352. ΔΟΜΕΣ ΔΕΔΟΜΈΝΩΝ

Σημειώσεις

Εαρινό Εξάμηνο 2024

Γιάννης Λιβιεράτος
Επιμέλεια: Σπύρος Ματιάτος
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Σχολή Θετικών Επιστημών, Τμήμα Μαθηματικών

Τελευταία ανανέωση: 13 Μαΐου 2024
Τελευταία επιμέλεια: 27 Μαΐου 2024

Contents

0 Εισαγωγή

Οι "Δομές Δεδομένων" (Data Structures) είναι ο κλάδος της (θεωρητικής) πληροφορικής που ασχολείται με την οργάνωση των δεδομένων ώστε να γίνει αποτελεσματική υπολογιστικά η πρόσβαση σε αυτά. Ως πρότυπο παράδειγμα (paradigm) έχουμε ένα πρόγραμμα που επιλύει κάποιο υπολογιστικό πρόβλημα, το οποίο και χρειάζεται να επεξεργαστεί έναν μεγάλο όγκο δεδομένων, που είναι καταχωρημένα με κάποιον προσβάσιμο τρόπο. Οι βασικές λειτουργίες που μας ενδιαφέρουν να γίνονται αποδοτικά είναι:

- Αρχικοποίηση δομής: αυτό εν γένει θα γίνεται από μία εντολή και δεν θα ενδιαφερόμαστε σε μεγαλύτερο βάθος για το πως μια γλώσσα προγραμματισμού ή ένα υπολογιστικό σύστημα την πραγματοποιεί.
- Έλεγχος του αν μία δομή δεδομένων είναι κενή.
- Προσπέλαση δομής: αυτή είναι μια πρότυπη λειτουργία που συνήθως θα συνδυάζεται με κάποια άλλη (π.χ. έλεγχος για το αν όλα τα δεδομένα στη δομή πληρούν κάποια κριτήρια ή εύρεση μεγίστου/ελαχίστου).
- Προσθήκη στοιχείου στην δομή: συνήθως θα μας απασχολεί η λειτουργία αυτή να διατηρεί την βασική ιδιότητα της δομής. Αλλιώς θα γίνεται αποδοτικά κατά τετριμμένο τρόπο.
- Διαγραφή στοιχείου από την δομή: συνήθως θα μας απασχολεί η λειτουργία αυτή να διατηρεί την βασική ιδιότητα της δομής. Αλλιώς θα γίνεται αποδοτικά κατά τετριμμένο τρόπο.
- Αναζήτηση στοιχείου στην δομή.

Μας απασχολεί κατά βάση η θεωρητική ανάλυση του πως τα δεδομένα μπορούν να οργανωθούν ώστε οι παραπάνω λειτουργίες να γίνονται αποτελεσματικά/αποδοτικά και όχι η εσωτερική λειτουργία ενός υπολογιστικού συστήματος (ή μιας υπολογιστικής μηχανής) και τον τρόπο που υλοποιεί τις δομές αυτές. Θα μας απασχολήσουν και δομές που κάνουν δυνατή ή/και αποδοτική την χρήση συγκεκριμένων αλγορίθμων για την επίλυση γνωστών προβλημάτων.

Εντελώς αφηρημένα, ένας αλγόριθμος είναι μία πεπερασμένη ακολουθία αυστηρά καθορισμένων (χωρίς αμφισημία) εντολών. Η αποδοτικότητά τους είναι η κατανάλωση πόρων (συνήθως χρόνος και μνήμη) που θα χρειαστούν για την διεκπεραίωση της εργασίας για την οποία χρησιμεύουν. Χρειάζεται πρώτα να αποκτήσουμε μια ιδέα του τι είδους εντολές μπορούν να εκτελεστούν από μία υπολογιστική μηχανή, ποιες από αυτές είναι στοιχειώδεις και με ποιους τρόπους μπορούμε να αναφερόμαστε σε αυτές, ώστε να μπορέσουμε να συγκεκριμενοποιήσουμε και την έννοια της αποδοτικότητας ή και της αποτελεσματικότητας των αλγορίθμων και των δομών που θα χρησιμοποιήσουμε. Αναλυτικές σημειώσεις για τα θέματα που θα ασχοληθούμε: [11].

0.1 Μνήμη

Η λεγόμενη αρχιτεκτονική του υπολογιστών, που βασίζεται σε μεγάλο βαθμό στο μοντέλο Von Neumann [7], δεν θα μας απασχολήσει. Θα θεωρούμε ότι ο υπολογιστής διαθέτει μία σειρά από θέσεις μνήμης, δηλαδή *διευθύνσεις*, στις οποίες αποθηκεύονται δεδομένα. Τα δεδομένα αυτά μπορούν να είναι διαφόρων ειδών: χαρακτήρες, ακέραιοι αριθμοί, δεκαδικοί, δυαδικοί, δείκτες, κόμβοι γραφημάτων κ.α. Θα υποθέτουμε ότι σε κάθε θέση μνήμης μπορεί να αποθηκευτεί οποιουδήποτε είδους δεδομένο, το οποίο εν γένει δεν είναι σωστό, καθώς κάθε τύπος δεδομένων καταλαμβάνει εν γένει μια σειρά από συνεχόμενες θέσεις μνήμης για να αποθηκευτεί. Π.χ. ένας χαρακτήρας καταλαμβάνει συχνά μία θέση μνήμης, ενώ ένας ακέραιος τέσσερις.

Η αμφισημία που προκύπτει είναι μη τετριμμένη. Για παράδειγμα, θα λέγαμε διαισθητικά ότι ένα διάνυσμα τριών πραγματικών αριθμών χρειάζεται τρεις συνεχόμενες από αυτές τις "υπερ-χωρητικές" θέσεις μνήμης που έχουμε στην διάθεσή μας. Όμως, θα δούμε ότι ένας κόμβος ενός γραφήματος, που περιέχει δύο ή και τρία διαφορετικά στοιχεία διαφορετικού κιόλας είδους (τιμή, δείκτες), θα καταλαμβάνει μία θέση μνήμης.

Ο γενικός κανόνας είναι ο εξής: αν ένα δεδομένο έχει σταθερό μέγεθος, θα μπορεί να αποθηκευτεί σε μία θέση μνήμης. Παρ' ότι αυτός ο κανόνας μπορεί να ξεχειλώσει εύκολα, έχει δύο προτερήματα. Αφενός μας επιτρέπει να ασχοληθούμε με ζητήματα αποθήκευσης δεδομένων στο επίπεδο αφαίρεσης που θέλουμε. Αφετέρου, στην πράξη έχει νόημα. Η εσωτερική λειτουργία του υπολογιστή επιτρέπει την αναφορά σε ένα αντικείμενο γνωρίζοντας απλώς την διεύθυνση της θέση μνήμης από την οποία ξεκινάει η αποθήκευσή του και τον τύπο του, δηλαδή πόσες συνεχόμενες θέσεις μνήμης θα δεσμεύσει ο υπολογιστής για να το καταχωρήσει. Θεωρούμε αυτή την εσωτερική λειτουργία ως "μαύρο κουτί", που εκτελείται πάντοτε σωστά από μια υπολογιστική μηχανή, χωρίς την επίβλεψη/επέμβασή μας.

0.2 Αλγόριθμοι και Ψευδογλώσσα

Για να εκτελέσει μία υπολογιστική μηχανή μία οποιαδήποτε λειτουργία, χρειάζεται να λάβει τις κατάλληλες εντολές σε αυτό που ονομάζουμε *γλώσσα μηχανής*, σε σειρά δηλαδή από δυαδικά ψηφία 0 και 1. Το να γράψουμε και να διαβάσουμε σε τέτοια γλώσσα, θα ήταν εξοντωτικό. Ένα πρώτο βήμα για γίνει πιο προσιτός σε εμάς ο σχεδιασμός ενός αλγορίθμου, είναι οι λεγόμενες *συμβολικές γλώσσες* (assembly languages) [6]. Αυτές οι γλώσσες, παρ' ότι όντως βοηθούν στην κωδικοποίηση των δυαδικών συμβολοσειρών με τρόπο που κάνουν αρκετά πιο κατανοητό το πως λειτουργεί ένα πρόγραμμα, παραμένουν αρκετά δύσχρηστες, κι έχουν και το βασικό μειονέκτημα ότι εξαρτώνται από τον εκάστοτε επεξεργαστή και την αρχιτεκτονική του.

Ο βασικός τρόπος που η πληροφορική διαδόθηκε και εξελίχθηκε σε τέτοιο βαθμό, είναι οι *γλώσσες προγραμματισμού* [5, 2]. Υπάρχει ένα τεράστιο πλήθος γλωσσών προγραμματισμού με το οποίο μπορούν να υλοποιηθούν αλγόριθμοι [10], με διάφορα επίπεδα ευκολίας χρήσης. Ακόμη κι αυτό το επίπεδο λεπτομέρειας όμως, θα είναι εν γένει πέρα από το ενδιαφέρον μας.

Ας δούμε ένα από τα κλασσικότερα παραδείγματα αλγορίθμου: την εύρεση μέγιστου και ελάχιστου στοιχείου σε πίνακα. Ας δούμε σε τι επίπεδο θέλουμε να διατυπώνονται οι αλγόριθμοί μας. Αρχικά, ως *υπολογιστικό βήμα* θα θεωρούμε μία στοιχειώδη πράξη: πρόσθεση/πολλαπλασιασμός δύο αριθμών, διάβασμα μίας θέσης ενός πίνακα, έλεγχος για το αν η τιμή μίας μεταβλητής είναι ίση/μικρότερη/μεγαλύτερη από μια άλλη κτλ (αυτό όπως θα φαντάζεται πλέον ο/η αναγνώστης/ρια είναι πάλι μια απλοποιητική παραδοχή). Εντολές που αποτελούνται από ένα πλήθος τέτοιων βημάτων, δεν θα είναι αποδεκτές αν δεν τις έχουμε ορίσει ήδη. Ως εκ τούτου, δεν θα αποδεχόμασταν ως λύση την φαινομενικά σαφή εντολή "βρες το μέγιστο και το ελάχιστο στοιχείο του πίνακα".

Αυτό ίσως φανεί λίγο παράξενο, δεδομένου ότι σε γλώσσες προγραμματισμού όπως η `python`, οι εντολές $\max(A)$, $\min(A)$ αποτελούν τον μάλλον αποδοτικότερο τρόπο να υπολογιστεί το μέγιστο και το ελάχιστο στοιχείο ενός πίνακα A οποιουδήποτε μεγέθους. Οι εντολές αυτές είναι στην πραγματικότητα συναρτήσεις, προγράμματα δηλαδή που είναι ήδη υλοποιημένα και βελτιστοποιημένα ως προς την απόδοσή τους στις βιβλιοθήκες που χρησιμοποιεί η γλώσσα αυτή.

Επανερχόμαστε στην λύση του προβλήματος. Ένας αποδεκτός αλγόριθμος, γραμμένος σε φυσική γλώσσα, θα μπορούσε να είναι ο παρακάτω:

1. Θέσε το στοιχείο στην πρώτη θέση του πίνακα ως το μέγιστο.
2. Προσπέλασε σειριακά τον πίνακα. Κάθε στοιχείο που διαβάζεις, έλεγξε αν είναι μεγαλύτερο του μέγιστου. Αν ναι, ενημέρωσε την τιμή του μέγιστου.
3. Κάνε το αντίστοιχο για το ελάχιστο.

Οι εντολές αυτές περιγράφουν τον κλασσικό αλγόριθμο εύρεσης μεγίστου και ελαχίστου. Υπάρχουν όμως δύο προφανή ζητήματα που προκύπτουν: η εντολή 2 περιλαμβάνει μια διαδικασία ελέγχου και μια εκχώρηση τιμής, που θα εκτελεστεί τόσες φορές όσο και το μέγεθος του πίνακα. Οπότε, ακόμη κι αν προσδιορίζαμε με τον ίδιο τρόπο την λέξη "αντίστοιχο" της εντολής 3, θα συνεχίζαμε να έχουμε "θολώσει κάπως τα νερά" ως προς τα υπολογιστικά βήματα που χρειάζονται.

Ας σταθεροποιήσουμε μία (ψευδο-)γλώσσα που θα χρησιμοποιούμε. Στόχος μας είναι να πετύχουμε την επιθυμητή αυστηρότητα ως προς τις έννοιες που χρησιμοποιούμε, κι όχι να περιοριστούμε σε κάποιο δύσκαμπτο συμβολισμό. Ως εκ τούτου, οποτεδήποτε χρειαζόμαστε επέκταση ή αλλαγή στην γλώσσα για να διευκολύνουμε την περιγραφή κάποιου αλγορίθμου, θα το κάνουμε. Εν γένει, υποθέτουμε ότι η είσοδος/έξοδος ενός προγράμματος, προέρχεται και επιστρέφεται σε κάποια υπολογιστική μηχανή (ανθρώπινη ή μη), που δεν λειτουργεί ως αντίπαλος (adversary). Ως εκ τούτου όταν γράφουμε αλγορίθμους, θα υποθέτουμε αφενός ότι η είσοδός τους μας παρέχεται στην κατάλληλη μορφή, αλλά και ότι η έξοδος τους είναι προσπελάσιμη από την εκάστοτε υπολογιστική μηχανή.

- Με **Algorithm**$(\texttt{args}_1, \ldots, \texttt{args}_n)$ θα δηλώνουμε δύο πράγματα: (ι) τον αλγόριθμο που έχουμε ονομάσει **Algorithm** που παίρνει ως είσοδο τα/τις $n \in \mathbb{N}$ ορίσματα/παραμέτρους $\texttt{args}_i$ και (ιι) την έξοδο του αλγορίθμου αφού τρέξει πάνω σε μια συγκεκριμένη είσοδο $(\texttt{args}_1, \ldots, \texttt{args}_n)$. Τόσο η είσοδος, όσο και η έξοδος ενός αλγορίθμου, μπορεί να είναι διαφόρων ειδών: χαρακτήρες, λέξεις, αριθμοί, πίνακες, γραφήματα.
- Με (συνήθως) αγγλικά γράμματα, θα συμβολίζουμε μεταβλητές που εμφανίζονται στους αλγορίθμους μας. Σπανίως θα χρησιμοποιούμε και την γραφή $<$ `placeholder` $>$ αντί για σκέτο όνομα μεταβλητής. Ακόμη, όταν χρειάζεται για να διευκολύνεται το διάβασμα του ψευδοκώδικα, θα χρησιμοποιούμε και ολόκληρες λέξεις που θα δηλώνουν τις επιθυμητές τιμές που θα λαμβάνουν.
- Στην περίπτωση που θέλουμε ένα γράμμα ή μια λέξη να είναι *σταθερά*, θα το προσδιορίζουμε ρητώς ή/και θα την γράφουμε με κεφαλαία γράμματα.

- Με την εντολή **return** $<$ placeholder $>$, ο αλγόριθμος θα τερματίζει την λειτουργία του και θα επιστρέφει την τιμή που είναι αποθηκευμένη στην μεταβλητή $<$ placeholder $>$.
- Με την εντολή **print** $<$ placeholder $>$, ο αλγόριθμος θα επιστρέφει την τιμή που είναι αποθηκευμένη στην μεταβλητή $<$ placeholder $>$, χωρίς όμως να τερματίζει την λειτουργία του.
- Θα χρησιμοποιούμε όλα τα γνωστά διαθέσιμα σύμβολα για τις συνήθεις πράξεις ($+, -, \ldots$) και συγκρίσεις ($\leq, >, \ldots$). Οποιοδήποτε μη τυπικό συμβολισμό χρησιμοποιήσουμε, θα τον ορίζουμε επί τόπου.
- Θα χρησιμοποιούμε την $=$ και για εκχώρηση και για σύγκριση. Οι συνήθεις εναλλακτικοί τρόποι συμβολισμού, με $\leftarrow$ για εκχώρηση και $==$ για σύγκριση, θα χρησιμοποιηθούν σε επιμέρους μόνο περιπτώσεις.
- Θα χρησιμοποιούμε παρενθέσεις, αγκύλες και άγκιστρα για να δηλώνουμε το πλαίσιο αναφοράς ενός αλγορίθμου, συνάρτησης ή πράξεις, καθώς και για την προτεραιότητα πράξεων.
- Με **if** $<$ cond$_1$ $>$ **then** $<$ instr$_1$ $>$ **else if** $<$ cond$_2$ $>$ **then** $<$ instr$_2$ $>$ $\cdots$ **else** $<$ instr$_n$ $>$ θα εκτελούμε το σύνολο εντολών $<$ instr$_i$ $>$ που αντιστοιχεί στην συνθήκη $<$ cond$_i$ $>$ που ισχύει, $i = 1, \ldots, n$. Στην περίπτωση που υπάρχει **else**, η συνθήκη $<$ cond$_n$ $>$ είναι η σύζευξη των αρνήσεων όλων των υπολοίπων.
- Με **while** $<$ cond $>$ **do** $<$ instr $>$, θα εκτελούμε το σύνολο εντολών $<$ instr $>$ όσο συνεχίζει να ισχύει η συνθήκη $<$ cond $>$. Αντίστοιχα θα χρησιμοποιούμε και όποια άλλη επαναληπτική δομή θέλουμε: **do**–**while**, **do**–**until**, **for**–**do**.
- Θα χρησιμοποιούμε την **end** για να δηλώνουμε το τέλος ενός μπλοκ εντολών που ελέγχουν συνθήκες ή εκτελούν επαναληπτικές διαδικασίες.
- Με % ή #, θα δηλώνουμε μια γραμμή σχολίων. Μια γραμμή δηλαδή που ο αλγόριθμός μας θα αγνοεί.

Ας δούμε ως πρώτο παράδειγμα τον αλγόριθμο εύρεσης μεγίστου-ελαχίστου που αναφέραμε παραπάνω.

Algorithm 1 MaxMin(A)

```
# A: πίνακας n θέσεων με τιμές πραγματικούς αριθμούς.
# A[i]: η τιμή του A στην θέση i.
 1: max = A[1]
 2: i = 1
 3: while i ≤ n do
 4:    if max < A[i] then
 5:       max = A[i]
 6:    end if
 7:    i = i + 1
 8: end while
 9: min = A[1]
10: i = 1
11: while i ≤ n do
12:    if min > A[i] then
13:       min = A[i]
14:    end if
15:    i = i + 1
16: end while
17: return max, min
```

Μπορούμε να υποθέσουμε ότι τα max, min που επιστρέφει ο αλγόριθμος στην γραμμή 17 είναι σε μορφή διανύσματος ή πίνακα δύο θέσεων. Παρατηρούμε επίσης ότι ο δείκτης i παίρνει διαδοχικά τις τιμές $1, 2, \ldots, n, 1, 2, \ldots, n$. Αυτό δεν μας δημιουργεί κάποιο ζήτημα. Το ίδιο θα ίσχυε κι αν μετά άλλαζε κι ο τύπος της τιμής που έπαιρνε (π.χ. σε γράμμα, λέξη κτλ). Παρ' όλα αυτά, για να μην μπερδευόμαστε εμείς όταν διαβάζουμε τους αλγόριθμους, είναι μάλλον καλή ιδέα να χρησιμοποιούμε διαφορετικά ονόματα μεταβλητών για διαφορετικές λειτουργίες. Η χρήση της δομής **while** γίνεται απλώς γιατί είναι πιο "διάφανος" ο μηχανισμός της, σε σχέση με την **for** που "κρύβει" την ενημέρωση της τιμής του δείκτη i. Τέλος, παρατηρείστε ότι οι επαναλήψεις παραπάνω, ξεκινάνε με τους πάντα ψευδείς, άρα και περιττούς, ελέγχους $A[1] < A[1]$ και $A[1] > A[1]$. Θα αναφερθούμε σε αυτό στην συνέχεια.

Απόδειξη ορθότητας **MaxMin**(A):

Έστω ότι το μέγιστο στοιχείο του πίνακα βρίσκεται στην θέση $1 \leq M \leq n$, και ότι αυτή είναι και η μικρότερη θέση που ο A έχει την τιμή αυτή. Τότε, για $i = M$, ο έλεγχος max $< A[M]$ θα ισχύει, οπότε το max θα πάρει την τιμή $A[M]$. Αφού αυτή είναι η μέγιστη, για όλα τα υπόλοιπα $i > M$, ο έλεγχος αυτός θα είναι ψευδής. Οπότε δεν θα υπάρξει άλλη ενημέρωση του max. Αντίστοιχα και για το min. □

0.3 Πολυπλοκότητα Αλγορίθμων

Μια υπολογιστική μηχανή καταναλώνει πόρους κατά την λειτουργία της. Αυτό μπορεί - καλώς ή κακώς - να εκφραστεί και στην περίπτωση που οι μηχανή αυτή είναι ένα έμβιο ον, στην οποία περίπτωση έχουμε κατανάλωση χρόνου, ενέργειας και μνήμης, ακόμη κι αν τόσο οι έννοιες η αυτές, όσο το τι σημαίνει και πως γίνεται η κατανάλωσή τους, δεν είναι εύκολα ορίσιμο. Στην περίπτωση μηχανών όπως ένας υπολογιστής, τα πράγματα είναι πιο καθαρά και μετρήσιμα: απαιτείται κατανάλωση ρεύματος, λειτουργία κάποιου πλήθους επεξεργαστών, η αποθήκευση δεδομένων σε θέσεις μνήμης και ο χρόνος που θα περάσει μέχρι να ολοκληρωθεί ο υπολογισμός.

Παρότι όλα τα παραπάνω είναι σημαντικά σε πρακτικό ή/και θεωρητικό επίπεδο, εμείς θα ασχοληθούμε κυρίως με τον χρόνο και, λιγότερο, με την μνήμη. Αυτά τα δύο μεγέθη μπορούν να θεωρηθούν και τα πιο βασικά στην θεωρητική μελέτη. Θα τα εξετάζουμε δε και με έναν συγκεκριμένο τρόπο.

Αρχικά, παρατηρούμε ότι η κατανάλωση πόρων όλων των παραπάνω ειδών εξαρτάται με καθοριστικό τρόπο από τις τεχνολογίες που χρησιμοποιούνται. Από τις υπολογιστικές μηχανές της αρχαιότητας, στα τρανζίστορ και στους "υπερ-υπολογιστές" (supercomputers), οι πραγματικοί χρόνοι των διαφόρων υπολογισμών καθώς και η διαθέσιμη μνήμη, έχουν αλλάξει ριζικά. Ακόμη περισσότερο, αλλάζουν συνεχώς με αυξανόμενους ρυθμούς. Μια ανάλυση κατανάλωσης πόρων λοιπόν που λαμβάνει υπόψιν την τεχνολογική εξέλιξη είναι και εξαιρετικά δύσκολη, αλλά και γρήγορα αναχρονιστική (σε καμία περίπτωση όμως χωρίς νόημα). Με στόχο όμως να παράγουμε μια "μαθηματικοποιημένη" θεωρία γύρω από αυτά τα ζητήματα, θα χρειαστεί να μιλήσουμε σε ένα επίπεδο που δεν επηρεάζεται εν γένει από τις τεχνολογικές εξελίξεις (όχι βέβαια όλες: [9]).

Σε ό,τι ακολουθεί ενδιαφερόμαστε για την *χρονική (υπολογιστική) πολυπλοκότητα* αλγορίθμων και προβλημάτων. Στις περιπτώσεις που θα ασχοληθούμε με μνήμη, θα μιλάμε για *χωρική (υπολογιστική) πολυπλοκότητα*. Μερικά από τα "κλασσικά" βιβλία για την γενική θεωρία αλγορίθμων και υπολογιστικής πολυπλοκότητας: [4], [3], [8], [1].

Έστω ένα αντικείμενο `obj` που θέλουμε να δώσουμε ως είσοδο σε έναν αλγόριθμο $\mathcal{A}$. Το `obj` μπορεί να είναι οτιδήποτε: ένας ακέραιος, ένας πίνακας με αριθμούς, ένα σύνολο με στοιχεία, ένα γράφημα, η έξοδος μιας συνάρτησης ή ένας ολόκληρος άλλος αλγόριθμος. Αποδεχόμαστε ότι κάθε τέτοιο αντικείμενο που θα μας απασχολήσει έχει κάποια κωδικοποίηση $<$ `obj` $>$ (πιθανώς ως δυαδική συμβολοσειρά), ώστε να μπορεί να το επεξεργαστεί ένας αλγόριθμος. Αν $w =<$ `obj` $>$ η συμβολοσειρά που κωδικοποιεί το `obj`, με $|w|$ συμβολίζουμε το *μήκος* της, δηλαδή το πλήθος των ψηφίων της.

Ορισμός 1. Έστω ένας αλγόριθμος $\mathcal{A}$. Η *χρονική πολυπλοκότητα* του $\mathcal{A}$ είναι μία αύξουσα συνάρτηση $T : \mathbb{N} \to (0, +\infty)$ τέτοια ώστε, αν n το μήκος της εισόδου $<$ `obj` $>$, τότε ο αλγόριθμος χρειάζεται $T(n)$ βήματα για να τελειώσει.

Παρατηρείστε ότι, βάσει του παραπάνω ορισμού, έχουμε υποθέσει ότι όλοι μας οι αλγόριθμοι τερματίζουν για κάθε είσοδο και σε πεπερασμένο αριθμό βημάτων. Ως εκ τούτου, ένας αλγόριθμος που "κολλάει" για κάποια είσοδο, θα θεωρείται λάθος. Αυτό, παρ' ότι στην πράξη είναι μία εύλογη υπόθεση, ειδικά σε ζητήματα θεωρητικής πολυπλοκότητας δεν ισχύει πάντα.

Αντίστοιχα, μπορούμε να ορίσουμε και την κατανάλωση μνήμης.

Ορισμός 2. Έστω ένας αλγόριθμος $\mathcal{A}$. Η *χωρική πολυπλοκότητα* του $\mathcal{A}$ είναι μία αύξουσα συνάρτηση $S : \mathbb{N} \to (0, +\infty)$ τέτοια ώστε, αν n το μήκος της εισόδου $<$ `obj` $>$, τότε ο αλγόριθμος χρειάζεται $S(n)$ θέσεις μνήμης για την εκτέλεσή του.

Εν γένει, ο χώρος που χρειάζεται ένας αλγόριθμος είναι σημαντικά μικρότερος από τον χρόνο. Αυτό συμβαίνει γιατί η μνήμη μπορεί να σβήνεται και να επαναχρησιμοποιείται, εξοικονομώντας πόρους.

Ας δούμε την συνάρτηση που περιγράφει την χρονική πολυπλοκότητα του **MaxMin** (1). Έχουμε τέσσερις εκχωρήσεις τιμών στις γραμμές 1, 2, 9, 10 καθώς και επιστροφή δύο τιμών στην γραμμή 17. Ας δούμε τώρα τις γραμμές 3–8 (αντίστοιχα για τις 11–16). Η μεταβλητή i ξεκινάει από την τιμή 1 και ανεβαίνει κατά 1 σε κάθε επανάληψη. Ως εκ τούτου, ο έλεγχος της γραμμής 3 θα εκτελεστεί $n + 1$ φορές (n θα είναι αληθής και μία ψευδής). Στις n πρώτες φορές, θα εκτελεστούν μετά οι γραμμές 4, 7 και ίσως και η 5. Μετράμε λοιπόν τρία βήματα, καθώς μας ενδιαφέρει η χειρότερη περίπτωση. Λαμβάνοντας υπόψιν όλα τα παραπάνω, μπορούμε να πούμε ότι η $T_1(n) := 8n+8 = 8(n+1)$ είναι η συνάρτηση χρονικής πολυπλοκότητας τον αλγορίθμου **MaxMin**.

Η πρώτη προφανής βελτίωση του αλγορίθμου μας, είναι να κόψουμε τους περιττούς ελέγχους των γραμμών 4 και 12.

Algorithm 2 MaxMin2(A)

```
# A: πίνακας n θέσεων με τιμές πραγματικούς αριθμούς.
# A[i]: η τιμή του A στην θέση i.
 1: max = A[1]
 2: i = 2
 3: while i ≤ n do
 4:    if max < A[i] then
 5:       max = A[i]
 6:    end if
 7:    i = i + 1
 8: end while
 9: min = A[1]
10: i = 2
11: while i ≤ n do
12:    if min > A[i] then
13:       min = A[i]
14:    end if
15:    i = i + 1
16: end while
17: return max, min
```

Με αυτόν τον τρόπο, γλιτώνουμε όχι μόνο από έναν έλεγχο στις γραμμές 3, 11, αλλά και μία εκτέλεση των γραμμών 4–8, 12–16. Ως εκ τούτου, για τον **MaxMin2** (2), έχουμε χρονική πολυπλοκότητα $T_2(n) := 8(n-1)+8 = 8n = {}_1(n) - 8$. Παρατηρούμε εξαρχής όχι για πολύ μεγάλα n, τα 8 λιγότερα βήματα μάλλον δεν αποτελούν κάποια άξιας αναφοράς βελτίωση.

Ας κινηθούμε λίγο πιο διεξοδικά. Παρατηρούμε ότι μία τιμή σε μια θέση του A δεν μπορεί να είναι ταυτόχρονα και μεγαλύτερη του τρέχοντος max και μικρότερη του τρέχοντος min. Ως εκ τούτου, αν ενοποιήσουμε τις δύο επαναληπτικές διαδικασίες, θα μπορέσουμε να γλυτώσουμε κάποιους ελέγχους. Θα χρησιμοποιήσουμε τώρα την εντολή `for` για εξοικείωση.

Algorithm 3 MaxMin3(A)

```
# A: πίνακας n θέσεων με τιμές πραγματικούς αριθμούς.
# A[i]: η τιμή του A στην θέση i.
 1: if A[1] <= A[2] then
 2:    min = A[1]
 3:    max = A[2]
 4: else
 5:    max = A[1]
 6:    min = A[2]
 7: end if
 8: for i = 3, ..., n do
 9:    if max < A[i] then
10:       max = A[i]
11:    else if min > A[i] then
12:       min = A[i]
13:    end if
14: end for
15: return max, min
```

Για τον **MaxMin3** (3), έχουμε έναν έλεγχο και δύο εκχωρήσεις τιμών στις γραμμές 1–7, καθώς και μια επαναληπτική διαδικασία $n-2$ φορές στις γραμμές 8–14, σε κάθε επανάληψη της οποίας έχουμε μία εκχώρηση τιμής στην μεταβλητή i, το πολύ δύο ελέγχους στις γραμμές 9, 11 και το πολύ μία εκχώρηση τιμής στις γραμμές 10 ή 12. Τέλος, έχουμε δύο επιστροφές τιμών στην γραμμή 15. Οπότε, $T_3(n) := 4(n-2)+5 = 4n-3$. Έχουμε φτάσει πλέον στα μισά βήματα:

$$T_3(n) < \frac{T_2(n)}{2},$$

κάτι που ίσως μοιάζει μη αμελητέο, ακόμη και για μεγάλα n.

Έστω τώρα ότι γνωρίζουμε πως ο πίνακας A είναι ταξινομημένος, με τα στοιχεία του σε αύξουσα σειρά. Ας δούμε πως μπορούμε να εκμεταλλευτούμε αυτή την ιδιότητα.

Algorithm 4 MaxMin4(A)

```
# A: ταξινομημένος πίνακας n θέσεων με τιμές πραγματικούς αριθμούς σε αύξουσα σειρά.
# A[i]: η τιμή του A στην θέση i.
1: max = A[n]
2: min = A[1]
3: return max, min
```

Ο **MaxMin4** (4) χρειάζεται λοιπόν μόλις $T_4(n) = 4$ βήματα (δεν παίζει καν ρόλο το μέγεθος του πίνακα), κάτι που μάλλον δεν πρέπει να μας εκπλήσσει καθώς για την εύρεση μέγιστου και ελάχιστου στοιχείου, η ιδιότητα του να είναι ταξινομημένα τα δεδομένα μας είναι πολύ ισχυρή. Παρ' όλα αυτά, αυτό αποτελεί το πρώτο παράδειγμα του πως μια πιο ισχυρή δομή στα δεδομένα μας, μπορεί να κάνει εξαιρετικά αποτελεσματική μια διαδικασία.

Μιας και τα πράγματα έγιναν τόσο καλά με την ταξινόμηση, ας δούμε τι θα γινόταν αν κινούμασταν με αυτόν τον τρόπο: πρώτα ταξινομούμε τον A και μετά βρίσκουμε μέγιστο και ελάχιστο στοιχείο.

Algorithm 5 BubbleSort(A)

```
# A: πίνακας n θέσεων με τιμές πραγματικούς αριθμούς.
# A[i]: η τιμή του A στην θέση i.
1: for i = 1, ..., n − 1 do
2:    for j = 1, ..., n − 1 do
3:       if A[j] > A[j + 1] then
4:          temp = A[j]
5:          A[j] = A[j + 1]
6:          A[j + 1] = temp
7:       end if
8:    end for
9: end for
10: return A
```

Ας δούμε αρχικά γιατί ο **BubbleSort** (5) δουλεύει. Η εσωτερική επαναληπτική διαδικασία των βημάτων 2–8 ανταλλάσσει τιμές σε διαδοχικές θέσεις του πίνακα όπου η αριστερά θέση περιέχει μικρότερη τιμή από την δεξιά. Παρατηρήστε ότι στην χειρότερη περίπτωση, το μόνο που θα γίνει μετά την πρώτη επανάληψη, θα ναι το μέγιστο στοιχείο να πάει στην θέση n. Αντίστοιχα, σε κάθε μία από τις $n-1$ επαναλήψεις της εξωτερικής διαδικασίας, θα πάει στην σωστή του θέση, τουλάχιστον άλλο ένα στοιχείο του A.

Το πλήθος βημάτων υπολογίζεται ως εξής: για κάθε ζεύγος τιμών που εκχωρούνται στις i, j, έχουμε μία σύγκριση και (ίσως) τρεις εκχωρήσεις τιμών. Έχουμε $(n-1)^2$ τέτοια ζεύγη, άρα συνολικά $T_B(n) := 6(n-1)^2$.

Algorithm 6 MaxMin5(A)

```
# A: πίνακας n θέσεων με τιμές πραγματικούς αριθμούς.
# A[i]: η τιμή του A στην θέση i.
1: BubbleSort(A)
2: return MaxMin4(A)
```

Προφανώς, $T_5(n) := T_B(n) + T_4(n) = 6(n-1)^2 + 4$. Εδώ πλέον τα πράγματα δεν είναι καλά. Και είναι λογικό: κάναμε μια πολύ πιο δύσκολη διαδικασία για να υπολογίσουμε απλώς δύο τιμές. Σε κάθε περίπτωση όμως, θέλουμε έναν αυστηρό μαθηματικό τρόπο να συγκρίνουμε τις $T_k(n)$, $k = 1, 2, 3, 4, 5$, που να αντικατοπτρίζει ακριβώς το τι μας είναι σημαντικό και τι όχι σε αυτήν την μελέτη.

Ορισμός 3. Έστω δύο συναρτήσεις $f, g : \mathbb{N} \to (0, +\infty)$. Θα λέμε ότι:

(i) $f(n) = O(g(n))$ αν υπάρχει σταθερά $c > 0$ και $n_0 \in \mathbb{N}$ τέτοια ώστε $f(n) \leq cg(n)$, για κάθε $n \geq n_0$.

(ii) $f(n) = o(g(n))$ αν για κάθε σταθερά $c > 0$, υπάρχει $n_0 \in \mathbb{N}$ τέτοιο ώστε $f(n) < cg(n)$, για κάθε $n \geq n_0$.

(iii) $f(n) = \Omega(g(n))$ αν υπάρχει σταθερά $c > 0$ και $n_0 \in \mathbb{N}$ τέτοια ώστε $f(n) \geq cg(n)$, για κάθε $n \geq n_0$.

(iv) $f(n) = \omega(g(n))$ αν για κάθε σταθερά $c > 0$, υπάρχει $n_0 \in \mathbb{N}$ τέτοιο ώστε $f(n) > cg(n)$, για κάθε $n \geq n_0$.

(v) $f(n) = \Theta(g(n))$ αν $f(n) = O(g(n))$ και $f(n) = \Omega(g(n))$.

Παρατηρούμε ευθύς εξαρχής ότι $f(n) = O(g(n))$ αν και μόνον αν $g(n) = \Omega(f(n))$ (αντίστοιχα $f(n) = o(g(n))$ αν και μόνον αν $g(n) = \omega(f(n))$). Πράγματι:

$$\begin{aligned}
&f(n) = O(g(n)),\\
&\exists\, c > 0,\ \exists\, n_0 \in \mathbb{N} :\ f(n) \leq cg(n) \Leftrightarrow\\
&\exists\, c > 0,\ \exists\, n_0 \in \mathbb{N} :\ \frac{1}{c} f(n) \leq g(n) \Leftrightarrow\\
&\exists\, c > 0,\ \exists\, n_0 \in \mathbb{N} :\ g(n) \geq df(n),\ d = \frac{1}{c} \Leftrightarrow\\
&g(n) = \Omega(f(n)).
\end{aligned}$$

Ως εκ τούτου, δεν θα ασχοληθούμε άλλο με τους ορισμούς (ii), (iii) και τον (v) θα τον σκεφτόμαστε στην ισοδύναμη μορφή του:

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ και } g(n) = O(f(n)).$$

Οι σχέσεις του Ορισμού 3 αφορούν τους *ρυθμούς αύξησης* των συναρτήσεων f, g. Η (i) λέει ότι η f δεν μπορεί να ξεπερνάει την g περισσότερο από μία σταθερά $c > 0$, τουλάχιστον για μεγάλες τιμές του n. Η (v) ότι η f, g αυξάνονται με "όμοιο τρόπο", ενώ η (ii) ότι η g τελικά είναι "πάνω" από την f. Συγκεντρώνουμε τις σχέσεις μεταξύ των συμβολισμών στην παρακάτω πρόταση:

Πρόταση 1. Έστω δύο συναρτήσεις $f, g : \mathbb{N} \to (0, +\infty)$. Ισχύουν τα ακόλουθα:

1. $f(n) \leq g(n) \Rightarrow f(n) = O(g(n))$, $f(n) < g(n) \Rightarrow f(n) = o(g(n))$, $f(n) = g(n) \Rightarrow f(n) = \Theta(g(n))$. Οι αντίστροφες κατευθύνσεις δεν ισχύουν.
2. $f(n) = o(n) \Rightarrow f(n) = O(g(n))$. Η αντίστροφη κατεύθυνση δεν ισχύει.
3. $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$.
4. Αν $f = o(g(n))$, τότε $g \neq \Theta(f(n))$. Το Θ μπορεί να αντικατασταθεί με O, o.

Απόδειξη:

1. Εύκολα βάσει ορισμού, για $c = 1$.
2. Εύκολα βάσει ορισμού. Για την αντίστροφη κατεύθυνση, έστω $f(n) = n$ και $g(n) = n + 1$. Έχουμε $n < n + 1$ για κάθε $n \in \mathbb{N}$, άρα, από 1, $n = O(n + 1)$. Αλλά, για $c = 1/2$ και $n_0 \geq 1$, έχουμε $n \geq (n + 1)/2$, για κάθε $n \geq n_0$, οπότε δεν ισχύει ότι $n = o(n + 1)$.
3. Προφανές, βάσει ορισμού.
4. Έστω $f(n) = \log_2(n)$, $g(n) = n$ και σταθερά $c > 0$. Έχουμε:

$$\begin{aligned}
&\log_2(n) < cn \overset{n \geq 1}{\Leftrightarrow}\\
&\frac{\log_2(n)}{n} < c,
\end{aligned}$$

Το οποίο ισχύει, αφού:

$$\lim_{n \to +\infty} \frac{\log_2(n)}{n} \overset{DLH}{=} \lim_{n \to +\infty} \frac{1}{n \ln(2)} = 0. \tag{1}$$

Αφού το c ήταν τυχαίο, έπεται ότι $\log_2(n) = o(n)$. Έστω τώρα, προς άτοπο, ότι $n = \Theta(\log_2(n))$. Τότε $n = O(\log_2(n))$, άρα υπάρχουν σταθερά $d > 0$ και $n_0 \in \mathbb{N}$ ώστε:

$$\begin{aligned}
&n \leq d \log_2(n) \Rightarrow\\
&\frac{1}{d} < \frac{\log_2(n)}{n},
\end{aligned}$$

το οποίο και είναι άτοπο από την εξίσωση (1).

□

Ας δούμε μερικά παραδείγματα με συναρτήσεις που θα χρειαστούμε στην ανάλυση των αλγορίθμων μας.

Άσκηση 1. Ισχύουν τα ακόλουθα:

(i) $\log_a(n) = \Theta(\log_b(n))$, για κάθε $a, b \in \mathbb{N}$.

(ii) Έστω $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$, με $a_i \in \mathbb{Z}$, $i = 1, \ldots, k$, $a_k > 0$, $k \in \mathbb{N}$. Τότε $p(n) = \Theta(n^k)$.

(iii) $S(n) = \Theta(n^2)$, όπου:

$$S(n) := \sum_{i=0}^{n} i = 0 + 1 + \cdots + n.$$

(iv) $\texttt{Bin}(n, k) = \Theta(n^k)$, όπου:

$$\texttt{Bin}(n, k) := \binom{n}{k} := \frac{n!}{k!(n-k)!},$$

το διώνυμο του Νεύτωνα.

Παρατήρηση 1. Λόγω του (i) της Άσκησης 1, θα γράφουμε $\log(n)$ χωρίς να προσδιορίζουμε την βάση, υπονοώντας όποτε χρειάζεται βάση 2. Επίσης, λόγω του (ii), σε οποιοδήποτε πολυώνυμο θα ενδιαφερόμαστε μόνον για βαθμό του, αγνοώντας όλους τους συντελεστές, τους μη-μεγιστοβάθμιους όρους και την σταθερά. Το (ii), (iv) θέλουν μεγάλη προσοχή όταν το k δεν είναι σταθερά.

Λύση Άσκησης 1

(i) Από τον τύπο αλλαγής βάσης του λογαρίθμου, έχουμε ότι:

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}.$$

Ως εκ τούτου, για $c = \log_b(a)^{-1}$ και $d = \log_b(a)$, έχουμε ότι, για κάθε $n \in \mathbb{N}$, $\log_a(n) = c \log_b(n)$ και $\log_b(n) = d \log_a(n)$.

(ii) Έχουμε $n^k \leq p(n)$, οπότε από το 1. της Πρότασης 1, $n^k = O(p(x))$. Για το αντίστροφο, έχουμε, για κάθε $n \in \mathbb{N}$:

$$\begin{aligned} p(n) =& a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 \\ \leq& |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_2| n^2 + |a_1| n + |a_0| \\ \leq& |a_k| n^k + |a_{k-1}| n^k + \cdots + |a_2| n^k + |a_1| n^k + |a_0| n^k \\ =& c n^k, \end{aligned}$$

όπου:

$$c = \sum_{i=0}^{k} |a_i|.$$

- Ξέρουμε ότι $S(n) = n(n+1)/2$. Από το (ii), έπεται ότι $S(n) = \Theta(n^2)$.
- Έχουμε ότι $\texttt{Bin}(n, k) = (k!)^{-1} n(n-1)\ldots(n-k+1)$, δηλαδή πολυώνυμο βαθμού k. Από το (ii), έπεται ότι $\texttt{Bin}(n, k) = \Theta(n^k)$.

◇

Άσκηση 2. Ισχύουν τα ακόλουθα, για κάθε $k \in \mathbb{N}$:

(i) $n^k = o(n^k \log(n))$.

(ii) $n^k \log(n) = o(n^{k+1})$.

(iii) $n^k = o(n^l)$, για κάθε $l > k > 0$.

(iv) $n^k = o(2^n)$.

(v) $k^n = o(l^n)$, για κάθε $l > k > 0$.

(vi) $2^{kn} = o(2^{ln})$, για κάθε $l > k > 0$.

Λύση Άσκησης 2

(i) Έστω $c > 0$. Εύκολα:

$$\begin{aligned} & n^k < cn^k \log(n) \overset{n \geq 1}{\Leftrightarrow} \\ & \frac{1}{c} < \log(n) \Leftrightarrow \\ & n > 2^{\frac{1}{c}}. \end{aligned}$$

(ii) Έστω $c > 0$. Εύκολα:

$$\begin{aligned} & n^k \log(n) < cn^{k+1} \overset{n \geq 0}{\Leftrightarrow} \\ & \log(n) < cn \Leftrightarrow \\ & \frac{\log(n)}{n} < c, \end{aligned}$$

το οποίο και ισχύει από την εξίσωση 1.

(iii) Άμεση συνέπεια των (i), (ii).

(iv) Έστω $c > 0$. Έχουμε:

$$\begin{aligned} & n^k < c2^n \Leftrightarrow \\ & k \log(n) < \log(c) + n \Leftrightarrow \\ & k \log(n) - \log(c) < n, \end{aligned}$$

που είναι άμεση συνέπεια του (ii).

(v) Έστω $c > 0$. Εύκολα:

$$\begin{aligned} & k^n < cl^n \Leftrightarrow \\ & \left(\frac{k}{l}\right)^n < c, \end{aligned}$$

που ισχύει, αφού για $l > k$:

$$\lim_{n \to +\infty} \left(\frac{k}{l}\right)^n = 0.$$

(vi) Έστω $c > 0$. Εύκολα:

$$\begin{aligned} & 2^{kn} < c2^{ln} \Leftrightarrow \\ & \left(\frac{1}{2^{l-k}}\right)^n < c, \end{aligned}$$

που ισχύει, αφού για $l > k$:

$$\lim_{n \to +\infty} \left(\frac{1}{2^{l-k}}\right)^n = 0.$$

◇

Στην βιβλιογραφία, κάποια προτιμούν να ορίζουν τα $O, o, \Omega, \omega, \Theta$ ως σύνολα συναρτήσεων, π.χ.

$$O(g(n)) := \{f : \mathbb{N} \to (0, +\infty) \mid \exists\, c > 0,\ \exists\, n_0 \in \mathbb{N} \text{ τ.ω. } f(n) \leq cg(n)\}.$$

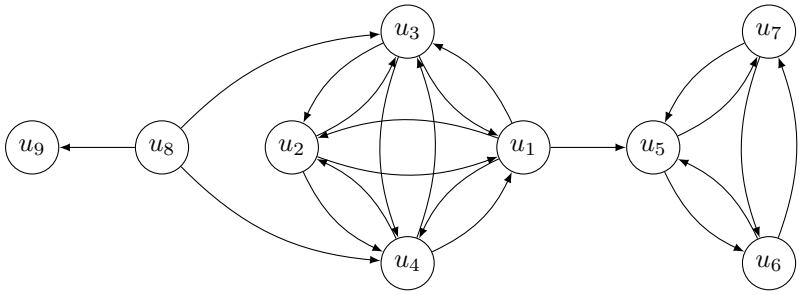
Σε αυτήν την περίπτωση, γράφουμε $f(n) \in O(g(n))$ αντί για $f(n) = O(g(n))$. Τέτοιες διαφορές στον φορμαλισμό δεν θα μας απασχολήσουν, τόσο γιατί η υπολογιστική πολυπλοκότητα δεν αποτελεί το κύριο αντικείμενο των σημειώσεων, όσο και γιατί αυτό που θα μας απασχολήσει είναι μια πιο "διαισθητική" χρήση των παραπάνω ορισμών.

0.4 Γραφήματα

Κάνουμε μια μικρή παρέκβαση από ζητήματα υπολογιστικής πολυπλοκότητας, για να συζητήσουμε κάποια στοιχεία από την Θεωρία Γραφημάτων που θα μας χρειαστούν. Ένα γράφημα είναι ένα σύνολο από σημεία και γραμμές που τις ενώνουν. Τα σημεία αυτά λέγονται συνήθως *κορυφές* και οι γραμμές που τα ενώνουν *ακμές*, οι οποίες μπορεί να έχουν ή να μην έχουν κατεύθυνση.

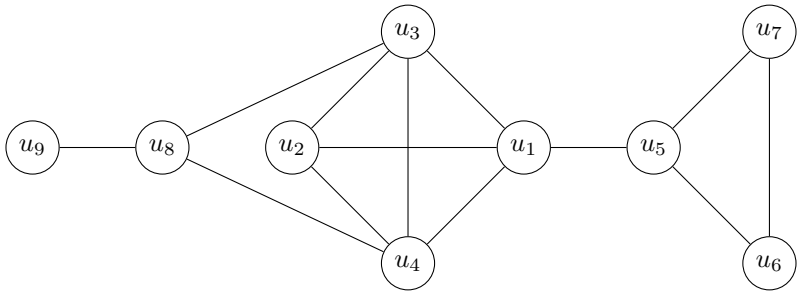
Διαισθητικά, τα γραφήματα απεικονίζουν κάποιες δυαδικές σχέσεις μεταξύ δεδομένων, που αναπαρίστανται από τις κορυφές. Σε ένα γράφημα που ακμές είναι χωρίς κατευθύνσεις, η σχέση που απεικονίζεται είναι συμμετρική. Για παράδειγμα, αν οι κορυφές $u_1, u_2, u_3, u_4, u_{5,6}, u_7, u_8, u_9$ απεικονίζουν τους χρήστες κάποιας πλατφόρμας κοινωνικής δικτύωσης, τότε το παρακάτω *κατευθυνόμενο* γράφημα θα μπορούσε να απεικονίζει το ποια χρήστρια ακολουθεί ποια:

Figure 0.1: Κατευθυνόμενο γράφημα D



ενώ το παρακάτω, ποιοι χρήστες είναι φίλοι μεταξύ τους:

Figure 0.2: Μη κατευθυνόμενο γράφημα G



Το γράφημα G (0.2) λέγεται και *υποκείμενο* μη κατευθυνόμενο γράφημα του γραφήματος D (0.1).

Ορισμός 4. Ένα ζεύγος συνόλων $G = (V, E)$ θα λέγεται *μη κατευθυνόμενο γράφημα* όταν V πεπερασμένο σύνολο και $E \subseteq \{\{u, v\} \mid u, v \in V\}$. Το V θα λέγεται σύνολο κορυφών και το E το σύνολο των ακμών. Το G θα λέγεται *απλό* αν:

- δεν υπάρχει $u \in V$ τέτοια ώστε $\{u, u\} \in E$ (τέτοιες ακμές λέγονται βρόγχοι) και
- για κάθε $u, v \in V$, υπάρχει το πολύ μία ακμή $u, v \in E$.

Το G θα λέγεται *κατευθυνόμενο*, αν $E \subseteq V \times V$, με το ζεύγος (u, v) να δηλώνει την ύπαρξη ακμής που ξεκινάει από το u και πηγαίνει στο v. Θα λέγεται απλό αν $E \subseteq V \times V \setminus \{(u, u) \mid u \in V\}$ και αν για κάθε $u, v \in V$, υπάρχει το πολύ μία ακμή $(u, v) \in E$.

Παρατηρείστε ότι η μη ύπαρξη πολλαπλών ακμών εξασφαλίζεται κι από το γεγονός ότι το E είναι σύνολο. Από εδώ και στο εξής, όλα τα γραφήματά μας θα είναι απλά και μη κατευθυνόμενα, εκτός αν δηλώσουμε κάτι διαφορετικό ρητώς. Αντίστοιχα, και τα κατευθυνόμενα γραφήματα θα θεωρούνται εν γένει "σιωπηλά" ότι είναι απλά. Παρατηρείστε ότι, στην περίπτωση των κατευθυνόμενων γραφημάτων, οι $(u, v), (v, u)$ είναι διαφορετικές ακμές.

Όταν υπάρχει ανάγκη, θα γράφουμε $V(G)$ και $E(G)$ για να συμβολίσουμε το σύνολο κορυφών και ακμών του G αντίστοιχα. Για τα γραφήματα 0.1, 0.2, έχουμε $V(G) = V(D) = \{u_1, u_2, u_3, u_4, u_{5,6}, u_7, u_8, u_9\}$ και

$$E(G) = \{\{u_i, u_j\} \mid 0 < i < j \leq 4\} \cup \{\{u_i, u_j\} \mid 5 \leq i < j \leq 7\} \cup \{\{u_8, u_i\} \mid i = 3, 4, 9\} \cup \{\{u_1, u_5\}\},$$
$$E(D) = \{(u_i, u_j) \mid i, j = 1, 2, 3, 4 : \ i \neq j\} \cup \{(u_i, u_j) \mid i, j = 5, 6, 7 : \ i \neq j\} \cup \{(u_8, u_i) \mid i = 3, 4, 9\} \cup \{(u_1, u_5)\}.$$

Θα αναφέρουμε εδώ τις ελάχιστες έννοιες που θα χρειαστούμε για να μιλάμε για γραφήματα και θα αναφερόμαστε στα γραφήματα 0.1, 0.2 ως παραδείγματα.

- *Βαθμός* (degree) μίας κορυφής u είναι το πλήθος των ακμών που προσπίπτουν σε αυτήν και συμβολίζεται με:

$$\deg_G(u) := |\{e \in E \mid e = \{u, v\}, \ v \in V\}|.$$

  Σε κατευθυνόμενα γραφήματα, διακρίνουμε τον *έσω-βαθμό* (in-degree) και τον *έξω-βαθμό* (out-degree) της u, ως το πλήθος των ακμών που καταλήγουν και ξεκινάνε από την u αντίστοιχα:

$$\deg_G^{in}(u) := |\{e \in E \mid e = (v, u), \ v \in V\}|, \ \deg_G^{out}(u) := |\{e \in E \mid e = (u, v), \ v \in V\}|.$$

  Στα γραφήματα 0.1, 0.2, έχουμε $\deg_G(u_5) = 3$, $\deg_D^{in}(u_5) = 3$, $\deg_D^{out}(u_5) = 2$, $\deg_G(u_9) = 1$, $\deg_D^{in}(u_9) = 1$, $\deg_D^{out}(u_9) = 0$.

- *Μονοπάτι* (path) θα λέγεται μια ακολουθία κορυφών $P = [u_1, \ldots, u_n]$ τέτοια ώστε (i) $u_i \neq u_j$, για κάθε $i \neq j$ και (ii) $\{u_i, u_{i+1}\} \in E$, για $i = 1, \ldots, n-1$ ((u_i, u_{i+1}) για κατευθυνόμενα γραφήματα). Θα γράφουμε και $P(u, v)$ για να δηλώσουμε το σύνολο των μονοπατιών από την u στην v.

  Κύκλος (cycle) θα λέγεται ένα μονοπάτι $[u_1, \ldots, u_n]$, που επιπλέον $\{u_n, u_1\} \in E$ ($(u_n, u_1) \in E$ για κατευθυνόμενα). Στην περίπτωση αυτή θα γράφουμε $C = (u_1, \ldots, u_n)$.

  Το *μήκος* ενός μονοπατιού ή κύκλου, ορίζεται ως το πλήθος των ακμών τους. Η *απόσταση* δύο κορυφών $u, v \in V(G)$ ορίζεται ως το μήκος του μικρότερου μονοπατιού που τις συνδέει.

  Στο γράφημα G (0.2), έχουμε:

  - $[u_9, u_8, u_3, u_2, u_1, u_5]$ μονοπάτι μήκους 5,
  - u_8, u_2, u_3 όχι μονοπάτι, αφού $\{u_8, u_2\} \notin E(G)$,
  - (u_3, u_2, u_1, u_4) κύκλος μήκους 4,
  - u_1, u_5, u_7, u_6, u_5 όχι κύκλος, αφού η u_5 εμφανίζεται δύο φορές.

  Στο γράφημα D (0.1), έχουμε:

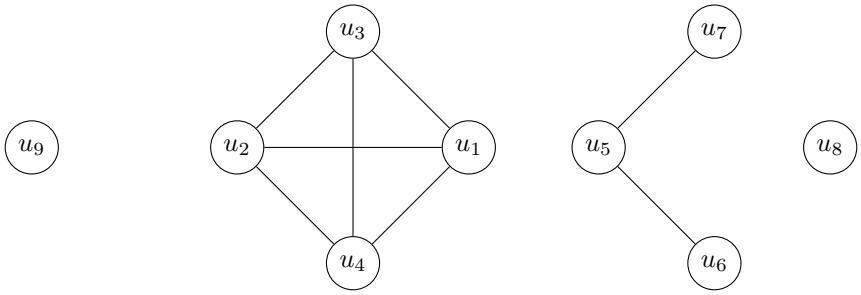
  - $u_9, u_8, u_3, u_2, u_1, u_5$ όχι μονοπάτι, αφού $(u_9, u_8) \notin E(D)$,
  - $[u_8, u_3, u_1, u_5, u_7, u_6]$ μονοπάτι μήκους 5,
  - (u_5, u_6, u_7) κύκλος μήκους 3,
  - u_8, u_3, u_1, u_4 όχι κύκλος, αφού η $(u_4, u_8) \notin E(D)$.

  Ένα κλασσικό αλγοριθμικό πρόβλημα είναι αυτό της "προσβασιμότητας", του αν δηλαδή υπάρχει μονοπάτι μεταξύ δύο κορυφών σε ένα γράφημα. Το πρόβλημα αυτό θα μας απασχολήσει στην επόμενη υπο-ενότητα.

- Ένα γράφημα H είναι *υπογράφημα* του G, αν προκύπτει από το G με αφαίρεση κορυφών ή/και ακμών. Παρατηρείστε ότι όταν αφαιρούμε μία κορυφή, πρέπει να αφαιρούμε και όλες τις ακμές που προσπίπτουν σε αυτή. Φορμαλιστικά: $V(H) \subseteq V(G)$ και $E(H) \subseteq E(G) \cap \{\{u, v\} \mid u, v \in V(H)\}$. Το H θα λέγεται *εναγώμενο* (induced) υπογράφημα του G αν προκύπτει από αφαίρεση ακμών. Φορμαλιστικά: $V(H) = V(G)$ και $E(H) \subseteq E(G)$. Έστω ένα υποσύνολο $V' \subseteq V(G)$ κορυφών του G. Το *επαγώμενο* υπογράφημα του V' στο G είναι το $H = (V', E(V') := \{\{u, v\} \in E \mid u, v \in V'\})$, που προκύπτει από όλες τις ακμές του G που ενώνουν κορυφές στο V'. Οι ορισμοί αυτοί δεν έχουν κάποια ουσιαστική διαφορά στα κατευθυνόμενα γραφήματα.

  Στο γράφημα G (0.2), έστω $G_1 = (\{u_9\}, \emptyset)$, $G_2 = (\{u_i \mid i = 1, 2, 3, 4\}, E[\{u_i \mid i = 1, 2, 3, 4\}])$, $G_3 = (\{u_i \mid i = 5, 6, 7, 8\}, \{\{u_5, u_6\}, \{u_5, u_7\}\})$, $H = (V(G_1) \cup V(G_2) \cup V(G_3), E(G_1) \cup E(G_2) \cup E(G_3))$:

Figure 0.3: Υπογραφήματα G_1, G_2, G_3, εναγώμενο υπογράφημα H



Παρατηρείστε ότι τα G_1, G_2 είναι επαγώμενα γραφήματα των συνόλων κορυφών $\{u_9\}$ και $\{u_1, u_2, u_3, u_4\}$, και το G_3 απλό υπογράφημα του G. Το H είναι εναγώμενο υπογράφημα του G.

- Ένα γράφημα G λέγεται *συνεκτικό* (ή *συνδεδεμένο*), αν υπάρχει μονοπάτι που συνδέει οποιεσδήποτε δύο κορυφές του. Δηλαδή, αν για κάθε $u, v \in V(G)$, το $P(u, v) \neq \emptyset$. Ένα κατευθυνόμενο γράφημα D λέγεται *ισχυρά συνεκτικό*, αν υπάρχουν μονοπάτια από και προς οποιεσδήποτε δύο κορυφές του. Δηλαδή, αν για κάθε $u, v \in V(G)$, τα $P(u, v)$, $P(v, u) \neq \emptyset$. Η απλή συνεκτικότητα σε κατευθυνόμενα γραφήματα ορίζεται ως το υποκείμενο μη κατευθυνόμενο γράφημα να είναι συνεκτικό. Δεν θα μας απασχολήσει όμως.

  Ένα υπογράφημα H κάποιου γραφήματος G είναι *συνεκτική συνιστώσα* (Connected Component) του G αν είναι μεγιστικά συνεκτικό εναγώμενο υπογράφημα του G. Δηλαδή:

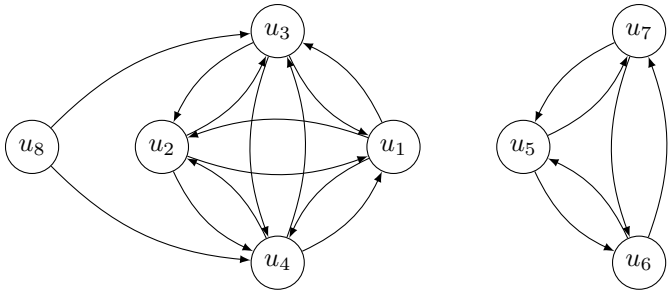
  - $V(H) \subseteq V(G)$,
  - $E(H) = E[V(H)]$,
  - H συνεκτικό και
  - για κάθε $u \in V(G) \setminus V(H)$, το γράφημα $(V(H) \cup \{u\}, E[V(H) \cup \{u\}])$ δεν είναι συνεκτικό.

  Αντίστοιχα ορίζεται και η *ισχυρά συνεκτική συνιστώσα* (Strongly Connected Component) στην περίπτωση που το γράφημα είναι κατευθυνόμενο.

  Τα γραφήματα G, D (0.2, 0.1) είναι συνεκτικά και ισχυρά συνεκτικά αντίστοιχα. Ως εκ τούτου, η μόνη συνεκτική και ισχυρά συνεκτική τους αντίστοιχα συνιστώσα, είναι τα ίδια τα γραφήματα ολόκληρα. Για τα γραφήματα G_1, G_2, G_3, H (0.3), έχουμε ότι τα G_1, G_2 συνεκτικά, αλλά τα G_3, H όχι. Επίσης, το H έχει τέσσερις συνεκτικές συνιστώσες: $(\{u_9\}, E[\{u_9\}])$, $(\{u_i \mid i = 1, 2, 3, 4\}, E[\{u_i \mid i = 1, 2, 3, 4\}]$, $(\{u_i \mid i = 5, 6, 7\}, E[\{u_i \mid i = 5, 6, 7\}])$, $(\{u_8\}, E[\{u_8\}])$. Το $(\{u_i \mid i = 1, 2, 3\}, E[\{u_i \mid i = 1, 2, 3\}])$ δεν είναι συνεκτική συνιστώσα του H καθώς η προσθήκη της u_4 σε αυτό δεν χαλάει την συνεκτικότητα.

  Ας πάρουμε το υπογράφημα D' του κατευθυνόμενου γραφήματος D (0.1).

Figure 0.4: Κατευθυνόμενο γράφημα D'



Το D' δεν είναι συνεκτικό, αφού το υποκείμενο μη κατευθυνόμενο γράφημα δεν είναι συνεκτικό. Άρα ούτε ισχυρά συνεκτικό. Έστω D_1 το εναγώμενο υπογράφημα από τις κορυφές $u_{1,2}$, u_3, u_4, u_8, D_2 αυτό από τις u_1, u_2, u_3, u_4 και D_3 αυτό από τις u_5, u_6, u_7. Το D_1 είναι συνεκτικό, αλλά δεν είναι ισχυρά συνεκτική συνιστώσα, καθώς τα $P(u_i, u_8)$ είναι κενά για $i = 1, 2, 3, 4$. Τα D_2 και D_3 είναι ισχυρά συνεκτικές συνιστώσες του D'.

Ο έλεγχος του αν ένα γράφημα είναι (ισχυρά) συνεκτικό, καθώς και η εύρεση των (ισχυρά) συνεκτικών συνιστωσών του αν δεν είναι, είναι επίσης ένα κλασσικό αλγοριθμικό πρόβλημα της περιοχής αυτής.

- Ένα γράφημα $G = (V, \emptyset)$ που αποτελείται μόνο από κορυφές, ονομάζεται *σκόνη*. Είναι προφανώς μη συνεκτικό, με τόσες συνεκτικές συνιστώσες, όσες είναι και οι κορυφές του. Έστω ένα γράφημα $G = (V, E)$ και $I \subseteq V$ ένα υποσύνολο των κορυφών του. Αν οι κορυφές του I δεν ενώνονται με καμία ακμή του G, δηλαδή αν το επαγώμενο γράφημα $(I, E[I])$ από τις κορυφές του I στο G είναι σκόνη ($E[I] = \emptyset$), θα λέμε ότι το I είναι *ανεξάρτητο σύνολο* στο G.

  Στο γράφημα G (0.2), το $I = \{u_9, u_4, u_7\}$ είναι ανεξάρτητο σύνολο.

  Παρατηρούμε ότι η εύρεση ενός ανεξάρτητου συνόλου σε ένα γράφημα, γίνεται τετριμμένα με το να παίρνουμε μονοσύνολα κορυφών του. Για μεγαλύτερα ανεξάρτητα σύνολα, το πρόβλημα γίνεται πιο ενδιαφέρον.

Algorithm 7 IndSet$(< G, k >)$

```
# G: γράφημα
# k ≥ 2: μέγεθος ανεξάρτητου συνόλου που μας ενδιαφέρει.
 1: for I ⊆ V(G), |I| = k do
 2:    found = 1
 3:    for u, v ∈ I do
 4:       if {u, v} ∈ E then
 5:          found = 0
 6:       end if
 7:    end for
 8:    if found = 1 then
 9:       return I
10:    end if
11: end for
12: return −1
```

Αρχικά, υποθέτουμε ότι το γράφημα G και ο ακέραιος k έχουν κωδικοποιηθεί με κάποιον τρόπο, ώστε ο αλγόριθμος να μπορεί να τα χειριστεί (δεν είναι τόσο δύσκολο όσο ίσως φαίνεται). Διατρέχουμε σε κάποια αυθαίρετη σειρά (δεν είναι απολύτως τετριμμένο αυτό, αλλά δεν είναι και δύσκολο) όλα τα υποσύνολα k κορυφών του G. Για κάθε ένα τέτοιο σύνολο I, ξεκινάμε θεωρώντας ότι είναι ανεξάρτητο σύνολο (μέσω της `found`) και στην συνέχεια ελέγχουμε αν υπάρχει ακμή του G στο $E[I]$. Αν υπάρχει, διορθώνουμε την τιμή της found και προχωράμε στο επόμενο I. Αλλιώς, τερματίζουμε επιστρέφοντας το I. Αν διατρέξουμε όλα αυτά τα υποσύνολα χωρίς να βρούμε τίποτα, επιστρέφουμε κάποια τιμή που να μην υπάρχει ποτέ περίπτωση να ήταν αποδεκτή για το συγκεκριμένο πρόβλημα.

Τι χρόνο θα χρειαστεί ο αλγόριθμος για να τερματίσει; Ξέρουμε ήδη ότι στο επίπεδο που δουλεύουμε, ουσιαστικά ενδιαφερόμαστε για το πλήθος των επαναλήψεων που θα γίνουν και όχι για το πλήθος ελέγχων και εκχωρήσεων τιμών που γίνονται σε κάθε μία από αυτές, καθώς αυτό είναι σταθερό. Μας ενδιαφέρει δηλαδή το πλήθος των φορών που θα εκτελεστεί η γραμμή 1 και 3, σε ένα γράφημα με n κορυφές και m ακμές. Αφού σε κάθε επανάληψη της 1, θα τρέχουμε όλες τις επαναλήψεις της 3, έχουμε ότι:

$$T_{IS}(n+m) = O(|\{I \subseteq |V(G)| \mid |I| = k\}| \cdot |\{\{u, v\} \mid u, v \in I,\ |I| = k\}|).$$

Η προσεκτική αναγνώστρια, ίσως αναρρωτηθεί εδώ πιο είναι το μέγεθος της εισόδου $n + m$, ως προς το οποίο γίνεται ο υπολογισμός της πολυπλοκότητας του **IndSet**. Η σωστή απάντηση είναι ότι το μέγεθος της εισόδου το μήκος της κωδικοποίησης $< G, k >$, βάσει της οποίας έχουμε πρόσβαση στο γράφημα G και τον θετικό ακέραιο k. Στην πράξη, μπορούμε να έχουμε στο μυαλό μας ότι το μέγεθος της εισόδου είναι $O(|V(G)|+|E(G)|)$. Θεωρούμε δηλαδή ότι τόσο η κωδικοποίηση του k, όσο και ο ίδιος ο τρόπος που κωδικοποιούμε το γράφημα, δεν μας κοστίζει κάτι που χρειάζεται να λάβουμε υπόψιν. Δεν θα επιμείνουμε σε αυτό, καθώς ούτε είναι αντικείμενο των σημειώσεων αυτών, αλλά ούτε και μπορούμε, με όσα έχουμε πει ως τώρα, να γίνουμε πιο ακριβείς.

Ας δούμε τον πρώτο παράγοντα του $T_{IS}(n)$. Θέλουμε να διαλέξουμε k από n διακεκριμένες αντικείμενα (κορυφές), χωρίς να μας ενδιαφέρει η σειρά επιλογής. Έτσι ορίζουμε συνδυαστικά τον αριθμό:

$$\binom{n}{k} = O(n^k)$$

όπως έχουμε ήδη δει. Ο δεύτερος παράγοντας είναι ο ίδιος, αλλά για k αντικείμενα εκ των οποίων διαλέγουμε 2. Άρα έχουμε $n^k k^2$, οπότε, αν το k θεωρηθεί σταθερά:

$$T_{IS}(n) = (n^k).$$

Τι συμβαίνει όμως αν το k δεν είναι σταθερό; Αφενός, χρειάζεται ένα πιο αναλυτικό επιχείρημα για το πως φεύγει ο όρος k^2. Πιο σημαντικό όμως είναι ότι το πλήθος των υπολογιστικών βημάτων έχει εκθετική εξάρτηση από το

μέγεθος του ανεξάρτητου συνόλου που ψάχνουμε, κάτι που σημαίνει ότι για μικρές τιμές του k (π.χ. $k = 100$), αρχίζει να υπάρχει πρόβλημα.

Μπορούμε να δούμε το πρόβλημα αυτό ακόμη πιο καθαρά και στον παρακάτω αλγόριθμο, που υπολογίζει το μέγιστο δυνατό ανεξάρτητο σύνολο.

Algorithm 8 MaxIndSet$(< G >)$

```
# G: γράφημα
1: k = 1
2: while IndSet(< G, k + 1 >) ≠ −1 do
3:     k = k + 1
4: end while
5: return k
```

Ο παραπάνω αλγόριθμος εκτελεί τον **IndSet** διαδοχικά για κάθε τιμή $k \geq 2$. Αν βρεθεί τιμή που δεν υπάρχει ανεξάρτητο σύνολο, τότε εύκολα διαπιστώνουμε ότι δεν μπορεί να υπάρξει για μεγαλύτερή της. Έχουμε:

$$T_{MIS} = O\Big(\sum_{k=1}^{n} n^k\Big) = O(n^{n+1}),$$

που είναι εκθετικό ως προς n.

Μία τελευταία παρατήρηση. Παρ' ότι, όπως θα δούμε, η παραπάνω προσέγγιση "ωμής βίας" (brute-force) μάλλον δεν μπορεί να βελτιωθεί αρκετά ώστε να σταματήσει η εκθετική εξάρτηση από το μέγεθος του ανεξάρτητου συνόλου, αυτό που μπορούμε να κάνουμε "αποδοτικά" είναι να ελέγξουμε αν ένα προτεινόμενο σύνολο, είναι ανεξάρτητο ή όχι.

Algorithm 9 VerIndSet$(< G, k, I >)$

```
# G: γράφημα
# k ≥ 2: μέγεθος ανεξάρτητου συνόλου που μας ενδιαφέρει
# I ⊆ V(G)
1: for u, v ∈ I do
2:     if {u, v} ∈ E then
3:         return FALSE
4:     end if
5: end for
6: return TRUE
```

Όπως εξηγήσαμε ήδη, $T_{VIS} = O(k^2)$.

0.5 Πολυπλοκότητα Προβλημάτων

Για λόγους πληρότητας, αναφέρουμε εδώ λίγα στοιχεία για τις λεγόμενες *κλάσεις πολυπλοκότητας*. Είδαμε μέχρι στιγμής πως υπολογίζουμε την χρονική πολυπλοκότητα ενός αλγορίθμου. Είναι αρκετά εύλογο να ορίσουμε ως την δυσκολία επίλυσης ενός προβλήματος, τον καλύτερο αλγόριθμο που το λύνει. Οι δυσκολίες που συναντάμε σε αυτό είναι διάφορες: από την μία, ένας γρήγορος αλγόριθμος δεν σημαίνει ότι έχει μικρή κατανάλωση μνήμης κι αντίστροφα. Επίσης, οι βέλτιστοι αλγόριθμοι συχνά χρησιμοποιούν διάφορες ad hoc ευρεσιτεχνίες (heuristics) που τους κάνουν πολύ περίπλοκους. Τέλος, το κυριότερο πρόβλημα είναι πως είναι εν γένει αρκετά δύσκολο να αποδειχθεί ότι ένα πρόβλημα δεν μπορεί να λυθεί "καλύτερα". Ως εκ τούτου, τα φράγματα που δίνονται σε αυτά τα μεγέθη είναι συνήθως άνω και σπανίως ταυτίζονται με τα κάτω, εφόσον αυτά καν υπάρχουν.

Υπάρχουν δύο γενικές κατηγορίες προβλημάτων: αυτά τις *βελτιστοποίησης*, όπου θέλουμε την μεγιστοποίηση ή την ελαχιστοποίηση κάποιας παραμέτρου (π.χ. το μέγεθος του μεγαλύτερου ανεξάρτητου συνόλου) και αυτά της *απόφασης*, όπου θέλουμε να αποφανθούμε θετικά ή αρνητικά σε κάποιο ζήτημα (π.χ. αν υπάρχει ανεξάρτητο σύνολο συγκεκριμένου μεγέθους).

Σε κάποιο επίπεδο, κάθε πρόβλημα των δύο παραπάνω κατηγοριών, καθώς και αυτά που δεν ταιριάζουν σε κάποια εκ των δύο, μπορούν να εκφραστούν με "ισοδύναμο" τρόπο. Αυτό, στην περίπτωση που μας απασχολεί ο χρόνος επίλυσής τους, ισχύει και με σχετικά μικρή "χασούρα" ως προς τον χρόνο επίλυσης. Σε αυτήν την μικρή αναφορά μας, δεν θα μπούμε καθόλου σε τέτοια ζητήματα. Θα ασχοληθούμε μόνο με προβλήματα απόφασης.

Έχοντας φιξάρει ένα αλφάβητο βάσει του οποίου γίνονται όλες οι κωδικοποιήσεις μας (π.χ. το $\{0,1\}$), θα ορίζουμε ένα πρόβλημα απόφασης Π ως το σύνολο των συμβολοσειρών εκείνων, που κωδικοποιούν αντικείμενα που αποτελούν ειδική περίπτωση (instance) του προβλήματος, για τα οποία η απάντηση είναι καταφατική. Για παράδειγμα, το πρόβλημα απόφασης του ανεξάρτητου συνόλου, ορίζεται ως:

$$IndSet := \{< G, k > | \text{ το } G \text{ έχει ανεξάρτητο σύνολο μεγέθους } k\}.$$

Έτσι λοιπόν, δεδομένου ενός γραφήματος G και μιας σταθεράς k, το πρόβλημα είναι να αποφασίσουμε αν το $< G, k >$ ανήκει ή όχι στο IndSet.

Ένας αλγόριθμος $\mathcal{A}$ επιλύει ένα πρόβλημα Π αν, για κάθε w που κωδικοποιεί κάποια αποδεκτή έκφανση του προβλήματος:

$$\mathsf{A}(w) = \begin{cases} \text{'ναι'}, & \text{αν } w \in \Pi, \\ \text{'όχι'}, & \text{αλλιώς.} \end{cases}$$

Αν λοιπόν ο αλγόριθμος $\mathcal{A}$ λύνει το πρόβλημα Π σε χρόνο $T(n)$ και χώρο $S(n)$, οι συναρτήσεις $T(n)$ και $S(n)$ αποτελούν άνω φράγματα στην χρονική και χωρική πολυπλοκότητα του προβλήματος Π. Ξεχωρίζουμε κάποιες βασικές κλάσεις ως προς την χρονική και χωρική πολυπλοκότητα προβλημάτων:

- L: προβλήματα που λύνονται από αλγόριθμο που χρησιμοποιεί λογαριθμικό χώρο ως προς το μέγεθος της εισόδου.
- P: προβλήματα που λύνονται από αλγόριθμο πολυωνιμικού χρόνου ως προς το μέγεθος της εισόδου.
- NP: προβλήματα που επαληθεύονται από αλγόριθμο πολυωνιμικού χρόνου ως προς το μέγεθος της εισόδου.
- PSPACE: προβλήματα που λύνονται από αλγόριθμο που χρησιμοποιεί πολυωνυμικό χώρο ως προς το μέγεθος της εισόδου.
- EXP: προβλήματα που επαληθεύονται από αλγόριθμο εκθετικού χρόνου ως προς το μέγεθος της εισόδου

Οι παραπάνω κλάσεις έχουν δοθεί σε "αύξουσα σειρά δυσκολίας": $\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$. Υπάρχουν διάφοροι εκ των παραπάνω εγκλεισμών που δεν γνωρίζουμε αν είναι γνήσιοι οι όχι (και υπάρχουν εκατοντάδες ακόμη κλάσεις πολυπλοκότητας που δεν έχουμε αναφέρει). Το κεντρικό και πιο διάσημο πρόβλημα είναι το $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, το αν δηλαδή η επίλυση είναι πιο δύσκολη από την επαλήθευση. Πιστεύεται ισχυρά πως ναι, και πως άρα $\mathsf{P} \neq \mathsf{NP}$, αλλά είναι σαφές ότι τα μαθηματικά που έχουμε στην διάθεσή μας δεν αρκούν για να απαντηθεί το ερώτημα.

Παραδείγματα προβλημάτων που έχουμε αναφερθεί για τις παραπάνω κλάσεις:

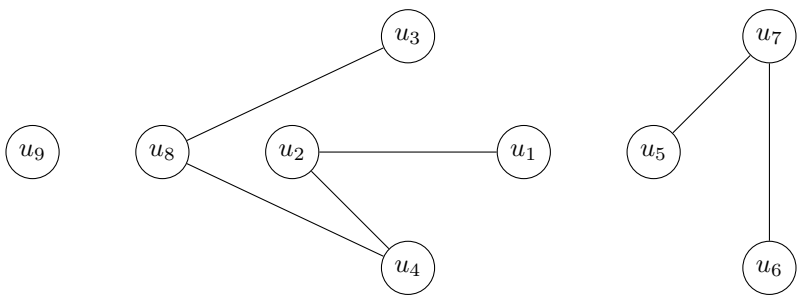
- $Reachability := \{< G, s, t > |\ s, t \in V(G) \text{ και } P(s,t) \neq \emptyset\} \in \mathsf{L}$,
- $MaxArray := \{< A, k > |\ \max(A) \geq k\} \in \mathsf{P}$,
- $IndSet := \{< G, k > |\ \text{ το } G \text{ έχει ανεξάρτητο σύνολο μεγέθους τουλάχιστον } k\} \in \mathsf{NP}$.

0.6 Δέντρα

Ολοκληρώνουμε αυτό το πρώτο εισαγωγικό κεφάλαιο με μια συγκεκριμένη κλάση γραφημάτων που θα μας φανεί πολύ χρήσιμη στην συνέχεια.

Ορισμός 5. Ένα γράφημα $T = (V, E)$ λέγεται *δέντρο*, αν είναι συνεκτικό και *άκυκλο* (δεν υπάρχουν κορυφές που να σχηματίζουν κύκλο). Ένα γράφημα $F = (V, E)$ λέγεται *δάσος* αν είναι μόνο άκυκλο.

Figure 0.5: Δάσος F

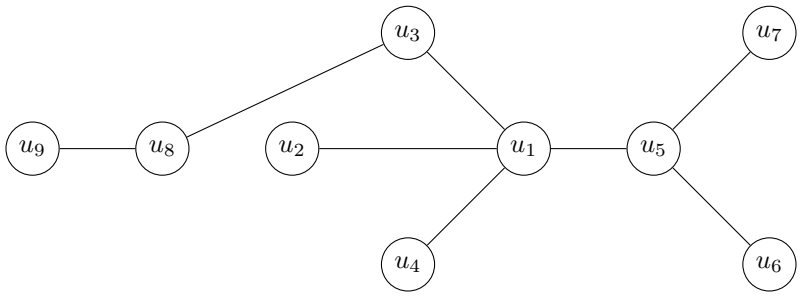


Θεώρημα 1. Ένα γράφημα $F = (V, E)$ είναι δάσος αν και μόνον αν κάθε συνεκτική συνιστώσα του είναι δέντρο.

Απόδειξη: Άμεσο καθώς το F είναι άκυκλο αν και μόνον αν κάθε συνεκτική του συνιστώσα δεν έχει κύκλους. □

Έστω ένα συνεκτικό γράφημα G. Κάθε δέντρο T ώστε $V(T) = V(G)$ και $E(T) \subseteq E(G)$, λέγεται *επικαλύπτον* δέντρο του G.

Figure 0.6: Δέντρο T - Επικαλύπτον δέντρο του G (0.2)



Συγκεντρώνουμε εδώ κάποια βασικά αποτελέσματα για τα δέντρα.

Θεώρημα 2. Έστω γράφημα $T = (V, E)$. Τα ακόλουθα είναι ισοδύναμα:

1. T δέντρο (συνεκτικό και άκυκλο).
2. Για κάθε $u, v \in V$, το $P(u, v)$ είναι μονοσύνολο.
3. T ακυκλικό και με την προσθήκη οποιασδήποτε ακμής δημιουργείται κύκλος.
4. T συνεκτικό και με την αφαίρεση οποιασδήποτε ακμής, σπάει η συνεκτικότητά του.
5. T συνεκτικό και $|E| = |V| - 1$.
6. T άκυκλο και $|E| = |V| - 1$.

Απόδειξη:

$(1 \Rightarrow 2)$ Αφού T συνεκτικό, $P(u, v) \neq \emptyset$. Άρα, υπάρχουν $u_1, \ldots, u_n \in V$ τέτοιες ώστε:

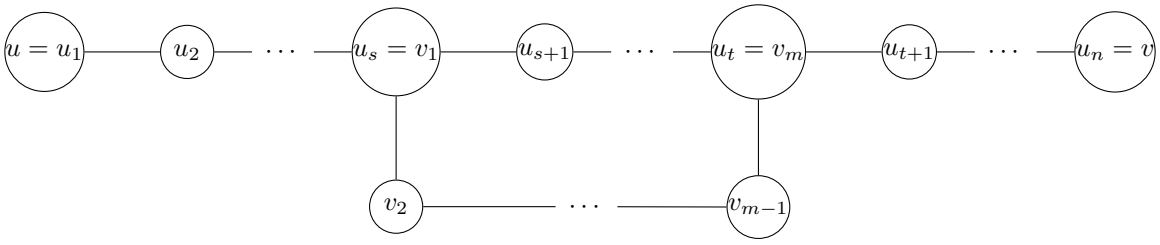
- $u_1 = u$, $u_n = v$,
- $\{u_i, u_i + 1\} \in E$, $i = 1, \ldots, n - 1$.

Έστω ότι $|P(u, v)| \geq 2$. Τότε, υπάρχουν $s \in \{1, \ldots, n - 1\}$, $t \in \{s + 1, \ldots, n\}$ και $v_1, \ldots, v_m \in V$ τέτοια ώστε:

- $v_1 = u_s$, $v_m = u_t$,
- $v_2, \ldots, v_{m-1} \notin \{u_1, \ldots, u_n\}$
- $\{v_k, v_k + 1\} \in E$, $k = 1, \ldots, m - 1$.

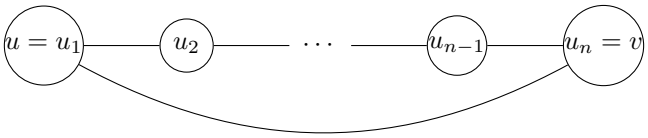
Έπεται ότι $(u_s = v_1, \ldots, v_m = u_t, u_{t-1}, \ldots, u_{s+1})$ κύκλος στο T. Άτοπο.

Figure 0.7: Θεώρημα 2, $(1 \Rightarrow 2)$



$(2 \Rightarrow 3)$ Έστω $u, v \in V$ τέτοιες ώστε $\{u, v\} \notin E$. Αφού $P(u, v) \neq \emptyset$, υπάρχουν $u = u_1, \ldots, u_n = v$ τέτοιες ώστε $\{u_i, u_{i+1}\} \in E, i = 1, \ldots, n-1$. Με την προσθήκη της $\{u, v\}$ στο T, έπεται ότι $(u = u_1, \ldots, u_n = v)$ κύκλος.

Figure 0.8: Θεώρημα 2, $(2 \Rightarrow 3)$



$(3 \Rightarrow 4)$ Έστω $u, v \in V$ κι έστω, προς άτοπο, ότι το $T' = (V, E \setminus \{\{u, v\}\})$ είναι συνεκτικό. Τότε, το $P(u, v) \neq \emptyset$ στο T' και, όπως στο $(2 \Rightarrow 3)$, με επαναπροσθήκη της u, v, δημιουργείται κύκλος στο T. Άτοπο.

$(4 \Rightarrow 5)$ Έστω $|V| = n$ και $G = (\emptyset, \emptyset)$ ένα κενό γράφημα. Θα προσθέτουμε μία μία τις κορυφές του T, μαζί με τις προσκείμενες ακμές τους, στο G. Θα ονομάζουμε u_i την κορυφή που προστίθεται στο i-οστό βήμα, $i = 1, \ldots, n$. Στο τέλος, θα έχουμε $G = T$. Δουλεύουμε επαγωγικά ως εξής:

- Έστω $u \in V$ μια οποιαδήποτε κορυφή. Την ονομάζουμε u_1 και θέτουμε $V(G) = \{u_1\}$. Προφανώς, $|E(G)| = |V(G)| - 1$.
- Αφού συνεκτικό, υπάρχει τουλάχιστον μία κορυφή $v \in V \setminus \{u_1\}$ ώστε $\{u_1, v\} \in E$. Την ονομάζουμε u_2 και θέτουμε $V(G) = \{u_1, u_2\}$, $E(G) = \{\{u_1, u_2\}\}$. Προφανώς, $|E(G)| = |V(G)| - 1$.
- Έστω ότι στα πρώτα $i-1$ βήματα, έχουμε $V(G) = \{u_1, \ldots, u_{i-1}\}$ και $|E(G)| = |V(G)| - 1, i = 2, \ldots, n$.
- Αφού T συνεκτικό, υπάρχει $w \in V \setminus \{u_1, \ldots, u_{i-1}\}$ και $j \in \{1, \ldots, i-1\}$ ώστε $\{w, u_j\} \in E$. Έστω ότι υπάρχει και $k \in \{1, \ldots, i-1\}$, $k \neq j$, με $\{w, u_k\} \in E$. Τότε, η αφαίρεση της $\{w, u_k\}$ από το T δεν σπάει την συνεκτικότητα. Άτοπο.

  Προσθέτοντας την w στο $V(G)$ και την $\{w, u_j\}$ στο $E(G)$, έχουμε πάλι $|E(G)| = |V(G)| - 1$. Επαγωγικά, μετά από n γύρους, $G = (V, E)$ και $|E(G)| = |V(G)| - 1$.

$(5 \Rightarrow 6)$ Έστω, προς άτοπο, ένας κύκλος C στο T μήκους k. Έστω $S = V \setminus V(C)$. Προφανώς $|S| = |V| - k$. Αφού T συνεκτικό, υπάρχει $u \in S$ και $v \in V(C)$, με $\{u, v\} \in E$.

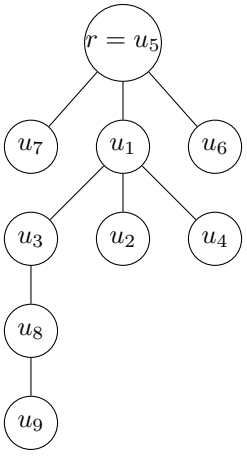
Αφού T συνεκτικό, υπάρχει $z \in S \setminus \{u\}$ και $w \in V(C) \cup \{u\}$, με $\{z, w\} \in E$. Επαγωγικά, αφού το S έχει $|V| - k$ κορυφές, βρίσκουμε έτσι και $|V| - k$ ακμές εκτός του C. Άτοπο, αφού $|E| = |V| - k$.

$(6 \Rightarrow 1)$ Έστω, προς άτοπο, ότι το T δεν είναι συνεκτικό. Έπεται ότι έχει τουλάχιστον δύο συνεκτικές συνιστώσες T_1, T_2. Έχουμε $|V(T_1)| + |V(T_2)| = |V(T)|$. Αφού το T είναι άκυκλο, τα T_1, T_2 είναι δέντρα. Άρα $|E(T_1)| + |E(T_2)| = |V(T_1)| + |V(T_2)| - 2 < |V(T)| - 1$. Άτοπο.

□

Έστω ένα δέντρο T. Διαλέγουμε μία κορυφή $r \in V(T)$, την ονομάζουμε *ρίζα* και θεωρούμε το δέντρο να κρέμεται από την ρίζα r.

Figure 0.9: Δέντρο T (0.6) με ρίζα $r = u_5$



Σε ένα δέντρο με ρίζα (T, r), χρησιμοποιούμε την εξής ορολογία:

- Μία κορυφή $u \in V(T)$ βρίσκεται στο *επίπεδο* $i \geq 0$ του T, ή αλλιώς το *βάθος* $\texttt{depth}(u)$ της u στο T είναι i, αν $d(r, u) = i$, με την ρίζα r να βρίσκεται στο επίπεδο 0. Στο δέντρο με ρίζα 0.9, έχουμε $\texttt{depth}(u_5) = 0$, $\texttt{depth}(u_7) = 1$, $\texttt{depth}(u_1) = 1$, $\texttt{depth}(u_6) = 1$, $\texttt{depth}(u_3) = 2$, $\texttt{depth}(u_2) = 2$, $\texttt{depth}(u_4) = 2$, $\texttt{depth}(u_8) = 3$, $\texttt{depth}(u_9) = 4$.
- Το ύψος του δέντρου ορίζεται ως:
$$h := \max_{u \in V} \texttt{depth}(u).$$
Στο δέντρο με ρίζα 0.9, $h = \texttt{depth}(u_9) = 4$.
- Αν $u, v \in V(T)$, με u σε κάποιο επίπεδο i, v στο επίπεδο $i + 1$ και $\{u, v\} \in E$, τότε u γονιός της v και v παιδί της u. Στο δέντρο με ρίζα 0.9, u_2 παιδί της u_1 και u_1 γονιός της u_2.
- Αν $u, v \in V(T)$, με u σε κάποιο επίπεδο i, v στο επίπεδο $j > i$ και το μοναδικό $P \in P(u, v)$ έχει μόνον κορυφές σε επίπεδα μεγαλύτερα του i, τότε u πρόγονος της v και v απόγονος της u. Στο δέντρο με ρίζα 0.9, u_8 απόγονος της u_1 και u_1 πρόγονος της u_8.
- Κορυφές με κοινό γονέα ονομάζονται αδέρφια. Στο δέντρο με ρίζα 0.9, u_3, u_2, u_4 αδέρφια.
- Βαθμός μιας κορυφής ορίζεται το πλήθος των παιδιών της. Στο δέντρο με ρίζα 0.9, $\deg u_7 = 0$, $\deg(u_1) = 3$ και $\deg(u_8) = 1$.
- *Φύλλο* ονομάζεται κάθε κορυφή βαθμού 0. Στο δέντρο με ρίζα 0.9, τα φύλλα του είναι οι κορυφές u_7, u_6, u_2, u_4, u_9.
- *Εσωτερική* ονομάζεται κάθε κορυφή που δεν είναι φύλλο. Στο δέντρο με ρίζα 0.9, οι εσωτερικές κορυφές του είναι οι κορυφές $r = u_5, u_1, u_3, u_8$.

Έστω $k \geq 1$. Ένα δέντρο ονομάζεται *k-αδικό*, αν κάθε εσωτερική κορυφή του έχει βαθμό το πολύ k. Ένα k-δικό δέντρο είναι *γεμάτο* αν κάθε κορυφή του είναι είτε φύλλο, είτε έχει βαθμό ακριβώς k. Ένα k-αδικό δέντρο ύψους h ονομάζεται *σχεδόν πλήρες*, αν όλες οι κορυφές του στα επίπεδα $0, 1, \ldots, h-1$ έχουν βαθμό ακριβώς k. Ένα k-αδικό δέντρο είναι *πλήρες* αν όλες οι κορυφές του έχουν βαθμό ακριβώς k.

Θα δουλεύουμε συνήθως με $k = 2$.

Figure 0.10: T_1 δυαδικό, T_2 γεμάτο δυαδικό

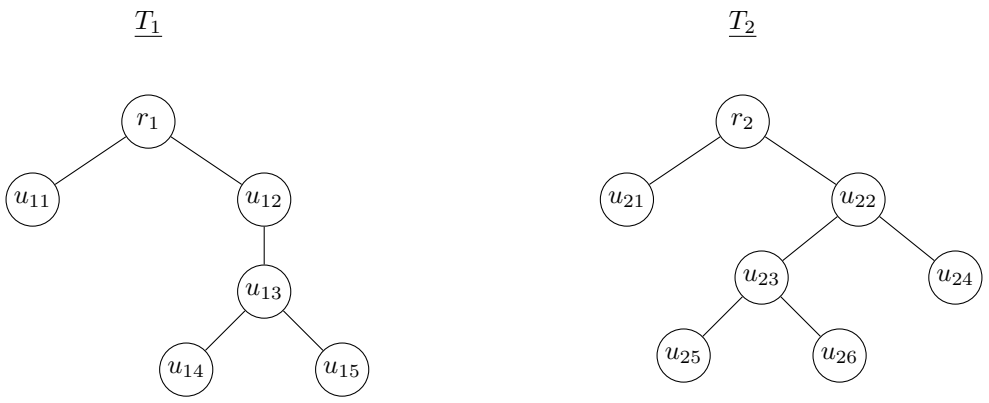
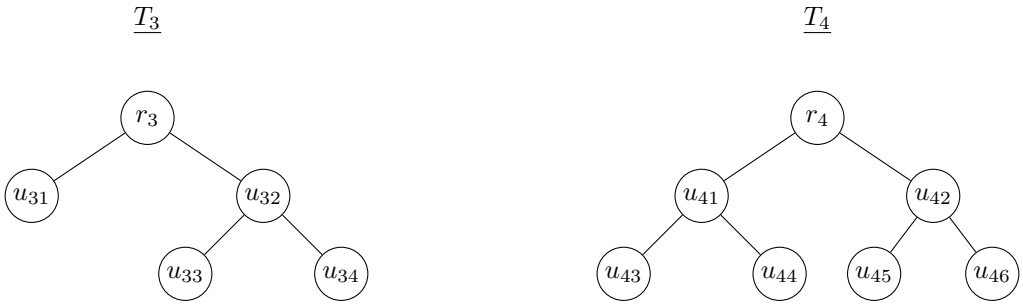


Figure 0.11: T_3 σχεδόν πλήρες δυαδικό, T_4 πλήρες δυαδικό



Πρόταση 2. Έστω ένα σχεδόν πλήρες δυαδικό δέντρο $T = (V, E)$, με $|V| = n$ και ύψος h. Τότε, $h = \Theta(\log(n))$.

Απόδειξη: Στο επίπεδο 0, έχουμε $2^0 = 1$ κορυφή. Στο επίπεδο 1, έχουμε $2^1 = 2$ κορυφές.

Έστω ότι στο επίπεδο $1 \leq i < h-1$ έχουμε 2^i κορυφές. Αφού κάθε τέτοια κορυφή έχει ακριβώς δύο παιδιά, στο επίπεδο $i+1$ θα έχουμε $2 \cdot 2^i = 2^{i+1}$ κορυφές. Επαγωγικά, για κάθε $1 \leq i \leq h-1$, έχουμε 2^i κορυφές.

Αφού το T είναι σχεδόν πλήρες δυαδικό, στο επίπεδο h θα έχουμε από 1 έως 2^h κορυφές. Συνολικά:

$$\begin{aligned}
&\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i \\
&2^h - 1 + 1 \leq n \leq 2^{h+1} - 1 \\
&2^h \leq n < 2 \cdot 2^h \\
&h \leq \log(n) < \log(2) + h \\
&h \leq \log(n) < h + 1 \\
&h = O(\log(n)) \text{ και } \log(n) = O(h) \\
&h = \Theta(\log(n)).
\end{aligned}$$

□

1 Πίνακες

Θεωρούμε τους πίνακες ως *στατικές* δομές δεδομένων, ότι καταλαμβάνουν δηλαδή έναν εξαρχής δηλωμένο και σταθερό αριθμό συνεχόμενων θέσεων μνήμης, παρότι υπάρχουν γλώσσες προγραμματισμού που υλοποιούν αλλιώς την συγκεκριμένη δομή (π.χ. `python`). Στον αντίποδα, ο συνήθης περιορισμός ότι σε έναν πίνακα τα δεδομένα που αποθηκεύονται πρέπει να είναι ίδιου τύπου δεν θα μας απασχολήσει (αν και θα προκύπτει κάπως "φυσιολογικά").

Στην γενική περίπτωση, ένας k-*διάστατος* πίνακας A, συμβολίζεται με (A, k), όπου $\dim(A) := k = (n_1 \cdot n_2 \cdots n_k)$, και το στοιχείο στη θέση $(i_1, i_2 \ldots, i_k)$ θα γράφεται $A[i_1, \ldots, i_k]$, όπου $1 \leq i_j \leq n_j$, $n_j \in \mathbb{N} \setminus \{0\}$, $j = 1, \ldots, k$, $k \in \mathbb{N} \setminus \{0\}$.

Στην πράξη, θα δουλεύουμε συνήθως με $k = 1, 2$ και θα μιλάμε για *μονοδιάστατους* και *δισδιάστατους* πίνακες.

Θα λέμε ότι ο A είναι ένας μονοδιάστατος πίνακας, εννοώντας ότι είναι ένας "πίνακας-γραμμή", δηλαδή $\dim(A) = 1 \times n$, για κάποιο $n \in \mathbb{N}$:

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix},$$

με $A[i] = a_i$, $i = 1, \ldots, n$. Αν θέλουμε έναν μονοδιάστατο "πίνακα-στήλη" B, θα γράφουμε ότι $\dim(B) = m \times 1$:

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix},$$

Ένας δισδιάστατος πίνακας C, είναι ένας πίνακας διαστάσεων $\dim(n, m)$, $n, m \in \mathbb{N}$:

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1j} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2j} & \cdots & c_{2m} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{im} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nj} & \cdots & c_{nm} \end{bmatrix},$$

με $C[i, j] = c_{ij}$, και τους μονοδιάστατους πίνακας $\texttt{row}(C, i)$, $\texttt{col}(C, j)$ i-οστής γραμμής και j-οστής στήλης αντίστοιχα, να ορίζονται ως εξής:

$$\texttt{row}(C, i) = \begin{bmatrix} c_{i1} & c_{i2} & \cdots & c_{in} \end{bmatrix},$$

και

$$\texttt{col}(C, j) = \begin{bmatrix} c_1 j \\ c_2 j \\ \vdots \\ c_n j \end{bmatrix},$$

$i = 1, \ldots, n$, $j = 1, \ldots, m$.

1.1 Βασικές λειτουργίες

Σε όλες τις δομές που θα ορίσουμε, θα μας απασχολούν αρχικά οι ακόλουθες έξι λειτουργίες:

1. αρχικοποίηση δομής,
2. έλεγχος για το αν η δομή είναι κενή,
3. προσπέλαση δομής,
4. προσθήκη στοιχείου,
5. διαγραφή στοιχείου,
6. αναζήτηση στοιχείου.

Ας δούμε τις λειτουργίες αυτές μία προς μία:

1. Για την αρχικοποίηση, θα έχουμε μια εντολή `new matrix` που θα παίρνει ως όρισμα τις διαστάσεις του υπό κατασκευή πίνακα:

$$A = \texttt{new matrix}(n_1, \ldots, n_k).$$

Το πως η υπολογιστική μηχανή μας δεσμεύει αυτές τις $n_1 \cdot n_2 \cdots n_k$ θέσεις μνήμης, ώστε να διακρίνει και τις k διαστάσεις, δεν μας απασχολεί. Το σημαντικό είναι ότι τόσο το k, όσο και τα $n_1, \ldots, n_k$ είναι σταθερές, οπότε θέλουμε σταθερό χρόνο $O(1)$ για την αρχικοποίηση. Για παράδειγμα, η εντολή $A = \texttt{new matrix}(5, 9)$ δημιουργεί έναν δισδιάστατο πίνακα με 5 γραμμές και 9 στήλες.

2. Ο έλεγχος αυτός στην περίπτωση των πινάκων δεν έχει νόημα, καθώς αυτοί αποτελούν στατικές δομές. Αν λοιπόν ένας πίνακας οριζόταν να μην πιάνει θέσεις μνήμης, αυτό δεν θα μπορούσε να αλλάξει. Οπότε, ακόμη κι αν υποθέσουμε ότι όλες οι θέσεις του δεν περιέχουν κάποια τιμή, θα πρέπει το πλήθος τους να είναι γνωστό και οι αντίστοιχες θέσεις μνήμης δεσμευμένες.

3. Η προσπέλαση της δομής σπανίως είναι αυτοσκοπός. Συνήθως θέλουμε να βρούμε το στοιχείο σε κάποια θέση της δομής ώστε να εφαρμόσουμε κάποια πράξη επάνω του, να το επιστρέψουμε στον χρήστη ή να το ενημερώσουμε. Προφανώς ο χρόνος που θα χρειαστούμε επηρεάζεται από το τι θα θελήσουμε να το κάνουμε. Σε κάθε περίπτωση, οι πίνακες είναι δομές *τυχαίας προσπέλασης* (βασισμένες στο μοντέλο *Random Access Memory (RAM)*). Αυτό σημαίνει ότι χρειαζόμαστε τον ίδιο, σταθερό χρόνο $O(1)$, για την πρόσβαση σε κάθε θέση τους. Δύο απλά παραδείγματα σταθερού, επίσης, χρόνου, είναι η εντολή:

$$\texttt{print} A[i, j],$$

που θα τύπωνε το στοιχείο a_{ij} του δισδιάστατου πίνακα A και η

$$B[i] = 10,$$

που θα ενημέρωνε την τιμή της θέσης i του πίνακα B σε 10.

4-5. Η προθήκη και η διαγραφή στοιχείου αντιστοίχως δεν έχουν νόημα στην περίπτωση των πινάκων. Αν υποθέσουμε ύπαρξη κενών θέσεων, η προσθήκη ανάγεται στην ενημέρωση της τιμής σε μια τέτοια θέση, ενώ η διαγραφή στοιχείου θα μπορούσε να αντιστοιχηθεί στην ενημέρωση μιας θέσης σε τιμή _ (κενό). Ως εκ τούτου, και οι δύο αυτές λειτουργίες γίνονται σε σταθερό χρόνο $O(1)$.

6. Η "φυσική" αναζήτηση ενός στοιχείου (που συχνά ονομάζεται *κλειδί*) που αντιστοιχεί σε έναν πίνακα οποιασδήποτε διάστασης, είναι η λεγόμενη *γραμμική*. Θα την δούμε σε έναν τρισδιάστατο πίνακα. Για περισσότερες διαστάσεις, απλώς αυξάνονται αντίστοιχα οι επαναληπτικές δομές που χρησιμοποιούμε:

Algorithm 10 LinSearch$(< A, key >)$

```
# A: πίνακας με dim(A) = p × q × r
# key: στοιχείο προς αναζήτηση
 1: for i = 1, ..., p do
 2:   for j = 1, ..., q do
 3:     for k = 1, ..., r do
 4:       if A[i, j, k] = key then
 5:         return (i, j, k)
 6:       end if
 7:     end for
 8:   end for
 9: end for
10: return −1
```

Σημειώνουμε ότι στην συγκεκριμένη περίπτωση, ο πίνακας και το κλειδί δεν χρειάζεται να είναι αριθμοί. Ούτε καν το κλειδί να είναι τύπος στοιχείου σαν αυτά που περιέχει ο A. Αρκεί να είναι στοιχεία όπου ο έλεγχος της ισότητας μεταξύ τους να γίνεται σε σταθερό $O(1)$ χρόνο. Επίσης υποθέτουμε ότι είναι στοιχεία που αποθηκεύονται σε μία θέση μνήμης.

Αν στο βήμα 4 ο έλεγχος βγει αληθής, τότε το βήμα 5 διακόπτει την λειτουργία του αλγορίθμου και επιστρέφει την θέση που βρέθηκε το στοιχείο key. Αν και οι τρεις επαναληπτικές διαδικασίες ολοκληρωθούν χωρίς να βρεθεί το key, επιστρέφεται μια τιμή που δεν θα μπορούσε ποτέ να είναι "σωστή" απάντηση (συχνά η σύμβαση είναι αυτή η τιμή να είναι το -1). Ο χρόνος είναι εύκολα $O(pqr)$, που ταυτίζεται και με το μέγεθος της εισόδου, άρα γραμμικός ως προς το μέγεθος της εισόδου. Εξού και το όνομα "γραμμική αναζήτηση".

1.2 Φραγμένοι πίνακες

Σε αυτή την ενότητα, θα περιοριστούμε σε πίνακες με ακέραιες τιμές.

Ορισμός 1.1. Ένας πίνακας A με $\dim(A) = n_1 \times \cdots \times n_k$, θα λέγεται *φραγμένος* αν υπάρχει $M \in \mathbb{Z}$ με:

$$-M \leq A[i_1, \ldots, i_k] \leq M,$$

για κάθε $0 \leq i_j \leq n_j$, $j = 1, \ldots, k$, $k \in \mathbb{N}$.

Δεδομένου ότι ένας ηλεκτρονικός υπολογιστής έχει άνω και κάτω όρια στις ακέραιες τιμές που μπορεί να υποστηρίξει, παρατηρούμε ότι αυτός ο ορισμός ανταποκρίνεται στην πραγματική συνθήκη που αντιμετωπίζουμε στην πράξη, κι ως εκ τούτου δεν είναι τόσο περιοριστικός όσο ίσως φαίνεται.

Ας δούμε έναν αλγόριθμο ταξινόμησης ενός μονοδιάστατου πίνακα με θετικές τιμές που το εκμεταλλεύεται αυτό. Τόσο η αύξηση των διαστάσεων του πίνακα, όσο και η δυνατότητα να παίρνει αρνητικές τιμές (ή 0), δεν αλλάζει κάτι στη λογική του αλγορίθμου, αλλά αυξάνει σαφώς την περιπλοκότητά του. Χρησιμοποιούμε την συντομογραφία $x += 1$ αντί για $x = x+1$.

Algorithm 11 BucketSort$(< A, M >)$

```
# A: 1 × n πίνακας, τα στοιχεία του οποίου ταξινομούμε σε αύξουσα σειρά.
# M ∈ ℤ : 1 ≤ A[i] ≤ M, i = 1, ..., n.
 1: B = new matrix(M)
 2: for j = 1, ..., M do
 3:     B[j] = 0
 4: end for
 5: for i = 1, ..., n do
 6:     B[A[i]] += 1
 7: end for
 8: l = 1
 9: for j = 1, ..., M do
10:     for k = 1, ..., B[j] do
11:         A[l] = j
12:         l += 1
13:     end for
14: end for
15: return A
```

Ο αλγόριθμος δουλεύει ως εξής: αρχικά, δημιουργεί έναν βοηθητικό πίνακα B, με τόσες θέσεις όσες αντιστοιχούν στο εύρος τιμών που μπορεί να έχουν τα στοιχεία του A και αρχικοποιεί όλες τις θέσεις του στο 0. Στην συνέχεια, διατρέχει τον πίνακα A και, για κάθε τιμή $A[i]$ που βλέπει, αυξάνει κατά 1 την αντίστοιχη θέση του πίνακα B (θυμίζουμε ότι οι πίνακες είναι τυχαίας προσπέλασης). Τέλος, ενημερώνει τις τιμές του πίνακα A, βάζοντας σε αυτόν $B[j]$ φορές τον αριθμό j, $j = 1, \ldots, M$.

Ο χρόνος που θα χρειαστεί ο αλγόριθμος είναι:

- $O(M)$ για την αρχικοποίηση του B.
- $O(n)$ για να διατρέξει τον A και να ενημερώσει κατάλληλα τις τιμές του B.
- Λόγω του σχεδιασμού του αλγορίθμου, πρέπει να ισχύει:

$$\sum_{j=1}^{M} B[j] = n.$$

  Έπεται ότι η ενημέρωση των τιμών του A θα χρειαστεί $O(n + M)$ χρόνο.

Συνολικά, ο **BucketSort** θα χρειαστεί $O(n + M)$ χρόνο. Είδαμε στην Ενότητα 0, τον αλγόριθμο ταξινόμησης **BubbleSort**, που χρειάζεται $O(n^2)$ χρόνο. Είναι επίσης γνωστό ότι η ταξινόμηση μονοδιάστατου πίνακα n θέσεων μπορεί να γίνει σε $O(n \log(n))$ χρόνο. Τι είναι καλύτερο;

Παρατηρούμε αρχικά ότι αν $M >> n$ (π.χ. n^3), τότε σίγουρα ο **BucketSort** δεν είναι καλή επιλογή. Αν από την άλλη $M << n$ σε τέτοιο βαθμό ώστε να μπορούσαμε να γράψουμε ότι $O(n+M) = O(n)$, τότε όντως έχουμε κάνει κάτι καλύτερο. Πόσο μικρότερο όμως θα έπρεπε να είναι το M από το n;

Βάσει όσων έχουμε πει, θα αρκούσε $M = O(n)$. Φτάνοντας σε ζητήματα που ξεφεύγουν από το ενδιαφέρον αυτών των σημειώσεων, αναφέρουμε απλώς πως δυστυχώς αυτό δεν αρκεί. Αυτό συμβαίνει καθώς η πολυπλοκότητα μετριέται ως προς το μέγεθος της εισόδου $| < A, M > |$. Αυτό, ενώ εκ πρώτης όψεως είναι $O(n + M)$, στην πραγματικότητα είναι $O(n + \log(M))$: το M είναι ένας ακέραιος αριθμός, που χρειάζεται $\log(M)$ δυαδικά ψηφία για να κωδικοποιηθεί. Ως εκ τούτου, ο **BucketSort** τρέχει σε $O(n + 2^{\log(M)})$ χρόνο, που εξαρτάται εκθετικά από την τιμή του M. Ως εκ τούτου, για σχετικά μικρές τιμές του M ($M = 100$), δημιουργείται ήδη πρόβλημα.

1.3 Ταξινομημένοι πίνακες

Έστω ότι έχουμε ταξινομημένους πίνακες αριθμών. Θα δούμε μία "κλασσική" διαδικασία αναζήτησης σε τέτοιους πίνακες, που χρειάζεται πολύ λιγότερο χρόνο από την γραμμική αναζήτηση που είδαμε πριν. Με την ευκαιρία, θα εξετάσουμε και τον πρώτο *αναδρομικό αλγόριθμο*, έναν αλγόριθμο δηλαδή που για να υπολογίσει την έξοδό του για κάποια είσοδο, καλεί τον εαυτό του για μικρότερου μεγέθους εισόδους.

Η πιο "κλασσική" αναδρομική συνάρτηση ίσως είναι η $f(n) = n!$, με αναδρομικό τύπου $f(n) = (n-1)!n$ και $f(0) = 1$. Ένας αναδρομικός αλγόριθμος θα καλούσε τον εαυτό του για όλο και μικρότερα n, μέχρι να φτάσει στην τιμή 0. Στην συνέχεια, θα επανερχόταν πολλαπλασιάζοντας το ήδη υπολογισμένο $f(n-i)$ με το $(n-i+1)$, $i = n, \ldots, 1$.

Έστω $\lfloor x \rfloor$ το *κάτω ακέραιο μέρος* του x και, για πίνακα A $1 \times n$,

$$A[i:j] := \left[a_i, a_{i+1}, \ldots a_{j-1}\right],$$

για κάθε $1 \leq i < j \leq n$. Επίσης έστω $\texttt{len}(A)$ το πλήθος στοιχείων του A.

Algorithm 12 BinarySearch$(< A, key >)$

```
# A: 1 × n πίνακας πραγματικών τιμών, με στοιχεία σε αύξουσα σειρά.
# key: πραγματικός αριθμός που θέλουμε να ελέγξουμε αν είναι στον A.
1: if len(A) = 1 and A[1] ≠ key then
2:     return −1
3: else if A[⌊n/2⌋] = key then
4:     return ⌊n/2⌋
5: else if A[⌊n/2⌋] > key then
6:     return BinarySearch(< A[1 : ⌊n/2⌋], key >)
7: else
8:     return BinarySearch(< A[⌊n/2⌋ + 1 : n + 1], key >)
9: end if
```

Ο παραπάνω αλγόριθμος, ελέγχει αν το μεσαίο στοιχείο του πίνακα είναι το key. Αν είναι μικρότερο, επαναλαμβάνει την διαδικασία στο πάνω μισό του A, αλλιώς πάει στο κάτω μισό του A. Αν φτάσει αναδρομικά σε πίνακα μήκους 1 και δεν βρει ούτε εκεί το key, επιστρέφει -1.

Έστω $T(n)$ η χρονική πολυπλοκότητα του **BinarySearch**. Για ευκολία, υποθέτουμε ότι $n = 2^k$ για κάποιο $k \in \mathbb{N}$. Από την λειτουργία του αλγορίθμου, έχουμε $T(n) = T(n/2) + 1 = T(n/2^2) + 2 = \cdots = T(n/2^k) + k = T(1) + k = 1 + k = O(\log(n))$. Η υπόθεση ότι $n = 2^k$ μπορεί θεωρητικά να υλοποιηθεί εύκολα: βρίσκουμε το ελάχιστο k ώστε $n \leq 2^k$ και προσθέτουμε $2^k - n$ αντίτυπα του μέγιστου στοιχείου του A στον ίδιον τον A. Στην πράξη, αυτό μπορεί να είναι ιδιαίτερα χρονοβόρο. Η αναδρομή μπορεί να λυθεί και χωρίς την απολοποιητική υπόθεση, αλλά δεν είναι αντικείμενο των σημειώσεων αυτών.

2 Συνδεδεμένες Λίστες

Παρά την ταχύτητα που επιτυγχάνουν στις βασικές λειτουργίες τους οι στατικές δομές δεδομένων των πινάκων (υποενότητα 1.1), έχουν το προφανές μειονέκτημα ότι πρέπει εξαρχής να γνωρίζουμε έστω ένα άνω όριο στον όγκο των δεδομένων που θα έχουμε να διαχειριστούμε. Ως εκ τούτου, θα θέλαμε να έχουμε και μία *δυναμική* δομή δεδομένων, στην οποία να μπορούμε να προσθέτουμε ή να αφαιρούμε στοιχεία κατά το δοκούν. Το βασικό ζήτημα που προκύπτει από μια τέτοια δομή, είναι ότι μια υπολογιστική μηχανή δεν μπορεί να την αποθηκεύσει σε συνεχόμενες θέσεις μνήμης.

Φανταστείτε σε μια οποδήποτε στιγμή τον Η/Υ σας. Ασχέτως με το πόσα πράγματα κάνετε εκείνη την στιγμή ταυτόχρονα (browsing, σχόλια/αναρτήσεις σε κάποιο social media, εκτέλεση κάποιου script) υπάρχουν ένα σωρό άλλα προγράμματα που τρέχουν ταυτόχρονα στο παρασκήνιο, χωρίς να φαίνονται, για να ρυθμίζουν διάφορες λειτουργίες του μηχανήματός σας. Ως εκ τούτου, διαφορετικά προγράμματα καταχωρούν, ενημερώνουν και σβήνουν συνεχώς πράγματα στις θέσεις μνήμης. Αν λοιπόν έχουμε αποθηκευμένο κάποιο σύνολο δεδομένων σε μια σειρά θέσεων μνήμης, και θέλουμε να προσθέσουμε ένα στοιχείο ακόμη, δεν υπάρχει καμία εγγύηση ότι η επόμενη θέση μνήμης στην συγκεκριμένη σειρά είναι ελεύθερη (ζήτημα που λυνόταν με τους πίνακες καθώς εξαρχής δέσμευαν όλο το χώρο που θα χρειαστούν).

Θα χρειαστούμε λοιπόν έναν "σύνθετο" τύπο αντικειμένου που θα ονομάζουμε *κόμβο* (node). Ένας κόμβος u θα έχει δύο μέρη:

- Στο ένα μέρος θα έχουμε το περιεχόμενο του κόμβου, δηλαδή το δεδομένο που έχει αποθηκευτεί στην συγκεκριμένη θέση μνήμης που καταλαμβάνει ο χώρος. Για να αποκτήσουμε πρόσβαση σε αυτό, θα γράφουμε $u.\texttt{val}$
- Στο άλλο μέρος θα έχουμε έναν πεπερασμένο αριθμό *δεικτών* $\texttt{pnt}_1, \texttt{pnt}_2, \ldots, \texttt{pnt}_d$, $d \geq 1$. Οι δείκτες αυτοί θα δείχνουν διευθύνσεις μνήμες που αποθηκευμένοι άλλοι κόμβοι. Για να αποκτήσουμε πρόσβαση σε κάποιον δείκτη, θα γράφουμε $u.\texttt{pnt}_i$, $i = 1, \ldots, k$.

Χρησιμοποιήσουμε το λεγόμενο "dot notation" (συμβολισμό της τελείας), που αποτελεί κλασσικό τρόπο γραφής σε γλώσσες προγραμματισμού για να αποκτάμε πρόσβαση στο κάθε μέρος ή λειτουργία ενός σύνθετου τύπου αντικειμένου.

Θα λέμε ότι ο κόμβος u *δείχνει* έναν κόμβο v, αν κάποιος από τους δείκτες $\texttt{pnt}_i$ περιέχει την διεύθυνση μνήμης που έχει αποθηκευτεί ο v. Δύο κόμβοι θα λέγονται *συνδεδεμένοι*, αν κάποιος από τους δύο δείχνει στον άλλον.

Τόσο η τιμή, όσο και ο δείκτης ενός κόμβου, θα μπορούν να είναι κενά. Θα γράφουμε $u.\texttt{val} = \emptyset$ και $u.\texttt{pnt} = \texttt{NIL}$. Στην περίπτωση που η τιμή είναι κενή, μερικές φορές θα μιλάμε για *κόμβο-δείκτη* και θα κρατάμε μόνο την τιμή του δείκτη, ταυτίζοντάς την με τον κόμβο (δηλαδή $u = u.\texttt{pnt}$).

Εν γένει θα δουλεύουμε με έναν δείκτη σε κάθε κόμβο. Στην περίπτωση αυτή, οι κόμβοι μας θα είναι της μορφής:

Figure 2.1: Κόμβος u και κόμβος-δείκτης v

$$u = \boxed{u.\texttt{val} \mid u.\texttt{next}}$$

$$\boxed{v.\texttt{pnt} = v}$$

όπου ο δείκτης $u.\texttt{pnt}$ γράφεται $u.\texttt{next}$ για να μας δίνει την διαίσθηση του επομένου.

Έστω u ένας κόμβος, a μία διεύθυνση μνήμης και x ένα αντικείμενο/δεδομένο που θέλουμε να αποθηκεύσουμε. Θα χρησιμοποιούμε τις εξής εντολές:

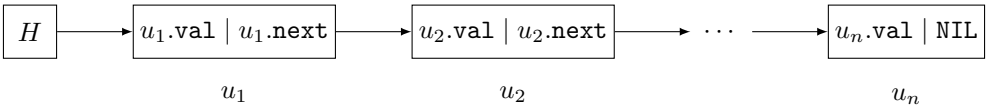
- $\texttt{addr}(u)$: επιστρέφει την διεύθυνση μνήμης στην οποία είναι αποθηκευμένος ο u.
- $\texttt{node}(a)$: επιστρέφει τον κόμβο που είναι αποθηκευμένος στην διεύθυνση a (ή `None` αν δεν υπάρχει τίποτα στην συγκεκριμένη θέση μνήμης).
- $v = \texttt{new node}(x)$: δημιουργεί τον κόμβο v, τον αποθηκεύει σε κάποια διεύθυνση στην μνήμη και θέτει $u.\texttt{val} = x$ και $u.\texttt{next} = \texttt{NIL}$.

Ορισμός 2.1. Μία *(συνδεδεμένη) λίστα* είναι μια πεπερασμένη ακολουθία κόμβων $u_0, u_1, \ldots, u_n$ όπου:

- u_0 είναι ένας κόμβος-δείκτης, με $u_0.\texttt{next} = \texttt{addr}(u_1)$,
- κάθε κόμβος u_i δείχνει τον u_{i+1}, δηλαδή $u_i.\texttt{next} = \texttt{addr}(u_{i+1})$, $i = 1, \ldots, n-1$ και
- $u_n.\texttt{next} = \texttt{NIL}$.

Για το u_0 συχνά θα χρησιμοποιούμε άλλα γράμματα που να δείχνουν και τον τύπο της δομής στην οποία δείχνει. Π.χ. H για να δείχνει την κεφαλή (head) μιας λίστας ή T για την ρίζα ενός δέντρου.

Figure 2.2: Λίστα με κεφαλή H



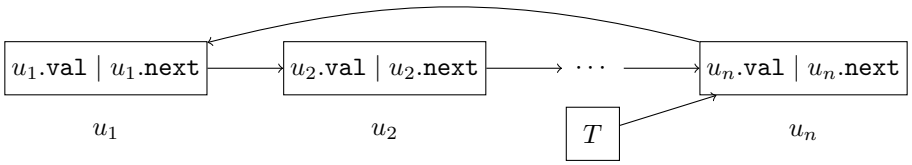
Τις παραπάνω λίστες θα τις λέμε καμιά φορά και *απλές*, όταν θέλουμε να τις διακρίνουμε από πιο σύνθετες λίστες. Ας δούμε δύο τέτοια ήδη.

Ορισμός 2.2. Μία *κυκλική* (συνδεδεμένη) λίστα είναι μια πεπερασμένη ακολουθία κόμβων $u_0, u_1, \ldots, u_n$ όπου:

- u_0 είναι ένας κόμβος-δείκτης, με $u_0.\texttt{next} = \texttt{addr}(u_n)$,
- κάθε κόμβος u_i δείχνει τον u_{i+1}, δηλαδή $u_i.\texttt{next} = \texttt{addr}(u_{i+1})$, $i = 1, \ldots, n-1$ και
- $u_n.\texttt{next} = \texttt{addr}(u_1)$.

Σε μια κυκλική λίστα, το ποιος κόμβος είναι ο πρώτος και ποιος ο τελευταίος αποτελεί απλή σύμβαση, που δεν έχει κάποια πραγματική σημασία. Χρειαζόμαστε παρ' όλα αυτά τον κόμβο-δείκτη u_0 να μας δείχνει κάπου, ώστε να έχουμε πρόσβαση στην λίστα. Κατά σύμβαση, συνηθίζεται να θεωρούμε ότι ο κόμβος στον οποίο δείχνει είναι ο τελευταίος, η *ουρά* (tail) δηλαδή της λίστας, οπότε χρησιμοποιούμε συχνά και το T αντί για το $u_0.\texttt{next}$.

Figure 2.3: Κυκλική λίστα με ουρά T

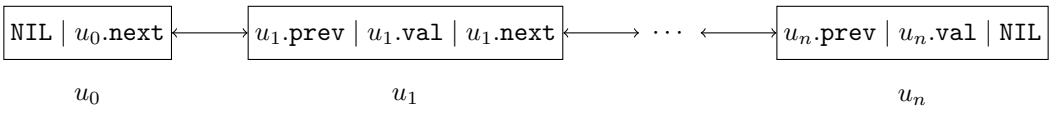


Ο τελευταίος τύπος λίστας θα χρησιμοποιεί κόμβους με δύο δείκτες. Αντί για $u.\texttt{pnt}_1$ και $u.\texttt{pnt}_2$, θα χρησιμοποιούμε τα ονόματα $u.\texttt{next}$ και $u.\texttt{prev}$ για να αντιστοιχούν στον επόμενο και τον προηγούμενο κόμβο στη σειρά που δίνεται από την λίστα. Ένας τέτοιος κόμβος θα δημιουργείται από την εντολή `new double node`.

Ορισμός 2.3. Μία *διπλά* (συνδεδεμένη) λίστα είναι μια πεπερασμένη ακολουθία κόμβων $u_0, u_1, \ldots, u_n$ όπου:

- u_0 είναι ένας κόμβος-δείκτης, με $u_0.\texttt{next} = \texttt{addr}(u_1)$ και $u_0.\texttt{prev} = \texttt{NIL}$,
- κάθε κόμβος u_i δείχνει τον u_{i+1}, δηλαδή $u_i.\texttt{next} = \texttt{addr}(u_{i+1})$, $i = 1, \ldots, n-1$,
- κάθε κόμβος u_i δείχνει τον u_{i-1}, δηλαδή $u_i.\texttt{prev} = \texttt{addr}(u_{i-1})$, $i = n, \ldots, 2$ και
- $u_n.\texttt{next} = \texttt{NIL}$.

Figure 2.4: Διπλή λίστα με κεφαλή H



2.1 Βασικές λειτουργίες λιστών

Όπως και για του πίνακες (υποενότητα 1.1), μας απασχολούν οι ακόλουθες έξι λειτουργίες:

1. αρχικοποίηση λίστας,
2. έλεγχος για το αν η λίστα είναι κενή,
3. προσπέλαση λίστας,
4. προσθήκη στοιχείου σε λίστα,
5. διαγραφή στοιχείου από λίστα,
6. αναζήτηση στοιχείου σε λίστα.

Εξετάζουμε τις λειτουργίες αυτές μία προς μία:

1. Η αρχικοποίηση μιας λίστας (οποιουδήποτε είδους), η οποία θα είναι κενή, ισοδυναμεί με την δημιουργία ενός κόμβου-δείκτη. Ως εκ τούτου, μπορεί να γίνει απλώς με την εντολή:

$$u = \texttt{new node}(),$$

όπου το όρισμα στην εντολή `new node` μένει κενό, καθώς ο κόμβος-δείκτης δεν έχει κάποια τιμή. Για τις διπλές λίστες, θα χρειαζόμαστε την εντολή `new double node`. Ειδικά για της απλές λίστες, θα γράφουμε μερικές φορές και $H = []$. Αυτό γίνεται σε σταθερό $O(1)$ χρόνο.

2. Ο έλεγχος για το αν έχει στοιχεία μία λίστα (οποιουδήποτε είδους), ισοδυναμεί με τον έλεγχο για το αν ο κόμβος-δείκτης δείχνει κάπου ή όχι. Αυτό γίνεται σε σταθερό $O(1)$ χρόνο.

Algorithm 13 IsEmpty$(< p >)$

```
# p : κόμβος-δείκτης που μας δίνεται ως δείκτης μιας λίστας
1: if p = NIL then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

3. Όπως και στους πίνακες, η προσπέλαση μιας λίστας σπανίως είναι αυτοσκοπός. Ας υποθέσουμε ότι θέλουμε να επιστρέψουμε την διεύθυνση και την τιμή του k-οστού κόμβου μιας λίστας. Εν γένει, δεν θα μας απασχολεί ο λεγόμενος "αμυντικός προγραμματισμός", το να ελέγχουμε δηλαδή αν η είσοδος στους αλγορίθμους μας είναι σωστή ώστε να τρέξει ο αλγόριθμος. Παρ' όλα αυτά, κάποιους ελάχιστους ελέγχους θα τους κάνουμε καμιά φορά. Στην συγκεκριμένη περίπτωση, θα έχουμε μια συνθήκη τερματισμού σε περίπτωση που η λίστα μας έχει λιγότερους από k κόμβους.

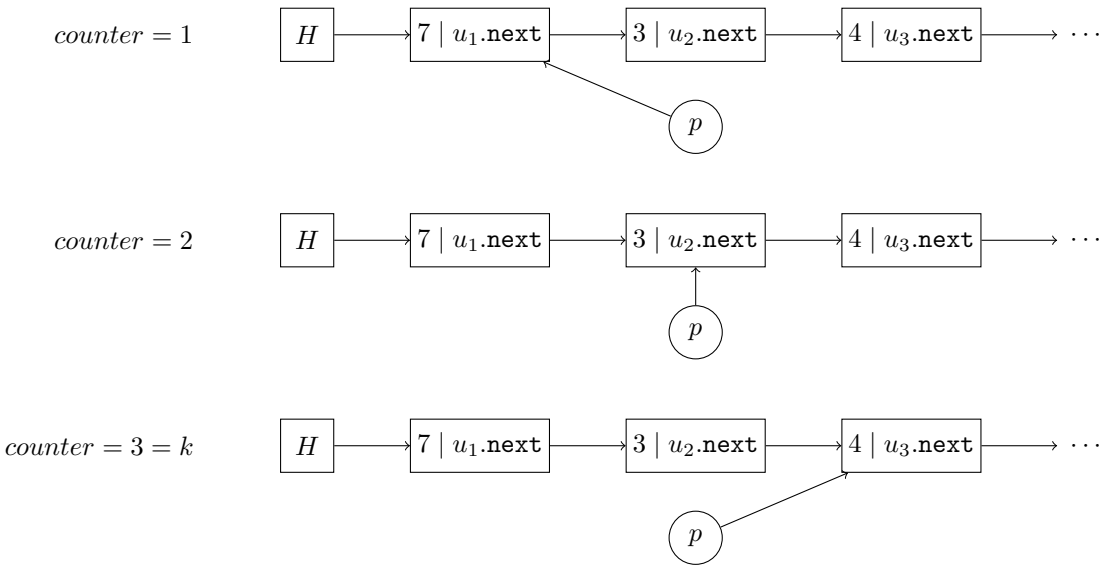
Algorithm 14 kthElement$(< H, k >)$

\# H : κόμβος-δείκτης κεφαλής μιας απλής ή διπλής λίστας
\# k: θετικός ακέραιος

1: $counter = 1, p = H$
2: **while** $p \neq$ `NIL` **do**
3: **if** $counter = k$ **then**
4: **return** p, `node`(p).`val`
5: **else**
6: $p =$ `node`(p).`next`, $counter += 1$
7: **end if**
8: **end while**
9: **return** -1

Ο αλγόριθμος **kthElement** προχωράει από κόμβο σε κόμβο, ενημερώνοντας τον δείκτη p σε κάθε βήμα ως τον δείκτη του επομένου από τον κόμβο που δείχνει και κρατώντας έναν μετρητή για το σε ποιον κόμβο βρίσκεται. Αν φτάσει στο k-οστό στοιχείο, τερματίζει την λειτουργία του επιστρέφοντας την διεύθυνση και την τιμή του κόμβου που είναι αποθηκευμένος εκεί. Αν φτάσει στο τέλος της λίστας πριν ο μετρητής πάρει την τιμή k, επιστρέφει -1. Ο χρόνος εκτέλεσης στην χειρότερη περίπτωση είναι $O(n)$, για λίστα με n κόμβους.

Figure 2.5: **kthElement**$(< H, k = 3 >)$



`return` `addr`(u_3), 4

Τι θα άλλαζε αν είχαμε κυκλική λίστα; Αρχικά, ο έλεγχος της γραμμής 2 δεν θα τερμάτιζε ποτέ, εκτός αν η λίστα μας ήταν κενή. Η μία εκδοχή θα ήταν να διορθώναμε την συνθήκη αυτή ώστε ο αλγόριθμος να τερμάτιζε όταν είχε διανύσει μια φορά έναν ολόκληρο κύκλο. Θα δούμε αυτήν την εκδοχή σε επόμενες λειτουργίες των λιστών.

Algorithm 15 kthCyclic$(< T, k >)$

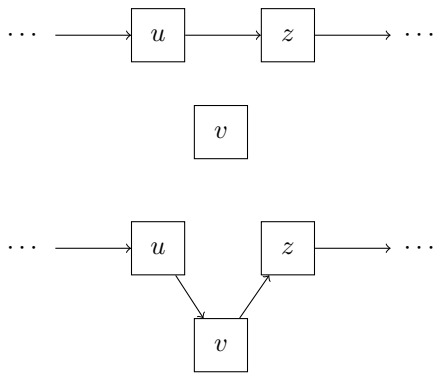
T : κόμβος-δείκτης ουράς μιας κυκλικής λίστας
k: θετικός ακέραιος

1: $counter = 1, p = T$
2: **while** $counter < k$ **do**
3: $\quad p = \texttt{node}(p).\texttt{next}, counter += 1$
4: **end while**
5: **return** $p, \texttt{node}(p).\texttt{val}$

Για μια λίστα με n κόμβους, ο **kthCyclic** επιστρέφει την διεύθυνση του $(k \mod n)$-οστού στοιχείου σε $O(k)$ χρόνο.

4. Έστω ότι θέλουμε να προσθέσουμε έναν νέο κόμβο σε μια λίστα. Με κάποιον τρόπο πρέπει να μας προσδιορίζεται το που θα μπει αυτό: στην αρχή, στο τέλος, μετά από κάποιον κόμβο, σε συγκεκριμένη διεύθυνση ή ως το k-οστό στοιχείο της λίστας. Θα υποθέσουμε ότι μας έχει δοθεί ο κόμβος έπειτα από τον οποίο θέλουμε να μπει ο νέος κόμβος.

Figure 2.6: **InsertAfter**$(< u, v >)$



Algorithm 16 InsertAfter$(< u, v >)$

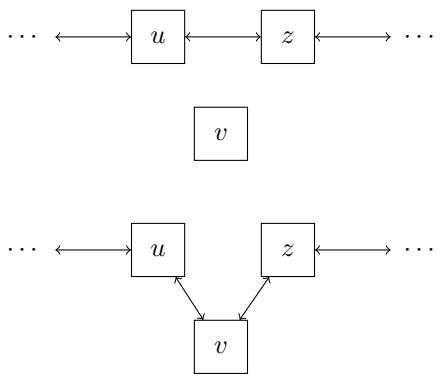
u : κόμβος μετά τον οποίο θέλουμε να προσθέσουμε τον κόμβο v

1: $v.\texttt{next} = u.\texttt{next}$
2: $u.\texttt{next} = \texttt{addr}(v)$

Ο αλγόριθμος χρειάζεται εμφανώς $O(1)$ χρόνο για να ολοκληρωθεί και δεν επιστρέφει τίποτα, καθώς ενημερώνει απλώς μια υπάρχουσα λίστα. Ο κόμβος v αρχικά ενημερώνεται να δείχνει όπου δείχνει και ο u. Στην συνέχεια ο u ενημερώνεται να δείχνει στην θέση μνήμης που είναι αποθηκευμένος ο v. Παρατηρείστε ότι όλες οι ακραίες περιπτώσεις (ο u να είναι ο κόμβος-δείκτης της κεφαλής ή ο τελευταίος), γίνονται σωστά με τον παραπάνω κώδικα.

Η κυκλική λίστα δεν παρουσιάζει καμία διαφορά ως προς αυτή τη λειτουργία. Στην διπλή λίστα, έχουμε την δυσκολία ότι δεν έχουμε άμεση πρόσβαση στον κόμβο που βρίσκεται μετά τον u.

Figure 2.7: **dInsertAfter**$(< u, v >)$



Algorithm 17 dInsertAfter$(< u, v >)$

u : κόμβος μετά τον οποίο θέλουμε να προσθέσουμε τον κόμβο v

1: v.next $= u$.next
2: v.prev $=$ addr(u)
3: **if** u.next $\neq$ NIL **then**
4: node$(u$.next$)$.prev $=$ addr(v)
5: **end if**
6: u.next $=$ addr(v)

Ο έλεγχος στην γραμμή 3 γίνεται καθώς αν ο u ήταν ο τελευταίος στην λίστα, δεν μπορεί να γίνει το συγκεκριμένο βήμα. Και πάλι, ο αλγόριθμος χρειάζεται $O(1)$ χρόνο για να ολοκληρωθεί.

Προτού περάσουμε στην διαγραφή κόμβων, ας δούμε μία εφαρμογή όσων έχουμε δει μέχρι τώρα.

Άσκηση 2.1. Να βρεθεί η κλάση ισοδυναμίας του $k \in \mathbb{Z}$ στον δακτύλιο $\mathbb{Z}_n := \{[0], [1], [2], \ldots, [n-1]\}$, όπου $[m] = \{z \in \mathbb{Z} \mid z \equiv m \mod n\}$, $0 \leq m \leq n-1$.

Λύση:

Algorithm 18 ModRing$(< k, n >)$

n, k : μη αρνητικοί ακέραιοι

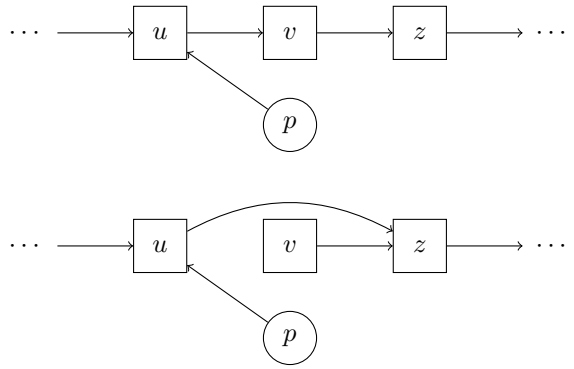
1: $T =$ new node$()$
2: $u =$ new node(0)
3: $T =$ addr(u)
4: **for** $i = 1, \ldots, n-1$ **do**
5: $v =$ new node(i)
6: u.next $=$ addr(v)
7: $u = v$
8: **end for**
9: u.next $= T$
10: $T =$ addr(u)
11: **return kthCyclic**$(< T, k >)$

Ο αλγόριθμος αρχικά κατασκευάζει τον κόμβο-δείκτη T και τον κόμβο με τιμή 0, την διεύθυνση του οποίου στην συνέχεια αποθηκεύει στον T, ώστε να μπορέσει να φτιάξει κυκλική λίστα. Παρατηρείστε ότι ενώ στην συνέχεια ο κόμβος u συνεχώς ενημερώνεται, ο T παραμένει να κρατάει την αρχική του διεύθυνση μέχρι και το βήμα 10. Στην συνέχεια, ο **ModRing** κατασκευάζει διαδοχικά κόμβους με τιμές $1, \ldots, n-1$. Στην αρχή κάθε επανάληψης, ο αλγόριθμος θέτει τον u να δείχνει τον v που μόλις κατασκευάστηκε, ενώ στο τέλος, η μεταβλητή u ενημερώνεται με τα στοιχεία του v.

Συνολικά, ο αλγόριθμος αυτός θα χρειαστεί $O(n)$ βήματα για την κατασκευή της κυκλικής λίστας, και $O(k)$ βήματα για την εύρεση της κλάσης, οπότε $O(n+m)$ συνολικά.

5. Έστω ότι θέλουμε να διαγράψουμε έναν κόμβο από μια λίστα. Όπως και με την εισαγωγή κόμβου, με κάποιον τρόπο πρέπει να μας προσδιορίζεται το ποιος θα είναι αυτός: ο πρώτος, ο τελευταίος, ο επόμενος κάποιου κόμβου, ο κόμβος που βρίσκεται σε συγκεκριμένη διεύθυνση ή ο k-οστός της λίστας. Θα υποθέσουμε ότι μας έχει δοθεί δείκτης που δείχνει τον προηγούμενο κόμβο από αυτόν που θέλουμε να διαγράψουμε. Παρατηρείστε ότι αν μας δινόταν η διεύθυνση του προς διαγραφή κόμβου και μόνο, δεν θα είχαμε τρόπο να αποκτήσουμε πρόσβαση στον προηγούμενο της λίστας, κάτι που είναι αναγκαίο.

Figure 2.8: **DelAfter**$(< p >)$



Algorithm 19 DelAfter$(< p >)$

p : ο δείκτης του κόμβου πριν από αυτόν που θέλουμε να διαγράψουμε

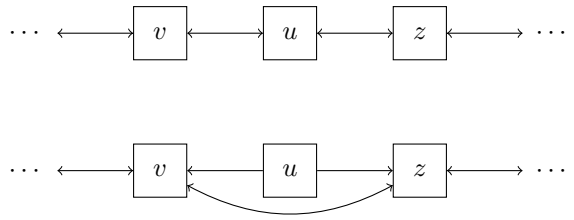
1: **if** node(p).next $\neq$ NIL **then**
2: node(p).next = node(node(p).next).next
3: **end if**

Ο έλεγχος στην γραμμή 1 γίνεται για να δούμε αν υπάρχει κόμβος να διαγράψουμε. Ο αλγόριθμος απλώς αλλάζει τον δείκτη επομένου του προηγούμενου κόμβου από αυτόν που θέλουμε να διαγράψουμε. Παρατηρείστε ότι μετά από αυτήν την αλλαγή, δεν υπάρχει πρόσβαση μέσω της λίστας στον διαγραμμένο κόμβο. Επίσης, ο **DelAfter** θέλει προφανώς $O(1)$ χρόνο.

Σε περίπτωση που αυτό που κάναμε δεν μοιάζει με διαγραφή, να πούμε ότι και στην πράξη, όταν διαγράφουμε ένα αρχείο από το λαπτοπ ή το κινητό μας, στις περισσότερες τον περιπτώσεων αυτό που γίνεται είναι να σπάει η σύνδεση που δείχνει την πρώτη θέση που καταλαμβάνει αυτό το αρχείο στην μνήμη. Εξού και υπάρχουν προγράμματα μετά που μπορούν να υπό συνθήκες να ανακτήσουν αρχεία. Αυτό γίνεται για εξοικονόμηση χρόνου, καθώς είναι αρκετά χρονοβόρο να καθαρίζουν όλες οι θέσεις μνήμης σε ένα π.χ. βίντεο. Ένα αρχείο χάνεται πραγματικά από την μνήμη, όταν τυχαία ή επίτηδες γραφτεί κάτι άλλο στις θέσεις μνήμης που ήταν καταχωρημένο.

Ας δούμε πως θα αντιμετωπίζαμε την περίπτωση που μας δίνεται ο ίδιος ο προς διαγραφή κόμβος, αλλά σε διπλή λίστα.

Figure 2.9: **DelEl**$(< u >)$



Algorithm 20 DelEl$(< u >)$

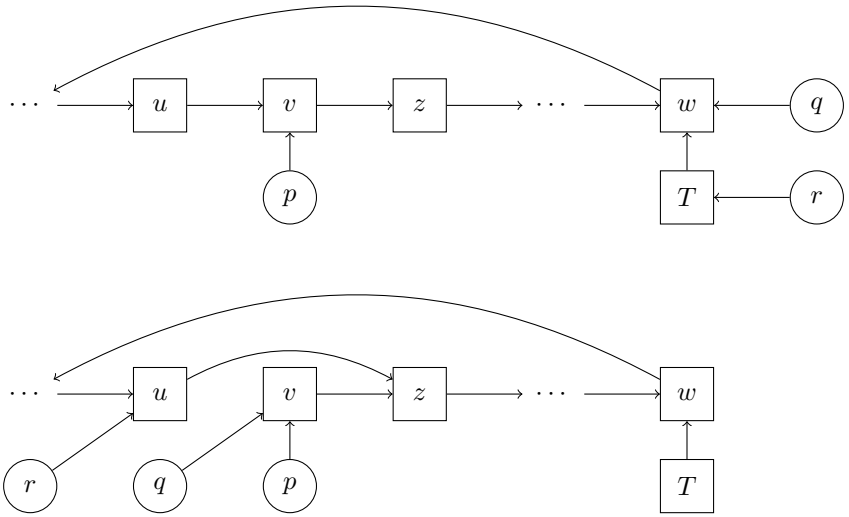
u : ο κόμβος που θέλουμε να διαγράψουμε

1: node$(u$.prev).next = u.next
2: node$(u$.next).prev = u.prev

Σε αυτήν την περίπτωση απλοποιήθηκαν αρκετά τα πράγματα. Και πάλι ο **DelEl** χρειάζεται $O(1)$ χρόνο.

Η τελευταία περίπτωση που θα δούμε, είναι σε μια κυκλική λίστα να μας δίνεται η διεύθυνση του προς διαγραφή κόμβου και ο δείκτης ουράς της λίστας. Θα λάβουμε υπόψιν και την περίπτωση η διεύθυνση του προς διαγραφή κόμβου να είναι λάθος.

Figure 2.10: **DelfromTail**$(< T, p >)$



Algorithm 21 DelfromTail$(< T, p >)$

\# p : ο δείκτης του κόμβου που θέλουμε να διαγράψουμε
\# T : η ουρά της κυκλικής λίστας

1: $q =$ `node`(T)`.next`, $r = T$
2: **while** $q \neq p$ **and** $q \neq T$ **do**
3: $r = q$
4: $q =$ `node`(q)`.next`
5: **end while**
6: **if** $q = p$ **then**
7: `node`(r)`.next` $=$ `node`(q)`.next`
8: **else**
9: **return** -1
10: **end if**

Ο **DelfromTail** χρησιμοποιεί δύο βοηθητικούς δείκτες. Έναν για να βρει το προς διαγραφή στοιχείο κι άλλον έναν για να κρατάει τον προηγούμενο κόμβο. Ο χρόνος εδώ είναι $O(n)$, για λίστα με n κόμβους.

6. Στους αλγορίθμους παρακάτω θα υποθέσουμε ότι θέλουμε να ελέγξουμε αν κάποιο κλειδί key βρίσκεται μέσα στην λίστα και, αν ναι, να επιστρέψουμε την διεύθυνση του κόμβου που το περιέχει.

   Ας δούμε τον αλγόριθμο για απλή λίστα:

Algorithm 22 Search$(< H, key >)$

\# H : κόμβος-δείκτης κεφαλής μιας απλής λίστας
\# key: στοιχείο προς αναζήτηση

1: $p = H$
2: **while** $p \neq$ `NIL` **and** `node`(p)`.val` $\neq key$ **do**
3: $p =$ `node`(p)`.next`
4: **end while**
5: **if** $p =$ `NIL` **then**
6: **return** -1
7: **else**
8: **return** p
9: **end if**

Ο αλγόριθμος **Search** αρχικοποιεί έναν κόμβο-δείκτη στην είσοδο H, τον οποίο και μετακινεί στον δείκτη του επόμενου κάθε φορά κόμβου, μέχρι να βρει κόμβο με τιμή ίση με key, ή να βρει κόμβο που να μην έχει επόμενο. Παρατηρείστε ότι θα κάνει το πολύ τόσα βήματα, όσοι είναι οι κόμβοι της λίστας. Οπότε αν η είσοδος έχει μέγεθος n, ο αλγόριθμος

22 χρειάζεται $O(n)$ βήματα για να ολοκληρώσει την εκτέλεσή του. Το ίδιο θα συνέβαινε αν, αντί για τον έλεγχο της γραμμής 2, θέλαμε μία άλλη οποιαδήποτε διαδικασία σταθερού χρόνου (π.χ. να επιστρέφουμε την τιμή όλων των κόμβων, ή να ελέγχουμε αν είναι θετικοί ακέραιοι κτλ).

Παρατηρείστε ότι στην περίπτωση των λιστών, ακόμη κι αν ξέραμε ότι είναι ταξινομημένες, δεν υπάρχει κάποιος προφανής τρόπος να υλοποιήσουμε κάτι σαν τον αλγόριθμο **BinarySearch** 12, καθώς δεν υπάρχει κάποιος τρόπος να φτάσουμε συγκεκριμένες θέσεις της λίστας, χωρίς να την διασχίσουμε ολόκληρη. Ακόμη, παρατηρείστε ότι οι διπλές λίστες δεν έχουν κάτι ιδιαίτερα διαφορετικό από τις απλές λίστες για την αναζήτηση. Θα μπορούσε, αν αντί για την κεφαλή, μας δινόταν ο δείκτης του τελευταίου κόμβου, να κάναμε την ίδια διαδικασία χρησιμοποιώντας τους δείκτες `prev` αντί για τους `next`.

Για την κυκλική λίστα, χρειαζόμαστε και έναν πιο ειδικό έλεγχο για το αν έχουμε φτάσει εκεί που ξεκινήσαμε.

Algorithm 23 Search$(< T, key >)$

```
# H : κόμβος-δείκτης ουράς μιας κυκλικής λίστας
# key: στοιχείο προς αναζήτηση
 1: if T = NIL then
 2:     return −1
 3: else if node(T).val = key then
 4:     return T
 5: else
 6:     p = node(T).next
 7:     while p ≠ T and node(p).val ≠ key do
 8:         p = node(p).next
 9:     end while
10: end if
11: if node(p).val ≠ key then
12:     return −1
13: else
14:     return p
15: end if
```

Από την στιγμή που τώρα κανένας κόμβος δεν δείχνει στο `NIL`, απλώς ελέγχουμε να μην έχουμε επανέλθει στον κόμβο που έδειχνα η ουρά της λίστας. Η διαδικασία και πάλι χρειάζεται $O(n)$ χρόνο για να ολοκληρωθεί.

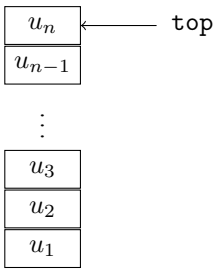
3 Στοίβες, Ουρές

Στις προηγούμενες ενότητες, είδαμε τις βασικές δομές διαχείρισης συνόλων δεδομένων. Πέραν της χρήσης πινάκων και λιστών άμεσα ως δομές, και οι δύο αυτοί τύποι δεδομένων χρησιμοποιούνται και ως βάση για την υλοποίηση πιο σύνθετων δομών, που συνήθως θυσιάζουν κάποιες λειτουργίες για να βελτιστοποιήσουν την απόδοση κάποιων επιμέρους.

Ορισμός 3.1. Η *στοίβα* είναι μία "Last In First Out (LIFO)" δομή, που επιτρέπει πρόσβαση μόνο στο στοιχείο που βρίσκεται στην κορυφή της. Όσα στοιχεία έχουν εισαχθεί στην στοίβα πριν το τελευταίο, παραμένουν αποθηκευμένα αλλά χωρίς δυνατότητα ανάκτησης και επεξεργασίας προτού εξαχθεί η κορυφή.

Το κλασσικό παράδειγμα που χρησιμοποιείται για την διαισθητική ερμηνεία της στοίβας, είναι το πλύσιμο των πιάτων, που εν γένει ξεκινάει από το τελευταίο πιάτο που προστέθηκε στην στοίβα των άπλυτων. Ένα ίσως καλύτερο παράδειγμα είναι η στοίβα εντολών που χρησιμοποιεί ο Η/Υ μας. Κάθε εντολή που καλείται από μία άλλη, μπαίνει από πάνω της στην στοίβα, ώστε να απαιτείται η ολοκλήρωσή της (και η εξαγωγή της από την στοίβα), προτού γίνει δυνατή η ολοκλήρωση της αρχικής εντολής.

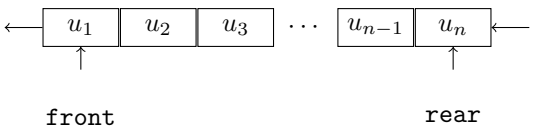
Figure 3.1: Στοίβα, με τα στοιχεία $u_1, \ldots, u_n$ να έχουν καταχωρηθεί με αυτή τη σειρά



Ορισμός 3.2. Η *ουρά* είναι μια "First In First Out (FIFO)" δομή, που επιτρέπει πρόσβαση μόνο στα στοιχεία που βρίσκονται στην αρχή και στο τέλος της. Το πρώτο στοιχείο που μπαίνει στην ουρά από πίσω, είναι το πρώτο που θα βγει από μπροστά. Όλα τα επόμενα, θα περιμένουν να συμβεί αυτό προτού γίνει δυνατή η επεξεργασία ή ανάκτισή τους.

Το κλασσικό παράδειγμα που χρησιμοποιείται για την διαισθητική ερμηνεία της στοίβας, είναι η ουρά σε μια οποιαδήποτε υπηρεσία, όπου έχει προτεραιότητα το άτομο που έφτασε νωρίτερα.

Figure 3.2: Ουρά, με τα στοιχεία $u_1, \ldots, u_n$ να έχουν καταχωρηθεί με αυτή τη σειρά



Θα δούμε πως υλοποιούνται οι δομές ως πίνακες 3.1 και ως λίστες 3.2. Από τις βασικές λειτουργίες που είδαμε στις προηγούμενες ενότητες, η προσπέλαση και η αναζήτηση δεν έχουν κάποια διαφορά: πρέπει να ελέγξουμε ένα προς ένα όλα τα στοιχεία, την στιγμή που τα αφαιρούμε από την δομή. Ως εκ τούτου, θα ασχοληθούμε με τέσσερις βασικές λειτουργίες:

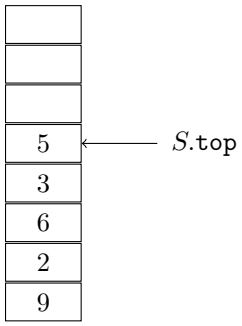
1. αρχικοποίηση,
2. έλεγχος για το αν η δομή είναι κενή,
3. εισαγωγή στοιχείου,
4. εξαγωγή στοιχείου.

3.1 Στοίβες και Ουρές ως πίνακες

Το να υλοποιήσουμε αυτές τις δομές ως στατικές κάνει πιο έντονο το μειονέκτημα των θέσεων μνήμης που μένουν εν γένει κενές καθώς δεν χρειάζονται στο μεγαλύτερο μέρος μιας λειτουργίας.

Μία στοίβα S θα περιγράφεται από το ζεύγος $(S.\texttt{matrix}, S.\texttt{top})$, όπου $S.\texttt{matrix}$ ο πίνακας που περιέχει τα στοιχεία της στοίβας και $S.\texttt{top}$ ο δείκτης κορυφής της στοίβας.

Figure 3.3: $(S.\texttt{matrix} = [9\ 2\ 6\ 3\ 5\ \_\ \_\ \_], S.\texttt{top} = 5)$



Η εντολή:

$$S = \texttt{new stack}(n),$$

αρχικοποιεί πίνακα $S.\texttt{matrix}(n)$ με n θέσεις και δείκτη $S.\texttt{top} = 0$ (υπενθυμίζουμε ότι η 1η θέση ενός πίνακα είναι η 1). Ως εκ τούτου, η στοίβα είναι κενή αν και μόνον αν $S.\texttt{top} = 0$.

Για την εισαγωγή στοιχείου, χρειάζεται απλώς να να βάλουμε, αν υπάρχει περιθώριο, άλλο ένα στοιχείο στο πάνω μέρος της στοίβας και να ενημερώσουμε τον δείκτη κορυφής.

Algorithm 24 Push$(< S, x >)$

```
# S = (S.matrix, S.top) : στοίβα
# x: στοιχείο για προσθήκη στην στοίβα
1: if S.top = len(S.matrix) then
2:     return −1
3: else
4:     S.top += 1
5:     S.matrix[S.top] = x
6: end if
```

Για την εξαγωγή στοιχείου, χρειάζεται απλώς να ενημερώσουμε τον δείκτη κορυφής, ώστε να δείχνει μία θέση πιο χαμηλά.

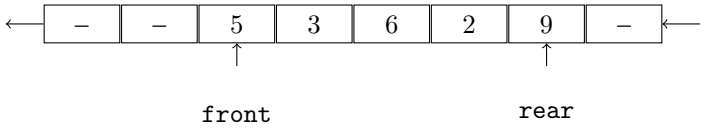
Algorithm 25 Pop$(< S >)$

```
# S = (S.matrix, S.top) : στοίβα
1: if S.top > 0 then
2:     x = S.matrix[S.top]
3:     S.top−− = 1
4:     return x
5: end if
```

Όλες οι παραπάνω λειτουργίες γίνονται σε σταθερό $O(1)$ χρόνο.

Μία ουρά Q θα περιγράφεται από την τριάδα $(Q.\texttt{matrix}, Q.\texttt{front}, Q.\texttt{rear})$, όπου $Q.\texttt{matrix}$ ο πίνακας που περιέχει τα στοιχεία της στοίβας, $Q.\texttt{front}$ ο δείκτης θέσης από την οποία γίνεται η εξαγωγή και $Q.\texttt{rear}$ ο δείκτης θέσης από την οποία γίνεται η εισαγωγή.

Figure 3.4: (Q.matrix $=$ [_ _ 5 3 6 2 9 _], Q.front $= 3$, Q.rear $= 7$)



Η εντολή:

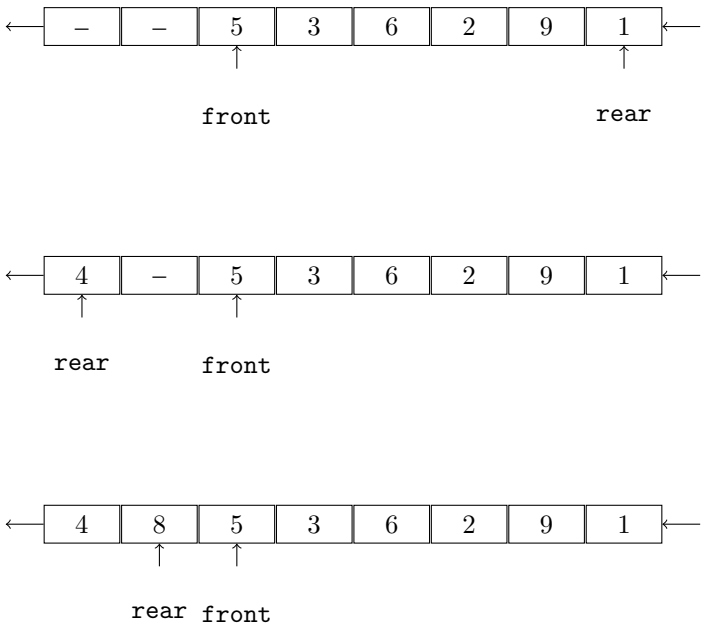
$$S = \texttt{new queue}(n),$$

αρχικοποιεί πίνακα $Q.\texttt{matrix}(n)$ με n θέσεις και δείκτη $Q.\texttt{front} = Q.\texttt{rear} = 0$ (υπενθυμίζουμε ότι η 1η θέση ενός πίνακα είναι η 1). Ως εκ τούτου, μια ουρά είναι κενή αν και μόνον αν $Q.\texttt{front} = 0$ (ισοδύναμα $Q.\texttt{rear} = 0$).

Για την εισαγωγή και την εξαγωγή στοιχείων σε ουρά, δημιουργείται το εξής πρόβλημα: καθώς η δομή είναι στατική, θα μπορούσαν να υπάρχουν κενές θέσεις στην αρχή του πίνακα (για θέσεις δηλαδή μικρότερες του $Q.\texttt{front}$) λόγω εξαγωγών στοιχείων, ενώ στα δεξιά όχι (δηλαδή $Q.\texttt{rear} = \texttt{len}(Q.\texttt{matrix})$). Μία λύση θα ήταν, οποτεδήποτε γίνεται μια εξαγωγή στοιχείου, να μετακινούνται όλα τα στοιχεία της ουράς μία θέση αριστερά, ώστε να εξασφαλίζουμε πάντα το $Q.\texttt{front} = 1$. Κάτι τέτοιο θα έκανε την λειτουργία αυτή να τρέχει σε χρόνο $O(n)$, ουσιαστικά αχρηστεύοντάς την (θυμηθείτε, ο στόχος εδώ είναι να έχουμε σταθερό χρόνο για τέτοιες απλές λειτουργίες). Κάτι αντίστοιχο θα μπορούσε να γίνει και με την λειτουργία της προσθήκης ενός στοιχείου.

Θα λύσουμε αυτό το πρόβλημα με την εξής, σχετικά απλή, σκέψη. Όταν ο πίνακας έχει γεμίσει από δεξιά, θα συνεχίζουμε στις πρώτες θέσεις του, μέχρι να φτάσουμε στην θέση που δείχνει ο $Q.\texttt{front}$. Ο πίνακας $Q.\texttt{matrix}$ λοιπόν θα είναι γεμάτος, αν και μόνον αν $Q.\texttt{rear} = Q.\texttt{front} - 1$, ή, στην ακραία περίπτωση που έχει πραγματοποιηθεί ολόκληρος κύκλος μέσω εξαγωγών και εισαγωγών στοιχείων, $Q.\texttt{rear} = \texttt{len}(Q.\texttt{matrix})$ και $Q.\texttt{front} = 1$. Θεωρείστε το σχήμα 3.4, και υποθέστε ότι προσθέτουμε διαδοχικά τα στοιχεία 1, 4, 8:

Figure 3.5: (Q.matrix $=$ [_ _ 5 3 6 2 9 _], Q.front $= 3$, Q.rear $= 7$), προσθήκη: $1, 4, 8$



Για την εισαγωγή στοιχείου πλέον, χρειάζεται απλώς να ελέγξουμε αν υπάρχει χώρος στην λίστα και να ενημερώσουμε κατάλληλα τους δείκτες:

Algorithm 26 Enqueue$(< Q, n, x >)$

$Q = (Q.\texttt{matrix}, Q.\texttt{front}, Q.\texttt{rear})$: ουρά
$\texttt{len}(Q.\texttt{matrix}) = n$
x: στοιχείο για προσθήκη στην ουρά

```
1:  if Q.front = 0 then
2:     Q.front = Q.rear = 1
3:     Q.matrix[1] = x
4:  else
5:     Q.rear = (Q.rear + 1)  mod n
6:     if Q.front = Q.rear then
7:        return −1
8:     else
9:        Q.matrix[Q.rear] = x
10:    end if
11: end if
```

Για την εξαγωγή στοιχείου, χρειάζεται να ενημερώσουμε τον δείκτη $Q.\texttt{front}$:

Algorithm 27 Dequeue$(< Q, n >)$

$S = (S.\texttt{matrix}, S.\texttt{top})$: ουρά
$\texttt{len}(Q.\texttt{matrix}) = n$

```
1:  if Q.front > 0 then
2:     x = Q.matrix[Q.front]
3:     if Q.front = Q.rear then
4:        Q.front = 0, Q.rear = 0
5:     else
6:        Q.front = (Q.front + 1)  mod n
7:        return x
8:     end if
9:  end if
```

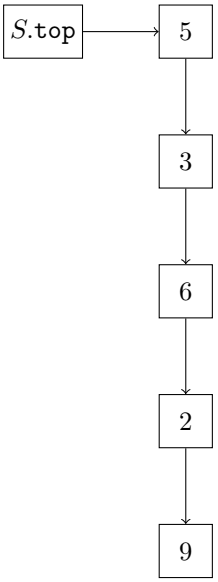
Παρατηρείστε ότι $Q.\texttt{front} = Q.\texttt{rear}$ αν και μόνον αν $Q.\texttt{front} = Q.\texttt{rear} = 1$.

Όλες οι παραπάνω λειτουργίες γίνονται σε σταθερό $O(1)$ χρόνο.

3.2 Στοίβες και Ουρές ως συνδεδεμένες λίστες

Για να αρχικοποιήσουμε μία στοίβα ως συνδεδεμένη λίστα, αρκεί να αρχικοποιήσουμε έναν κόμβο-δείκτη με την εντολή $S = \texttt{new node}()$. Θα χρησιμοποιούμε τον δείκτη $S.\texttt{top}$ αντί για $S.\texttt{next}$. Προφανώς, η στοίβα έχει στοιχεία αν και μόνον αν $S.\texttt{top} \neq \texttt{NIL}$.

Figure 3.6: Υλοποίηση στοίβας με λίστα



Για την εισαγωγή στοιχείου, έχουμε:

Algorithm 28 Push$(< S, x >)$

S : στοίβα
x: στοιχείο για προσθήκη στην στοίβα

1: $p =$ addr(new node(x))
2: node(p).next $=$ node(S.top)
3: S.top $= p$

Για την εξαγωγή στοιχείου, χρειάζεται απλώς να να να ενημερώσουμε τον δείκτη κορυφής, ώστε να δείχνει μία θέση πιο χαμηλά.

Algorithm 29 Pop$(< S >)$

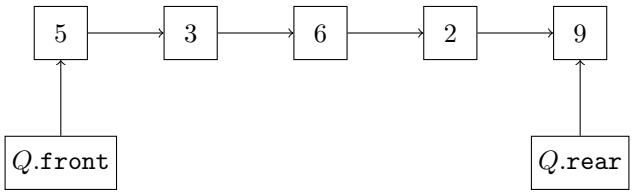
S : στοίβα

1: **if** S.top > 0 **then**
2: $x =$ node(S.top).val
3: S.top $=$ node(S.top).next
4: **return** x
5: **end if**

Όλες οι παραπάνω λειτουργίες γίνονται σε σταθερό $O(1)$ χρόνο.

Για να αρχικοποιήσουμε μία ουρά ως συνδεδεμένη λίστα, αρκεί να αρχικοποιήσουμε δύο κόμβους-δείκτες με τις εντολές Q.front $=$ new node(), Q.rear $=$ new node(). Προφανώς, η ουρά έχει στοιχεία αν και μόνον αν Q.front $\neq$ NIL (ισοδύναμα Q.rear $\neq$ NIL).

Figure 3.7: Υλοποίηση ουράς με λίστα



Για την εισαγωγή στοιχείου, έχουμε:

Algorithm 30 Enqueue$(< Q, x >)$

```
# Q : ουρά
# x: στοιχείο για προσθήκη στην ουρά
1: p = addr(new node(x))
2: if Q.front = Q.rear then
3:    Q.front = Q.rear = p
4: else
5:    node(Q.rear).next = p
6:    Q.rear = p
7: end if
```

Για την εξαγωγή στοιχείου, έχουμε:

Algorithm 31 Dequeue$(< Q >)$

```
# Q : ουρά
1: if Q.front > 0 then
2:    x = node(Q.front).val
3:    if Q.front = Q.rear then
4:       Q.front = NIL, Q.rear = NIL
5:    else
6:       Q.front = node(Q.front).next
7:       return x
8:    end if
9: end if
```

Παρατηρείστε ότι Q.front $= Q$.rear αν και μόνον αν Q.front $= Q$.rear $=$ NIL.

Όλες οι παραπάνω λειτουργίες γίνονται σε σταθερό $O(1)$ χρόνο.

3.3 Ακολουθία Fibonacci

Αναφέραμε και στην αρχή της ενότητας ότι η δομή της στοίβας χρησιμεύει και για την εκτέλεση προγραμμάτων που καλούν άλλα υποπογράμματα, ο τερματισμός των οποίων προϋποτίθεται για να τερματίσει και το αρχικό πρόγραμμα. Ας δούμε ένα τέτοιο εύκολο παράδειγμα.

Έστω $F(n) := 0, 1, 1, 2, 3, 5, 8, \ldots$ η ακολουθία Fibonacci. Είναι γνωστό ότι η $F(n)$ ορίζεται από τον αναδρομικό τύπο:

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1.$$

Ένας απλός αναδρομικός αλγόριθμος για τον υπολογισμό του n-οστού όρου της ακολουθίας θα ήταν ο ακόλουθος:

Algorithm 32 RecFib$(< n >)$

```
# n ∈ ℕ
1: if n = 0 or n = 1 then
2:    return n
3: else
4:    return RecFib(< n − 1 >) + RecFib(< n − 2 >)
5: end if
```

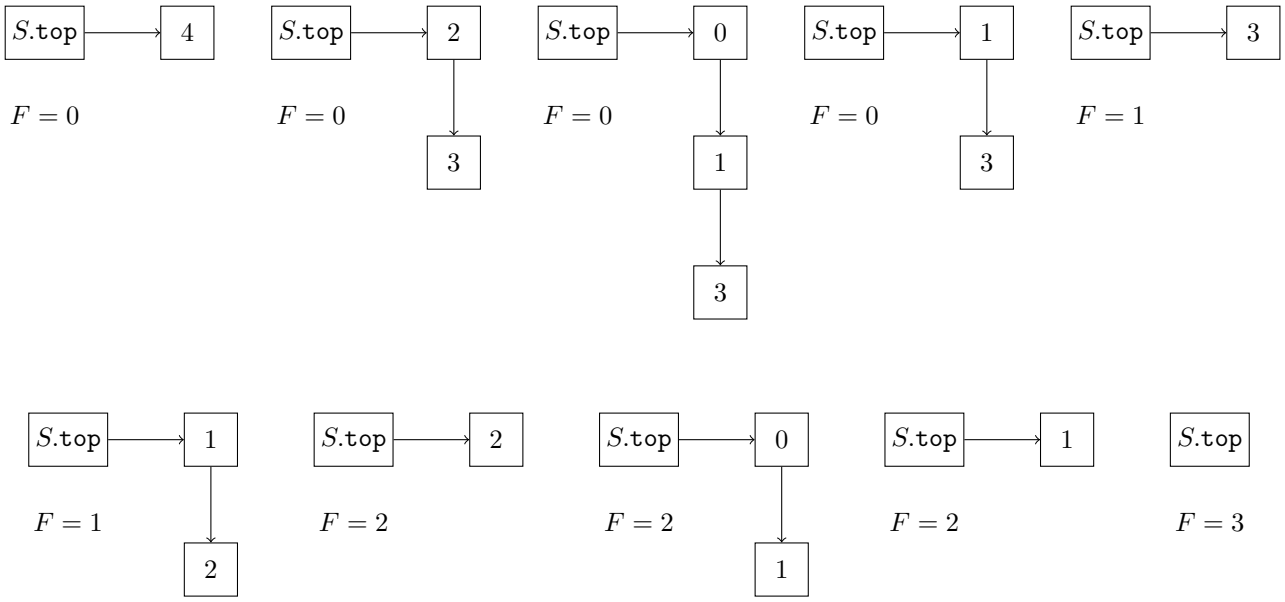
Ας δούμε πως θα υλοποιούταν η παραπάνω αναδρομή με χρήση στοίβας:

Algorithm 33 StackFib$(< n >)$

$\# n \in \mathbb{N}$

```
1:  F = 0, S = new node()
2:  Push(< S, n >)
3:  while S.top ≠ NIL do
4:      cur = Pop(< S >)
5:      if cur = 0 or cur = 1 then
6:          F+ = cur
7:      else
8:          Push(< S, cur − 1 >)
9:          Push(< S, cur − 2 >)
10:     end if
11: end while
12: return F
```

Figure 3.8: **StackFib**(<4>)



Δύο σημεία έχουν σημασία να παρατηρήσουμε εδώ. Πρώτον, στις σύγχρονες γλώσσες προγραμματισμού, σπανίως χρειάζεται να προγραμματίζουμε στο επίπεδο του **StackFib**$(< n, S >)$. Το ποιος μηχανισμός θα χρησιμοποιηθεί για την υλοποίηση ενός αναδρομικού αλγορίθμου, εν γένει αφορά τους εσωτερικούς μηχανισμούς της ίδιας της γλώσσας, στους οποίους είτε δεν έχουμε καν πρόσβαση, είτε ούτως ή άλλως χρησιμοποιούν διάφορες τεχνικές βελτιστοποίησης που καθιστά μη αποδοτικό το να το κάνουμε εμείς με το χέρι. Ως εκ τούτου, εν γένει θα γράφαμε τον **RecFib**$(< n >)$ και θα αφήναμε την γλώσσα προγραμματισμού να χειριστεί τα πράγματα σε επίπεδο μνήμης.

Δεύτερον, ενώ η αναδρομή είναι μια αλγοριθμική τεχνική με πλούσιο θεωρητικό και πρακτικό ενδιαφέρον, η χρήση της για τον υπολογισμό των αριθμών Fibonacci δεν είναι καθόλου αποδοτική, τουλάχιστον έτσι όπως την γράψαμε. Ο λόγος είναι ο υπολογισμός αριθμών $F(k)$, $k < n$, που έχουμε ήδη υπολογίσει, ξανά και ξανά. Για παράδειγμα, Ο $F(1)$ υπολογίζεται 3 φορές.

Ως εκ τούτου, θα ήταν πολύ πιο αποδοτικό να χρησιμοποιούσαμε έναν αλγόριθμο σαν τον παρακάτω.

Algorithm 34 Fib$(< n >)$

$\# \, n \in \mathbb{N}$

1: **if** $n = 0$ `or` $n = 1$ **then**
2: **return** n
3: **else**
4: $pre = 0, cur = 1$
5: **for** $k = 2, \ldots, n$ **do**
6: $new = pre + cur$
7: $pre = cur$
8: $cur = new$
9: **end for**
10: **return** cur
11: **end if**

Υπάρχουν τρόποι να γραφτεί και πιο αποδοτικός αναδρομικός αλγόριθμος. Βγαίνουν όμως εκτός του ενδιαφέροντως των σημειώσεων αυτών.

References

[1] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[2] Jinho D. Choi. Should I learn computer programming in the AI era?, 2023.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[4] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms*. McGraw-Hill Higher Education New York, 2008.

[5] Forbes Technology Council Expert Panel. Should everyone learn to code? 15 tech pros weigh in on why or why not, 2020.

[6] geeksforgeeks. What is assembly language?, 2023.

[7] jaroeducation. What is Von Neumann architecture and its impact on modern computing?, 2023.

[8] Jon Kleinberg and Éva Tardos. *Introduction to algorithms*, volume 482. 2003.

[9] Qiskit. Scott Aaronson says complexity theory is 'inextricable' from quantum computing, 2022.

[10] wiki. List of programming languages by type, 2024.

[11] Δημήτρης Ζώρος. (Πρόχειρες) σημειώσεις στις ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ, 2023.