kc = 1 + c\log n = O(\log_2 n)$$

2. **Υπολογισμός του minmax ενός πίνακα**

Γράψτε μια διαδικασία για τον υπολογισμό του μέγιστου και ελάχιστου στοιχείου σ'ένα πίνακα s από n στοιχεία. Υπολογίστε την πολυπλοκότητα του αλγορίθμου σας.
Λύση:

```
procedure maxmin (s:πίνακας; l,u: integer; max, min: τύπος_
                    στοιχείου);

var max1, max2, min1, min2: τύπος στοιχείου; m: integer;
begin
  if l=u  then   begin
                    max: = s[l];
                    min: = s[l]
                  end
  else
     begin
       m: = (l+u) div 2;
       maxmin (s,l,m, max1, min1);
       maxmin (s,m+1,u,max2, min2);
       if max1>max2  then  max: = max1  else  max:= max2;
       if min1<min2  then  min: = min1  else  min:= min2;
     end

end
```

Πολυπλοκότητα: Αν ο πίνακας έχει 1 στοιχείο θα κάνουμε μία σύγκριση και δύο εκχωρήσεις: $T(1) = 3$. Αν ο πίνακας έχει περισσότερα από ένα στοιχεία τότε

$$T(n) = 2T(n/2) + 7$$

$T(n) = 2T(n/2) + 7 = 2\{2T(n/2) + 7\} + 7 = 2^2T(n/2^2) + 2 \times 7 + 7 =$
$= \ldots = 2^kT(n/2^k) + 2^{k-1} \times 7 + \ldots + 2 \times 7 + 7 = 2^kT(n/2^k) + 7 \times (2^k - 1)$
Αν $n = 2^k$ τότε $T(n) = 3n + 7n - 7 = 10n - 7 = O(n)$.

3. Οι πύργοι του Ανόϊ

Στο μεγάλο ναό του Benares υπάρχουν τρείς στύλοι. Σ'ένα από αυτούς υπάρχουν περασμένοι από μια τρύπα στο κέντρο τους 64 δίσκοι που η διάμετρός τους μικραίνει από κάτω προς τ'απάνω. Ενας Ινδουϊστής μοναχός προσπαθεί να μεταφέρει όλους τους δίσκους σ'ένα άλλο στύλο τηρώντας τους παρακάτω κανόνες.

1. Σε οποιαδήποτε στιγμή όλοι οι δίσκοι (εκτός φυσικά από τον μεταφερόμενο) πρέπει να είναι τοποθετημένοι σ'ένα από τους στύλους.
2. Σε μια δοσμένη στιγμή μεταφέρεται ένας μόνο δίσκος.
3. Δεν επιτρέπεται να τοποθετηθεί ένας δίσκος πάνω σε μικρότερο από αυτόν τον δίσκο. Οταν τελειώσει η μεταφορά των δίσκων θα έρθει η συντέλεια του κόσμου!

Να λυθεί το πρόβλημα των "πύργων του Ανόϊ".

Λύση: Η λύση του προβλήματος είναι να τυπώσουμε όλες τις μετακινήσεις των δίσκων. Χρειάζεται λοιπόν μια γενική αναδρομική διαδικασία η οποία να χρησιμοποιηθεί για την μετακίνηση οποιουδήποτε αριθμού από δίσκους από ένα στύλο σ'ένα άλλο στύλο μέσω ενός βοηθητικού στύλου.

```
if N=1 then
    μετακίνησε δίσκο από τον στύλο 1 στον 3
else
begin
    μετακίνησε N − 1 δίσκους από τον στύλο 1 στον 2 μέσω του 3
    μετακίνησε τον Nιστό δίσκο από τον στύλο 1 στον 3
    μετακίνησε N − 1 δίσκους από τον στύλο 2 στον 3 μέσω του 1
end
```

Πράγματι ο μόνος τρόπος για να μετακινηθεί ο δίσκος της βάσης (N- ιοστός) του αρχικού στύλου ώστε ν'αποτελέσει την βάση ενός νέου πύργου είναι πρώτα να μετακινηθούν οι προηγούμενοι (N-1) δίσκοι που βρίσκονται πάνω του. Η πολυπλοκότητα του αλγόριθμου είναι:

$$T(n) = 2T(n-1) + 1, \quad T(1) = 1$$

$$T(n) = 2T(n-1)+1 = 2\{2T(n-2)+1\}+1 = 2^2T(n-2)+2\times 1+1 =$$
$$= 2^{n-1}T(1) + (2^{n-2} + \ldots + 2 + 1) = 2^{n-1} + (2^{n-1} - 1) = 2^n - 1$$

Επομένως για $N = 64$ αν υποθέσουμε ότι ο μοναχός μετακινεί 1 δίσκο/sec τότε χωρίς καμμία διακοπή μέρα-νύχτα θα τελειώσει το πρόβλημά του σε

$$2^{64} - 1 \;\; sec = 5.85 \times 10^{11} \text{ χρόνια!!}$$

Ασκήσεις

1. Γράψτε το πρόγραμμα Pascal που τυπώνει όλες τις μετακινήσεις για την λύση του προβλήματος των πύργων του Ανόϊ όταν N = 4, 5, 6.

2. Υπολογίστε την ελάχιστη μικρότερη ακολουθία κινήσεων που μεταφέρει ένα πύργο από n δίσκους από τον αριστερό στύλο A στον δεξιό B εάν απ'ευθείας κινήσεις από τον A στον B απαγορεύονται. Κάθε κίνηση από τον A στον B πρέπει να γίνεται μέσω ενός άλλου ενδιάμεσου στύλου. Ως συνήθως ένας μεγαλύτερος δίσκος ποτέ δεν μπορεί να βρεθεί πάνω από ένα μικρότερο.

3. Ενας διπλός πύργος του Ανόϊ περιέχει 2n δίσκους οι οποίοι ανά δύο είναι ίσοι.

   α) Πόσες κινήσεις απαιτούνται για να μεταφέρουμε ένα διπλό πύργο από ένα στύλο σ'έναν άλλο όταν οι στύλοι του ίδιου μεγέθους δεν διαφοροποιούνται μεταξύ τους;

   β) Ομοια πόσες κινήσεις απαιτούνται όταν στο πύργο προορισμού θέλουμε να βρίσκονται οι δίσκοι ακριβώς με την ίδια σειρά συμπεριλαμβανομένων και των δίσκων ίσου μεγέθους. Υποτίθεται ότι πάντα μεταφέρουμε ένα δίσκο και ποτέ ένας μεγαλύτερος δίσκος δεν τοποθετείται πάνω σ'ένα μικρότερο.

2.3. Μέθοδος αντικατάστασης

Για ορισμένους αλγόριθμους όσο αυτό φαίνεται παράξενο μπορούμε να υποθέσουμε την πολυπλοκότητά τους. Αυτό μπορεί να οφείλεται σε διάφορους λόγους π.χ. πείρα από άλλα παρόμοια προβλήματα ή μπορεί να υποθέσει κανείς την αναμενόμενη πολυπλοκότητα από την κατασκευή του αλγόριθμου. Στη περίπτωση αυτή για να δείξουμε ότι η υποτιθέμενη λύση είναι πράγματι ορθή εφαρμόζουμε τελεία επαγωγή.

Για παράδειγμα έστω ότι θέλουμε να δείξουμε ότι: $T(n) = O(n \log n)$ όπου $T(n) = 2T(\lfloor n/2 \rfloor) + n$. $T(1) = 1$.
Αρκεί να δείξουμε ότι υπάρχει $c, n_0 > 0 : T(n) \leq cn \log n \quad \forall n \geq n_0$.
Εστω ότι η σχέση ισχύει για $\lfloor n/2 \rfloor$ δηλαδή

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \times \log(\lfloor n/2 \rfloor)$$

Τότε:

$$T(n) \leq 2T(\lfloor n/2 \rfloor) + n \leq 2c\lfloor n/2 \rfloor \times \log\lfloor n/2 \rfloor + n \leq$$

$$cn\log(n/2) + n = cn\log n - cn + n \leq cn\log n - (c-1)n \leq cn\log n$$

για κάθε $c \geq 1$.

Τέλος θα δείξουμε ότι η λύση ισχύει και για τις οριακές τιμές. Δυστυχώς δεν μπορούμε να επιλέξουμε ένα c: $T(1) \leq c\log 1 = 0$ Το πρόβλημα αυτό ξεπερνιέται εύκολα γιατί εδώ μας ενδιαφέρει να δείξουμε ότι $T(n) \leq cn\log n$ ισχύει για κάθε $n \geq n_0$. Ετσι μπορούμε να θεωρήσουμε μεγαλύτερες τιμές του n ως οριακές τιμές π.χ. $T(2) = 4$, $T(3) = 5$. Για τις τιμές αυτές παρατηρούμε ότι ισχύουν $T(2) \leq c \times 2 \times \log 2$ και $T(3) \leq c \times 3\log 3$ για $c \geq 2$.

Η γενική μορφή των επαναλ. σχέσεων που εξετάσαμε παραπάνω είναι:

$$S(n) = aS(\frac{n}{b}) + d(n) \quad S(1) = 1$$

όπου n είναι το μέγεθος του προβλήματος το οποίο χωρίζεται σε a υπο-προβλήματα μεγέθους (n/b) το καθένα, $a \geq 1$, $b > 1$ και ο όρος d(n) παριστά τον χρόνο "συνένωσης" των λύσεων των υποπροβλημάτων για να υπολογιστεί η λύση του αρχικού προβλήματος. Στη σχέση (1) υποθέτουμε χωρίς περιορισμό της γενικότητας ότι το n είναι ακεραία δύναμης του b. Ετσι λοιπόν με συνεχείς αντικαταστάσεις στο δεξιό μέλος της (1) έχουμε:

$$S(n) = aS(\frac{n}{b}) + d(n) = a^2S(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n) = \ldots =$$

$$= a^kS(1) + \sum_{i=0}^{k-1} a^i d(n/b^i) = a^k + \sum_{i=0}^{k-1} a^i d(n/b^i) \quad (2)$$

Επειδή $n = b^k$ ή $k = \log_b n$ έχουμε ότι: $a^k = a^{\log_b n} = n^{\log_b a}$ ή $S(n) = O(n^{\log_b a})$.

Σε αναλογία με την ορολογία στις διαφορικές εξισώσεις ο όρος a^k του $S(n)$ ονομάζεται ομογενής λύση και αντιστοιχεί στην ακριβή λύση του $S(n)$ όταν το $d(n) = 0$ για κάθε n. Επειδή το $d(n)$ παριστά τον χρόνο "συνένωσης" των υποπροβλημάτων αν $d(n) = 0$ τότε αυτό σημαίνει ότι το κόστος συνένωσης

είναι μηδέν. Ο δεύτερος όρος του $S(n)$ ονομάζεται ειδική λύση. Ο όρος αυτός εξαρτάται από τον αριθμό των υποπροβλημάτων a και από το $d(n)$. Συνεπώς όταν ψάχνουμε για τυχόν βελτιώσεις στον σχεδιασμό αλγορίθμων θα πρέπει να εξετάσουμε προσεκτικά την αλληλοσχέτιση μεταξύ των δύο όρων του $S(n)$. Για παράδειγμα αν η ομογενής λύση είναι μεγαλύτερη από το $d(n)$ τότε τυχόν βελτίωση του $d(n)$ δεν βελτιώνει την απόδοση του αλγόριθμου. Στη περίπτωση αυτή θα πρέπει να εξετάσουμε μήπως μπορούμε να μεγαλώσουμε το a δηλ. να διαιρέσουμε το πρόβλημα σε λιγότερα ή μικρότερα (n/b) υποπροβλήματα. Μόνο έτσι θα βελτιωθεί το $S(n)$. Μια ειδική λύση της (2) είναι δύσκολο να υπολογιστεί γενικώς ακόμη και όταν το $d(n)$ είναι γνωστό. Παρακάτω θα εξετάσουμε την περίπτωση που το $d(n)$ είναι μία γραμμική συνάρτηση του n. Εστω $d(n) = cn$ τότε

$$\sum_{i=0}^{k-1} a^i d(n/b^i) = \sum_{i=0}^{k-1} a^i \frac{cn}{b^i} = cn \sum_{i=0}^{k-1} (a/b)^i$$

Διακρίνουμε τρείς περιπτώσεις:

α. $a < b$ Τότε $cn \sum_i (a/b)^i = O(n)$

και

$$T(n) = n^{\log_b a} + \Theta(n) = O(n)$$

β. $a = b$ Τότε: $cn \sum 1^i = cnk = cn \log_b n$ και

$$T(n) = n^{\log_b a} + cn \log_b n = \Theta(n \log_b n)$$

γ. $a > b$ Τότε:

$$cn \sum (a/b)^i = cn \frac{(a/b)^k - 1}{(a/b) - 1} = \Theta(n^{\log_b a})$$

και

$$T(n) = n^{\log_b a} + O(n^{\log_b a}) = O(n^{\log_b a})$$

Ασκηση. Ομοια με την παραπάνω ανάλυση υπολογίστε την ειδική λύση στη περίπτωση που $d(n) = n^{\alpha}$ όπου $\alpha > 1$.

Κεφάλαιο 3

ΤΑΞΙΝΟΜΗΣΗ

3.1. Γενικά

Το πρόβλημα της ταξινόμησης είναι πολύ παλιό πρόβλημα και η λύση του έχει μελετηθεί εξωνυχιστικά. Το πρόβλημα είναι σημαντικό τόσο από θεωρητικής όσο και από πρακτικής άποψης. Ενα μεγάλο μέρος της επεξεργασίας στοιχείων για εμπορικές εφαρμογές αποτελείται από ταξινόμηση μεγάλων αρχείων. Συνεπώς αποδοτικοί αλγόριθμοι παίζουν σημαντικό ρόλο από οικονομικής άποψης.

Ορισμός Εστω ένα σύνολο A και μία σχέση R επί των στοιχείων του A. Θα λέμε ότι η R είναι μία σχέση μερικής διάταξης (partial order) όταν για κάθε $a, b \in A$ ισχύουν:

1. aRa (η σχέση R είναι ανακλαστική)

2. Αν aRb και bRc τότε aRc (μεταβατική)

3. Αν aRb και bRa τότε $a = b$ (αντισυμμετρική)

Παράδειγμα: $A = I$ (ακέραιοι) και $R \equiv' \leq'$. Θα λέμε ότι το σύνολο A είναι γραμμικώς ή ολικώς διατεταγμένο (linearly or totally ordered) όταν (A, R) είναι μερικώς διατεταγμένο και για κάθε $a, b \in A$ είτε aRb είτε bRa. Το πρόβλημα της ταξινόμησης περιγράφεται ως εξής: Δοθέντων n στοιχείων $a_1, ..., a_n$ από ένα σύνολο A το οποίο έχει μια ολική διάταξη $' \leq'$ να ευρεθεί μία μετάθεση π των n στοιχείων η οποία ν'απεικονίζει την δοθείσα ακολουθία σε μία μη αύξουσα ακολουθία $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ έτσι ώστε $a_{\pi(i)} \leq a_{\pi(i+1)}$ για

$1 \le i \le n-1$. Οι μέθοδοι ταξινόμησης χωρίζονται σε εσωτερικές (internal) και εξωτερικές (external) με βάση αν τα στοιχεία είναι στη RAM ή σε δίσκο. Η εξωτερική ταξινόμηση έχει να κάνει με αρχεία και για το λόγο αυτό θα εξετάσουμε τέτοιες μεθόδους στο μάθημα Δομές Αρχείων. Η εσωτερική ταξινόμηση είναι σημαντική τόσο στον σχεδιασμό αλγορίθμων όσο και σε εμπορικές εφαρμογές. Συνήθως στην εσωτερική ταξινόμηση το πλήθος των προς ταξινόμηση στοιχείων είναι μικρό ώστε να χωρούν στη κύρια μνήμη. Ωστόσο όμως στην ανάλυσή μας παρακάτω θα εξετάσουμε την συμπεριφορά των αλγορίθμων ταξινόμησης όταν το πλήθος των στοιχείων n τείνει στο ∞. Παρακάτω θα χρησιμοποιήσουμε την τεχνική του "διαίρει και βασίλευε" για την ανάπτυξη αλγορίθμων ταξινόμησης. Για παράδειγμα, έστω A ένας πίνακας και θέλουμε να διατάξουμε τα στοιχεία του σύμφωνα με μία σχέση διάταξης. Διαιρούμε τον πίνακα A σε δύο ίσου μεγέθους υποπίνακες A1, A2, τους ταξινομούμε και μετά τους "συγχωνεύουμε" για να φτιάξουμε τον ταξινομημένο πίνακα A. Ετσι ο αλγόριθμος της ταξινόμησης είναι:

```
if μέγεθος(A) > 1 then
  begin
    split(A, A1, A2);
    sort(A1);
    sort(A2);
    merge(A1, A2, A)
  end
```

Στον παραπάνω αλγόριθμο η διαδικασία split ορίζει την θέση που θα γίνει το χώρισμα του πίνακα A η οποία στην περίπτωσή μας είναι το μεσαίο στοιχείο του πίνακα ($A[(l+u)/2)]$). Στη περίπτωση αυτή η διαδικασία merge υλοποιεί την ένωση δύο συνόλων. Μια άλλη λύση θα ήταν να χωρίσουμε τον πίνακα A σε δύο υποπίνακες $A1$, $A2$ έτσι ώστε τα στοιχεία του $A1$ να είναι $\le$ των στοιχείων του $A2$. Στη περίπτωση αυτή λέμε ότι η διαδικασία split βασίζεται στο περιεχόμενο ή τα στοιχεία του A και η merge υλοποιεί την συνένωση (concat) των $A1$, $A2$. Στην πρώτη περίπτωση το χώρισμα του πίνακα ήταν μια εύκολη διαδικασία ενώ η σύνθεση (merge) δεν ήταν τόσο απλή. Στη δεύτερη περίπτωση συμβαίνει ακριβώς το αντίθετο. Ετσι λοιπόν χωρίζουμε τους αλγόριθμους ταξινόμησης σε δύο κατηγορίες.

α) Easy split/Hard join

β) Hard spit/Easy join

Και οι δύο αυτές κατηγορίες μπορούν να βελτιωθούν. Ετσι έχουμε διάφορους αλγόριθμους ταξινόμησης καθένας με τα δικά του χαρακτηριστικά και ιδιότητες.

3.2. Mergesort

Ο αλγόριθμος αυτός ανήκει στη πρώτη κατηγορία. Ετσι λοιπόν χωρίζουμε τον πίνακα A χρησιμοποιώντας ένα στοιχείο οδηγό, έστω p: οπότε $A1 = \{\alpha_1, ...\alpha_p\}$ και $A2 = \{\alpha_{p+1}, ..., \alpha_n\}$. Οταν ως p λάβουμε το μέσο του $[l..u]$ τότε έχουμε την μέθοδο ταξινόμησης γνωστή ως mergesort. Η πολυπλοκότητα χρόνου της mergesort υπολογίζεται από την σχέση:

$$T(n) = 2T(n/2) + cn \qquad T(1) = 1$$

$$T(n) = 2T(n/2) + cn = 2\{2T(n/2^2) + cn/2\} + cn = 2^2T(n/2^2) + 2cn/2 + cn =$$

$$= 2^3T(n/2^3) + 2^2cn/2^2 + 2cn/2 + cn = ... = 2^kT(1) + kcn =$$

$$2^k + ckn = n + cn\log n = O(n\log n).$$

Η πολυπλοκότητα χρόνου είναι πολύ καλή, αλλά ο αλγόριθμος απαιτεί ένα επιπλέον πίνακα μεγέθους ίσου με τον A για την διαδικασία της συγχώνευσης των υποπινάκων $A1$, $A2$. Αρα η πολυπλοκότητα χώρου είναι $2n + O(\log n)$ όπου ο δεύτερος όρος είναι ο χώρος που απαιτείται για την στοίβα της αναδρομής. Συνεπώς για μεγάλα n θα θέλαμε ένα καλύτερο αλγόριθμο από πλευράς χώρου. Πρέπει να σημειωθεί εδώ ότι ο αλγόριθμος έχει την ίδια πολυπλοκότητα για την καλύτερη την μέση και την χειρότερη περίπτωση.

3.3. Απ' ευθείας εισαγωγή (Insertion sort)

Στη περίπτωση αυτή επιλέγουμε ως $p = u - 1$ έτσι έχουμε δύο υποπίνακες $A1 = [l..u-1]$ και $A2[u..u]$ μ'ένα μόνο στοιχείο. Η πολυπλοκότητα του αλγόριθμου είναι:

$$T(n) = T(n-1) + O(1) + O(n)$$

Ο όρος O(1) αντιστοιχεί στην ταξινόμηση του $A2$ και ο όρος $O(n)$ στη συγχώνευση. Αρα:

$$T(n) = T(n-1)+O(1)+O(n) = T(n-2)+O(1)+O(n-1)+O(1)+O(n) =$$

$$T(n-2)+2O(1)+O(n-1)+O(n) =$$

$$T(1)+(n-1)O(1)+O(2)+...+O(n) = O(n^2).$$

Για τον υπολογισμό ενός κάτω φράγματος στη καλύτερη περίπτωση μπορούμε να απλοποιήσουμε την ανάλυση μετατρέποντας την αναδρομική λύση σε επαναληπτική. Μία μή αναδρομική λύση της ταξινόμησης με απ' ευθείας εισαγωγή είναι:

```
for i:= 2  to n do
  εισάγαγε A[i]  στη σωστή θέση στην ταξινομιμένη
  ακολουθία A[1],..., A[i-1];
for  i:= 2  to n do
 begin
(1)   j:= i-1;  e:= A[i];
      while (A[i] > e) and (j > 1)  do
      begin
        A[j+1]:= A[j];
        j:= j-1
      end;
      if j=1  then
        if A[j] > e then
          begin
            A[j+1]:= A[j];
(2)         j:= j-1
          end;
        A[j+1]:= e
      end
```

Στην προηγούμενη (αναδρομική λύση) η πολυπλοκότητα στη καλύτερη και τη μέση περίπτωση είναι πάντα $O(n^2)$. Δεν συμβαίνει όμως το ίδιο στη μή αναδρομική λύση όπως θα δούμε παρακάτω. Στη πολυπλοκότητα χρόνου θα θεωρήσουμε ως μέτρο χρόνου το πλήθος των συγκρίσεων και μεταφορών (shift)

στοιχείων. Οπως βλέπουμε και από το πρόγραμμα στην i φάση του αλγορίθμου το νέο στοιχείο e μπαίνει στη θέση $A[i]$ και μετά το συγκρίνουμε με τα στοιχεία $A[i-1]$, $A[i-2], \ldots$ μετακινώντας τα στοιχεία αυτά μια θέση δεξιά αν $e < A[j] = i-1, \ldots$ μέχρις ότου βρούμε ένα στοιχείο $A[j] < e$. Τότε το e μπαίνει στη θέση $j+1$. Ετσι λοιπόν το πλήθος των συγκρίσεων είναι ανάλογο του πλήθους των στοιχείων που είναι μεγαλύτερα από το i − οστό στοιχείο. Συνεπώς:

$$C = \sum_{i=1}^{n} (1 + d_i)$$

όπου d_i είναι το πλήθος των στοιχείων που είναι μεγαλύτερα από το $A[i]$ στον πίνακα $A[1], \ldots, A[i-1]$. Μία επιπλέον σύγκριση απαιτείται για να σταματήσει το loop. Αρα:

$$C = n - 1 + \sum_{i=2}^{n} d_i$$

Ομοια οι εκχωρήσεις {1}, {2} απαιτούν χρόνο $3(n-2)$. Αρα συνολικά έχουμε:

$$T(n) = 3(n-2) + (n-1) + \sum d_i = 4n - 7 + \sum d_i$$

Αν $d_i = 0$ δηλαδή ο πίνακας είναι ήδη ταξινομημένος τότε $T(n) = O(n)$ (best case). Οταν $d_i = i-1$ δηλαδή όταν τα στοιχεία του πίνακα είναι σε αντίστροφη σειρά από αυτή που θέλουμε τότε

$$T(n) = 4n - 7 + \sum_{i=2}^{n} (i-1) = 3n - 7 + n(n+1)/2 = O(n^2)$$

Για την μέση πολυπλοκότητα αν υποθέσουμε οτι όλες οι μεταθέσεις $(n!)$ των στοιχείων έχουν την ίδια πιθανότητα να συμβούν, τότε $d_i = (i-1)/2$ και συνεπώς

$$T(n) = O(n^2)$$

Αρα ο αλγόριθμος μπορεί να προτιμηθεί όταν ο πίνακας που θέλουμε να ταξινομήσουμε είναι σχεδόν ταξινομημένος.

Ασκηση: Δώστε ένα αλγόριθμο που βελτιώνει την πολυπλοκότητα του αλγόριθμου της απ'ευθείας εισαγωγής ως προς τις συγκρίσεις.(binary insertion).

3.4. Quicksort

Παρακάτω θα εξετάσουμε μεθόδους ταξινόμησης οι οποίες βασίζονται στο χώρισμα του αρχικού συνόλου $A = \{\alpha_1, ..., \alpha_n\}$ σε δύο υποσύνολα $A1 = \{\alpha_{i1}, \alpha_{i2}, ...\alpha_{im}\}$ και $A2 = \{\alpha_{im+1}, ..., \alpha_{in}\}$ τέτοια ώστε $a_{ij} < a_{ik}$ $1 \leq j \leq m$, $m + 1 \leq k \leq n$ όπου $(i_1, ..., i_n)$ είναι μία μετάθεση του $(1, ..., n)$. Οταν τα δύο σύνολα $A1$, $A2$ είναι περίπου ίσα σε μέγεθος τότε έχουμε την μέθοδο ταξινόμησης του Quicksort. Στη περίπτωση αυτή η διαδικασία "merge" δεν χρειάζεται και ο αλγόριθμος είναι:

```
quicksort: if |A| > 1 then
           begin
             split(A, A1, A2);
             quicksort(A1);
             quicksort(A2);
           end
```

Η διαδικασία split αποτελείται από ένα συστηματικό πέρασμα του πίνακα A "ταυτόχρονα" και από τα δύο του άκρα. Πρώτα επιλέγουμε ένα στοιχείο οδηγό α_p και μετά σχηματίζουμε τα υποσύνολα $A1$, $A2$ έτσι ώστε όλα τα στοιχεία του $A1$ να είναι μικρότερα του α_p και όλα τα στοιχεία του $A2$ να είναι μεγαλύτερα ή ίσα του α_p. Ενας απλός τρόπος να επιτύχουμε αυτό είναι:

1. Πέρασε όλα τα στοιχεία από το αριστερό άκρο του A μέχρις ότου βρείς ένα στοιχείο $\alpha_i \geq x = \alpha_p$.
2. Πέρασε όλα τα στοιχεία από το δεξιό άκρο του A μέχρις ότου βρείς ένα στοιχείο $\alpha_j < x = \alpha_p$.
3. Αλλαξε αμοιβαία τα α_i και α_j.
4. Επανέλαβε το βήμα 1 από τη θέση $i + 1$ και το βήμα 2 από τη θέση $j - 1$ μέχρις ότου οι δείκτες i και j διασταυρωθούν.
5. Τέλος άλλαξε αμοιβαία τα α_j και α_p

Στο τέλος της διαδικασίας αυτής το στοιχείο α_p έχει έρθει στη θέση του στον ταξινομημένο πίνακα A. Το χώρισμα αυτό του αρχικού πίνακα επιτρέπει την ταξινόμηση στο επόμενο βήμα του καθενός υποπίνακα $A1$ και $A2$ χωριστά. Για παράδειγμα έστω ότι έχουμε να ταξινομήσουμε τους αριθμούς:

3 9 6 2 5 8 1 7 4

και επιλέγουμε ως στοιχείο οδηγό το $\alpha[1] = 3$. Εκτελούμε τα βήματα 1 και 2 του αλγόριθμου με αρχικές τιμές των δεικτών $i = 1$ και $j = n + 1$, $(n = 9)$, οπότε έχουμε:

βήμα 1,2:	3	9	6	2	5	8	1	7	4
		$i\uparrow$					$\uparrow j$		
βήμα 3:	3	1	6	2	5	8	9	7	4
		$i\uparrow$					$\uparrow j$		
βήμα 1,2:	3	1	6	2	5	8	1	7	4
			$i\uparrow$	$\uparrow j$					
βήμα 3:	3	1	2	6	5	8	9	7	4
			$i\uparrow$	$\uparrow j$					
βήμα 1,2:	3	1	2	6	5	8	1	7	4
			$j\uparrow$	$\uparrow i$					

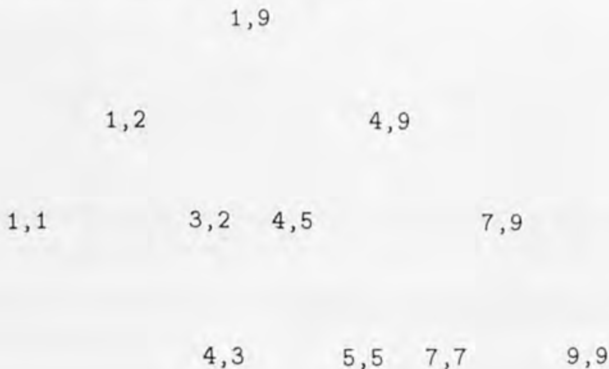
βήμα 5: [2 1] 3 [6 5 8 9 7 4]

Το στοιχείο 3 έχει πάει στη θέση του και συνεχίζουμε τον αλγόριθμο με την ταξινόμηση του $A1 = \{2,1\}$ και του $A2 = \{6,5,8,9,7,4\}$.

```
procedure quicksort(l,u:integer);
var
   i,j,t:integer;
begin
   if l < u  then
     begin
        i:= l; j:= u+1;
        repeat
          repeat i:=i+1 until (a[i] >= a[l])  or (i>=n);
          repeat j:= j-1 until (a[j] < a[l])  or (j<=l);
          if i<j  then
             begin
```

```
                t:= a[i];
                a[i]:= a[j];
                a[j]:= t
              end
        until i>j;
        t:= a[l];
        a[l]:= a[j];
        a[j]:= t;
        quicksort(l, j-1);
        quicksort(j+1, u)
    end {if l<u}
end {quicksort}
```

Για το παραπάνω παράδειγμα αρχικά η διαδικασία καλείται με παραμέτρους (1,9). Στο παρακάτω δένδρο δίνονται όλα τα καλέσματα της quicksort μέχρις ότου το σύνολο A να είναι ταξινομημένο.



Μία προδεδιατεταγμένη διάσχιση του δένδρου αυτού μας δίνει τη σειρά με την οποία εκτελούνται οι διαδικασίες της αναδρομής. Στο παράδειγμά μας το στοιχείο οδηγός βάσει του οποίου γίνεται το χώρισμα της ακολουθίας σε δύο άλλες μικρότερες ήταν το αριστερότερο στοιχείο. Αυτό όμως μπορεί ν'αποβεί "ολέθρια" επιλογή όταν η αρχική μας ακολουθία ήταν ήδη ταξινομημένη ή ήταν σχεδόν ταξινομημένη. Επομένως ένας βελτιωμένος αλγόριθμος quicksort θα ήταν καλό να επιλέγει ως στοιχείο οδηγό το μέσο στοιχείο από ένα δείγμα π.χ.

τριών στοιχείων. Επιλέγουμε λοιπόν ως στοιχείο οδηγό το μεσαίο στοιχείο από τα: $\alpha[l]$, $\alpha[(l+u)/2]$, $\alpha[u]$. Εστω ότι η ταξινομημένη τριάδα είναι η: α_μ, α_m, α_M. Τότε ανακατανέμουμε τα στοιχεία αυτά ως εξής:

$$\alpha[l] \longleftrightarrow \alpha_m$$

$$\alpha[(l+u)/2] \longleftrightarrow \alpha[l+1]$$

$$\alpha[l+1] \longleftrightarrow \alpha_\mu$$

$$\alpha[u] \longleftrightarrow \alpha_M$$

ή σε Pascal:

```
swap(a[(l+u) div 2], a[l+1]);
if a[l+1] > a[u]  then  swap(a[l+1], a[u]);
if a[l] > a[u]  then  swap(a[l], a[u]);
if a[l+1] > a[l]  then  swap(a[l+1], a[l])
```

Παρατηρούμε ότι με αυτό τον τρόπο τα στοιχεία α_μ, α_M και $\alpha[l]$ πάνε στα σωστά υποσύνολα και το $\alpha[l]$ στη σωστή του θέση. Επίσης οι μεταβλητές i, j τώρα ξεκινούν από τις τιμές $l+1$ και u αντί για l, $u+1$ το οποίο σημαίνει ότι εξοικονομούμε δύο ακόμη συγκρίσεις σε κάθε βήμα. Μια τελευταία βελτίωση του αλγόριθμου θα ήταν να γράψουμε το μη αναδρομικό πρόγραμμα. Αυτό μπορεί να γίνει διότι το μέγεθος της στοίβας είναι μικρότερο του logn αν το μικρότερο υποσύνολο ταξινομείται πρώτο.

3.5. Πολυπλοκότητα του quicksort

Είπαμε ότι τα A1, A2 είναι περίπου ίσα σε μέγεθος. Αυτό όμως δεν συμβαίνει πάντα γιατί εξαρτάται από την επιλογή του στοιχείου οδηγού. Για να υπολογίσουμε την πολυπλοκότητα χρόνου ας δούμε πρώτα την διαδικασία split. Στη διαδικασία αυτή κάνουμε ένα πέρασμα του συνόλου από αριστερά και δεξιά μεχρις ότου τα δύο αυτά περάσματα συναντηθούν κάπου στη μέση. Συνεπώς θέλουμε χρόνο $O(n)$ όπου n είναι το πλήθος των στοιχείων. Η πολυπλοκότητα στη καλύτερη περίπτωση επιτυγχάνεται όταν $|A1| = |A2| = n/2$. Αυτό συμβαίνει όταν το στοιχείο οδηγός είναι το μέσο (median) στοιχείο του A. Στη περίπτωση αυτή το δένδρο των κλήσεων της διαδικασίας θα έχει ύψος logn και σε κάθε επίπεδο i του δένδρου θα έχουμε 2^{i-1} καλέσματα της διαδικασίας split

για υποσύνολα μεγέθους $n/2^{i-1}$ το καθένα. Αρα η πολυπλοκότητα θα είναι $\log n(2^{i-1}n/2^{i-1}) = n\log n$.

Η πολυπλοκότητα στη χειρότερη περίπτωση επιτυγχάνεται όταν το στοιχείο οδηγός χωρίζει το αρχικό σύνολο δύο υποσύνολα $|A_1| = 1$ και $|A_2| = n-1$. Τώρα το δένδρο των κλήσεων έχει ύψος n και σε κάθε επίπεδο i έχω ένα κάλεσμα της split σε σύνολο μεγέθους $(n-i+1)$. Αρα ο συνολικός χρόνος θα είναι:

$$n + (n-1) + (n-2) + ... + 1 = n(n-1)/2 = O(n^2)$$

Δηλαδή στη χειρότερη περίπτωση η "γρήγορη ταξινόμηση" (quicksort) στην πραγματικότητα είναι πολύ αργή.

Παρακάτω θα δείξουμε ότι η πολυπλοκότητα στη μέση περίπτωση είναι $O(n\log n)$. Για το σκοπό αυτό θα υποθέσουμε ότι όλα τα στοιχεία του πίνακα A είναι διαφορετικά μεταξύ τους. Επίσης θα υποθέσουμε ότι η επιλογή του στοιχείου οδηγού γίνεται τυχαία έτσι ώστε κάθε στοιχείο έχει την ίδια πιθανότητα $(1/n)$ για να επιλεγεί ως οδηγός. Εστω ότι μετά την επιλογή του στοιχείου οδηγού το μέγεθος του ενός υποπίνακα είναι m_p και του άλλου $n - m_p$, $1 \leq m_p \leq n$. Τότε ο μέσος χρόνος για την ταξινόμηση των δύο υποπινάκων θά είναι:

$$P_{m1}(T(m_1 - 1) + T(n - m_1)) + ... + P_{mn}(T(m_n - 1) + T(n - m_n))$$

όπου $P_{mi} = 1/n$ είναι η πιθανότητα να επιλέξουμε το στοιχείο m_i ως οδηγό. Συνεπώς η πολυπλοκότητα στη μέση περίπτωση θα είναι:

$$T(n) = n - 1 + 1/n\sum_{i=1}^{n}((T(m_i - 1) + T(n - m_i)), \quad T(0) = 0 \quad (1)$$

Αλλά

$$\sum_{i=1}^{n} T(m_i) = T(1) + T(2) + ... + T(n-1) = \sum_{i=1}^{n} T(n - m_i)$$

Συνεπώς η (1) γράφεται

$$T(n) = (n-1) + 2/n\sum_{i=1}^{n-1} T(_i), \quad T(0) = 0$$

ή

$$nT(n) = n(n-1) + 2\sum_{i=1}^{n-1} T(i)$$

για $(n-1)$ η (2) γράφεται:

$$(n-1)T(n-1) = (n-1)(n-2) + 2\sum_{i=1}^{n-2} T(i) \qquad (3)$$

Αφαιρώντας τις (2) και (3) κατά μέλη έχουμε:

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1)$$

ή

$$nT(n) - (n+1)T(n-1) = 2(n-1)$$

ή

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2(n-1)}{n(n+1)} = \frac{-2}{n} + \frac{4}{n+1}$$

Εφαρμόζοντας την τεχνική του telescoping έχουμε

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2(n-1)}{n(n+1)} = \frac{-2}{n} + \frac{4}{n+1}$$

$$\frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} = \frac{-2}{n-1} + \frac{4}{n}$$

.

.

$$\frac{T(1)}{2} - \frac{T(0)}{1} = \frac{-2}{-1} + \frac{4}{2}$$

Προσθέτοντας τις εξισώσεις αυτές έχουμε ότι

$$T(n)/(n+1) = -2\{1 + 1/2 + ... + 1/n\} + 4\{1/2 + 1/3 + ... + 1/(n+1)\}$$

Αν $H_n = 1 + 1/2 + ... + 1/n$ (n-οστός αρμονικός αριθμός) τότε

$$T(n)/n + 1 = -2H_n + 4(H_n - 1 + 1/n + 1)$$

$$T(n) = 2(n+1)H_n - 4n$$

Ισχύει ότι $H_n = lnn + \gamma + O(1/n)$ όπου $\gamma = 0.5772156649$ η σταθερά Euler
Αρα $T(n) = 2(n+1)lnn + O(n) = 1.386n \log n + O(n)$

Σημείωση: Οταν τα στοιχεία που θέλουμε να ταξινομήσουμε είναι λίγα τότε ένας απλός αλγόριθμος ταξινόμησης είναι γρηγορότερος από το quicksort γιατί οι αναδρομικές κλήσεις είναι μια χρονοβόρος διεργασία. Ετσι λοιπόν το quicksort μπορεί να βελτιωθεί ως εξής:

if μέγεθος(A) <= min_size **then**
  insertion_sort(A)
else
  quicksort(A)

Η τιμή του min_size εξαρτάται από διάφορους παράγοντες όπως π.χ. την μηχανή, τον μεταγλωττιστή. Πειραματικά έχει δειχθεί ότι $5 \leq$ min_size ≤ 15.

```
procedure quicksort(ll,rr:integer; var a:pin);
label 9,99;
var
  M:integer; (* size of subfiles *)

  i,j,l,r,zz:integer;
  sv,w:longint;
  done:boolean;
  stack:stack_type;
begin
  M:=9;
  (* initialize stack *)
  stack.top := 0;
```

```
l:=ll; r:=rr; done:=(r-l) < M;
while not done do
begin
  i:=l+1;
  w := a[(l+r) div 2];
  a[(l+r) div 2]:=a[i];
  a[i]:=w;
  if (a[i] > a[r]) then
  begin  w:=a[i]; a[i]:=a[r]; a[r]:=w; end;
  if (a[l] > a[r])  then
  begin  w:=a[l]; a[l]:=a[r]; a[r]:=w; end;
  if (a[i] > a[l])  then
  begin  w:=a[i]; a[i]:=a[l]; a[l]:=w; end;

  j:=r; sv:=a[l];

  goto 9;
  99:
    w:=a[i]; a[i]:=a[j]; a[j]:=w;
   9:
    repeat i:=i+1; until (a[i] >= sv);
    repeat j:=j-1; until (a[j] <= sv);
  if (j >= i) then goto 99;
  w:=a[l]; a[l]:=a[j]; a[j]:=w;

  if (j-l) > (r-i+1) then zz:= j-l else zz:=r-i+1;

  if zz <= M then
  begin
    if  stack.top = 0 then (*empty(stack)*)
      done:=true
    else
    begin (*pop(stack,&l,&r)*)
      stack.top:=stack.top-1;
      l := stack.pairs[stack.top].il;
      r := stack.pairs[stack.top].jr;
    end
```

```
  end
  else
  begin
     if (j-l) < (r-i+1) then zz:= j-l else zz:=r-i+1;
     if zz <= M then
        if ((j-l) > (r-i+1)) then r:=j-1
        else l:=i
     else
     begin
       if ((j-l) > (r-i+1)) then
       begin  (* push(stack,l,j-1);  *)
          stack.pairs[stack.top].il := l;
          stack.pairs[stack.top].jr := j-1;
          stack.top:=stack.top+1;
          l:=i;
        end
        else
        begin  (* push(stack,i,r);  *)
           stack.pairs[stack.top].il := i;
           stack.pairs[stack.top].jr := r;
           stack.top:=stack.top+1;
           r:=j-1;
        end
     end
  end
end;
for  i:=rr-1 downto ll  do
begin
  if (a[i] > a[i+1]) then
  begin
    sv:=a[i]; j:=i+1;
    repeat
      a[j-1]:=a[j]; j:=j+1;
    until  not (((a[j] < sv) and ( j <= rr))) ;
    a[j-1]:=sv
  end
end
```

```
end;
```

3.6. Ταξινόμηση με επιλογή (Selection sort)

Και η μέθοδος αυτή όπως το quicksort ανήκει στη κατηγορία Hard split/Easy join. Εδώ το χώρισμα του αρχικού συνόλου A γίνεται έτσι ώστε $|A1| = 1$ και $|A2| = n - 1$. Επειδή για κάθε στοιχείο $x \in A1$ και $y \in A2$ θα πρέπει $x \leq y$ το x θα είναι το μικρότερο στοιχείο του A. Ο αλγόριθμος του selection sort περιγράφεται ως εξής:

```
selection_sort: if |A| > 1 then
                  begin
                    split(A, A1, A2);
                    selection_sort(A2)
                  end
```

Η μή αναδρομική διαδικασία είναι:

```
for i := 1 to n - 1 do
begin
    pos:= min(A2);
    swap(A[i], A[pos])
end
```

Η συνάρτηση min απαιτεί ένα πέρασμα του A2

```
pos:= i;
for j := i + 1 to n do
    if A[j] < A[pos] then pos := j;
min := A[pos]
```

Εύκολα βλέπουμε ότι ο αλγόριθμος απαιτεί $O(n)$ αμοιβαίες αλλαγές και $O(n^2)$ συγκρίσεις. Η απόδοση όμως του αλγόριθμου αυτού δεν μπορεί να συγκριθεί με αυτή του insertion sort ή του quicksort. Σε μερικές εφαρμογές για παράδειγμα θέλουμε να ταξινομήσουμε τα πρώτα k μόνο στοιχεία ενός συνόλου. Στη περίπτωση αυτή ο αλγόριθμος έχει πολυπλοκότητα $O(kn)$. Στις περιπτώσεις αυτές ο αλγόριθμος είναι πολύ καλός.

3.7. Heapsort

Ο αλγόριθμος του heapsort που αποτελεί βελτίωση του selection sort έχει περιγραφεί στο μάθημα "Δομές Δεδομένων με Pascal". Εδώ θα υπολογίσουμε μόνο την πολυπλοκότητα του αλγορίθμου.

Πρόταση. Ο αλγόριθμος κατασκευής του σωρού έχει πολυπλοκότητα $O(n)$.
Απόδειξη: Οπως ξέρουμε ένα πλήρες δυαδικό δένδρο έχει N φύλλα έχει $N-1$ εσωτερικούς κόμβους ($n = N + (N-1) = 2N - 1$). Σε κάθε επίπεδο i του δένδρου υπάρχουν 2^{i-1} κόμβοι. Αφού το δένδρο έχει ύψος $h = \log n$ στο τελευταίο επίπεδο θα υπάρχουν $n/2$ στοιχεία. Εχουμε λοιπόν n/2 στοιχεία τα οποία θα πρέπει να κάνουμε "shiftdown". Η διαδικασία shiftdown ορίζεται ως εξής:

procedure shiftdown(A[i], i, n)

όπου $A[i]$ είναι το στοιχείο που θα κάνουμε "shiftdown", i είναι η θέση του στοιχείου αυτού και n είναι το μέγεθος του σωρού. Από τα $n/2$ στοιχείων που πρέπει να γίνουν shiftdown τα 2^{h-i} βρίσκονται στο επίπεδο $(h - i + 1)$. Οταν κάνω shiftdown σ'ένα στοιχείο που είναι στο επίπεδο i τότε θέλω $h - i$ συγκρίσεις. Αρα έχουμε:

Για καθένα από τους 2^{h-2} κόμβους του επιπέδου $h-1$ μία σύγκριση ή 2^{h-2} συγκρίσεις συνολικά. Για καθένα από τους 2^{h-3} κόμβους του επιπέδου $h-2$ δύο συγκρίσεις ή $2 \times 2^{h-3}$ συγκρίσεις συνολικά. Για καθένα από τους 2^{h-4} κόμβους του επιπέδου $h-3$ τρείς συγκρίσεις ή $3 \times 2^{h-4}$ συγκρίσεις συνολικά. κ.ο.κ. Για καθένα από τους 2 κόμβους του επιπέδου 2, $h-2$ συγκρίσεις ή $2(h-2)$ συγκρίσεις συνολικά. Αρα το σύνολο των συγκρίσεων είναι:

$$2^{h-2} + 2 \times 2^{h-3} + 3 \times 2^{h-4} + 4 \times 2^{h-5} + \ldots + (h-2) \times 2 =$$

$$2^h(1/2^2 + 2/2^3 + 3/2^4 + 4/2^5 + \ldots + (h-2)/2^{h-1}) =$$

$$(n+1)/2 \sum_{i=1}^{h-2} i/2^i = O(n)$$

Η ταξινόμηση του σωρού περιγράφεται ως εξής:

```
for i:= n down to 2 do
   begin
     swap(A[i], A[1]);
     shiftdown(A[1], 1, i - 1)
   end
```

Η shiftdown έχει πολυπλοκότητα $O(\log i)$. Αρα η συνολική πολυπλοκότητα είναι:

$$\sum O(\log i) = O(\sum \log i) = O(n \log n)$$

Ασκήσεις:

1. Γράψτε το πρόγραμμα του bubblesort και υπολογίστε την πολυπλοκότητά του.
2. Ομοια για το tree selection

Πόσο γρήγορα μπορούμε να ταξινομήσουμε n στοιχεία μόνο με συγκρίσεις;

Εστω οτι θέλουμε να ταξινομήσουμε n στοιχεία με συγκρίσεις. Αν δίνεται η n-ιάδα $(\alpha_1, \alpha_2, ..., \alpha_n)$ τότε το πιθανό αποτέλεσμα (δηλαδή η ταξινομημένη n-ιάδα) είναι μία από τις $n!$. Σε κάθε μία απο τις n-ιάδες αυτές καταλήγουμε μετά από συγκρίσεις των στοιχείων ανά δύο. Σύνεπώς αν θεωρήσουμε την δοθείσα n-ιάδα ως ρίζα ενός δυαδικού δένδρου τότε μετά από κάθε σύγκριση έχουμε δύο δυνατές περιπτώσεις (δύο παιδιά). Το μονοπάτι από τη ρίζα στα φύλλα του δένδρου θα μας δίνει και μια πιθανή λύση του προβλήματος. Επομένως το δένδρο αυτό θα έχει $n!$ φύλλα. Ενα τέτοιο δένδρο ονομάζεται δένδρο αποφάσεων (decision tree). Επειδή το δένδρο αυτό είναι δυαδικό ισχύει ότι: Ενα δυαδικό δένδρο ύψους h έχει το πολύ 2^{h-1} φύλλα. Ετσι λοιπόν αν ένα δυαδικό δένδρο όπως το δένδρο αποφάσεων έχει $n!$ φύλλα τότε θα έχει ύψος τουλάχιστον $\log(n!) + 1$.

Πρόταση. Ενα δένδρο αποφάσεων που ταξινομεί n διακεκριμένα στοιχεία έχει ύψος τουλάχιστον $\log(n!) + 1$. (Απόδειξη: Αφού το δένδρο έχει $n!$ φύλλα έχουμε $n! = 2^{(\log n! + 1) - 1}$). Το δένδρο αποφάσεων για την ταξινόμηση των στοιχείων $(\alpha_1, \alpha_2, \alpha_3)$ είναι: $(n = 3, n! = 6)$

α1,α2,α3
α1<=α2

α2,α1,α3
α1<=α3

α1,α2,α3
α2<=α3

α2,α3,α1
α2<=α3

α2,α1,α3
3

α1,α3,α2
α1<=α3

α1,α2,α3
6

α3,α2,α1
1

α2,α3,α1
2

α3,α1,α2
4

α1,α3,α2
5

Πρόταση. Κάθε αλγόριθμος που ταξινομεί n στοιχεία μόνο με συγκρίσεις πρέπει να έχει πολυπλοκότητα χρόνου στη χειρότερη περίπτωση $O(n \log n)$. Σύμφωνα με την προηγούμενη πρόταση ο ελάχιστος αριθμός των συγκρίσεων που θα γίνουν από τον αλγόριθμο ισούται με το μήκος του μικροτέρου μονοπατιού από τη ρίζα σε μιά πιθανή λύση (φύλλο). Συνεπώς αν $T(n)$ είναι το πλήθος των συγκρίσεων τότε

$$T(n) = \log n! + 1$$

Αλλά όπως είδαμε: $n! \geq (n/e)^n$ άρα:

$$T(n) = \log n! + 1 \geq n \log n - n \log e = O(n \log n).$$

Ασκήσεις

1. Δείξτε ότι: $\sum_{i=1}^{n} i/2^i = 2 - (n+2)/2^n$
2. Υπολογίστε την πολυπλοκότητα της ταξινόμησης των πρώτων k στοιχείων από μια λίστα με n στοιχεία με heapsort. Για ποιές τιμές του k η πολυπλοκότητα είναι γραμμική;

3. Ποιό αλγόριθμο ταξινόμησης θα επιλέγατε για την ταξινόμηση μιάς συνδεδεμένης γραμμικής λίστας; (με pointers).
4. Εστω ότι δίνεται ένα σύνολο από n στοιχεία. Κατασκευάστε το δένδρο αναζήτησης με τα στοιχεία αυτά. Ποιά είναι η πολυπλοκότητα χρόνου και χώρου;
5. Εστω ότι δίνεται μια ακολουθία από στοιχεία τα οποία είναι ταξινομημένα αλλά στο τέλος υπάρχουν μερικά στοιχεία που δεν είναι ταξινομημένα. Ποιά μέθοδο ταξινόμησης θα επιλέγατε για να ταξινομήσετε αυτή την ακολουθία;
6. Γράψτε ένα πρόγραμμα που υπολογίζει τα k μικρότερα στοιχεία από ένα πίνακα με n στοιχεία. Υπολογίστε την πολυπλοκότητα του αλγόριθμου. Για ποιές τιμές του k ο αλγόριθμος είναι γραμμικός;

3.8. Bin Sort

Ολοι οι αλγόριθμοι που εξετάσαμε μέχρι τώρα βασίζονται σε συγκρίσεις των στοιχείων ανα δύο. Οι μέθοδες αυτές όπως είδαμε έχουν πολυπλοκότητα $O(nlogn)$.Εδώ θα δούμε αλγόριθμουσ που βασίζονται σε κάποια γνώση του κλειδιού ταξινόμησης. Για παράδειγμα έστω οτι θέλουμε να ταξινομήσουμε τον πίνακα $A : array[1..n]$ *of integer* και έστω οτι δεν υπάρχουν επαναλαμβανόμενα κλειδιά. Τοτε με την βοήθεια ενός πίνακα B (του ίδιου μεγέθους με τον A) βάζουμε τα στοιχεία του A στη σωστή τους θέση στον πίνακα B π.χ.

```
for i := 1 to n do
  B[A[i]] := A[i]
```

Ο αλγόριθμος έχει πολυπλοκότητα $O(n)$ και ισχύει εφόσον δεν υπάρχουν επαναλαμβανόμενα στοιχεία στον A. Ενας άλλος αλγόριθμος με την ίδια πολυπλοκότητα που δεν χρησιμοποιεί βοηθητικό πίνακα είναι:

```
for i := 1 to n do
  while (A[i] <> i) do
    swap (A[i], A[A[i]])
```

Σε κάθε επανάληψη του while το στοιχείο $A[i]$ πηγαίνει στη θέση του και δεν ξανακινείται. Συνεπώς ο αλγόριθμος έχει πολυπλοκότητα $O(n)$. Στη περίπτωση που υπάρχουν επαναλαμβανόμενα κλειδιά χρησιμοποιούμε ως βοηθητική δομή ένα πίνακα από λίστες (δεν γνωρίζουμε πόσα επαναλαμβανόμενα κλειδιά υπάρχουν κάθε φορά) π.χ.

```
B:array[1..n] of lists;
list = record
        head,rear:listpointer
       end
```

Συνεπώς για κάθε στοιχείο του A υπάρχει ένα στοιχείο του B (bin). Ο αλγόριθμος είναι:

```
for i := 1 to n do
   εισάγαγε A[i] στη λίστα B[A[i]];
συνένωσε τις λίστες B[i], i = 1,...,n
```

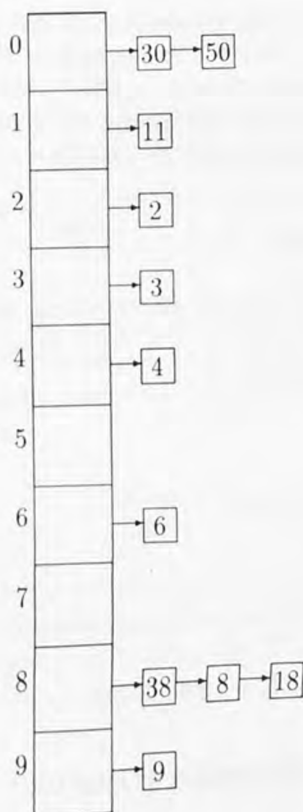
Για ευκολία στη συνένωση σε κάθε λίστα υπάρχει ένας δείκτης στο τέλος. Αν υπάρχουν προς ταξινόμηση n με m διαφορετικές τιμές ($m \leq n$) τότε ο αλγόριθμος απαιτεί χρόνο $O(n+m) = O(n)$. Τι γίνεται όμως όταν $m = n^2$; Τότε ο αλγόριθμος έχει πολυπλοκότητα $O(n+n^2) = O(n^2)$. Παρακάτω θα δούμε ένα αλγόριθμο (radix sort) ο οποίος και σ'αυτή ακόμη την περίπτωση έχει πολυπλοκότητα $O(n)$. Εστω για παράδειγμα οτι θέλουμε να ταξινομήσουμε n ακεραίους στο διάστημα $[0..n^2-1]$ π.χ.

```
38  3  8  11  30  4  18  50  2  6  9
```

($n = 11$ και οι τιμές ανήκουν στο διάστημα $[0..99]$)

Τότε η ταξινόμηση των στοιχείων μπορεί να γίνει με δύο περάσματα των n στοιχείων ως εξής: Εφαρμόζουμε bin sort ως προς το λιγότερο σημαντικό ψηφίο των ακεραίων δηλαδή το τελευταίο. Αφού πρόκειται για ακεραίους οι δυνατές τιμές είναι 0..9. Ετσι έχουμε ένα πίνακα $B : array[0..9]$ *of lists* και εισάγουμε τους αριθμούς στο κατάλληλο bin σύμφωνα με το τελευταίο τους ψηφίο. Βλέπε σχήμα 3.1(a).

Συνενώνουμε τις λίστες οπότε έχουμε την ακολουθία:



Σχήμα 3.1

30 50 11 2 3 4 6 38 8 18 9.

Στην ακολουθία που προέκυψε από το πρώτο πέρασμα επαναλαμβάνουμε τη ίδια διαδικασία για το επόμενο λιγότερο σημαντικό ψηφίο οπότε προκύπτει το σχήμα 3.1(b)

Μετά την συνένωση των λιστών (3.1.b) έχουμε την ταξινομημένη ακολουθία:

2 3 4 6 8 9 11 18 30 38 50.

(Ορθότης αλγορίθμου) Ο αλγόριθμος που περιγράψαμε ταξινομεί σωστά την ακολουθία των στοιχείων στο $[0..\beta^2 - 1]$.

Πράγματι αν θεωρήσουμε δύο ακεραίους $i = a\beta + b$ και $j = c\beta + d$ όπου $a, b, c, d \in [0..\beta - 1]$ θα δείξουμε ότι αν $i < j$ τότε ο i θα προηγείται του j. Αφού $i < j$ τότε $a < c$. Αν $a < c$ τότε ο i θα προηγείται του j από το δεύτερο πέρασμα. Αν $a = c$ τότε θα πρέπει $b = d$. Τότε ο i θα προηγείται του j από το πρώτο πέρασμα αφού στο δεύτερο πέρασμα μπαίνουν στο ίδιο bin ($a = c$).

3.9. Γενίκευση του Radix-sort

Εστω ότι τα στοιχεία που θέλουμε να ταξινομήσουμε αποτελούνται από k διαφορετικά πεδία $f_1, f_2, ..., f_k$ το καθένα τύπου $t_1, t_2, ..., t_k$ π.χ. Εστω για παράδειγμα ότι τα στοιχεία μας είναι τύπου date

```
date=record
        day: 1..31;
        month: JAN..DEC;
        year: 1900..2000
     end
```

$$(t_1 = 1..31, \quad t_2 = JAN..DEC, \quad t_3 = 1900..2000)$$

3.9.1. Λεξικογραφική ταξινόμηση στοιχείων (Ορισμός)

Θα λέμε ότι ένα στοιχείο $(a_1, a_2, ..., a_n)$ προηγείται λεξικογραφικά ενός στοιχείου $(b_1, b_2, ..., b_n)$ όταν υπάρχει $j \in [1, ..., n]$ τέτοιο ώστε $a_i = b_i$ για $i = 1, ..., j - 1$ και $a_j < b_j$. Σύμφωνα με τον ορισμό τα στοιχεία τύπου date μπορούμε να τα δούμε ως year με βάση 100, month με βάση 12 και day με βάση 31. Ετσι εφαρμόζουμε binsort πρώτα ως προς το ελάχιστα σημαντικό πεδίο δηλαδή το πεδίο day και εισάγουμε τα στοιχεία πρώτα σ'ενα πίνακα

B1:**array**[1..31] **of** lists

συνενώνουμε τις λίστες και επαναλαμβάνουμε την διαδικασία μ'ενα πίνακα

B2:**array**[1..12] **of** lists

και τέλος

B3:**array**[1..100] **of** lists

Ετσι ο αλγόριθμος περιγράφεται ως εξής:

```
for i:=k down to 1 do
    begin
        δημιούργησε L_j(j = 1,...,β_i) κενές λίστες
        για κάθε στοιχείο a_m εισάγαγε το a_m
        στο τέλος της λίστας L(v) όπου v είναι η τιμή του πεδίου f_i
        συνένωσε τις λίστες L(j)j = 1,...,β_i
    end
```

Η δημιουργία των λιστών σημαίνει χρόνο $O(\beta_i)$. Η εισαγωγή των στοιχείων του αρχικού πίνακα στις λίστες απαιτεί χρόνο $O(n)$ και τέλος η συνένωση απαιτεί χρόνο $O(\beta_i)$. Συνεπώς ο συνολικός χρόνος του Radix sort είναι:

$$\sum_{i=1}^{k} O(n+\beta_i) = O(kn + \sum_{i=1} k\beta_i) = O(n + \sum_{i=1} k\beta_i)$$

όπου το k είναι μία σταθερά.

Παράδειγμα Εστω ότι έχουμε να ταξινομήσουμε n ακεραίους που βρίσκονται στο διάστημα $[0..\beta^k - 1](\beta = 10)t_i = 0..9$ και$\beta_i = 10$. Τότε θέλουμε χρόνο:

$$O(kn + \sum^{k} 10) = O(kn + 10k) = O(n)$$

3.10. Το πρόβλημα της επιλογής

Πρόβλημα: Δοθείσης μιας ακολουθίας από n στοιχεία και ενός αριθμού k βρές το k-ιοστό στοιχείο στη ταξινομημένη ακολουθία των n στοιχείων. Για παράδειγμα αν $k = 1$ έχουμε το μικρότερο στοιχείο $(O(n))$, για $k = n$ έχουμε το μεγαλύτερο στοιχείο $(O(n))$ και για $k = n/2$ έχουμε το μέσο στοιχείο (median). Οπως είδαμε από τις ασκήσεις με το heapsort μπορούμε να βρούμε τα k πρώτα στοιχεία μιας ακολουθίας με n στοιχεία σε χρόνο $O(k \log n)$. Συνεπώς όταν $k < n/\log n$ τότε ο χρόνος για να βρούμε τα k πρώτα στοιχεία θα είναι $O(n)$. Το ίδιο ισχύει όταν έχουμε την συμμετρική περίπτωση δηλαδή θέλουμε τα k τελευταία στοιχεία. Εδώ θα περιγράψουμε μιά μέθοδο επιλογής (selection) του k-ιοστού στοιχείου από μια ακολουθία n στοιχείων με μια

μέθοδο όμοια με το Quicksort. Η διαδικασία select(i,j,k) επιλέγει το k-ιοστό στοιχείο στον πίνακα a[i..j].

Αλγόριθμος:

1. Επέλεξε το στοιχείο οδηγό, έστω p
2. Χώρισε τον πίνακα $a[i],...,a[j]$ σε δύο υποπίνακες με την διαδικασία split του Quicksort έτσι ώστε: $A1 = \{a[i],...,a[m-1]\}$ τέτοια ώστε $a[l] < a[p]$, $a[l] \in A1$ $A2 = \{a[m],...,a[j]\}$ τέτοια ώστε $a[l] > a[p]$, $a[l] \in A2$
3. Αν $k \leq m - i$ τότε το k-ιοστό στοιχείο του $a[i],...,a[j]$ είναι στον υποπίνακα A1 και καλούμε την select$(i, m-1, k)$ διαφορετικά select$(m, j, k-m+i)$

```
procedure select(var A:pinakas; l,u,k:OriaPinaka; var stoixeio:
                    TupowStoixeiou);
  var i,j,p: integer;
  begin
     if  l<u then
       begin
          split (A,l,u,i,j);
          if k in [j-l+2..i-1] then stoixeio:= A[l+k-1]
          else
          if k<= j-l+1 then select(A,l,u,k,stoixeio)
          else  select(A,i,u,k-i+1)
       end
     else if l=u then
     stoixeio:=A[l+k-1]
  end
```

Οπως και με το Quicksort η select απαιτεί στη χειρότερη περίπτωση χρόνο $O(n^2)$. Για παράδειγμα όταν ζητούμε το πρώτο στοιχείο και ως στοιχείο οδηγό λαμβάνουμε από κακή τύχη πάντα το μεγαλύτερο στοιχείο. Στη μέση περίπτωση η select είναι καλύτερη από την Quicksort διότι καλεί τον εαυτό της μια μόνο φορά. Στη καλύτερη περίπτωση η select καλεί τον εαυτό της για ένα υποπίνακα του μισού μεγέθους απότι ο αρχικός πίνακας.

Κεφάλαιο 4

ΑΝΑΖΗΤΗΣΗ ΣΥΜΒΟΛΟΣΕΙΡΩΝ

4.1. Γενικά

Η συμβολοσειρά είναι μία πεπερασμένη ακολουθία από σύμβολα μέσα από ένα αλφάβητο. Οι πράξεις με συμβολοσειρές αποτελούν τις πλέον κοινές και βασικές στην επεξεργασία κειμένων. Πράγματι αρχεία κειμένων (text files) αποτελούνται απο συμβολοσειρές χαρακτήρων και δυαδικά αρχεία (binary files) από συμβολοσειρές από bits. Εκδότες κειμένων (text editors) επεξεργάζονται αρχεία κειμένων και μεταγλωττιτές παραλαμβάνουν αρχεία κειμένων και παράγουν δυαδικά αρχεία. Στο κεφάλαιο αυτό θα περιοριστούμε στο πρόβλημα της αναζήτησης μιας συμβολοσειράς μέσα σε μία άλλη συμβολοσειρά. Η πράξη αυτή υπάρχει σε όλους τους εκδότες κειμένων.

4.2. Αλγόριθμος Brute-Force

Ζητείται να βρεθεί η πρώτη θέση, έστω k, μιας συμβολοσειράς p μήκους M, σ'ενα κείμενο t το οποίο παριστάνεται ως μία συμβολοσειρά μήκους N, όπου $N >> M$. Η συμβολοσειρά p είναι γνωστή και ως υπόδειγμα (pattern). Οταν βρεθεί η θέση k της p τότε θα ισχύει: $p[i] = t[k+i]$ για $i = 0, 1, ..., M-1$.

Ο αλγόριθμος Brute-Force είναι η πιό απλή (προφανής) μέθοδος που έρχεται στο μυαλό μας δηλαδή να εξετάσουμε για κάθε δυνατή θέση i της συμβολο-

σειράς t αν ισχύει: $t[i+j] = p[j]$ για $j = 0, 1, ..., M-1$

Ο αλγόριθμος σε C είναι:

```
int brutesearch(char *p, char *t)
{
  int i,j, M=strlen(p), N=strlen(t);
  for (i=0,j=0; j<M && i<N; i++, j++)
    while(t[i] != p[i]) { i -=j-1; j=0;}
  if (j==M) return i-M; else return i;
}
```

Πολυπλοκότητα: Ο αλγόριθμος Brute-Force απαιτεί NM συγκρίσεις χαρακτήρων. Η χειρότερη περίπτωση για τον αλγόριθμο συμβαίνει όταν

t="000000 .... 000001" και

p="000001"

Τότε για κάθε μια από τις $N-M+1$ θέσεις στο t θα κάνουμε συγκρίσεις με όλα τα στοιχεία του p. Αρα θα κάνουμε $M(N-M+1)$ συγκρίσεις και επειδή $M << N$ τοπλήθος των συγκρίσεων είναι περίπου NM. Οπωσδήποτε συμβολοσειρές όπως οι παραπάνω δεν υπάρχουν σε καμιά φυσική γλώσσα. Στη πράξη ότι για τυχαίες συμβολοσειρές το πλήθος των συγκρίσεων είναι ανάλογο του N.

Πρόταση Για μία τυχαία συμβολοσειρά p μεγέθους M, ένα τυχαίο κείμενο t μεγέθους N ($M << N$) και ένα αλφάβητο μεγέθους c, ο Brute-Forse αλγόριθμος απαιτεί $\approx cN/(c-1)$ συγκρίσεις στη μέση περίπτωση.

Μία τυχαία συμβολοσειρά μήκους μήκους L είναι μια ακολουθία από L χαρακτήρες μέσα από ένα αλφάβητο που παράγονται ανεξάρτητα και ομοιόμορφα. Η πιθανότητα δύο συμβολοσειρές μήκους L να είναι ίσες είναι $1/c^L$. Ο αλγόριθμος συγκρίνει τους χαρακτήρες $p[i+1]$ και $t[i+1]$ μόνο όταν $p[1..i] = t[1..i]$. Η ισότητα όμως αυτή ισχύει με πιθανότητα $1/c^i$ συνεπώς μία σύγκριση στη θέση $i+1$ θα γίνει με πιθανότητα $1/c^i$ και επομένως ο αναμενόμενος αριθμός συγκρίσεων του αλγόριθμου είναι:

$$1 + \sum_{i=2}^{L} \frac{1}{c^{i-1}} = \frac{c}{c-1}(1 - \frac{1}{c^L}$$

Επειδή ο αλγόριθμος ελέγχει $N-M+1$ ελέγχους συμβολοσειρών μεγέθους

L ο αναμενόμενος αριθμός των συγκρίσεων θα είναι:

$$\frac{c}{c-1}(1-\frac{1}{c^L})(N-M+1)$$

Δύο τυπικά αλφάβητα είναι το δυαδικό και το ASCII. Για το δυαδικό έχουμε ότι η μεση πολυπλοκότητα του αλγορίθμου είναι $\approx 2(N-M+1)$ και με το ASCII είναι $\approx N-M+1$. Επειδή οταν επεξεργαζόμαστε δυαδικά κείμενα (binary texts) μπορεί να έχουμε συμβολοσειρές για τις οποίες ο αλγόριθμος πλησιάζει την χειρότερή του απόδοση θέλουμε ακόμη καλύτερους αλγόριθμους.

4.2.1. Αλγόριθμος των Knuth-Morris-Pratt

Ο αλγόριθμος Brute-Forse αν και έχει πολύ καλή επίδοση στη μέση περίπτωση γίνεται πολύ αργός στη χειρότερη περίπτωση και τούτο διότι ενώ η μεταβλητή i πάντα αυξάνεται κατά ένα ο αλγόριθμος επανεξετάζει τα στοιχεία του t χωρίς να λαμβάνει υπόψη του ορισμένες πληροφορίες από το προηγούμενο βήμα που θα μπορούσαν να βελτιώσουν την επίδοσή του. Στα επόμενα θα εξετάσουμε ένα νέο αλγόριθμο ο οποίος επιτυγχάνει $O(N)$ συγκρίσεις χαρακτήρων στη χειρότερη περίπτωση αλλά απαιτεί κάποιο επιπλέον χώρο και μια προεπεξεργασία πάνω στη συμβολοσειρά p. Ο αλγόριθμος KMP στηρίζεται στη παρατήρηση ότι μετά από μιά αποτυχημένη μερική ταύτηση του p με το t δεν χρειάζεται ν'αρχίσουμε την σύγκριση από τον επόμενο χαρακτήρα του t αλλά να εκμεταλευτούμε την πληροφορία που έχουμε. Με άλλα λόγια να προχωρήσουμε στην συμβολοσειρά t πρός τα δεξιά κατά i+d χαρακτήρες και να επαναλάβουμε την σύγκριση. Φυσικά επιθυμούμε η μετακίνηση προς τα δεξιά d θέσεις να είναι όσο το δυνατο μεγαλύτερη. Εστω για παράδειγμα ότι έχουμε:

```
t:   A B C B A B C A B A B C A B C B A B
p:   A B C A B C B
```

Παρατηρούμε ότι το p ταυτίζεται με το t στους τρείς πρώτους χαρακτήρες και αποτυγχάνει στον τέταρτο. Ετσι στην επόμενη σύγκριση του p με το t δεν θα μετακινήσουμε το p μία μόνο θέση δεξιά αλλά θα εκμεταλευτούμε την γνώση που έχουμε για τους πρώτους τέσσερις χαρακτήρες. Αφού η αποτυχία συνέβει στον τέταρτο χαρακτήρα αυτό σημαίνει ότι ο τέταρτος χαρακτήρας του t δεν

είναι A. Το ερώτημα είναι πόσες θέσεις δεξιά μπορούμε να μετακινήσουμε το p; Την απάντηση την δίνει ο αλγόριθμος KMP. Από το συγκεκριμένο παράδειγμα μπορούμε εύκολα να διαπιστώσουμε ότι το p μπορεί να μετακινηθεί το πολύ 4 θέσεις δεξιά. Πράγματι αφού ο πρώτος χαρακτήρας του p είναι το A δεν μπορεί να ταυτίζεται με τον δεύτερο και τρίτο χαρακτήρα του t (B C) και με τον τέταρτο γιατί στον τέταρτο χαρακτήρα συνέβει η αποτυχία συνεπώς ο χαρακτήρας αυτός δεν είναι A. Ετσι έχουμε στο επόμενο βήμα:

```
t:    A B C B A B C A B A B C A B C B A B
p:            A B C A B C B
```

Εδώ η αποτυχία συμβαίνει στον 6ο χαρακτήρα. Πόσες θέσεις δεξιά θα μετακινηθεί το p; Μια πιθανή περίπτωση είναι να μετακινηθεί το p τρείς θέσεις οπου ταυτίζεται τουλάχιστον ο πρώτος χαρακτήρας. Αλλά αφού η αποτυχία συνέβει στον 6ο χαρακτήρα εχουμε πληροφορίες για 6 χαρακτήρες. Πράγματι αν μετακινήσουμε το p τρείς θέσεις θα έχουμε νέα αποτυχία στον τρίτο χαρακτήρα. Ετσι μετακινούμε το p 5 θέσεις δεξιά αφού στη θέση αυτή έχουμε αποτυχία που σημαίνει ότι ο αντίστοιχος χαρακτήρας του t δεν είναι ο C και επομένως μπορεί να είναι ο A. Εχουμε λοιπόν:

```
t:    A B C B A B C A B A B C A B B B A B
p:                      A B C A B C B
```

Και στη περίπτωση αυτή έχουμε αποτυχία στον 6ο χαρακτήρα. Και πάλι θα μετακινήσουμε το p 5 θέσεις δεξιά. Πράγματι αφού συνέβει αποτυχία στον 6ο χαρακτήρα το μόνο που ξέρουμε είναι ότι ο αντίστοιχος χαρακτήρας του t δεν είναι C, άρα μπορεί να είναι A. Για να δούμε πόσες θέσεις θα μετακινήσουμε το p όταν ξέρουμε ότι συνέβει αποτυχία σε κάποιο χαρακτήρα βασιζόμαστε μόνο στο p. Ετσι λοιπόν προεπεξεργαζόμαστε το p και για κάθε δυνατή θέση αποτυχίας υπολογίζουμε το μεγαλύτερο πλήθος θέσεων που μπορούμε να μετακινήσουμε το p προς τα δεξιά. Ο αλγόριθμος περιγράφεται ως εξής: Οταν συμβεί αποτυχία στον i-οστό χαρακτήρα $\{i = 1, 2, ...strlen(p)\}$ βρές το μεγαλύτερο επίθεμα (suffix) $p[i-j+1, .., i]$ που είναι πρόθεμα (prefix) $p[1, ..j]$ του $p[1, .., i]$.

Παρακάτω δίνουμε τα προγράμματα Pascal του αλγορίθμου {βλέπε R. Sedgewick, Algorithms, σελίδα 246}.

{ t(ext) είναι η συμβολοσειρά (text) στην οποία ψάχνουμε να βρούμε το υπόδειγμα (pattern) συμβολοσειρά, και next είναι ο πίνακας ο οποίος δοθείσης μιας αποτυχίας δίνει το πλήθος των θέσεων που πρέπει να μετακινηθεί το p. M=length(p), N=length(t)}

```
function kmpsearch:integer;
var i,j:integer;
begin
  i:=1; j:=1;
  repeat
   if (j=0) or (t[i]=p[j]) then
      begin
        i:=i+1;
        j:=j+1
      end
    else
      j:=next[j]
   until (j>M) or (i>N);
   if j>M then kmpsearch:=i-M else kmpsearch:=i
 end


procedure initnext;
var i,j:integer;
begin
  i:=1; j:=0; next[1]=0;
  repeat
    if (j=0) or (p[i]=p[j]) then
      begin
        i:=i+1;
        j:=j+1;
        next[i]:=j
      end
    else j:=next[j]
  until i>M;
 end
```

Η σύγκριση αρχίζει από τον τελευταίο χαρακτήρα του p. Επειδή έχουμε αποτυχία θα πρέπει ν'αποφασίσουμε πόσες θέσεις δεξιά θα μετακινηθεί το p για να συνεχίσουμε την σύγκριση. Επειδή ο χαρακτήρας F δεν υπάρχει στο p θα μετακινήσουμε το p 7 θέσεις δεξιά. Αφού συνέβει αποτυχία στον τελευταίο χαρακτήρα του p (T) ξέρουμε ότι ο αντίστοιχος χαρακτήρας του t δεν είναι T, άρα μπορεί να είναι A. Συνεπώς θα μετακινήσουμε το p μία θέση δεξιά διότι έτσι μπορεί τα έχουμε ταύτιση. Τελικά το p μετακινείται τόσες θέσεις οσο είναι η μεγίστη τιμή των δύο παραπάνω τιμών. Ετσι έχουμε:

```
t:   W H I C H _ F I N A L L Y _ H A L T S   _ A T _ T H A T
p:                 A T _ T H A T
```

Και στη περίπτωση αυτή έχουμε αποτυχία με την πρώτη σύγκριση. Τότε θα μετακινήσουμε το p 4 θέσεις δεξιά διότι ενδέχεται να έχουμε ταύτιση στο (_). Επίσης αφού είχαμε αποτυχία στον τελευταίο χαρακτήρα, σίγουρα ο αντίστοιχος χαρακτήρας του t δεν είναι T, άρα μπορούμε να μετακινήσουμε το p μία μόνο θέση δεξιά. Τελικά το p μετακινείται 4 θέσεις και έχουμε:

```
t:   W H I C H _ F I N A L L Y _ H A L T S . _ A T _ T H A T
p:                         A T _ T H A T
```

Τώρα αποτυχία συμβαίνει στον δεύτερο από το τέλος χαρακτήρα. Επειδή το L δεν υπάρχει στο p θα μετακινήσουμε το p 6 θέσεις δεξιά. Επίσης από την αποτυχία ξέρουμε ότι ο αντίστοιχος χαρακτήρας του t δεν είναι A ενδέχεται να έχω ταύτιση για το ζεύγος T, άρα το p μπορεί να μετακινηθεί 3 θέσεις δεξιά. Τελικά το p μετακινείται 6 θέσεις. Παρατηρούμε ότι στον αλγόριθμο BM σε αντίθεση με τον KMP οι χαρακτήρες του t λαμβάνονται υπόψη. Ο αλγόριθμος BM συνοψίζεται ως εξής:

1. Ξεκίνα τις συγκρίσεις χαρακτήρων από το τέλος του p προς τ'αριστερά.
2. Οταν συμβεί αποτυχία σε μία θέση i του p τότε μετακίνησε το p, k θέσεις δεξιά, όπου k είναι το max από τις δύο παρακάτω τιμές.

   α. Η πρώτη τιμή εξαρτάται από την τιμή του χαρακτήρα του t που συνέβει η αποτυχία και την απόσταση από την αρχή του p που υπαρχει ο χαρακτήρας αυτός. Ετσι κατασκευάζουμε ένα πίνακα που κάθε του στοιχείο j περιέχει την απόσταση της μετακίνησης όταν συμβεί

αποτυχία στη θέση αυτή. Δηλαδή για κάθε γράμμα του αλφαβήτου εξετάζουμε αν το γράμμα υπάρχει στο p. Αν δεν υπάρχει τότε το p μετακινείται strlen(p) θέσεις δεξιά διαφορετικά το p μετακινείται m θέσεις, όπου m είναι η απόσταση του χαρακτήρα p[j] από την αρχή του p.

β. Η δεύτερη τιμή εξαρτάται από όλους τους χαρακτήρες του t που ταυτίζονται μερικώς με το p μαζί με τον χαρακτήρα που συνέβει η αποτυχία. Εδώ κατασκευάζουμε ένα δεύτερο πίνακα με τον ίδιο αλγόριθμο όπως στον KMP.

1. Smit G.De V., A comparison of three string matching algorithms, Software Practice and Experience, 12, 1982, 57-66

2. Horsepool R.N., Practical Fast Searching in Strings, Software Practice and Experience, 10, 1980, 501-506.

3. Boyer R.S., Moore J.S., A Fast String Searching Algorithm, Communications of the ACM, 1977, 762-772

Κεφάλαιο 5

ΟΡΟΛΟΓΙΑ ΓΡΑΦΗΜΑΤΩΝ

5.1. Γραφήματα

Ενα **γράφημα** (graph) G αποτελείται από ένα μη κενό πεπερασμένο σύνολο **κόμβων** (ή σημείων) (vertices, points) $V = V(G)$ και ένα σύνολο E μη διατεταγμένων ζευγών από στοιχεία του V. Κάθε ζεύγος $x = [u, v]$, το οποίο είναι στοιχείο του E, ονομάζεται **ακμή** (ή γραμμή) (edge, line) που συνδέει τους κόμβους u και v. Τα σημεία u και v ονομάζονται **γειτονικά** (adjacent) και η ακμή x ονομάζεται προσκείμενη (incident) στους κόμβους u και v. Δύο ακμές που πρόσκεινται στον ίδιο κόμβο ονομάζονται γειτονικές. Ενα γράφημα παριστάνεται συνήθως με ένα διάγραμμα. Για παράδειγμα, στο γράφημα G του σχήματος (5.1) οι κόμβοι v_0 και v_1 είναι γειτονικοί, ενώ οι v_0 και v_3 δεν είναι. Οι ακμές x και y είναι γειτονικές, ενώ οι x και z δεν είναι. Προσέξτε ότι, μολονότι οι ακμές x και z τέμνονται στο διάγραμμα, η τομή τους δεν είναι κόμβος του γραφήματος.

Βαθμός (degree) ενός κόμβου είναι ο αριθμός των ακμών που πρόσκεινται στον κόμβο αυτό. Για παράδειγμα στο γράφημα του σχήματος (5.1) ο κόμβος v_4 έχει βαθμό 2 και οι υπόλοιποι 3. Το άθροισμα των βαθμών όλων των κόμβων ενός γραφήματος υπολογίζεται πολύ εύκολα: Κάθε ακμή πρόσκειται σε δύο κόμβους, άρα για κάθε ακμή προστίθεται στο άθροισμα αυτό ο αριθμός 2. Οπότε το άθροισμα των βαθμών είναι διπλάσιο του αριθμού των ακμών.

Πρόταση Για μία συμβολοσειρά p μεγέθους M και ένα κείμενο t μεγέθους N ο αλγόριθμος KMP απαιτεί το πολύ 2N συγκρίσεις χαρακτήρων στη χειρότερη περίπτωση.

Παρατηρούμε ότι ως αποτέλεσμα μιας σύγκρισης είτε το i αυξάνεται κατά ένα είτε το p μετακινείται προς τα δεξιά. Αφού το i μπορεί ν'αυξηθεί το πολύ N φορές και το p μπορεί να μετακινηθεί το πολύ N φορές συμπεραίνουμε ότι ο αλγόριθμος απαιτεί $2N$ συγκρίσεις χαρακτήρων στη χειρότερη περίπτωση.

Η αποτελεσματικότητα του αλγορίθμου KMP εξαρτάται από το πόσο αποτελεσματικά μπορούμε να υπολογίσουμε την συνάρτηση next. Χρησιμοποιώντας τον Brute-Forse αλγόριθμο για να συγκρίνουμε την συμβολοσειρά p με τα προθέματά της, επειδή η p έχει $M-1$ τέτοια προθέματα η next απαιτεί $O(M^2)$ συγκρίσεις χαρακτήρων στην χειρότερη περίπτωση. Είναι δυνατόν όμως να υπολογίσουμε την next με $O(M)$ συγκρίσεις χαρακτήρων στη χειρότερη περίπτωση χρησιμοποιώντας ένα πολύπλοκο αλγόριθμο που χρησιμοποιεί τον αλγόριθμο KMP. Ο αλγόριθμος KMP είναι πολύ καλύτερος από τον Brute-Force αλγόριθμο ωστόσο όμως στη πράξη συμπεριφέρεται όπως ο Brute-Force αλγόριθμος διότι σπανίως έχουμε συμβολοσειρές t, p με συνεχώς επαναλαμβανόμενα στοιχεία. Επειδή όμως δεν έχει πισογυρίσματα μπορει να χρησιμοποιείται με αρχεία που διαβάζονται από δίσκο.

1. Knuth D.E.,Morris J.H., Pratt V.R., Fast Pattern Matching in Strings, SIAM Journal on Computing, 1977, 323-349

4.2.2. Αλγόριθμος των Boyer-Moore

Στον αλγόριθμο Brute-Forse αλλά και στον KMP μετακινούμεθα και στο υπόδειγμα p και στο κείμενο t συνέεια από αριστερά προς τα δεξιά. Οι Boyer-Moore πρότειναν ένα ενδιαφέρον αλγόριθμο κατά τον οποίο μετακινούμεθα στο μεν υπόδειγμα από δεξιά προς τ'αριστερά στο δε κείμενο από αριστερά προς τα δεξιά. Ενα επιπλέον χαρακτηριστικό του αλγορίθμου των είναι ότι αν παρατηρηθεί ότι ένας χαρακτήρας από το κείμενο δεν υπάρχει στο υπόδειγμα τότε μπορούμε να μετακινήσουμε το υπόδειγμα αμέσως μετά το χαρακτήρα αυτό. Εστω ότι έχουμε:

```
t:   W H I C H _ F I N A L L Y _ H A L T S . _ A T _ T H A T
p:   A T _ T H A T
```

Η σύγκριση αρχίζει από τον τελευταίο χαρακτήρα του p. Επειδή έχουμε αποτυχία θα πρέπει ν'αποφασίσουμε πόσες θέσεις δεξιά θα μετακινηθεί το p για να συνεχίσουμε την σύγκριση. Επειδή ο χαρακτήρας F δεν υπάρχει στο p θα μετακινήσουμε το p 7 θέσεις δεξιά. Αφού συνέβει αποτυχία στον τελευταίο χαρακτήρα του p (T) ξέρουμε ότι ο αντίστοιχος χαρακτήρας του t δεν είναι T, άρα μπορεί να είναι A. Συνεπώς θα μετακινήσουμε το p μία θέση δεξιά διότι έτσι μπορεί τα έχουμε ταύτιση. Τελικά το p μετακινείται τόσες θέσεις οσο είναι η μεγίστη τιμή των δύο παραπάνω τιμών. Ετσι έχουμε:

```
t:   W H I C H _ F I N A L L Y _ H A L T S   _ A T _ T H A T
p:                 A T _ T H A T
```

Και στη περίπτωση αυτή έχουμε αποτυχία με την πρώτη σύγκριση. Τότε θα μετακινήσουμε το p 4 θέσεις δεξιά διότι ενδέχεται να έχουμε ταύτιση στο (_). Επίσης αφού είχαμε αποτυχία στον τελευταίο χαρακτήρα, σίγουρα ο αντίστοιχος χαρακτήρας του t δεν είναι T, άρα μπορούμε να μετακινήσουμε το p μία μόνο θέση δεξιά. Τελικά το p μετακινείται 4 θέσεις και έχουμε:

```
t:   W H I C H _ F I N A L L Y _ H A L T S . _ A T _ T H A T
p:                         A T _ T H A T
```

Τώρα αποτυχία συμβαίνει στον δεύτερο από το τέλος χαρακτήρα. Επειδή το L δεν υπάρχει στο p θα μετακινήσουμε το p 6 θέσεις δεξιά. Επίσης από την αποτυχία ξέρουμε ότι ο αντίστοιχος χαρακτήρας του t δεν είναι A ενδέχεται να έχω ταύτιση για το ζεύγος T, άρα το p μπορεί να μετακινηθεί 3 θέσεις δεξιά. Τελικά το p μετακινείται 6 θέσεις. Παρατηρούμε ότι στον αλγόριθμο BM σε αντίθεση με τον KMP οι χαρακτήρες του t λαμβάνονται υπόψη. Ο αλγόριθμος BM συνοψίζεται ως εξής:

1. Ξεκίνα τις συγκρίσεις χαρακτήρων από το τέλος του p προς τ'αριστερά.
2. Οταν συμβεί αποτυχία σε μία θέση i του p τότε μετακίνησε το p, k θέσεις δεξιά, όπου k είναι το max από τις δύο παρακάτω τιμές.

   α. Η πρώτη τιμή εξαρτάται από την τιμή του χαρακτήρα του t που συνέβει η αποτυχία και την απόσταση από την αρχή του p που υπαρχει ο χαρακτήρας αυτός. Ετσι κατασκευάζουμε ένα πίνακα που κάθε του στοιχείο j περιέχει την απόσταση της μετακίνησης όταν συμβεί

αποτυχία στη θέση αυτή. Δηλαδή για κάθε γράμμα του αλφαβήτου εξετάζουμε αν το γράμμα υπάρχει στο p. Αν δεν υπάρχει τότε το p μετακινείται strlen(p) θέσεις δεξιά διαφορετικά το p μετακινείται m θέσεις, όπου m είναι η απόσταση του χαρακτήρα p[j] από την αρχή του p.

β. Η δεύτερη τιμή εξαρτάται από όλους τους χαρακτήρες του t που ταυτίζονται μερικώς με το p μαζί με τον χαρακτήρα που συνέβει η αποτυχία. Εδώ κατασκευάζουμε ένα δεύτερο πίνακα με τον ίδιο αλγόριθμο όπως στον KMP.

1. Smit G.De V., A comparison of three string matching algorithms, Software Practice and Experience, 12, 1982, 57-66
2. Horsepool R.N., Practical Fast Searching in Strings, Software Practice and Experience, 10, 1980, 501-506.
3. Boyer R.S., Moore J.S., A Fast String Searching Algorithm, Communications of the ACM, 1977, 762-772

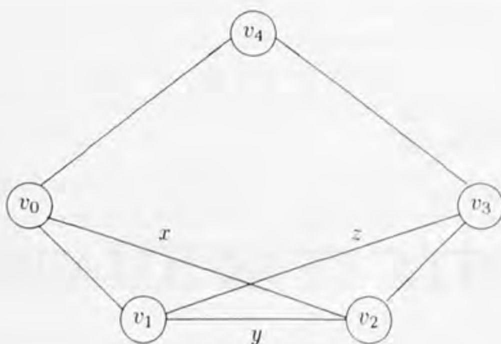
Κεφάλαιο 5

ΟΡΟΛΟΓΙΑ ΓΡΑΦΗΜΑΤΩΝ

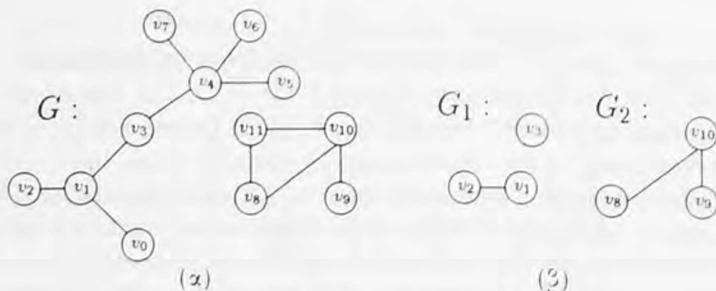
5.1. Γραφήματα

Ενα **γράφημα** (graph) G αποτελείται από ένα μη κενό πεπερασμένο σύνολο **κόμβων** (ή σημείων) (vertices, points) $V = V(G)$ και ένα σύνολο E μη διατεταγμένων ζευγών από στοιχεία του V. Κάθε ζεύγος $x = [u, v]$, το οποίο είναι στοιχείο του E, ονομάζεται **ακμή** (ή γραμμή) (edge, line) που συνδέει τους κόμβους u και v. Τα σημεία u και v ονομάζονται **γειτονικά** (adjacent) και η ακμή x ονομάζεται προσκείμενη (incident) στους κόμβους u και v. Δύο ακμές που πρόσκεινται στον ίδιο κόμβο ονομάζονται γειτονικές. Ενα γράφημα παριστάνεται συνήθως με ένα διάγραμμα. Για παράδειγμα, στο γράφημα G του σχήματος (5.1) οι κόμβοι v_0 και v_1 είναι γειτονικοί, ενώ οι v_0 και v_3 δεν είναι. Οι ακμές x και y είναι γειτονικές, ενώ οι x και z δεν είναι. Προσέξτε ότι, μολονότι οι ακμές x και z τέμνονται στο διάγραμμα, η τομή τους δεν είναι κόμβος του γραφήματος.

Βαθμός (degree) ενός κόμβου είναι ο αριθμός των ακμών που πρόσκεινται στον κόμβο αυτό. Για παράδειγμα στο γράφημα του σχήματος (5.1) ο κόμβος v_4 έχει βαθμό 2 και οι υπόλοιποι 3. Το άθροισμα των βαθμών όλων των κόμβων ενός γραφήματος υπολογίζεται πολύ εύκολα: Κάθε ακμή πρόσκειται σε δύο κόμβους, άρα για κάθε ακμή προστίθεται στο άθροισμα αυτό ο αριθμός 2. Οπότε το άθροισμα των βαθμών είναι διπλάσιο του αριθμού των ακμών.



Σχήμα (5.1)



Σχήμα (5.2)

Ενα **μονοπάτι** (path) $< v_0, v_1, \ldots, v_k >$ από τον κόμβο v στον v' είναι μια σειρά από κόμβους , τέτοια που (α) $v = v_0$ (ο πρώτος κόμβος είναι ο v_0) και $v' = v_k$ (ο τελευταίος κόμβος είναι ο v' ,(β)$[v_{i-1}, v_i] \in E$ για $i = 1, 2, \ldots, k$ (κάθε ζεύγος διαδοχικών κόμβων συνδέεται με ακμή) και (γ) όλοι οι κόμβοι είναι διαφορετικοί μεταξύ τους. Μια σειρά από κόμβους $< v_0, v_1, \ldots, v_k >$, στην οποία κάθε ζεύγος διαδοχικών κόμβων είναι γειτονικοί, είναι **κύκλος** (cycle) αν (α) έχει τον ίδιο αρχικό και τελικό κόμβο $v_0 = v_k$ και (β) οι υπόλοιποι κόμβοι $v_1, \ldots, v_k$ είναι διαφορετικοί μεταξύ τους. Στο σχήμα (5.1) το $< v_0, v_1, v_3, v_4 >$ είναι μονοπάτι, ενώ το $< v_0, v_3, v_4 >$ δεν είναι. Ο $< v_0, v_1, v_3, v_4, v_0 >$ είναι κύκλος. Ενα γράφημα δίχως κύκλους ονομάζεται

ακυκλικό. Ενα γράφημα ονομάζεται **συνεκτικό** (connected) αν κάθε ζεύγος κόμβων συνδέεται με μονοπάτι.

Υπογράφημα (subgraph) ενός γραφήματος G είναι ένα γράφημα, του οποίου όλοι οι κόμβοι και ακμές ανήκουν στό G. Στο σχήμα (5.2) τα γραφήματα G_1 και G_2 είναι υπογραφήματα του G.

Ενα μέγιστο συνεκτικό υπογράφημα γραφήματος G (δηλαδή, ένα συνεκτικό υπογράφημα, που δεν περιέχεται σε κανένα αλλο συνεκτικό υπογράφημα του G) καλείται **συνεκτική συνιστώσα** (connected component) ή απλώς συνιστώσα του G.

Ενα γράφημα είναι **δένδρο** (tree) αν έχει οποιεσδήποτε δύο από τις εξής τρεις ιδιότητες:

- Αριθμό ακμών κατά 1 μικρότερο από τον αριθμό των κόμβων.
- Είναι ακυκλικό.
- Είναι συνεκτικό.

ή ισοδύναμα, αν εχει μία από τις εξής δύο ιδιότητες:

- Αν προσθέσουμε μια ακμή, ένας και μόνο κύκλος προκύπτει.
- Για κάθε ζεύγος κόμβων υπάρχει ακριβώς ένα μονοπάτι μεταξύ τους.

Παράδειγμα: Τό γράφημα του σχήματος (5.2α) δεν είναι συνεκτικό. Εχει δύο συνεκτικές συνιστώσες, εκ των οποίων η συνιστώσα με κόμβους $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$ είναι δένδρο, ενώ η συνιστώσα με κόμβους v_8, v_9, v_{10}, v_{11} δεν είναι.

5.2. Παράσταση γραφημάτων

Εχουμε ήδη δει δύο τρόπους παράστασης γραφημάτων. Ο ένας είναι η απαρίθμηση όλων των κόμβων και ακμών του γραφήματος και ο άλλος η κατασκευή ενός σχήματος , στο οποίο οι κόμβοι παρίστανται με σημεία και οι ακμές με γραμμές μεταξύ των σημείων. Για να λύσουμε υπολογιστικά προβλήματα (να δώσουμε το γράφημα ως είσοδο σε αλγόριθμο) χρειαζόμαστε άλλες παραστάσεις. Δυο τυπικοί τρόποι παράστασης είναι η *μήτρα γειτνίασης* και η *λίστα γειτνίασης*

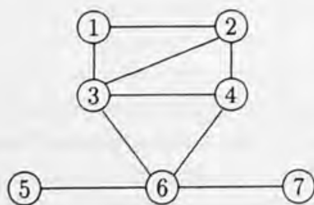
Η **μήτρα γειτνίασης** (adjacency matrix) ενός γραφήματος $G = (V, E)$ με n κόμβους $V = \{v_1, \ldots, v_n\}$ και m ακμές είναι ένας $n \times n$ πίνακας $A = a_{ij}$, που ορίζεται ως εξής για $1 \leq x, y \leq n$:

$$a_{ij} = \begin{cases} 1 & \text{αν } [v_i, v_j] \in E \\ 0 & \text{αλλιώς.} \end{cases}$$

Η μήτρα γειτνίασης ενός γραφήματος είναι, δηλαδή, ένας τετραγωνικός πίνακας που έχει ως σειρές και στήλες τους κόμβους και έχει την τιμή 1 σε όλες τις θέσεις που αντιστοιχούν σε ζεύγη κόμβων που είναι γειτονικά και 0 στις υπόλοιπες θέσεις. Η μήτρα αυτή είναι προφανώς συμμετρική, ($a_{ij} = a_{ji}$), άρα αρκεί να φυλάξουμε τη μισή.

Ο δεύτερος τρόπος παράστασης, η **λίστα γειτνίασης** (adjacency list), περιλαμβάνει, για κάθε κόμβο v του G μια λίστα με τους γειτονικούς κόμβους του v. Δηλαδή, στην παράσταση αυτή έχουμε ένα μονοδιάστατο πίνακα, ας τον ονομάσουμε adj, με n στοιχεία, κάθε στοιχείο του οποίου αντιστοιχείται σε ενα κόμβο του G και ειναι μια λίστα. Το adj$[v]$ περιλαμβάνει δείκτες προς όλους τους κόμβους v', για τους οποίους η ακμή $[v, v'] \in E$. Οι γειτονικοί κόμβοι του v φυλάγονται με αυθαίρετη σειρά.

Στο σχήμα (5.3α) παρουσιάζεται η παράσταση ενός γραφήματος με μήτρα γειτνίασης, ενώ στο σχήμα (5.3β) η παράσταση του ίδιου γραφήματος με λίστα γειτνίασης.



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Σχήμα (5.3α) Παράσταση με μήτρα γειτνίασης

λίστα γειτνίασης

Εγγράφημα κόμβου

κόμβος δείκτης

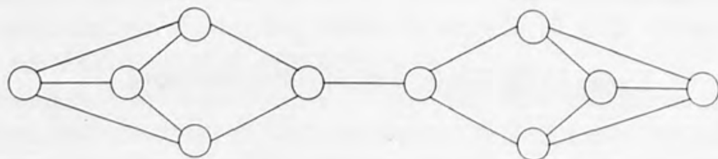
Σχήμα (5.3β) Παράσταση με λίστα γειτνίασης

Η παράσταση με μήτρα γειτνίασης απαιτεί $|V|^2$ βήματα μόνο για την αρχικοποίησή της και $|V|^2$ στοιχειώδεις θέσεις μνήμης. Αρα, είναι προτιμότερη όταν το γράφημα είναι **πυκνό** (dense), δηλαδή, όταν ο αριθμός των ακμών είναι κοντά στο τετράγωνο του αριθμού των κόμβων. (Ενα γράφημα έχει το μέγιστο αριθμό ακμών όταν είναι **πλήρες** (complete), δηλαδή αν κάθε δύο κόμβοι είναι γειτονικοί, οπότε ο αριθμός των ακμών είναι $\frac{1}{2}V(V-1)$). Επίσης η παράσταση με μήτρα γειτνίασης είναι ορισμένες φορες προτιμότερη για αλγορίθμους που απαιτούν περισσότερα από $|V|^2$ βήματα για να εκτελεστούν.

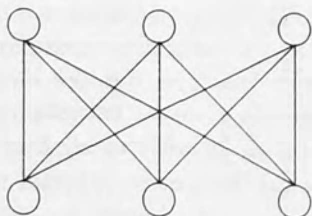
Η παράσταση με λίστα γειτνίασης είναι προτιμότερη όταν το γράφημα είναι **αραιό** (sparse), δηλαδή όταν έχει σχετικά λίγες ακμές. Ενδιαφέρουσα περίπτωση αραιών γραφημάτων είναι τα **επίπεδα** (planar) γραφήματα. Ενα γράφημα είναι επίπεδο αν είναι δυνατόν να σχεδιαστεί στο επίπεδο δίχως να τέμνονται δύο ακμές. Για παράδειγμα, το γράφημα του σχήματος (5.4α) είναι επίπεδο, επειδή είναι δυνατόν να σχεδιαστεί στο επίπεδο, όπως φαίνεται στο σχήμα (5.4β). Αντιθέτως, το γράφημα του σχήματος (5.5) δεν είναι επίπεδο. Ο μεγαλύτερος αριθμός ακμών που μπορεί να έχει ένα επίπεδο γράφημα με n

κόμβους (όπου $n \geq 3$) είναι $3n - 6$, όπως για παράδειγμα το γράφημα του σχήματος (5.6).

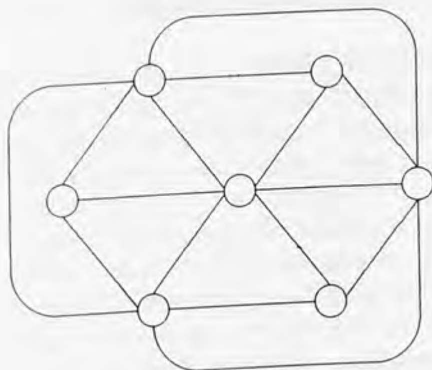
Σχήμα (5.4α)



Σχήμα (5.4β) Επίπεδο γράφημα



Σχήμα (5.5) Μη επίπεδο γράφημα



Σχήμα (5.6) Επίπεδο γράφημα με μέγιστο αριθμό ακμών

Αν n είναι ο αριθμός των κόμβων, η μήτρα γειτνίασης θα μπορούσε να οριστεί στην γλώσσα Pascal ως εξης:

A: **array**[1..n, 1..n] of **boolean**;

ενώ η λίστα γειτνίασης ως εξής:

```
type link = ↑ node;
     node = record
          v : integer;
          next : link end;

A : array [1..m] of link;
```

Ασκηση: Να γράψετε ένα πρόγραμμα που να παίρνει ως είσοδο το σύνολο των κόμβων V και ακμών E ενός γραφήματος και να τοποθετεί τις πληροφορίες στις δομές που ορίστηκαν παραπάνω.

Στούς περισσότερους από τους αλγορίθμους που ακολουθούν θεωρούμε ότι το γράφημα παριστάνεται με λίστα γειτνίασης. Με την ψευδοεντολή:

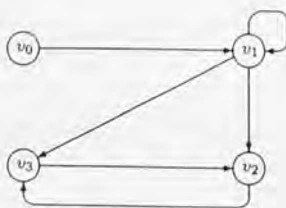
for each $y \in \text{adj}(x)$

εξετάζουμε όλους τους κόμβους y που είναι γειτονικοί με τον x.

Ασκηση: Γράψτε κώδικα Pascal που να υλοποιεί την ψευδοεντολή αυτή.

5.3. Γραφήματα με κατεύθυνση

Ενα **γράφημα με κατεύθυνση** (directed graph) είναι ένα γράφημα, οι ακμές του οποίου έχουν κατεύθυνση, είναι δηλαδή διατεταγμένα ζευγη. Μια ακμή με κατεύθυνση από το v στο w συμβολίζεται (v, w), ονομάζεται και **τόξο** (arc) και παριστάνεται στα διαγράμματα ως $v \longrightarrow w$ (ξεκινά από το (incident from) v και προσπίπτει στο (incident to) w). Ο **προς τα εξω βαθμός** (out-degree) κόμβου v είναι ο αριθμός των ακμών που ξεκινούν από τον v, ενώ ο **προς τα έσω βαθμός** (in-degree) κόμβου v είναι ο αριθμός των ακμών που προσπίπτουν στον v. Για παράδειγμα, στο σχήμα (7) ο κόμβος v_1 έχει προς τα έσω βαθμό 2, προς τα έξω βαθμό 3 και συνολικό βαθμό 5 (Προσοχή! η (x, x) υπολογίζεται δύο φορές). **Πηγή** (source) είναι ένας κόμβος με προς τα έσω βαθμό 0 (για παράδειγμα ο κόμβος v_0 στο σχήμα (5.7)). Ο κύκλος στά γραφήματα με κατεύθυνση ορίζεται όπως στα γραφήματα χωρίς κατεύθυνση.



Σχήμα (5.7)

Στη λίστα γειτνίασης που παριστάνει ένα γράφημα με κατεύθυνση κάθε ακμή (x, y) παρουσιάζεται μόνο μια φορά (προσθέτει μόνο τον x στη λίστα του y και όχι συμμετρικά τον x στη λίστα του y όπως γίνεται στα γραφήματα χωρίς κατεύθυνση). Επίσης, η μήτρα γειτνίασης δεν είναι συμμετρική.

5.4. Γραφήματα με βάρη

Ενα **γράφημα με βάρη** (weighted graph) είναι ένα γράφημα, σε κάθε ακμή του οποίου έχει αντιστοιχηθεί ένας ακέραιος. Ο ακέραιος που αντιστοιχεί-ται σε μια ακμή ονομάζεται **βάρος** (weight) της ακμής αυτής και ανάλογα με την εφαρμογή μπορεί να αναφέρεται στο κόστος, τη χωρητικότητα ή άλλα

χαρακτηριστικά της ακμής. Βάρη αντιστοιχούνται και στα γραφήματα με κατεύθυνση.

Οι δομές της λίστας γειτνίασης και της μήτρας γειτνίασης εύκολα επεκτείνονται ώστε να αποτελούν παραστάσεις γραφημάτων με βάρη. Στη μήτρα γειτνίασης κάθε στοιχείο του πίνακα δεν είναι boolean, αλλά integer, ενώ στη λίστα γειτνίασης η εγγραφή (record) ενός κόμβου περιέχει και το βάρος του.

5.4 Γραφήματα με βάρη

Κεφάλαιο 6

ΣΤΟΙΧΕΙΩΔΕΙΣ ΑΛΓΟΡΙΘΜΟΙ ΓΡΑΦΗΜΑΤΩΝ

6.1. Εξερεύνηση σε βάθος

Οι περισσότεροι αλγόριθμοι που λύνουν προβλήματα σε γραφήματα εξετάζουν ή επεξεργάζονται κάθε ένα κόμβο και ακμή. Η εξερεύνηση σε βάθος (Depth First Search, DFS) είναι μια μέθοδος διάσχισης του γραφήματος, με την οποία εξετάζονται όλοι οι κόμβοι και ακμές, αποκαλύπτεται δηλαδή η "δομή" του γραφήματος, με ελάχιστο κόστος. Για το λόγο αυτό, η εξερεύνηση σε βάθος έχει πολλές εφαρμογές και αποτελεί τη βάση πολλών αλγορίθμων, μερικούς από τους οποίους θα αναφέρουμε στο μάθημα αυτό.

Κατά την επίσκεψη σε ένα κόμβο x, η οποία στον αλγόριθμο παριστάνεται με την κλήση της ρουτίνας visit(x), μπορεί να γίνει η επεξεργασία των στοιχείων του κόμβου που απαιτεί η εφαρμογή. Αρχικά θεωρούμε ότι η visit(x) απλώς σημαδεύει τον κόμβο (με την εντολή visited[x] $\leftarrow$ true) και δίνει μοναδική επιγραφή label[x] σε κάθε κόμβο x, για τον οποίο καλείται. Η επιγραφή αυτή είναι η τιμή t ενός μετρητή που παίρνει αρχική τιμή στο κύριο πρόγραμμα και αυξάνεται από την visit κάθε φορά που καλείται.

```
visit(x);
begin
  visited[x] ← true;
  t ← t + 1; label[x] ← t;
end;
```

Αλγόριθμος 6.1.1: Εξερεύνηση σε βάθος.
Είσοδος: Γράφημα $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

```
0  DFS(x);
1  begin
2     visit(x);
3     for each y ∈ adj(x) do
4        if not visited[y] then DFS(y)
5  end

6  begin
7  for x ← 1 to n do
8     if not visited[x] then DFS(x)
9  end.
```

Βασική ιδιότητα: Ο DFS(x) επισκέπτεται τον y ανν ο y είναι συνδεδεμένος με το x.

Απόδειξη: α) Αν. Επαγωγή στο μήκος του μονοπατιού από τον x στον y:

Βάση: Εστω ότι ο x είναι γειτονικός με τον y. (Μονοπάτι μήκους 1). Τότε ο DFS(x) επισκέπτεται τον y κατά την εκτέλεση των γραμμών 3 και 4 του αλγορίθμου.

Υπόθεση: Εστω ότι ο DFS(x) επισκέπτεται όλους τους κομβους που είναι συνδεδεμένοι με τον x με μονοπάτι μήκους το πολύ k.

Βήμα: Εστω κόμβος y' είναι συνδεδεμένος με τον x με μονοπάτι μήκους $k+1$. Ο κόμβος y' θα είναι γειτονικός κάποιου κόμβου y'' που είναι συνδεδεμένος με τον x με μονοπάτι μήκους k. Σύμφωνα με την υπόθεση, ο DFS(x) επισκέπτεται τον y'' και κατά την εκτέλεση του DFS(y'') θα γίνει επίσκεψη στον y'.

β) Μόνο αν: Επαγωγή στο βάθος της αναδρομής (αριθμό αναδρομικών κλήσεων του DFS):

Βάση: Εστω ότι κατά την εκτέλεση του DFS(x) γίνεται αναδρομική κλήση του DFS(y). Αυτό συμβαίνει κατά την εκτέλεση της γραμμής 4, άρα ο y είναι γειτονικός του x.

Υπόθεση: Εστω ότι όλοι οι κόμβοι , για τους οποίους κλήθηκε ο DFS στις k πρώτες αναδρομικές κλήσεις είναι συνδεδεμένοι με τον x.

Βήμα: Εστω y'' ο κόμβος, για τον οποίο γίνεται η $k+1$ αναδρομική κλήση του DFS. Η κλήση αυτή γίνεται από κόμβο y' συνδεδεμένο με τον x, σύμφωνα με την επαγωγική υπόθεση. Η κλήση γίνεται από τη γραμμή 4 του αλγορίθμου, συνεπώς ο y'' είναι γειτονικός με τον y', άρα συνδεδεμένος με τον x.

Βασιζόμενοι στην ιδιότητα αυτή του DFS, μπορούμε να χρησιμοποιήσουμε τον DFS για να βρούμε τις συνεκτικές συνιστώσες ενός γραφήματος. Στην μεταβλητή c κρατάμε τον αριθμό των συνεκτικών συνιστωσών και προσθέτουμε ένα πεδίο στο εγγράφημα κάθε κόμβου για τον αριθμό συνεκτικής συνιστώσας, στην οποία ανήκει ο κόμβος. Για απλούστευση, ας θεωρησουμε ότι έχουμε ένα array comp[1..$|V|$] με τιμές την συνεκτική συνιστώσα, στην οποία ανήκει κάθε κόμβος. Η μόνη μεταβολή που χρειάζεται να γίνει στον αλγόριθμο είναι αύξηση κατά 1 της μεταβλητής c στη γραμμή 8. Η συνεκτική συνιστώσα comp[x] κάθε κόμβου x ενημερώνεται κατά την κλήση της visit(x).

Αλγόριθμος 6.1.2: Συνεκτικές συνιστώσες.
Είσοδος: Γράφημα $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

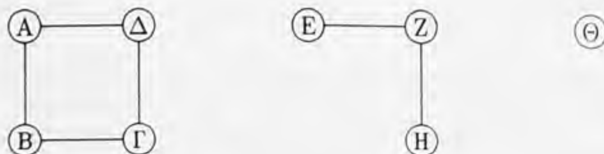
```
visit(x);
begin
   visited[x] ← true;
   t ← t + 1; label[x] ← t;
   comp[x] ← c
end;

0  DFS(x);
1  begin
2     visit(x);
3     for each y ∈ adj(x) do
4        if not visited[y] then DFS(y)
5  end

6  begin
7  for x ← 1 to n do
8     if not visited[x] then begin c ← c + 1; DFS(x) end
9  end
```

Παράδειγμα: Για να τρέξουμε τον Αλγόριθμο 6.1.2 στο γράφημα του Σχήματος (6.1), θα πρέπει πρώτα να αποφασίσουμε μια αντιστοιχία των αριθμών από 1 μέχρι 8 με τους κόμβους, την οποία θα χρησιμοποιήσουμε στη γραμμή 7 του αλγορίθμου. Εστω ότι η αντιστοιχία αυτή δίνεται από την αλφαβητική σειρά των γραμμάτων. Εστω ακόμη ότι οι γειτονικοί κόμβοι ενός κόμβου βρίσκονται στη μήτρα γειτνίασης με αλφαβητική σειρά, π.χ. αν ψάξουμε τη μήτρα γειτνίασης στην είσοδο A, βρίσκουμε πρώτα τον B και μετά τον Δ. Με τις υποθέσεις αυτές τα αποτελέσματα του Αλγορίθμου 6.1.2 για το γράφημα του Σχήματος (6.1) είναι τα εξής.

$$\begin{gathered}
\text{label}[A]=1,\ \text{label}[B]=2,\ \text{label}[\Gamma]=3,\ \text{label}[\Delta]=4,\\
\text{label}[E]=5,\ \text{label}[Z]=6,\ \text{label}[H]=7,\ \text{label}[\Theta]=8\\
\text{comp}[A]=\text{comp}[B]=\text{comp}[\Gamma]=\text{comp}[\Delta]=1,\\
\text{comp}[E]=\text{comp}[Z]=\text{comp}[H]=2, \text{comp}[\Theta]=3
\end{gathered}$$

Αν υποθέταμε ότι η σειρά των κόμβων ήταν η αντίστροφη αλφαβητική, τότε τα αποτελέσματα θα ήταν διαφορετικά:

$$\begin{gathered}
\text{label}[A]=8,\ \text{label}[B]=7,\ \text{label}[\Gamma]=6,\ \text{label}[\Delta]=5,\\
\text{label}[E]=4,\ \text{label}[Z]=3,\ \text{label}[H]=2,\ \text{label}[\Theta]=1\\
\text{comp}[A]=\text{comp}[B]=\text{comp}[\Gamma]=\text{comp}[\Delta]=3,\\
\text{comp}[E]=\text{comp}[Z]=\text{comp}[H]=2, \text{comp}[\Theta]=1
\end{gathered}$$



Σχήμα (6.1)

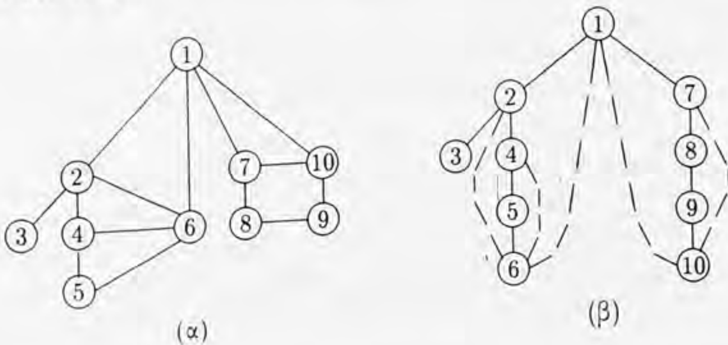
Μια άλλη απλή χρήση της εξερεύνησης σε βάθος είναι η δημιουργία ενος δένδρου που ονομάζεται **δένδρο εξερεύνησης σε βάθος** (depth first search tree). Το δένδρο αυτό αποτελείται από ακμές που οδηγούν σε νέους μη σημαδεμένους

(not visited) κόμβους κατά τη διάρκεια της εξερεύνησης. Αν το αρχικό γράφημα δεν είναι συνεκτικό, τότε μια πλήρης διάσχιση από τον DFS δημιουργεί ένα δάσος (σύνολο δενδρων) εξερεύνησης σε βάθος.

Το δένδρο εξερευνησης σε βάθος μπορεί να δημιουργηθεί ως εξής: Αρχικά δεν έχει καμία ακμή. Για κάθε νέα αναδρομική κλήση του DFS για ενα κόμβο y που δεν έχουμε επισκευθεί προστίθεται η ακμή $[x, y]$ που συνδέει τον y με τον κόμβο x από τον οποίο έγινε η κλήση. Συγκεκριμένα, η γραμμή 4 του αλγορίθμου τροποποιείται ως εξής:

4 **if** not visited[y] **then begin**
  πρόσθεσε την ακμή $[x, y]$ στο δένδρο;
  DFS(y) **end**

Με τον τρόπο αυτόν, όταν τελειώσει η εξερεύνηση ενος γραφήματος από τον DFS, οι ακμές του εχουν χωριστεί σε δύο ομάδες: αυτές που ανήκουν στο δένδρο, που ονομάζονται **ακμές δένδρου** (tree edges) και αυτές που δεν ανήκουν στο δένδρο.



Σχήμα (6.2)

Βασική ιδιότητα: Οι ακμές που δεν ανήκουν στο δένδρο συνδέουν δύο κόμβους, εκ των οποίων ο ένας είναι πρόγονος του άλλου στο δένδρο. (Ο x είναι πρόγονος του y αν βρίσκεται στο μοναδικό μονοπάτι από τη ρίζα στον y.) Για το λόγο αυτό, οι ακμές που δεν ανήκουν στο δένδρο εξερεύνησης σε βάθος ονομάζονται και ακμές προς τα πάνω (back edges). (Η ιδιότητα αυτή δεν ισχύει αν κάνουμε εξερεύνηση σε βάθος σε γραφήματα με κατεύθυνση, όπως θα δούμε σε επόμενα κεφάλαια.)

Απόδειξη: Εστω $[x, y]$ μία ακμή του G και έστω ότι ο DFS επισκέπτεται τον x πριν από τον y. Εφόσον ο y είναι γειτονικός του x, είτε ο DFS θα ξεκινήσει από τον y (οπότε η $[x, y]$ θα ανήκει στο δένδρο), είτε θα ξεκινήσει από κάποιο άλλο κόμβο και θα επισκεφθεί τον y γυρίσει πίσω στον x, οπότε ο y είναι απόγονος του x στο δένδρο.

Παράδειγμα: Στο Σχήμα (6.2β) εικονίζεται το δένδρο εξερεύνησης σε βάθος του γραφήματος του Σχήματος (6.2α). Οι προς τα πάνω ακμές παρουσιάζονται με διακεκομμένες γραμμές.

ΧΡΟΝΟΣ: Ο χρόνος εκτελέσεως του DFS είναι $\mathcal{O}(|E| + |V|)$ με μικρή σταθερά. Πρέπει να σημαδέψουμε όλους τους κόμβους (μια φορά αρχικά ως not visited και τους ελέγχουμε όλους στο κυρίως πρόγραμμα), οπότε έχουμε τον όρο $|V|$ και εξετάζουμε κάθε ακμή δύο φορές (επειδή για κάθε κόμβο εξετάζουμε όλους τους γειτονικούς), οπότε προστίθεται ο όρος $|E|$. Αρα, ο χρόνος είναι της τάξης του μέγιστου μεταξύ του αριθμού των ακμών και του αριθμού των κόμβων, που είναι συνήθως ο αριθμός των ακμών.

6.2. Δισυνεκτικές συνιστώσες, κόμβοι άρθρωσης

Εχουμε ήδη δει την εφαρμογή του DFS για τον επιμερισμό ενος γραφήματος σε συνεκτικές συνιστώσες. Ο DFS μπορεί επίσης να χρησιμοποιηθεί για να χωρίσουμε το γράφημα σε δισυνεκτικές συνιστώσες. Η έννοια της δισυνεκτικότητας είναι πολύ φυσική επέκταση της έννοιας της συνεκτικότητας. Ενα γράφημα ονομάζεται συνεκτικό αν υπάρχει τουλάχιστον ένα μονοπάτι μεταξύ κάθε ζεύγους κόμβων. Ενα γράφημα ονομάζεται **δισυνεκτικό** (biconnected) αν υπάρχουν τουλάχιστον δύο ξένα μονοπάτια (μονοπάτια που δεν έχουν άλλους κοινούς κόμβους εκτός από τον αρχικό και τελικό) μεταξύ κάθε ζεύγους κόμβων. Ετσι, αν για κάποιο λόγο ένα από τα μονοπάτια που συνδέουν τους δύο κόμβους δεν είναι δυνατόν να χρησιμοποιηθεί, οι κόμβοι εξακολουθούν να είναι συνδεδεμένοι.

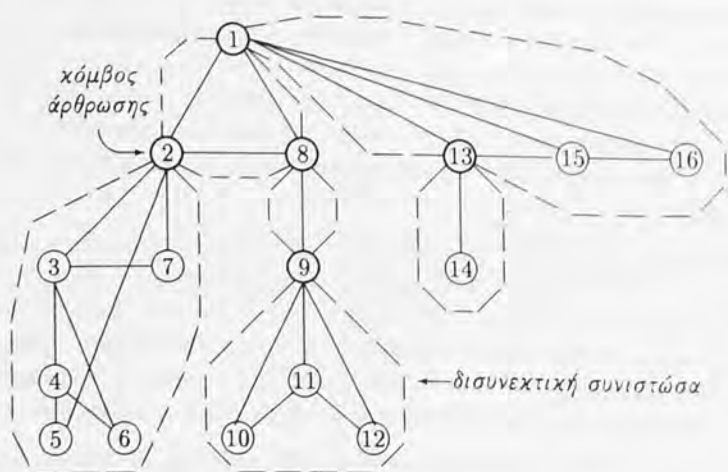
Βασική ιδιότητα: Ενα συνεκτικό γράφημα είναι δισυνεκτικό ανν δεν υπάρχουν κόμβοι που με την αφαίρεσή τους (μαζί με τις προσκείμενες ακμές) να παράγεται γράφημα μη συνεκτικό. Οι κόμβοι που έχουν την ιδιότητα αυτή (η αφαίρεσή τους να αποσυνδέει το γράφημα) ονομάζονται **κόμβοι άρθρωσης** (articulation vertices). Δηλαδή, η βασική ιδιότητα μπορεί να διατυπωθεί ως εξής: ένα γράφημα είναι δισυνεκτικό ανν δεν έχει κόμβους άρθρωσης.

Απόδειξη: α) Αν: Εστω ότι σε ένα δισυνεκτικό γράφημα G υπάρχει κόμβος άρθρωσης v. Η αφαίρεση του v χωρίζει το γράφημα σε τουλάχιστον δύο συνεκτικές συνιστώσες. Εστω δύο κόμβοι x και y που βρίσκονται σε διαφορετικές συνιστώσες. Εφόσον δεν υπάρχει μονοπάτι από τον x στον y στο $G-\{v\}$, όλα τα μονοπάτια μεταξύ των δύο κόμβων περνούσαν από τον v, αρα δεν υπήρχαν τουλάχιστον δύο ξένα μονοπάτια μεταξύ των x και y, δηλαδή το G δεν είναι δισυνεκτικό.

β) Μονο αν: Εστω συνεκτικό γράφημα που δεν έχει κόμβους άρθρωσης, και παρ' όλα αυτά στο γράφημα αυτό υπάρχει ζεύγος κόμβων (x, y), μεταξύ των οποίων δεν υπαρχουν δύο ξένα μονοπάτια. Ας θεωρήσουμε το μονοπάτι από το x στο y, που υπάρχει λόγω συνεκτικότητας. Βασισμένοι στο γεγονός ότι κάθε κόμβος του μονοπατιού αυτού δεν είναι κόμβος άρθρωσης, μπορούμε να κατασκευάσουμε και δεύτερο μονοπάτι απο το x στο y (παραλείπουμε τις κάπως πολύπλοκες λεπτομέρειες).

Δισυνεκτική συνιστώσα (biconnected component, bicomponent) είναι ένα μέγιστο δισυνεκτικό υπογράφημα ενος γραφήματος.

Παράδειγμα: Στο Σχήμα (6.3) εικονίζονται οι κόμβοι άρθρωσης και οι δισυνεκτικές συνιστώσες ενος γραφήματος. Κόμβοι άρθρωσης είναι οι 1, 2, 8, 9 και 13. Δυσινεκτικές συνιστώσες είναι το σύνολο των κόμβων που βρίσκονται μέσα σε διακεκομμένο κύκλο. Οι κόμβοι άρθρωσης, οι οποίοι είναι όρια του κύκλου συμπεριλαμβάνονται στη συνιστώσα.



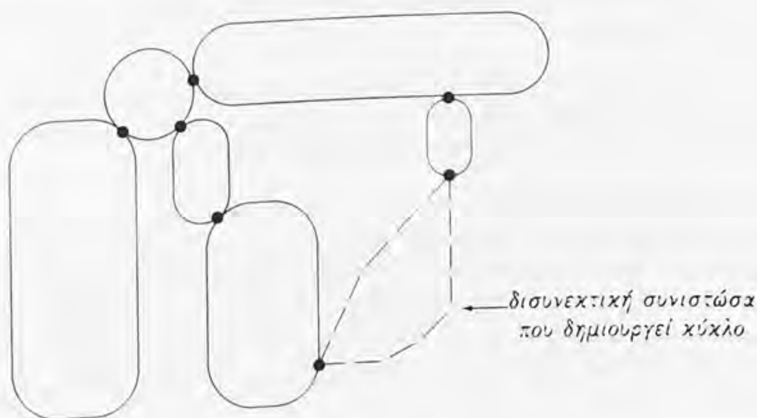
Σχήμα (6.3)

Προσέξτε ότι, μολονότι οι δισυνεκτικές συνιστώσες χωρίζουν το σύνολο των ακμών σε ξένα μεταξύ τους υποσύνολα, δεν συμβαίνει το ίδιο με το σύνολο των κόμβων. *Οι κόμβοι άρθρωσης είναι κοινοί σε περισσότερες από μία συνεκτικές συνιστώσες.*

Βασική ιδιότητα: **Κάθε γράφημα είναι ένα δένδρο των δισυνεκτικών συνιστωσών του.**

Απόδειξη: Εστω ότι το γράφημα G' των δισυνεκτικών συνιστωσών (δηλαδή το γράφημα G', το οποίο έχει ως κόμβους τους κόμβους άρθρωσης του αρχικού γραφήματος G, οι οποίοι συνδέονται μεταξύ τους με δισυνεκτικές συνιστώσες) περιέχει ένα κύκλο (άρα δεν είναι δένδρο). (Βλ. Σχήμα (6.4)). Τότε υπάρχουν δύο διαφορετικά μονοπάτια ανάμεσα σε κάθε ζεύγος κόμβων, οι οποίοι βρίσκονται σε οποιαδήποτε δισυνεκτική συνιστώσα του κύκλου. Αρα όλες οι

συνιστώσες στον κύκλο περιλαμβάνονται σε μια μεγαλύτερη δισυνεκτική συνιστώσα, όπερ άτοπον, γιατί ορίσαμε τη δισυνεκτική συνιστώσα ως το *μέγιστο* δισυνεκτικό υπογράφημα.



Σχήμα (6.4)

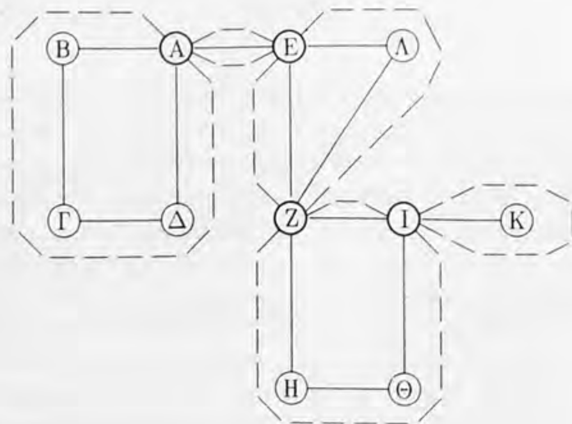
Εστω ότι χρησιμοποιούμε τον Αλγόριθμο DFS για την εξερεύνηση ενος γραφήματος. Υπάρχουν δύο σημεία, στα οποία επεξεργαζόμαστε ένα κόμβο x: Οταν πρωτοσυναντάμε τον x (μέσα στην visit(x)) και όταν επιστρέφουμε στον x, αφού έχουμε επισκεφτεί ένα παιδί του y (και προφανώς, λόγω της αναδρομής, όλους τους απογόνους του y). Η βασική ιδέα του αλγορίθμου είναι η εξής: Αν ούτε ο y, ούτε κανένας από τους απογόνους του δεν είναι γειτονικός με κάποιο κόμβο που επισκεφτήκαμε πριν από τον x, τότε ο μόνος τρόπος να συνδεθεί οποιοσδήποτε από τους κόμβους του ήδη κατασκευασμένου δένδρου με τον y και τους απογόνους του είναι μέσω του x. Αν αφαιρεθεί ο x, δεν υπάρχει μονοπάτι από τον y και τους απογόνους του προς το ήδη κατασκευασμένο δένδρο, δηλαδή ο x είναι κόμβος άρθρωσης. Αρα, ενώ κάνουμε εξερεύνηση σε βάθος, θα πρέπει να κρατάμε στο εγγράφημα κάθε κόμβου την εξής πληροφορία: την μικρότερη επιγραφή κόμβου, με τον οποίο ο y ή κάποιος απόγονός του είναι γειτονικός. Οι κόμβοι που βρίσκονται στο τμήμα του δένδρου εξερεύνησης σε βάθος που έχει ρίζα το y συνδέονται με ακμές

που δεν ανήκουν στο δένδρο (προς τα πάνω ακμές) με προγόνους τους στο δένδρο. Εστω back$[y]$ ο μικρότερος $label[w]$, ώστε να υπάρχει κόμβος γειτονικός στον w στο δένδρο με ρίζα y. Ο αλγόριθμος που βρίσκει τις δισυνεκτικές συνιστώσες ενος γραφήματος βασίζεται στην εξής ιδιότητα:

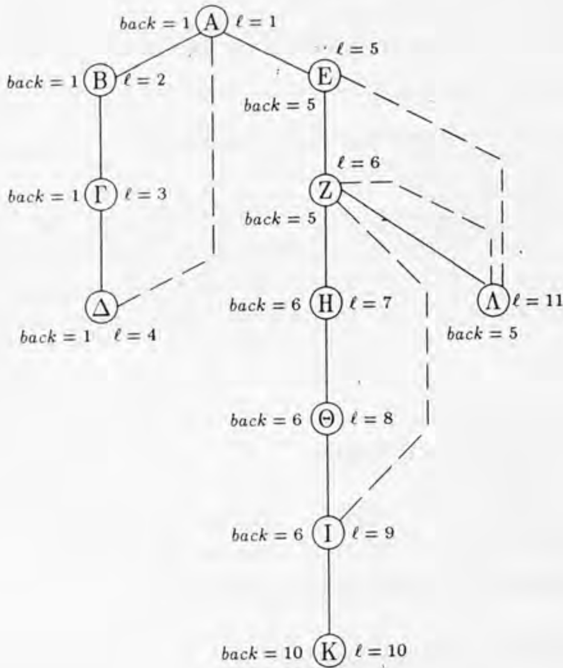
Ο x είναι κόμβος άρθρωσης αν για κάποιο παιδί y του x στο δενδρο back$[y]$ $\geq$label$[x]$

Απόδειξη: Αν δεν υπάρχει προς τα πάνω ακμή από κάποιο απόγονο του y προς κόμβο με επιγραφή μικρότερη του x, τότε όλα τα μονοπάτια από τη ρίζα στον y περνάνε από τον x, άρα ο x είναι κόμβος άρθρωσης.

Παράδειγμα: Στο Σχήμα (6.5β), που είναι το δένδρο εξερεύνησης σε βάθος του γραφήματος του Σχήματος (6.5α), παρουσιάζονται οι τιμές του πίνακα back για όλους τους κόμβους. (Υποθέτουμε ότι ακολουθούμε αλφαβητική σειρά κατά την εξερεύνηση σε βάθος.) Παρατηρείστε για κάθε ένα από τους κόμβους άρθρωσης A, E, Z και I, τις τιμές του πεδίου back των απογόνων του στο δένδρο. Στο Σχήμα (6.5β)οι τιμές του label συμβολίζονται με ℓ.



Σχήμα (6.5α)



Σχήμα (6.5β)

Στον αλγόριθμο που ακολουθεί χρησιμοποιούμε μια στοίβα την οποία χειριζόμαστε με τις συναρτήσεις push() και pop(). (Κώδικα για τις συναρτήσεις αυτές μπορείτε να βρείτε στην Ενότητα 3.2.1 της αναφοράς [5].) Θεωρούμε ότι το γράφημα είναι συνεκτικό.

Παράδειγμα: Τα αποτελέσματα του Αλγορίθμου 6.2.1 αν κληθεί ως BCDFS(A) για το γράφημα του Σχήματος (6.5) είναι τα εξής.

Κόμβος άρθρωσης A. Συνιστώσα Δ, Γ, B, A.
Κόμβος άρθρωσης I. Συνιστώσα K, I.
Κόμβος άρθρωσης Z. Συνιστώσα I, Θ, H, Z.
Κόμβος άρθρωσης E. Συνιστώσα Λ, Z, E.
Κόμβος άρθρωσης A. Συνιστώσα E, A.

Αλγόριθμος 6.2.1: Δισυνεκτικές συνιστώσες.
Είσοδος: Γράφημα $V = (G, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

```
visit(x);
begin
  visited[x] ← true;
  t ← t + 1; label[x] ← t;
  back[x] ← t; push(x)
end;

BCDFS(x);
begin
  visit(x);
  for each y ∈ adj(x) do
    if not visited[y] then begin
      BCDFS(y);
      if back[y] ≥ label[x] then /* κόμβος άρθρωσης x */
        pop νέα δισυνεκτική συνιστώσα ως το x
        else back[x] ←min(back[x], back[y])
    end
    else back[x] ← min(back[x], label[y])
end
```

ΧΡΟΝΟΣ

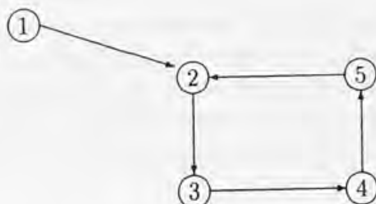
Ο Αλγόριθμος DFS, που αποτελεί τον σκελετό του BCDFS, εκτελείται σε χρόνο $\mathcal{O}(|E|)$. Αρα, χρειάζεται να ελέγξουμε αν κάνουμε σταθερό αριθμό βημάτων για κάθε ακμή. Το μόνο σημείο που δεν είναι προφανές είναι το σημείο που βρίσκουμε νέα δισυνεκτική συνιστώσα. Ομως, κάθε κόμβος θα μπει και θα βγει από τη στοίβα το πολύ δύο φορες, άρα και ο BCDFS εκτελείται σε χρόνο $\mathcal{O}(|E|)$.

6.3. Ακυκλικά γραφήματα με κατεύθυνση, τοπολογική ταξινόμηση

Τα ακυκλικά γραφήματα με κατεύθυνση έχουν την εξής βασική ιδιότητα:

Βασική ιδιότητα: **"Κάθε ακυκλικό γράφημα με κατεύθυνση έχει μία πηγή"**

Απόδειξη: Αρχισε από ένα κόμβο x του γραφήματος. Αν δεν είναι πηγή, τότε τουλάχιστον μία ακμή (x, y) πρόσκειται σε αυτόν. Επανάλαβε το ίδιο για τον κόμβο y. Εφόσον το γράφημα δεν έχει κύκλους, δεν θα συναντήσουμε τον ίδιο κόμβο δύο φορές, άρα ο αλγόριθμος θα σταματήσει μετά από $|V|$ βήματα το πολύ.



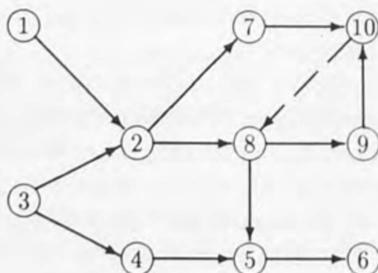
Σχήμα (6.6)

Υπάρχουν όμως γραφήματα με κατεύθυνση που έχουν πηγή και δεν είναι ακυκλικά, όπως για παράδειγμα το γράφημα του Σχήματος 6.6. Αρα, η Βασική ιδιότητα που αποδείξαμε δεν αρκεί για να αποδείξει ότι το γράφημα είναι ακυκλικό. Χρειάζεται να προσθέσουμε κάτι ακόμα στην ιδιότητα αυτή, ώστε να μπορούμε να τη χρησιμοποιούμε για να αποδείξουμε ότι ένα γράφημα είναι ακυκλικό (να είναι δηλαδή ικανή συνθήκη).

Ικανή και αναγκαία συνθήκη για να είναι ένα γράφημα με κατεύθυνση ακυκλικό:

"Το γράφημα έχει τουλάχιστον μια πηγή. Και αν σβήσουμε την πηγή αυτή (και τις προσκείμενες ακμές) το προκύπτον γράφημα έχει πηγή κ.ο.κ., έως ότου σβήσουμε όλους τους κόμβους."

Αρα καταλήγουμε στον εξής αλγόριθμο που ελέγχει αν ένα γράφημα με κατεύθυνση είναι ακυκλικό.



Σχήμα (6.7β)

Αν στο γράφημα προστεθεί η διακεκομμένη γραμμή του Σχήματος (6.7β), τότε ο αλγόριθμος θα σβήσει τους κόμβους 1, 3, 2, 7, 4, 5, 6 και θα σταματήσει δίχως να σβήσει τον κύκλο 8, 9, 10.

ΧΡΟΝΟΣ: Για τις τρείς πρώτες γραμμές, που γίνεται η αρχικοποίηση του αλγορίθμου, απαιτείται χρόνος $\mathcal{O}(|V| + |E|)$. Εν συνεχεία κάθε ακμή (x, y) εξετάζεται ακριβώς μία φορά (όταν ο κόμβος x βγαίνει από τη στοίβα). Αρα, ο χρόνος που απαιτείται από τον αλγόριθμο είναι $\mathcal{O}(|E|)$.

6.4. Συνεκτικές συνιστώσες γραφημάτων με κατεύθυνση

Ενα **γράφημα με κατεύθυνση** είναι **ισχυρά συνεκτικό** (strongly connected) αν για κάθε ζεύγος κόμβων x και y υπάρχει μονοπάτι από τον x στον y. (Το ζεύγη στον ορισμό αυτόν θεωρούνται διατεταγμένα, δηλαδή το y, x είναι διαφορετικό απο το x, y.) Στο εξής, όταν αναφερόμαστε σε γραφήματα με κατεύθυνση, με τον όρο **συνεκτικό** γράφημα θα εννοούμε ισχυρά συνεκτικό. Ο ορισμός είναι ίδιος με τον ορισμό της συνεκτικότητας στα γραφήματα χωρίς κατεύθυνση, αλλά στην περίπτωση αυτή η κατεύθυνση των ακμών παίζει ρόλο (δηλαδή, υπάρχει μονοπάτι από τον x στον y αν μπορούμε να πάμε από τον x στον y ακολουθώντας τις ακμές στην κατεύθυνση των βελών τους).

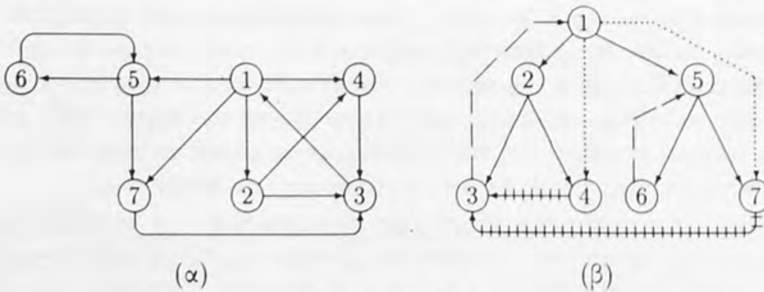
Ο ορισμός της συνεκτικής συνιστώσας στα γραφήματα με κατεύθυνση είναι ίδιος με τον ορισμό στα γραφήματα. **Συνεκτική συνιστώσα** είναι ένα μέγιστο συνεκτικό υπογράφημα, ή αλλιώς ένα συνεκτικό υπογράφημα που δεν περιέχεται σε κανένα άλλο συνεκτικό υπογράφημα.

Η εξερεύνηση σε βάθος στα γραφήματα με κατεύθυνση είναι ίδιος αλγόριθμος με τον Αλγόριθμο 6.1.1, μόνο που οι ακμές, οι οποίες δεν ανήκουν στο δένδρο εξερεύνησης σε βάθος (DFS δένδρο) χωρίζονται τώρα σε τρείς κατηγορίες: ακμές προς τα πάνω, προς τα κάτω και προς τα πίσω. (Θυμηθείτε ότι, όταν το γράφημα δεν έχει κατεύθυνση, όλες οι ακμές που δεν ανήκουν στο δενδρο είναι ακμές προς τα πάνω.) Οι **ακμές προς τα πάνω** (back edges) συνδέουν έναν κόμβο με έναν πρόγονό του στο DFS δένδρο. Οι **ακμές προς τα κάτω** (descendent edges) συνδέουν έναν κόμβο με έναν απόγονό του στο δένδρο και οι **ακμές προς τα πίσω** (cross edges) συνδέουν δύο κόμβους, εκ των οποίων ο ένας δεν είναι πρόγονος του άλλου.

Παράδειγμα 6.4.1: Στο Σχήμα (6.8β) παρουσιάζεται το δένδρο εξερεύνησης σε βάθος του γραφήματος του Σχήματος (6.8α). Οι ακμές $(3, 1)$ και $(6, 5)$ είναι προς τα πάνω, οι $(1, 4)$ και $(1, 7)$ προς τα κάτω, οι $(4, 3)$ και $(7, 3)$ προς τα πίσω και οι υπόλοιπες ακμές δένδρου.

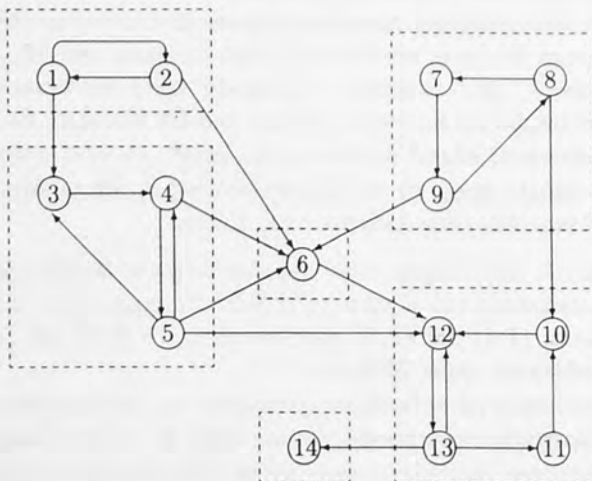
Απο ένα γράφημα με κατεύθυνση μπορούμε να κατασκευάσουμε το γράφημα των συνεκτικών συνιστωσών του ως εξής: (α) όλοι οι κόμβοι μιας συνεκτικής συνιστώσας αποτελούν έναν κόμβο στο γράφημα των συνιστωσών και (β) υπάρχει ακμή ανάμεσα σε δύο κομβους στο γράφημα των συνιστωσών, αν στο αρχικό γράφημα δύο κόμβοι που ανήκουν στις συνιστώσες αυτές είναι γειτονικοί.

Παράδειγμα 6.4.2: Το γράφημα των συνεκτικών συνιστωσών του Σχήματος 6.9α εικονίζεται στο 6.9β.

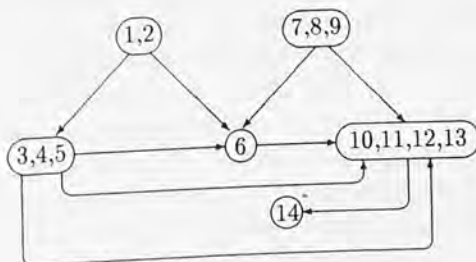


(α) (β)

Σχήμα (6.8)



Σχήμα (6.9α)



Σχήμα (6.9β)

Βασική ιδιότητα: **"Ενα γράφημα με κατεύθυνση είναι ένα ακυκλικό γράφημα με κατεύθυνση των συνεκτικών συνιστωσών του."**

Απόδειξη: Αν υπήρχε κύκλος στο γράφημα των συνεκτικών συνιστωσών, τότε όλες οι συνεκτικές συνιστώσες στον κύκλο θα αποτελούσαν μια μεγαλύτερη συνεκτική συνιστώσα. Ατοπο, επειδή η συνεκτική συνιστώσα ορίστηκε ως το *μέγιστο* συνεκτικό υπογράφημα.

Θα αναφέρουμε δύο αλγορίθμους, με τους οποίους βρίσκονται οι συνεκτικές συνιστώσες ενός γραφήματος με κατεύθυνση. Οταν τους καταλάβετε αρκετά, θα δείτε ότι δεν είναι τόσο διαφορετικοί μεταξύ τους, όσο φαίνεται από πρώτη ματιά.

Ο πρώτος που θα αναφέρουμε θυμίζει πολύ τον αλγόριθμο που βρίσκει τις δισυνεκτικές συνιστώσες σε γράφημα χωρίς κατεύθυνση. Η πληροφορία που εξετάζουμε τώρα επιστρέφοντας στον κόμβο x για να ελέγξουμε αν τελειώσαμε την διάσχιση μιας συνεκτικής συνιστώσας ονομάζεται low$[x]$.

low$[x]$: είναι η μικρότερη επιγραφή label$[y]$ κόμβου y τέτοιου που να υπάρχει μονοπάτι από τον y στον x και ο y να είναι γειτονικός με κάποιον απόγονο του x στο δένδρο. Ο y δεν είναι αναγκαστικά πρόγονος του x.

Αλγόριθμος 6.4.1: Συνεκτικές συνιστώσες γραφημάτων με κατεύθυνση.
Είσοδος: Γράφημα $V = (G, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην ενότητα 1.2.

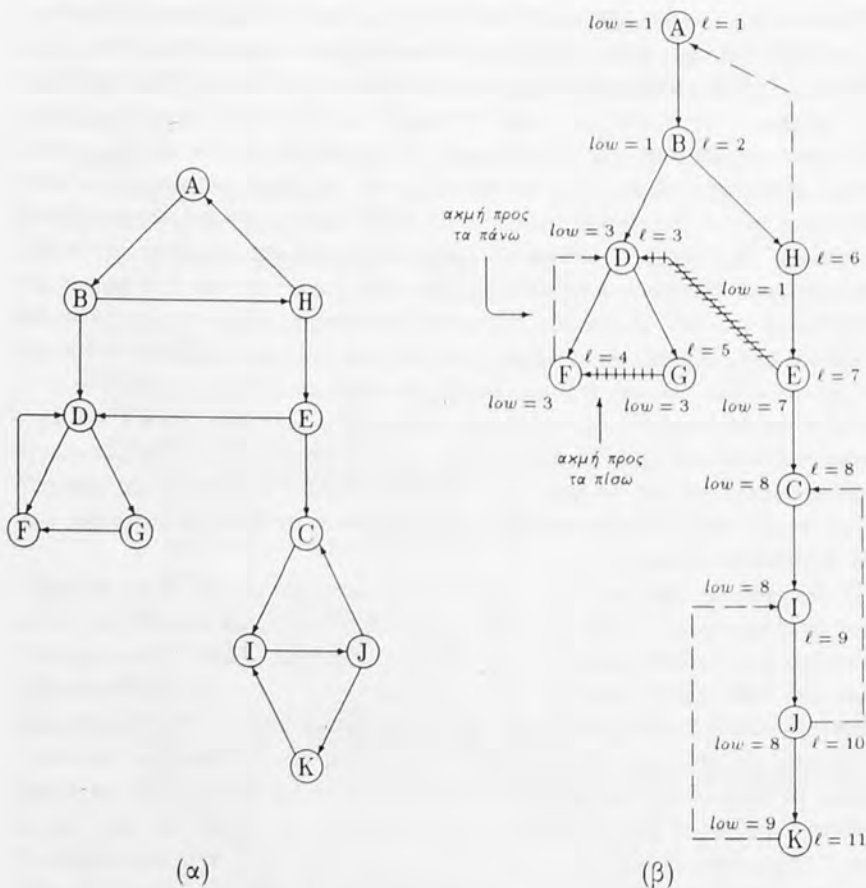
```
visit(x);
begin
  visited[x] ← true;
  t ← t + 1; label[x] ← t;
  low[x] ← t; removed[x] ← false
end;

SCDFS(x);
begin
  visit(x);
  for each y ∈ adj(x) do
    if not visited[y] then begin
      SCDFS(y);
      low[x] ← min(low[x], low[y])
      end
    else if not removed[y]
      low[x] ← min(low[x], low[y]);
  push(x);
  if  low[x]=label[x] then begin
    output("νέα συνεκτική συνιστώσα");
    while υπάρχουν στοιχεία στη στοίβα and l[top]≥label[x] do begin
      pop(y);
      output(y);
      removed[y] ← true
    end
  end
end
```

Θεωρούμε ότι το υποπρόγραμμα SCDFS(x) καλείται από ένα κυρίως πρόγραμμα ίδιο με εκείνο που παρουσιάζεται στον Αλγόριθμο 6.1.

```
begin
for x ← 1 to n do
  if not visited[x] then SCDFS(x)
end.
```

Η βασική ιδέα του αλγορίθμου είναι η εξής: Κατά τη διάρκεια της εξερεύνησης σε βάθος φυλάμε κάθε κόμβο που επισκεπτόμαστε σε μια στοίβα και όταν αποφασίσουμε ότι ολοκληρώσαμε μια συνεκτική συνιστώσα, εξάγουμε όλους τους κόμβους της συνιστώσας από τη στοίβα και δεν τους ξαναεξετάζουμε. Οταν κατά τη διάρκεια της εξερεύνησης επιστρέψουμε σε ένα κόμβο x αφού έχουμε επισκεφτεί *όλους* τους γειτονικούς του κόμβους, μπορούμε να αποφασίσουμε αν ο x και οι απόγονοί του στο DFS δένδρο αποτελούν συνεκτική συνιστώσα. Αν υπάρχει τουλάχιστον ένας κύκλος που περιλαμβάνει τον x, κάποιον απόγονό του και ένα τρίτο κόμβο με label μικρότερη του x, τότε ο x και οι απόγονοί του δεν αποτελούν συνεκτική συνιστώσα αλλά περιλαμβάνονται σε κάποια μεγαλύτερη. Αν υπάρχει όμως κύκλος που περιλαμβάνει τον x και κάποιον απόγονό του και δεν περιλαμβάνει κόμβο με label μικρότερη του x, τότε ο x και οι απόγονοί του είναι νεα συνεκτική συνιστώσα. Αν ο x δεν έχει κανένα απόγονο στο DFS δένδρο (ή όλοι του οι απόγονο έχουν ήδη καταταγεί σε συνιστώσες) και δεν υπάρχει προς τα πίσω ή προς τα πάνω ακμή από τον x προς κόμβο του δένδρου που δεν περιλαμβάνεται σε άλλη συνιστώσα, ο x είναι συνιστώσα μόνος του.

Ο έλεγχος αν βρέθηκε νέα συνεκτική συνιστώσα γίνεται όταν επιστρέψουμε σε ένα κόμβο x αφού έχουμε επισκεφτεί *όλα* του τα παιδιά (ενώ στον αλγόριθμο για τις δισυνεκτικές συνιστώσες ο έλεγχος γίνεται όταν επιστρέφουμε από *κάθε* παιδί). Αν όταν επιστρέψουμε στον x από όλα του τα παιδιά το low[x]=label[x], τότε αποφασίζουμε ότι ολοκληρώθηκε η εξερεύνηση μιας συνεκτικής συνιστώσας. Εχουμε μόλις τελειώσει την εξερεύνηση του υποδένδρου με ρίζα x και δεν βρέθηκε τρόπος να πάμε σε κόμβο y με επιγραφή μικρότερη από αυτή του x, τέτοιο που να υπάρχει και μονοπάτι από τον y στον x. Αρα, όλοι οι απόγονοι του x (οι οποίοι δεν έχουν ήδη καταταγεί σε άλλη συνεκτική συνιστώσα, έχουν δηλαδή removed[z]=false) έχουν τις εξής ιδιότητες: (α) βρίσκονται ανά δύο σε ένα κύκλο που περνάει από το x και (β) δεν περιλαμβάνονται σε κανένα κύκλο με κόμβους διαφορετικούς από τον x και τους απογόνους του. Προφανώς, οι προς τα πίσω ακμές μας οδηγούν μόνο σε κόμβους που έχουμε ήδη επισκεφτεί και όχι σε νέους ανεξερεύνητους κόμβους, άρα όταν επιστρέφουμε στον x, έχουμε εξερευνήσει όλα τα μονοπάτια (αποδείξτε το).



Σχήμα (6.10)

Παράδειγμα 6.4.3: Τα αποτελέσματα του Αλγορίθμου 6.4.1 αν ακολουθήσει τη γνωστή αλφαβητική σειρά για το γράφημα του Σχήματος (6.10α) είναι τα εξής:

νέα συνεκτική συνιστώσα: G, F, D, νέα συνεκτική συνιστώσα: K, J, I, C
νέα συνεκτική συνιστώσα: E, νέα συνεκτική συνιστώσα: H, B, A

Το δένδρο εξερεύνησης σε βάθος παρουσιάζεται στο Σχήμα 6.10β. Οι ακμές που δεν ανήκουν στο δένδρο χωρίζονται σε ακμές προς τα πάνω (οι οποίες παρουσιάζονται με διακεκομμένες γραμμές) και ακμές προς τα πίσω (οι οποίες παρουσιάζονται με μικρές κάθετες γραμμές).

Ασκηση: Να τρέξετε τον αλγόριθμο για το γράφημα του σχήματος 6.10 αφού του προσθέσετε τις ακμές (C, G) και (G, B).

Πριν παρουσιάσουμε τον δεύτερο αλγόριθμο, ας προσθέσουμε μία ακόμη πληροφορία στο εγγράφημα κάθε κόμβου ενώ κάνουμε εξερεύνηση σε βάθος. Υποθέτουμε ότι ο μετρητής t δεν αυξάνεται μόνο κάθε φορά που επισκεπτόμαστε ένα νέο κόμβο x, αλλά επίσης αυξάνεται κάθε φόρα που τελειώνει ο DFS για ένα κόμβο x επειδή εχουμε αναδρομικά δεί όλους τους γειτονικούς του κόμβους. Στο εγγράφημα κάθε κόμβου x φυλάγεται και η τιμή του t την στιγμή που ο DFS(x) τελειώνει και πρόκειται να βγεί από τη στοίβα. Ας θεωρήσουμε, ως συνήθως, ότι έχουμε ένα array (ας το ονομάσουμε finished), στο οποίο φυλάγεται η πληροφορία αυτή για όλους τους κόμβους. Ο Αλγόριθμος 6.1.1 (εξερεύνηση σε βάθος) μετατρέπεται ως εξής για να χειρίζεται και τη νέα αυτή πληροφορία:

Αλγόριθμος 6.4.2: Εξερεύνηση σε βάθος.
Είσοδος: Γράφημα $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην ενότητα 5.2.

```
DFS(x);
begin
  visit(x);
  for each y ∈ adj(x) do
    if not visited[y] then DFS(y)
  t ← t + 1
  finished[x] = t
end

begin
for x ← 1 to n do
  if not visited[x] then DFS(x)
end.
```

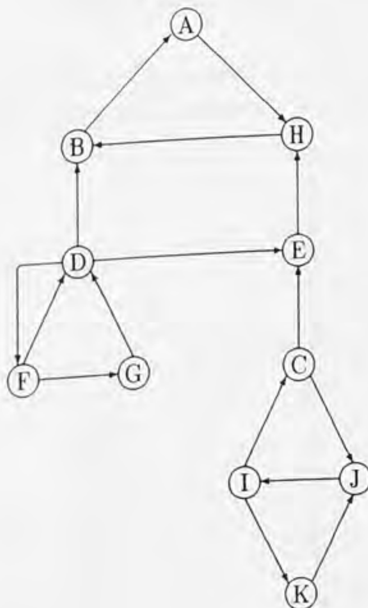
Παράδειγμα 6.4.4: Αν ο Αλγόριθμος 6.4.2 τρέξει στο γράφημα (6.10α), θα φυλαχθούν οι εξής πληροφορίες στους κόμβους:

label[A] = 1, finished[A] = 22
label[B] = 2, finished[B] = 21
label[D] = 3, finished[D] = 8
label[F] = 4, finished[F] = 5
label[G] = 6, finished[G] = 7
label[H] = 9, finished[H] = 20
label[E] = 10, finished[E] = 19
label[C] = 11, finished[C] = 18
label[I] = 12, finished[I] = 17
label[J] = 13, finished[J] = 16
label[K] = 14, finished[K] = 15

Ας εξετάσουμε τον κόμβο που έχει την μεγαλύτερη τιμή στο πεδίο finished, δηλαδή τον κόμβο τελείωσε τελευταίος (στο Παράδειγμα 6.4.4 ο κόμβος A). Ο κόμβος αυτός ανήκει σε μια συνεκτική συνιστώσα (μπορεί και να αποτελεί από μόνος του συνεκτική συνιστώσα). Ποιοί είναι οι υπόλοιποι κόμβοι (αν υπάρχουν) της συνιστώσας αυτής; Οσοι κόμβοι X του γραφήματος μπορούν να συνδεθούν με τον A (υπάρχει δηλαδή μονοπάτι από τον X στον A). (Στο Παράδειγμα οι κόμβοι B και H με το μονοπάτι $< B, H, A >$.) Για να αποτελουν όμως οι A, B, H συνεκτική συνιστώσα, θα πρέπει να υπάρχει και μονοπάτι από τον A προς τους B και H. Αυτό το έχουμε εξασφαλίσει από το γεγονός ότι οι B και H έχουν μικρότερη τιμή finished από τον A (επειδή τον A τον επιλέξαμε να έχει τη μεγαλύτερη τιμή finished από όλους τους κόμβους). (Να αποδείξετε ότι αν ο X τελειώσει μετά τον A, τότε υπάρχει μονοπάτι από τον A στον X.) Ας αφαιρέσουμε τώρα όλη τη συνιστώσα του A από το γράφημα και ας επιλέξουμε τον κόμβο με τη μεγαλύτερη τιμή finished από τους υπόλοιπους (στο παράδειγμα ο κόμβος E). Ποιοί είναι οι υπόλοιποι κόμβοι της συνιστώσας του E; Σίγουρα κανένας από τους κόμβους που αφαιρέσαμε, γιατί με κανέναν από τους κόμβους αυτούς δεν συνδέεται ο E (αν συνδεόταν, θα υπήρχε μονοπάτι από τον E στον A και θα είχαμε βάλει τον E στη συνιστώσα του A). Από τους υπόλοιπους κόμβους του γραφήματος βρίσκονται στη συνιστώσα του E εκείνοι, από τους οποίους υπάρχει μονοπάτι προς τον E. Αφαιρούμε τον E και τη συνιστώσα του από το γράφημα και συνεχίζουμε με τον ίδιο τρόπο έως ότου εξαντληθούν οι κόμβοι.

Η διαδικασία της προηγουμένης παραγράφου είναι η βασική ιδέα του αλγο-

ρίθμου που ακολουθεί για τον υπολογισμό των συνεκτικών συνιστωσών γραφήματος με κατεύθυνση. Αν αντιστρέψουμε τη φορά που εχουν τα βέλη του αρχικού γραφήματος G, παίρνουμε ένα νέο γράφημα G^T που ονομάζεται ανάστροφο (transpose) του αρχικού. (Στο Σχήμα (6.11) παρουσιάζεται το ανάστροφο γράφημα του γραφήματος του Σχήματος (6.10α)) Στο νέο γράφημα G^T μπορούμε πολύ εύκολα να εκτελέσουμε το βήμα της διαδικασίας που βρίσκει τους υπόλοιπους κόμβους στη συνιστώσα ενός κόμβου A, ο οποίος έχει την μεγαλύτερη τιμή finished από όλους τους εναπομείναντες κόμβους στο γράφημα: Τρέχουμε τον αλγόριθμο DFS(A). Οι κόμβοι που βρίσκονται στο δένδρο εξερεύνησης σε βάθος με ρίζα το A αποτελούν τη συνιστώσα του A. (Οι κόμβοι στο δένδρο με ρίζα A είναι οι κόμβοι προς τους οποίους υπάρχει μονοπάτι από τον A στο γράφημα G^T, συνεπώς οι κόμβοι από τους οποίους υπάρχει μονοπάτι προς τον A στο γράφημα G.)



Σχήμα (6.11)

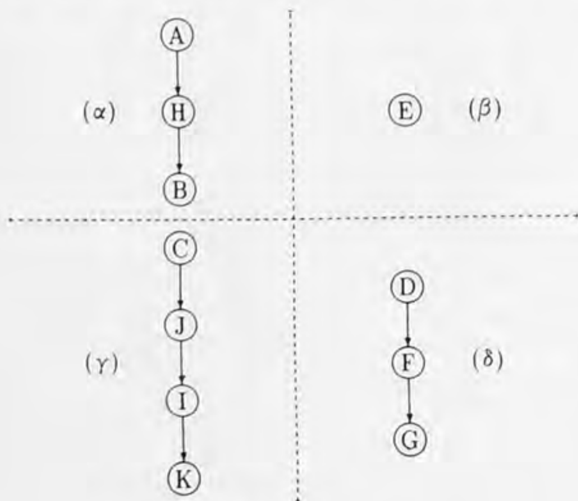
Αλγόριθμος 6.4.3: Συνεκτικές συνιστώσες γραφημάτων με κατεύθυνση.
Είσοδος: Γράφημα $V = (G, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

1. κάλεσε τον DFS(G) για να υπολογίσει την τιμή finished[x] κάθε κόμβου x
2. υπολόγισε το γράφημα G^T
3. κάλεσε τον DFS(G^T), αλλά στο κυρίως πρόγραμμα του DFS οι κόμβοι x εξετάζονται κατά φθίνουσα σειρά τῆς τιμής finished[x]
4. παρουσίασε στην έξοδο τους κόμβους κάθε DFS δένδρου ως διαφορετική συνιστώσα

Παράδειγμα 6.4.5: Ας τρέξουμε τον Αλγόριθμο 6.4.3 με είσοδο το γράφημα του Σχήματος (6.10α). Το Βήμα 1 έχει ήδη παρουσιαστεί στο Παράδειγμα (6.4.4). Το Βήμα 2 παρουσιάζεται στο Σχήμα (6.11). Η σειρά, με την οποία θα εξεταστούν οι κόμβοι στο Βήμα 3 είναι η εξής:

$A, B, H, E, C, I, J, K, D, G, F$

Τα DFS δένδρα, και συνεπώς οι συνεκτικές συνιστώσες παρουσιάζονται στο Σχήμα (6.12) με τη σειρά που παρουσιάζονται από τον DFS.



Σχήμα 6.12

ΧΡΟΝΟΣ

Η ανάλυση του Αλγορίθμου 6.4.1 είναι ανάλογη με την ανάλυση του BCDFS. Για τον Αλγόριθμο DFS, που αποτελεί τον σκελετό του SCDFS, ο χρόνος είναι $\mathcal{O}(|E|)$. Κάθε κόμβος μπαίνει και βγαίνει από τη στοίβα μία φορά, άρα και ο SCDFS εκτελέιται σε χρόνο $\mathcal{O}(|E|)$. Εύκολα μπορούμε να αποδείξουμε ότι και ο Αλγόριθμος 6.4.3 εκτελείται σε χρόνο $\mathcal{O}(|E|)$, αφού κάθε ένα από τα βήματα 1, 2, 3 του αλγορίθμου εκτελείται σε χρόνο $\mathcal{O}(|E|)$.

6.5. Μη αναδρομική εξερεύνηση σε βάθος, εξερεύνηση κατά πλάτος

Εστω ότι θέλουμε να αφαιρέσουμε την αναδρομή από τον Αλγόριθμο DFS της Ενότητας 6.1 (Αλγόριθμος 6.1.1). Για να εξομοιώσουμε έναν αναδρομικό αλγόριθμο με ένα μη αναδρομικό χρειαζόμαστε μια στοίβα. Ουσιαστικά, η αναδρομή μας παρέχει μία στοίβα. Σε όλους τους αλγορίθμους του κεφαλαίου αυτού καλούμε τη γνωστή υπορουτίνα visit(x), η οποία σημαδεύει τον κόμβο (visited[x] ← **true**) και του δίνει μοναδική επιγραφή label[x] από ένα μετρητή που παίρνει αρχική τιμή στο κυρίως πρόγραμμα και αυξάνεται σε κάθε κλήση της visit().

Αλγόριθμος 6.5.1: Μη αναδρομική εξερεύνηση σε βάθος.
Είσοδος: Γράφημα $V = (G, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

```
DFS(a);
begin
  push(a);
  while η στοίβα έχει στοιχεία do begin
    pop(x);
    if not visited[x] then begin
      visit(x);
      for each y ∈ adj(x) do
      if not visited[y] then push(y)
    end
  end
end

  begin
  for x = 1, 2, ..., |V| do visited[x] ← false;
  for x ← 1 to n do
    if not visited[x] then DFS(x)
  end.
```

Ελέγξτε με παραδείγματα ότι οι Αλγόριθμοι 6.1.1 και 6.5.1 έχουν τα ίδια αποτελέσματα.

Αν αντί για στοίβα χρησιμοποιήσουμε ουρά αναμονής (queue), τότε προκύπτει ένας άλλος αλγόριθμος που ονομάζεται **εξερεύνηση κατά πλάτος** (breadth first search). Ονομάζουμε add(x) τη συνάρτηση που προσθέτει ένα στοιχείο στο τέλος της ουράς αναμονής και remove(x) τη συνάρτηση που αφαιρεί ένα στοιχείο από το τέλος της ουράς. (Κώδικας για τις συναρτήσεις add(x)- εισαγωγή στοιχείου σε ουρά-και remove(x)- εξαγωγή στοιχείου από ουρά- βρίσκεται στην Ενότητα 3.2.5 της αναφοράς [5].)

Αλγόριθμος 6.5.2: Εξερεύνηση κατά πλάτος.
Είσοδος: Γράφημα $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

```
BFS(a);
begin
  add(a);
  while η ουρά έχει στοιχεία do begin
    remove(x);
    if not visited[x] then begin
      visit(x);
      for each y ∈ adj(x) do
      if not visited[y] then add(y)
    end
  end
end
  begin
  for x = 1, 2, ..., |V| do visited[x] ← false;
  for x ← 1 to n do
    if not visited[x] then BFS(x)
  end.
```

Η κατασκευή του δένδρου εξερεύνησης σε βάθος παρουσιάστηκε στην Ενότητα 6.1. Για να κατασκευάσουμε το δένδρο εξερεύνησης κατά πλάτος (Breadth First Search tree, BFS δένδρο), να βρούμε δηλαδή τις ακμές που μας οδηγούν σε νέους ανεξερεύνητους κόμβους κατά την εκτέλεση του αλγορίθμου, αρκεί να παρατηρήσουμε το εξής: Αν η ακμή $[x, y]$ ανήκει στο BFS δένδρο και ο x είναι γονέας του y, τότε ο x έχει τη μικρότερη επιγραφή (label[x]) από όλους τους γειτονικούς κόμβους του y. Το συμπέρασμα αυτό είναι προφανές από την FIFO (First In First Out, δηλαδή το πρώτο στοιχείο που εισάγεται στην

ουρά είναι το πρώτο που βγαίνει) ιδιότητα της ουράς αναμονής. Άρα, πατέρας κάθε κόμβου στο BFS δένδρο είναι αυτός που τον πρωτοβάζει στην ουρά. Την πληροφορία για τον πατέρα κάθε κόμβου x την φυλάμε στο πεδίο parent[x] και προσθέτουμε ένα ακόμα boolean πεδίο, το discovered[x], που έχει την τιμή **true** αν ο κόμβος έχει μπει στην ουρά. Μια άλλη χρήσιμη πληροφορία, η οποία φυλάγεται στο πεδίο d[x], είναι η απόσταση κάθε κόμβου x από τη ρίζα του BFS δένδρου (δηλαδή ο αριθμός των ακμών στο μοναδικό μονοπάτι από την ρίζα στον x). Ο λόγος, για τον οποίο η απόσταση δεν αναφέρθηκε στον Αλγόριθμο DFS και αναφέρεται στον BFS είναι ο εξής: *Στο BFS δένδρο η απόσταση κάθε κόμβου από τη ρίζα είναι το μήκος (σε αριθμό ακμών) του ελάχιστου μονοπατιού από τη ρίζα.* Το DFS δένδρο δεν έχει την ιδιότητα αυτή. Οι λίγες τροποποιήσεις που πρέπει να γίνουν στον Αλγόριθμο 6.5.2 ώστε να ενημερώνει τα πεδία που προσθέσαμε φαίνονται στο Αλγόριθμο 6.5.3.

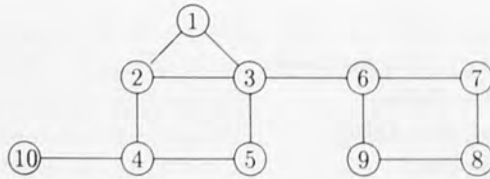
Αλγόριθμος 6.5.3: Εξερεύνηση κατά πλάτος.
Είσοδος: Γράφημα $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$. Το γράφημα παριστάνεται με μήτρα γειτνίασης, όπως περιγράφεται στην Ενότητα 5.2.

```
BFS(a);
begin
  add(a); d[a]←0; parent[a]←nil;
  while η ουρά έχει στοιχεία do begin
    remove(x);
    if not visited[x] then begin
      visit(x);
      for each y ∈ adj(x) do
      if not discovered[y] then begin
        discovered[y] ←true;
        parent[y] ← x; d[y] ←d[x] + 1;
        add(y);
        end
      end
    end
end
```

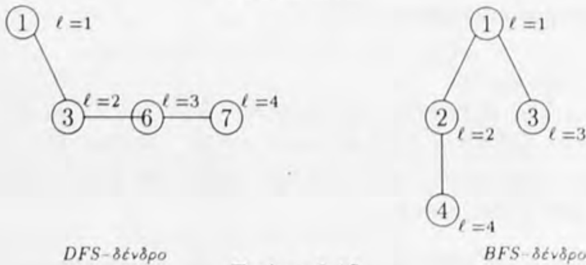
```
begin
for x = 1, 2, ..., |V| do begin
   visited[x] ← false;
   discovered[x] ← false;
   d[x] ← bignum
end
for x ← 1 to n do
   if not visited[x] then BFS(x)
end.
```

Παράδειγμα: Στα σχήματα 6.1β, 6.1γ και 6.1δ φαίνονται τρία στιγμιότυπα κατασκευής των DFS και BFS δένδρων για το γράφημα του σχήματος 6.1α. Με το γράμμα ℓ παριστάνεται η τιμή του πεδίου label κάθε κόμβου, το οποίο παίρνει τιμή στη ρουτίνα visit.

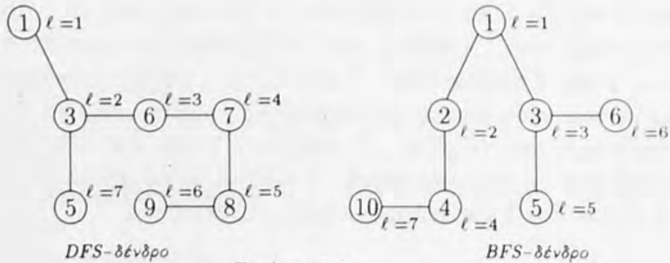
Παρατηρείστε ότι το BFS δένδρο περιλαμβάνει τα μικρότερα μονοπάτια (σε αριθμό ακμών) από τη ρίζα (που είναι ο κόμβος 1) σε κάθε κόμβο. Για παράδειγμα, οι κόμβοι 1 και 2 συνδέονται με μια ακμή που είναι το ελάχιστο μονοπάτι μεταξύ τους. Αντίθετα, στο DFS δένδρο το μονοπάτι ανάμεσα στους κόμβους 1 και 2 περιλαμβάνει 4 ακμές. Στο παράδειγμα του σχήματος 6.1 οι κόμβοι του γραφήματος χωρίζονται σε ομάδες ανάλογα με την ελάχιστη απόστασή τους από τη ρίζα . Οι κόμβοι $2, 3$ έχουν $d = 1$, δηλαδή ελάχιστη απόσταση από τη ρίζα μία ακμή, οι κόμβοι $4, 5, 6$ έχουν $d = 2$, οι κόμβοι $10, 9, 7$ έχουν $d = 3$ και τέλος ο κόμβος 8 έχει $d = 4$.



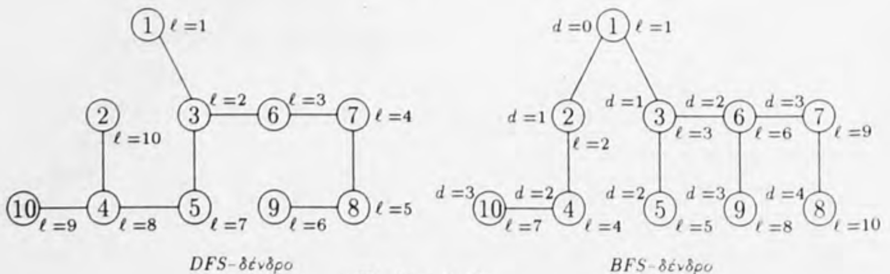
Σχήμα 6.1α



Σχήμα 6.1β



Σχήμα 6.1γ



Σχήμα 6.1δ

Κεφάλαιο 7

ΠΡΟΒΛΗΜΑΤΑ ΣΕ ΓΡΑΦΗΜΑΤΑ ΜΕ ΒΑΡΗ

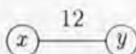
7.1. Ελάχιστα μονοπάτια

Η εξερεύνηση κατά πλάτος βρίσκει τα ελάχιστα μονοπάτια (μονοπάτια με ελάχιστο αριθμό ακμών) από κάποιο αρχικό κόμβο προς όλους τους κόμβους του γραφήματος, οι οποίοι είναι συνδεδεμένοι με τον αρχικό. Οταν κάναμε εξερεύνηση κατά πλάτος θεωρούσαμε ότι όλες οι ακμές έχουν το ίδιο μήκος, μήκος ίσο με 1. Τι γίνεται όμως αν δεν έχουν όλες οι ακμές το ίδιο μήκος ή αλλιώς αν το γράφημα έχει βάρη στις ακμές; Τότε το βάρος (μήκος) του μονοπατιού είναι το άθροισμα των βαρών των ακμών του. Ενα μονοπάτι από τον x στον y είναι ελάχιστο αν δεν υπάρχει μονοπάτι από τον x στον y με μικρότερο βάρος. Στο εξής θα χρησιμοποιούμε και τους δύο όρους (μήκος ή βάρος) για τον μοναδικό ακέραιο που αντιστοιχείται σε κάθε ακμή στα γραφήματα με βάρη στις ακμές. Οπως αναφέραμε στην εισαγωγή, ο αριθμός αυτός μπορεί να εκφράσει μήκος, κόστος, χωρητικότητα, χρόνο ή οτιδήποτε άλλο απαιτείται από την εφαρμογή.

Το πρόβλημα του ελάχιστου μονοπατιού έχει πολλές εφαρμογές. Για παράδειγμα, σε ένα γράφημα όπου οι κόμβοι παριστάνουν πόλεις, οι ακμές διαδρομές αεροπλάνων και τα βάρη των ακμών το κόστος των εισιτηρίων, το ελάχιστο μονοπάτι είναι ο οικονομικότερος τρόπος να ταξιδέψουμε από μια πόλη σε άλλη.

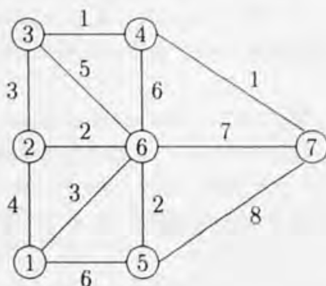
Μπορούμε να λύσουμε το πρόβλημα του ελάχιστου μονοπατιού σε γρα-

φήματα με βάρη με τον ήδη γνωστό αλγόριθμο εξερεύνησης κατά πλάτος αν μετασχηματίσουμε το αρχικό γράφημα σε ένα νέο γράφημα που έχει βάρη 1 σε όλες τις ακμές, ως εξής: Μια ακμή με βάρος 12 , για παράδειγμα, μπορούμε να τη θεωρήσουμε 12 γειτονικές ακμές βάρους 1 που συνδέουν 11 νέους ενδιάμεσους κόμβους. Δηλαδή:



Αφού μετασχηματίσουμε το γράφημα, κάνουμε εξερεύνηση κατά πλάτος στο νέο γράφημα. Η πρώτη κλήση του BFS γίνεται για τον κόμβο, από τον οποίο θέλουμε να μετρήσουμε ελάχιστα μονοπάτια. Η τιμή του πεδίου d κάθε κόμβου (από τους κόμβους του αρχικού γραφήματος) είναι το ελάχιστο μονοπάτι από τον αρχικό κόμβο.

Ο αλγόριθμος του Dijkstra υπολογίζει τα ελάχιστα μονοπάτια από ένα κόμβο με ένα τρόπο πιο έξυπνο από αυτόν που αναφέρθηκε στην προηγούμενη παράγραφο. Ο αλγόριθμος αυτός δίνει σε κάθε κόμβο μια προτεραιότητα, που είναι μια εκτίμηση, μια πρόβλεψη, της ελάχιστης απόστασης του από τον αρχικό κόμβο. Η εκτίμηση αυτή βασίζεται καθε στιγμή στις τιμές των ελάχιστων μονοπατιών που έχουν ήδη υπολογιστεί. Η πρόβλεψη για την τιμή του ελάχιστου μονοπατιού προς ένα κόμβο μπορει να μεταβληθεί όσο υπολογίζουμε τις τιμές των ελάχιστων μονοπατιών προς άλλους κόμβους. Είμαστε όμως σίγουροι για τα τα εξής: (α) η πρόβλεψη είναι πάντα μεγαλύτερη ή ίση της ελάχιστης απόστασής και (β) *η μικρότερη από όλες τις προβλέψεις μας είναι πάντα σωστή.*



Σχήμα (7.1)

Παράδειγμα: Θέλουμε να βρούμε το ελάχιστο μονοπάτι από τον κόμβο 1 στον κόμβο 7 στο γράφημα του Σχήματος (7.1). Θα υπολογίσουμε αναγκαστικά τα ελάχιστα μονοπάτια από τον 1 προς **όλους** τους κόμβους του γραφήματος (όλοι οι αλγόριθμοι που ξέρουμε λειτουργούν με τον τρόπο αυτό). Ποιά είναι η αρχική πρόβλεψη για τα ελάχιστα μονοπάτια;

(2, 4) κόμβος 2, με απόσταση 4, ελάχιστο μονοπάτι 4,

(6, 3) κόμβος 6, με απόσταση 3, ελάχιστο-μονοπάτι 3,

(5, 6) κόμβος 5, με απόσταση 6, ελάχιστο μονοπάτι 6.

Η τελευταία πρόβλεψη (5, 6) είναι λανθασμένη. Στην πραγματικότητα το ελάχιστο μονοπάτι προς τον κόμβο 5 έχει μήκος 5 και είναι μέσω του κόμβου 6. Άρα, η αρχική πρόβλεψη για τον κόμβο 5 μειώνεται όταν υπολογίσουμε το ελάχιστο μονοπάτι πρός τον 6. Ομως, η μικρότερη από όλες τις προβλέψεις (τιμή 3 για το ελάχιστο μονοπάτι προς τον κόμβο 6) είναι σίγουρα σωστή. Εφόσον ο 6 έχει την μικρότερη προβλεπόμενη τιμή για το ελάχιστο μονοπάτι, αποκλείεται να πάμε σε κάποιον άλλο κόμβο *πριν* από τον 6 και αυτή η νέα διαδρομή να έχει μικρότερο μήκος. Σε κάθε βήμα, προσθέτουμε ένα νέο κόμβο στο σύνολο των κόμβων, για τους οποίους έχουμε υπολογίσει τα ελάχιστα μονοπάτια. Ο κόμβος αυτός επιλέγεται ως εξής: είναι κάθε φορά αυτός, ο οποίος έχει την μικρότερη προβλεπόμενη τιμή. Οι πράξεις, λοιπόν, που γίνονται από τον αλγόριθμο στις προβλεπόμενες τιμές για τα ελάχιστα μονοπάτια είναι οι εξής: (α) αναζήτηση της μικρότερης από τις τιμές αυτές και (β) μείωση μίας από τις τιμές αυτές. Είναι προφανές ότι η δομή σωρός (heap) προσφέρεται για να φυλάμε τις προβλεπόμενες τιμές αυτές.

Ενας άλλος τρόπος να κατανοήσουμε τον αλγόριθμο του Dijstra είναι να θεωρήσουμε ότι το ελάχιστο μονοπάτι από ένα κόμβο x σε έναν άλλο y είναι ο χρόνος που χρειάζεται ένα ηχητικό κύμα, το οποίο μεταδίδεται πάνω στις ακμές διασχίζοντας κάθε ακμή σε χρόνο ίσο με το βάρος της (στο μετασχηματισμένο γράφημα σε μοναδιαίο χρόνο), για να φτάσει από τον x στον y. Υπάρχουν πολλά μονοπάτια από τον x στον y (πολλές διαδρομές που ακολουθεί το κύμα για να φτάσει από τον x στον y). Ο χρόνος που θα κάνει για να φτάσει *πρώτη φορά* στον y (το χρονικό διάστημα που μεσολαβεί ανάμεσα στη στιγμή που δημιουργήθηκε το κύμα στον x και τη στιγμή που το πρωτοακούσαμε στον y) είναι το μήκος (βάρος) του ελάχιστου μονοπατιού.

Οι πληροφορίες που φυλάμε στο εγγράφημα κάθε κόμβου x κατά την εκτέλεση του αλγορίθμου που ακολουθεί είναι οι εξής:

dist[x]: προβλεπόμενη ελάχιστη απόσταση του x από τον 1.

done[x]: έχει την τιμή **true** αν έχουμε βρεί την ελάχιστη απόσταση του x από τον 1, που είναι η dist[x].

prev[x]: ο προηγούμενος κόμβος του x στο ελάχιστο μονοπάτι από τον 1.

Με d(x, y) συμβολίζεται το μήκος (βάρος) της ακμής $[x, y]$.

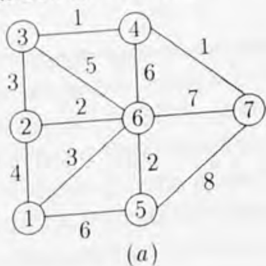
Σημείωση: Ο αλγόριθμος που ακολουθεί δέχεται ως είσοδο ένα γράφημα με βάρη χωρίς κατεύθυνση. Ο ίδιος αλγόριθμος ισχύει και για γραφήματα με κατεύθυνση.

Αλγόριθμος 7.1.1: Ελάχιστα μονοπάτια από ένα κόμβο.

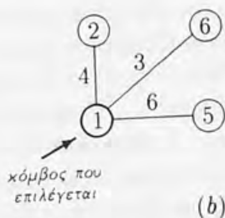
Είσοδος: Γράφημα $G = (V, E)$ με μη αρνητικά βάρη στις ακμές, και ο κόμβος 1, από τον οποίο μετράμε τις αποστάσεις.

Εξοδος: Για κάθε κόμβο x dist[x] είναι το μήκος του ελάχιστου μονοπατιού από τον 1 στον x.

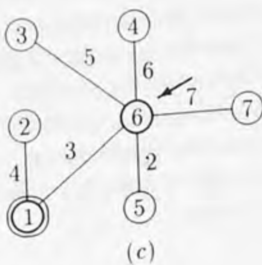
```
Shortest-paths;
begin
  for all x do begin
    dist[x] ← bignum; done[x] ← false
    end;
  dist[1]← 0;
  repeat
    από όλα τα x με done[x] = false διάλεξε εκείνο με τη μικρότερη
dist[x];
    done[x] ← true;
    for each y ∈adj(x) do
      if not done[y] then
        if dist[y] ≥ dist[x]+d[x,y] then begin
          dist[y] ← dist[x]+d[x,y];
          prev[y] ← x
          end
  until όλοι οι κόμβοι να έχουν done[ ] = true.
```



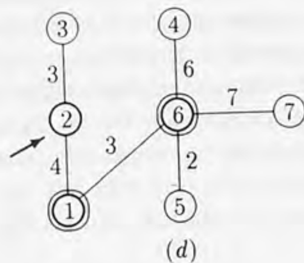
(a)



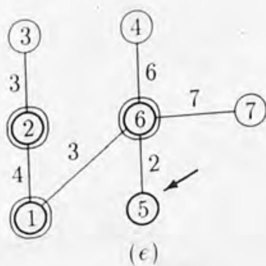
(b)



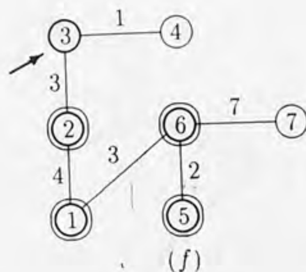
(c)



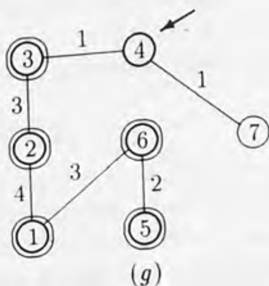
(d)



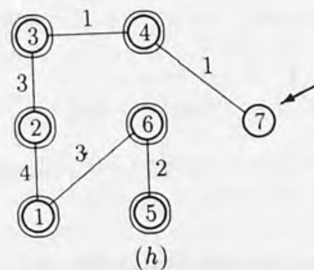
(ϵ)



(f)



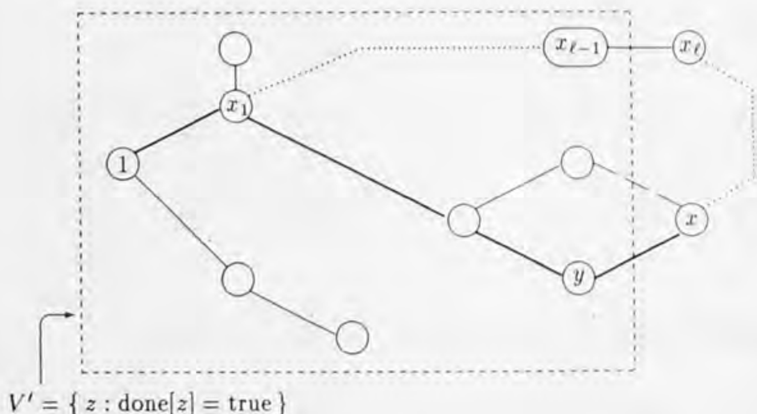
(g)



(h)

Σχήμα (7.2). Εκτέλεση του αλγόριθμου 7.1.1 για το γράφημα του Σχήματος 7.2α.

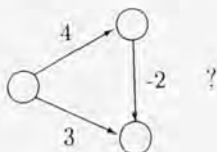
Πως αποδεικνύεται ότι ο αλγόριθμος αυτός βρίσκει τα ελάχιστα μονοπάτια; Το βήμα που πρέπει να συζητηθεί είναι εκείνο που διαλέγει τον επόμενο κόμβο, δηλαδή το βήμα "από όλα τα x που έχουν done[x] = **false** διάλεξε εκείνο με τη μικρότερη dist[x]". Η επιλογή αυτή είναι μια επιλογή κοντόφθαλμη ή άπληστη (greedy): διαλέγουμε αυτό που εκείνη τη στιγμή, τοπικά, είναι καλλίτερο και ελπίζουμε να είναι τμήμα της σφαιρικά καλλίτερης (βέλτιστης) λύσης. Σε ορισμένα προβλήματα οι άπληστοι αλγόριθμοι οδηγούν σε βέλτιστη λύση, σε άλλα όχι. Ο αλγόριθμος του Dijkstra γαι τα ελάχιστα μονοπάτια είναι μια περίπτωση όπου η άπληστη επιλογή οδηγεί σε βέλτιστη λύση. Το βήμα που επιλέγει τον επόμενο κόμβο x επιλέγει και ένα μονοπάτι από τον 1 στον x, το οποίο μπορούμε να βρούμε ακολουθώντας τα πεδία prev[ ] από τον x. Για να αποδείξουμε ότι το βήμα αυτό είναι σωστό πρέπει να αποδείξουμε ότι το μονοπάτι που επιλέγεται με τον τρόπο αυτόν είναι μικρότερο ή ίσο από *οποιοδήποτε* άλλο μονοπάτι από τον 1 στον x (ακόμα και από μονοπάτια που περιλαμβάνουν κόμβους που δεν έχουμε ακόμα εξετάσει).



Σχήμα (7.3)

Εστω ότι από όλους τους κόμβους V του γραφήματος για ένα υποσύνολο V' έχουμε ήδη υπολογίσει τα ελάχιστα μονοπάτια (δηλαδή κάθε κόμβος v του συνόλου V' έχει done[v]=**true**). (Βλέπε και το Σχήμα (7.3). Στο σχήμα αυτό οι κόμβοι του V' είναι μέσα σε τετράγωνο.) Εστω x ο επόμενος κόμβος

που επιλέγεται με το κριτήριο να έχει dist[x] ελάχιστο. Ο κόμβος αυτός συνδέεται με κάποιο από τους κόμβους του V' (έστω τον y) με την ακμή $[x, y]$ (prev[x]=y). Εστω P το μονοπάτι από τον 1 στον x που περιλαμβάνει την $[x, y]$. Θα αποδείξουμε ότι το P είναι ελάχιστο. Εστω P' οποιοδήποτε μονοπάτι $< 1, x_1, \ldots, x_{l-1}, x_l, \ldots, x >$ από τον 1 στον x και x_{l-1} ο τελευταίος κόμβος του P' που έχει done[x_{l-1}]=**true**. Ο x_l μπορεί να είναι ίδιος με τον x, αν το μονοπάτι δεν περιλαμβάνει κόμβους που δεν έχουμε ακόμη εξετάσει, (βλέπε διακεκομμένη γραμμή στο Σχήμα (7.3)). Στην περίπτωση αυτή το αποτέλεσμα προκύπτει αμέσως από τον τρόπο υπολογισμού του dist. Ισχύει, δηλαδή, ότι dist[x_l] $\geq$dist[x], γιατι αν δεν ίσχυε ο αλγόριθμος θα είχε επιλέξει το x_l. Αρα, το τμήμα του P' εως τον x_l είναι μεγαλύτερο ή ισο από ολόκληρο το P. Εφόσον μόνο θετικά βάρη προστίθενται, ολόκληρο το P' είναι μεγαλύτερο ή ίσο από το P. Το επιχείρημα προφανώς δεν ισχύει αν έχουμε αρνητικά βάρη (βλέπε Σχήμα (7.4)).



Σχήμα (7.4)

Στην περίπτωση που υπάρχουν αρνητικά βάρη μπορεί να χρησιμοποιηθεί ο αλγόριθμος των Bellman–Ford, ο οποίος δεν περιγράφεται στις σημειώσεις αυτές. Ο αλγόριθμος αυτός λύνει το πρόβλημα των ελάχιστων μονοπατιών από ένα κόμβο στην περίπτωση γενικών βαρών σε χρόνο $O(|E||V|)$.

ΧΡΟΝΟΣ

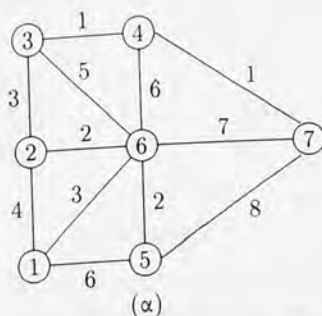
Ο χρόνος του αλγορίθμου είναι $\mathcal{O}(|V|^2)$. Ο βρόγχος του **repeat** θα εκτελεστεί $|V|$ φορές (κάθε φορά μεταβάλεται το done ενος κόμβου και τελειώνει όταν όλα τα done έχουν γίνει **true**). Σε κάθε επανάληψη επιλέγεται ο κόμβος με το μικρότερο dist, επιλογή που απαιτεί χρόνο $\mathcal{O}(|V|)$. Εξάλλου, ελέγχουμε όλες τις ακμές (με την εντολή **for** each y $\in$adj(x) που εκτελείται για όλους τους κόμβους του γραφήματος), άρα για όλες τις επαναλήψεις απαιτείται χρόνος $\mathcal{O}(|E|)$.

Σημείωση: Ο αλγόριθμος μπορεί να υλοποιηθεί και σε χρόνο $\mathcal{O}(|E|\log|V|)$ αν χρησιμοποιηθεί μια σωρός (heap) όπου φυλάγονται οι κόμβοι με κλειδί την τιμή του dist. (Η υλοποίηση αυτή είναι προτιμότερη για αραιά γραφήματα.) Οι ουρές προτεραιότητας περιγράφονται στην ενότητα 5.7 της αναφοράς [6]. Τότε, σε κάθε επανάληψη του βρόγχου **repeat** η επιλογή του κόμβου με το μικρότερο dist υλοποιείται σε χρόνο $\mathcal{O}(\log|V|)$, αρα για όλες τις επαναλήψεις ο χρόνος είναι $\mathcal{O}(|V|\log|V|)$. Γιά κάθε ακμή που ελέγχεται μπορεί να μειωθεί το dist ενός κόμβου, οπότε χρειάζεται αναπροσαρμογή της θέσης του κόμβου αυτού στην ουρά, η οποία υλοποιείται σε χρόνο $\mathcal{O}(\log|V|)$ (αν βεβαίως μπορούμε να εντοπίσουμε τη αρχική θέση του κόμβου στην ουρά σε σταθερό χρόνο). Αρα, επειδή, όπως αναφέραμε, ελέγχονται όλες οι ακμές ο τελικός χρόνος είναι $\mathcal{O}(|E|\log|V|)$.

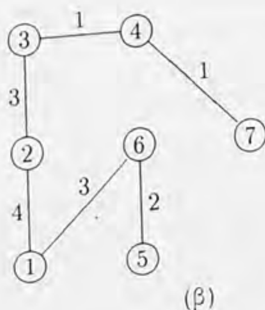
7.2. Ελάχιστα δένδρα

Ενα **επικαλυπτικό δένδρο** (spanning tree) γραφήματος G είναι ένα υπογράφημα του G που είναι δένδρο και περιλαμβάνει όλους τους κόμβους. Οταν το γράφημα έχει βάρη, **βάρος του επικαλυπτικού δένδρου** είναι το άθροισμα των βαρών των ακμών του. Το **πρόβλημα του ελάχιστου δένδρου** είναι το εξής:

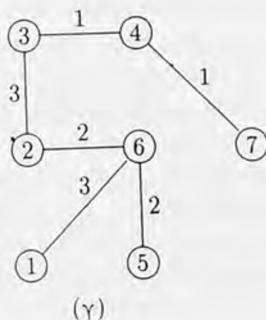
"Δεδομένου γραφήματος G, να βρεθεί επικαλυπτικό δένδρο με ελάχιστο βάρος"



(α)



(β)



(γ)

Σχήμα (7.5)

Παράδειγμα: Στα Σχήματα (7.5β) και (7.5γ) εικονίζονται επικαλυπτικά δένδρα του γραφήματος του Σχήματος (7.5α) (που είναι το ίδιο με το Σχήμα (7.1),

για το οποίο υπολογίσαμε τα ελάχιστα μονοπάτια). Βλέπουμε ότι το δένδρο του Σχήματος (7.5β) (που είναι το δένδρο των ελάχιστων μονοπατιών από τον 1) έχει βάρος 14 και δεν είναι ελάχιστο, ενώ το δένδρο του Σχήματος (7.5γ) είναι ελάχιστο με βάρος 12. Προσέξτε ακόμα ότι: το ελάχιστο δένδρο δεν περιλαμβάνει ελάχιστα μονοπάτια ανάμεσα σε κάθε ζεύγος κόμβων (βλέπε μονοπάτι από 1 σε 3). Το ελάχιστο δένδρο δεν είναι αναγκαστικά μοναδικό. Τα δένδρα που παρουσιάζονται στα Σχήματα (β), (γ) και (δ) του (7.6) είναι επικαλυπτικά του γραφήματος του Σχήματος (7.6α). Απο αυτά, τα δένδρα των σχημάτων(7.6γ) και (7.6δ) είναι ελάχιστα.

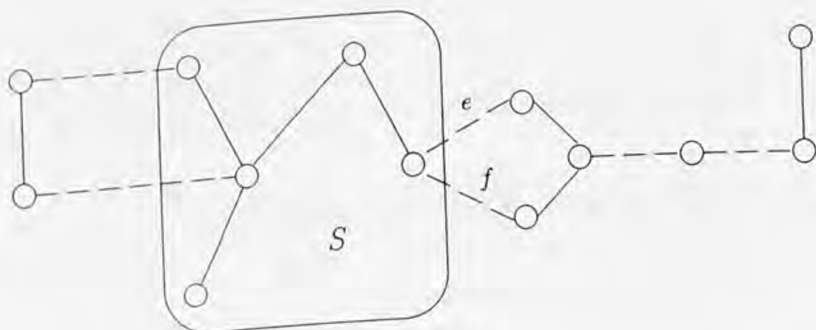
Σχήμα (7.6)

Πολλές είναι οι εφαρμογές του ελάχιστου δένδρου. Για παράδειγμα, θέλουμε το φθηνότερο σιδηροδρομικό οδικό ή τηλεφωνικό δίκτυο που συνδέει ένα σύνολο από πόλεις (στην περίπτωση αυτή κόμβοι είναι οι πόλεις και βάρος των ακμών το κόστος κατασκευής σιδηροδρομικής ή τηλεφωνικής γραμμής ή το κόστος κατασκευής δρόμου) ή θέλουμε το φθηνότερο τρόπο να συνδέσουμε ένα σύνολο τερματικών κ.λ.π.

Προφανώς αν το αρχικό γράφημα δεν είναι συνεκτικό, τότε δεν υπάρχει επικαλυπτικό δένδρο. Στην περίπτωση αυτή έχουμε δάσος από δένδρα. Στο υπόλοιπο του κεφαλαίου θεωρουμε ότι η είσοδός μας είναι συνεκτικό γράφημα.

Η βασική ιδέα των αλγορίθμων που θα περιγράψουμε είναι η εξής: Το ελάχιστο δένδρο κατασκευάζεται σιγά-σιγά. Σε ένα ενδιάμεσο στάδιο έχουμε ένα δάσος. Βλέπε και το Σχήμα (7.7). Το σχήμα αυτό παριστάνει ένα ενδιάμεσο στάδιο καταχευής του δένδρου. Εχουμε ήδη κατασκευάσει δάσος που αποτελείται από 5 δενδράκια (5 συνεκτικές συνιστώσες). Οι ακμές, με τις οποίες είναι δυνατόν να συνδεθούν οι συνιστώσες ώστε να αποτελέσουν ένα ενιαίο δένδρο, παρουσιάζονται με διακεκομμένες γραμμές. Ποιά από τις διακεκομμένες γραμμές θα επιλέξουμε για να προσθέσουμε στο δένδρο στο

επόμενο βήμα; Η απάντηση είναι απλή: Από όλες τις διακεκομμένες γραμμές (τις ακμές, δηλαδή, που συνδέουν ένα κόμβο μιας συνιστώσας με ένα κόμβο εκτός της συνιστώσας) διαλέγουμε εκείνη με το μικρότερο βάρος.



Σχήμα (7.7)

Βασική ιδιότητα: "Εστω e η **μικρότερη** ακμή που συνδέει έναν κόμβο κάποιας συνιστώσας S με έναν κόμβο εκτός της S. Ανάμεσα σε **όλους τους τρόπους να προχωρήσουμε**, το να προσθέσουμε την e είναι βέλτιστος".

Απόδειξη: Εστω ότι όχι, και υπάρχει βέλτιστο δένδρο που δεν περιέχει την e. Προσθέτουμε την e. Δημιουργείται κύκλος που περιέχει κάποια "νέα" ακμή την f. Η f ήταν συνυποψήφια της e, δηλαδή *δεν* είχε ήδη προστεθεί σε κάποια συνιστώσα και μπορούσε να προστεθεί στο στάδιο αυτό. (Που ξέρουμε ότι υπάρχει f με τις ιδιότητες αυτές;) Απαλείφουμε την f και έχουμε καλλίτερο δένδρο! Αυτό ισχύει επειδή η e έχει το μικρότερο βάρος ανάμεσα σε **όλες** τις ακμές που συνδέουν έναν κόμβο της S με έναν εκτός S, άρα μικρότερο βάρος από την f. (Που ξέρουμε ότι το γράφημα που προκύπτει με πρόσθεση της e και αφαίρεση της f είναι δένδρο;)

Ασκηση Να απαντήσετε στα ερωτήματα που βρίσκονται σε παρένθεση στην απόδειξη.

Ολοι οι αλγόριθμοι που θα περιγράψουμε είναι κοντόφθαλμοι (άπληστοι), κάνουν δηλαδή την επιλογή που είναι βέλτιστη σε κάθε στιγμή και βρίσκουν τελικά σφαιρικά βέλτιστη λύση. Γνήσιος άπληστος αλγόριθμος είναι ο 7.2.4

Στους αλγορίθμους που ακολουθούν χρησιμοποιούνται οι εξής μεταβλητές:

S: το σύνολο των κόμβων που έχουν ήδη προστεθεί στο δένδρο.

T: το σύνολο των ακμών που έχουν προστεθεί στο δένδρο.

Ο πρώτος αλγόριθμος αρχίζει από ένα κόμβο (έστω τον 1) και προσπαθεί να επεκτείνει τη συνιστώσα που περιέχει τον κόμβο αυτόν (προσθέτοντας κάθε φορά τη μικρότερη ακμή που συνδέει έναν κόμβο της συνιστώσας με έναν που δεν ανήκει στη συνιστώσα) έως ότου η συνιστώσα αυτή γίνει επικαλυπτικό δένδρο. Οπως αποδείξαμε, το δένδρο που παράγεται με τον τρόπο αυτό είναι ελάχιστο.

Αλγόριθμος 7.2.1: Ελάχιστο επικαλυπτικό δένδρο (Prim).

Είσοδος: Γράφημα $G = (V, E)$ με βάρη στις ακμές.

Εξοδος: Οι ακμές σε ένα ελάχιστο δένδρο.

> $S \leftarrow \{1\}; T \leftarrow \emptyset;$
> **while** $S \neq V$ **do begin**
>   διάλεξε τη μικρότερη ακμή $[x, y]$ με $x \in S$ και $y \notin S$;
> $S \leftarrow S \cup \{y\}; T \leftarrow T \cup \{[x, y]\}$
> **end**.

Πόσος χρόνος απαιτείται από τον Αλγόριθμο 7.2.1; Αν δεν χρησιμοποιήσουμε καμία δομή δεδομένων, ο χρόνος είναι $\mathcal{O}(n^3)$, όπου $n = |V|$. Εχουμε n επαναλήψεις του βρόγχου **while** (επειδή σε κάθε επανάληψη προσθέτουμε ένα κόμβο στο δένδρο), και σε κάθε επανάληψη χρειαζόμαστε χρόνο $\mathcal{O}(n^2)$ για την εντολή "βρες τη μικρότερη ακμή $[x, y]$ με $x \in S$ και $y \notin S$".

Μια δομή δεδομένων διευκολύνει την προσπέλαση (βρίσκουμε πιο γρήγορα τη μικρότερη ακμή, η οποία έχει την ιδιότητα που θέλουμε), αλλά απαιτεί και ενημέρωση (κάθε φορά που προστίθεται κόμβος στο δένδρο αλλάζουν και οι υποψήφιες ακμές). Για να μας είναι χρήσιμη η δομή δεδομένων στον αλγόριθμο αυτό θα πρέπει το άθροισμα του χρόνου που απαιτείται για την προσπέλαση και του χρόνου που απαιτείται για την ενημέρωση να είναι μικρότερο από $\mathcal{O}(n^2)$ (επειδή όταν δεν χρησιμοποιούμε δομή, ο χρόνος για την προσπέλαση είναι $\mathcal{O}(n^2)$ και ο χρόνος ενημέρωσης 0).

Η δομή που είναι χρήσιμη στην περίπτωση αυτή είναι η εξής: Για κάθε κόμβο x που δεν έχει ακόμα προστεθεί στο δένδρο, δηλαδή $x \notin S$, κρατάμε

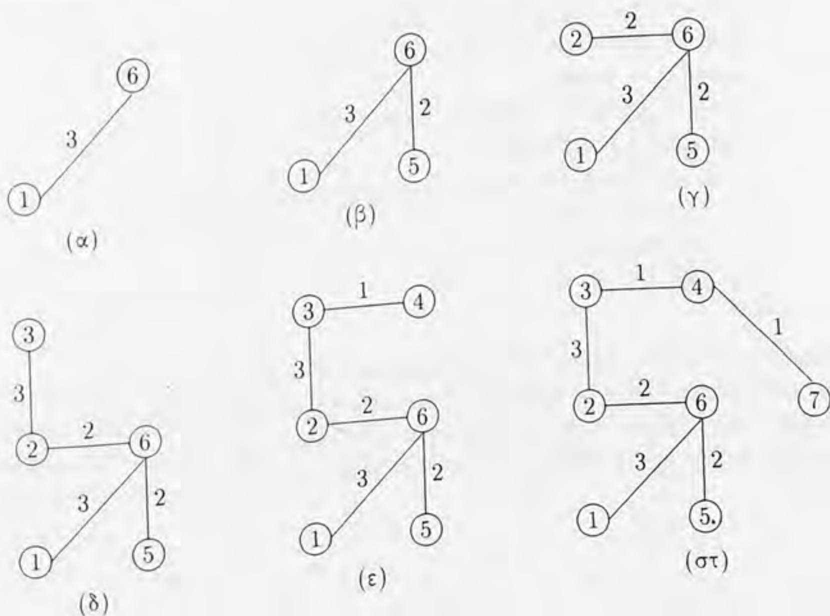
τις εξής πληροφορίες:

dist$[x]$: το μήκος της μικρότερης ακμής που συνδέει το x με κάποιο κόμβο στο S.

nhbr$[x]$: y αν $[x, y]$ είναι η μικρότερη ακμή.

Τη δομή αυτή μπορούμε να την προσπελάσουμε σε χρόνο $\mathcal{O}(n)$. Για να βρούμε τη μικρότερη ακμή εξετάζουμε όλους τους κόμβους που δεν έχουν προστεθεί στο δένδρο και διαλέγουμε εκείνο με το μικρότερο dist[ ]. Μπορούμε επίσης να την ενημερώσουμε σε χρόνο $\mathcal{O}(n)$ (για κάθε κόμβο που προστίθεται στο S ενημερώνεται το dist[ ] όλων των γειτονικών του κόμβων εκτός S).

Παράδειγμα: Στα Σχήματα (α), (β), (γ), (δ), (ε) και (στ) του (7.8) φαίνονται όλα τα στάδια ανάπτυξης του ελάχιστου δένδρου του Σχήματος (7.5) κατά τον Αλγόριθμο 7.2.1.



Σχήμα (7.8)

Οι τροποποιήσεις που πρέπει να γίνουν στον Αλγόριθμο 7.2.1 για να κάνει χρήση της δομής που αναφέραμε φαίνονται στον Αλγόριθμο 7.2.2. Ο Αλγόριθμος 7.2.2 θυμίζει πολύ τον αλγόριθμο του Dijkstra για ελάχιστα μονοπάτια.

Αλγόριθμος 7.2.2: Ελάχιστο επικαλυπτικό δένδρο (Dijkstra/Prim).

Είσοδος: Γράφημα $G = (V, E)$ με βάρη στις ακμές.

Εξοδος: Οι ακμές σε ένα ελάχιστο δένδρο.

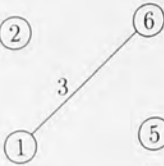
```
Minimum spanning tree;
begin
  for all x do begin
    dist[x] ← bignum; done[x] ← false
    end;
  dist[1]← 0; nhbr[1] ← nil;
  S ← {1}; T ← ∅
  while S ≠ V do begin
    από όλα τα x με done[x] = false
    διάλεξε εκείνο με τη μικρότερη dist[x];
    done[x] ← true;
    S ← S ∪ {x}; T ← T ∪ {[x, nhbr[x]]};
    for each y ∈adj(x) do
      if not done[y] and d(x,y) < dist[y] then begin
        dist[y] ← d(x,y);
        nhbr[y] ← x
        end
  end.
```

Παράδειγμα: Στο Σχήμα (7.9) εικονίζονται οι τιμές των dist και nhbr κατά την εκτέλεση του Αλγορίθμου 7.2.2 για κάθε βήμα κατασκευής του ελάχιστου δένδρου του Σχήματος (7.5). (Οπου δεν αναγράφονται τιμές στους κόμβους που δεν ανήκουν στο δένδρο, ισχύουν οι τιμές του προηγούμενου βήματος.)

$dist = 5$ $nhbr = 6$ (3)

(4) $dist = 6$ $nhbr = 6$

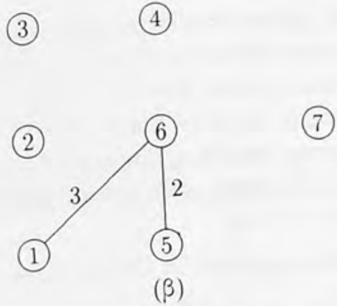
$dist = 2$ $nhbr = 6$ (2)



(7) $dist = 7$ $nhbr = 6$

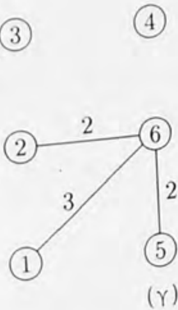
(5) $dist = 2$ $nhbr = 6$

(α)



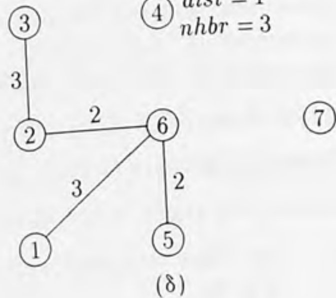
(β)

$dist = 3$ $nhbr = 2$ (3)

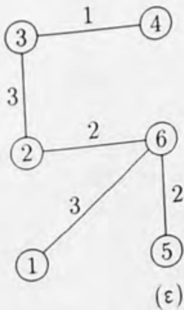


(7)

(γ)

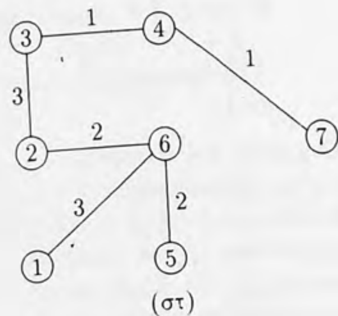


(δ)



(ε)

(7) $dist = 1$ $nhbr = 4$



(στ)

Σχήμα (7.9)

ΧΡΟΝΟΣ

Ο χρόνος που απαιτείται από τον Αλγόριθμο 7.2.2 είναι $\mathcal{O}(n^2)$. (Εχουμε n επαναλήψεις και σε κάθε επανάληψη $\mathcal{O}(n)$ βήματα για να βρούμε τον κόμβο με το μικρότερο dist και να ενημερώσουμε τη δομή.) Για πυκνά γραφήματα ο χρόνος αυτός είναι ο καλλίτερος δυνατός. Αν όμως το γράφημα είναι αραιό, τότε μπορούμε να χρησιμοποιήσουμε δύο άλλους αλγορίθμους που έχουν χρόνο $\mathcal{O}(|E|\log n)$. (Θα μπορούσαμε να πετύχουμε το χρόνο αυτό με τον αλγόριθμο του Prim;)

Παρατήρηση: Το $\mathcal{O}(\log n)$ είναι το ίδιο με το $\mathcal{O}(\log |E|)$.

Στον αλγόριθμο του Prim αρχίζουμε από ένα κόμβο και προσπαθούμε να επεκτείνουμε το δένδρο. Σε κάθε ενδιάμεσο στάδιο του αλγορίθμου έχουμε κατασκευάσει ένα συνδεδεμένο υποδένδρο. Στον επόμενο αλγόριθμο που θα περιγράψουμε (τον αλγόριθμο του Kruskal) σε ένα ενδιάμεσο στάδιο έχουμε κατασκευάσει ένα δάσος από δένδρα και σε κάθε βήμα συνδέουμε δύο δένδρα.

Αλγόριθμος 7.2.3: Ελάχιστο επικαλυπτικό δένδρο (Kruskal).

Είσοδος: Γράφημα $G = (V, E)$ με βάρη στις ακμές.

Εξοδος: Οι ακμές σε ένα ελάχιστο δένδρο.

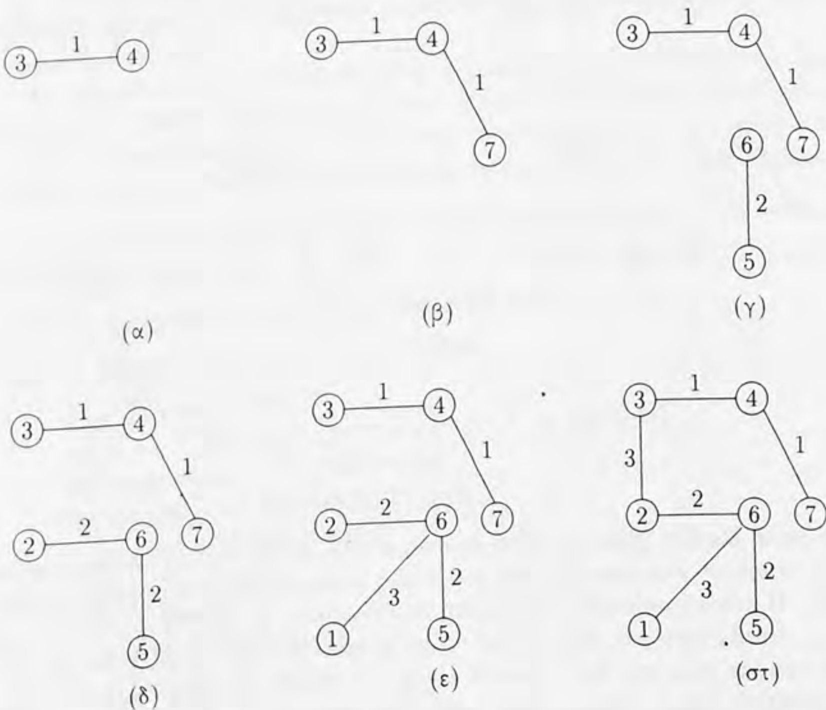
```
sort όλες τις ακμές κατα αύξουσα σειρά βάρους;
T ← ∅;
έστω [x, y] η επόμενη ακμή στη σειρά;
if comp[x] ≠ comp[y] then begin
   T ← T ∪ {[x, y]};
   merge(comp[x], comp[y])
end.
```

Ο χρόνος του αλγορίθμου αυτού εξαρτάται από τη δομή που χρησιμοποιείται για να βρίσκουμε σε ποιά συνιστώσα ανήκει κάθε κόμβος. Χρειαζόμαστε δηλαδή μια δομή με τις εξής δυνατότητες: (α) Να *βρίσκει* τη συνιστώσα κάθε κόμβου και (β) να *ενώνει* δύο συνιστώσες (κατά την εντολή merge(comp[x], comp[y])). Οι δομές αυτές ονομάζονται union–find και υλοποιούνται πολύ εύκολα αν τα σύνολα είναι ξένα μεταξύ τους (όπως στην περίπτωση μας, όπου κάθε κόμβος δεν μπορεί να ανήκει σε παραπάνω από μία συνιστώσες).

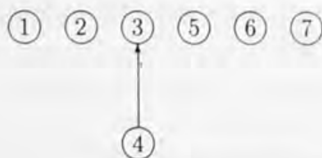
Η πιο συνηθισμένη υλοποίηση των δομών union–find είναι με ένα δένδρα προς τα πάνω (up trees). Στην υλοποίηση αυτή κάθε σύνολο είναι είναι ένα δένδρο, στο οποίο κάθε κόμβος δείχνει προς τον πατέρα του. (Το δένδρο αυτό

ονομάζεται δένδρο προς τα πάνω (up tree).) Η ταυτότητα κάθε συνόλου είναι η ρίζα του δένδρου (βλέπε και το Σχήμα (7.11), όπου παριστάνονται ως δένδρα προς τα πάνω ολα τα στάδια του Σχήματος (7.10)).

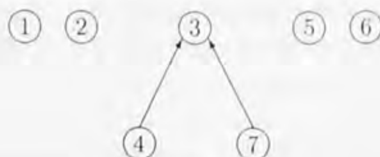
Παράδειγμα: Στο Σχήμα (7.10) εικονίζονται όλα τα βήματα που κάνει ο Αλγόριθμος 7.2.3 για να κατασκευάσει το ελάχιστο δένδρο του γραφήματος του Σχήματος (7.5).



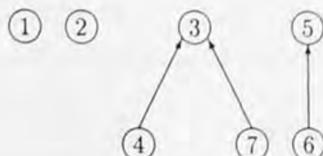
Σχήμα (7.10)



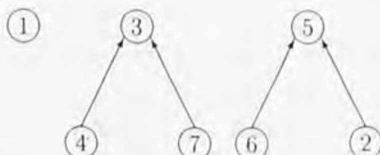
(α)



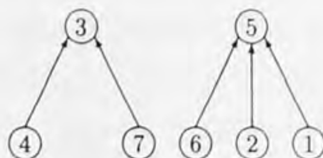
(β)



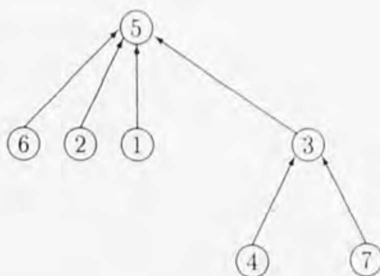
(γ)



(δ)



(ε)



(στ)

Σχήμα (7.11)

Η εντολή find(x) (βρες σε ποιό σύνολο ανήκει ο κόμβος x) υλοποιείται ως εξής: ακολουθούμε τα τόξα από τον x έως ότου φτάσουμε στη ρίζα του δένδρου. Η εντολή union(X, Y) (ένωση των συνόλων X και Y) υλοποιείται ως εξής: διαλέγουμε ένα από τα δύο σύνολα και κάνουμε τη ρίζα του να δείχνει προς τη ρίζα του άλλου (βλέπε Σχήμα (7.11στ)). Η union λοιπόν γίνεται σε σταθερό χρόνο. Πόσο χρόνο απαιτεί η find; Προφανώς όσο ο ύψος του δένδρου. Αν δεν προσέξουμε πώς κάνουμε τη union, το ύψος του δένδρου μπορεί και να γίνει $\mathcal{O}(n)$. (Σκεφτείτε ένα παράδειγμα όπου μπορεί να συμβεί αυτό.) Αν όμως χρησιμοποιήσουμε τον εξής ευριστικό κανόνα στη υλοποίηση

του union: κάθε ρίζα θυμάται τον αριθμό των κόμβων που περιέχει το σύνολό της και κάθε φορά κρεμάμε το μικρότερο σύνολο από το μεγαλύτερο, τότε το ύψος του δένδρου είναι $\mathcal{O}(\log n)$ (αποδείξτε το), οπότε και η find γίνεται σε χρόνο $\mathcal{O}(\log n)$. Αν χρησιμοποιήσουμε και ένα δεύτερο ευριστικό κανόνα, ο χρόνος μειώνεται σημαντικά. Ο κανόνας αυτός ονομάζεται συμπίεση μονοπατιών (path compression) και είναι ο εξής: όταν εκτελείται η find(x), όλοι οι κόμβοι που βρίσκονται στο μονοπάτι που ακολουθούμε από το x ως τη ρίζα του δένδρου συνδέονται κατ'ευθείαν με τη ρίζα του δένδρου (αποκτούν ως πατέρα τους τη ρίζα του δένδρου). Στην περίπτωση αυτή αποδεικνύεται ότι ο χρόνος που απαιτείται για να εκτελεστούν m λειτουργίες union–find σε ένα σύνολο με n στοιχεία είναι $O(m\alpha(m,n))$. Η συνάρτηση $\alpha(m,n)$ ονομάζεται συνάρτηση του Ackerman και αυξάνεται ιδιαίτερα αργά, δηλαδή έχει την τιμή 4 για κάθε πρακτική τιμή των m και n (κάθε τιμή μικρότερη απο 10^{100}). Η απόδειξη βρίσκεται στην αναφορά [3] και δεν περιλαμβάνεται στους σκοπούς του μαθήματος αυτού.

Ο αλγόριθμος που ακολουθεί είναι ο αλγόριθμος του Kruskal με δομές union–find. Εκτός από τις συναρτήσεις union και find χρησιμοποιούνται και οι εξής συναρτήσεις: η makeset(x), που κατασκευάζει σύνολο με μοναδικό κόμβο τον x και η deletemin, που παίρνει τον ελάχιστο όρο ουράς αναμονής.

Αλγόριθμος 7.2.4: Ελάχιστο επικαλυπτικό δένδρο (Kruskal). Υλοποίηση με δομές Union–Find.

Είσοδος: Γράφημα $G = (V, E)$ με βάρη στις ακμές.

Εξοδος: Οι ακμές σε ένα ελάχιστο δένδρο.

```
T ← ∅; comp ← |V|;
edjes ← φτιάξε ουρά προτεραιότητας με τις ακμές;
for each κόμβο x do makeset(x);
while comp > 1 do begin
  [x, y] ← deletemin(edges);
  X ← find(x);
  Y ← find(y);
  if X ≠ Y then begin
    T ← T ∪ {[x, y]};
    union(X, Y)
    comp ← comp − 1
    end
```

ΧΡΟΝΟΣ: Για την αρχικοποίηση του αλγορίθμου απαιτείται χρόνος $\mathcal{O}(n)$ για να κατασκευάσουμε τα αρχικά σύνολα με τη makeset και $\mathcal{O}(|E|)$ για να φτιάξουμε ουρά αναμονής με τις ακμές. Ο βρόγχος **while** του προγράμματος εκτελείται $|E|$ φορές (κάθε φορά σβήνει μια ακμή από την ουρά). Για κάθε επανάληψη του βρόγχου απαιτείται χρόνος:

$\mathcal{O}(\log|E|)$ για να βρούμε τη μικρότερη ακμή και

$\mathcal{O}(\log n)$ για να κάνουμε τα find, αν δεν χρησιμοποιηθεί και ο ευριστικός κανόνας συμπίεσης μονοπατιού.

Αρα, ο χρόνος είναι $\mathcal{O}(|E|\log n)$.
Στην περίπτωση που χρησιμοποιηθεί και ο ευριστικός κανόνας συμπίεση μονοπατιών των δομών union-find, ο χρόνος γίνεται περίπου ανάλογος του αριθμού των ακμών, δηλαδή $O(|E|\alpha(|E|,|V|))$.

Στον αλγόριθμο του Kruskal συνθέτουμε το ελάχιστο δένδρο από ένα δάσος από δένδρα και σε κάθε επανάληψη του κυρίου βρόγχου ο αριθμός των συνεκτικών συνιστωσών μειώνεται το πολύ κατά 1. Στον αλγόριθμο που ακολουθεί σε κάθε επανάληψη του κυρίου βρόγχου ο αριθμός των συνιστωσών μένει ο μισός.

Αλγόριθμος 7.2.5: Ελάχιστο επικαλυπτικό δένδρο (ένας ακόμα αλγόριθμος).
Είσοδος: Γράφημα $G = (V, E)$, στο οποίο κάθε ακμή έχει διαφορετικό βάρος.
Εξοδος: Οι ακμές σε ένα ελάχιστο δένδρο.

```
    T ← ∅;
    while το T δεν είναι συνεκτικό do begin
1       βρές τις συνεκτικές συνιστώσες του T;
2       για κάθε συν. συνιστώσα βρες τη μικρότερη ακμή που βγαίνει;
3       πρόσθεσε όλες αυτές τις ακμές στο T
    end
```

ΧΡΟΝΟΣ: Σε κάθε επανάληψη του **while** εκτελούνται οι γραμμές 1, 2 και 3 του αλγορίθμου.

Για την εκτέλεση της γραμμής 1 απαιτείται χρόνος $\mathcal{O}(n)$ με εξερεύνηση σε βάθος. Στο βήμα αυτό δημιουργείται ο πίνακας comp$[x]$ κατά τον Αλγόριθμο 2.1.2 (Γιατί ο χρόνος είναι $\mathcal{O}(n)$ και όχι $\mathcal{O}(|E|)$;)

Για να εκτελέσουμε αποδοτικά τη γραμμή 2 χρειαζόμαστε μια δομή. Για κάθε συνεκτική συνιστώσα X:

dist$[X]$ είναι η μικρότερη απόσταση κόμβου από τη συνιστώσα αυτή.

nhbr$[X]$ είναι ο κόμβος με τη μικρότερη απόσταση.

Ως συνήθως, με $\mathrm{d}(x,y)$ συμβολίζουμε το βάρος της ακμής $[x,y]$. Ο πίνακας dist παίρνει νέες τιμές για *κάθε* επανάληψη του **while** με το τμήμα κώδικα που ακολουθεί.

Τμήμα κώδικα που υλοποιεί τη γραμμή 2 του Αλγορίθμου 7.2.5.

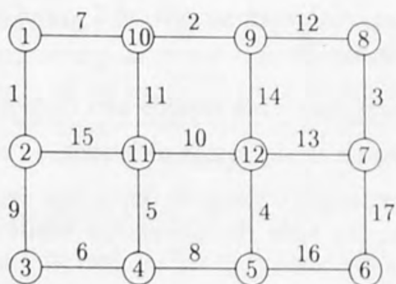
Είσοδος: Γράφημα $G=(V,E)$. Για κάθε $x \in V$, comp$[x]$ είναι η συνεκτική συνιστώσα του T, στην οποία ανήκει ο x, όπως παράγεται από τη γραμμή 1 του Αλγορίθμου 7.2.5.

```
for each συνεκτική συνιστώσα X του T dist[X] ← bignum;
for each ακμή [x, y] do
  if comp[x] ≠ comp[y] then begin
    if d(x, y) < dist[comp[x]] then begin
      dist[comp[x]] ← d(x, y);
      nhbr[comp[x]] ← y
    end;
    if d(x, y) < dist[comp[y]] then begin
      dist[comp[y]] ← d(x, y);
      nhbr[comp[y]] ← x
    end
  end.
```

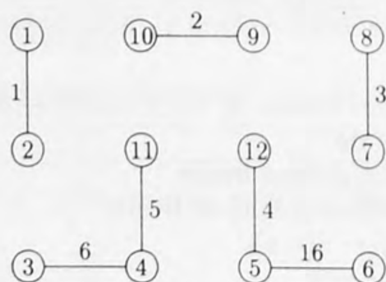
Για να δώσουμε λοιπόν τιμές στον πίνακα dist χρειαζόμαστε χρόνο $\mathcal{O}(|E|)$, άρα το βήμα 2 απαιτεί χρόνο $\mathcal{O}(|E|)$.

Συνεπώς, για κάθε επανάληψη του **while** χρειαζόμαστε χρόνο $\mathcal{O}(|E|)$. Πόσες επαναλήψεις θα γίνουν; Σε κάθε επανάληψη ο αριθμός των συνεκτικών συνιστωσών υποδιπλασιάζεται, άρα θα γίνουν $\log n$ επαναλήψεις. Ο χρόνος συνεπώς του Αλγορίθμου 7.2.5 είναι $\mathcal{O}(|E|\log n)$.

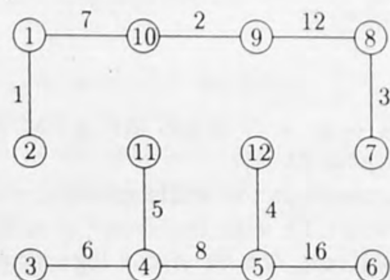
Παράδειγμα: Στο Σχήμα (7.12) παρουσιάζονται οι φάσεις κατασκευής του ελάχιστου δένδρου από τον Αλγόριθμο 7.2.5.



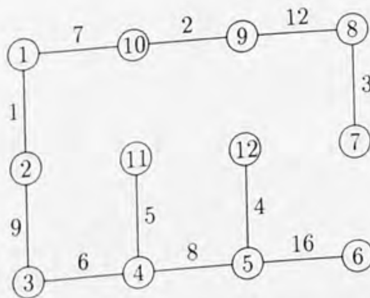
(7.12α) Αρχικό γράφημα.



(7.12β) T μετά την πρώτη επανάληψη του **while**

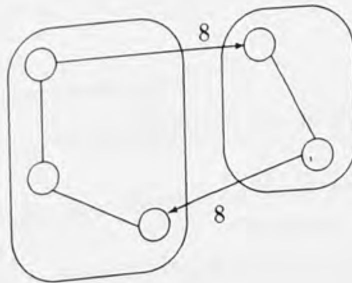


(7.12γ) T μετά την δεύτερη επανάληψη του **while**.

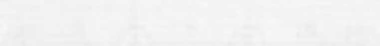


(7.12δ) T μετά την τρίτη επανάληψη του **while**.

Σημείωση: Για τον Αλγόριθμο 7.2.5 απαιτείται τα μήκη των ακμών να είναι **όλα διαφορετικά** μεταξύ τους. Γιατί αλλιώς μπορεί να προκύψει το πρόβλημα που φαίνεται στό Σχήμα (7.13). Μπορούμε να κάνουμε όλα τα μήκη των ακμών διαφορετικά πολυ εύκολα αν, πριν την εκτέλεση του αλγορίθμου, προσθέσουμε ένα μικρό τυχαίο αριθμό σε όλες τις ακμές.



Σχήμα (7.13)



Κεφάλαιο 8

ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

8.1. Εισαγωγή

Εχετε ήδη μελετήσει την τεχνική του "διαίρει και βασίλευε" στη σχεδίαση αλγορίθμων που είναι η εξής:

- διαίρεσε το πρόβλημα σε δύο συνήθως υποπροβλήματα που λύνουν το αρχικό,
- λύσε τα υποπροβλήματα αναδρομικά,
- συνδίασε τις λύσεις των υποπροβλημάτων για να λύσεις το συνολικό.

Δυστυχώς η τεχνική αυτή δεν είναι δυνατόν να εφαρμοστεί στα περισσότερα προβλήματα. Μια άλλη τεχνική, που μπορεί να εφαρμοστεί σε πολλά προβλήματα είναι να λύσουμε *όλα τα δυνατά υποπροβλήματα* με αύξουσα σειρά μεγέθους χρησιμοποιώντας τη λύση των μικρότερων υποπροβλημάτων για να λύσουμε τα αμέσως μεγαλύτερα, μέχρι να φτάσουμε στο συνολικό πρόβλημα. Η τεχνική αυτή ονομάζεται **δυναμικός προγραμματισμός** (dynamic programming). Πως ορίζεται όμως το υποπρόβλημα; Η έννοια του υποπροβλήματος θα γίνει σαφής αν δούμε πολλά παραδείγματα.

8.2. Μεταβατική κλειστότητα γραφήματος

Η μεταβατική κλειστότητα ενός γραφήματος με κατεύθυνση $G = (V, E)$ είναι ένα άλλο γράφημα $G' = (V, E')$ που έχει τους ίδιους κόμβους με το G, αλλά έχει υπερσύνολο των ακμών του G, δηλαδή $E \subseteq E'$. Μια ακμή (x, y) περιλαμβάνεται στο σύνολο E' αν υπάρχει μονοπάτι από τον x στον y στον G.

Μια δυαδική σχέση R μπορεί να θεωρηθεί ως γράφημα με κατεύθυνση $G = (V, E)$ (οι κόμβοι V είναι τα στοιχεία του συνόλου P, πάνω στο οποίο ορίζεται η σχέση, και η ακμή $(x, y) \in E$ ανν xRy). Η μεταβατική κλειστότητα του γραφήματος G είναι τότε η μικρότερη σχέση που είναι υπερσύνολο της R και είναι μεταβατική. Για παράδειγμα, η σχέση R:

$$\{(a, b) : a, b \in P \text{ και ο } a \text{ είναι πρόγονος του } b\}$$

Αν ο a είναι πρόγονος του b και ο b είναι πρόγονος του c, τότε και ο a είναι πρόγονος του c, δηλαδή το $(a, c) \in R$. Αρα, στην περίπτωση αυτή η μεταβατική κλειστότητα της R είναι ίδια με την σχέση R και η σχέση R ονομάζεται μεταβατική.

Το πρόβλημα της μεταβατικής κλειστότητας γραφήματος έχει πολλές εφαρμογές: Για παράδειγμα, έστω υπολογιστικό συστημα, στο οποίο έχουν λογαριασμό διάφοροι χρήστες. Οι χρήστες για να συνεργαστούν μπορεί να δίνουν άδεια χρήσης του κωδικού τους σε άλλους χρήστες. Μια ακμή από τον A στον B συμβολίζει ότι ο κωδικός A έχει άδεια χρήσης του κωδικού B. Αλλά, αν ο A έχει άδεια χρήσης του B και ο B του C, τότε ο A μπορεί να χρησιμοποιήσει τον C. Μια εφαρμογή της μεταβατικής κλειστότητας είναι να βρούμε για κάθε χρήστη όλους τους άλλους που έχουν άδεια (άμεση ή έμμεση) να χρησιμοποιήσουν το λογαριασμό του.

Σκοπός μας λοιπόν είναι: δεδομένης της μήτρας γειτνίασης του αρχικού γραφήματος G, να κατασκευάσουμε μια νέα μήτρα T τέτοια που για $1 \leq i, j \leq n$

$$T_{ij} = \begin{cases} 1, & \text{αν υπάρχει μονοπάτι } i \to j \text{ στον } G; \\ 0, & \text{αλλιώς.} \end{cases}$$

Είναι μια εφαρμογή του δυναμικού προγραμματισμού. Οπως πάντα, το δύσκολο είναι να ορίσουμε έξυπνα τι είναι "υποπρόβλημα". Ενας πρώτος τρόπος να αντιμετωπισουμε το προβλημα είναι ο εξής αλγόριθμος:

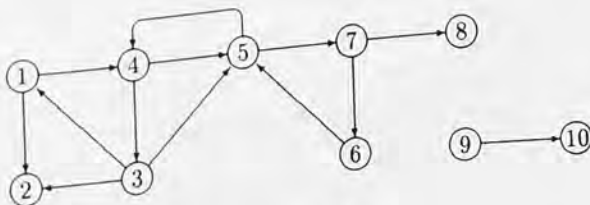
Το αρχικό σύνολο ακμών T της μεταβατικής κλειστότητας είναι οι ακμές του γραφήματος εισόδου.

Επανάλαβε το εξής για κάθε ζεύγος ακμών του T:
 αν $(a,b) \in T, (b,c) \in T$ αλλά $(a,c) \notin T$, τότε πρόσθεσε το (a,c) στο T.
 έως οτου να μην προστίθενται πλέον νέες ακμές.

Ο αλγόριθμος αυτός εκτελείται σε χρόνο $\mathcal{O}(n^5)$. Ο λόγος είναι ότι, για κάθε νέα ακμή που προστίθεται, θα πρέπει να ελεγχθούν ξανά όλες οι ακμές. Αν όμως κάναμε μια αναδιάταξη του προβλήματος και σε κάθε βήμα αντί να ελέγχουμε όλους τους δυνατούς ενδιάμεσους κόμβους b, ελέγχαμε ένα συγκεκριμένο αριθμό από ενδιάμεσους (ο αριθμός αυτός των ενδιάμεσων θα αυξάνεται σε καθε βήμα κατά 1), τότε δεν χρειάζεται δεύτερος έλεγχος όταν προσθέσουμε νέα ακμή. Η έξυπνη επιλογή υποπροβλήματος για τη μεταβατική κλειστότητα βασίζεται στην ιδέα αυτή.

Το υποπρόβλημα ορίζεται ως εξής:

$T_{i\ j\ k} \equiv$ υπάρχει μονοπάτι $i \to j$ που όλοι οι ενδιάμεσοι κόμβοι του είναι $\leq k$.



Σχήμα (8.1)

Παράδειγμα: Στο γράφημα του σχήματος (8.1) ισχύει

$T_{1\ 8}$, $T_{1\ 6}$, $T_{6\ 1}$, $\neg T_{9\ 7}$, $\neg T_{8\ 5}$, $T_{2\ 2}$,
$T_{1\ 4}$, $T_{1\ 3\ 4}$, $\neg T_{1\ 3\ 2}$, $T_{1\ 6\ 7}$, $\neg T_{1\ 6\ 5}$ κ.λ.π.

Η λύση του προβλήματος ορίζεται αναδρομικά ως εξής:

$$T_{i\ j\ 0} \equiv (i,j) \in E \vee i = j \quad (8.1)$$

$$T_{i\ j\ k+1} = T_{i\ j\ k} \vee (T_{i\ k+1\ k} \wedge T_{k+1\ j\ k}) \quad (8.2)$$

Η σχέση (1) λέει ότι υπάρχει μονοπάτι $i \to j$ δίχως να χρησιμοποιηθεί κανένας ενδιάμεσος κόμβος αν υπάρχει ακμή (i, j) ή αν τα i και j ταυτίζονται. Η σχέση (2) λέει ότι υπάρχει μονοπάτι $i \to j$ που όλοι οι ενδιάμεσοι κόμβοι να είναι $\leq k+1$, (α) αν υπάρχει μονοπάτι που όλοι οι ενδιάμεσοι να είναι $\leq k$ ή (β) αν υπάρχει τρόπος να πάμε από τον i στον $k+1$ και από τον $k+1$ στον j χρησιμοποιώντας ως ενδιάμεσους μόνο κόμβους του συνόλου $\{1, \ldots, k\}$.

Το $T_{i,j}$ είναι εξ ορισμού ίσο με το $T_{i,j,n}$

Οι σχέσεις (1) και (2) υλοποιούνται με ένα πρόγραμμα που έχει τρείς φωλιασμένους βρόγχους.

Αλγόριθμος 8.1.1: Μεταβατική κλειστότητα.

Είσοδος: Γράφημα με κατεύθυνση $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$ που παριστάνεται με μήτρα γειτνίασης.

Εξοδος: Μήτρα T που είναι η μεταβατική κλειστότητα του G.

```
for all i, j  T[i, j] ←false;
for each ακμή (i, j)  T[i, j] ←true;
for each κόμβο i  T[i, i] ←true;
for k := 1 to n do
   for i := 1 to n do
      for j := 1 to n do
         if T[i, k] and T[k, j] then T[i, j] ← true.
```

Παράδειγμα: Η έξοδος του αλγορίθμου για το γράφημα του σχήματος (8.1) παρουσιάζεται στον πίνακα 8.1. Η τιμή του k, κατά την οποία κάθε στοιχείο έγινε 1, παρουσιάζεται ως εκθέτης του στοιχείου.

$$\begin{array}{cccccccccc}
1 & 1 & 1^4 & 1 & 1^4 & 1^7 & 1^5 & 1^7 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1^1 & 1 & 1^7 & 1^5 & 1^7 & 0 & 0 \\
1^3 & 1^3 & 1 & 1 & 1 & 1^7 & 1^5 & 1^7 & 0 & 0 \\
1^4 & 1^4 & 1^4 & 1 & 1 & 1^7 & 1 & 1^7 & 0 & 0 \\
1^5 & 1^5 & 1^5 & 1^5 & 1 & 1 & 1^5 & 1^7 & 0 & 0 \\
1^6 & 1^6 & 1^6 & 1^6 & 1^6 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

Πίνακας 8.1

ΧΡΟΝΟΣ: Ο χρόνος του αλγορίθμου είναι προφανώς $\mathcal{O}(n^3)$. Εχουμε τρείς φωλιασμένους βρόγχους, κάθε ένας εκ των οποίων εκτελείται n φορές.

8.3. Ελάχιστα μονοπάτια μεταξύ κάθε ζεύγους σημείων

Αν στην αναδρομική σχέση που υπολογίζει τη μεταβατική κλειστότητα αντικαταστήσουμε το $\wedge$ με $+$ και το $\vee$ με min, προκύπτει δυναμικός αλγόριθμος που υπολογίζει τα ελάχιστα μονοπάτια ανάμεσα σε κάθε ζεύγος σημείων (all pairs shortest paths). Σκοπός του αλγορίθμου είναι, δεδομένου γραφήματος G με κατεύθυνση και βάρη στις ακμές, να παράγει ένα πίνακα T, όπου για $1 \leq i, j \leq n$

$$T_{ij} = \begin{cases} \text{mindist}(i,j), & \text{η ελάχιστη απόσταση μεταξύ } i \text{ και } j \text{ στον } G \\ \infty, & \text{αν δεν υπάρχει μονοπάτι } i \rightarrow j \text{ στον } G. \end{cases}$$

Το "υποπρόβλημα" ορίζεται όπως και στην προηγούμενη ενότητα:

$T_{i\,j\,k} \equiv$ η ελάχιστη απόσταση μεταξύ i και j χρησιμοποιώντας ως ενδιάμεσους κόμβους μόνο κόμβους με αριθμό $\leq k$.

Η αρχικοποίηση του πίνακα γίναται στην περίπτωση αυτή ως εξής:

$$T_{i\,j\,0} \equiv \begin{cases} 0, & \text{αν } i = j \\ \text{d}(i,j), & \text{αν } (i,j) \in E \\ \infty, & \text{αλλιώς.} \end{cases}$$

Η σχέση που δίνει το $T_{i\ j\ k+1}$ συναρτήσει του $T_{i\ j\ k}$ γίνεται:

$$T_{i\ j\ k+1} = \min(T_{i\ j\ k},\quad (T_{i\ k+1\ k} + T_{k+1\ j\ k}))$$

Η υλοποίηση είναι και πάλι ένα πρόγραμμα με τρεις φωλιασμένους βρόγχους.

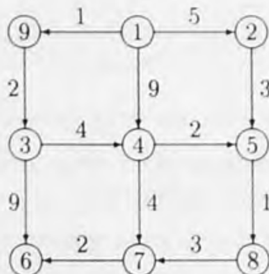
Αλγόριθμος 8.2.1: Ελάχιστα μονοπάτια μεταξύ κάθε ζεύγους κόμβων.

Είσοδος: Γράφημα με κατεύθυνση $G = (V, E)$ με $V = \{1, 2, \ldots, n\}$ που παριστάνεται με μήτρα γειτνίασης. (Τα βάρη των ακμών μπορεί να είναι και αρνητικά.)

Εξοδος: Μήτρα T με τιμές τα ελάχιστα μονοπάτια μεταξύ κάθε ζεύγους κόμβων.

```
for all i, j   T[i, j] ←bignum;
for each ακμή (i, j)   T[i, j] ← d(i, j);
for each κόμβο i   T[i, i] ← 0;
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      if T[i, k] + T[k, j] < T[i, j] then
        T[i, j] ← T[i, k] + T[k, j].
```

Παράδειγμα: Η έξοδος του αλγορίθμου για το γράφημα του σχήματος (8.2) παρουσιάζεται στον πίνακα 8.2. Η τιμή του k, κατά την οποία κάθε στοιχείο έγινε 1, παρουσιάζεται ως εκθέτης του στοιχείου.



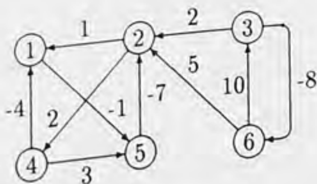
Σχήμα (8.2)

0	5	3	$7^{(9)}$	$8^{(2)}$	12	$11^{(9)}$	$9^{(5)}$	1
∞	0	5	∞	∞	∞	7	$4^{(5)}$	∞
∞	∞	0	4	$6^{(4)}$	9	$8^{(4)}$	$7^{(5)}$	∞
∞	∞	∞	0	2	$6^{(7)}$	4	$3^{(5)}$	∞
∞	∞	∞	∞	0	∞	4	1	∞
∞	∞	∞	∞	∞	0	∞	∞	∞
∞	∞	∞	∞	∞	2	0	∞	∞
∞	∞	∞	∞	∞	5	3	0	∞
∞	∞	2	$6^{(3)}$	$8^{(4)}$	$11^{(3)}$	$10^{(4)}$	∞	0

Πίνακας 8.2

ΧΡΟΝΟΣ: Ο χρόνος του αλγορίθμου είναι $\mathcal{O}(n^3)$. Το πρόβλημα των ελάχιστων μονοπατιών μεταξύ κάθε ζεύγους κόμβων θα μπορούσε να λυθεί και αν εκτελούσαμε τον αλγόριθμο του Dijkstra για κάθε κόμβο του γραφήματος. Ο χρόνος θα ήταν τότε $\mathcal{O}(n|E|\log n)$, που είναι προτιμότερος μόνο στην περίπτωση που το γράφημα ειναι αραιό.

Σημείωση: Ο αλγόριθμος δουλεύει ακόμα και στην περίπτωση που έχουμε αρνητικά βάρη, αρκεί να μην έχουμε αρνητικούς κύκλους. Αν υπάρχουν αρνητικοί κύκλοι, ο αλγόριθμος δεν είναι αξιόπιστος, αλλά μας προειδοποιεί, επειδή στο τέλος $T_{i\,i\,n} < 0$ για κάποιο i. Για παράδειγμα, στο γράφημα του σχήματος (8.3) υπάρχουν αρνητικοί κύκλοι και ο αλγόριθμος; ήδη από το βήμα $k = 2$, υπολογίζει $T_{5\,5\,2} = -7$.



Σχήμα (8.3)

8.4. Το πρόβλημα του περιοδεύοντος πωλητή

Οι δυναμικοί αλγόριθμοι που εξετάσαμε ως τώρα είχαν το εξής κοινό χαρακτηριστικό: Μπορούσαμε να τους περιγράψουμε με ένα πρόγραμμα, το οποίο είχε έναν αριθμό από φωλιασμένα **for** (βλέπε αλγορίθμους 8.1.1 και 8.2.1) και η πολυπλοκότητα τους ήταν πολυωνυμική. Ο εκθέτης του n ήταν όσο το μέγιστο βάθος φωλιάσματος των **for**. Είναι αυτό χαρακτηριστικό όλων των δυναμικών αλγορίθμων; Με το παράδειγμα που ακολουθεί θα δούμε ότι αυτό δεν ισχύει.

Το πρόβλημα που θα εξετάσουμε ονομάζεται Πρόβλημα του Περιοδεύοντος Πωλητή (Traveling Salesman Problem, TSP). Το προβλήμα αυτό θυμίζει το πρόβλημα του ελάχιστου δένδρου, αλλά έχει *βασικές* διαφορές από αυτό, όπως θα εξηγήσουμε στο επόμενο κεφάλαιο. Εχουμε ως είσοδο ένα γράφημα με βάρη $G = (V, E)$, όπου $V = \{1, 2, \ldots, n\}$. Μπορείτε να φανταστείτε ότι οι κόμβοι του γραφήματος είναι ένα σύνολο από πόλεις και τα βάρη στις ακμές οι χιλιομετρικές αποστάσεις μεταξύ τους. Θέλουμε να βρούμε τη συντομότερη διαδρομή, με την οποία γίνεται επίσκεψη σε όλες τις πόλεις και επιστροφή στην αφετηρία.

Ο πιό απλός τρόπος να λύσουμε το προβλημα είναι να εξετάσουμε μια μια όλες τις δυνατές διαδρομές, να συγκρίνουμε τα μήκη τους και να επιλέξουμε τη μικρότερη. Αυτό όμως απαιτεί χρόνο $\mathcal{O}(n!)$, όσες είναι οι δυνατές μεταθέσεις (permutations) των n πόλεων.

Ποιό είναι το σωστό υποπρόβλημα στην περίπτωση αυτή; Το ορίζουμε ως εξής:

Εστω C ένα σύνολο από πόλεις ($C \subseteq \{1, 2, \ldots, n\}$) που περιλαμβάνει την αφετηρία. (Θεωρούμε αυθαίρετα ότι ο κόμβος 1 είναι η αφετηρία.) Ορίζουμε το εξής υποπρόβλημα: Για κάθε πόλη c του συνόλου C εκτός της αφετηρίας,

cost[C,c] = το μήκος της συντομότερης διαδρομής, η οποία ξεκινά από την πόλη 1,
περνάει από *όλες* τις πόλεις του συνόλου C
και καταλήγει στην πόλη c.

Η ιδέα του αλγορίθμου είναι η εξής. Το $\text{cost}(C, c)$ είναι γνωστό, όταν το C περιλαμβάνει ένα μόνο κόμβο εκτός από τον 1: Είναι το βάρος της ακμής $(1, c)$, το οποίο συμβολιζουμε ως συνήθως με $d(1, c)$. Αν βρούμε μια σχέση, με την οποία να αυξάνεται ο αριθμός στοιχείων του C κατά 1, τότε μπορούμε να

λύσουμε αναδρομικά το πρόβλημα (αυξάνουμε τον αριθμό των στοιχείων του C έως ότου γίνει n και μετά υπολογίζουμε εύκολα την ελάχιστη διαδρομή που καταλήγει στην αφετηρία). Η αναδρομική σχέση υπολογισμού του $\text{cost}[C, c]$, είναι η εξής:

$$\text{cost}[C, c] = \min_{i \in C, i \neq 1, c} (\text{cost}[C - \{c\}, i] + d(i, c))$$

Στη βέλτιστη διαδρομή από την 1 στην c ξεκινάμε από την 1, περνάμε από ολες τις πόλεις του C εκτός από τις c και i, περνάμε από την i και καταλήγουμε στην c. Η i είναι δηλαδή η τελευταία πόλη που επισκεπτόμαστε πριν από τη c. Για να υπολογίσουμε την ελάχιστη διαδρομή από 1 στο c, πρέπει να βάλουμε στη θέση του i (δηλαδή στη θέση της πόλης που επισκεπτόμαστε ακριβώς πριν την c) όλες τις πόλεις του C, (εκτός βεβαίως από την αφετηρία και το c) και να εξετάσουμε ποιά από αυτές δίνει μικρότερη διαδρομή. Αυτό είναι το νόημα του min που παρουσιάζεται έξω από την περένθεση. Ενοείται ότι τη στιγμή που γίνεται ο υπολογισμός αυτός έχουμε ήδη υπολογίσει την συντομότερη διαδρομή $\text{cost}[C - \{c\}, i]$ για όλες τις πόλεις i.

Οπως και στα προηγούμενα παραδείγματα, η αναδρομική σχέση γίνεται ένας αλγόριθμος που έχει έναν αριθμό από φωλιασμένα **for**.

> **for** $i \leftarrow 2$ to n **do**
> $\text{cost}[\{1, i\}, i] \leftarrow d(1, i)$
> **for** $k \leftarrow 3, 4, \ldots, n$ **do**
>   **for each** $C \subseteq \{1, 2, \ldots, n\}$ με $|C| = k$ και $1 \in C$ **do**
>   % για κάθε υποσύνολο των πόλεων που έχει k στοιχεία
>   % και περιλαμβάνει την 1
>   **for each** $c \in C, c \neq 1$ **do**
> $\text{cost}[C, c] = \min_{i \in C, i \neq 1, c}(\text{cost}[C - \{c\}, i] + d(i, c))$
> **βέλτιστο–κόστος** $\leftarrow \min_{c \in C, c \neq 1}(\text{cost}[\{1, 2, \ldots, n\}, c] + d(c, 1))$

ΧΡΟΝΟΣ: Ο χρόνος του αλγορίθμου είναι $\mathcal{O}(n^2 2^n)$. Είναι ο πρώτος μη πολυωνυμικός αλγόριθμος που παρουσιάζουμε. Η ανάλυση είναι πιό πολύπλοκη από συνήθως. Εχουμε 4 φωλιασμένα **for**: (α) για κάθε μέγεθος υποσυνόλου, (β) για κάθε υποσύνολο του μεγέθους αυτού, (γ) για κάθε στοιχείο c του υποσυνόλου και (δ) για κάθε άλλο στοιχείο τού ίδιου υποσυνόλου εκτός του c. (Το τελευταίο **for** είναι ο υπολογισμός του min.) Εστω ότι εξετάζουμε το πρώτο μέγεθος υποσυνόλου $k = 3$. Πόσα διαφορετικά σύνολα μεγέθους 3

υπάρχουν; Εφόσον όλα τα σύνολα περιέχουν την αφετηρία, θα πρεπει να διαλέξουμε $k-1=2$ πόλεις από το σύνολο των πόλεων εκτός της αφετηρίας, δηλαδή $\binom{n-1}{k-1}$ διαφορετικά σύνολα. Για κάθε ένα από τα σύνολα αυτά επιλέγουμε ένα κόμβο c εκτός της αφετηρίας, δηλαδή $k-1=2$ επιλογές, και για κάθε c επιλέγουμε και ένα δεύτερο κόμβο, δηλαδή έχουμε $k-2=1$ επιλογές. Αρα, για κάθε μέγεθος υποσυνόλου k το σύνολο των βημάτων είναι:

$$\binom{n-1}{k-1}(k-1)(k-2)$$

Οπότε ο συνολικός απαιτούμενος χρόνος είναι το άθροισμα για όλα τα μεγέθη υποσυνόλων, δηλαδή:

$$\sum_{k=3}^{n}(k-1)(k-2)\binom{n-1}{k-1}$$

Αν κάνουμε τις πράξεις παίρνουμε το αποτέλεσμα που αναφέρθηκε στην αρχή της ενότητας.

Κεφάλαιο 9

NP–ΠΛΗΡΗ ΠΡΟΒΛΗΜΑΤΑ

9.1. Εισαγωγή

Στο Κεφάλαιο 1 αναφέρθηκε ότι οι αλγόριθμοι χωρίζονται σε δύο βασικές κατηγορίες: (α) τους πολυωνυμικούς, δηλαδή τους αλγορίθμους, η πολυπλοκότητα των οποίων είναι $\mathcal{O}(n^k) : n \geq 0$ και (β) τους εκθετικούς, δηλαδή τους αλγορίθμους, για τους οποίους η παραπάνω σχέση δεν ισχύει. Πολυωνυμικοί είναι σχεδόν όλοι οι αλγόριθμοι που περιλαμβάνονται στα προηγούμενα κεφάλαια: οι αλγόριθμοι ταξινόμησης, (Κεφάλαιο 3), οι αλγόριθμοι αναζήτησης συμβολοσειρών (Κεφάλαιο 4), οι στοιχειώδεις αλγόριθμοι γραφημάτων (Κεφάλαιο 5) κλπ. Οι αλγόριθμοι αυτοί χρησιμοποιούνται συχνά για την επίλυση προβλημάτων που παρουσιάζονται στην πράξη, επειδή απαιτείται "λογικό" χρονικό διάστημα για την εκτέλεσή τους. Τι εννοούμε λογικό χρονικό διάστημα; Γιατί έχουν τόση σημασία οι πολυωνυμικοί αλγόριθμοι; Ας εξετάσουμε έναν *μη* πολυωνυμικό αλγόριθμο με πολυπλοκότητα έστω 2^n: Για κάποια είσοδο μεγέθους n χρειάζεται χρόνο 2^n. Αν έχουμε, για παράδειγμα, μια είσοδο μεγέθους 60, υπάρχει περίπτωση να πρέπει να περιμένουμε τον αλγόριθμο να εκτελέσει 2^{60} βήματα πριν σταματήσει. (Αν ο υπολογιστής εκτελεί ένα βήμα το microsecond, θα πρέπει να περιμένουμε 366 αιώνες.) Είναι προφανές ότι οι πρόοδοι της τεχνολογίας δεν βοηθάνε στην περίπτωση αυτή. Ισχύει όμως πάντοτε ότι αρκεί να έχουμε έναν πολυωνυμικό αλγόριθμο και το πρόβλημα λύνεται γρήγορα; Θα μπορούσε να έχουμε έναν πολυωνυμικό αλγόριθμο της τάξης n^{80}. Βεβαίως, για αρκετά μεγάλες τιμές του n και αυτός ο αλγόριθμος (όπως όλοι οι πολυωνυμικοί) είναι γρηγορότερος από οποιονδήποτε εκθετικό,

αλλά θα μπορούσε κάποιος να επιχειρηματολογήσει ότι οι τιμές αυτές είναι πολύ μεγάλες και δεν παρουσιάζονται στα προβλήματα που έχουμε συνήθως προς επίλυση. Υπάρχουν δύο απαντήσεις στο ερώτημα αυτό. Η πρώτη είναι ότι τέτοιοι αλγόριθμοι δεν παρουσιάζονται στην πράξη. Οι πολυωνυμικοί αλγόριθμοι συνήθως είναι δυνατόν να βελτιωθούν, ώστε να έχουν μικρές σταθερές και εκθέτες. Η δεύτερη απάντηση είναι ότι ο διαχωρισμός των αλγορίθμων σε γρήγορους και όχι, ανάλογα με το αν είναι πολυωνυμικοί ή όχι, πράγματι έχει ορισμένα αντιπαραδείγματα, αλλά έχει σημαντικά πλεονεκτήματα: Οι πολυωνυμικοί αλγόριθμοι παραμένουν πολυωνυμικοί σε οποιδήποτε μοντέλο υπολογιστή, όταν προστίθενται ή πολλαπλασιάζονται παραμένουν πολυωνυμικοί κλπ.

Ο μόνοι μή πολυωνυμικοί αλγόριθμοι που παρουσιάστηκαν στις σημειώσεις αυτές είναι οι δύο αλγόριθμοι για τό πρόβλημα του περιοδεύοντος πωλητή (Ενότητα 8.4): Ο πρώτος ψάχνει εξαντλητικά όλες τις λύσεις και επιλέγει τη μικρότερη, οπότε απαιτεί χρόνο $\mathcal{O}(n!)$ και ο δεύτερος χρησιμοποιεί τεχνικές του δυναμικού προγραμματισμού και μειώνει την πολυπλοκότητα σε $\mathcal{O}(n^2 2^n)$. Είναι ο δεύτερος ο ταχύτερος δυνατός αλγόριθμος για το πρόβλημα αυτό, ή θα ήταν δυνατόν να κατασκευαστεί ένας πολυωνυμικός αλγόριθμος; Οι καλίτεροι θεωρητικοί της επιστήμης μας έχουν προσπαθήσει να βρούν πολυωνυμικό αλγόριθμο για το πρόβλημα του περιοδεύοντος πωλητή και στάθηκε αδύνατον. Είναι μήπως η εκθετική πολυπλοκότητα εγγενής στο πρόβλημα αυτό, μπορούμε δηλαδή να *αποδείξουμε* ότι ειναι αδύνατο να κατασκευαστεί πολυωνυμικός αλγόριθμος που να το λύνει. Δυστυχώς, μια τέτοια απόδειξη είναι εξίσου δύσκολη με την κατασκευή πολυωνυμικού αλγορίθμου στην περίπτωση αυτή.

Η θεωρία της $\mathcal{NP}$-πληρότητας παρέχει έναν τρόπο να αντιμετωπίζουμε προβλήματα που έχουν την ίδια μορφή δυσκολίας με το πρόβλημα του περιοδεύοντος πωλητή. Τα προβλήματα αυτά παρουσιάζονται πολύ συχνά στην πράξη και έχουν, οπως θα δούμε σε επόμενη ενότητα, ελάχιστες φαινομενικές διαφορές από προβλήματα, για τα οποία υπάρχει πολυωνυμικός αλγόριθμος. Η θεωρία της $\mathcal{NP}$-πληρότητας παρέχει ένα σύνολο από τεχνικές, με τις οποίες αποδεικνύουμε ότι ενα πρόβλημα είναι *εξίσου δύσκολο* με ένα σύνολο από ανάλογα προβλήματα που απασχολούν τους θεωρητικούς εδώ και πολύ καιρό. Αν μπορέσουμε να αποδείξουμε ότι το πρόβλημα που αντιμετωπίζουμε είναι εξίσου δύσκολο με το πρόβλημα του περιοδεύοντος πωλητή (ή με οποιοδήποτε άλλο από ένα πλήθος προβλημάτων που είναι ισοδύναμα μεταξύ τους), έχουμε τα εξής οφέλη: (α) δεν σπαταλάμε πλεόν χρόνο για την κατασκευή πολυωνυμικού αλγορίθμου και (β) μπορούμε να εφαρμόσουμε ένα σύνολο από γνωστές

τεχνικές, με τις οποίες λύνονται τα προβλήματα αυτά, όπως προσεγγιστικούς αλγορίθμους.

9.2. Προβλήματα απόφασης

Στα προηγούμενα κεφάλαια αντιμετωπίζαμε ένα πρόβλημα (πχ. το πρόβλημα του ελάχιστου δένδρου, ή το πρόβλημα του περιοδεύοντος πωλητή) ως ένα θέμα προς επίλυση. Στην θεωρία της υπολογιστικής πολυπλοκότητας (τμήμα της οποίας είναι η θεωρία της $\mathcal{NP}$-πληρότητας) ένα πρόβλημα είναι μαθηματικό αντικείμενο, του οποίου μελετάμε τις ιδιότητες. Ας υιοθετίσουμε ένα συμβολισμό, με τον οποίο περιγράφουμε ένα πρόβλημα όταν το εξετάζουμε από την σκοπία αυτή . Περιγράφουμε πρώτα το στιγμιότυπο (instance) του προβλήματος (δηλαδή την είσοδο ενος αλγορίθμου που θα έλυνε το πρόβλημα) και μετά το ζητούμενο (δηλαδή την έξοδο του αλγορίθμου). Παράδειγμα:

Πρόβλημα Περιοδεύοντος Πωλητή: (εκδοχή βελτιστοποίησης)
Στιγμιότυπο: Πεπερασμένο σύνολο από πόλεις $C = \{c_1, c_2, \ldots, c_m\}$ και για κάθε ζεύγος πόλεων $c_i, c_j \in C$ ένας θετικός ακέραιος που δηλώνει την απόσταση μεταξύ των πόλεων.
Ζητούμενο: Η ελάχιστη διαδρομή που περνάει από όλες τις πόλεις του C και καταλήγει στην αφετηρία.

Υπάρχει μια κατηγορία προβλημάτων που παίζουν βασικό ρόλο στην μελέτη της πολυπλοκότητας των προβλημάτων. Αυτά είναι τα **προβλήματα απόφασης** (decision problems), δηλαδή τα προβλήματα, τα οποία έχουν δύο δυνατές απαντήσεις “ναι” ή “όχι”. Στην περίπτωση αυτή το ζητούμενο είναι απάντηση σε ένα ερώτημα. Παραδείγματα προβλημάτων απόφασης είναι τα εξής:

Πρόβλημα Περιοδεύοντος Πωλητή: (εκδοχή απόφασης)
Στιγμιότυπο: Πεπερασμένο σύνολο από πόλεις $C = \{c_1, c_2, \ldots, c_m\}$, ένας θετικός ακέραιος $d(i, j)$ για κάθε ζεύγος πόλεων $c_i, c_j \in C$, ο οποίος δηλώνει την απόσταση μεταξύ των πόλεων και ένας θετικός ακέραιος B που είναι ο στόχος.
Ερώτημα: Υπάρχει διαδρομή που περνάει από όλες τις πόλεις του C, καταλήγει στην αφετηρία και έχει μήκος μικρότερο ή ίσο του B;

Μια βασική κατηγορία προβλημάτων απόφασης είναι εκείνα, τα οποία είναι δυνατόν να επιλυθούν με πολυωνυμικούς αλγορίθμους. Παραδείγματα προβλημάτων της κατηγορίας αυτής είναι:

Πρόβλημα Συνεκτικότητας:
Στιγμιότυπο: Γράφημα $G = (V, E)$ και δύο διακεκριμένοι κόμβοι x και y.
Ερώτημα: Υπάρχει μονοπάτι από τον x στον y;

Πώς μπορούμε να απαντήσουμε στο ερώτημα αυτό σε πολυωνυμικό χρόνο; Τρέχουμε τον αλγόριθμο εξερεύνησης σε βάθος (Ενότητα 6.1) ή τον αλγόριθμο εξερεύνησης κατά πλάτος (Ενότητα 6.5) και ελέγχουμε αν οι x και y βρίσκονται στην ίδια συνεκτική συνιστώσα.

Πρόβλημα Ελάχιστου Δένδρου:
Στιγμιότυπο: Πεπερασμένο σύνολο από πόλεις $C = \{c_1, c_2, \ldots, c_m\}$, ένας θετικός ακέραιος $d(i, j)$ για κάθε ζεύγος πόλεων $c_i, c_j \in C$, ο οποίος δηλώνει την απόσταση μεταξύ των πόλεων και ένας θετικός ακέραιος B που είναι ο στόχος.
Ερώτημα: Υπάρχει δένδρο (υπογράφημα του αρχικού γραφήματος) που συνδέει όλες οι πόλεις του C και έχει μήκος μικρότερο ή ίσο του B;

Πώς μπορούμε να απαντήσουμε στο ερώτημα αυτό σε πολυωνυμικό χρόνο; Τρέχουμε οποιονδήποτε από τους αλγορίθμους που υπολογίζουν το μήκος του ελάχιστου δένδρου (Ενότητα 7.2) και συγκρίνουμε το μήκος που υπολογίσαμε με το B.

Παρατηρείστε πόσο μοιάζουν φαινομενικά τα προβλήματα του Περιοδεύοντος Πωλητή και του Ελάχιστου Δένδρου. (Οταν αναφερόμαστε σε προβλήματα που έχουμε ήδη ορίσει στο κεφάλαιο αυτό, θα γράφουμε τα αρχικά τους με κεφαλαία.) Φαίνεται απίστευτο ότι υπάρχουν τόσοι πολυωνυμικοί αλγόριθμοι για το δεύτερο πρόβλημα και είναι μάλλον αδύνατο να κατασκευαστεί ένας για το πρώτο.

Γιατί είναι τόσο σημαντικά τα προβλήματα απόφασης; Υπάρχουν δύο κύριοι λόγοι: Ο πρώτος είναι ότι είναι περίπου ισοδύναμα με τα προβλήματα βελτιστοποίησης, από τα οποία προήλθαν (αν υπάρχει πολυωνυμικός αλγόριθμος για το ένα από τα δύο προβλήματα, τότε υπάρχει και για το δεύτερο). Ο δεύτερος είναι ότι έχουν μια πολύ κομψή μαθηματική περιγραφή ως γλώσσες. Θα εξηγήσουμε μόνο τον πρώτο λόγο στις σημειώσεις αυτές.

Για οποιοδήποτε πρόβλημα βελτιστοποίησης με ζητούμενο: "Να βρεθεί η λύση με το μικρότερο (μεγαλύτερο) κόστος," είναι δυνατόν να κατασκευαστεί το αντίστοιχο πρόβλημα απόφασης με ερώτημα "Υπάρχει λύση με κόστος $\leq$ (αντιστοίχως $\geq$) ενός στόχου, που είναι ένας θετικός ακέραιος. Το πρόβλημα απόφασης αυτό δεν είναι δυσκολότερο από το πρόβλημα, από το οποίο προήλθε. Αν ξέρουμε έναν αλγόριθμο που βρίσκει τη βέλτιστη λύση, τότε μπορούμε να

συγκρίνουμε τη λύση αυτή με το στόχο. Αντιστρόφως, αν υπάρχει πολυωνυμικός αλγόριθμος που απαντά στο ερώτημα αν η λύση είναι $\geq$ (ή $\leq$ ανάλογα με το πρόβλημα) από ένα στόχο, τότε υπάρχει πολυωνυμικός αλγόριθμος που βρίσκει την τιμή της βέλτιστης λύσης με δυαδική αναζήτηση.

Ασκηση: Να γράψετε αναλυτικά τον αλγόριθμο που υπολογίζει τη βέλτιστη λύση με δυαδική αναζήτηση και χρησιμοποιεί ως υπορουτίνα το αντίστοιχο πρόβλημα απόφασης. Πρέπει να βρεθεί το άνω όριο

9.3. Οι κλάσεις $\mathcal{P}$ και $\mathcal{NP}$

Η κλάση όλων των προβλημάτων απόφασης, για τα οποία υπάρχουν πολυωνυμικοί αλγόριθμοι ονομάζεται $\mathcal{P}$. Στην κλάση $\mathcal{P}$ ανήκουν τα προβλήματα του Ελάχιστου Δένδρου και της Συνεκτικότητας. Τα προβλήματα που ανήκουν στην κλάση $\mathcal{P}$ θεωρούνται εύκολα.

Πριν από τον ορισμό της κλάσης $\mathcal{NP}$ ας εξετάσουμε ένα παράδειγμα: Εστω ότι έχετε ένα στιγμιότυπο από το πρόβλημα απόφασης του Περιοδεύοντος Πωλητή (ένα σύνολο από πόλεις, τις αποστάσεις τους και το στόχο). Πώς μπορείτε να πείσετε κάποιον ότι *υπάρχει* διαδρομή μικρότερη ή ίση από το στόχο; (Οτι, δηλαδή, η απάντηση στο ερώτημα είναι "ναι".) Μπορείτε να του δείξετε μια διαδρομή με μήκος μικρότερο ή ίσο του στόχου. Αυτός τότε μπορεί να ελέγξει την διαδρομή αυτή (αν ξεκινά και τελειώνει στην ίδια πόλη, αν περνάει από όλες τις πόλεις), να αθροίσει τις αποστάσεις και πολύ γρήγορα να πειστεί ότι έχετε δίκιο. Το χρονικό διάστημα που χρειαστήκατε για να βρείτε τη διαδρομή μπορεί να ήταν πολύ μεγάλο (μπορεί να ήταν αναγκαίο να ελέγξετε όλες τις διαδρομές μέχρι να βρείτε μία μικρότερη ή ίση με το στόχο). Από τη στιγμή όμως που τη βρήκατε, ειναι πολύ εύκολο να πείσετε κάποιον ότι η απάντηση είναι "ναι". Εχετε ένα "πιστοποιητικό" (certificate) ότι η απάντηση είναι "ναι": τη συγκεκριμένη διαδρομή. Οποιος αμφιβάλει, μπορεί να ελέγξει το πιστοποιητικό σας γρήγορα.

Η περίπτωση, όπου η απάντηση είναι όχι, *δεν είναι ανάλογη*. Εστω ότι ελέγξατε όλες τις διαδρομές, όλες είχαν μήκος μεγαλύτερο από το στόχο και έχετε πεισθεί ότι η απάντηση είναι "όχι". Πως μπορείτε να πείσετε κάποιον *γρήγορα* ότι η απάντηση είναι "όχι"; Δεν είναι καθόλου προφανές ότι αυτό είναι εφικτό, η υπόθεση των θεωρητικών είναι ότι δεν είναι, αλλά και αυτό δεν έχει ακόμα αποδειχθεί.

Ενας από τους ορισμούς της κλάσης $\mathcal{NP}$ (ίσως αυτός που γίνεται ευκολότερα κατανοητός) βασίζεται στην έννοια του πιστοποιητικού:

Ορισμός: Ενα πρόβλημα απόφασης ανήκει στην κλάση $\mathcal{NP}$ αν για κάθε στιγμιότυπο, για το οποίο η απάντηση είναι "ναι", υπάρχει πιστοποιητικό που είναι δυνατόν να ελεγθεί σε πολυωνυμικό χρόνο.

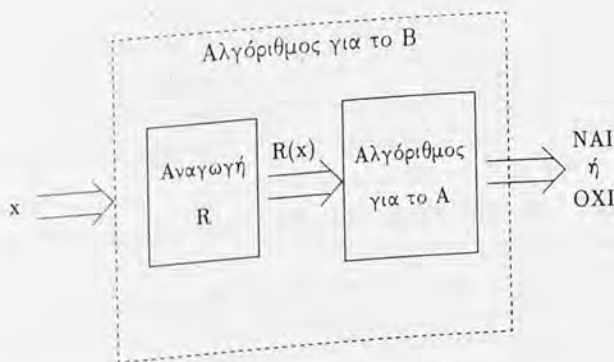
Ολα τα προβλήματα της κλάσης $\mathcal{P}$ ανήκουν στην κλάση $\mathcal{NP}$. Αρα είναι γνωστό ότι $\mathcal{P} \subseteq \mathcal{NP}$. Το μεγάλο ανοικτό πρόβλημα της πολυπλοκότητας σήμερα είναι αν οι δύο αυτές κλάσεις είναι ίδιες, αν δηλαδή $\mathcal{P} =^? \mathcal{NP}$. Οι περισσότεροι θεωρητικοί πιστεύουν ότι οι δύο αυτές κλάσεις *δεν είναι ίδιες*, δηλαδή υπάρχουν προβλήματα της κλάσης $\mathcal{NP}$, για τα οποία δεν υπάρχει πολυωνυμικός αλγόριθμος. Κανείς όμως δεν έχει αποδείξει την ικασία αυτή ακόμα.

9.4. Πολυωνυμικές αναγωγές και πλήρη προβλήματα

Στα προηγούμενα κεφάλαια ορίσαμε τα εξής προβλήματα της κλάσης $\mathcal{NP}$: το πρόβλημα της Συνεκτικότητας, του Ελάχιστου Δένδρου και του Περιοδεύοντος Πωλητή. Θα έχετε ισως την διαίσθηση ότι το πρόβλημα του Περιοδεύοντος Πωλητή περιέχει την πολυπλοκότητα της κλάσης περισσότερο από τα υπόλοιπα. Στο κεφάλαιο αυτό θα ορίσουμε μια έννοια που εξηγεί την διαίσθηση αυτή. Την έννοια του $\mathcal{NP}$-πλήρους προβλήματος. Θα αποδείξουμε ότι το πρόβλημα του Περιοδεύοντος Πωλητή είναι πλήρες για την κλάση $\mathcal{NP}$. Για να εξηγήσουμε όμως τι σημαίνει πλήρες πρόβλημα, θα πρέπει πρώτα να ορίσουμε την έννοια της πολυωνυμικής αναγωγής, καθώς και ένα ακόμη πρόβλημα, το οποίο έχει την τιμή να είναιτο πρώτο $\mathcal{NP}$-πλήρες πρόβλημα.

Ικανοποιησιμότητα: (Satisfiability, SAT)
Στιγμιότυπο: Συλλογή $C = \{c_1, c_2, \ldots, c_m\}$ από συνθήκες (clauses) σε ένα πεπερασμένο σύνολο μεταβλητών U.
Ερώτημα: Υπάρχει απόδοση τιμών αληθείας στο U, με την οποία ικανοποιούνται όλες οι συνθήκες;

Μια συνθήκη είναι το OR των στοιχείων (literals) που περιλαμβάνει. (Στοιχείο είναι μια μεταβλητή x_i ή η άρνησή της $\neg x_i$.) Για να ικανοποιηθεί μια συνθήκη πρέπει τουλάχιστον ένα από τα στοιχεία που περιλαμβάνει να πάρει την τιμή 1.



Σχήμα 9.1: Αναγωγή του B στο A

Παράδειγμα: Εστω ότι το σύνολο U περιλαμβάνει τις 3 μεταβλητές Boole $\{u_1, u_2, u_3\}$ και το σύνολο C περιλαμβάνει τις εξής συνθήκες: $c_1 = (u_1 \vee u_2 \vee u_3)$, $c_2 = (\neg u_1 \vee \neg u_2 \vee \neg u_3)$ και $c_3 = (\neg u_1 \vee u_2 \vee u_3)$. Η απάντηση στο ερώτημα είναι "ναι". Από τις 8 αποδόσεις τιμών αληθείας στις 3 μεταβλητές, οι εξής 5 ικανοποιούν όλες τις συνθήκες: $\{001, 010, 011, 101, 110\}$.

Το πρόβλημα της Ικανοποιησιμότητας είναι βασικό στη θεωρία της $\mathcal{NP}$ πληρότητας και, επειδή χρησιμοποιείται πολύ συχνά, ονομάζεται χαιδευτικά SAT. Ετσι θα το αποκαλούμε και εμείς στο εξής.

Το SAT ανήκει προφανώς στην κλάση $\mathcal{NP}$. Για κάθε στιγμιότυπο, για το οποίο η απάντηση είναι "ναι", υπάρχει πιστοποιητικό που είναι δυνατόν να ελεγχθεί σε πολυωνυμικό χρόνο: η απόδοση τιμών αληθείας που ικανοποιεί όλες τις συνθήκες.

Ενα πρόβλημα B ανάγεται σε ένα πρόβλημα A αν έχουμε έναν αλγόριθμο R που μετασχηματίζει κάθε στιγμιότυπο του B σε στιγμιότυπο του A με τέτοιο τρόπο ώστε η απάντηση στο ερώτημα του ενός στιγμιοτύπου να είναι "ναι", αν και μόνο αν η απάντηση στο ερώτημα του δεύτερου είναι "ναι". (Βλέπε και το Σχήμα (9.1).) Ετσι, αν έχουμε έναν αλγόριθμο που λύνει το A, μπορούμε να λύσουμε και το B ως εξής: Εκτελούμε τον R με είσοδο το στιγμιότυπο x του B. Ο R μετασχηματίζει το x σε στιγμιότυπο $R(x)$ του B. Εκτελούμε

τέλος τον αλγόριθμο για το A με είσοδο $R(x)$.

Ο αλγόριθμος R ονομάζεται **αναγωγή** (reduction) του B στο A. Αν ο R είναι πολυωνυμικός, τότε η αναγωγή ονομάζεται **πολυωνυμική**. (Μπορούμε να περιορίσουμε παραπάνω την έννοια της "γρήγορης" αναγωγής, αλλά αυτό δεν περιλαμβάνεται στους στόχους των σημειώσεων αυτών.)

Ας ορίσουμε ένα ακόμη πρόβλημα που θα το χρησιμοποιήσουμε στο πρώτο παράδειγμα αναγωγής:

Χρωματισμός Γραφήματος με Τρία Χρώματα: Graph 3-Colorability
Στιγμιότυπο: Γράφημα $G = (V, E)$.
Ερώτημα: Υπάρχει τρόπος να χρωματιστεί το γράφημα με τρία χρώματα, (κάθε κόμβος να έχει ένα χρώμα) τέτοιος που δύο γειτονικοί κόμβοι να μην έχουν το ίδιο χρώμα;

Παράδειγμα: Θα παρουσιάσουμε μια αναγωγή του Χρωματισμού Γραφήματος με Τρία Χρώματα στο SAT. Εστω ότι έχουμε ένα γράφημα $G = (V, E)$. Θα κατακευάσουμε ένα σύνολο από μεταβλήτες Boole και ένα σύνολο από συνθήκες $R(G)$, τέτοιες που: το G είναι δυνατόν να χρωματιστεί με τρία χρώματα αν και μόνο αν υπάρχει απόδοση τιμών αληθείας που ικανοποιεί όλες τις συνθήκες του $R(G)$.

Εχουμε $3|V|$ μεταβλητές (τριπλάσιες από τους κόμβους του γραφήματος). Σε κάθε κόμβο i αντιστοιχούνται 3 μεταβλητές, ας τις ονομάσουμε u_{i0}, u_{i1}, u_{i2}, οι οποίες έχουν το εξής νόημα: η u_{i0} θα πάρει την τιμή 1 αν ο κόμβος i χρωματιστεί με το πρώτο χρώμα, η u_{i1} θα πάρει την τιμή 1 αν ο κόμβος i χρωματιστεί με το δεύτερο και η u_{i2} θα παρει την τιμή 1 αν χρωματιστεί με το τρίτο. Οι συνθήκες που πρέπει να ικανοποιούν οι μεταβλητές αυτές είναι οι εξής: Για κάθε κόμβο i πρέπει να ισχύουν οι συνθήκες (9.1) και (9.2) και για κάθε ζεύγος κόμβων $[i, j]$ που συνδέονται με ακμή πρέπει να ισχύουν οι συνθήκες (9.3).

9.1 $(u_{i0} \vee u_{i1} \vee u_{i2})$: κάθε κόμβος πρέπει να χρωματιστεί με ένα τουλάχιστον χρώμα.

9.2 $(\neg u_{i0} \vee \neg u_{i1}), (\neg u_{i1} \vee \neg u_{i2}), (\neg u_{i0} \vee \neg u_{i2})$: δεν επιτρέπεται ένας κομβος να έχει δύο χρώματα

9.3 $(\neg u_{i0} \vee \neg u_{j0}), (\neg u_{i1} \vee \neg u_{j1}), (\neg u_{i2} \vee \neg u_{j2})$: δύο γειτονικοί κόμβοι δεν επιτρέπεται να έχουν το ίδιο χρώμα.

Είναι προφανές ότι η R είναι δυνατόν να εκτελεστεί σε χρόνο πολυωνυμικό στο μήκος τής εισόδου (του γραφήματος G). Εστω ότι το G είναι δυνατόν να χρωματιστεί με 3 χρώματα. (Υπάρχει συνάρτηση f, με την οποία κάθε κόμβος αντιστοιχείται σε ένα από τα τρία χρώματα $\{0,1,2\}$) Τότε υπάρχει απόδοση τιμών αληθείας που ικανοποιεί όλες τις συνθήκες και είναι η εξής: αν ο κόμβος i έχει για παράδειγμα το χρώμα 2, δίνουμε την τιμή 1 στην μεταβλητή που περιγράφει το χρώμα του (δηλαδή την u_{i2}) και την τιμή 0 στις υπόλοιπες 2 μεταβλητές του κόμβου (u_{i0}, u_{i1}). Είναι εύκολο να ελεχθεί ότι η απόδοση αυτή τιμών ικανοποιεί όλες τις συνθήκες. Εστω τώρα ότι το σύνολο των συνθηκών που κατασκευάσαμε είναι ικανοποιήσιμο. Τότε και το αρχικό γράφημα είναι δυνατόν να χρωματιστεί ως εξής: Παίρνουμε μια απόδοση τιμών που ικανοποιεί όλες τις συνθήκες. Στην απόδοση αυτή ακριβώς μια από τις 3 μεταβλητές που έχουν αντιστοιχηθεί στον κόμβο έχει πάρει την τιμή 1 (αποδείξτε ότι ο ισχυρισμός αυτός ισχύει). Δίνουμε στον κόμβο το χρώμα που αντιστοιχεί στη μεταβλητή αυτή. Αν χρωματίσουμε το γράφημα με τον τρόπο αυτόν, αποκλείεται δύο γειτονικοί κόμβοι να έχουν το ίδιο χρώμα. (Αποδείξτε ότι και αυτός ο ισχυρισμός ισχύει.) Αρα αποδείξαμε ότι ο Χρωματισμός Γραφήματος με Τρία Χρώματα είναι δυνατόν να αναχθεί πολυωνυμικά στο SAT. □

Δεν είναι ο Χρωματισμός Γραφήματος το μόνο πρόβλημα της κλάσης $\mathcal{NP}$, το οποίο είναι δυνατόν να αναχθεί στο SAT. Συγκεκριμένα, ο Cook απέδειξε το 1971 [9.1] ότι: **όλα τα προβλήματα της κλάσης $\mathcal{NP}$ είναι δυνατόν να αναχθούν πολυωνυμικά στο SAT** (Θεώρημα του Cook).

Ορισμός: Ενα πρόβλημα ονομάζεται $\mathcal{NP}$- πλήρες αν ανήκει στην κλάση $\mathcal{NP}$ και *όλα* τα προβλήματα της κλάσης $\mathcal{NP}$ είναι δυνατόν να αναχθούν πολυωνυμικά σε αυτό.

Αρα το SAT είναι ένα $\mathcal{NP}$-πλήρες πρόβλημα. Το πρώτο $\mathcal{NP}$-πλήρες πρόβλημα. Παρατηρείστε ότι, από τη στιγμή που βρέθηκε το πρώτο πλήρες πρόβλημα, εύκολα μπορούμε να αποδείξουμε ότι ένα νέο πρόβλημα X της κλάσης $\mathcal{NP}$ είναι $\mathcal{NP}$-πλήρες: Το μόνο που έχουμε να κάνουμε είναι να βρούμε πολυωνυμική αναγωγή του SAT στο X. Εφόσον όλα τα προβλήματα στο $\mathcal{NP}$ ανάγονται πολυωνυμικά στο SAT και το SAT στο X, όλα τα προβλήματα στο $\mathcal{NP}$ ανάγονται πολυωνυμικά στο X, άρα το X είναι $\mathcal{NP}$-πλήρες.

Τα $\mathcal{NP}$-πλήρη προβλήματα έχουν τεράστια σημασία για τον εξής λόγο: *αν βρεθεί πολυωνυμικός αλγόριθμος για ένα από αυτά, τότε υπάρχει πολυωνυμικός αλγόριθμος για όλα τα προβλήματα στο $\mathcal{NP}$*. (Από τον ορισμό της $\mathcal{NP}$-πληρότητας). Αρα αν βρεθεί πολυωνυμικός αλγόριθμος για κάποιο $\mathcal{NP}$-

πλήρες πρόβλημα, θα αποδειχθεί ότι $\mathcal{P} = \mathcal{NP}$. Κανείς βεβαίως δεν θεωρεί ότι αυτό είναι πιθανό.

9.5. Μερικά $\mathcal{NP}$–πλήρη προβλήματα.

Στις προηγούμενες ενότητες αναφέρθηκαν τα απαιτούμενα βήματα για να αποδείξουμε ένα ότι ένα πρόβλημα είναι $\mathcal{NP}$-πλήρες.

α. Αποδεικνύουμε ότι το πρόβλημα ανήκει στο $\mathcal{NP}$. (Οτι δηλαδή για κάθε στιγμιότυπο, για το οποίο η απάντηση είναι "ναι", υπάρχει πιστοποιητικό που είναι δυνατόν να ελεγχθεί σε πολυωνυμικό χρόνο.)

β. Κατασκευάζουμε μια πολυωνυμική αναγωγή ενός ήδη γνωστού $\mathcal{NP}$-πλήρους προβλήματος στο νέο πρόβλημα.

Στην ενότητα αυτή θα παρουσιάσουμε δύο αποδείξεις $\mathcal{NP}$-πληρότητας και θα ορίσουμε μερικά ακόμη $\mathcal{NP}$-πλήρη προβλήματα.

Υπάρχει μία εκδοχή του SAT, η οποία είναι πολύ πιο εύχρηστη στις αποδείξεις. Κάθε συνθήκη περιορίζεται να μήν έχει περισσότερα από 3 στοιχεία (literals)

3–Ικανοποιησιμότητα: (3–Satisfiability, 3SAT)
Στιγμιότυπο: Συλλογή $C = \{c_1, c_2, \ldots, c_m\}$ από συνθήκες σε ένα πεπερασμένο σύνολο μεταβλητών U. Κάθε συνθήκη περιλαμβάνει το πολύ τρία στοιχεία.
Ερώτημα: Υπάρχει απόδοση τιμών αληθείας στο U, με την οποία ικανοποιούνται όλες οι συνθήκες;

Ασκηση: Να αποδείξετε ότι το 3SAT είναι $\mathcal{NP}$-πλήρες πρόβλημα. Αν δεν τα καταφέρετε, να κοιτάξετε οποιαδήποτε από τις αναφορές [1], [4], [9], [9.1].

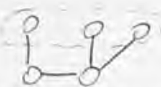
Αν όμως περιορίσουμε παραπάνω το SAT ώστε να έχει το πολύ δύο μεταβλητές σε κάθε συνθήκη (ονομάζεται τότε 2SAT), τότε υπάρχει πολυωνυμικός αλγόριθμος που το λύνει.

Ασκηση: Προσπαθείστε να σχεδιάσετε ένα πολυωνυμικό αλγόριθμο για το 2SAT. Αν δεν τα καταφέρετε, να κοιτάξετε την αναφορά [9].

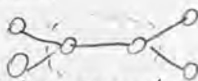
Παρατηρείστε πόσο λεπτή είναι η διαχωριστική γραμμή μεταξύ ενός προβλήματος που είναι $\mathcal{NP}$-πλήρες και ενός προβλήματος στο $\mathcal{P}$: Το 3SAT είναι $\mathcal{NP}$-πλήρες, το 2SAT είναι στο $\mathcal{P}$.

Μια από τις απλούστερες πολυωνυμικές αναγωγές είναι η αναγωγή του 3SAT στα προβλήματα της Κλίκας, του Ανεξάρτητου Συνόλου και της Κάλυψης με Κόμβους. Ας ορίσουμε τα προβλήματα αυτά:

Κλίκα: (Clique)
Στιγμιότυπο: Γράφημα $G = (V, E)$ και ακέραιος $j \leq |V|$.
Ερώτημα: Περιέχει το G μια κλίκα μεγέθους j ή μεγαλύτερη; (Υπάρχει δηλαδή κάποιο υποσύνολο V' των κόμβων μεγέθους τουλάχιστον j, τέτοιο που κάθε ζεύγος κόμβων στο V' να είναι γειτονικοί;)



Ανεξάρτητο Σύνολο (Independent Set)
Στιγμιότυπο: Γράφημα $G = (V, E)$ και ακέραιος $j \leq |V|$.
Ερώτημα: Περιέχει το G ανεξάρτητο σύνολο μεγέθους j ή μεγαλύτερο; (Υπάρχει δηλαδή κάποιο υποσύνολο V' των κόμβων μεγέθους τουλάχιστον j, τέτοιο που κάθε ζεύγος κόμβων στο V' να μήν είναι γειτονικοί;)



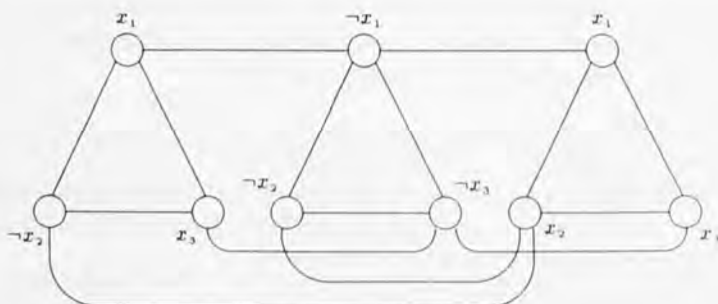
Κάλυψη με Κόμβους (Vertex Cover)
Στιγμιότυπο: Γράφημα $G = (V, E)$ και ακέραιος $j \leq |V|$.
Ερώτημα: Υπάρχει κάποιο υποσύνολο V' των κόμβων $|V'| \leq j$, με το οποίο να καλύπτονται όλες οι ακμές του E; (Για κάθε ακμή τουλάχιστον ένας, από τους δύο κόμβους που συνδέει, να ανήκει στο V';)

Παρατηρείστε πρώτα ότι τα τρία αυτά προβλήματα είναι περίπου το ίδιο πρόβλημα. Ανάγονται πολύ εύκολα το ένα στο άλλο. Συγκεκριμένα, για ένα γράφημα $G = (V, E)$ οι εξής τρεις προτάσεις είναι ισοδύναμες:

α. Το V' είναι ένα σύνολο κόμβων που καλύπτει όλες τις ακμές του G.

β. Οι υπόλοιποι κόμβοι $V - V'$ αποτελούν ανεξάρτητο σύνολο του G.

γ. Το σύνολο $V - V'$ αποτελεί κλίκα στο συμπληρωματικό γράφημα G^c του G. (Στο συμπληρωματικό γράφημα G^c του G δύο κόμβοι είναι γειτονικοί, αν δεν είναι γειτονικοί στο G.)



Σχήμα 9.2

Αναγωγή του 3SAT στο Ανεξάρτητο Σύνολο: Εστω ότι έχουμε ένα σύνολο από συνθήκες $C = \{c_1, c_2, \ldots, c_m\}$ στις μεταβλητές U. Θα κατασκευάσουμε ένα γράφημα $G = (V, E)$ και ένα στόχο j, για τα οποία ισχύει το εξής: Το γράφημα έχει ένα σύνολο V' από ανεξάρτητους κόμβους με $|V'| \geq j$, αν και μόνο αν υπάρχει απόδοση τιμών στο U που ικανοποιεί όλες τις συνθήκες του C. Το G έχει τρείς κόμβους για κάθε συνθήκη. Κάθε κόμβος αντιπροσωπεύει ένα στοιχείο της συνθήκης. Οι τρείς αυτοί κόμβοι είναι συνδεδεμένοι με ακμές σε ένα τρίγωνο. Συνδέουμε δύο κόμβους που ανήκουν σε διαφορετικά τρίγωνα με ακμή, αν και μόνο αν οι κόμβοι αυτοί παριστάνουν αντίθετα στοιχεία $(x_i, \neg x_i)$. Ο στόχος j γίνεται ίσος με τον αριθμό των συνθηκών m. Στο Σχήμα 9.2. παρουσιάζεται το γράφημα που κατασκευάζεται για το εξής σύνολο συνθηκών $c_1 = (x_1 \vee \neg x_2 \vee x_3)$, $c_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ και $c_3 = (x_1 \vee x_2 \vee x_3)$.

Εστω ότι το G περιλαμβάνει ανεξάρτητο σύνολο V' μεγέθους τουλάχιστον j. Δεν είναι δυνατόν δύο κόμβοι του V' να ανήκουν στο ίδιο τρίγωνο. Αρα το V' θα περιλαμβάνει ακριβώς j κόμβους, έναν από κάθε τρίγωνο. Κατασκευάζουμε απόδοση τιμών αληθείας ως εξής: Τα στοιχεία που παριστάνονται από κόμβους του ανεξάρτητου συνόλου V' παίρνουν την τιμή 1 και τα υπόλοιπα την τιμή 0. Η απόδοση τιμών αυτή ικανοποιεί όλες τις συνθήκες (ένα στοιχείο από κάθε συνθήκη πήρε την τιμή 1) και αποκλείεται να έχει την τιμή 1 για κάποιο στοιχείο και το αντίθετό του επειδή όλα τα ζεύγη αντίθετων στοιχείων συνδέονται μεταξύ τους με ακμές (άρα ένα ζεύγος αντίθετων στοιχείων δεν είναι δυνατόν να ανήκει στο ανεξάρτητο σύνολο).

Εστω τώρα ότι υπάρχει απόδοση τιμών που ικανοποιεί όλες τις συνθήκες. Στην απόδοση αυτή υπάρχει τουλάχιστον ένα στοιχείο που έχει πάρει την τιμή 1 σε κάθε συνθήκη. Ο αντίστοιχος κόμβος του G (ένας από κάθε συνθήκη) θα ανήκει στο ανεξάρτητο σύνολο. Με τον τρόπο αυτον μπορούμε να κατασκευάσουμε ένα ανεξάρτητο σύνολο μεγέθους m.

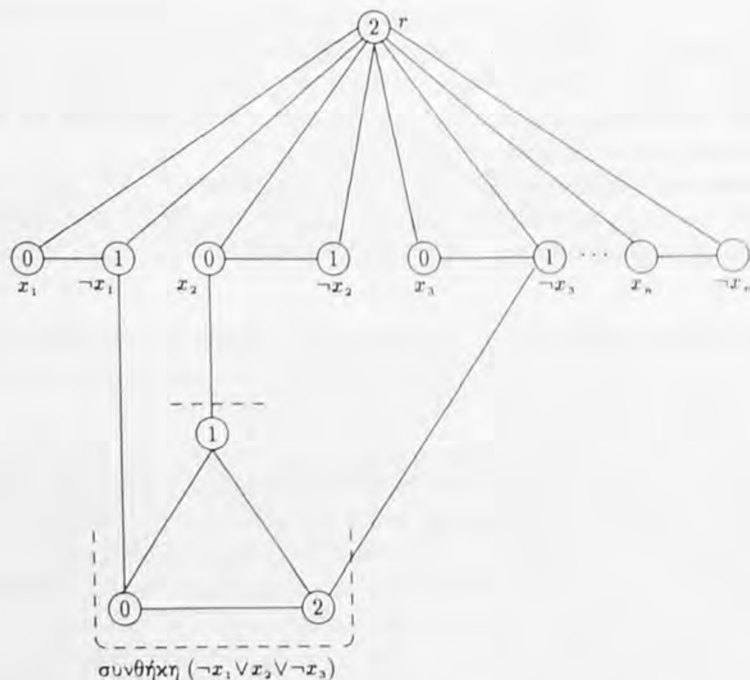
Εφόσον το Ανεξάρτητο Σύνολο ανήκει προφανώς στο $\mathcal{NP}$ και κάναμε αναγωγή του ήδη γνωστού $\mathcal{NP}$-πλήρους προβλήματος 3SAT στο πρόβλημα αυτό, έχουμε αποδείξει ότι το Ανεξάρτητο Σύνολο είναι $\mathcal{NP}$-πλήρες πρόβλημα. □

Ασκηση: Να αποδείξετε ότι τα προβλήματα της Κλίκας και της Κάλυψης με Κόμβους είναι $\mathcal{NP}$-πλήρη. (Να κάνετε αναγωγή του Ανεξάρτητου Συνόλου σε αυτά. Χρησιμοποιείστε την ισοδυναμία των Προτάσεων α. β. γ.)

Στην προηγούμενη ενότητα παρουσιάσαμε μια αναγωγή του Χρωματισμού Γραφήματος με Τρία Χρώματα στο 3SAT (παρατηρείστε ότι δεν κατασκευάσαμε καμία συνθήκη με περισσότερα από 3 στοιχεία). Με την αναγωγή αυτή *δεν αποδείξαμε* ότι το πρόβλημα αυτό είναι $\mathcal{NP}$-πλήρες. Αποδείξαμε μόνο ότι δεν είναι δυσκολότερο από το 3SAT. Για να αποδείξουμε ότι το πρόβλημα είναι $\mathcal{NP}$-πλήρες θα πρέπει να κάνουμε την αντίστροφη αναγωγή: Αναγωγή του 3SAT στο Χρωματισμό Γραφήματος. Για την αναγωγή αυτή θα χρησιμοποιήσουμε μια παραλαγή του 3SAT, η οποία είναι επίσης $\mathcal{NP}$-πλήρες πρόβλημα.

3SAT, όχι όλα ίσα: (Not all equal 3SAT)
Στιγμιότυπο: Συλλογή $C = \{c_1, c_2, \ldots, c_m\}$ από συνθήκες σε ένα πεπερασμένο σύνολο μεταβλητών U. Κάθε συνθήκη περιλαμβάνει το πολύ τρία στοιχεία.
Ερώτημα: Υπάρχει απόδοση τιμών αληθείας στο U, με την οποία ικανοποιούνται όλες οι συνθήκες και κάθε συνθήκη έχει τουλάχιστον ένα στοιχείο 1 και τουλάχιστον ένα στοιχείο 0;

Ασκηση: Να αποδείξετε ότι το 3SAT (όχι όλα ίσα) είναι $\mathcal{NP}$-πλήρες πρόβλημα. (Αναφορές [9], [9.4].)



Σχήμα 9.3

Αναγωγή του 3SAT (όχι όλα ίσα) στο Χρωματισμό Γραφήματος με 3 Χρώματα: Εστω ότι έχουμε ένα σύνολο από συνθήκες C στις μεταβλητές U. Θα κατασκευάσουμε ένα γράφημα $G = (V, E)$, για το οποίο ισχύει το εξής: Είναι δυνατόν να χρωματιστεί με τρία χρώματα αν και μόνο αν υπάρχει απόδοση τιμών στο U που ικανοποιεί όλες τις συνθήκες του C με τον περιορισμό να μην είναι όλες οι τιμές ίδιες σε μια συνθήκη. Για κάθε μεταβλητή x_i κατασκευάζουμε 2 κόμβους. Ο ένας αντιστοιχεί στο x_i και ο άλλος στο $\neg x_i$. Συνδέουμε με ακμή τους κόμβους που έχουν αντιστοιχηθεί σε ζεύγος αντίθετων στοιχείων. Ολοι οι κόμβοι που αντιπροσωπεύουν στοιχεία συνδέονται επίσης με έναν άλλο κόμβο που ονομάζεται r. (Σχήμα 9.3). Για κάθε συνθήκη κατασκευάζουμε ένα τρίγωνο. Κάθε κόμβος του τριγώνου αυτού συνδέεται με ένα κόμβο που αντιπροσωπεύει στοιχείο, το οποίο παρουσιάζεται στη συνθήκη

αυτή. Συνδέουμε λοιπόν τους τρεις κόμβους του τριγώνου της συνθήκης με τα τρία στοιχεία που εμφανίζονται στη συνθήκη.

Εστω ότι το γράφημα που κατασκευάσαμε με τον τρόπο αυτόν είναι δυνατόν να χρωματιστεί με τα τρία χρώματα $\{0,1,2\}$. Τότε προφανώς σε κάθε τρίγωνο παρουσιάζονται και τα τρία χρώματα. Μπορούμε να υποθέσουμε ότι το r παίρνει το χρώμα 2, οπότε κάθε x_i παίρνει το χρώμα 0 ή 1. Η απόδοση τιμών αληθείας είναι η εξής: Αν ο κόμβος x_i έχει πάρει το χρώμα 1, η αντίστοιχη μεταβλητή παίρνει την τιμή 1, αλλιώς 0. Η απόδοση αυτή τιμών ικανοποιεί όλες τις συνθήκες για τον εξής λόγο: Κάθε τρίγωνο που αντιπροσωπεύει συνθήκη αναγκαστικά έχει και τα τρία χρώματα. Αρα, κάποιος κόμβος του τριγώνου θα έχει το χρώμα 0 και ο κόμβος αυτός μπορεί να είναι γειτονικός μόνο με στοιχείο που έχει το χρώμα 1 (εφόσον το χρώμα 2 δεν παρουσιάζεται στους κόμβους των στοιχείων). Αρα, κάθε συνθήκη έχει τουλάχιστον ένα στοιχείο που έχει την τιμή 1. Εύκολα αποδεικνύεται ότι μια συνθήκη δεν είναι δυνατόν να έχει πάνω από 2 στοιχεία με την τιμή 1. (Το πρώτο θα συνδεθεί με τον κόμβο που έχει χρώμα 0, το δεύτερο με τον κόμβο με χρώμα 2, το τρίτο;)

Εστω τώρα ότι υπάρχει απόδοση τιμών αληθείας που ικανοποιεί όλες τις συνθήκες (με τον περιορισμό όχι όλα ίσα). Τότε υπάρχει και τρόπος να χρωματιστεί το γράφημα ως εξής: το r χρωματίζεται με το 2, οι κόμβοι που αντιστοιχούν στα στοιχεία χρωματίζονται με 0 ή 1 ανάλογα με την τιμή που έχουν στην απόδοση τιμών που ικανοποιεί όλες τις συνθήκες. Ενα τρίγωνο συνθήκης χρωματίζεται ως εξής: Ο κόμβος που συνδέεται με το στοιχείο που έχει την τιμή 1 (σίγουρα υπάρχει ένα τουλάχιστον στοιχείο με την τιμή 1) χρωματίζεται με το 0. Ο κόμβος που συνδέεται με το στοιχείο που έχει την τιμή 0 (σίγουρα υπάρχει ένα τουλάχιστον στοιχείο με την τιμή 0) χρωματίζεται με το 1. Ο τρίτος κόμβος χρωματίζεται με το 2.

Εφόσον ο Χρωματισμός Γραφήματος με Τρία Χρώματα ανήκει προφανώς στο $\mathcal{NP}$ και κάναμε αναγωγή του ήδη γνωστού $\mathcal{NP}$-πλήρους προβλήματος 3SAT (όχι όλα ίσα) στο πρόβλημα αυτό, έχουμε αποδείξει ότι ο Χρωματισμός Γραφήματος με Τρία Χρώματα είναι $\mathcal{NP}$-πλήρες πρόβλημα. □

Βασικό $\mathcal{NP}$-πλήρες πρόβλημα είναι ο κύκλος του Hamilton. Ο κύκλος του Hamilton ανάγεται πολύ εύκολα στο Πρόβλημα του Περιοδεύοντος Πωλητή.

Κύκλος Hamilton: (Hamiltonian Circuit)
Στιγμιότυπο: Γράφημα $G = (V, E)$.
Ερώτημα: Περιέχει το G έναν κύκλο που περιλαμβάνει όλους τους κόμβους; (Υπάρχει δηλαδή μια διάταξη όλων των κόμβων $< v_1, v_2, \ldots, v_{|V|} >$, τέτοια

που κάθε ζεύγος συνεχόμενων στη διάταξη κόμβων να είναι γειτονικοί και επίσης να είναι γειτονικοί οι $v_{|V|}$, v_1;)

Αναγωγή της Κάλυψης με Κόμβους στον Κύκλο του Hamilton υπάρχει στις Αναφορές [9.3] και [4]. Αναγωγή του 3SAT στον κύκλο του Hamilton υπάρχει στην Αναφαρά [9].

Ασκηση Δοκιμάστε να κάνετε την αναγωγή του Κύκλου του Hamilton στο Πρόβλημα του Περιοδεύοντος Πωλητή. Είναι από τις απλούστερες αναγωγές. (Αναφορές [4], [6], [9].)

Στην ενότητα αυτή παρουσιάστηκαν μερικά $\mathcal{NP}$-πλήρη προβλήματα. Παραλήφθηκαν πολλά σημαντικά προβλήματα της κατηγορίας αυτής. Ενας πλήρης κατάλογος με μερικές εκατοντάδες προβλήματα παρουσιάζεται στην αναφορά [4]. Ο κατάλογος αυτός ενημερώνεται συνεχώς στην αναφορά [9.2].

ΑΝΑΦΟΡΕΣ

Η αρχική απόδειξη του Θεωρήματος του Cook υπάρχει στην αναφορά:

9.1 S. A. Cook "The Complexity of Theorem Proving Procedures" *Proc. of 3rd Ann. ACM Symp. on Theory of Computing*, 1971.

Απόδειξη του θεωρήματος του Cook μπορείτε επίσης να βρείτε στην Αναφορά [4]. Στην [9] το θεώρημα του Cook αποδεικνύεται με ένα νέο πιο κατανοητό τρόπο.

9.2 D. S. Johnson "The NP-completeness column: An on-going guide," *J. of Algorithms, 4*, 1981 και έκτοτε.

9.3 R. M. Karp "Reducibility among combinatorial problems" *Complexity of Computer Computations*, Plenum Press 1972.

9.4 T. J. Schaefer "The Complexity of satisfiability problems" *Proc. of 10th Ann. ACM Symp. on Theory of Computing*, 1978.

Κεφάλαιο 10

ΑΛΓΟΡΙΘΜΟΙ ΘΕΩΡΙΑΣ ΑΡΙΘΜΩΝ

10.1. Κρυπτογράφηση με δημόσιο κλειδί κρυπτογράφησης

'Εστω A,B,C,... οι επικοινωνούντες μέσω ενός δημοσίου, μη ασφαλούς διαύλου. Στο σύστημα "ασφαλείας με δημόσιο κλειδί" κάθε ένας από A,B,... *δημοσιεύει* ένα κλειδί $P_A, P_B, \ldots$. Δεδομένου ενός μηνύματος που κάποιος τρίτος, έστω ο Z θέλει να στείλει π.χ. στον A, αυτό πρέπει να κρυπτογραφηθεί από τον Z με μια διαδικασία που στηρίζεται στο κλειδί του παραλήπτη. 'Ετσι, η αποστολή μηνύματος M στον A συνεπάγεται την αποστολή του $P_A * M$, όπου με $*$ συμβολίζεται η διαδικασία κρυπτογράφησης. Φυσικά θα πρέπει η αποκρυπτογράφηση του $P_A * M$ να είναι δυνατή *μόνο* από τον A.

Κάθε ένας των A,B,C,... διαθέτει ένα κλειδί αποκρυπτογράφησης $S_A, S_B, S_C, \ldots$ και μια διαδικασία $S_A*, \ldots$ ώστε να ισχύει

$$S_A * (P_A * M) = M, \ \ldots \ S_Z * (P_Z * M) = M.$$

Επιπλέον θα πρέπει να είναι δύσκολη η εύρεση των S από τα P.

Αν *επιπλέον ισχύει ότι* για κάθε μήνυμα

$$C : P_A * (S_A * C) = C, \ldots, P_Z * (S_Z * C) = C$$

τότε είναι δυνατή η *"ηλεκτρονική υπογραφή"* ως εξής: 'Εστω ότι ο Z θέλει να στείλει στον A το μήνυμα M. Κατόπιν συμφωνίας του στέλνει το μήνυμα

M " + "$S_Z * M$, που κρυπτογραφείται ως

$$P_A * M \text{ " + " } P_A * (S_Z * M)$$

Το " + " σημαίνει παράθεση.
Ο Α αποκρυπτογραφεί με το S_A και έχει

$$S_A * (P_A * M) \text{ " + " } S_A * (P_A * (S_Z * M)) = M \text{ " + " } S_Z * M$$

Στο κείμενο του Μ αναφέρεται ότι αυτό το μήνυμα εστάλη από τον Ζ. Για να το επιβεβαιώσει αυτό ο Α εφαρμόζει την κρυπτογράφηση ως προς Ζ στο δεύτερο μέρος του μηνύματος, δηλαδή εφαρμόζει την διαδικασία

$$P_Z * (S_Z * M)$$

που είναι εφικτό εφόσον το P_Z είναι γνωστό. Αν όντως ο Ζ έχει στείλει το μήνυμα θα πρέπει να ισχύει

$$M = P_Z * (\text{"Δεύτερο μέρος μηνύματος"})$$

εφόσον όντως

$$M = P_Z * (S_Z * M)$$

Έτσι το $S_Z * M$ είναι η ηλεκτρονική υπογραφή του Ζ.

Για λεπτομέρειες σε θέματα ασφάλειας δες το σύγγραμμα του κ.Κιουντούζη "Ασφάλεια Πληροφοριακών Συστημάτων".

10.2. Το σύστημα RSA (Rivest, Shamir, Adleman)

Ένα σύστημα που ικανοποιεί (προς το παρόν) τα επιθυμητά στοιχεία που αναφέρθηκαν παραπάνω είναι και το εξής, που οφείλεται στους Rivest, Shamir και Adleman:
Η περιγραφή είναι σε βήματα
Βήμα 1°

Υπολόγισε δύο μεγάλους πρώτους αριθμούς p, q και θέσε $n = p \cdot q$, $\phi = (p-1)(q-1)$

p, q, n, φ, e, d, P, S

Παρατηρήσεις • Είναι δύσκολο να βρει κανείς μεγάλους πρώτους π.χ. με 200 ψηφία. Μικρότεροι πρώτοι κάνουν το σύστημα ανασφαλές.
• Γνωρίζοντας το ϕ δεν είναι υπολογιστικά δυνατόν να βρεθούν οι $p-1, q-1$.
Βήμα 2° Βρες ένα "μικρό" αριθμό e πρώτο προς το ϕ, δηλαδή τέτοιο ώστε $gcd(e,\phi) = 1$ (gcd: μέγιστος κοινός διαιρέτης).
Παρατήρηση Το πρόβλημα της εύρεσης υποψηφίων e είναι εύκολο.
Βήμα 3° Βρες d έτσι ώστε $ed = 1 \bmod \phi$ και $d < \phi$.
Παρατήρηση Το d υπάρχει και είναι μοναδικό. Υπολογίζεται εύκολα με τον Ευκλείδιο αλγόριθμο, εφόσον από την παραπάνω σχέση, πρέπει $gcd(ed,\phi) = 1$.
Βήμα 4° Θέσε $P = (e,n)$, $S = (d,n)$.
Βήμα 5° Έστω M μήνυμα υπό μορφή αριθμού μικρότερου του n. Τότε τα βήματα της κρυπτογράφησης είναι:
Κρυπτογράφηση $P * M = M^e \bmod n$
Αποκρυπτογράφηση $S * C = C^d \bmod n$.
Παρατήρηση Πρέπει να έχουμε αποτελεσματικό αλγόριθμο ύψωσης σε δύναμη.
Αριθμητικό Παράδειγμα
Βήμα 1° Έστω $p = 37$ $q = 41$, οπότε $n = 37 \cdot 41 = 1517$
$\phi = (37-1)\cdot(41-1) = 1440$.
Βήμα 2° Θέτουμε $e = 13$.
Από τον Ευκλείδιο αλγόριθμο (δες παρακάτω) βρίσκουμε ότι

$$13899 = 1 + 9 * 1440$$

άρα $13 * 997 = 1 \bmod 1440$.
Βήμα 3° Από το Βήμα 2 έχουμε $d = 997$.
Βήμα 4° $P = (13, 1517)$ $S = (997, 1517)$.
Βήμα 5° Έστω $M = 256$. Αυτό κρυπτογραφείται ως

$$P * 256 = 256^{13} \bmod 1517$$

που ισούται (πράξεις...) με 303.

Η αποκρυπτογράφηση δίνει

$$303^{997} \bmod 1517$$

που όντως ισούται με 256.

Αντίστροφα $S * 256 = 256^{997} \bmod 1517 = 551$ και $P * 551 = 551^{13} \bmod 1517$ που όντως ισούται με 256.

ΕΡΓΑΣΙΕΣ

1. Περιγράψτε, υλοποιείστε άλλα συστήματα από την βιβλιογραφία.
2. Κάνετε επισκόπηση και υλοποίηση των μεθόδων του συγγράμματος MIT-CLR για
   - εύρεση πρώτων αριθμών
   - παραγοντοποίηση αριθμών
   - ελέγχου αν ένας αριθμός είναι πρώτος

10.3. Δικαιολόγηση συστήματος RSA

Ορισμός *Για δύο ακέραιους* α, β *λέμε* $\alpha = \beta \bmod n$ *για ακέραιο* n, *αν το* n *διαιρεί το* $\alpha - \beta$ *ή ισοδύναμα* $\alpha = \beta + \lambda n$ *για ακέραιο* λ.

Συμβολισμός *Για ακέραιους* α, β *το σύμβολο* $gcd(\alpha, \beta)$ *συμβολίζει τον μέγιστο κοινό διαιρέτη τους (μ.κ.δ.).*

Ο μ.κ.δ. βρίσκεται με τον περίφημο Ευκλείδιο Αλγόριθμο που βασίζεται στην παρακάτω παρατήρηση.

Λήμμα Έστω $\alpha > \beta$ και $\alpha = \mu\beta + r$ με $r < \beta$ (διαίρεση: μ πηλίκον, r υπόλοιπο). Είναι $gcd(\alpha, \beta) = gcd(\beta, r)$.

Απόδειξη Εφόσον $\alpha = \mu\beta + r$, κάθε διαιρέτης των β, r διαιρεί τον α. Επίσης διαιρεί και τον β. Άρα

$$gcd(\beta, r) \leq gcd(\alpha, \beta)$$

Αντίστροφα, εφόσον $r = \alpha - \mu\beta$ κάθε διαιρέτης των α, β διαιρεί το r και φυσικά το β. Άρα $gcd(\alpha, \beta) \leq gcd(\beta, r)$ και τελικά $gcd(\alpha, \beta) = gcd(\beta, r)$.

Συμβολίζουμε με $\lfloor \alpha/\beta \rfloor$ το πηλίκο της διαίρεσης $\alpha : \beta$, οπότε έχουμε $\alpha = \lfloor \alpha/\beta \rfloor \beta + r$. Φυσικά $\alpha = r \bmod \beta$ ή $r = \alpha \bmod \beta$, οπότε έχουμε $\alpha = \lfloor \alpha/\beta \rfloor \beta + \alpha \bmod \beta$.

Προφανώς $gcd(a, 0) = a$.

Με αυτές τις παρατηρήσεις έχουμε την εξής υλοποίηση του Ευκλείδιου αλγόριθμου που χρησιμοποιεί αναδρομή:

```
function gcd(α: integer, β: integer): integer;
(* Υποθέτουμε α>β *)
  if β=0 then gcd ← α
   else
    gcd ← gcd(β, α mod β)
```

Άσκηση Γράψτε ένα μη αναδρομικό πρόγραμμα για τον Ευκλείδιο αλγόριθμο.

Η χειρότερη συμπεριφορά τοπυ αλγόριθμου παρατηρείται όταν το $\alpha \bmod \beta$ είναι σχετικά "κοντά" στο α. Αυτό θα συμβεί αν η διαίρεση $\alpha : \beta$ δώσει πηλίκον 1, η επόμενη διαίρεση δώσει πάλι πηλίκον 1, κ.ο.κ., πράγμα που μπορεί να περιγραφεί ως εξής: Έστω $\alpha = \phi_{n+1}$ $\beta = \phi_n$. Για να ισχύσει η χειρότερη περίπτωση πρέπει

$$\begin{aligned}
\phi_{n+1} &= 1 \cdot \phi_n + \phi_{n-1} \\
\phi_n &= 1 \cdot \phi_{n-1} + \phi_{n-2} \\
\phi_{n-1} &= 1 \cdot \phi_{n-2} + \phi_{n-3} \\
&\vdots \\
\phi_{n-k} &= 1 \cdot 1 + 1 = 2
\end{aligned}$$

Άρα στην χειρότερη περίπτωση ισχύει $\phi_{n+1} = \phi_n + \phi_{n-1}$ με $\phi_{\alpha\rho\chi\iota\kappa\acute{\alpha}} = 1, 1$.

Άρα το ϕ_n είναι οι αριθμοί Fibonacci με $\phi_n \approx \left(\frac{1+\sqrt{5}}{2}\right)^n / \sqrt{5}$.

Πιο συγκεκριμένα, ο υπολογισμός του $gcd(\alpha, \beta)$ χρειάζεται το πολύ k υπολογισμούς αν $\beta < \phi_{k+1}$ $(\alpha > \beta)$.

Για να λύσουμε την σχέση $ed = 1 \bmod \phi$, που είδαμε στην περιγραφή του RSA, χρειαζόμαστε ένα πρόσθετο χαρακτηριστικό του $gcd(\alpha, \beta)$.

Πρόταση Υπάρχουν ακέραιοι x, y ώστε $a\chi + by = gcd(\alpha, \beta)$.

Προφανώς αν β διαιρεί τον α, η σχέση προκύπτει εύκολα: Αν $\alpha = \lambda\beta$ τότε

$$\underline{\alpha - (\lambda - 1)\beta = \beta.}$$

Αν ο β δεν διαιρεί τον α, τότε είναι $\alpha = \lfloor \alpha/\beta \rfloor \beta + \alpha \bmod \beta$.

Έστω ότι μπορούσαμε να γράψουμε

$$gcd(\beta, \alpha \bmod \beta) = \beta x' + \alpha \bmod \beta y'.$$

Είναι βέβαιο $gcd(\beta, \alpha \bmod \beta) = gcd(\alpha, \beta)$. Επίσης

$$\beta x' + \alpha \bmod \beta y' = \beta x' + (\alpha - \lfloor \alpha/\beta \rfloor) y' = \alpha \underbrace{y'}_{x} + \beta \underbrace{(x' - \lfloor \alpha/\beta \rfloor y')}_{y}$$

Η σχέση αυτή αποτελεί βάση αναδρομής με ίδια βήματα όπως ο Ευκλείδιος αλγόριθμος. Αν ξέρουμε τα x', y' αναδρομικά μπορούμε να υπολογίσουμε τα x, y από τις σχέσεις

$$x \leftarrow y' \quad y \leftarrow x' - \lfloor \alpha/\beta \rfloor y'$$

Παράδειγμα Είναι $gcd(48,18) = 6$. Θέλουμε να υπολογίσουμε x, y ώστε $48x + 18y = 6$. Τα βήματα του αλγορίθμου είναι ως εξής:

Βήμα	α	β	$\lfloor\alpha/\beta\rfloor$	$\alpha \bmod \beta$	x		y
1	48	18	2	12	2		−5
		↙				↖↗	↑
2	18	12	1	6	−1		2
		↙				↖↗	↑
3	12	6	2	0	1		−1

Ο υπολογισμός των x, y προχωρά από το βήμα 3 προς το βήμα 2 κ.λπ. Π.χ. το y στο βήμα 1 προκύπτει από τον τύπο $y \to x' - \lfloor\alpha/\beta\rfloor y'$ θέτοντας $x' = -1$ $\lfloor\alpha/\beta\rfloor = 2$ $y' = 2$. Όντως ισχύει

Βήμα 1	$48 \cdot 2 + 18 \cdot (-5) = 6$
Βήμα 2	$18 \cdot (-1) + 12 \cdot 2 = 6$
Βήμα 3	$12 \cdot (1) + 6 \cdot (-1) = 6$

Από το Βήμα 1 είναι $48 \cdot 2 + 18 \cdot (-5) = 6$.

Ασκήσεις

(α) Γράψτε ένα αναδρομικό αλγόριθμο που υπολογίζει το $gcd(\alpha, \beta)$ και δύο x, y ώστε $\alpha x + \beta y = gcd(\alpha, \beta)$.

(β) Γράψτε έναν μη αναδρομικό αλγόριθμο για το (α).

Λέμε ότι δύο αριθμοί είναι *πρώτοι μεταξύ των* αν ο μέγιστος κοινός διαιρέτης των είναι 1. Οι α, β δηλαδή είναι πρώτοι μεταξύ των αν $gcd(a, b) = 1$. Άρα στο βήμα 2 του RSA προσπαθούμε να βρούμε ένα μικρό αριθμό e ώστε $gcd(e, \phi) = 1$, πράγμα σχετικά εύκολο δοκιμάζοντας μικρούς πρώτους αριθμούς π.χ. 5,7,13 κ.λπ. Αλλά από τις προηγούμενες παρατηρήσεις ο Ευκλείδιος αλγόριθμος μας δίνει ένα x ώστε $ex + \phi y = 1$ που μας διευκολύνει πολύ στον υπολογισμό του $d(< \phi)$, ώστε $ed = 1 \bmod \phi$ $(d = x \bmod \phi)$.

Παράδειγμα Ο υπολογισμός στο Βήμα 2 της προηγούμενης σελίδας είναι ο εξής σύμφωνα με το προηγούμενο παράδειγμα:

Βήμα	α	β	$\lfloor\alpha/\beta\rfloor$	$\alpha \bmod \beta$	x	y
1	1440	13	110	10	−9	997
2	13	10	1	3	7	−9
3	10	3	3	1	−2	7
4	3	1	3	0	1	−2

που επιβεβαιώνονται εφόσον

$$\begin{array}{rcrcrcrl} 1440 & \cdot & (-9) & + & 13 & \cdot & 997 & = 1 \\ 13 & \cdot & 7 & + & 10 & \cdot & (-9) & = 1 \\ 10 & \cdot & (-2) & + & 3 & \cdot & 7 & = 1 \\ 3 & \cdot & 1 & + & 1 & \cdot & (-2) & = 1 \end{array}$$

Για να επιβεβαιώσουμε τις σχέσεις $P * S * M = M$ και $S * P * M = M$ υπολογίζουμε για $M < n(= p \cdot q)$ τις παραστάσεις

$$\begin{aligned} S \cdot P \cdot M &= (M^e)^d \bmod n \quad \text{και} \\ P \cdot S \cdot M &= (M^d)^e \bmod n \end{aligned}$$

Αρκεί να εξετάσουμε προφανώς μόνο την μία απ' αυτές: (* Τα παρακάτω προαιρετικά *)

$$M^{ed} = M^{1+\lambda(p-1)(q-1)} = M \cdot (M^{p-1})^{\lambda(q-1)}$$

Εξετάζουμε το $M^{p-1} \bmod p$, όπου p πρώτος. Η παράσταση αυτή ισούται με 1 σύμφωνα με ένα περίφημο θεώρημα του Fermat. Αυτό ισχύει διότι:

- Το σύνολο $M^0, M^1, \ldots, M^k, \ldots \bmod p$ είναι πεπερασμένο ($< p$), και δεν περιλαμβάνει το 0 (μηδέν) εφόσον p πρώτος.
- Υπάρχει λοιπόν ελάχιστο k με $M^k = 1 \bmod p$. Διαφορετικά λόγω της κυκλικότητας θα ήταν $M^{k_1} = M^{k_2} \Rightarrow M^{k_1-k_2} = 1$.
- Αν υπάρχει $x < n$ με $x = M^y$ εξετάζουμε τὸ σύνολο $x, xM, xM^2, \ldots, xM^k$.
- Τελικά όλοι οι αριθμοί $1, \ldots, p-1$ διαμερίζονται σε ομάδες καθεμιά με k στοιχεία, άρα k διαιρεί το $p-1$ και $M^{p-1} = (M^k)^\lambda = 1^\lambda = 1 \bmod p$.

Έχουμε λοιπόν $M^{ek} = M \bmod p$ και επίσης $M^{ed} = M \bmod q$. Από αυτό συνεπάγεται ότι $M^{ed} = M \bmod pq$ ή τελικά $M^{ed} = M \bmod n$. Πράγματι, αν $\alpha = \beta \bmod p$ και $\alpha = \beta \bmod q$ με p, q πρώτους θα είναι και $\alpha - \beta = 0 \bmod p$ και $a - b = 0 \bmod q$, δηλαδή ο αριθμός $\alpha - \beta$ διαιρείται με τον πρώτο p, καθώς και με τον πρώτο q. Άρα ο $\alpha - \beta$ διαιρείται και με το γινόμενό τους $p \cdot q$, δηλαδή $\alpha - \beta = 0 \bmod pq$ ή $\alpha = \beta \bmod pq$.

<u>Παράδειγμα</u> Έστω $p = 37$. Τότε θα είναι $M^{36} = 1 \bmod 37$ για κάθε αριθμό M. Έστω $M = 12$ και $M^2 = 12^2 = 144 = -4 \bmod 37$ (εφόσον $144 + 4 =$

$4 \cdot 37)$. Άρα $12^{36} = (-4)^{18} = 4^{18} \bmod 37 = (4^3)^6 = 10^6 \bmod 37$. Αλλά $10^6 = (10^3)^2$ και $10^3 = 1 \bmod 37$, έτσι ώστε τελικά να επιβεβαιώνεται η σχέση $M^{36} = 1$.

* Τέλος προαιρετικής ύλης *

Από τα παραπάνω φαίνεται ότι υπάρχει ανάγκη ενός αποτελεσματικού αλγορίθμου για ύψωση σε δυνάμεις με πράξεις mod n.

Μία απλή μέθοδος είναι ύψωση μέσω διαδοχικών τετραγωνισμών. Παρατηρούμε ότι οι διαδοχικοί τετραγωνισμοί: $\alpha \xrightarrow{1^o} \alpha^{2^1} \xrightarrow{2^o} (\alpha^2)^2 = \alpha^{2^2} \xrightarrow{3^o} (\alpha^{2^2})^2 = \alpha^{2^3}$ δημιουργούν σε k βήματα την δύναμη α^{2^k}. Αν τώρα χρειάζεται να υπολογίσουμε το α^β, όπου $\beta = \sum_{n=0}^{k} \beta_n 2^n$, με β_n 0 ή 1, ο υπολογισμός μπορεί να γίνει αφού συνδυάσουμε τα αποτελέσματα των τετραγωνισμών: $\alpha^\beta = \alpha^{\sum \beta_n 2^n} = \prod_{n=0} \alpha^{\beta_n 2^n}$

<u>*Παράδειγμα*</u> $\alpha^{17} =$; Είναι $17 = 1 \cdot 2^4 + 1$ άρα $\alpha^{17} = \alpha^{2^4} \cdot \alpha$. Το α^{2^4} υπολογίζεται με τέσσερις διαδοχικούς τετραγωνισμούς:

$$\alpha \to \alpha^2 \to \alpha^4 \to \alpha^8 \to \alpha^{16}$$

Τα παραπάνω μπορούν να συνοψισθούν σε ένα αλγόριθμο: Το $\beta = \sum_{n=0}^{k} \beta_n 2^n$ υπολογίζεται και με διαδοχικούς διπλασιασμούς ως εξής: Έστω $r_k = \beta_k$ και $r_{n-1} = 2 \cdot r_n + \beta_{n-1}$. Είναι προφανές ότι $r_{k-1} = 2 \cdot \beta_k + \beta_{k-1}$, $r_{k-2} = 2^2\beta_k + 2\beta_{k-1} + \beta_{k-2}$, και $r_0 = \sum_{n=0}^{k} \beta_n 2^n$. Η παραπάνω διαδικασία ύψωσης δυνάμεων οδηγεί στη σχέση

$$S_k = a^{r_k} = \alpha^{\beta_k}$$
$$S_{n-1} = \alpha^{r_{n-1}} = \alpha^{\beta_{n-1}}(\alpha^{r_n})^2 = \alpha^{\beta_{n-1}} S_n^2$$

ή τελικά $\underline{S_{n-1} = \alpha^{\beta_{n-1}} \cdot S_n^2}$

και $\underline{S_0 = \alpha^{r_0} = \alpha^\beta}$

Προφανώς αν $\beta_{n-1} = 0$, $S_{n-1} = S_n^2$, ενώ αν $\beta_{n-1} = 1$, $S_{n-1} = \alpha \cdot S_n^2$. Τα παραπάνω οδηγούν στον εξής αλγόριθμο:

```
function power (α,β,n)
```
(* Υπολογίζει $\alpha^\beta \bmod n, \quad \beta = \sum_{n=0}^{k} \beta_n \cdot 2^n$ *)

```
s:=1
for i:=k downto 0 do
    begin
      s:= s · s mod n;
      if βi = 1 then s:=α · s mod n
    end
power:=s
```

Παράδειγμα Θέλουμε να υπολογίσουμε το α^9. Αλλά $9 = 1001_{(2)}$, οπότε ο εκθέτης σε δυαδική έκφραση είναι $\beta = 1001$ με $\beta_3 = 1$, $\beta_2 = \beta_1 = 0$, $\beta_0 = 1$. Ο αλγόριθμος αρχίζει με $k = 3$, οπότε $s = \alpha' = \alpha$. Τα επόμενα βήματα είναι

$$
\begin{aligned}
s_2 &= \alpha^{\beta_2} s_3^2 = \alpha^0 \alpha^2 = \alpha^2 \\
s_1 &= \alpha^{\beta_1} s_2^2 = \alpha^0 (\alpha^2)^2 = \alpha^4 \\
s_0 &= \alpha^{\beta_0} s_1^2 = \alpha^1 (\alpha^4)^2 = \alpha^9
\end{aligned}
$$

Άσκηση Γράψτε ένα αλγόριθμο για τον πολλαπλασιασμό δύο (2) αριθμών που λειτουργεί με διαδοχικούς διπλασιασμούς.

Κεφάλαιο 11

ΑΛΛΕΣ ΕΦΑΡΜΟΓΕΣ ΤΟΥ ΔΥΝΑΜΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

11.1. Το πρόβλημα του σακκιδίου

Έστω K τύποι αντικειμένων με δείκτη $j = 1, \ldots, K$. Κάθε αντικείμενο έχει δεδομένα ένα βάρος B_j και μία αξία A_j. Έστω ότι διαθέτουμε ένα σακκίδιο χωρητικότητας M. Ποιά είναι η καλύτερη επιλογή αντικειμένων ώστε να χρησιμοποιήσουμε την αξία των αντικειμένων που περιλαμβάνουμε στο σακκίδιο, χωρίς βέβαια να υπερβούμε την χωρητικότητά του.

Διακρίνουμε δύο περιπτώσεις: Στην πρώτη, μας επιτρέπεται να επιλέξουμε όσα αντικείμενα θέλουμε από κάθε είδος. Στην δεύτερη, επιτρέπεται η επιλογή ενός (ή κανενός) αντικειμένου από κάθε είδος.

11.1.1. Πρώτη περίπτωση - (Integer Knapsack)

Έστω $F(M)$ η αξία της βέλτιστης φόρτωσης χωρητικότητας M. Αν αφαιρέσουμε ένα αντικείμενο, π.χ. το j, τότε τα υπόλοιπα αντικείμενα πρέπει να αποτελούν βέλτιστη φόρτωση σακκιδίου χωρητικότητας $M - B_j$ και άρα $F(M) = A_j + F(M - B_j)$. Αν τώρα εξετάσουμε ένα αντικείμενο k που δεν υπάρχει στην βέλτιστη φόρτωση θα πρέπει $F(M) > A_k + F(M - B_k)$, αλλιώς

η φόρτωση θα βελτιωνόταν με την περίληψη του k. Επομένως ισχύει

$$F(M) = \max_j [A_j + F(M - B_j)]$$

Προφανώς είναι $F(M) = 0$ για $M < \min_j B_j$, και επομένως η επίλυση της εξίσωσης μπορεί να γίνει υπολογίζοντας διαδοχικά τα $F(0), F(1), \ldots, F(M)$, δεδομένου ότι αν γνωρίζουμε την τιμή του F για τιμές $1, 2, \ldots, M-1$ μπορούμε να υπολογίσουμε την τιμή του F στο M με βάση την εξίσωση δυναμικού προγραμματισμού.

Άσκηση Γράψτε έναν αλγόριθμο που

(α) λύνει την εξίσωση του δυναμικού προγραμματισμού για το σακκίδιο και

(β) υπολογίζει την βέλτιστη φόρτωση.

11.1.2. Δεύτερη περίπτωση - $(0-1$ Knapsack$)$

Η περίπτωση αυτή είναι δυσκολότερη από την προηγούμενημ παρ' όλο που το σύνολο των λύσεων είναι αισθητά μικρότερο απ' ότι προηγουμένως. Δεν είναι επίσης σαφές με ποιό τρόπο θα εφαρμοσθεί ο δυναμικός προγραμματισμός.

Θεωρούμε το σύνολο των αντικειμένων $\widehat{J} = j | j = 1, 2, \ldots, J$, καθώς επίσης όλες τις συλλογές αντικειμένων του $\widehat{J}$ που έχουν επιτρεπτό συνολικό βάρος, δηλαδή μικρότερο ή ίσο με M. Έστω δύο συλλογές S_1, S_2 με στοιχεία του $\widehat{J}$. Επίσης, έστω ότι το βάρος των στοιχείων της S_1 είναι μικρότερο ή ίσο αυτών της S_2, αλλά ταυτόχρονα η αξία των στοιχείων της πρώτης συλλογής S_1 είναι μεγαλύτερη (ή ίση) με αυτή της S_2. Προφανώς η δεύτερη συλλογή δεν προσθέτει τίποτα στην λύση του προβλήματος, εφόσον αποκλείεται να αποτελεί μέρος μιας βέλτιστης λύσης. Διότι αν αποτελούσε μέρος μιας λύσης, η αντικατάστασή της με την πρώτη συλλογή θα έδινε καλύτερο αποτέλεσμα: Συγκεκριμένα, έστω $j_1, j_2, \ldots, j_k \in \widehat{J}$, καθώς και $j'_1, j'_2, \ldots, j'_k \in \widehat{J}$ και επίσης

$$B_{j_1} + \ldots + B_{j_k} \leq B_{j'_1} + \ldots + B_{j'_k}$$

αλλά

$$W_{j_1} + \ldots + W_{j_k} \geq W_{j'_1} + \ldots + W_{j'_k}$$

Έστω μία συλλογή S που περιλαμβάνει τα $j'_1, \ldots, j'_k$ αφενός, καθώς και στοιχεία με δείκτες l *μεγαλύτερους* του J, $l_1, l_2, \ldots, l_m > J$. Τότε η συλλογή S' που περιλαμβάνει τα $j_1, j_2, \ldots, j_k$ αντί των $j'_1, \ldots, j'_k$, καθώς και τα $l_{1..m}$ είναι *(α)* εφικτή και *(β)* έχει μεγαλύτερη αξία από την S. (Επιβεβαιώστε το.)

Η παρατήρηση αυτή οδηγεί σε ένα αλγόριθμο μορφής δυναμικού προγραμματισμού, ο οποίος έχει ως εξής:

```
{ Τα m_j είναι σύνολα υποψηφίων λύσεων. Τα
  στοιχεία του m_j είναι ζεύγη (Βάρος, Αξία) }
• m_0 = (0,0)
• FOR j = 1 to N DO
  BEGIN {*}
    • N_j = M_j
    • FOR (B,A) ∈ M_{j-1}
      BEGIN{**}
        • ΕΣΤΩ ΤΟ (B + B_j, A + A_j)
        • AN (B + B_j ≤ M) ΠΡΟΣΘΕΣΕ ΤΟ
              (B + B_j, A + A_j) ΣΤΟ N_j
      END {**}
    • ΑΦΑΙΡΕΣΕ ΑΠΟ ΤΟ N_j ΣΤΟΙΧΕΙΑ (B_l, A_l)
      ΓΙΑ ΤΑ ΟΠΟΙΑ ΥΠΑΡΧΕΙ (B_k, A_k) ΜΕ
        (B_l ≥ B_k) ΚΑΙ (A_l ≤ A_k)
  END {*}
• ΕΠΙΛΕΞΕ ΤΟ ΣΤΟΙΧΕΙΟ ΤΟΥ m_N ΜΕ
  ΜΕΓΙΣΤΗ ΔΕΥΤΕΡΗ ΣΥΝΙΣΤΩΣΑ
```

Προφανώς το πρόβλημα με τον αλγόριθμο αυτόν είναι ο δυνητικά μεγάλος πληθικός αριθμός των M_j, όπως φαίνεται στο παρακάτω παράδειγμα.

<u>Παράδειγμα</u> Έστω 5 αντικείμενα

j	1	2	3	4	5
B	2	3	4	5	6
A	11	15	23	25	32

Ποιά η βέλτιστη φόρτιση για $M = 5$. Ο αλγόριθμος προχωρά ως εξής:

j	ΝΕΟ ΑΝΤΙΚΕΙΜΕΝΟ	m
0	–	$\{(0,0)\}$
1	$(2,11)$	$\{(0,0),(2,11)\}$
2	$(3,15)$	$\{(0,0),(2,11),(3,15),(5,26)\}$
3	$(4,23)$	$\{(0,0),(2,11),(3,15),(4,23),$ $(5,26),(6,34),(7,38),(9,49)\}$
4	$(5,25)$	$\{(0,0),(2,11),(3,15),(4,23),$ ~~(5,25)~~, $(5,26),(6,34)$, ~~(7,37)~~, $(7,38),(8,40),(9,49),(11,59),$ $(12,63),(14,73)\}$
5	$(6,32)$	$\{(0,0),(2,11),(3,15),(4,23),$ $(5,26)$, ~~(6,32)~~, $(6,34),(7,38)$ ~~(8,40)~~, $(8,43),(9,49)$, ~~(9,47)~~, $(10,55),(11,59)$, ~~(11,58)~~, ~~(12,63)~~, $(12,66),(13,70)$, ~~(14,72)~~, $(14,73),$ $(15,81)\}$

Η βέλτιστη φόρτιση για $M = 15$ είναι τα αντικείμενα 1,2,3,,5με αξία 81. Για $M = 12$ προκύπτει, εξετάζοντας το m_5, ότι η βέλτιστη αξία είναι 66 και η φόρτιση περιέχει τα αντικείμενα 1,2,5.

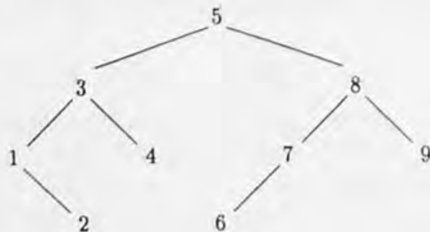
Άσκηση Υλοποιείστε τον αλγόριθμο "$0-1$ σακκίδιο" και επιπλέον επεκτείνατέ τον ώστε να υπολογίζει και τα αντικείμενα της βέλτιστης φόρτισης.

11.2. Βέλτιστο Δυαδικό Δένδρο Αναζήτησης

'Εστω $i = 1, \ldots, N$ *κλειδιά* εγγραφών σε μια δομή δεδομένων και p_i η πιθανότητα αναζήτησης του i. 'Ενα δυαδικό δένδρο αναζητήσης έχει την ιδιότητα:

Αν i είναι η ρίζα ενός υποδένδρου, το αριστερό υπόδενδρο του i έχει στοιχεία μικρότερα ή ίσα του i, ενώ το δεξί μεγαλύτερα ή ίσα.

Παράδειγμα



Προφανώς ο αριθμός των αναζητήσεων μέχρι την εύρεση του i είναι 1 συν το επίπεδο του i (η ρίζα θεωρείται στο επίπεδο 0). Ο μέσος αριθμός αναζητήσεων υπολογίζεται ως εξής:

Κλειδί	1	2	3	4	5	6	7	8	9
Πιθανότητα	0.15	0.10	0.10	0.15	0.10	0.10	0.10	0.05	0.05
Αναζητήσεις	3	4	2	3	1	4	3	2	3

$$\begin{aligned}\text{Αναμ. Τιμή (Αναζητήσεων)} &= 3\cdot 0,15 + 4\cdot 0,10 + 2\cdot 0,10 + 3\cdot 0,15+ \\ &\quad +1\cdot 0,10 + 4\cdot 0,20 + 3\cdot 0,10 + 2\cdot 0,05 \\ &\quad +3\cdot 0,05 = 2,95\end{aligned}$$

Πώς σχεδιάζουμε το δένδρο ώστε να ελαχιστοποιηθεί ο αναμενόμενος χρόνος αναζήτησης; Η σχεδίαση μπορεί να γίνει με δυναμικό προγραμματισμό:

Έστω k το στοιχείο-ρίζα. Αριστερά του είναι τα στοιχεία-κλειδιά $(1, 2, \ldots, k-1)$ και δεξιά του τα $(k+1, \ldots, n)$. Προφανώς τόσο το υπόδενδρο $(1, 2, \ldots, k-1)$, όσο και το $(k+1, \ldots, n)$ πρέπει να είναι βέλτιστα, διαφορετικά θα μπορούσε να επιτευχθεί καλύτερη συνολική σχεδίαση. Επίσης αν $c_{1,k-1}$ είναι ο αναμενόμενος χρόνος αναζήτησης στο υπόδενδρο $(1, \ldots, k-1)$ τότε με το να είναι τοποθετημένο ένα επίπεδο κάτω από την ρίζα, ο χρόνος είναι μεγαλύτερος κατά $p_1 + p_2 + \ldots + p_{k-1}$.

Έτσι έχουμε την σχέση δυναμικού προγραμματισμού:

$$c_{1,n} = \min_{1\le k\le n} \{p_1 + \ldots + p_{k-1} + c_{1,k-1} + p_k + p_{k+1} + \ldots + p_n + c_{k+1,n}\}$$

$$c_{1,n} = 1 + \min_{1\le k\le n} \{c_{1,k-1} + c_{k+1,n}\}$$

Για να λυθεί η σχέση αυτή θέτουμε

$$c_{i,j} = \sum_{l=i}^{j} p_l + \min_{i\le k\le j} \{c_{i,k-1} + c_{k+1,j}\}$$

με $c_{i,i} = p_i$, $c_{i,i-1} = c_{j+1,j}$.

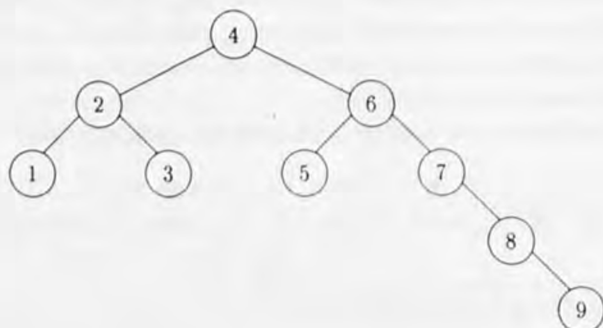
Η σχέση δυναμικού προγραμματισμού είναι παρόμοια με αυτή του πολλαπλασιασμού πινάκων και λύνεται ανάλογα.

<u>*Παράδειγμα (Συνέχεια)*</u>

Όπως και στην εφαρμογή των πινάκων ο υπολογισμός γίνεται για $j = i+k$ με $k = 0, 1, 2, \ldots$ κ.λπ.

$i \setminus j$	1	2	3	4	5	6	7	8	9
1	15	35_1	60_2	100_2	130_2	180_4	210_4	230_4	255_4
2		10	30_2	60_3	85_4	135_4	165_4	165_4	205_6
3			10	35_4	55_4	105_4	130_6	145_6	165_6
4				15	35_4	80_5	100_6	115_6	135_6
5					10	40_6	60_6	75_6	95_6
6						20	40_6	55_6	75_6
7							10	20_7	35_7
8								5	15_8
9									5

Τα αποτελέσματα δίνονται στον παραπάνω πίνακα, όπου τα στοιχεία είναι τα $c_{i,j}$ καθώς και το βέλτιστο k. Έτσι, από τον πίνακα φαίνεται ότι $c_{1,9} = 2.55$ (255/100) και η ρίζα είναι το 4. Το δεξί υποδένδρο (5,6,7,8,9) έχει ρίζα το 6, ενώ το αριστερό υποδένδρο (1,2,3) έχει ρίζα το 2. Συνεχίζοντας έτσι έχουμε ότι το βέλτιστο δένδρο είναι το



<u>**Άσκηση**</u> Σε μία βάση δεδομένων δίνονται n κλειδιά $\alpha_1, \alpha_2, \ldots, \alpha_n$. Οι πιθανότητες p_i αναζήτησης των α_i δίνονται στον παρακάτω πίνακα. Δίνονται επίσης οι πιθανότητες q_i να αναζητηθεί στοιχείο μεταξύ των α_i και α_{i+1}. Κατασκευάστε το βέλτιστο δένδρο για τα παρακάτω δεδομένα

		α_1	α_2	α_3	α_4	α_5	α_6	α_7
p_i	–	3	5	15	3	5	1	15
q_i	10	2	5	2	15	10	4	15

11.3. Πολλαπλασιασμός Πινάκων

Έστω πίνακες $A : m \times q$ $B : q \times n$ με στοιχεία $\{\alpha_{ij}\}$, $\{\beta_{ij}\}$. Το γινόμενο $A \cdot B$ ορίζεται ως $(AB)_{ij} = \sum_{k=1}^{q} \alpha_{ik}\beta_{kj}$

Προφανώς ο απλοϊκός αλγόριθμος πολλαπλασιασμού πινάκων απαιτεί $m \cdot q \cdot n$ πολλαπλασιασμούς αριθμού. Είναι δυνατές βελτιώσεις με τεχνικές διαίρει και βασίλευε (βλέπε σύγγραμμα MIT), που όμως δεν δίνουν ιδιαίτερα σημαντικά αποτελέσματα. Θεωρούμε ότι ο πολλαπλασιασμός απαιτεί $m \cdot q \cdot n$ πράξεις.

Το γινόμενο 3 ή περισσοτέρων πινάκων που έχουν συμβατές διαστάσεις δεν εξαρτάται από την σειρά που θα γίνουν οι πολλαπλασιασμοί των επιμέρους πινάκων. Με άλλα λόγια ισχύει η ιδιότης:

$$M_1 * (M_2 * M_3) = (M_1 * M_2) * M_3 \quad \text{ΠΡΟΣΕΤΑΙΡΙΣΤΙΚΗ}$$

<u>Άσκηση</u> Αποδείξτε το.

Η σειρά των επιμέρους πολλαπλασιασμών πινάκων έχει επίπτωση στον αριθμό των αριθμητικών πολλαπλασιασμών που απαιτούνται. Αυτό φαίνεται στο εξής παράδειγμα.

<u>*Παράδειγμα*</u> $M_1 : 3 \times 1$ $M_2 : 1 \times 100$ $M_3 : 100 \times 5$.

Για την εκτέλεση των πράξεων $M_1 * (M_2 * M_3)$ απαιτούνται

- $1 \cdot 100 \cdot 5 = 500$ πράξεις για τον πολλαπλασιασμό $M_2 * M_3$.
- $3 \cdot 1 \cdot 5 = 15$ πράξεις εφόσον η M_1 είναι $3*$ και η $(M_1 * M_3)$ είναι $1 * 5$,

δηλαδή συνολικά <u>515</u> πράξεις.

Όμως για την εκτέλεση των πράξεων

$$(M_1 * M_2) * M_3$$

απαιτούνται

- $3 \cdot 1 \cdot 100$ πράξεις για τον πολλαπλασιασμό $M_1 * M_2$.
- $3 \cdot 100 \cdot 5$ πράξεις για τον πολλαπλασιασμό $M_1 * M_2) * M_3$,

δηλαδή συνολικά $300 + 1500 =$ <u>1800</u> πράξεις.

Η βέλτιστη σειρά εκτέλεσης πράξεων υπολογίζεται με τη βοήθεια του δυναμικού προγραμματισμού ως εξής:

Έστω ότι έχουμε να υπολογίσουμε το $M_1 \cdot M_2 \cdot \ldots \cdot M_n$. Έστω $m(1,n)$ ο ελάχιστος αριθμός πολλαπλασιασμών που απαιτείται. Προφανώς ο τελευταίος πολλαπλασιασμός πινάκων για τον υπολογισμό του γινομένου γίνεται μεταξύ δύο υπο-γινομένων $\{M_1 \ldots M_k\} * \{M_{k+1} \ldots M_n\}$ όπου $k \geq 1$ και $k+1 \leq n$ ή $1 \leq k \leq n-1$. Προφανώς το υπογινόμενο $\{M_1 \ldots M_k\}$ έχει διάσταση $p_1 \times p_{k+1}$ - όπου δεχόμαστε ότι ο πίνακας $M_l : p_l \times p_{l+1}$, ενώ το $\{M_{k+1} \ldots M_n\}$ έχει διάσταση $p_{k+1} \times p_{n+1}$.

Επομένως, το κόστος του τελευταίου πολλαπλασιασμού είναι $p_1 \cdot p_{k+1} \cdot p_{n+1}$. Για να είναι η σειρά πολλαπλασιασμών βέλτιστη το υπογινόμενο $\{M_1 \ldots M_k\}$ απαιτεί $m(1,k)$ πράξεις, ενώ το $\{M_{k+1} \ldots M_n\}$ απαιτεί $m(k+1,n)$ πράξεις. Έτσι έχουμε τον δυναμικό προγραμματισμό:

$$m(1,n) = \min_{1 \leq k \leq n-1} (m(1,k) + m(k+1,n) + p_i p_{k+1} p_{n+1}$$

Γενικά

$$m(1,n) = \min_{1 \leq k \leq j-1} (m(1,k) + m(k+1,j) + p_i p_{k+1} p_{j+1}$$

και $m(i,i+1) = \phi$ για κάθε i.

<u>Μη αναδρομικά</u>, η σχέση αυτή επιλύεται υπολογίζοντας όχι τα $m(i,j)$ αλλά τα $m(i,i+k)$ για $i = 1, \ldots, N-2, \quad k = 2, \ldots, N-i$. Η σειρά υπολογισμού είναι κατά αύξοντα k, δηλαδή γράφουμε τον δυναμικό προγραμματισμό ως

$$m(i,i+k) = \min_{1 \leq l \leq k-1} \{m(i,i+l) + m(i+l+1,i+k) + p_i p_{i+l+1} p_{i+k+1}\}$$

$$m(i,i+1) = \phi$$

και έχουμε τον κώδικα:

```
for i:=1 to N-1 do m(i,i+1):=0;
  for k:=2 to N-1 do
    for i:=1 to N-k do
    begin
       m(i,i+k) ← ∞
       for l:=1 to k-1 do
       begin
          t ← m(i,i+l) + m(i+l+1, i+k) + pi pi+l+1 pi+k+1
          if m(1,1+k) > t then m(i,i+k) ← t
       end
    end
```

Ένας αναδρομικός αλγόριθμος γράφεται πολύ απλούστερα ως

```
function m(i,j)

  if i+1 ≥ j then m:=0
  else
  begin (* 1 *)
    m ← ∞
    for k:=i to j-1 do
    begin
      t ← m(i,k) + m(k+1,j) + p_i p_{k+1} p_{j+1}
      if m > t then m ← t
    end
  end (* 1 *)
```

<u>Άσκηση</u> Γιατί χρησιμοποιήθηκε η μεταβλητή t; Εναλλακτικά, γράψτε ένα κώδικα χωρίς την προσωρινή μεταβλητή t και σχολιάστε την απόδοσή του.

Ο αναδρομικός κώδικας έχει το μειονέκτημα ότι θα κάνει πολλές περιττές πράξεις, καθώς κάθε $m(i,j)$ θα υπολογισθεί πολλές φορές εφόσον η τιμή του $m(i,j)$ δεν αποθηκεύεται πουθενά. Έτσι η απόδοση του αναδρομικού αλγορίθμου βελτιώνεται σημαντικά αν μπορεί να θυμάται πότε έχει τελειώσει ένας επιμέρους υπολογισμό. Αυτό γίνεται στο παρακάτω παράδειγμα

```
ΚΥΡΙΟ ΠΡΟΓΡΑΜΜΑ

   q(i,j) ← ∞           q(i,i+1) ← 0
   {compute} m(i,j)

   function m(i,j)

     if q(i,j) < ∞ then m ← q(i,j)
     else
     begin (* 1 *)
     { Ίδιος κώδικας όπως προηγουμένως
       από το begin (* 1 *) ως το end (* 1 *)}
     end (* 1 *)
```

Το παραπάνω τέχνασμα της αποθήκευσης των τιμών του m (στο q) λέγεται <u>*memoization*</u> (από το memorization: "αποθήκευση"). Το ίδιο τέχνασμα μπορεί να εφαρμοσθεί για την βελτίωση πολλών αναδρομικών κωδίκων δυναμικού προγραμματισμού.

11.4. Πρόβλημα Ελάχιστης Διαδρομής Αλγόριθμος Bellman-Ford

Έστω γράφημα με αποστάσεις $c_{i,j}$ επί των ακμών. Θέλουμε να υπολογίσουμε τις διαδρομές ελαχίστου μήκους προς δεδομένο τελικό κόμβο t. Έστω επίσης ότι δεν υπάρχει κύκλος αρνητικού μήκους στο γράφημα και $c_{ii} = 0$.

Έστω $d(i|k)$ η ελάχιστη απόσταση $i \to t$ για διαδρομές που περιέχουν k ή λιγότερες πλευρές. Η σχέση δυναμικού προγραμματισμού είναι

$$d(i|k+1) = \min_{j \in \Gamma(i)} \{c_{ij} + d(j|k)\}$$

όπου $\Gamma(i) = \{j \in V \,|\, \text{Υπάρχει πλευρά } i \to j \text{ ή } j = i\}$.

Επίσης είναι $d(i|0) = \begin{cases} \infty & i \neq t \\ 0 & i = t \end{cases}$

Προφανώς εφόσον δεν υπάρχουν κύκλοι αρνητικού μήκους ισχύει

$$d(i|N) = \text{μήκος ελάχιστης διαδρομής } i \to t$$

αν το N είναι ίσο ή μεγαλύτερο του αριθμού των πλευρών, εφόσον οποιαδήποτε μη περιέχουσα κύκλο διαδρομή περιέχει το πολύ όλες τις πλευρές του γραφήματος. Άρα η λύση του δυναμικού προγραμματισμού δίνει την λύση για *όλες* τις κορυφές σε $|E|$ βήματα ($|E|$: αριθμός πλευρών).

Συμβολίζουμε τώρα τον αριθμό τον αριθμό $d(i|k)$ με d_i^k και με d^k το διάνυσμα

$$d^k = (d_1^k, d_2^k, \ldots, d_n^k) \text{ με } n = |V| : \text{ κορυφές}$$

Τότε η σχέση δυναμικού προγραμματισμού γράφεται σε διανυσματική μορφή ως

$$d^{k+1} = (d_1^{k+1}, d_2^{k+1}, \ldots) = \left(\min_j(c_{1j} + d_j^k), \min_j(c_{2j} + d_j^k), \ldots\right)$$

Το τελευταίο σκέλος μπορεί να θεωρηθεί σαν ένας "μηχανισμός" όπου δεδομένου ενός διανύσματος d^k παράγει ένα καινούργιο διάνυσμα με συντεταγμένες $\min_j(c_{ij} + d_j^k)$ κ.λπ. Έχουμε δηλαδή μία διανυσματική συνάρτηση

$F : R_v \to R_v$ (τα c θεωρούνται σταθερά). Με αυτή την έννοια ο δυναμικός προγραμματισμός είναι μία εξίσωση διαφορών

$$\begin{aligned} d^{k+1} &= F(d^k) \\ d^0 &= (\infty, \infty, \ldots, 0) \end{aligned}$$

$^{\uparrow} k -$ συνιστώσα

έτσι ώστε σε $|E|$ βήματα να έχουμε

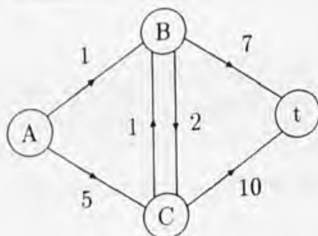
$$d^{|E|} = F(d^{|E|})$$

Άρα ο δυναμικός προγραμματισμός μπορεί να θεωρηθεί σαν μέθοδος επίλυσης της εξίσωσης

$$\mu = F(\mu).$$

Παραδείγματα

1.



Τα βήματα του αλγορίθμου είναι:

$$d^0 = (d_A^0, d_B^0, d_C^0, d_t^0) = (\infty, \infty, \infty, 0)$$

$$d^1 = (\infty, 7, 10, 0) \qquad d^2 = (8, 7, 8, 0)$$

$$d^3 = (8, 7, 8, 0)$$

Ο αλγόριθμος σταματά εφόσον $d^2 = d^3$ εφόσον $d^3 = F(d^2) = F(d^3)$ και άρα το $d^3 = d^2$ ικανοποιεί την σχέση δυναμικού προγραμματισμού.

2. Έστω $\mu = (\mu_A, \mu_B, \mu_C \mu_t) = (4, 2, 3, 1)$ τότε $F(\mu) = (\min\,(1 + 2, 5 + 2)$, $\min\,(2 + 3, 7 + 1), \min\,(1 + 2, 10 + 1), \min\,(0 + 1)) = (3, 5, 3, 1)$.

(Σημειώσεις στο περιθώριο: Ⓐ, Ⓑ, Ⓒ, ⓣ)

Έστω δύο διανύσματα μ, λ. Γράφουμε $\mu \geq \lambda$ αν κάθε συνιστώσα του μ είναι μεγαλύτερη ή ίση αυτής του λ. Θα είναι προφανώς $F(\mu) \geq F(\lambda)$, και επίσης η F είναι "συνεχής" συνάρτηση.

Θα δείξουμε ότι η αναδρομή $d^{n+1} = F(d^n)$ οδηγεί σε λύση της εξίσωσης δυναμικού προγραμματισμού.

Αν θέσουμε $d^0 = (\infty, \infty, \ldots, \infty, 0)$, ισχύει προφανώς ότι $F(d^0) \leq (\infty, \infty, \ldots, \infty, 0)$ ή

$$d^1 = F(d^0) \leq (\infty, \ldots, \infty, 0) = d^0$$

ή τελικά $d^1 \leq d^0$ άρα και $F(d^1) \leq F(d^0)$. Επίσης $d^2 = F(d^1) \leq F(d^0) = d^1$. Γενικά $d^n \leq d^{n-1}$ και επίσης $d_n \geq 0$. Άρα υπάρχει όριο $\lim_{n\to\infty} d^n = d$. Λόγω της συνέχειας της F θα είναι

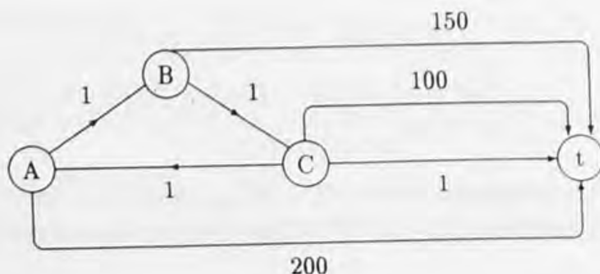
$$\begin{aligned} \lim d^n &= \lim F(d^{n-1}) \\ &= F(\lim d^{n-1}) \end{aligned}$$

ή

$$d = F(d)$$

Τα παραπάνω δείχνουν ότι ο αλγόριθμος συγκλίνει αν το αρχικό σημείο d^0 είναι "μεγάλο". Σε διάφορες όμως εφαρμογές το d^0 είναι αυθαίρετο, και γεννάται το ερώτημα αν συγκλίνει ο αλγόριθμος.

Παράδειγμα Σε packet switching network έχουμε τους εξής κλάδους και κόμβους:



Ο αλγόριθμος B-Ford μπορεί να εφαρμοσθεί αποκεντρωμένα: Ανά χρόνο ΔΤ, κάθε κόμβος διαβιβάζει στους γειτονικούς του την τρέχουσα εκτίμησή του για το χρόνο που χρειάζεται ένα πακέτο του να φθάσειστο t. Κάθε κόμβος αναθεωρεί (στιγμιαία) τις εκτιμήσεις του και δρομολογεί για το διάστημα ΔΤ

αυτό σύμφωνα με τον αλγόριθμο B-Ford:

Χρόνος	A	B	C	t
$0 \cdot \Delta T$	∞	∞	∞	0
$1 \cdot \Delta T$	200^t	150^t	1^t	0
$2 \cdot \Delta T$	151^B	2^C	1^t	0
$3 \cdot \Delta T$	3^B	2^C	1^t	ϕ

Στον πίνακα σημειώνεται η τιμή του d και άνω δεξιά η δρομολόγηση.

Έστω τώρα ότι ο ταχύς δίαυλος $C \to t$ παθαίνει βλάβη, αλλά εξακολουθεί να είναι ενεργός ο βραδύς δίαυλος. Ο αλγόριθμος B-Ford προχωρά τώρα με αρχική τιμή (3,2,1,0) και όχι $(\infty, \infty, \infty, 0)$! Τα βήματα θα είναι τα εξής:

Καινούργιος Χρόνος				
$0 \cdot \Delta T$	3^B	2^C	1^t	ϕ
$1 \cdot \Delta T$	3^B	2^C	4^A	ϕ

<u>ΣΧΟΛΙΟ:</u> Ο C ΑΓΝΟΕΙ ΤΟΝ ΚΛΑΔΟ ΠΡΟΣ t ΚΑΙ ΕΠΙΛΕΓΕΙ ΤΗΝ ΑΠΟΣΤΟΛΗ ΠΑΚΕΤΩΝ ΠΡΟΣ A, ΟΠΟΥ ΟΜΩΣ ΔΕΝ ΕΧΕΙ ΓΙΝΕΙ ΑΝΤΙΛΗΠΤΗ ΑΚΟΜΑ Η ΒΛΑΒΗ!

$2 \cdot \Delta T$	3^B	5^C	4^A	ϕ
$3 \cdot \Delta T$	6^B	5^C	4^A	ϕ
$4 \cdot \Delta T$	6^B	5^C	4^A	ϕ
$5 \cdot \Delta T$	6^B	8^C	4^A	ϕ.

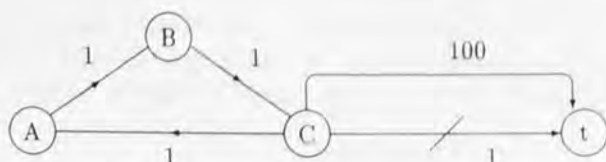
Προφανώς είναι $d_A^n = d_B^{n-1} + 1$, $\quad d_B^n = d_C^{n-1} + 1$ και $d_C^n = d_A^{n-1} + 1$ ή $d_A^{n+3} = d_A^n + 3$ ή τελικά $d_A^k \approx k$, πράγμα που δείχνει ότι θα υπάρξει μεγάλη καθυστέρηση στην σύγκλιση του αλγορίθμου $B-F$ στην ορθή τιμή 102,101,100. Επίσης όλο αυτό το διάστημα κανένα πακέτο δεν θα δρομολογηθεί προς το t αλλά θα ανακυκλώνουν μεταξύ A, B, C!

* <u>ΠΡΟΑΙΡΕΤΙΚΗ ΥΛΗ</u> *

Μια διαδικασία που βελτιώνει την απόδοση του Bellman-Ford σε δυναμικές καταστάσεις λέγεται <u>hold-down</u>. Σε αυτή, ένας κόμβος που κατά την υλοποίηση του $B-F$ παρατηρεί ότι πρέπει να κάνει αλλαγή δρομολόγησης, εξακολυθεί για k χρονικά βήματα να χρησιμοποιεί την παλαιά δρομολόγηση, και αναφέρει στους γειτονικούς τις εκτιμήσεις χρόνου με βάση την παλαιά

δρομολόγηση. Αυτό σημαίνει ότι για k βήματα ο κόμβος ενημερώνει για τυχόν βλάβες-αλλαγές που παρατήρησε στους κλάδους που πηγάζουν από αυτόν. Προφανώς αν το k είναι μεγάλο, η δρομολόγηση είναι ίσως κακή για μεγάλο διάστημα. Αν όμως το k είναι μικρό, ενδέχεται ο κόμβος να μην "ενημέρωσε" το υπόλοιπο δίκτυο για τις βλάβες που παρατήρησε αρκετά έντονα.

<u>*Παράδειγμα*</u>



Το hold-down διαρκεί 3 βήματα. Αρχικά τα d είναι (3,2,1,0) όταν χαλάει ο ταχύς δίαυλος $C \to t$. Τα βήματα είναι:

Βήμα	A	B	C	
$0 \cdot \Delta T$	3^B	2^C	1^t	
				$-Βλάβη$
$1 \cdot \Delta T$	3^B	2^C	100^t	$HOLD1$
$2 \cdot \Delta T$	3^B	101^C	100^t	$HOLD2$
$3 \cdot \Delta T$	102^B	101^C	100^t	$HOLD3$
$4 \cdot \Delta T$	102^B	101^C	100^t	

Παρατηρούμε ότι στο βήμα $1 \cdot \Delta T$ ο κόμβος C θεωρεί ότι θα ήταν καλύτερα να στείλει μέσω A, εφόσον ο χρόνος είναι 1+3=4, αλλά λόγω hold down εξακολουθεί να στελνει προς τ μέσω του ενεργού διαύλου και παρατηρεί χρόνο 100, που διαβιβάζει στον γειτονικό κόμβο B (και A). Στο βήμα $3 \cdot \Delta T$ ο χρόνος του A έχει γίνει 102. Αν όμως δεν υπήρχε hold down για 3 βήματα αλλά μόνο για 2, ο κόμβος C θα είχε αναθεωρήσει τη δρομολόγηση και θα είχε στείλει τα πακέτα προς A. Έτσι ο αλγόριθμος θα είχε δώσει αποτελέσματα:

	$\underline{A}$	$\underline{B}$	$\underline{C}$	
$2 \cdot \Delta T$	3^B	101^C	100^t	$HOLD2 - ΤΕΛΕΥΤΑΙΟ$
$3 \cdot \Delta T$	102^B	101^C	4^A	$(ΑΛΛΑΓΗ...)$
$4 \cdot \Delta T$	102^B	5^C	103^A	$HOLD1$
$5 \cdot \Delta T$	6^B	104^C	103^A	$HOLD2$
$6 \cdot \Delta T$	105^B	104^C	7^A	
$7 \cdot \Delta T$	105^B	8^C	106^A	$HOLD1$
$8 \cdot \Delta T$	9^B	107^C	106^A	$HOLD2$
$9 \cdot \Delta T$	108^B	107^C	10^A	

Βλέπουμε έτσι ώστε το hold down 2 βημάτων δεν θα είχε δώσει αποτελέσματα.

Ασκήσεις Αποδείξτε ότι η F είναι αύξουσα, δηλαδή $\mu \geq \lambda \Rightarrow F(\mu) \geq F(\lambda)$ και συνεχής, δηλαδή αν

$$\mu^\nu \to \mu \text{ τότε } F(\mu^n) \to F(\mu).$$

Βιβλιογραφία

[1] Φ. Αφράτη, Γ. Παπαγεωργίου *Αλγόριθμοι: Μέθοδοι Σχεδίασης και Ανάλυση Πολυπλοκότητας* Συμμετρία 1993.

[2] S. Baase *Computer Algorithms. Introduction to Design and Analysis* Addison Wesley, 1988.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest *Introduction to algorithms*, MIT Press, 1990.

[4] M. R. Garey, D. S. Johnson *Computers and Intractability. A Guide to the Theory of NP-Completeness* Freeman, 1979.

[5] Θ.Ζ. Καλαμπούκη *Δομές Δεδομένων με Pascal*, 1989.

[6] H. R. Lewis, C. H. Papadimitriou *Elements of the Theory of Computation*, Prentice Hall 1981 (Εχει εκδοθεί σε ελληνική μετάφραση από το ΤΕΕ με τον τίτλο *Στοιχεία Θεωρίας Υπολογισμού*)

[7] D.E. Knuth *Fundamental Algorithms*, Vol 1, Addison Wesley, 1968.

[8] U. Manber *Introduction to algorithms*, Addison Wesley, 1989.

[9] C. H. Papadimitriou *Computational Complexity*, Addison Wesley 1993.

[10] R. Sedgewick *Algorithms in C*, Addison Wesley, 1990.

[11] S. S. Skiena *The Algorithm Design Manual*, Springer Verlag, 1998.

[12] D. Wood *Data Structures, Algorithms, and Performance*, Addison Wesley, 1993.