Ταξινόμηση – Αναζήτηση – Επιλογή

Δημήτρης Φωτάκης

Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων

Σχολή Θετικών Επιστημών

Πανεπιστήμιο Αιγαίου

Νοέμβριος 2006

Πίνακας Περιεχομένων

0. Προκαταρκτικά

Σε αυτή την ενότητα θα μελετήσουμε αλγόριθμους για τα προβλήματα της *ταξινόμησης* (sorting) και της *επιλογής* (selection) και θα αναφερθούμε σε τεχνικές *αναζήτησης* (searching) σε ταξινομημένους πίνακες.

Στο *πρόβλημα της ταξινόμησης* (σε αύξουσα σειρά), δίνεται μία ακολουθία από n αριθμούς $(a_1, a_2, \ldots a_n)$ και ζητείται να υπολογιστεί μία μετάθεση (permutation) $(a'_1, a'_2, \ldots a'_n)$ της ακολουθίας εισόδου ώστε τα στοιχεία να είναι ταξινομημένα σε αύξουσα σειρά $a'_1 \le a'_2 \le \cdots \le a'_n$. Το πρόβλημα της ταξινόμησης σε φθίνουσα σειρά ορίζεται αντίστοιχα. Θα εστιάσουμε στην ταξινόμηση σε αύξουσα σειρά υπογραμμίζοντας ότι οι ίδιοι ακριβώς αλγόριθμοι και τεχνικές εφαρμόζονται και στο πρόβλημα της ταξινόμησης σε φθίνουσα σειρά.

Στο πρόβλημα της *αναζήτησης* (searching), δίνεται μια ακολουθία n αριθμών ταξινομημένη σε αύξουσα σειρά και ένα αριθμός k, και ζητείται να βρεθεί αν υπάρχει και ποια είναι η θέση του αριθμού k στην ταξινομημένη ακολουθία.

Στο πρόβλημα της *επιλογής*, δίνεται μια ακολουθία από n αριθμούς (που δεν είναι κατ' ανάγκην ταξινομημένοι) και ένας αριθμός k, $1 \le k \le n$, και ζητείται να βρεθεί ο k-οστός μικρότερος αριθμός της ακολουθίας (δηλαδή ο αριθμός που βρίσκεται στην k-οστή θέση όταν ταξινομηθεί η ακολουθία σε αύξουσα σειρά). Το πρόβλημα της επιλογής αποτελεί γενίκευση των προβλημάτων εύρεσης του μέγιστου και του ελάχιστου στοιχείου ($k = n$ και $k = 1$ αντίστοιχα). Προφανώς, το πρόβλημα της επιλογής λύνεται εύκολα αν πρώτα ταξινομήσουμε την ακολουθία. Το ζητούμενο είναι αν υπάρχει αλγόριθμος που λύνει το πρόβλημα της επιλογής σε χρόνο σημαντικά μικρότερο από το χρόνο που χρειάζεται για να ταξινομήσουμε την ακολουθία (όπως συμβαίνει με τα προβλήματα της εύρεσης του μέγιστου και του ελάχιστου).

Μια σημαντική διαφορά με άλλες δομές δεδομένων που εξετάζονται στο μάθημα (π.χ. λίστες, ουρές, δέντρα αναζήτησης) είναι ότι η συλλογή των στοιχείων είναι *στατική* και γνωστή εκ των προτέρων στον αλγόριθμο. Δεν υποστηρίζονται λοιπόν οι λειτουργίες της εισαγωγής και διαγραφής.

Όλα τα στοιχεία είναι αποθηκευμένα σε έναν (στατικό) πίνακα, τον οποίο θα συμβολίζουμε με A. Θεωρούμε ότι ο A έχει n στοιχεία A[1], A[2], …, A[n] (εκτός αν ρητά δηλώνεται κάτι διαφορετικό). Κάθε στοιχείο αντιστοιχεί στο *κλειδί* (key) μιας *εγγραφής* (record) βάση του οποίου θα γίνει η ταξινόμηση, η αναζήτηση, ή η επιλογή. Τα υπόλοιπα πεδία της εγγραφής μπορούν να περιέχουν κάθε είδους *συσχετιζόμενη πληροφορία* (satellite data). Πάντα θα θεωρούμε ότι τα κλειδιά / στοιχεία του πίνακα A είναι (φυσικοί) αριθμοί αν και θα μπορούσαν να είναι στοιχεία ενός οποιουδήποτε ολικά διατεταγμένου συνόλου.

1. Αλγόριθμοι Ταξινόμησης

Πολλοί συγγραφείς θεωρούν ότι το πρόβλημα της ταξινόμησης αποτελεί ένα από τα πλέον θεμελιώδη αλγοριθμικά προβλήματα. Οι λόγοι είναι πολλοί και διαφορετικής φύσης. Ενδεικτικά αναφέρουμε:

- Η ταξινόμηση των στοιχείων είναι απαραίτητη σε πολλές πρακτικές εφαρμογές. Με βάση κάποιες μετρήσεις (που έγιναν πριν 20 ή και περισσότερα χρόνια), υποστηρίζεται ότι περίπου το 25% του υπολογιστικού χρόνου αφιερώνεται σε λειτουργίες που έχουν σχέση με το πρόβλημα της ταξινόμησης.

- Διάφορες λειτουργίες, με σημαντικότερη αυτή της αναζήτησης, γίνονται πολύ γρήγορα όταν η ακολουθία των στοιχείων είναι ταξινομημένη. Για παράδειγμα, η αναζήτηση σε ταξινομημένη ακολουθία ολοκληρώνεται σε χρόνο $O(\log n)$, έναντι $\Theta(n)$ που απαιτεί η γραμμική αναζήτηση στη μέση περίπτωση όταν η ακολουθία δεν είναι ταξινομημένη.

- Η ταξινόμηση προκύπτει σαν βασικό ενδιάμεσο βήμα κατά το σχεδιασμό άλλων αλγορίθμων. Για παράδειγμα, ο αλγόριθμος του Kruskal για τον υπολογισμό ενός Ελάχιστου Γενετικού Δέντρου (θα τον μάθουμε στο άλλο εξάμηνο) απαιτεί την ταξινόμηση των ακμών του γραφήματος σε αύξουσα σειρά βαρών.

- Υπάρχουν πολλοί διαφορετικοί αλγόριθμοι ταξινόμησης που βασίζονται σε πολλές διαφορετικές ιδέες και τεχνικές. Οι ίδιες ιδέες και τεχνικές βρίσκουν εφαρμογή σε πολλά σημαντικά αλγοριθμικά προβλήματα.

Στα πλαίσια του μαθήματος, θα αναφερθούμε μόνο σε *συγκριτικούς* αλγόριθμους ταξινόμησης (comparison-based sorting algorithms). Οι αλγόριθμοι αυτοί επιτυγχάνουν την ταξινόμηση των στοιχείων βασιζόμενοι *αποκλειστικά* στα αποτελέσματα των συγκρίσεων μεταξύ των στοιχείων. Η ανάλυση των συγκριτικών αλγορίθμων γίνεται μετρώντας τον αριθμό των συγκρίσεων και των μετατοπίσεων στοιχείων που εκτελούν. Στο τέλος της ενότητας, θα αποδείξουμε ότι κάθε συγκριτικός αλγόριθμος χρειάζεται τουλάχιστον $\Omega(n \log n)$ συγκρίσεις (στη χειρότερη περίπτωση) για την ταξινόμηση n στοιχείων.

Δεν θα αναφερθούμε σε μια σημαντική κατηγορία αλγορίθμων που χρησιμοποιούν επιπλέον τη δομή της εσωτερικής αναπαράστασης των στοιχείων (π.χ. το γεγονός ότι τα στοιχεία είναι φυσικοί αριθμοί σε ένα συγκεκριμένο διάστημα) και επιτυγχάνουν καλύτερο χρόνο εκτέλεσης χειρότερης περίπτωσης. Υπάρχουν αλγόριθμοι αυτής της κατηγορίας (π.χ. counting sort και radix sort) που ταξινομούν n φυσικούς αριθμούς στο διάστημα $\{1, \ldots, k\}$ σε χρόνο χειρότερης περίπτωσης $O(k + n)$.

Υπάρχουν πολλές διαφορετικές ιδέες που οδηγούν σε αποτελεσματικούς συγκριτικούς αλγόριθμους ταξινόμησης:

- ***Εισαγωγή*** του επόμενου στοιχείου στην κατάλληλη θέση μεταξύ των ήδη ταξινομημένων στοιχείων ενός υποπίνακα. Ο αλγόριθμος ταξινόμησης με *εισαγωγή* (insertion sort) βασίζεται σε αυτή την ιδέα.

- ***Αντιμετάθεση*** κάθε ζεύγους γειτονικών στοιχείων που βρίσκεται εκτός διάταξης (δηλαδή το μικρότερο στοιχείο του ζεύγους εμφανίζεται αμέσως μετά το μεγαλύτερο). Αυτή η ιδέα οδηγεί στον αλγόριθμο ταξινόμησης *φυσαλίδας* (bubble sort).

- ***Επιλογή*** του μικρότερου (μεγαλύτερου) στοιχείου από το σύνολο των μη-ταξινομημένων στοιχείων και τοποθέτησή του σαν τελευταίο (πρώτο αντίστοιχα) στοιχείο στον υποπίνακα των ταξινομημένων στοιχείων. Οι αλγόριθμοι ταξινόμησης με *επιλογή* (selection sort) και ταξινόμησης *σωρού* (heap sort) βασίζονται στην ιδέα της επιλογής.

- ***Συγχώνευση*** δύο ταξινομημένων υποπινάκων. Οι ταξινομημένοι υποπίνακες προέρχονται από διαίρεση του αρχικού πίνακα στη μέση και ταξινόμηση είτε με αναδρομική κλήση της ίδιας διαδικασίας είτε με άλλον αλγόριθμο. Η ιδέα αυτή οδηγεί στην ταξινόμηση με *συγχώνευση* (merge sort).

- ***Διαίρεση*** των στοιχείων σε δύο υποπίνακες ανάλογα με το αν είναι μικρότερα ή μεγαλύτερα από ένα κατάλληλα επιλεγμένο στοιχείο και ταξινόμηση (είτε με αναδρομική κλήση της ίδιας μεθόδου είτε με άλλο αλγόριθμο) των δύο υποπινάκων. Αυτή η ιδέα οδηγεί στον αλγόριθμο *ταχείας* ταξινόμησης (quicksort).

1.1. Ταξινόμηση με Εισαγωγή

Ο αλγόριθμος ταξινόμησης με *εισαγωγή* (insertion sort) λειτουργεί σε φάσεις. Διατηρεί στο αριστερό μέρος του πίνακα Α ένα ταξινομημένο υποσύνολο των στοιχείων. Σε κάθε φάση, το επόμενο στοιχείο τοποθετείται στη σωστή του θέση ανάμεσα στα ταξινομημένα στοιχεία αυξάνοντας το μέγεθος του ταξινομημένου υποσυνόλου κατά 1. Ο ψευδοκώδικας που ακολουθεί υλοποιεί την ταξινόμηση με εισαγωγή.

```
Insertion-Sort(A[1...n])
    for j ← 2 to n do
        key ← A[j]; i ← j – 1;
        while i > 0 and A[i] > key do
            A[i+1] ← A[i];
            i ← i – 1;
        A[i+1] ← key;
```

Ορθότητα. Η ορθότητα του αλγόριθμου βασίζεται στην εξής αμετάβλητη συνθήκη (invariant): Πριν αρχίσει η εκτέλεση του for-loop με τιμή j, $j = 2, \ldots, n$, τα στοιχεία του υποπίνακα Α[1…j – 1] είναι ταξινομημένα σε αύξουσα σειρά.

Το γεγονός ότι ο αλγόριθμος ικανοποιεί πάντα αυτή την αμετάβλητη συνθήκη μπορεί να αποδειχθεί εύκολα με μαθηματική επαγωγή. Προφανώς, η αμετάβλητη συνθήκη ικανοποιείται για $j = 2$, αφού το ταξινομημένο μέρος του υποπίνακα αποτελείται από ένα μόνο στοιχείο. Έστω ότι η συνθήκη ισχύει για κάποια τιμή του j. Θα δείξουμε ότι η συνθήκη ισχύει και στην αρχή του επόμενου for-loop, δηλαδή για j+1.

Το while-loop που εκτελείται βάζει το στοιχείο Α[j] στη σωστή του θέση ανάμεσα στα στοιχεία του υποπίνακα Α[1…j]. Πράγματι, το while-loop αποθηκεύει το Α[j] στη μεταβλητή *key* και ολισθαίνει όλα τα στοιχεία του Α[1…j–1] που είναι μεγαλύτερα από το *key* μία θέση δεξιά. Το περιεχόμενο της *key* τοποθετείται αμέσως δεξιότερα από το πρώτο στοιχείο του Α[1…j–1] που δεν ξεπερνά το *key*. Επομένως, πριν την επόμενη εκτέλεση του for-loop, ο υποπίνακας Α[1…j] εξακολουθεί να είναι ταξινομημένος.

Άσκηση 1.1. Περιγράψτε τη λειτουργία του αλγορίθμου για τον πίνακα Α = [5, 2, 4, 6, 1, 3]. Επαναλάβετε για τους πίνακες Β = [1, 2, 3, 4, 5, 6], και Γ = [6, 5, 4, 3, 2, 1];

Χρονική Πολυπλοκότητα. Το for-loop εκτελείται ακριβώς (n – 1) φορές ($j = 2, \ldots, n$). Ο αριθμός των εκτελέσεων του while-loop σε κάθε εκτέλεση του for-loop εξαρτάται από τη θέση που πρέπει να πάρει το Α[j] στον ταξινομημένο υποπίνακα Α[1…j].

Στην καλύτερη περίπτωση το Α[j] θα παραμείνει στη θέση του. Αυτό συμβαίνει σε κάθε εκτέλεση του for-loop όταν ο πίνακας Α είναι από την αρχή ταξινομημένος σε αύξουσα σειρά. Τότε το while-loop δεν θα εκτελεστεί καμία φορά και το for-loop θα εκτελέσει μία και μόνο σύγκριση (μεταξύ στοιχείων). Επομένως, στην καλύτερη περίπτωση, ο αριθμός των συγκρίσεων (μεταξύ στοιχείων) που πραγματοποιεί ο αλγόριθμος είναι n – 1 και ο χρόνος εκτέλεσης του αλγόριθμου είναι $\Theta(n)$.

Στην χειρότερη περίπτωση το Α[j] είναι μικρότερο από όλα τα στοιχεία του υποπίνακα Α[1…j–1] και θα τοποθετηθεί στην πρώτη θέση. Αυτό συμβαίνει όταν ο πίνακας Α δίνεται ταξινομημένος σε φθίνουσα σειρά. Τότε το while-loop εκτελείται συνολικά j–1 φορές και το for-loop εκτελεί συνολικά $\Theta(j)$ συγκρίσεις και μετατοπίσεις στοιχείων. Επομένως, στη χειρότερη περίπτωση, ο αριθμός των συγκρίσεων είναι περίπου $n^2 / 2$ και ο χρόνος εκτέλεσης $\Theta(n^2)$.

Ένα χαρακτηριστικό της ταξινόμησης με εισαγωγή είναι ότι λειτουργεί ακόμη και αν τα στοιχεία δεν είναι εκ των προτέρων γνωστά αλλά δίνονται στον αλγόριθμο το ένα-μετά-το-άλλο (οι αλγόριθμοι που δεν προϋποθέτουν πλήρη γνώση της εισόδου είναι γνωστοί και σαν *online* αλγόριθμοι). Ο αλγόριθμος τοποθετεί κάθε νέο στοιχείο στη σωστή του θέση διατηρώντας ταξινομημένο το σύνολο των στοιχείων που έχει εξετάσει.

Ένα δεύτερο χαρακτηριστικό της ταξινόμησης με εισαγωγή είναι ότι χρειάζεται σταθερό αριθμό θέσεων μνήμης επιπλέον του χώρου που καταλαμβάνει ο πίνακας Α. Λέμε ότι τέτοιου είδους αλγόριθμοι λειτουργούν *εντός του δοθέντος χώρου* (*in place*).

Άσκηση 1.2. Ο αλγόριθμος ταξινόμησης με εισαγωγή ουσιαστικά χρησιμοποιεί γραμμική αναζήτηση για να εντοπίσει τη θέση που πρέπει να καταλάβει το στοιχείο Α[j] ανάμεσα στα στοιχεία του υποπίνακα Α[1…j]. Η γραμμική αναζήτηση έχει γραμμικό χρόνο εκτέλεσης στη χειρότερη περίπτωση. Συνεπώς, ο χρόνος εκτέλεσης χειρότερης περίπτωσης για την ταξινόμηση με εισαγωγή δεν μπορεί να είναι μικρότερος από $\sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$. Σαν εναλλακτική της γραμμικής αναζήτησης, μπορεί να χρησιμοποιηθεί η δυαδική αναζήτηση με χρόνο εκτέλεσης $O(\log n)$. Αυτό είναι εφικτό γιατί τα στοιχεία στον υποπίνακα Α[1…j–1] είναι ταξινομημένα. Ποιος θα ήταν ο συνολικός αριθμός των συγκρίσεων για τον τροποποιημένο αλγόριθμο; Θα βελτίωνε αυτή η τροποποίηση το χρόνο εκτέλεσης χειρότερης περίπτωσης σε $O(n \log n)$ ή όχι; Να αιτιολογήσετε την απάντησή σας.

1.2. Ταξινόμηση Φυσαλίδας

Η ταξινόμηση *φυσαλίδας* (bubble sort) είναι ένας από τους πλέον δημοφιλείς αλγόριθμους ταξινόμησης. Ο αλγόριθμος λειτουργεί σε φάσεις διατηρώντας στο αριστερό μέρος του πίνακα Α τα μικρότερα στοιχεία ταξινομημένα. Σε κάθε φάση, ο αλγόριθμος σαρώνει τον μη-ταξινομημένο υποπίνακα από δεξιά προς τα αριστερά αντιμεταθέτοντας κάθε ζευγάρι γειτονικών στοιχείων που βρίσκεται εκτός διάταξης. Σαν αποτέλεσμα, στο τέλος κάθε περάσματος το μικρότερο στοιχείο του μη-ταξινομημένου υποπίνακα καταλαμβάνει την αριστερότερη θέση και αυξάνοντας τον αριθμό των ταξινομημένων στοιχείων κατά 1 (το μικρότερο στοιχείο «ανεβαίνει» στη φυσαλίδα που το οδηγεί στην «κατώτερη» θέση του μη-ταξινομημένου υποπίνακα).

Ο παρακάτω ψευδοκώδικας υλοποιεί τη βασική εκδοχή της ταξινόμησης φυσαλίδας. Η κλήση `swap(A[i], A[j])` αντιμεταθέτει τα στοιχεία των αντίστοιχων θέσεων του πίνακα τοποθετώντας το περιεχόμενο της θέσης Α[i] στη θέση Α[j] και το περιεχόμενο της θέσης Α[j] στη θέση Α[i].

```
Bubble-Sort(A[1...n])
    for i ← 1 to n-1 do
        for j ← n downto i+1 do
            if A[j] < A[j-1] then
                swap(A[j], A[j-1]);
```

Ορθότητα. Η ορθότητα του αλγόριθμου βασίζεται στην εξής αμετάβλητη συνθήκη: στο τέλος της i-οστής εκτέλεσης του εξωτερικού for-loop, το μικρότερο στοιχείο του υποπίνακα Α[i…n] βρίσκεται στη θέση Α[i]. Αυτό αποδεικνύεται παρατηρώντας ότι το εσωτερικό for-loop διατηρεί πάντα (δηλαδή για κάθε τιμή του j) το μικρότερο στοιχείο του υποπίνακα Α[j–1…n] στη θέση Α[j–1]. Η τελευταία παρατήρηση αποδεικνύεται εύκολα με μαθηματική επαγωγή.

Η αμετάβλητη συνθήκη δείχνει ότι στο τέλος της πρώτης επανάληψης του εξωτερικού for-loop το μικρότερο στοιχείο του πίνακα βρίσκεται στην πρώτη θέση, στο τέλος της δεύτερης επανάληψης, το δεύτερο μικρότερο στοιχείο βρίσκεται στη δεύτερη θέση, κοκ. Επομένως, ο αλγόριθμος τερματίζει με όλα τα στοιχεία του πίνακα ταξινομημένα.

Χρονική Πολυπλοκότητα. Η βασική εκδοχή της ταξινόμησης φυσαλίδας που υλοποιείται εδώ έχει χρόνο εκτέλεσης $\Theta(n^2)$ σε κάθε περίπτωση (καλύτερη και

χειρότερη). Πράγματι, το εξωτερικό for-loop εκτελείται ακριβώς $n - 1$ φορές. Κατά την i-οστή επανάληψη του εξωτερικού for-loop, το εσωτερικό for-loop εκτελείται $n - i - 1$ φορές. Σε κάθε εκτέλεση του εσωτερικού for-loop έχουμε 1 σύγκριση μεταξύ στοιχείων, οπότε ο συνολικός αριθμός των συγκρίσεων είναι περίπου $n^2 / 2$.

Άσκηση 1.3. Περιγράψτε τη λειτουργία της ταξινόμησης φυσαλίδας για τον πίνακα Α = [5, 2, 4, 6, 1, 3]. Επαναλάβετε για τους πίνακες Β = [1, 2, 3, 4, 5, 6], και Γ = [6, 5, 4, 3, 2, 1]; Να συγκρίνετε τον αριθμό των συγκρίσεων και των μετατοπίσεων στοιχείων με τους αντίστοιχους αριθμούς για την ταξινόμηση με εισαγωγή.

Ο χρόνος εκτέλεσης *καλύτερης περίπτωσης* της ταξινόμησης φυσαλίδας μπορεί να βελτιωθεί σε $\Theta(n)$ όταν ο πίνακας Α είναι αρχικά ταξινομημένος σε αύξουσα σειρά. Όμως καμία βελτίωση δεν μπορεί να γίνει στη χειρότερη περίπτωση που ο χρόνος εκτέλεσης παραμένει τετραγωνικός. Όσον αφορά στις θέσεις μνήμης, η ταξινόμηση φυσαλίδας λειτουργεί εντός του δοθέντος χώρου.

Άσκηση 1.4. Να προτείνετε δύο βελτιώσεις στον αλγόριθμο της ταξινόμησης φυσαλίδας ώστε ο χρόνος εκτέλεσης καλύτερης περίπτωσης να είναι γραμμικός. (*Υπόδειξη*: Προσπαθήστε να μην συγκρίνετε το ίδιο ζεύγος στοιχείων περισσότερες από μία φορές).

1.3. Ταξινόμηση με Επιλογή

Η ιδέα της ταξινόμησης με *επιλογή* (selection sort) είναι η πλέον φυσιολογική για αλγόριθμο ταξινόμησης. Ο αλγόριθμος λειτουργεί σε φάσεις. Στην πρώτη φάση, το μικρότερο στοιχείο τοποθετείται στην πρώτη θέση του πίνακα, στη δεύτερη φάση, το δεύτερο μικρότερο στοιχείο τοποθετείται στη δεύτερη θέση, κοκ. Η ιδέα μοιάζει πολύ με αυτή της ταξινόμησης φυσαλίδας, αλλά η υλοποίηση διαφέρει.

Για να υλοποιήσουμε την ταξινόμηση με επιλογή, διατηρούμε στο αριστερό μέρος του πίνακα τα μικρότερα στοιχεία ταξινομημένα. Στην επόμενη φάση, εκτελούμε γραμμική αναζήτηση για να επιλέξουμε το μικρότερο στοιχείο από το μη-ταξινομημένο μέρος του πίνακα και το προσθέτουμε στο ταξινομημένο μέρος. Ο παρακάτω ψευδοκώδικας υλοποιεί την ταξινόμηση με επιλογή.

```
Selection-Sort(A[1...n])
    for i ← 1 to n-1 do
        min_pos ← i; min_key ← A[i];
        for j ← i+1 to n do
            if A[j] < min_key then
                min_pos ← j; min_key ← A[j];
        A[min_pos] ← A[i]; A[i] ← min_key;
```

Ορθότητα. Η ορθότητα του αλγόριθμου βασίζεται στην ίδια αμετάβλητη συνθήκη με την ταξινόμηση φυσαλίδας: στο τέλος της *i*-οστής εκτέλεσης του εξωτερικού for-loop, το μικρότερο στοιχείο του υποπίνακα A[*i*…*n*] βρίσκεται στη θέση A[*i*]. Αυτό αποδεικνύεται παρατηρώντας ότι για κάθε τιμή του *i*, το εσωτερικό for-loop βρίσκει το μικρότερο στοιχείο του πίνακα A[*i*…*n*] (το αποθηκεύει στη μεταβλητή *min_key* και τη θέση του στη *min_pos*). Στην τελευταία εντολή του εξωτερικού for-loop το μικρότερο στοιχείο του πίνακα A[*i*…*n*] και το στοιχείο A[*i*] αντιμετατίθενται.

Η αμετάβλητη συνθήκη δείχνει ότι στο τέλος της πρώτης επανάληψης του εξωτερικού for-loop το μικρότερο στοιχείο του πίνακα βρίσκεται στην πρώτη θέση, στο τέλος της δεύτερης επανάληψης, το δεύτερο μικρότερο στοιχείο βρίσκεται στη δεύτερη θέση, κοκ. Επομένως, ο αλγόριθμος τερματίζει με όλα τα στοιχεία του πίνακα ταξινομημένα.

Χρονική Πολυπλοκότητα. Η ταξινόμηση με επιλογή έχει χρόνο εκτέλεσης $\Theta(n^2)$. Το εξωτερικό for-loop εκτελείται ακριβώς $n - 1$ φορές. Κατά την *i*-οστή επανάληψη του εξωτερικού for-loop, το εσωτερικό for-loop εκτελείται $n - i - 1$ φορές. Σε κάθε εκτέλεση του εσωτερικού for-loop έχουμε 1 σύγκριση μεταξύ στοιχείων, οπότε ο συνολικός αριθμός των συγκρίσεων είναι περίπου $n^2 / 2$. Όσον αφορά στις θέσεις μνήμης, η ταξινόμηση φυσαλίδας λειτουργεί εντός του δοθέντος χώρου.

Οι αλγόριθμοι ταξινόμησης με εισαγωγή, ταξινόμησης φυσαλίδας, και ταξινόμησης με επιλογή είναι κατάλληλοι για την ταξινόμηση μικρού σχετικά αριθμού στοιχείων (π.χ. ≤ 500 στοιχεία).

Άσκηση 1.5. Περιγράψτε τη λειτουργία της ταξινόμησης με επιλογή για τον πίνακα A = [5, 2, 4, 6, 1, 3]. Επαναλάβετε για τους πίνακες B = [1, 2, 3, 4, 5, 6], και Γ = [6, 5, 4, 3, 2, 1]; Να συγκρίνετε τον αριθμό των συγκρίσεων και των μετατοπίσεων στοιχείων με τους αντίστοιχους αριθμούς για την ταξινόμηση φυσαλίδας και την ταξινόμηση με εισαγωγή.

1.4. Ταξινόμηση Σωρού

Η ιδέα της ταξινόμησης *σωρού* (heapsort) είναι ουσιαστικά η ίδια με την ιδέα της ταξινόμησης με επιλογή. Στο δεξιό μέρος του πίνακα διατηρούμε τα μεγαλύτερα στοιχεία ταξινομημένα. Σε κάθε φάση επιλέγεται το μεγαλύτερο στοιχείο από το μη-ταξινομημένο μέρος του πίνακα και προστίθεται σαν πρώτο στοιχείο στο ταξινομημένο μέρος. Η ουσιαστική διαφορά είναι ότι αντί της γραμμικής αναζήτησης για την επιλογή του μεγαλύτερου στοιχείου χρησιμοποιείται η δομή του *σωρού* (heap), μια δομή δεδομένων για διαχείριση ουρών προτεραιότητας. Η δομή του σωρού επιτρέπει την εξαγωγή του μέγιστου στοιχείου από τον μη-ταξινομημένο υποπίνακα σε χρόνο $O(\log n)$. Έτσι ο χρόνος χειρότερης περίπτωσης της ταξινόμησης σωρού είναι $O(n \log n)$ (n εξαγωγές του μεγαλύτερου στοιχείου από τον μη-ταξινομημένο υποπίνακα).

1.4.1. Ο Σωρός σαν Ουρά Προτεραιότητας

Ο (*δυαδικός*) *σωρός* (binary heap) είναι ένα *σχεδόν πλήρες* (ή συμπληρωμένο, complete) δυαδικό δέντρο με στοιχεία στους κόμβους του. Η ιδιότητα που χαρακτηρίζει το σωρό είναι ότι το στοιχείο κάθε κόμβου είναι μεγαλύτερο ή ίσο από τα στοιχεία των δύο παιδιών του. Η ιδιότητα του σωρού συνεπάγεται ότι το μεγαλύτερο στοιχείο κάθε υποδέντρου βρίσκεται πάντα στη ρίζα. Επομένως, το μεγαλύτερο στοιχείο του σωρού βρίσκεται πάντα στη ρίζα[1], το δεύτερο μεγαλύτερο στοιχείο σε ένα από τα παιδιά της ρίζας, και το μικρότερο στοιχείο σε ένα από τα φύλλα.

Με τον όρο *σχεδόν πλήρες* εννοούμε ότι το αντίστοιχο δέντρο είναι πλήρες σε όλα τα επίπεδα εκτός ίσως από το τελευταίο επίπεδο που γεμίζει από τα αριστερά προς τα δεξιά. Δηλαδή, όλα τα φύλλα βρίσκονται στο τελευταίο και το προ-τελευταίο επίπεδο και όλα τα φύλλα του τελευταίου επιπέδου βρίσκονται προς τα αριστερά. Επιπλέον, υπάρχει

[1] Ο σωρός με τη συγκεκριμένη ιδιότητα ονομάζεται και σωρός-μέγιστου (max-heap). Στη διαχείριση ουρών προτεραιότητας χρησιμοποιείται συχνά και ο σωρός-ελάχιστου (min-heap) όπου το στοιχείο κάθε κόμβου είναι μικρότερο ή ίσο των στοιχείων των παιδιών του. Στο σωρό-ελάχιστου, η ρίζα περιέχει το ελάχιστο στοιχείο του σωρού. Στη συνέχεια εστιάζουμε στο σωρό-μέγιστου γιατί αυτός χρησιμοποιείται για την ταξινόμηση στοιχείων σε αύξουσα σειρά με τον αλγόριθμου σωρού. Οι ίδιες λειτουργίες υλοποιούνται με αντίστοιχο τρόπο και στον σωρό-ελάχιστου.

ένας το πολύ κόμβος που δεν είναι φύλλο και έχει ένα μόνο παιδί και αυτός βρίσκεται (αν υπάρχει) στο προ-τελευταίο επίπεδο (Σχήμα 1.1).

Συνεπώς, ένας σωρός ύψους h περιέχει τουλάχιστον h πλήρη επίπεδα και τουλάχιστον ένα φύλλο στο τελευταίο επίπεδο. Ο αριθμός των στοιχείων του είναι

$$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h \Rightarrow h \leq \log n$$

Από την άλλη πλευρά, ένας σωρός ύψους h περιέχει το πολύ h+1 πλήρη επίπεδα. Σε αυτή την περίπτωση, ο αριθμός των στοιχείων του είναι

$$n \leq 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1 \Rightarrow h \geq \log(n+1) - 1.$$

Συνεπώς, το ύψος ενός σωρού με n στοιχεία κυμαίνεται πάντα μεταξύ[2] $\log(n+1) - 1 \leq h \leq \log n$, είναι δηλαδή ίσο με $\lfloor \log n \rfloor$.

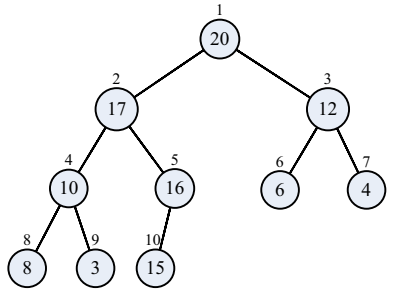
Ένας σωρός με n στοιχεία μπορεί εύκολα να αποθηκευθεί στις αριστερότερες n θέσεις ενός πίνακα Α (που έχει n ή περισσότερες θέσεις). Χρειάζεται πάντα να γνωρίζουμε τον αριθμό των στοιχείων n στον πίνακα Α τα οποία συγκροτούν το σωρό. Τα στοιχεία του Α σε θέσεις δεξιότερα του n (εφόσον υπάρχουν) θεωρούνται στοιχεία του πίνακα αλλά όχι στοιχεία του σωρού.

Η ρίζα είναι αποθηκευμένη στο πρώτο στοιχείο του πίνακα Α[1]. Για το στοιχείο του σωρού στη θέση Α[i], $2 \leq i \leq n$, το στοιχείο στη θέση $\lfloor i/2 \rfloor$ είναι ο πατέρας του και τα στοιχεία στις θέσεις $2i$ και $2i$+1 (εφόσον φυσικά $2i \leq n$ και $2i+1 \leq n$) αποτελούν το αριστερό και το δεξί του παιδί αντίστοιχα[3] (Σχήμα 1.1). Στο εξής, θα θεωρούμε ότι οι συναρτήσεις *Left*(i), *Right*(i), και *Parent*(i) επιστρέφουν $2i$, $2i$+1, και $\lfloor i/2 \rfloor$, δηλαδή τις θέσεις του αριστερού παιδιού, του δεξιού παιδιού, και του πατέρα του Α[i] στο σωρό. Επίσης, θα θεωρούμε ότι *hs*(Α) είναι ο αριθμός των στοιχείων του σωρού στον πίνακα Α.

[2] Σε αυτές τις σημειώσεις χρησιμοποιούμε το log n για να συμβολίσουμε το λογάριθμο με βάση 2 και ln n για το φυσικό (ή νεπέριο) λογάριθμο.

[3] Οι λειτουργίες υπολογισμού των θέσεων του πατέρα και των παιδιών του Α[i] υλοποιούνται εύκολα και γρήγορα χρησιμοποιώντας τις πράξεις της δεξιάς και αριστερής ολίσθησης της δυαδικής αναπαράστασης του i.

Τέλος, θα θεωρούμε ότι ο πίνακας Α έχει αρκετές διαθέσιμες θέσεις για να χωρέσει τα νέα στοιχεία που προστίθενται στο σωρό.



Σχήμα 1.1. Σωρός-μέγιστου με 10 στοιχεία. Δίνεται η αναπαράσταση με δυαδικό δέντρο και η αποθήκευση σε πίνακα. Πάνω από κάθε κόμβο του δυαδικού δέντρου φαίνεται η θέση που καταλαμβάνει το αντίστοιχο στοιχείο στον πίνακα.

Άσκηση 1.6. Είναι ένας πίνακας ταξινομημένος σε φθίνουσα σειρά ένας σωρός-μέγιστου; Είναι η ακολουθία (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) ένας σωρός-μέγιστου;

Ο σωρός είναι μια δομή δεδομένων για διαχείριση ουρών προτεραιότητας (priority queues). Υποστηρίζει τις λειτουργίες της εύρεσης του μέγιστου / ελάχιστου στοιχείου, εξαγωγής του μέγιστου / ελάχιστου στοιχείου, αύξησης της προτεραιότητας ενός στοιχείου, και εισαγωγής νέου στοιχείου. Επίσης, θα δούμε πως μπορεί να κτιστεί ένας σωρός από τα στοιχεία ενός πίνακα σε γραμμικό χρόνο.

Η *εύρεση του μέγιστου* στοιχείου σε ένα σωρό-μέγιστου (αντίστοιχα του ελάχιστου στοιχείου σε ένα σωρό-ελάχιστου) γίνεται εύκολα σε χρόνο Ο(1) αφού το μέγιστο (ελάχιστο) στοιχείο είναι πάντα αποθηκευμένο στη ρίζα.

Κατά την *εξαγωγή του μέγιστου* στοιχείου, το περιεχόμενο της ρίζας αποθηκεύεται για να επιστραφεί. Το τελευταίο στοιχείο του σωρού παίρνει τη θέση της ρίζας και ο αριθμός των στοιχείων μειώνεται κατά 1. Η μετάθεση του τελευταίου στοιχείου στη θέση της ρίζας μπορεί να οδηγήσει σε παραβίαση της ιδιότητας του σωρού (το αντίστοιχο στοιχείο να είναι μικρότερο από κάποιο από τα παιδιά του). Η αποκατάσταση της ιδιότητας του σωρού γίνεται με κλήση της συνάρτησης Fix-Heap.

```
Extract-Max(A)
    if hs(A) < 1 then error("empty heap");
    max ← A[1];  A[1] ← A[hs(A)];
    hs(A) ← hs(A) – 1;
    Fix-Heap(A, 1);
    return(max);
```

Ο χρόνος εκτέλεσης για την εξαγωγή του μέγιστου στοιχείου εξαρτάται από το χρόνο που χρειάζεται για να αποκατασταθεί η ιδιότητα του σωρού από τη Fix-Heap.

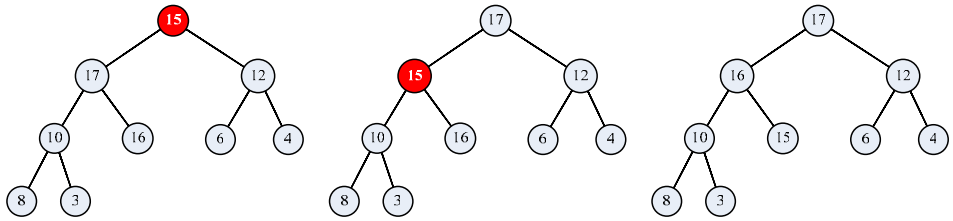
Η Fix-Heap δέχεται σαν παράμετρο τον πίνακα A και τη θέση *i* στην οποία είναι ενδεχόμενο να έχει παραβιαστεί η ιδιότητα του σωρού. Η Fix-Heap λειτουργεί αναδρομικά υποθέτοντας ότι τα δύο υποδέντρα του A[*i*] έχουν την ιδιότητα του σωρού (αυτό προφανώς ισχύει στην περίπτωση της εξαγωγής του μέγιστου στοιχείου αφού η ιδιότητα του σωρού παραβιάζεται τοπικά εξαιτίας της αντικατάστασης του στοιχείου της ρίζας ή της αντιμετάθεσης δύο στοιχείων). Η Fix-Heap αρχικά εξετάζει αν πράγματι υπάρχει παραβίαση της ιδιότητας του σωρού, δηλαδή αν το A[*i*] είναι μικρότερο από το μεγαλύτερο των παιδιών του. Σε αυτή την περίπτωση, το A[*i*] αντιμετατίθεται με το μεγαλύτερο από τα παιδιά του και η ιδιότητα του σωρού αποκαθίσταται τοπικά. Όμως είναι δυνατόν η παραβίαση να έχει μεταφερθεί στο υποδέντρο του οποίου η ρίζα αντιμετατέθηκε με τον πατέρα της. Για να διορθωθεί αυτή η παραβίαση, η Fix-Heap λειτουργεί αναδρομικά καλώντας τον εαυτό της με παράμετρο τη ρίζα του αντίστοιχου υποδέντρου (Σχήμα 1.2).

```
Fix-Heap(A, i)
largest ← i; left ← Left(i); right ← Right(i);
if left ≤ hs(A) and A[left] > A[largest] then
      largest ← left;
if right ≤ hs(A) and A[right] > A[largest] then
      largest ← right;
if largest ≠ i then
      swap(A[i], A[largest]);
      Fix-Heap(A, largest);
```

Η ορθότητα της Fix-Heap εξασφαλίζεται γιατί το μεγαλύτερο από τα δύο παιδιά τοποθετείται στη ρίζα. Έτσι η ιδιότητα του σωρού αποκαθίσταται τοπικά. Το στοιχείο που «ανέβηκε» στη ρίζα δεν μπορεί να είναι μεγαλύτερο από τον πατέρα του γιατί η

ιδιότητα του σωρού εξασφαλίζει ότι (πριν συμβεί η μεταβολή) το μεγαλύτερο στοιχείο κάθε υποδέντρου είναι στη ρίζα του.

Η Fix-Heap χρειάζεται χρόνο O(1) για να ελέγξει και να αποκαταστήσει την ιδιότητα του σωρού τοπικά. Αφού σε κάθε αναδρομική κλήση, η παραβίαση μεταφέρεται ένα επίπεδο χαμηλότερα, ο συνολικός αριθμός των κλήσεων της Fix-Heap δεν μπορεί να ξεπερνά το ύψος του A[*i*] στο σωρό. Το ύψος του σωρού είναι λογαριθμικό στον αριθμό των στοιχείων του και συνεπώς η Fix-Heap έχει χρόνο εκτέλεσης O(log *n*).



Σχήμα 1.2. Εξαγωγή του μέγιστου από το σωρό του Σχήματος 1.1. Το τελευταίο στοιχείο (15) παίρνει τη θέση του μέγιστου (20). Η ιδιότητα του σωρού παραβιάζεται γιατί 17 > 15. Η Fix-Heap αντιμεταθέτει το 15 και το 17 και συνεχίζει στο αριστερό υποδέντρο. Η ιδιότητα του σωρού παραβιάζεται γιατί 16 > 15. Η Fix-Heap αντιμεταθέτει το 16 με το 16 και συνεχίζει στο δεξιό υποδέντρο. Το 15 δεν έχει πλέον παιδιά και άρα η ιδιότητα του σωρού έχει αποκατασταθεί πλήρως.

Άσκηση 1.7. Τι συμβαίνει αν καλέσετε τη Fix-Heap με παράμετρο $i > hs(A) / 2$;

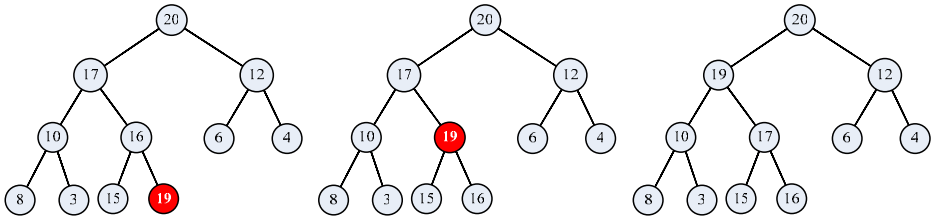
Άσκηση 1.8. Να γράψετε σε ψευδοκώδικα την επαναληπτική εκδοχή της συνάρτησης Fix-Heap.

Κάθε *νέο στοιχείο εισάγεται* αρχικά σαν τελευταίο στοιχείο του σωρού (δεξιότερο φύλλο στο τελευταίο επίπεδο) και ο αριθμός των στοιχείων του σωρού αυξάνεται κατά 1. Η εισαγωγή του νέου στοιχείου μπορεί να προκαλέσει παραβίαση της ιδιότητας του σωρού αφού το νέο στοιχείο μπορεί να είναι μεγαλύτερο από τον πατέρα του. Η ιδιότητα του σωρού αποκαθίσταται αντιμεταθέτοντας το νέο στοιχείο με τον πατέρα του ενόσω η παραβίαση συνεχίζει να υφίσταται (Σχήμα 1.3). Ο παρακάτω ψευδοκώδικας υλοποιεί την εισαγωγή νέου στοιχείου σε ένα σωρό-μέγιστου.

```
Insert-Heap(A, k)
    hs(A) ← hs(A)+1;
    A[hs(A)] ← k;
    i ← hs(A);
    while i > 1 and A[Parent(i)] < A[i] do
        swap(A[Parent(i)], A[i]);
        i ← P(i);
```

Για την ορθότητα του αλγόριθμου εισαγωγής, η ιδιότητα του σωρού μπορεί να παραβιάζεται μόνο τοπικά στην τρέχουσα θέση του νέου στοιχείου. Η «υποβάθμιση» ενός στοιχείου σε χαμηλότερο επίπεδο δεν μπορεί να οδηγήσει σε παραβίαση της ιδιότητας του σωρού, γιατί (πριν την εισαγωγή του νέου στοιχείου) όλα τα στοιχεία κάθε υποδέντρου είναι μικρότερα ή ίσα του στοιχείου που βρίσκεται στη ρίζα του.

Σε κάθε εκτέλεση του while-loop, το νέο στοιχείο ανεβαίνει κατά ένα επίπεδο στο δυαδικό δέντρο που αναπαριστά το σωρό. Αφού το ύψος του δέντρου είναι λογαριθμικό, ο αριθμός των επαναλήψεων του while-loop και ο χρόνος εκτέλεσης της λειτουργίας εισαγωγής είναι O(log *n*).



Σχήμα 1.3. Εισαγωγή του στοιχείου 19 στο σωρό του Σχήματος 1.1. Το 19 εισάγεται σαν τελευταίο στοιχείο. Η ιδιότητα του σωρού παραβιάζεται αφού 16 < 19 και τα 16 και 19 αντιμετατίθενται. Η ιδιότητα του σωρού συνεχίζει να παραβιάζεται αφού 17 < 19 και τα 17 και 19 αντιμετατίθενται. Τέλος η ιδιότητα του σωρού αποκαθίσταται.

Άσκηση 1.9. Η *αύξηση της προτεραιότητας* ενός στοιχείου (στη μέχρι τώρα συζήτηση, η προτεραιότητα ενός στοιχείου ταυτίζεται με το ίδιο το στοιχείο) αντιμετωπίζεται με τρόπο παρόμοιο προς την εισαγωγή νέου στοιχείου: Ενόσω έχουμε παραβίαση της ιδιότητας του σωρού, το στοιχείο με την αυξημένη προτεραιότητα αντιμετατίθεται με τον πατέρα του. Να γράψετε τον ψευδοκώδικα για τη λειτουργία της αύξησης προτεραιότητας. Να περιγράψετε την εκτέλεση του ψευδοκώδικα όταν το στοιχείο 3 στο

σωρό του Σχήματος 1.1 αυξάνεται σε 25. Να αποδείξετε ότι η αύξηση προτεραιότητας έχει χρόνο εκτέλεσης O(log *n*).

Άσκηση 1.10. Μία ακολουθία *n* στοιχείων μπορεί να μεταβληθεί σε σωρό εισάγοντας (με κλήση της Insert-Heap) τα στοιχεία το ένα μετά το άλλο σε έναν αρχικά άδειο σωρό. Να αποδείξετε ότι αυτή η διαδικασία δημιουργίας σωρού έχει χρόνο εκτέλεσης χειρότερης περίπτωσης Θ(*n* log *n*).

Χρησιμοποιώντας τη Fix-Heap μπορούμε να μετατρέψουμε μία ακολουθία *n* στοιχείων σε σωρό σε χρόνο Θ(*n*).

```
Create-Heap(A[1...n])
    hs(A) ← n;
    for i ← ⌊n/2⌋ downto 1 do
        Fix-Heap(A, i);
```

Κατ' αρχήν παρατηρούμε ότι η Fix-Heap χρειάζεται να κληθεί μόνο για τιμές της παραμέτρου *i* από $\lfloor n/2 \rfloor$ μέχρι 1 (Άσκηση 1.7). Χρησιμοποιώντας μαθηματική επαγωγή, θα αποδείξουμε ότι μετά την ολοκλήρωση του for-loop για κάθε τιμή της μεταβλητής *i*, $\lfloor n/2 \rfloor \geq i \geq 1$, το A[*i*] και οι απόγονοί του (A[*L*(*i*)], A[*R*(*i*)], A[*L*(*L*(*i*))], A[*R*(*L*(*i*))], A[*L*(*R*(*i*))], A[*R*(*R*(*i*))], …) ικανοποιούν την ιδιότητα του σωρού.

Η παραπάνω αμετάβλητη συνθήκη προφανώς ισχύει για τα φύλλα πριν την πρώτη εκτέλεση του for-loop (βάση της επαγωγής – βλ. Άσκηση 1.7). Όταν τα υποδέντρα A[*L*(*i*)] και A[*R*(*i*)] ικανοποιούν την ιδιότητα του σωρού (επαγωγική υπόθεση), η Fix-Heap με παράμετρο *i* αποκαθιστά την ιδιότητα του σωρού στο υποδέντρο με ρίζα *i*. Συνεπώς, η αμετάβλητη συνθήκη ισχύει μετά την ολοκλήρωση του for-loop για κάθε τιμή της μεταβλητής *i*, $\lfloor n/2 \rfloor \geq i \geq 1$. Μετά την ολοκλήρωση και του τελευταίου for-loop, το υποδέντρο με ρίζα A[1] ικανοποιεί την ιδιότητα του σωρού και ο πίνακας A αποτελεί ένα σωρό με *n* στοιχεία.

Ο χρόνος εκτέλεσης της Fix-Heap με παράμετρο *i* είναι ανάλογος του ύψους του A[*i*]. Αφού ο σωρός αντιστοιχεί σε ένα σχεδόν πλήρες δυαδικό δέντρο, το ύψος ενός σωρού με *n* στοιχεία δεν ξεπερνά το log *n* και ο αριθμός των στοιχείων σε ύψος *k*, $0 \leq k \leq \log n$, δεν ξεπερνά το $\frac{n}{2^k}$ (δηλαδή, έχουμε το πολύ *n* στοιχεία σε ύψος 0 – φύλλα, το πολύ *n* / 2 στοιχεία σε ύψος 1, κοκ., και ένα στοιχείο σε ύψος log *n* – τη ρίζα).

Αφού η Fix-Heap χρειάζεται χρόνο O(k) για κάθε στοιχείο σε ύψος k, ο συνολικός χρόνος εκτέλεσης της Create-Heap δεν ξεπερνά το άθροισμα:

$$\sum_{k=0}^{\log n} \frac{n}{2^k} O(k) = O\left(n \sum_{k=0}^{\log n} \frac{k}{2^k} \right) = O(n) \text{, επειδή } \sum_{k=0}^{\infty} \frac{k}{2^k} = 2$$

Συνεπώς, ο χρόνος εκτέλεσης της Create-Heap είναι Θ(n) (σε καμία περίπτωση δεν μπορεί να είναι ο(n) αφού το for-loop εκτελείται τουλάχιστον n/2 φορές).

Άσκηση 1.11. Εξηγήστε πως λειτουργεί η συνάρτηση Create-Heap με είσοδο την ακολουθία (3, 4, 6, 10, 8, 15, 16, 17, 12, 20). Πώς λειτουργεί ο αλγόριθμος που προτείνεται στην Άσκηση 1.10 με είσοδο την ίδια ακολουθία; Να συγκρίνετε τον αριθμό των συγκρίσεων και αντιμεταθέσεων στοιχείων που κάνει κάθε αλγόριθμος. Καταλήγουν οι δύο αλγόριθμοι στον ίδιο σωρό;

Άσκηση 1.12. Εξηγήστε γιατί η συνάρτηση Create-Heap λειτουργεί μειώνοντας την τιμή της μεταβλητής i (ξεκινάει από $\lfloor n/2 \rfloor$ και καταλήγει σε 1) και όχι αυξάνοντάς την (π.χ. ξεκινώντας από 1 και καταλήγοντας σε $\lfloor n/2 \rfloor$); Με άλλα λόγια, πρέπει να εξηγήσετε γιατί η Create-Heap κτίζει το σωρό από κάτω-προς-τα-πάνω (bottom-up) και όχι από πάνω-προς-τα-κάτω (top-down).

Άσκηση 1.13. Εξηγήστε πως λειτουργεί η συνάρτηση Create-Heap με είσοδο την ακολουθία (3, 4, 6, 10, 8, 15, 16, 17, 12, 20). Πώς λειτουργεί ο αλγόριθμος που προτείνεται στην Άσκηση 1.10 με είσοδο την ίδια ακολουθία; Να συγκρίνετε τον αριθμό των συγκρίσεων και αντιμεταθέσεων στοιχείων που κάνει κάθε αλγόριθμος. Καταλήγουν οι δύο αλγόριθμοι στον ίδιο σωρό;

1.4.2. Ο Αλγόριθμος Ταξινόμησης Σωρού

Γνωρίζοντας τις βασικές ιδιότητες και τις λειτουργίες που υποστηρίζονται από τη δομή δεδομένων του σωρού, ο αλγόριθμος ταξινόμησης *σωρού* (heapsort) προκύπτει μάλλον εύκολα από την ιδέα της ταξινόμησης με επιλογή.

Η ταξινόμηση σωρού λειτουργεί σε φάσεις διατηρώντας στο δεξιό μέρος του πίνακα τα μεγαλύτερα στοιχεία ταξινομημένα και στο αριστερό μέρος του πίνακα τα μη-ταξινομημένα στοιχεία συγκροτημένα σε σωρό. Σε κάθε φάση, αντιμεταθέτουμε το

μεγαλύτερο στοιχείο του μη-ταξινομημένου υποπίνακα (βρίσκεται στην πρώτη θέση του σωρού) με το τελευταίο στοιχείο του σωρού / μη-ταξινομημένου υποπίνακα. Έτσι το μεγαλύτερο στοιχείο του μη-ταξινομημένου υποπίνακα γίνεται το πρώτο στοιχείο του ταξινομημένου υποπίνακα. Ο αριθμός των ταξινομημένων στοιχείων αυξάνεται κατά 1 με αντίστοιχη μείωση των στοιχείων του σωρού. Όσον αφορά το σωρό, η αντιμετάθεση του μεγαλύτερου στοιχείου με το τελευταίο στοιχείο ισοδυναμεί με εξαγωγή του μέγιστου στοιχείο. Η κλήση της Fix-Heap(A, 1) αποκαθιστά την ιδιότητα του σωρού (βλ. επίσης Extract-Max) και η τρέχουσα φάση ολοκληρώνεται.

```
Heapsort(A[1...n])
    Create-Heap(A[1...n]);
    hs(A) ← n;
    for i ← n downto 2 do
        swap(A[1], A[i]);
        hs(A) ← hs(A)-1;
        Fix-Heap(A, 1);
```

Η ορθότητα του αλγόριθμου ταξινόμησης προκύπτει από την εξής αμετάβλητη συνθήκη: Πριν την (n – i + 1)-οστή επανάληψη του for-loop, ο υποπίνακας A[1…i] περιέχει τα i μικρότερα στοιχεία και το στοιχείο A[1] είναι το μεγαλύτερο στοιχείο του υποπίνακα A[1…i]. Η αμετάβλητη συνθήκη ισχύει γιατί πριν την (n – i)-οστή επανάληψη του for-loop, τα στοιχεία του υποπίνακα A[1…i +1] συγκροτούν ένα σωρό-μέγιστου. Αρχικά ο σωρός κατασκευάζεται με κλήση της Create-Heap. Μετά από κάθε εξαγωγή του μέγιστου στοιχείου, ο σωρός χάνει ένα στοιχείο (το τελευταίο του στοιχείο παίρνει τη θέση της ρίζας) και αποκαθίσταται με κλήση της Fix-Heap. Τα στοιχεία που απομένουν στον υποπίνακα A[1…i] είναι τα i μικρότερα στοιχεία αν δεχτούμε (επαγωγικά) ότι τα στοιχεία του υποπίνακα A[1…i+1] ήταν τα i+1 μικρότερα.

Επομένως, μετά την πρώτη εκτέλεση του for-loop, το μεγαλύτερο στοιχείο του πίνακα βρίσκεται στην τελευταία θέση, μετά τη δεύτερη εκτέλεση, το δεύτερο μεγαλύτερη στοιχείο βρίσκεται στην προτελευταία θέση, κοκ. Τελικά ο αλγόριθμος τερματίζει με όλα τα στοιχεία του πίνακα ταξινομημένα.

Όσον αφορά το χρόνο εκτέλεσης, η Create-Heap έχει χρόνο εκτέλεσης Θ(n). Το for-loop εκτελείται n – 1 φορές και κάθε επανάληψή του χρειάζεται χρόνο O(1) για τη swap και τη μείωση του μεγέθους του σωρού και O(log n) για την κλήση της Fix-Heap. Ο

συνολικός χρόνος εκτέλεσης του αλγόριθμου είναι $O(n \log n)$. Επιπλέον, ο αλγόριθμος ταξινόμησης σωρού λειτουργεί εντός του δοθέντος χώρου.

Ο αλγόριθμος ταξινόμησης σωρού είναι κατάλληλος για την ταξινόμηση μεγάλου αριθμού στοιχείων που μπορούν να χωρέσουν στην κύρια μνήμη. Εξ' αιτίας των συχνών τυχαίων προσπελάσεων που απαιτεί η δομή του σωρού, η ταξινόμηση σωρού δεν είναι κατάλληλη για δεδομένα που δεν μπορούν να χωρέσουν στην κύρια μνήμη.

Άσκηση 1.14. Εξηγήστε πως λειτουργεί ο αλγόριθμος ταξινόμησης σωρού με είσοδο την ακολουθία (3, 8, 17, 10, 4, 20, 16, 15, 12, 6). Ποιος είναι ο συνολικός αριθμός συγκρίσεων και μετατοπίσεων στοιχείων που εκτελεί ο αλγόριθμος.

Άσκηση 1.15. Ποιος είναι ο χρόνος εκτέλεσης της ταξινόμησης σωρού όταν ο πίνακας Α δίνεται ταξινομημένος σε αύξουσα σειρά; Τι συμβαίνει στην περίπτωση που ο πίνακας Α δίνεται ταξινομημένος σε φθίνουσα σειρά;

Άσκηση 1.16. Να αποδείξετε ότι ο χρόνος εκτέλεσης χειρότερης περίπτωσης της ταξινόμησης σωρού είναι $\Theta(n \log n)$ βρίσκοντας στιγμιότυπα για τα οποία ο αλγόριθμος χρειάζεται αυτό το χρόνο.

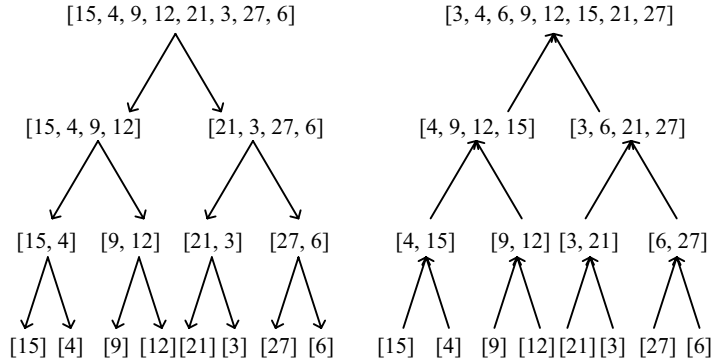
1.5. Ταξινόμηση με Συγχώνευση

Ο αλγόριθμος ταξινόμησης με *συγχώνευση* (mergesort) είναι ένας τυπικός αναδρομικός αλγόριθμος της κατηγορίας «διαίρει και βασίλευε» (divide and conquer). Η ταξινόμηση με συγχώνευση διαιρεί την ακολουθία εισόδου σε δύο υποακολουθίες ίσου μήκους, ταξινομεί τις δύο υποακολουθίες καλώντας αναδρομικά τον εαυτό της, και συγχωνεύει τις ταξινομημένες ακολουθίες ώστε η τελική ακολουθία να είναι ταξινομημένη.

Ο παρακάτω ψευδοκώδικας υλοποιεί τον αλγόριθμο ταξινόμησης με συγχώνεση. Επειδή ο αλγόριθμος είναι αναδρομικός (recursive), χρησιμοποιούμε τα p και r για να δηλώσουμε τα όρια του υποπίνακα $A[p \ldots r]$ που πρέπει να ταξινομηθεί από κάθε αναδρομική κλήση. Αν $p \geq r$ και ο αντίστοιχος υποπίνακας περιέχει ένα μόνο στοιχείο, ο υποπίνακας θεωρείται ταξινομημένος και ο αλγόριθμος τερματίζει χωρίς άλλη ενέργεια. Διαφορετικά, η ακολουθία / υποπίνακας εισόδου διαιρείται σε δύο υποακολουθίες με (σχεδόν) ίσου μέγεθος, τις $A[p \ldots q]$ και $A[q+1 \ldots r]$, οι οποίες ταξινομούνται με

αναδρομικές κλήσεις του ίδιου αλγόριθμου. Οι ταξινομημένες υποακολουθίες συγχωνεύονται σε μία ταξινομημένη ακολουθία από την συνάρτηση Merge.

```
Merge-Sort(A[p...r])
    /* Ταξινομεί τον υποπίνακα A[p...r] */
    if p < r then
    /* Διαίρεση σε δύο υποακολουθίες και ταξινόμησή τους
       με αναδρομική κλήση του ίδιου αλγόριθμου */
        q ← (p + r)/2;
        Merge-Sort(A[p...q]);
        Merge-Sort(A[q+1...r]);
    /* Συγχώνευση σε μία ταξινομημένη ακολουθία */
        Merge(A[p...q], A[q+1...r]);
```



Σχήμα 1.4. Παράδειγμα ταξινόμησης με συγχώνευση για A = [15, 4, 9, 12, 21, 3, 27, 6]. Οι αναδρομικές κλήσεις της Merge-Sort διαιρούν τον πίνακα στη μέση. Η ανάπτυξη της αναδρομής σταματά όταν κάθε υποπίνακας περιέχει ένα μόνο στοιχείο. Από αυτό το σημείο, με διαδοχικές συνενώσεις, καταλήγουμε στον ταξινομημένο πίνακα A = [3, 4, 6, 9, 12, 15, 21, 27]. Αριστερά φαίνονται οι διαδοχικές διαιρέσεις του πίνακα Α κατά την ανάπτυξη της αναδρομής. Δεξιά φαίνονται οι συγχωνεύσεις των ταξινομημένων υποπινάκων από τη συνάρτηση Merge.

Σημαντικό ρόλο στη λειτουργία της ταξινόμησης με συγχώνευση παίζει η συνάρτηση Merge που συγχωνεύει δύο ταξινομημένες ακολουθίες σε μία ακολουθία επίσης ταξινομημένη. Η συγχώνευση εξελίσσεται σε φάσεις. Σε κάθε φάση συγκρίνουμε

τα μικρότερα στοιχεία των δύο υποακολουθιών που δεν έχουν ακόμη μεταφερθεί στην «τελική» ακολουθία, δηλαδή την ακολουθία που θα αποτελέσει το αποτέλεσμα της συγχώνευσης. Το μικρότερο από τα δύο στοιχεία μεταφέρεται στην «τελική» ακολουθία και η φάση ολοκληρώνεται.

Ο παρακάτω ψευδοκώδικας υλοποιεί τη λειτουργία της συγχώνευσης. Ο πίνακας X χρησιμοποιείται σαν περιοχή προσωρινής αποθήκευσης για τις δύο υποακολουθίες. Τα στοιχεία των υποπινάκων A[p…q] και A[q+1…r] αντιγράφονται στις αντίστοιχες θέσεις του πίνακα X. Ο πίνακας A[p…r] χρησιμοποιείται για να αποθηκευθεί η «τελική» ακολουθία που θα προκύψει από τη συγχώνευση. Ο δείκτης k, $p \le k \le r$, υποδεικνύει τη θέση στην «τελική» ακολουθία όπου θα μεταφερθεί το επόμενο στοιχείο, και οι δείκτες i, $p \le i \le q$, και j, $q+1 \le j \le r$, υποδεικνύουν τη θέση των μικρότερων στοιχείων της πρώτης και της δεύτερης υποακολουθίας που δεν έχουν ακόμη μεταφερθεί στην «τελική» ακολουθία. Το μικρότερο από τα στοιχεία X[i] και X[j] αντιγράφεται στη θέση A[k] με μετακίνηση των αντίστοιχων δεικτών μία θέση προς τα δεξιά. Ειδική μέριμνα λαμβάνεται και για την περίπτωση που τα στοιχεία της μίας υποακολουθίας εξαντληθούν.

```
Merge(A[p...q], A[q+1...r])
    array X[p...r];
    X[p...q] ← A[p...q];
    X[q+1...r] ← A[q+1...r];

    i ← p; j ← q + 1;

    for k ← p to r do
        if i > q then
            A[k] ← X[j]; j ← j + 1; continue;
        if j > r then
            A[k] ← X[i]; i ← i + 1; continue;
        if (i ≤ q) and (j ≤ r) and (X[i] > X[j]) then
            A[k] ← X[j]; j ← j + 1;
        else
            A[k] ← X[i]; i ← i + 1;
```

Ορθότητα και Χρόνος Εκτέλεσης της Merge. Εφόσον οι δύο υποακολουθίες εισόδου είναι ταξινομημένες, όταν το k-οστό στοιχείο μεταφέρεται στον πίνακα A, δεν υπάρχει στοιχείο μεγαλύτερο από το A[k] στους υποπίνακες X[i…q] και X[j…r]. Επομένως, μετά την ολοκλήρωσης της συνάρτησης Merge, η ακολουθία A[p…r] που προκύπτει από τη συγχώνευση είναι ταξινομημένη σε αύξουσα σειρά.

Όσον αφορά στον χρόνο εκτέλεσης, η αντιγραφή των στοιχείων στον πίνακα X χρειάζεται γραμμικό χρόνο στο μέγεθος της ακολουθίας που προκύπτει από τη συγχώνευση. Το for-loop εκτελείται r–p+1 φορές και κάθε εκτέλεσή του ολοκληρώνεται σε χρόνο Θ(1). Ο συνολικός χρόνος εκτέλεσης της συνάρτησης Merge είναι Θ(r–p+1), δηλαδή γραμμικός στον αριθμό των στοιχείων της ακολουθίας που προκύπτει από τη συγχώνευση.

Ορθότητα της Merge-Sort. Αν ο υποπίνακας A[p…r] περιέχει ένα μόνο στοιχείο, τότε ο αλγόριθμος επιστρέφει τον υποπίνακα ταξινομημένο (αυτό είναι τετριμμένο αφού δεν υπάρχει τίποτα προς ταξινόμηση). Επαγωγικά μπορούμε να υποθέσουμε ότι οι αναδρομικές κλήσεις Merge-Sort(A[p...q]) και Merge-Sort(A[q+1...r]) επιστρέφουν τις αντίστοιχες υποακολουθίες ταξινομημένες. Η ορθότητα του αλγόριθμου προκύπτει από το γεγονός ότι η συνάρτηση Merge συγχωνεύει τις (ταξινομημένες) υποακολουθίες A[p...q] και A[q+1...r] σε μία ταξινομημένη ακολουθία A[p…r]. Επομένως, η αρχική κλήση Merge-Sort(A[1…n]) ολοκληρώνεται με τον πίνακα A ταξινομημένο σε αύξουσα σειρά.

Χρόνος Εκτέλεσης της Merge-Sort. Ο χρόνος εκτέλεσης των αναδρομικών αλγόριθμων, όπως είναι η ταξινόμηση με συγχώνευση, προκύπτει διατυπώνοντας και λύνοντας την αντίστοιχη αναδρομική εξίσωση.

Έστω $T(n)$ ο χρόνος για την ταξινόμηση n στοιχείων, $n > 1$, με τη μέθοδο της συγχώνευσης. Ο χρόνος αυτός είναι ίσος με $2\,T(n/2)$, το οποίο εκφράζει το χρόνο που χρειάζονται οι δύο αναδρομικές κλήσεις για την ταξινόμηση των δύο υποακολουθιών μεγέθους n / 2, συν Θ(n), που είναι ο χρόνος που χρειάζεται η συνάρτηση Merge για να συγχωνεύσει τις δύο ταξινομημένες υποακολουθίες σε μία ταξινομημένη ακολουθία n στοιχείων. Δηλαδή είναι $T(n) = 2T(n/2) + \Theta(n)$. Στην περίπτωση που $n = 1$ (αρχική συνθήκη), η ταξινόμηση με συγχώνευση χρειάζεται σταθερό χρόνο. Δηλαδή είναι $T(1) = \Theta(1)$. Επομένως, ο χρόνος εκτέλεσης για την ταξινόμηση με συγχώνευση δίνεται από την αναδρομική εξίσωση:

$$T(n) = \begin{cases} \Theta(1) & \text{αν n} = 1, \\ 2T(n/2) + \Theta(n) & \text{αν n} > 1. \end{cases}$$

Λύνοντας την αναδρομική εξίσωση (π.χ. με το Θεώρημα της Κυριαρχίας), προκύπτει ότι ο χρόνος εκτέλεσης για τον αλγόριθμο ταξινόμησης με συγχώνευση είναι Θ($n \log n$).

Ο αλγόριθμος ταξινόμησης με συγχώνευση δεν λειτουργεί εντός του δοθέντος χώρου, αφού η συνάρτηση Merge χρειάζεται να δημιουργήσει ένα αντίγραφο των στοιχείων πριν προχωρήσει στη συγχώνευση. Όμως μπορεί να υλοποιηθεί ώστε τα δεδομένα να προσπελαύνονται σειριακά (από αριστερά προς τα δεξιά) τόσο στη φάση της ανάπτυξης της αναδρομής όσο και στη φάση της συγχώνευσης. Το γεγονός αυτό καθιστά την ταξινόμηση με συγχώνευση κατάλληλη για την ταξινόμηση δεδομένων που είναι αποθηκευμένα στη δευτερεύουσα μνήμη (π.χ. σκληρό δίσκο).

1.5.1. Η Μέθοδος «Διαίρει και Βασίλευε»

Η ταξινόμηση με συγχώνευση είναι ένα τυπικό παράδειγμα αναδρομικού αλγόριθμου που βασίζεται στη μέθοδο «διαίρει και βασίλευε» (divide-and-conquer). Η ιδέα του «διαίρει και βασίλευε» παρέχει μια απλή και ισχυρή τεχνική σχεδιασμού αλγορίθμων. Η μέθοδος «διαίρει και βασίλευε» βρίσκει εφαρμογή σε πληθώρα προβλημάτων από διαφορετικά πεδία (π.χ. προβλήματα ταξινόμησης, πολλαπλασιασμού πινάκων, πολλαπλασιασμού πολυωνύμων, μετασχηματισμός Fourier).

Η μέθοδος του «διαίρει και βασίλευε» συνίσταται στη διάσπαση της εισόδου σε μικρότερα, επιμέρους στιγμιότυπα του ίδιου προβλήματος, στην επίλυση των επιμέρους στιγμιότυπων, και στον υπολογισμό της λύσης για το αρχικό στιγμιότυπο από τις λύσεις των επιμέρους στιγμιότυπων. Η επίλυση των επιμέρους στιγμιότυπων γίνεται συνήθως με αναδρομική επίκληση του ίδιου αλγορίθμου. Σε αυτές τις περιπτώσεις, το αποτέλεσμα της μεθόδου «διαίρει και βασίλευε» είναι ένας αναδρομικός αλγόριθμος. Μερικές φορές, όταν το μέγεθος των επιμέρους στιγμιότυπων γίνει αρκετά μικρό, είναι προτιμότερη η εφαρμογή κάποιου άλλου, μη-αναδρομικού αλγορίθμου.

Η μέθοδος «διαίρει και βασίλευε» βρίσκει εφαρμογή σε όσα προβλήματα η λύση ενός αρχικού στιγμιότυπου εισόδου μπορεί να συντεθεί από τις λύσεις επιμέρους στιγμιότυπων που έχουν προέλθει από τη διάσπαση του αρχικού. Για παράδειγμα, στον αλγόριθμο ταξινόμησης με συγχώνευση, κατασκευάζουμε την ταξινομημένη ακολουθία συγχωνεύοντας τις δύο ταξινομημένες υποακολουθίες που έχουν προέλθει από τη διαίρεση της (μη ταξινομημένης) ακολουθίας εισόδου. Η ταξινόμηση των υποακολουθιών γίνεται αναδρομικά εφαρμόζοντας τον ίδιο αλγορίθμο.

Εκτός από την απλότητα στη σύλληψη και την εφαρμογή, ένα σημαντικό πλεονέκτημα της μεθόδου «διαίρει και βασίλευε» είναι ότι οδηγεί σε αλγορίθμους που

είναι εύκολο να αναλυθούν. Η ανάλυση ενός αλγορίθμου που βασίζεται στη μέθοδο «διαίρει και βασίλευε» συνίσταται στη διατύπωση και την επίλυση της αναδρομικής εξίσωσης που διέπει τη λειτουργία του αλγορίθμου. Η αναδρομική εξίσωση συνήθως προκύπτει απευθείας από την ιδέα στην οποία βασίζεται ο αλγόριθμος. Επιπλέον, είναι γνωστά πολλά ισχυρά μαθηματικά εργαλεία για την επίλυση αναδρομικών εξισώσεων.

1.6. Ταχεία Ταξινόμηση

Ο αλγόριθμος *ταχείας* ταξινόμησης (quicksort) παρουσιάστηκε από τον C.A.R. Hoare το 1962 και βασίζεται επίσης στη μέθοδο του «διαίρει και βασίλευε». Αν και ο χρόνος εκτέλεσης χειρότερης περίπτωσης του αλγόριθμου ταχείας ταξινόμησης είναι $\Theta(n^2)$, στην πράξη αποδεικνύεται πολύ γρήγορος και έχει χρόνο εκτέλεσης μέσης περίπτωσης $O(n \log n)$.

Όπως κάθε αλγόριθμος που βασίζεται στη μέθοδο «διαίρει και βασίλευε», η ταχεία ταξινόμηση ενός πίνακα $A[p\ldots r]$ ακολουθεί τα παρακάτω βήματα.

- **Διαίρεση.** Με βάση κάποιο στοιχείο του πίνακα $A[p\ldots r]$ (π.χ. το πρώτο ή κάποιο τυχαία επιλεγμένο στοιχείο), ο πίνακας αναδιατάσσεται και διαιρείται σε δύο υποπίνακες $A[p\ldots q]$ και $A[q+1\ldots r]$. Μετά τη διαίρεση κάθε στοιχείο του υποπίνακα $A[p\ldots q]$ είναι μικρότερο ή ίσο από κάθε στοιχείο του υποπίνακα $A[q+1\ldots r]$. Ο υπολογισμός του δείκτη q είναι μέρος της διαδικασίας διαίρεσης (σε αντίθεση με την ταξινόμηση συγχώνευσης που είναι πάντα $q = (p + r) / 2$).

- **Επίλυση επιμέρους στιγμιότυπων.** Οι υποπίνακες $A[p\ldots q]$ και $A[q+1\ldots r]$ ταξινομούνται με αναδρομικές κλήσεις της ταχείας ταξινόμησης.

- **Σύνθεση λύσεων επιμέρους στιγμιότυπων.** Αφού κάθε στοιχείο του υποπίνακα $A[p\ldots q]$ είναι μικρότερο ή ίσο από κάθε στοιχείο του υποπίνακα $A[q+1\ldots r]$ και οι υποπίνακες $A[p\ldots q]$ και $A[q+1\ldots r]$ είναι ταξινομημένοι, ο πίνακας $A[p\ldots r]$ είναι επίσης ταξινομημένος.

Ο παρακάτω ψευδοκώδικας υλοποιεί τον αλγόριθμο ταχείας ταξινόμησης. Για την ταξινόμηση του πίνακα $A[1\ldots n]$, καλούμε Quicksort($A[1\ldots n]$).

```
Quicksort(A[p...r])
    if p < r then
        q ← Partition(A[p...r]);
        Quicksort(A[p...q]);
        Quicksort(A[q+1...r]);
```

Το πιο σημαντικό βήμα στη λειτουργία της ταχείας ταξινόμησης είναι η συνάρτηση Partition, η οποία πραγματοποιεί την αναδιάταξη και τη διαίρεση του πίνακα A[p…r].

```
Partition(A[p...r])
    x ← A[p]; i ← p − 1; j ← r + 1;
    while TRUE do
        repeat
            i ← i + 1
        until A[i] ≥ x;
        repeat
            j ← j − 1
        until A[j] ≤ x;
        if i < j then
            swap(A[i], A[j])
        else return(j);
```

Η συνάρτηση Partition οργανώνει τη διαίρεση του πίνακα A[p…r] γύρω από το στοιχείο x = A[p]. Η Partition συνεχώς επεκτείνει τις περιοχές A[p…i] και A[j…r] ξεκινώντας από το αριστερό και το δεξιό άκρο του Α αντίστοιχα. Η επέκταση γίνεται ώστε κάθε στοιχείο της περιοχής A[p…i] να είναι μικρότερο του x και κάθε στοιχείο της περιοχής A[j…r] να είναι μεγαλύτερο του x. Αν εντοπιστούν στοιχεία A[i] ≥ x (δεν μπορεί να συνεχίσει η επέκταση του A[p…i] προς τα δεξιά) και A[j] ≤ x (δεν μπορεί να συνεχίσει η επέκταση του A[j…r] προς τα αριστερά), ενώ είναι i < j (διαφορετικά έχει καλυφθεί ολόκληρος ο υποπίνακας A[p…q]), τότε τα A[i] και A[j] αντιμετατίθενται αμοιβαία και η επέκταση συνεχίζεται.

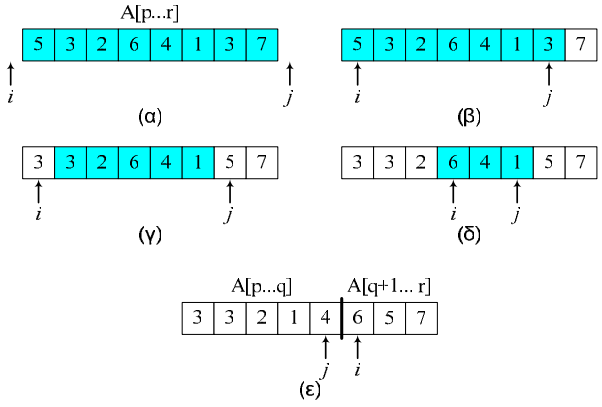
Συγκεκριμένα, αρχικά είναι $i = p - 1$ και $j = r + 1$. Συνεπώς, οι περιοχές A[p…i] και A[j…r] δεν περιέχουν στοιχεία. Το πρώτο repeat-loop εντοπίζει το πρώτο προς τα δεξιά στοιχείο που εμποδίζει την ανάπτυξη της περιοχής A[p…i], δηλαδή την πρώτη θέση i για την οποία ισχύει A[i] ≥ x. Το δεύτερο repeat-loop εντοπίζει το πρώτο προς τα αριστερά στοιχείο που εμποδίζει την ανάπτυξη της περιοχής A[j…r], δηλαδή την πρώτη θέση j για την οποία ισχύει A[j] ≤ x. Και στις δύο περιπτώσεις η επέκταση σταματά ακόμη και αν το A[i] / A[j] είναι ίσο με το x ώστε να εξασφαλιστεί ότι αμφότεροι οι υποπίνακες που

παράγονται από τη διαίρεση θα είναι μη-κενοί (θεωρήστε τις περιπτώσεις που το στοιχείο x είναι το μεγαλύτερο ή το μικρότερο στοιχείο του πίνακα A[p…r]).

Εφόσον οι δύο περιοχές δεν επικαλύπτονται (δηλαδή, εφόσον είναι i < j), το στοιχείο A[i] είναι μεγάλο για να ανήκει στην περιοχή του αριστερού άκρου, και το στοιχείο A[j] είναι μικρό για να ανήκει στην περιοχή του δεξιού άκρου. Έτσι, τα στοιχεία A[i] και A[j] αλλάζουν αμοιβαία θέσεις, και συνεχίζεται η επέκταση των δύο περιοχών. Όταν για πρώτη φορά οι περιοχές A[p…i] και A[j…r] φθάσουν να επικαλύπτονται, δηλαδή γίνει i ≥ j, η διαδικασία της διαίρεσης ολοκληρώνεται, και σαν όριο των δύο υποπινάκων ορίζεται η θέση j.

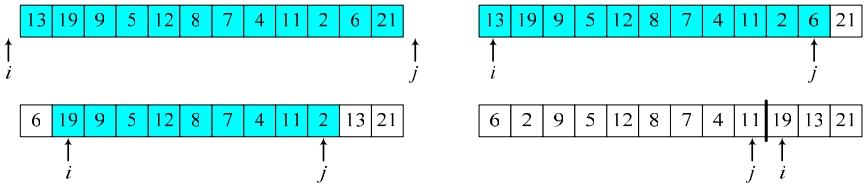
Η ορθότητα του αλγόριθμου ταχείας ταξινόμησης προκύπτει από το γεγονός ότι όλα τα στοιχεία του υποπίνακα A[p…q] είναι μικρότερα ή ίσα από όλα τα στοιχεία του υποπίνακα A[q+1…r]. Αν υποθέσουμε επαγωγικά ότι οι αναδρομικές κλήσεις Quicksort(A[p...q]) και Quicksort(A[q+1...r]) όντως ταξινομούν τα στοιχεία των αντίστοιχων υποπινάκων σε αύξουσα σειρά, όλα τα στοιχεία του πίνακα A[p…r] είναι ταξινομημένα σε αύξουσα σειρά κατά την ολοκλήρωση της κλήσης Quicksort(A[p...r]).

Άσκηση 1.17. Περιγράψτε τη λειτουργία της συνάρτησης Partition για τον πίνακα A = [5, 3, 2, 6, 4, 1, 3, 7]. Επαναλάβετε για τον πίνακα B = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21].



Σχήμα 1.5. Η λειτουργία της Partition για τον πίνακα A = [5, 3, 2, 6, 4, 1, 3, 7].

Λύση. Η αναπαράσταση λειτουργίας της συνάρτησης Partition για τον πίνακα Α φαίνεται στο Σχήμα 1.5. Τα στοιχεία που δεν έχουν τοποθετηθεί ακόμη στο σωστό υποπίνακα σημειώνονται με γκρίζο φόντο. Το στοιχείο γύρω από το οποίο οργανώνεται η διαίρεση είναι το $x = 5$. Αρχικά, το στοιχείο 5 ($5 \geq x$) δεν επιτρέπει στην περιοχή του αριστερού άκρου να μεγαλώσει, και το στοιχείο 3 ($3 \leq x$) δεν επιτρέπει στην περιοχή του δεξιού άκρου να μεγαλώσει (Σχήμα 1.5.β). Αυτό διορθώνεται με την αντιμετάθεση των στοιχείων 5 και 3 (Σχήμα 3.5.γ). Τα επόμενα στοιχεία που δεν επιτρέπουν την ανάπτυξη των δύο περιοχών είναι τα 6 και 1 (Σχήμα 3.5.δ). Μετά την αντιμετάθεση των 6 και 1, η διαίρεση ολοκληρώνεται με όριο το σημείο $q = j = 5$.



Σχήμα 1.6. Η λειτουργία της Partition για τον πίνακα Β.

Η λειτουργία της Partition για τον πίνακα Β = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21] φαίνεται στο Σχήμα 1.6.

Άσκηση 1.18. Να αποδείξετε ότι ο χρόνος εκτέλεσης της διαδικασίας Partition είναι $\Theta(n)$, όπου $n = r - p + 1$ είναι ο αριθμός των στοιχείων που συμμετέχουν στη διαίρεση.

Λύση. Κάθε στοιχείο του πίνακα Α[$p \ldots r$], εκτός από τα δύο στοιχεία που βρίσκονται εκατέρωθεν του ορίου q, συγκρίνεται ακριβώς μία φορά με το στοιχείο x γύρω από το οποίο οργανώνεται η διαίρεση. Τα στοιχεία Α[q] και Α[q+1] συγκρίνονται με το στοιχείο x το πολύ δύο φορές. Ο λόγος είναι ότι ο δείκτης i πάντα αυξάνεται, ο δείκτης j πάντα μειώνεται, και όταν ο i σταματήσει σε κάποια θέση (συμπεριλαμβανομένης και θέσης q+1), ο δείκτης j θα σταματήσει είτε δεξιότερα της θέσης i (οπότε έχουμε αντιμετάθεση στοιχείων) είτε σε κάποια από τις θέσεις i ή $i - 1$ (οπότε έχουμε ολοκλήρωση της Partition). Όσον αφορά στις μετατοπίσεις των στοιχείων, κάθε στοιχείο αλλάζει θέση το πολύ μία φορά κατά την κλήση Partition(Α[$q \ldots r$]). Μετά τη μετατόπιση, το στοιχείο δεν συγκρίνεται πάλι με το στοιχείο x. Επομένως, η συνάρτηση Partition αφιερώνει σταθερό χρόνο σε κάθε στοιχείο. Συνολικά ο χρόνος εκτέλεσης της Partition είναι $\Theta(r - p + 1) = \Theta(n)$ – γραμμικός στον αριθμό των στοιχείων. ■

Ο χρόνος εκτέλεσης $T(n)$ που χρειάζεται η ταχεία ταξινόμηση ενός πίνακα n στοιχείων είναι ίσος με το άθροισμα του $\Theta(n)$ για την εκτέλεση της Partition και των χρόνων εκτέλεσης των δύο αναδρομικών κλήσεων για την ταξινόμηση των δύο υποπινάκων. Οι χρόνοι εκτέλεσης των αναδρομικών κλήσεων εξαρτώνται από τη θέση της διαίρεσης q που καθορίζει το μέγεθος των δύο υποπινάκων. Επομένως, $T(n) = T(q) + T(n - q) + \Theta(n)$. Σαν αρχική συνθήκη μπορούμε πάντα να θεωρήσουμε ότι $T(1) = \Theta(1)$, αφού ένα μόνο στοιχείο δεν χρειάζεται ταξινόμηση.

Διαισθητικά, το μέγεθος του μεγαλύτερου από τα επιμέρους στιγμιότυπα που προκύπτει από τη διαδικασία της διαίρεσης αποτελεί μέτρο για την πρόοδο ενός αλγόριθμου «διαίρει και βασίλευε». Για την ταχεία ταξινόμηση συγκεκριμένα, αν ο μεγαλύτερος από τους δύο υποπίνακες έχει μέγεθος σχεδόν ίσο με αυτό του αρχικού πίνακα, αφενός ο αλγόριθμος ανάλωσε χρόνο $\Theta(n)$ για να κάνει τη διαίρεση, και αφετέρου ο μεγαλύτερος υποπίνακας είναι σχεδόν το ίδιο δύσκολο να ταξινομηθεί όσο και ο αρχικός πίνακας. Αντίθετα, αν ο μεγαλύτερος από τους δύο υποπίνακες είναι σημαντικά μικρότερος από τον αρχικό πίνακα (π.χ. περιέχει περίπου τα μισά στοιχεία), η διαδικασία της διαίρεσης «έπιασε τόπο» αφού η ταξινόμηση των δύο υποπινάκων χωριστά είναι σημαντικά ευκολότερη από την ταξινόμηση ολόκληρου του αρχικού πίνακα.

Με βάση τις παραπάνω παρατηρήσεις, η χειρότερη περίπτωση για την ταχεία ταξινόμηση συμβαίνει όταν η συνάρτηση Partition παράγει σε κάθε της εφαρμογή έναν υποπίνακα μεγέθους $r - p$ ($= n - 1$), και έναν υποπίνακα μεγέθους 1. Σε αυτή την περίπτωση, η αναδρομική εξίσωση που περιγράφει το χρόνο εκτέλεσης γίνεται $T(n) = T(n - 1) + \Theta(n)$. Η λύση αυτής της εξίσωσης είναι $T(n) = \Theta(n^2)$ (ουσιαστικά πρόκειται για το άθροισμα των n πρώτων φυσικών αριθμών).

Η καλύτερη περίπτωση συμβαίνει όταν η συνάρτηση Partition παράγει σε κάθε της εφαρμογή δύο υποπίνακες μεγέθους $(r - p + 1) / 2$ ($= n / 2$). Τότε, ο χρόνος εκτέλεσης του αλγόριθμου ταχείας ταξινόμησης δίνεται από την αναδρομική εξίσωση $T(n) = 2\,T(n / 2) + \Theta(n)$, η οποία έχει λύση $T(n) = \Theta(n \log n)$. Δηλαδή, ο χρόνος εκτέλεσης της ταχείας ταξινόμησης είναι $O(n^2)$ και $\Omega(n \log n)$.

Άσκηση 1.19. Να δώσετε στιγμιότυπα για τα οποία η ταχεία ταξινόμηση χρειάζεται χρόνο $\Theta(n \log n)$ και $\Theta(n^2)$.

Λύση. Είδαμε ότι ένα στιγμιότυπο εισόδου για την ταχεία ταξινόμηση χρειάζεται χρόνο $\Theta(n \log n)$ όταν η συνάρτηση Partition παράγει δύο ισομεγέθεις υποπίνακες. Ας θεωρήσουμε τον πίνακα [5, 1, 2, 4, 7, 3, 6, 8]. Αρχικά, η διαίρεση παράγει τους υποπίνακες [3, 1, 2, 4] και [7, 5, 6, 8]. Ο υποπίνακας [3, 1, 2, 4] διαιρείται περαιτέρω στους υποπίνακες [2, 1] και [3, 4], ενώ ο υποπίνακας [7, 5, 6, 8] διαιρείται περαιτέρω στους υποπίνακες [6, 5] και [7, 8]. Επομένως, ο πίνακας [5, 1, 2, 4, 7, 3, 6, 8] έχει τα χαρακτηριστικά των στιγμιότυπων που ταξινομούνται σε χρόνο $\Theta(n \log n)$ από τον αλγόριθμο ταχείας ταξινόμησης. Ένα άλλο παράδειγμα πίνακα με ανάλογα χαρακτηριστικά είναι ο [9, 1, 2, 4, 7, 3, 6, 8, 13, 5, 10, 12, 15, 11, 14, 16].

Από την άλλη πλευρά, ένα στιγμιότυπο εισόδου χρειάζεται χρόνο $\Theta(n^2)$ όταν η διαδικασίας της διαίρεσης παράγει έναν υποπίνακα μεγέθους 1 και έναν υποπίνακα μεγέθους $n - 1$. Τέτοια παραδείγματα είναι κάθε πίνακας του οποίου τα στοιχεία είναι ταξινομημένα σε αύξουσα ή φθίνουσα σειρά.

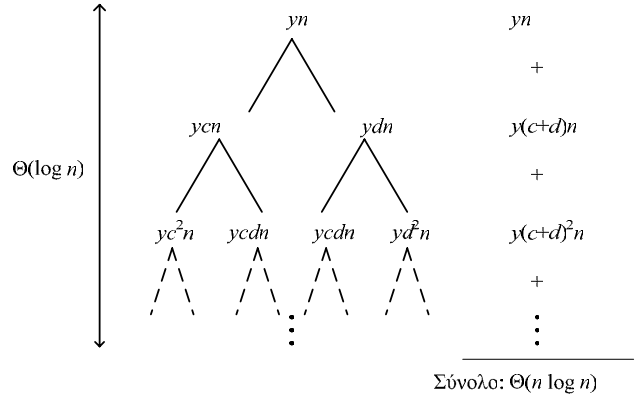
1.6.1. Χρήση Τυχαιότητας και Ανάλυση Μέσης Τιμής

Κατά την ανάλυση της ταχείας ταξινόμησης, είναι πολύ περιοριστικό να υποθέσουμε ότι η συνάρτηση Partition παράγει σε κάθε της εφαρμογή δύο σχεδόν ισομεγέθεις υποπίνακες. Αντ' αυτού, ας υποθέσουμε ότι οι δύο υποπίνακες που παράγονται σε κάθε εφαρμογή της Partition έχουν την ίδια τάξη μεγέθους $\Theta(r - p)$[4]. Και σε αυτή την περίπτωση, μπορούμε να αποδείξουμε ότι ο χρόνος εκτέλεσης της ταχείας ταξινόμησης είναι $T(n) = \Theta(n \log n)$. Πράγματι, οι δύο υποπίνακες έχουν την ίδια τάξη μεγέθους σε κάθε αναδρομική κλήση, το ύψος του δέντρου της αναδρομής είναι $\Theta(\log n)$, ενώ κάθε επίπεδο του δέντρου συνεισφέρει $\Theta(n)$ στο συνολικό χρόνο εκτέλεσης. Το ίδιο επιχείρημα ισχύει και όταν για όλα, εκτός από $O(\log n)$ βήματα, η Partition παράγει υποπίνακες της ίδιας τάξης μεγέθους.

Άσκηση 1.20. Έστω ότι κάθε εφαρμογή της Partition σε πίνακα με k στοιχεία παράγεται ένας υποπίνακας μεγέθους ck και ένας υποπίνακας μεγέθους dk, όπου c, d θετικές σταθερές, $c + d = 1$. Να αποδείξτε σε αυτή την περίπτωση ότι ο χρόνος για την ταχεία ταξινόμηση n στοιχείων είναι $\Theta(n \log n)$.

[4] Θεωρούμε ότι η Partition καλείται με όρισμα τον πίνακα $A[p \ldots r]$.

Λύση. Έστω $T(n)$ ο χρόνος εκτέλεσης της ταχείας ταξινόμησης. Από υπόθεση, το $T(n)$ δίνεται από την αναδρομική εξίσωση $T(n) = T(cn) + T(dn) + \Theta(n)$. Έστω $y \geq 1$ μία σταθερά τέτοια ώστε ο χρόνος εκτέλεσης της Partition (όταν εφαρμόζεται σε n στοιχεία) να μην ξεπερνά το yn. Παρατηρώντας το δέντρο της αναδρομής (Σχήμα 1.7), βλέπουμε ότι η συνεισφορά του επιπέδου i (η ρίζα βρίσκεται στο επίπεδο 0) είναι $y(c+d)^i n = yn$. Επιπλέον, το ύψος του δέντρου είναι μεταξύ $\log_c n$ και $\log_d n$, είναι δηλαδή $\Theta(\log n)$. Επομένως, εφόσον $c + d = 1$, $T(n) = \Theta(y\, n \log n) = \Theta(n \log n)$, αφού το y είναι σταθερά.



Σχήμα 1.7. Το δέντρο της αναδρομής για την εξίσωση $T(n) = T(cn) + T(dn) + yn$.

Ο λόγος που η ταχεία ταξινόμηση, αν και έχει χρόνο εκτέλεσης χειρότερης περίπτωσης $\Theta(n^2)$, είναι πολύ γρήγορη στην πράξη, είναι ότι στις περισσότερες περιπτώσεις η συνάρτηση Partition παράγει δύο υποπίνακες της ίδιας τάξης μεγέθους.

Για να τεκμηριώσουμε και θεωρητικά την ταχύτητα του αλγόριθμου στην πράξη, θα τροποποιήσουμε τη συνάρτηση Partition, ώστε να επιλέγει με τυχαίο τρόπο το στοιχείο γύρω από το οποίο οργανώνεται η διαίρεση σε υποπίνακες, και θα υπολογίσουμε το χρόνο εκτέλεσης μέσης περίπτωσης για αυτήν την παραλλαγή. Συγκεκριμένα, στον ψευδοκώδικα της Quicksort αντικαθιστούμε την κλήση της Partition με κλήση της ακόλουθης παραλλαγής της.

```
Random-Partition(A[p...r])
Επέλεξε έναν ακέραιο i τυχαία στο διάστημα [p, r), ώστε
κάθε τιμή να έχει πιθανότητα 1/(r – p) να επιλεγεί.
swap(A[i], A[p]);
return partition(A[p...r]);
```

Σημείωση. Οι αλγόριθμοι που δεν χρησιμοποιούν τυχαιότητα στις επιλογές τους ονομάζονται *ντετερμινιστικοί αλγόριθμοι* (deterministic algorithms), ενώ οι αλγόριθμοι, όπως η παραπάνω παραλλαγή της Partition, που χρησιμοποιούν τυχαιότητα σε κάποιες από τις επιλογές τους ονομάζονται *πιθανοτικοί αλγόριθμοι* (randomized algorithms). Οι πιθανοτικοί αλγόριθμοι είναι συχνά ευκολότεροι στο σχεδιασμό και αποτελεσματικότεροι από τους ντετερμινιστικούς. Επιπλέον, υπάρχουν προβλήματα για τα οποία δεν είναι γνωστοί ντετερμινιστικοί αλγόριθμοι των οποίων η απόδοση να είναι παρόμοια με αυτή των καλύτερων πιθανοτικών. Από την άλλη πλευρά, η ανάλυση των πιθανοτικών αλγορίθμων είναι συχνά δυσκολότερη τεχνικά από αυτή των ντετερμινιστικών αλγορίθμων. ■

Συχνά οι τυχαίες επιλογές ενός πιθανοτικού αλγόριθμου επηρεάζουν σημαντικά το χρόνο εκτέλεσης δημιουργώντας διαφορετικά ενδεχόμενα εκτέλεσης του αλγόριθμου, καθένα από τα οποία εκτελείται σε διαφορετικό χρόνο (π.χ. ο πραγματικός χρόνος εκτέλεσης της πιθανοτικής εκδοχής της ταχείας ταξινόμησης εξαρτάται από τα στοιχεία γύρω από τα οποία θα οργανωθεί η διαίρεση, και τα οποία επιλέγονται τυχαία από την Random-Partition). Σε αυτές τις περιπτώσεις, προσπαθούμε να προσδιορίσουμε τη *μέση τιμή* του χρόνου εκτέλεσης του αλγόριθμου που δίνεται από το άθροισμα, για όλα τα διαφορετικά ενδεχόμενα, των γινομένων του χρόνου εκτέλεσης για κάθε ενδεχόμενο επί την πιθανότητα να προκύψει αυτό το ενδεχόμενο (και επομένως ο αλγόριθμος να έχει αυτόν το χρόνο εκτέλεσης).

Επιστρέφουμε στην ανάλυση της πιθανοτικής εκδοχής της ταχείας ταξινόμησης. Έστω $n = r - p + 1$ ο αριθμός των στοιχείων του πίνακα $A[p \ldots r]$. Στη Random-Partition, κάθε στοιχείο $A[i]$ ($p \le i < r$) μπορεί να επιλεγεί με πιθανότητα $1/(n-1)$ σαν το στοιχείο x γύρω από το οποίο θα οργανωθεί η διαίρεση. Σε κάθε δυνατό όριο q ($q = p, \ldots, r-1$) για τη διαίρεση του πίνακα $A[p \ldots r]$ αντιστοιχεί σε ένα στοιχείο $A[i_q]$: Οι υποπίνακες που προκύπτουν είναι $A[p \ldots q]$ και $A[q+1 \ldots r]$ αν και μόνο αν επιλεγεί το στοιχείο $A[i_q]$.

Έστω $S(n)$ ο μέσος χρόνος που χρειάζεται η πιθανοτική εκδοχή για την ταχεία ταξινόμηση n στοιχείων. Το $S(n)$ είναι ίσο με το άθροισμα του $\Theta(n)$, για την εκτέλεση της Random-Partition, και των μέσων χρόνων για την ταξινόμηση των δύο υποπινάκων που προκύπτουν. Αφού κάθε διαφορετικό όριο διαίρεσης q, $q = p, \ldots, r-1$, προκύπτει με πιθανότητα $1/(n-1)$, έχουμε:

$$S(n) = \Theta(n) + \frac{1}{n-1}\sum_{i=1}^{n-1}\big(S(i) + S(n-i)\big) = \Theta(n) + \frac{2}{n-1}\sum_{i=1}^{n-1} S(i) \ .$$

Σαν αρχική συνθήκη μπορούμε να θεωρήσουμε $S(1) = \Theta(1)$.

Προφανώς, $S(n) = \Omega(n \log n)$ και $S(n) = O(n^2)$, αφού ο μέσος χρόνος εκτέλεσης πρέπει να βρίσκεται μεταξύ της καλύτερης και της χειρότερης περίπτωσης. Η λύση της παραπάνω αναδρομικής εξίσωσης είναι $S(n) = \Theta(n \log n)$.

Επομένως, όταν το στοιχείο γύρω από το οποίο οργανώνεται η διαίρεση επιλέγεται με τυχαίο και ομοιόμορφο τρόπο, ο μέσος χρόνος εκτέλεσης της ταχείας ταξινόμησης είναι $\Theta(n \log n)$. Με προσεκτική ανάλυση, μπορεί να αποδειχθεί ότι, για τη διάταξη ενός πίνακα n στοιχείων, αυτή η πιθανοτική εκδοχή του αλγόριθμου ταχείας ταξινόμησης χρειάζεται κατά μέσον όρο το πολύ $2\, n \ln n + O(n)$ συγκρίσεις μεταξύ των στοιχείων του πίνακα.

Εκτός από την ταχύτητα, ένα άλλο σημαντικό πλεονέκτημα της ταχείας είναι ότι μπορεί να υλοποιηθεί ώστε να εκτελείται εντός του δοθέντος χώρου. Στην πράξη, η ταχεία ταξινόμηση είναι ίσως ο ταχύτερος αλγόριθμος για την ταξινόμηση μεγάλου αριθμού στοιχείων που χωρούν στην κύρια μνήμη.

Άσκηση 1.21. Σε πρακτικές εφαρμογές, μπορούμε να βελτιώσουμε το χρόνο εκτέλεσης της ταχείας ταξινόμησης εκμεταλλευόμενοι την ταχύτητα της ταξινόμησης με εισαγωγή όταν ο πίνακας εισόδου είναι «σχεδόν διατεταγμένος». Ένας τρόπος να συνδυάσουμε τους δύο αλγόριθμους είναι όταν η ταχεία ταξινόμηση φτάνει σε πίνακα με λιγότερα από k στοιχεία, να επιστρέφει χωρίς να εκτελεί καμία ενέργεια. Μετά την ολοκλήρωσης της ταχείας ταξινόμησης, ολοκληρώνουμε τη διαδικασία ταξινόμησης εκτελώντας ταξινόμηση με εισαγωγή σε ολόκληρο τον πίνακα. Υποθέστε ότι χρησιμοποιείται η πιθανοτική εκδοχή της ταχείας ταξινόμησης, και αποδείξτε ότι ο παραπάνω αλγόριθμος

έχει χρόνο εκτέλεσης μέσης περίπτωσης $O(nk + n \log(n / k))$. Με ποια κριτήρια πρέπει να επιλέγεται η παράμετρος k στη θεωρία και την πράξη;

Λύση. Ο χρόνος εκτέλεσης του παραπάνω αλγορίθμου έχει δύο συνιστώσες: Τον χρόνο εκτέλεσης μέσης περίπτωσης για την πιθανοτική εκδοχή της ταχείας ταξινόμησης, η οποία εκτελείται σε πίνακες με τουλάχιστον k στοιχεία, και το χρόνο εκτέλεσης της ταξινόμησης με εισαγωγή.

Αφού η ταχεία ταξινόμηση σταματά να εκτελείται όταν ο πίνακας εισόδου έχει k ή λιγότερα στοιχεία, από τα $\log n$ επίπεδα της αναδρομής αφαιρούνται τα τελευταία $\log k$. Επομένως, ο αναμενόμενος αριθμός επιπέδων της αναδρομής είναι $\log n - \log k = \log(n/k)$. Αφού κάθε επίπεδο συνεισφέρει $\Theta(n)$ χρόνο, ο μέσος χρόνος εκτέλεσης της συγκεκριμένης παραλλαγής της ταχείας ταξινόμησης είναι $O(n \log(n / k))$. Ένας άλλος τρόπος απόδειξης του ίδιου αποτελέσματος είναι να δείξουμε ότι η αναδρομική εξίσωση

$$S(n) = \Theta(n) + \frac{1}{n-1}\left(\sum_{i=k}^{n-1} S(i) + \sum_{i=1}^{n-k-1} S(n-i)\right)$$

έχει λύση $O(n \log(n / k))$, όπου $S(i) = \Theta(1)$, για $i < k$.

Με την ολοκλήρωση της ταχείας ταξινόμησης, κάθε στοιχείο βρίσκεται το πολύ k θέσεις μακριά από την τελική του θέση στον ταξινομημένο πίνακα. Επομένως, η ταξινόμηση με εισαγωγή χρειάζεται χρόνο $O(kn)$, αφού κάθε στοιχείο θα συγκριθεί και θα μετακινηθεί το πολύ k φορές μέχρι να φτάσει την τελική του θέση.

Άρα, ο χρόνος εκτέλεσης για τον παραπάνω αλγόριθμο είναι $O(kn + n \log(n / k))$.

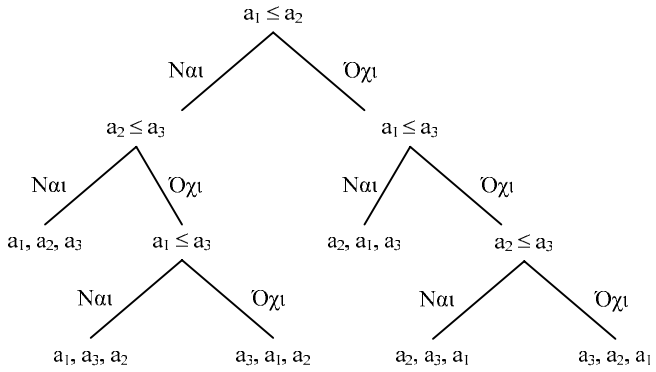
Έστω c_1 και c_2 οι «κρυμμένες» πολλαπλασιαστικές σταθερές σε αυτή την έκφραση, δηλαδή ο χρόνος εκτέλεσης είναι $c_1 nk + c_2 n \log(n / k)$. Στη θεωρία, η τιμή του k θα επιλεγεί ώστε να ελαχιστοποιηθεί η τιμή της παραπάνω έκφρασης. Στην πράξη, η τιμή του k θα επιλεγεί ώστε ο μέσος χρόνος εκτέλεσης της ταχείας ταξινόμησης να είναι μικρότερος από το χρόνο εκτέλεσης της ταξινόμησης με εισαγωγή σε έναν πίνακα με τουλάχιστον k στοιχεία.

Άσκηση 1.22. Δίνεται ένας πίνακας Α με n διαφορετικούς αριθμούς και αριθμός x. Να διατυπώσετε έναν αλγόριθμο με χρόνο εκτέλεσης χειρότερης περίπτωσης $O(n \log n)$ που να αποφασίζει αν υπάρχει ένα ζεύγος στοιχείων στον πίνακα Α που να έχει άθροισμα x.

1.7. Κάτω Φράγμα στον Αριθμό των Συγκρίσεων

Θα ολοκληρώσουμε την παρουσίαση των αλγόριθμων ταξινόμησης παρουσιάζοντας ένα κάτω φράγμα στο χρόνο εκτέλεσης χειρότερης περίπτωσης για κάθε συγκριτικό αλγόριθμο που δεν χρησιμοποιεί τυχαιότητα. Συγκεκριμένα, θα δείξουμε ότι για κάθε τέτοιο αλγόριθμο, υπάρχει ένα στιγμιότυπο εισόδου, για το οποίο ο αλγόριθμος χρειάζεται $\Omega(n \log n)$ συγκρίσεις. Επομένως, ο χρόνος εκτέλεσης χειρότερης περίπτωσης για κάθε ντετερμινιστικό συγκριτικό αλγόριθμο είναι $\Omega(n \log n)$.

Κάθε ντετερμινιστικός συγκριτικός αλγόριθμος ταξινόμησης μπορεί να αναπαρασταθεί με το *δέντρο των συγκρίσεων* (comparison tree, ή δέντρο αποφάσεων – decision tree) τις οποίες εκτελεί. Στο Σχήμα 1.8 φαίνεται το δέντρο συγκρίσεων του αλγόριθμου ταξινόμησης με εισαγωγή για 3 αριθμούς.



Σχήμα 1.8. Το δέντρο συγκρίσεων της ταξινόμησης με εισαγωγή για 3 αριθμούς.

Το δέντρο συγκρίσεων για την ταξινόμηση n αριθμών πρέπει να έχει τουλάχιστον $n!$ φύλλα / αποτελέσματα, αφού κάθε διαφορετική αναδιάταξη των αριθμών εισόδου αποτελεί πιθανό αποτέλεσμα του αλγορίθμου και ο αλγόριθμος πρέπει να μπορεί να καταλήξει σε αυτή. Το ύψος του δέντρου των συγκρίσεων είναι ίσο με τον αριθμό των συγκρίσεων που εκτελεί ο αλγόριθμος στη χειρότερη περίπτωση.

Αφού ο αλγόριθμος, ανάλογα με το αποτέλεσμα μιας σύγκρισης, μπορεί να διαλέξει το πολύ ανάμεσα σε δύο ενδεχόμενα / μονοπάτια του δέντρου, το δέντρο των συγκρίσεων είναι ένα δυαδικό δέντρο. Είναι γνωστό ότι κάθε δυαδικό δέντρο με ύψος h

έχει το πολύ 2^h φύλλα. Επομένως, αν h είναι το ύψος του δέντρου των συγκρίσεων, πρέπει να είναι $2^h \geq n!$. Λογαριθμώντας αυτή την ανισότητα, έχουμε:

$$\begin{aligned} h &\geq \log(n!) \\ &= \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=n/2}^{n} \log i \\ &\geq \sum_{i=n/2}^{n} \log(n/2) \\ &= \Omega(n \log n) \ . \end{aligned}$$

Επομένως, για κάθε ντετερμινιστικό συγκριτικό αλγόριθμο, υπάρχει τουλάχιστον ένα στιγμιότυπο εισόδου για το οποίο ο αλγόριθμος χρειάζεται $\Omega(n \log n)$ συγκρίσεις. Συνεπώς, ο χρόνος εκτέλεσης χειρότερης περίπτωσης κάθε ντετερμινιστικού συγκριτικού αλγορίθμου είναι $\Omega(n \log n)$.

Άσκηση 1.22. Θεωρείστε ότι σε ένα πρόβλημα ταξινόμησης τα στοιχεία είναι χωρισμένα σε n / k ομάδες που καθεμία περιέχει k στοιχεία. Τα στοιχεία κάθε συγκεκριμένης ομάδας είναι μεγαλύτερα από αυτά της προηγούμενης, και είναι μικρότερα από αυτά της επόμενης. Επομένως, για να ταξινομήσουμε το σύνολο των στοιχείων, αρκεί να ταξινομήσουμε τα στοιχεία κάθε ομάδας. Αποδείξτε ένα κάτω φράγμα στον αριθμό των συγκρίσεων που χρειάζεται κάθε ντετερμινιστικός συγκριτικός αλγόριθμος για αυτό το πρόβλημα.

Σπαζοκεφαλιά. Έχετε 12 χρυσά νομίσματα από τα οποία *ένα* είναι κάλπικο. Το κάλπικο νόμισμα έχει διαφορετικό βάρος από τα γνήσια, αλλά δεν γνωρίζετε αν είναι βαρύτερο ή ελαφρύτερο. Έχετε στη διάθεσή σας *τρεις μόνο ζυγίσεις* σε μια ζυγαριά ακριβείας στην οποία τοποθετείτε μερικά νομίσματα στα δεξιό μέρος και μερικά νομίσματα στο αριστερό μέρος και μαθαίνετε αν τα νομίσματα στα δεξιά είναι ελαφρύτερα, βαρύτερα ή ίδιου βάρους με τα νομίσματα στα αριστερά. Θεωρείστε ότι η ζυγαριά σας είναι ικανή να διακρίνει τη διαφορά βάρους ανάμεσα σε ένα κάλπικο και ένα γνήσιο νόμισμα. Να βρείτε έναν αλγόριθμο που χρησιμοποιώντας *μόνο τρεις ζυγίσεις* διακρίνει το κάλπικο νόμισμα και αν αυτό είναι βαρύτερο ή ελαφρύτερο από τα γνήσια.

2. Αλγόριθμοι Αναζήτησης

Για το πρόβλημα της αναζήτησης, γνωρίζουμε τον αλγόριθμο της *γραμμικής αναζήτησης* (linear search), ο οποίο λύνει το πρόβλημα σε χρόνο $O(n)$ ακόμη και όταν η ακολουθία δεν είναι ταξινομημένη, και τον αλγόριθμο της *δυαδικής αναζήτησης* (binary search), ο οποίος εφαρμόζεται σε ταξινομημένες ακολουθίες, λύνει το πρόβλημα σε χρόνο $O(\log n)$ και είναι βέλτιστος (ως προς το χρόνο εκτέλεσης *χειρότερης* περίπτωσης). Αφού επαναλάβουμε αυτούς τους αλγόριθμους, θα παρουσιάσουμε τον αλγόριθμο *αναζήτησης με παρεμβολή* (interpolation search) που επίσης εφαρμόζεται σε ταξινομημένες ακολουθίες και λύνει το πρόβλημα σε χρόνο $O(\text{loglog } n)$ κατά μέσον όρο (χρόνος μέσης περίπτωσης) όταν οι αριθμοί έχουν επιλεγεί ανεξάρτητα και με ομοιόμορφα τυχαίο τρόπο σε ένα δεδομένο διάστημα (ή γενικότερα όταν ο αλγόριθμος γνωρίζει την κατανομή πιθανότητας με βάση την οποία έχουν επιλεγεί οι αριθμοί της ταξινομημένης ακολουθίας).

2.1.1. Γραμμική Αναζήτηση

Η γραμμική αναζήτηση επιτρέπει την αναζήτηση στοιχείων σε έναν πίνακα ακόμη και αν αυτός δεν είναι ταξινομημένος. Ο πίνακας Α διατρέχεται από αριστερά προς τα δεξιά και τα στοιχεία του συγκρίνονται με το ζητούμενο στοιχείο. Η αναζήτηση ολοκληρώνεται όταν βρεθεί η πρώτη θέση στην οποία υπάρχει το ζητούμενο στοιχείο ή όταν εξαντληθούν τα στοιχεία του πίνακα. Στην πρώτη περίπτωση, επιστρέφεται η θέση στην οποία βρέθηκε το στοιχείο, ενώ στη δεύτερη περίπτωση επιστρέφεται 0 σαν ένδειξη ότι το στοιχείο δεν υπάρχει στον πίνακα Α.

```
Linear-Search(A[1...n], k)
    for i ← 1 to n do
        if A[i] = k then return(i);
    return(0);
```

Η ορθότητα του αλγόριθμου είναι προφανής. Η χρονική πολυπλοκότητα μιας αποτυχημένης αναζήτησης είναι πάντα $\Theta(n)$ αφού το ζητούμενο στοιχείο συγκρίνεται με όλα τα στοιχεία του πίνακα Α πριν ο αλγόριθμος επιστρέψει 0 σε ένδειξη αποτυχημένης αναζήτησης. Η χρονική πολυπλοκότητα μιας επιτυχημένης αναζήτησης είναι $\Theta(1)$ στην καλύτερη περίπτωση (το στοιχείο βρίσκεται στην πρώτη θέση) και $\Theta(n)$ στη χειρότερη περίπτωση (το στοιχείο βρίσκεται στην τελευταία θέση).

Για να προσδιορίσουμε τη χρονική πολυπλοκότητα μιας επιτυχημένης αναζήτησης στη μέση περίπτωση, υποθέτουμε ότι το ζητούμενο στοιχείο k επιλέγεται ισοπίθανα από τα στοιχεία του πίνακα Α. Δηλαδή, το ζητούμενο στοιχείο k έχει πιθανότητα $1/n$ να βρίσκεται σε καθεμία από τις θέσεις $i = 1, \ldots, n$.

Συνεπώς, η πιθανότητα το for-loop να εκτελεστεί ακριβώς i φορές, $i = 1, \ldots, n$, είναι ίση με $1/n$ και ο μέσος αριθμός επαναλήψεων του for-loop δίνεται από το άθροισμα $\sum_{i=1}^{n} \frac{i}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{n+1}{2}$. Υποθέτοντας λοιπόν ότι το ζητούμενο στοιχείο επιλέγεται ισοπίθανα από τα στοιχεία του πίνακα Α, ο μέσος χρόνος για μια επιτυχημένη γραμμική αναζήτηση είναι επίσης γραμμικός στο μέγεθος του Α.

Αν υποθέσουμε όμως ότι κάποια στοιχεία έχουν μεγαλύτερη πιθανότητα να αναζητηθούν από τα υπόλοιπα (π.χ. οι πιο «δημοφιλείς» εγγραφές σε μία τηλεφωνική ατζέντα), ο χρόνος μέσης περίπτωσης της γραμμικής αναζήτησης μπορεί να βελτιωθεί μετακινώντας τα στοιχεία που αναζητούνται με μεγαλύτερη συχνότητα στην αρχή.

Για παράδειγμα, αν στον πίνακα [1, 4, 3, 2] το στοιχείο 1 ζητείται με πιθανότητα 0.1, το στοιχείο 2 με πιθανότητα 0.5, το στοιχείο 3 με πιθανότητα 0.3, και το στοιχείο 4 με πιθανότητα 0.1, ο μέσος αριθμός συγκρίσεων της γραμμικής αναζήτησης είναι

$$0.1\times1 + 0.1\times2 + 0.3\times3 + 0.5\times4 = 3.2$$

πολύ κοντά στη χειρότερη περίπτωση όπου οι συγκρίσεις είναι 4. Αν όμως αναδιοργανώσουμε τον πίνακα με βάση τις συχνότητες αναζήτησης τοποθετώντας τα πιο «δημοφιλή» στοιχεία στην αρχή, οπότε έχουμε [2, 3, 1, 4], ο μέσος αριθμός συγκρίσεων γίνεται

$$0.5\times1 + 0.3\times2 + 0.1\times3 + 0.1\times4 = 1.8$$

Στις περισσότερες πρακτικές εφαρμογές, οι συχνότητες αναζήτησης των στοιχείων δεν είναι γνωστές εκ των προτέρων. Μια λύση είναι να ενημερώνουμε τις συχνότητες αναζήτησης σε κάθε νέα αναζήτηση και να αναδιοργανώνουμε τον πίνακα με βάση τις νέες συχνότητες. Αυτή η λύση είναι χρονοβόρα, απαιτεί σημαντικά περισσότερο χώρο αποθήκευσης, και δεν ακολουθείται στην πράξη.

Μια άλλη ιδέα είναι αυτή της σταδιακής αναδιοργάνωσης του πίνακα με βάση την τελευταία αναζήτηση. Για παράδειγμα, κάθε φορά που αναζητούμε ένα στοιχείο,

μπορούμε να το μετακινούμε στην αρχή του πίνακα (move-to-front) ή μια θέση στα αριστερά (move-one-forward). Ο κανόνας μετακίνησης μίας θέσης αριστερά μπορεί να υλοποιηθεί ευκολότερα σε πίνακες (χρειάζεται απλά μία αντιμετάθεση στοιχείων σε κάθε αναζήτηση). Όμως υπάρχουν περιπτώσεις που δεν βελτιώνει καθόλου το μέσο χρόνο αναζήτησης (π.χ. στον πίνακα [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], αναζητούμε συνεχώς πρώτα το 10 και μετά το 9). Ο κανόνας μετακίνησης στην αρχή χρειάζεται περισσότερες αντιμεταθέσεις (όταν πρόκειται για πίνακες, μπορεί όμως να υλοποιηθεί εύκολα σε διασυνδεδεμένες λίστες) αλλά εγγυάται σημαντική βελτίωση στο μέσο χρόνο αναζήτησης. Συγκεκριμένα μπορεί να αποδειχθεί ότι αν θεωρήσουμε μια ακολουθία αναζητήσεων, ο κανόνας μετακίνησης στην αρχή χρειάζεται συνολικό χρόνο το πολύ διπλάσιο από το χρόνο της βέλτιστης οργάνωσης (η οποία όμως προϋποθέτει ότι οι συχνότητες αναζήτησης είναι γνωστές εκ των προτέρων).

Πρέπει να σημειώσουμε ότι η γραμμική αναζήτηση προσπελαύνει τα στοιχεία σειριακά (το ένα μετά το άλλο) και μπορεί να χρησιμοποιηθεί όταν τα στοιχεία είναι αποθηκευμένα στο σκληρό δίσκο (ή γενικά σε συσκευές που δεν επιτρέπουν τυχαία προσπέλαση των στοιχείων) ή οργανωμένα σε διασυνδεδεμένη λίστα. Αντίθετα, η δυαδική αναζήτηση και η αναζήτηση με παρεμβολή μπορούν να εφαρμοστούν μόνο όταν τα στοιχεία είναι ταξινομημένα και αποθηκευμένα σε μνήμη τυχαίας προσπέλασης (ή οργανωμένα σε πίνακα).

2.1.2. Δυαδική Αναζήτηση

Η δυαδική αναζήτηση χρησιμοποιείται ευρύτατα για την αναζήτηση στοιχείων σε ταξινομημένους πίνακες. Η ιδέα της δυαδικής αναζήτησης προκύπτει φυσιολογικά από την εφαρμογή της μεθόδου «διαίρει και βασίλευε» στο πρόβλημα της αναζήτησης. Το μεσαίο στοιχείο του πίνακα συγκρίνεται με το ζητούμενο στοιχείο. Σε περίπτωση ισότητας, η αναζήτηση τερματίζεται επιτυχώς και επιστρέφεται η θέση του στοιχείου στον πίνακα Α. Αν το ζητούμενο στοιχείο είναι μεγαλύτερο από το μεσαίο στοιχείο του Α, το ζητούμενο στοιχείο μπορεί να βρίσκεται μόνο στο δεξιό μισό του Α, εκεί δηλαδή που βρίσκονται όλα τα στοιχεία που είναι μεγαλύτερα από το μεσαίο. Σε αυτή την περίπτωση, ο αλγόριθμος εφαρμόζεται αναδρομικά στον υποπίνακα που αντιστοιχεί στο δεξιό μισό του πίνακα Α. Ομοίως, αν το ζητούμενο στοιχείο είναι μικρότερο από το μεσαίο στοιχείο, ο αλγόριθμος εφαρμόζεται αναδρομικά στο αριστερό μισό του Α.

Ο παρακάτω ψευδοκώδικας αποτελεί μία επαναληπτική υλοποίηση της δυαδικής αναζήτησης. Είναι εύκολο να δώσετε μία αναδρομική υλοποίηση του αλγόριθμου.

```
Binary-Search(A[1...n], k)
    low ← 1; up ← n;
    while low ≤ up do
        mid ← (low + up) / 2;
        if A[mid] = k then return(mid);
        else  if A[mid] < k then low ← mid + 1;
              else up ← mid − 1;
    return(0);
```

Η ορθότητα του αλγόριθμου προκύπτει από την ακόλουθη παρατήρηση: Έστω ότι έχουμε καταλήξει πως το k πρέπει να βρίσκεται στον υποπίνακα A[low…up]. Αν είναι A[mid] < k, το k μπορεί να βρίσκεται μόνο στο δεξιό μισό του πίνακα, δηλαδή στον υποπίνακα A[mid+1…up]. Αυτό συμβαίνει γιατί ο A είναι ταξινομημένος και όλα τα στοιχεία που είναι μεγαλύτερα από το A[mid] βρίσκονται στο A[mid+1…up]. Για τους ίδιους λόγους, αν είναι A[mid] > k, το k μπορεί να βρίσκεται μόνο στο αριστερό μισό του πίνακα, δηλαδή στον υποπίνακα A[low…mid–1] που περιέχει όλα τα στοιχεία που είναι μικρότερα από το A[mid]. Εφαρμόζοντας αυτή την ιδέα επαγωγικά είτε θα βρούμε το στοιχείο όταν A[mid] = k είτε θα καταλήξουμε στον κενό πίνακα (θα γίνει low > up), οπότε και συμπεραίνουμε ότι το k δεν υπάρχει στον πίνακα A.

Για το χρόνο εκτέλεσης, σε κάθε εκτέλεση του while-loop, το μέγεθος του υποπίνακα στον οποίο έχει περιοριστεί η αναζήτηση μειώνεται στο μισό. Στο πρώτο while-loop η αναζήτηση εκτείνεται σε ολόκληρο τον πίνακα A (n στοιχεία) και στο τελευταίο η αναζήτηση έχει περιοριστεί σε ένα τουλάχιστον στοιχείο. Συνεπώς, το while-loop εκτελείται το πολύ $\lfloor \log n \rfloor + 1$ φορές. Κάθε εκτέλεση του while-loop χρειάζεται σταθερό χρόνο, όποτε ο χρόνος εκτέλεσης της δυαδικής αναζήτησης είναι O(log n). Στη χειρότερη περίπτωση ή σε περίπτωση αποτυχημένης αναζήτησης ο χρόνος εκτέλεσης της δυαδικής αναζήτησης είναι Θ(log n).

Εναλλακτική ανάλυση του χρόνου εκτέλεσης. Έστω $T(n)$ ο χρόνος εκτέλεσης χειρότερης περίπτωσης για δυαδική αναζήτηση σε έναν πίνακα n στοιχείων. Το $T(n)$ είναι ίσο με το άθροισμα του χρόνου για τον υπολογισμό του mid, τη σύγκριση του A[mid] με το k, και την αναδρομική εφαρμογή της δυαδικής αναζήτησης είτε στον αριστερό είτε στο δεξιό υποπίνακα. Ο χρόνος για τον υπολογισμό του mid και τη σύγκριση των A[mid]

και k είναι σταθερός, ενώ ο χρόνος για την αναδρομική εφαρμογή είναι $T(n/2)$ αφού το μέγεθος του υποπίνακα στο οποίο εφαρμόζεται η δυαδική αναζήτηση μειώνεται στο μισό σε κάθε εκτέλεση του while-loop. Δηλαδή, $T(n) = T(n/2) + \Theta(1)$. Σαν αρχική συνθήκη μπορούμε να χρησιμοποιήσουμε $T(1) = \Theta(1)$. Η λύση αυτής της αναδρομικής εξίσωσης είναι $T(n) = \Theta(\log n)$.

Ένας διαφορετικός τρόπος να σκεφθούμε (και να υλοποιήσουμε) τη διαδικασία της δυαδικής αναζήτησης είναι να θεωρήσουμε τη δυαδική αναπαράσταση των αριθμών που αντιστοιχούν στις θέσεις του πίνακα A. Αφού ο A έχει n στοιχεία, η θέση κάθε στοιχείου προσδιορίζεται από $\lceil \log n \rceil$ δυαδικά ψηφία. Κάθε σύγκριση του ζητούμενου στοιχείου k με το μεσαίο στοιχείο του υποπίνακα A[low…up] προσδιορίζει το επόμενο (σε σειρά σημαντικότητας) bit (δυαδικό ψηφίο) της θέσης που πρέπει να καταλαμβάνει το k στον πίνακα A. Η πρώτη σύγκριση καθορίζει το πρώτο (σημαντικότερο) bit. Έστω $mid = 01^{\lceil \log n \rceil - 1}$. Αν $k \le$ A[mid], το k βρίσκεται στο αριστερό μισό του πίνακα A και το πρώτο bit της θέσης του είναι 0. Διαφορετικά το k βρίσκεται στο δεξιό μισό του A και το πρώτο bit της θέσης του είναι 1. Ομοίως, κάθε σύγκριση καθορίζει το επόμενο πιο σημαντικό bit της θέσης του k στον A. Προφανώς, χρειαζόμαστε $\lceil \log n \rceil$ συγκρίσεις για να καθορίσουμε όλα τα bits της θέσης του k στον πίνακα A και μία επιπλέον σύγκριση για να αποφανθούμε αν το k βρίσκεται στον πίνακα A ή όχι.

Από την άλλη πλευρά, κάθε αλγόριθμος αναζήτησης πρέπει να καθορίσει τα $\lceil \log n \rceil$ bits της θέσης του ζητούμενου στοιχείου k στον πίνακα A. Τα bits θα καθοριστούν συσχετίζοντας την τιμή του k με την τιμή (ή την κατανομή) των υπόλοιπων στοιχείων του A και συγκρίνοντας το k με κάποιο κατάλληλα επιλεγμένο στοιχείο του A. Όταν δεν γνωρίζουμε τίποτα σχετικά με την κατανομή των στοιχείων του πίνακα A (ανάλυση χειρότερης περίπτωσης), κάθε bit της θέσης του ζητούμενου στοιχείου k θα καθοριστεί από το αποτέλεσμα μιας τουλάχιστον σύγκρισης του k με κατάλληλα επιλεγμένο στοιχείο του A (αφού δεν γνωρίζουμε την κατανομή των στοιχείων του A, δεν μπορούμε να καθορίσουμε τα bits της θέσης του k συσχετίζοντας την τιμή του k με τα στοιχεία του A). Επομένως, στη χειρότερη περίπτωση, κάθε ντετερμινιστικός αλγόριθμος αναζήτησης πρέπει να κάνει $\lceil \log n \rceil$ συγκρίσεις ώστε να μπορέσει να καθορίσει όλα τα bits της θέσης του ζητούμενου στοιχείου. Χωρίς να είναι τυπική απόδειξη, αυτό το επιχείρημα σκιαγραφεί τους λόγους για τους οποίους η δυαδική αναζήτηση είναι ένας *βέλτιστος αλγόριθμος* ως προς τον αριθμό των συγκρίσεων στη χειρότερη περίπτωση.

Άσκηση 2.1. Η *ακέραια τετραφωνική ρίζα* ενός φυσικού αριθμού n είναι ο μεγαλύτερος φυσικός αριθμός x του οποίου το τετράγωνο δεν ξεπερνά το n (δηλ. είναι $x^2 \leq n$ και $(x+1)^2 > n$). Δώστε έναν αλγόριθμο που υπολογίζει την ακέραια τετραγωνική ρίζα του n σε χρόνο χειρότερης περίπτωσης O(log n).

2.1.3. Αναζήτηση με Παρεμβολή

Η δυαδική αναζήτηση είναι η πλέον ενδεδειγμένη λύση για την αναζήτηση στοιχείων σε έναν ταξινομημένο πίνακα όταν δεν γνωρίζουμε τίποτα για την κατανομή των στοιχείων του πίνακα. Όμως μπορούμε να ψάξουμε πολύ γρηγορότερα έναν τηλεφωνικό κατάλογο χρησιμοποιώντας την πληροφορία που διαθέτουμε σχετικά με την κατανομή των στοιχείων του. Για παράδειγμα, δεν χρειάζεται να ανοίξουμε τον κατάλογο στη μέση όταν αναζητούμε ένα επώνυμο που αρχίζει από Ω.

Η *αναζήτηση με παρεμβολή* (interpolation search) χρησιμοποιείται για την αναζήτηση στοιχείων σε έναν ταξινομημένο πίνακα όταν γνωρίζουμε την κατανομή των στοιχείων του πίνακα. Στη συνέχεια εστιάζουμε την προσοχή μας στην περίπτωση που τα στοιχεία του πίνακα Α έχουν επιλεγεί με ανεξάρτητο και ομοιόμορφα τυχαίο τρόπο σε ένα δεδομένο διάστημα. Η αναζήτηση με παρεμβολή μπορεί να εφαρμοστεί (με κατάλληλη προσαρμογή) επιτυγχάνοντας αντίστοιχους χρόνους εκτέλεσης και σε άλλες κατανομές.

Η αναζήτηση με παρεμβολή βασίζεται στη μέθοδο του «διαίρει και βασίλευε» όπως και η δυαδική αναζήτηση. Η βασική διαφορά είναι ότι η αναζήτηση με παρεμβολή συσχετίζει την τιμή του ζητούμενου στοιχείου k με την μέση πυκνότητα των στοιχείων του πίνακα Α ώστε η πληροφορία για τη θέση του k που προκύπτει από μία σύγκριση να ξεπερνά σημαντικά το ένα bit (κατά μέσον όρο).

Η αναζήτηση ξεκινάει από ολόκληρο τον πίνακα A[1…n] και σταδιακά περιορίζεται σε υποπίνακες της μορφής A[low…up], όπου οι τιμές των low και up προκύπτουν από τα αποτελέσματα των συγκρίσεων του k με κατάλληλα επιλεγμένα στοιχεία του Α. Η αναμενόμενη θέση του ζητούμενου στοιχείου k στον πίνακα A[low…up] υπολογίζεται από τον τύπο:

$$pos \leftarrow low + (k - \mathrm{A}[low]) \times (up - low) / (\mathrm{A}[up] - \mathrm{A}[low])$$

Ο παράγοντας (up – low) / (A[up] – A[low]) αντιστοιχεί στη μέση πυκνότητα των στοιχείων του υποπίνακα A[low…up]. Η αναμενόμενη θέση του k προκύπτει πολλαπλασιάζοντας την απόσταση του k από το ελάχιστο στοιχείο A[low] με τη μέση πυκνότητα των στοιχείων στον υποπίνακα A[low…up]. Στο αποτέλεσμα προσθέτουμε το δείκτη low που αντιστοιχεί στην πρώτη θέση στον υποπίνακα A[low…up].

Αν εξαιρέσουμε τον υπολογισμό της αναμενόμενης θέσης pos για το ζητούμενο στοιχείο k, η αναζήτηση με παρεμβολή λειτουργεί όπως και η δυαδική αναζήτηση. Αν το στοιχείο A[pos] είναι ίσο με το k, η αναζήτηση τερματίζεται επιτυχώς. Αν το ζητούμενο στοιχείο k είναι μεγαλύτερο από το A[pos], το k μπορεί να βρίσκεται μόνο στον δεξιό υποπίνακα A[pos+1…up], όπου ο αλγόριθμος εφαρμόζεται αναδρομικά. Ομοίως, αν το k είναι μικρότερο από το A[pos], ο αλγόριθμος εφαρμόζεται αναδρομικά στον αριστερό υποπίνακα A[low…pos–1].

Ο παρακάτω ψευδοκώδικας αποτελεί μία επαναληπτική υλοποίηση της αναζήτησης με παρεμβολή. Είναι εύκολο να δώσετε μία αναδρομική υλοποίηση του αλγόριθμου.

```
Interpolation-Search(A[1...n], k)
   low ← 1; up ← n;
   while low ≤ up do
     if k < A[low] or k > A[up] then return(0);
     pos ← low+(k – A[low])×(up – low)/(A[up] – A[low]);
     if A[pos] = k then return(pos);
     else if A[pos] < k then  low ← pos + 1;
          else                up ← pos – 1;
   return(0);
```

Η βασική ιδέα του αλγόριθμου μπορεί να αποδοθεί με ένα απλό παράδειγμα. Ας θεωρήσουμε έναν πίνακα 1000 στοιχείων ομοιόμορφα κατανεμημένων στο σύνολο των ακεραίων από 1 μέχρι και 1000, και ας υποθέσουμε ότι ψάχνουμε το στοιχείο 700. Υποθέτοντας ότι A[1] = 1 και A[1000] = 1000, η μέση πυκνότητα των στοιχείων του πίνακα είναι 1. Είναι εύλογο λοιπόν να περιμένουμε ότι το στοιχείο 700 θα βρίσκεται περίπου στη θέση 700. Ας υποθέσουμε ότι A[700] = 680 < 700. Τότε γνωρίζουμε ότι το 700 πρέπει να βρίσκεται στον υποπίνακα A[701…1000], του οποίου η σχετική πυκνότητα είναι 300/320 = 0,9375. Η νέα αναμενόμενη θέση του στοιχείου 700 είναι λοιπόν 701 + (700 – 680)×0,9375 ≈ 720, κοκ.

Η απόδειξη ορθότητας της αναζήτησης με παρεμβολή έπεται από την απόδειξη ορθότητας της δυαδικής αναζήτησης (υπενθυμίζουμε ότι η μόνη διαφορά μεταξύ αναζήτησης με παρεμβολή και δυαδικής αναζήτησης είναι στον υπολογισμό της αναμενόμενης θέσης του ζητούμενου στοιχείου k). Στη *χειρότερη* περίπτωση, η αναζήτηση με παρεμβολή μπορεί να εκφυλιστεί σε γραμμική αναζήτηση και να χρειαστεί γραμμικό χρόνο $\Theta(n)$. Στη *μέση* περίπτωση, η αναζήτηση με παρεμβολή έχει χρόνο εκτέλεσης O(loglog n) όταν τα στοιχεία είναι ομοιόμορφα κατανεμημένα σε ένα δεδομένο διάστημα (δηλαδή εντοπίζει το ζητούμενο στοιχείο ανάμεσα σε ένα τρισεκατομμύριο στοιχεία = 2^{40} με περίπου 6 συγκρίσεις κατά μέσον όρο).

Ένα άλλο σημαντικό πλεονέκτημα της αναζήτησης με παρεμβολή είναι ότι οι προσπελάσεις του πίνακα A είναι συνήθως στη γειτονιά της αναμενόμενης θέσης *pos* που υπολογίζεται στην πρώτη επανάληψη. Αυτό επιτρέπει τη σημαντική επιτάχυνση της αναζήτησης σε συστήματα με ιεραρχική οργάνωση μνήμης (όλα τα σύγχρονα υπολογιστικά συστήματα έχουν ιεραρχική οργάνωση μνήμης) επειδή η προσπέλαση μιας θέσης προκαλεί τη μεταφορά όλων των γειτονικών θέσεων σε γρηγορότερες θέσεις μνήμης (π.χ. από το σκληρό δίσκο στην κύρια μνήμη, ή από την κύρια μνήμη στην cache μνήμη).

3. Αλγόριθμοι Επιλογής

Στο πρόβλημα της *επιλογής* (selection) δίνεται ένας πίνακας A[1…n] (όχι κατ' ανάγκη ταξινομημένος) και ένας ακέραιος k, $1 \le k \le n$, και ζητείται να υπολογισθεί το k-οστό μικρότερο στοιχείο του A (δηλαδή το στοιχείο που θα βρίσκεται στην k-οστή θέση του A όταν αυτός ταξινομηθεί σε αύξουσα σειρά). Μερικές ειδικές περιπτώσεις του προβλήματος της επιλογής με ιδιαίτερο ενδιαφέρον είναι για $k = 1$, που αντιστοιχεί στην εύρεση του *ελάχιστου* στοιχείου, για $k = n$, που αντιστοιχεί στην εύρεση του *μέγιστου* στοιχείου, και για $k = n/2$, που αντιστοιχεί στην εύρεση του λεγόμενου *μεσαίου* (median) στοιχείου.

Άσκηση 3.1. Δώστε έναν αλγόριθμο με χρόνο εκτέλεσης $O(n)$ για την εύρεση του ελάχιστου στοιχείου ενός (μη-ταξινομημένου) πίνακα A[1…n]. Τροποποιείστε αυτόν τον αλγόριθμο ώστε να επιστρέφει το μέγιστο στοιχείο. Υπάρχει (ντετερμινιστικός) αλγόριθμος για αυτά τα προβλήματα με χρόνο εκτέλεσης χειρότερης περίπτωσης $o(n)$;

Λύση. Η ακόλουθη συνάρτηση υπολογίζει το ελάχιστο στοιχείο ενός πίνακα A[1…n]. Αν η σύγκριση (key > A[i]) γίνει (key < A[i]), η συνάρτηση θα επιστρέφει το μέγιστο στοιχείο του πίνακα A[1…n].

```
Minimum(A[1...n])

    key ← A[1];
    for i ← 2 to n do
        if key > A[i] then key ← A[i];
    return(key);
```

Η συνάρτηση Minimum εκτελείται σε χρόνο $\Theta(n)$ και εκτελεί ακριβώς $n - 1$ συγκρίσεις μεταξύ στοιχείων του A.

Ο αριθμός των $n - 1$ συγκρίσεων είναι και ο ελάχιστος δυνατός για την εύρεση του ελάχιστου / μέγιστου ενός μη-ταξινομημένου πίνακα n στοιχείων. Κάθε ντετερμινιστικός αλγόριθμος που βρίσκει το ελάχιστο στοιχείο μπορεί να θεωρηθεί σαν ένα τουρνουά μεταξύ των n στοιχείων. Οι συγκρίσεις αντιστοιχούν σε αγώνες του τουρνουά, σε καθέναν από τους οποίους νικητής είναι το μικρότερο στοιχείο. Όλα τα στοιχεία, εκτός από το μικρότερο, πρέπει να χάσουν σε τουλάχιστον ένα αγώνα. Επομένως, απαιτούνται τουλάχιστον $n - 1$ αγώνες / συγκρίσεις για την εύρεση του ελάχιστου στοιχείου. Αυτό

σημαίνει ότι δεν υπάρχει ντετερμινιστικός αλγόριθμος με χρόνο εκτέλεσης χειρότερης περίπτωσης ο(n) για την εύρεση του ελάχιστου / μέγιστου ενός (μη-ταξινομημένου) πίνακα n στοιχείων. ■

Μια απλή λύση για το πρόβλημα της επιλογής είναι να ταξινομήσουμε τον πίνακα Α σε χρόνο Ο(n log n) και να επιστρέψουμε το Α[k]. Από την άλλη πλευρά, τα προβλήματα της εύρεσης του μέγιστου και του ελάχιστου στοιχείου μπορούν να λυθούν σε γραμμικό χρόνο. Επομένως ανακύπτει το ερώτημα αν το γενικότερο πρόβλημα της επιλογής μπορεί επίσης να λυθεί σε χρόνο Ο(n) στη χειρότερη περίπτωση.

Μια απάντηση σε αυτό το ερώτημα προκύπτει από την επανεξέταση της λειτουργίας της ταχείας αναζήτησης (quicksort). Υπενθυμίζουμε ότι μετά τη διαίρεση του πίνακα Α[p…r] από τη συνάρτηση Partition, κανένα στοιχείο του υποπίνακα Α[p…q] δεν είναι μεγαλύτερο από τα στοιχεία του Α[q+1 …r]. Επομένως, αν $k \le q$, το k-οστό στοιχείο του Α[p…r] θα είναι το k-οστό στοιχείο του υποπίνακα Α[p…q]. Αν $k > q$, το k-οστό στοιχείο του Α[p…r] θα είναι το (k – (q + 1 – p))-οστό στοιχείο του υποπίνακα Α[q+1…r]. Ο όρος ((q+1) – p) που αφαιρείται από το k εκφράζει τον αριθμό των στοιχείων που «αποκλείονται» επειδή βρίσκονται στον αριστερό υποπίνακα Α[p…q]. Αυτή η παρατήρηση μας οδηγεί στον ακόλουθο πιθανοτικό «διαίρει και βασίλευε» αλγόριθμο για το πρόβλημα της επιλογής. Ο αλγόριθμος αυτός ονομάζεται *πιθανοτική ταχεία επιλογή* (randomized quick selection).

```
Random-Quickselect(A[p...r], k)
if p = r then return A[p];
q ← Random-Partition(A[p...r]);
m ← q + 1 – p;
if k ≤ m then
    return Random-Quickselect(A[p...q], k);
else
    return Random-Quickselect(A[q+1...r], k – m);
```

Η χειρότερη περίπτωση για αυτό τον αλγόριθμο συμβαίνει όταν το k-οστό στοιχείο του Α βρίσκεται πάντα στον μεγαλύτερο από τους δύο υποπίνακες με μέγεθος $n – 1$ ($n = r – p + 1$). Σε αυτή την περίπτωση, ο αλγόριθμος χρειάζεται χρόνο $\Theta(n^2)$ για την εύρεση ακόμα και του ελάχιστου ή του μέγιστου στοιχείου του Α. Λόγω όμως της πιθανοτικής του φύσης, δεν υπάρχει συγκεκριμένο στιγμιότυπο εισόδου για το οποίο ο αλγόριθμος να επιδεικνύει αυτή τη συμπεριφορά. Η χειρότερη περίπτωση μπορεί να συμβεί μόνο εξ΄

αιτίας συνεχών «άτυχων» επιλογών του στοιχείου γύρω από το οποίο οργανώνεται η διαίρεση.

Ο χρόνος εκτέλεσης μέσης περίπτωσης του αλγόριθμου Random-Quickselect είναι σημαντικά μικρότερος. Έστω $S(n)$ ο μέσος χρόνος για να επιλέξει ο Random-Quickselect το k-οστό στοιχείο από τον πίνακα Α[1…n]. Το $S(n)$ είναι ίσο με $\Theta(n)$, χρόνος που απαιτείται για την κλήση της Random-Partition, συν το μέσο χρόνο για την αναδρομική κλήση της Random-Quickselect. Ο μέσος χρόνος για την αναδρομική κλήση εξαρτάται τόσο από το σημείο διαίρεσης q (δηλαδή από το μέγεθος των δύο υποπινάκων), όσο και από τον υποπίνακα στον οποίο θα συνεχιστεί η αναζήτηση για το k-οστό στοιχείο. Χωρίς βλάβη της γενικότητας, θεωρούμε ότι η αναζήτηση συνεχίζεται πάντα στον μεγαλύτερο υποπίνακα. Επομένως, έχουμε:

$$S(n) = \Theta(n) + \frac{1}{n-1}\sum_{i=1}^{n-1} S(\max\{i, n-i\})$$
$$\le \Theta(n) + \frac{2}{n-1}\sum_{i=n/2}^{n-1} S(i) \ .$$

με αρχική συνθήκη $S(1) = \Theta(1)$. Λύνοντας την παραπάνω αναδρομική εξίσωση προκύπτει ότι ο χρόνος εκτέλεσης μέσης περίπτωσης του αλγόριθμου Random-Quickselect είναι $\Theta(n)$.

3.1.1. Ντετερμινιστική Επιλογή σε Γραμμικό Χρόνο

Σε αυτή την ενότητα, θα περιγράψουμε έναν ντετερμινιστικό αλγόριθμο για το πρόβλημα της επιλογής, ο οποίος επίσης βασίζεται στην ιδέα του «διαίρει και βασίλευε», και έχει γραμμικό χρόνο εκτέλεσης χειρότερης περίπτωσης. Ο αλγόριθμος ταχείας επιλογής (quickselect) είναι παρόμοιος με τον Random-Quickselect, εκτός από το γεγονός ότι το στοιχείο γύρω από το οποίο οργανώνεται η διαίρεση επιλέγεται με ντετερμινιστικό τρόπο. Στον παρακάτω ψευδοκώδικα για τον αλγόριθμο ταχείας επιλογής, απομένει να καθοριστεί ο τρόπος με τον οποίο επιλέγεται το στοιχείο Α[i].

```
Quickselect(A[p...r], k)
    if p = r then return(A[p]);
    Επέλεξε ένα στοιχείο A[i] γύρω από το οποίο θα
    οργανωθεί η διαίρεση του A[p, ..., r].
    swap(A[p], A[i]);
    q ← Partition(A[p...r]);
    m ← q + 1 – p;
    if k ≤ m then
        return Quickselect(A[p...q], k);
    else
        return Quickselect(A[q+1...r], k – m);
```

Από τα παραδείγματα που είδαμε μέχρι τώρα, οι μεγάλοι χρόνοι εκτέλεσης για αλγορίθμους «διαίρει και βασίλευε» συμβαίνουν όταν το μεγαλύτερο επιμέρους στιγμιότυπο που προκύπτει από το βήμα της διαίρεσης έχει μέγεθος σχεδόν ίσο με αυτό του αρχικού στιγμιότυπου. Σε αυτές τις περιπτώσεις, ο αλγόριθμος πληρώνει το κόστος της διαίρεσης χωρίς να κάνει σημαντική πρόοδο.

Στην περίπτωση του προβλήματος της επιλογής, αν είχαμε μια ντετερμινιστική διαδικασία για να βρούμε ένα στοιχείο κοντά στο μεσαίο (π.χ. στοιχείο με σειρά τουλάχιστον n/d, για κάποια σταθερά $d > 1$) και οργανώναμε τη διαίρεση γύρω από αυτό το στοιχείο, το μέγεθος των επιμέρους στιγμιότυπων θα περιοριζόταν σημαντικά σε κάθε αναδρομική κλήση. Έτσι, θα μπορούσαμε να λύσουμε αναδρομικά το πρόβλημα της επιλογής σε γραμμικό χρόνο (στη χειρότερη περίπτωση).

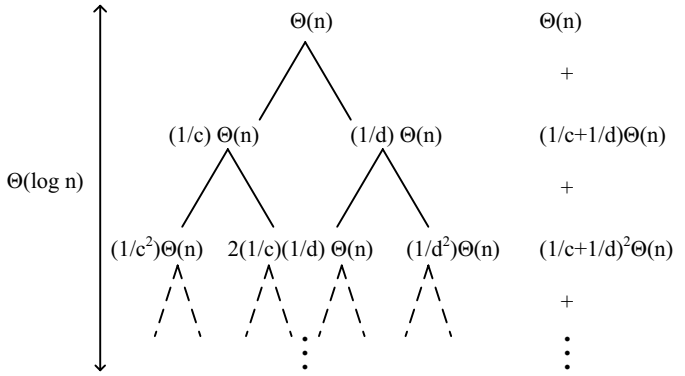
Μια ιδέα είναι να επιλέξουμε ένα σημαντικού μεγέθους δείγμα στοιχείων του πίνακα Α, και αναδρομικά, να βρούμε το μεσαίο στοιχείο (median) αυτού του δείγματος. Έστω ότι σε χρόνο $\Theta(n)$ μπορούμε να επιλέξουμε ένα δείγμα n/c στοιχείων, για κάποια σταθερά $c > 1$. Έστω επίσης ότι ο αλγόριθμος Quickselect λύνει το πρόβλημα της επιλογής σε χρόνο (στη χειρότερη περίπτωση) $T(n)$. Με αναδρομική κλήση του Quickselect, μπορούμε να υπολογίσουμε το μεσαίο στοιχείο αυτού του δείγματος σε χρόνο $T(n/c)$.

Ας θεωρήσουμε ότι το μέγεθος n/c και ο τρόπος επιλογής του δείγματος εγγυώνται ότι αν η διαίρεση οργανωθεί γύρω από το μεσαίο στοιχείο του δείγματος, ο μεγαλύτερος από τους δύο υποπίνακες θα έχει μέγεθος το πολύ n/d. Άρα, η αναδρομική κλήση της Quickselect χρειάζεται χρόνο το πολύ $T(n/d)$. Επομένως, ο χρόνος χειρότερης περίπτωσης για την Quickselect θα δίνεται από την αναδρομική εξίσωση $T(n) = \Theta(n) +$

$T(n/c) + T(n/d)$, όπου $\Theta(n)$ είναι ο χρόνος για την επιλογή του δείγματος και για την εκτέλεση της συνάρτησης Partition, $T(n/c)$ είναι ο χρόνος (χειρότερης περίπτωσης) για τον υπολογισμό του μεσαίου στοιχείου του δείγματος αναδρομικά με κλήση της Quickselect, και $T(n/d)$ είναι ο χρόνος (χειρότερης περίπτωσης) για την αναδρομική κλήση της Quickselect στον κατάλληλο υποπίνακα.

Άσκηση 3.3. Αν $1/c + 1/d < 1$, αποδείξτε ότι η λύση της αναδρομικής εξίσωσης $T(n) = \Theta(n) + T(n/c) + T(n/d)$ είναι $T(n) = \Theta(n)$. Τι συμβαίνει στην περίπτωση που $1/c + 1/d = 1$;

Λύση. Παρατηρώντας το δέντρο της αναδρομής (Σχήμα 3.1), βλέπουμε ότι η συνεισφορά του επιπέδου i (η ρίζα βρίσκεται στο επίπεδο 0) είναι $(1/c+1/d)^i \Theta(n)$. Επιπλέον, το ύψος του δέντρου είναι μεταξύ $\log_{(1/c)}n$ και $\log_{(1/d)}n$, είναι δηλαδή $\Theta(\log n)$.



Σχήμα 3.1. Το δέντρο της αναδρομής για την εξίσωση T(n) = T(n/c) + T(n/d) + Θ(n).

Επομένως, είναι:

$$T(n) = \sum_{i=0}^{\Theta(\log n)} \Theta(n)(1/c+1/d)^i = \Theta(n) \sum_{i=0}^{\Theta(\log n)} (1/c+1/d)^i \quad .$$

Όταν $1/c + 1/d < 1$, το άθροισμα $\sum_{i=0}^{\Theta(\log n)} (1/c+1/d)^i$ είναι ίσο με μια σταθερά, επειδή είναι άθροισμα όρων γεωμετρικής προόδου με λόγο μικρότερο της μονάδας. Επομένως, σε αυτή την περίπτωση $T(n) = \Theta(n)$.

Στην περίπτωση που $1/c + 1/d = 1$, είναι $\sum_{i=0}^{\Theta(\log n)} (1/c + 1/d)^i = \Theta(\log n)$, και $T(n) = \Theta(n \log n)$. ■

Ας εστιάσουμε λοιπόν στην επιλογή του δείγματος. Έχοντας θέσει σαν στόχο τον σχεδιασμό ενός ντετερμινιστικού αλγορίθμου, δεν μπορούμε να επιλέξουμε το δείγμα με τυχαίο τρόπο (π.χ. τυχαία δειγματοληψία με ή χωρίς αντικατάσταση των στοιχείων). Επίσης δεν μπορούμε να επιλέξουμε αυθαίρετα οποιαδήποτε n/c στοιχεία του πίνακα Α. Πράγματι, αν οργανώσουμε τη διαίρεση γύρω από το μεσαίο στοιχείο ενός αυθαίρετα επιλεγμένου δείγματος, τότε μπορούμε να εγγυηθούμε μόνο ότι ο μεγαλύτερος υποπίνακας θα έχει μέγεθος $(1 - 1/(2c))\, n$.

Γι' αυτό διαιρούμε τον πίνακα Α σε $\lceil n/5 \rceil$ ομάδες, 5 στοιχείων η κάθε μία (η τελευταία ομάδα θα έχει ακριβώς $n \bmod 5$ στοιχεία), και βρίσκουμε το μεσαίο στοιχείο κάθε τέτοιας ομάδας. Το δείγμα αποτελείται από αυτά τα $\lceil n/5 \rceil$ μεσαία στοιχεία. Προφανώς, το μεσαίο στοιχείο κάθε ομάδας μπορεί να υπολογισθεί σε σταθερό χρόνο, οπότε ο χρόνος χειρότερης περίπτωσης για την επιλογή του δείγματος είναι $\Theta(n)$. Ένα παράδειγμα αυτής της διαδικασίας φαίνεται στον Πίνακα 3.1. Το μεσαίο στοιχείο του δείγματος είναι το 7, και η διαίρεση που προκύπτει είναι [2, 1, 4, 6, 5, 3] και [9, 7, 15, 10, 11, 13, 8, 12, 14].

Πίνακας Α	9	7	15	2	1	10	4	11	6	5	13	8	12	14	3
Ομάδες	[9, **7**, 15, 2, 1]					[10, 4, 11, **6**, 5]					[14, 15, **12**, 10, 9]				
Δείγμα	**7**					6					12				

Πίνακας 3.1. Ένα παράδειγμα της διαδικασίας επιλογής του δείγματος.

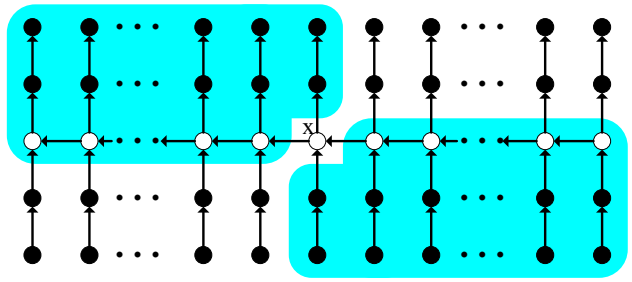
Όσον αφορά το μέγεθος του μεγαλύτερου από τους δύο υποπίνακες που προκύπτουν από τη διαίρεση γύρω από το μεσαίο στοιχείο του δείγματος, αυτό δεν μπορεί να είναι μεγαλύτερο του $7n/10$. Για να αποδείξουμε αυτό, θεωρούμε ότι τα στοιχεία του Α είναι οργανωμένα σε έναν πίνακα με $\lceil n/5 \rceil$ στήλες και 5 γραμμές. Κάθε στήλη περιέχει τα στοιχεία μιας από τις $\lceil n/5 \rceil$ ομάδες (στις οποίες έχουν χωριστεί τα στοιχεία του Α για την επιλογή του δείγματος) σε αύξουσα σειρά. Επομένως, όλα τα μεσαία στοιχεία των στηλών βρίσκονται στην 3η γραμμή. Επιπλέον, η θέση των στηλών είναι τέτοια ώστε το μεσαίο στοιχείο μιας στήλης να μην είναι μεγαλύτερο από το μεσαίο στοιχείο της επόμενης. Συνεπώς το μεσαίο στοιχείο του δείγματος βρίσκεται στην $(n/10)$-η στήλη και στην 3η γραμμή.

Από την κατασκευή αυτή φαίνεται ότι τα στοιχεία που ανήκουν στις στήλες από 1 μέχρι και $n/10$ και στις γραμμές 1η, 2η, και 3η (συνολικά $3n/10$ στοιχεία) είναι μικρότερα του μεσαίου στοιχείου του δείγματος. Επομένως, αυτά τα στοιχεία θα τοποθετηθούν στον πρώτο υποπίνακα. Επίσης, τα στοιχεία που βρίσκονται στις στήλες από $n/10$ μέχρι και $\lceil n/5 \rceil$ και στις γραμμές 3η, 4η, και 5η (συνολικά $3n/10$ στοιχεία) είναι μεγαλύτερα του μεσαίου στοιχείου του δείγματος, και συνεπώς θα τοποθετηθούν στον δεύτερο υποπίνακα. Άρα, κανένας από τους δύο υποπίνακες δεν μπορεί να έχει περισσότερα από $7n/10$ στοιχεία.

Η κατασκευή αυτή για το παράδειγμα του Πίνακα 3.1 φαίνεται στον Πίνακα 3.2. Επίσης, η γενική μορφή της κατασκευής φαίνεται στο Σχήμα 3.2, όπου τα στοιχεία σημειώνονται με κύκλους, τα στοιχεία του δείγματος με μαύρους κύκλους, και το μεσαίο στοιχείο του δείγματος με x. Τα βέλη δηλώνουν ότι το στοιχείο προορισμού είναι μικρότερο από το στοιχείο αφετηρίας.

4	1	3
5	2	8
6	**7**	12
10	9	13
11	15	14

Πίνακας 3.2. Τουλάχιστον 5 στοιχεία είναι μεγαλύτερα και άλλα πέντε μικρότερα από το μεσαίο στοιχείο του δείγματος.



Σχήμα 3.2. Το μέγεθος του μικρότερου υποπίνακα είναι τουλάχιστον $3n/10$.

Αφού $1/5 + 7/10 = 9/10 < 1$, η Άσκηση 3.3 δείχνει ότι ο χρόνος εκτέλεσης χειρότερης περίπτωσης του αλγορίθμου quickselect είναι $\Theta(n)$.

Άσκηση 3.4. Ποιός ο χρόνος εκτέλεσης χειρότερης περίπτωσης για τον αλγόριθμο Quickselect όταν τα στοιχεία του πίνακα Α χωρίζονται σε ομάδες των 7 αντί σε ομάδες των 5; Τι συμβαίνει όταν τα στοιχεία του Α χωρίζονται σε ομάδες των 3;

Άσκηση 3.5. Τροποποιείστε τον αλγόριθμο ταχείας ταξινόμησης (quicksort) ώστε να έχει χρόνο εκτέλεσης χειρότερης περίπτωσης $O(n \log n)$.

Άσκηση 3.6. Έστω Α και Β δύο πίνακες n στοιχείων ταξινομημένοι σε αύξουσα σειρά. Δώστε έναν «διαίρει και βασίλευε» αλγόριθμο με χρόνο εκτέλεσης $O(\log n)$ που να επιστρέφει το μεσαίο (n-οστό) στοιχείο της ένωσης των δύο πινάκων. Για απλότητα, μπορείτε να υποθέσετε ότι όλα τα $2n$ στοιχεία είναι διαφορετικά μεταξύ τους.

Λύση. Έστω ότι τα ενδιάμεσα στοιχεία $A[n/2]$ και $B[n/2]$ είναι διαφορετικά μεταξύ τους, και έστω $A[n/2] < B[n/2]$. Τότε το μεσαίο στοιχείο του $A \cup B$ πρέπει να βρίσκεται μεταξύ των $A[n/2]$ και $B[n/2]$, αφού όλα τα στοιχεία του $A[1\ldots n/2]$ είναι μικρότερα από το $A[n/2]$ και όλα τα στοιχεία του $B[n/2\ldots n]$ είναι μεγαλύτερα από το $B[n/2]$. Συγκεκριμένα το μεσαίο στοιχείο του $A \cup B$ θα είναι το μεσαίο στοιχείο του πίνακα $A[n/2\ldots n] \cup B[1\ldots n/2]$. Ομοίως, αν $A[n/2] > B[n/2]$, το μεσαίο στοιχείο του $A \cup B$ θα είναι το μεσαίο στοιχείο του $A[1\ldots n/2] \cup B[n/2\ldots n]$. Ο αλγόριθμος συνεχίζει με την ίδια μέθοδο, καλώντας αναδρομικά τον εαυτό του. Στη συνέχεια δίνεται ο ψευδοκώδικας.

```
median_of_union(A[p1...r1], B[p2...r2])
if r1 − p1 ≤ 1 then
    Υπολόγισε και επέστρεψε το μεσαίο στοιχείο
    ταξινομώντας τον πίνακα A[p1...r1]∪B[p2...r2];
q1 = (p1 + r1) / 2; q2 = (p2 + r2) / 2;
if A[q1] < B[q2] then
    return median_of_union(A[q1...r1], B[p2...q2]);
else
    return median_of_union(A[p1...q1], B[q2...r2]);
```

Παρατηρούμε ότι σε κάθε αναδρομική κλήση, ο αλγόριθμος, χρησιμοποιώντας το αποτέλεσμα μιας σύγκρισης, υποδιπλασιάζει το μέγεθος του προβλήματος. Επομένως, ο χρόνος εκτέλεσης $T(n)$ του αλγορίθμου δίνεται από την αναδρομική εξίσωση $T(n) = T(n/2) + \Theta(1)$, η οποία γνωρίζουμε ότι έχει λύση $T(n) = \Theta(\log n)$.

Άσκηση 3.7. Έστω n διαφορετικά στοιχεία $x_1, x_2, \ldots, x_n$ στα οποία έχουμε αντιστοιχήσει θετικά βάρη $w_1, w_2, \ldots, w_n$ τέτοια ώστε $\sum_{i=1}^{n} w_i = 1$. Το πρόβλημα της εύρεσης του

βεβαρυμένου μεσαίου στοιχείου (weighted median) είναι να βρεθεί ένα στοιχείο x_k που ικανοποιεί τις σχέσεις $\sum_{x_i < x_k} w_i \le \frac{1}{2}$ και $\sum_{x_i > x_k} w_i \le \frac{1}{2}$.

Σημειώστε ότι τα στοιχεία $x_1, \ldots, x_n$ δεν είναι ταξινομημένα. Αποδείξτε ότι:

- Το μεσαίο στοιχείο των $x_1, \ldots, x_n$ είναι ίδιο με το βεβαρυμένο μεσαίο στοιχείο, όταν όλα τα βάρη είναι $w_i = 1/n$, $i = 1, 2, \ldots, n$.
- Το βεβαρυμένο μεσαίο στοιχείο μπορεί να υπολογιστεί σε χρόνο $O(n \log n)$ ταξινομώντας την ακολουθία $x_1, \ldots, x_n$.
- Το βεβαρυμένο μεσαίο στοιχείο μπορεί να υπολογιστεί σε γραμμικό χρόνο χρησιμοποιώντας έναν «διαίρει και βασίλευε» αλγόριθμο γραμμικού χρόνου για τον υπολογισμό του μεσαίου στοιχείου ενός πίνακα.

Λύση. Εξ' ορισμού, το βεβαρυμένο μεσαίο στοιχείο x_k έχει την ιδιότητα $\sum_{x_i < x_k} w_i \le \frac{1}{2}$ και $\sum_{x_i > x_k} w_i \le \frac{1}{2}$. Όταν τα βάρη $w_i = 1/n$, $i = 1, 2, \ldots, n$, οι παραπάνω σχέσεις υποδεικνύουν ότι το πολύ $n/2$ στοιχεία είναι μικρότερα και το πολύ $n/2$ στοιχεία είναι μεγαλύτερα από το x_k. Επομένως, το x_k είναι και το μεσαίο στοιχείο της ακολουθίας $x_1, \ldots, x_n$.

Αφού ταξινομήσουμε τα στοιχεία $x_1, \ldots, x_n$ σε αύξουσα σειρά, βαδίζοντας από το μικρότερο προς το μεγαλύτερο, υπολογίζουμε το άθροισμα των βαρών των στοιχείων που έχουμε επισκεφθεί μέχρι στιγμής. Το βεβαρυμένο μεσαίο στοιχείο x_k είναι το πρώτο για το οποίο το παραπάνω άθροισμα φθάσει ή ξεπεράσει το 1/2. Ο αλγόριθμος αυτός χρειάζεται $\Theta(n \log n)$ χρόνο για την ταξινόμηση των στοιχείων και γραμμικό χρόνο για την εύρεση του x_k στην ταξινομημένη ακολουθία.

Θεωρούμε τον ακόλουθο αλγόριθμο:

1. Υπολόγισε το μεσαίο στοιχείο x_m της ακολουθίας $x_1, \ldots, x_n$ χρησιμοποιώντας τη διαδικασία Quickselect.
2. Καλώντας τη διαδικασία Partition, διαίρεσε την ακολουθία $x_1, \ldots, x_n$ σε δύο επιμέρους ακολουθίες. Οργάνωσε τη διαίρεση γύρω από το μεσαίο στοιχείο x_m.

3. Υπολόγισε τα αθροίσματα $W_L = \sum_{x_i < x_m} w_i$ και $W_R = \sum_{x_i > x_m} w_i$.

4. Αν το x_m είναι και το βεβαρυμένο μεσαίο στοιχείο, επέστρεψέ το.

5. Διαφορετικά, πρόσθεσε το μικρότερο από τα αθροίσματα W_L και W_R στο βάρος του x_m και εφάρμοσε αναδρομικά τον αλγόριθμο στην επιμέρους ακολουθία με το μεγαλύτερο άθροισμα βαρών (στην οποία έχεις προσθέσει και το στοιχείο x_m).

Για την ορθότητα, στην περίπτωση που το βεβαρυμένο μεσαίο στοιχείο είναι ίδιο με το μεσαίο στοιχείο x_m, ο αλγόριθμος το βρίσκει και το επιστρέφει (βήμα 4). Σε διαφορετική περίπτωση, η πρόσθεση του βάρους της «ελαφρότερης» επιμέρους ακολουθίας στο x_m δεν αλλάζει το βεβαρυμένο μεσαίο στοιχείο της αρχικής ακολουθίας, αλλά επιτρέπει να περιορίσουμε την αναζήτηση στην επιμέρους ακολουθία με το μεγαλύτερο βάρος, όπου πρέπει να βρίσκεται το βεβαρυμένο μεσαίο στοιχείο.

Ο χρόνος εκτέλεσης $T(n)$ του παραπάνω αλγορίθμου δίνεται από την αναδρομική εξίσωση $T(n) = T(n/2) + \Theta(n)$, η οποία έχει λύση $T(n) = \Theta(n)$.

4. Βιβλιογραφία

Υπάρχουν πολλά βιβλία στη διεθνή βιβλιογραφία και αρκετά στην ελληνική βιβλιογραφία που καλύπτουν σε επαρκή έκταση και βάθος όλα τα θέματα που θίχτηκαν σε αυτές τις εκπαιδευτικές σημειώσεις. Ενδεικτικά αναφέρουμε:

[1] D. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching, Second Edition.* Addison-Wesley, 1998.

[2] T.H. Cormen. C.E. Leiserson, R.L. Rivest, and C. Stein. *An Introduction to Algorithms, 2nd Edition*. The MIT Press, 2003.

[3] M.T. Goodrich, R. Tamassia, and D.M. Mount. *Data Structures and Algorithms in C++*. Wiley.

[4] S. Sahni. *Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++*. Μετάφραση Ι. Μανωλόπουλος και Ι. Θεοδωρίδης, Εκδόσεις Τζιόλα, 2004.

[5] Γ.Φ. Γεωργακόπουλος. *Δομές δεδομένων. Έννοιες, τεχνικές και αλγόριθμοι*. Πανεπιστημιακές Εκδόσεις Κρήτης.

[6] Π. Μποζάνης. *Δομές Δεδομένων: Ταξινόμηση και Αναζήτηση με Java*. Εκδόσεις Τζιόλα, 2003.