Κεφάλαιο 2

Αναδρομή

2.1 Divide and Conquer

Πολλοί χρήσιμοι αλγόριθμοι οι οποίοι θα εξεταστούν στη συνέχεια έχουν αναδρομική δομή. Η *αναδρομή* είναι μια ισχυρή αλγοριθμική μέθοδος, πιο αισθητική (όσο αφορά στο μέγεθος των αλγορίθμων και κατά συνέπεια των προγραμμάτων που γράφουμε), πιο φυσική (αφού συμβαδίζει με την διαίσθησή μας) και πιο ισχυρή (μιας και μας δίδει την δυνατότητα να περιγράψουμε με εύκολο και εύληπτο τρόπο δύσκολα προβλήματα).

Για την επίλυση ενός δεδομένου προβλήματος ένας αναδρομικός αλγόριθμος καλεί τον εαυτό του αναδρομικά μία ή περισσότερες φορές επιλύοντας διάφορα υποπροβλήματα του αρχικού προβλήματος. Με τον όρο υποπρόβλημα εννοούμε ένα πρόβλημα της ίδιας φύσης με το αρχικό, αλλά μικρότερου μεγέθους. Ένας αλγόριθμος τέτοιου είδους τυπικά ακολουθεί μια **διαίρει και βασίλευε (divide and conquer)** προσέγγιση, κατά την οποία το αρχικό πρόβλημα διασπάται σε μια σειρά από υποπροβλήματα τα οποία επιλύονται ένα προς ένα αναδρομικά και έπειτα συνδυάζονται οι λύσεις αυτών για να δημιουργηθεί η λύση του αρχικού προβλήματος. Σε κάθε αναδρομικό επίπεδο, η τεχνική **divide and conquer** περιλαμβάνει τρία βήματα:

Divide: Διαίρεση του προβλήματος σε μικρότερα προβλήματα (υποπροβλήματα).

Conquer: Επίλυση των μικρότερων προβλημάτων (είτε αναδρομικά, είτε με άμεσο τρόπο εάν έχουν ικανοποιητικά μικρό μέγεθος).

Combine: Συνδυασμός των λύσεων για την εύρεση της αρχικής λύσης.

2.1.1 Merge-Sort

Ένα κλασσικό παράδειγμα **divide and conquer** αλγορίθμου, είναι ο αλγόριθμος ταξινόμησης Merge-Sort. Διαισθητικά, ο αλγόριθμος εξελίσσεται ως εξής:

Divide: Διαίρεση του πίνακα n στοιχείων σε δύο υποπίνακες $\frac{n}{2}$ στοιχείων ο καθένας.

Conquer: Αναδρομική ταξινόμηση των δύο υποπινάκων με χρήση της MERGE-SORT.

Combine: Συγχώνευση των δύο ταξινομημένων υποπινάκων.

Η αναδρομή στο βήμα **Conquer** σταματά όταν οι υποπίνακες οι οποίοι προκύπτουν μετά τη διάσπαση έχουν μόνο ένα στοιχείο, διότι σε αυτή την περίπτωση ο πίνακας είναι ήδη ταξινομημένος. Η βασική λειτουργία του αλγορίθμου MERGE-SORT είναι η συγχώνευση δύο ταξινομημένων υποπινάκων η οποία γίνεται στο βήμα **Combine** χρησιμοποιώντας τον αλγόριθμο MERGE.

Ο αλγόριθμος MERGE (σχήμα 2.1) ξεκινάει συγκρίνοντας το πρώτο στοιχείο του πίνακα a με το πρώτο στοιχείο του πίνακα b και επιλέγει το μικρότερο από τα δύο για να το τοποθετήσει πρώτο στον ταξινομημένο πίνακα c ο οποίος τελικά θα περιέχει τα στοιχεία και των δύο πινάκων. Εν συνεχεία ο μετρητής του πίνακα από τον οποίο επελέγει το πρώτο στοιχείο αυξάνεται κατά ένα και η διαδικασία σύγκρισης συνεχίζεται έτσι ώστε τελικά όλα τα στοιχεία των πινάκων a και b να τοποθετηθούν ταξινομημένα στον πίνακα c.

MERGE (a, b, c)

```
1.  i = 1
2.  j = 1
3.  for k = 1 to m + n do
4.      if a_i ≤ b_j then
5.          c_k = a_i
6.          i = i + 1
7.      else
8.          c_k = b_j
9.          j = j + 1
10.     end if
11. end for
```

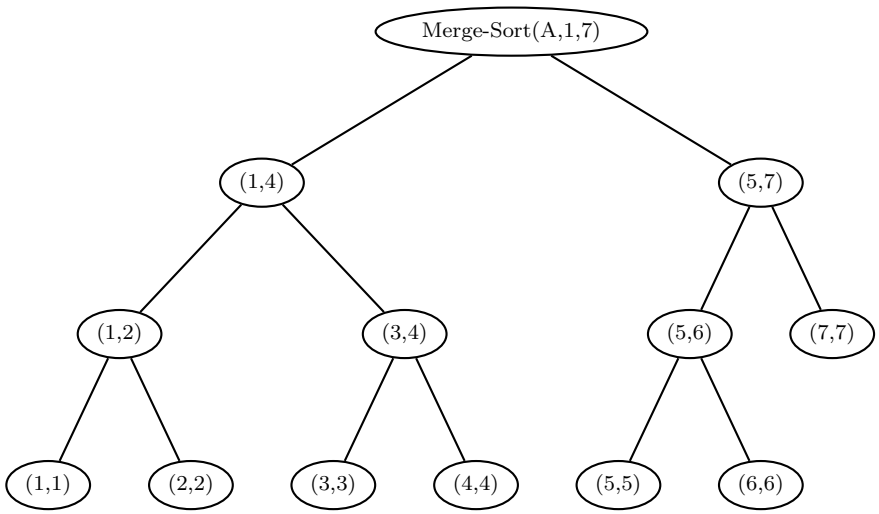
Σχήμα 2.1: Αλγόριθμος MERGE για την συγχώνευση των ταξινομημένων πινάκων a (μήκους m) και b (μήκους n) στον πίνακα c (μήκους $m + n$).

Για παράδειγμα, εάν ο πίνακας a περιέχει τα στοιχεία $2, 6, 7, 9$ και ο πίνακας b τα στοιχεία $4, 5, 8$ τότε ο πίνακας c ο οποίος θα προκύψει μετά την εκτέλεση του αλγορίθμου MERGE θα έχει ως εξής: $2, 4, 5, 6, 7, 8, 9$. Η πολυπλοκότητα του αλγορίθμου είναι $O(m + n)$.

Τον αλγόριθμο MERGE μπορούμε να τον χρησιμοποιήσουμε ως υπορουτίνα του αλγορίθμου MERGE-SORT (σχήμα 2.2), όπου A είναι ο πίνακας στοιχείων και l, r δείκτες που καθορίζουν τα άκρα του προς ταξινόμηση υποπίνακα του A.

MergeSort (A, l, r)
1. **if** $l < r$ **then**
2. $q \leftarrow \lfloor (l + r)/2 \rfloor$
3. MergeSort(A, l, q)
4. MergeSort$(A, q + 1, r)$
5. Merge /* Τους δύο υποπίνακες */
6. **end if**

Σχήμα 2.2: Αναδρομικός αλγόριθμος MergeSort για την ταξινόμηση των στοιχείων ενός πίνακα.



Σχήμα 2.3: Το δένδρο αναδρομικών κλήσεων του αλγορίθμου Merge-Sort.

Θα δούμε την λειτουργία του αλγορίθμου Merge-Sort με ένα παράδειγμα. Έστω ο πίνακας $A = 6, 9, 2, 7, 4, 5, 8$. Το δένδρο των αναδρομικών κλήσεων φαίνεται στο σχήμα 2.3. Ο αλγόριθμος υλοποιημένος σε μια γλώσσα τύπου C είναι ο εξής:

```
Merge-Sort int b[N] void MergeSort (int p, int r) {
    int i, j, k, m;
    if (p < r) {
      q = (p + r) / 2;
      MergeSort (p, q);
```

```
      MergeSort (q + 1, r);
      for (i = q; i >= p; --i)
        b[i] = a[i];
      for (j = q+1; j <= r; ++j)
        b[r+q+1-j] = a[j];
      i = p; j = r;
      for (k = p; k <= r; ++k)
        if (b[i] < b[j] {
           a[k] = b[i];
           ++i;
        } else {
           a[k] = b[j];
           --j;
        }
    }
}
```

Όταν ένας αλγόριθμος περιέχει μία ή περισσότερες αναδρομικές κλήσεις στον εαυτό του, ο συνολικός χρόνος εκτέλεσής του μπορεί συνήθως να περιγραφεί από μια *αναδρομική εξίσωση* ή απλά *αναδρομή*, μια συνάρτηση δηλαδή η οποία περιγράφει το συνολικό χρόνο εκτέλεσης του αλγορίθμου σε ένα πρόβλημα μεγέθους n χρησιμοποιώντας το χρόνο εκτέλεσης σε προβλήματα μικρότερου μεγέθους. Εν συνεχεία μπορούν να χρησιμοποιηθούν διάφορες μέθοδοι τις οποίες θα δούμε εκτενώς στη συνέχεια, για την επίλυση της αναδρομικής εξίσωσης η οποία έχει προκύψει.

Πιο συγκεκριμένα, ας συμβολίσουμε με $T(n)$ το χρόνο εκτέλεσης του αλγορίθμου MERGE-SORT για ένα πρόβλημα μεγέθους n. Εάν $n = 1$ τότε ο υποπίνακας με ένα στοιχείο είναι ήδη ταξινομημένος και ο αλγόριθμος παίρνει σταθερό χρόνο τον οποίο γράφουμε με $\Theta(1)$. Επίσης ο αλγόριθμος MERGE ο οποίος επιτυγχάνει τη συγχώνευση χρειάζεται χρόνο $\Theta(n)$. Συνολικά λοιπόν μπορούμε να γράψουμε την εξής αναδρομική εξίσωση:

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

Η επίλυση της παραπάνω αναδρομικής εξίσωσης μας δίδει $T(n) = \Theta(n \log n)$ όπως θα δούμε παρακάτω.

Γενικά, η εύρεση της αναδρομικής εξίσωσης η οποία περιγράφει το χρόνο εκτέλεσης ενός **divide and conquer** αλγορίθμου βασίζεται στα τρία βήματα (divide, conquer, combine) της μεθόδου. Αν υποθέσουμε ότι η υποδιαίρεση του αρχικού προβλήματος σε υποπροβλήματα οδηγεί στη δημιουργία a υποπροβλημάτων το καθένα από τα οποία έχει μέγεθος $1/b$ του αρχικού και ότι το βήμα combine χρειάζεται χρόνο $d(n)$, τότε λαμβάνουμε την παρακάτω αναδρομική εξίσωση:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ aT(\frac{n}{b}) + d(n) & \text{if } n > 1. \end{cases}$$

Μπορούμε να χρησιμοποιήσουμε την μέθοδο της *αντικατάστασης (substitution)* για να βρούμε τη λύση στην εξίσωση αυτή, ως εξής:

$$\begin{aligned}
T(n) &= aT(n/b) + d(n) \\
&= a[aT(n/b^2) + d(n/b)] + d(n) \\
&= a^2T(n/b^2) + ad(n/b) + d(n) \\
&= a^2[aT(n/b^3) + d(n/b^2)] + ad(n/b) + d(n) \\
&= a^3T(n/b^3) + a^2d(n/b^2) + ad(n/b) + d(n) \\
&= \dots \\
&= a^iT(n/b^i) + \sum_{j=0}^{i-1} a^j d(n/b^j)
\end{aligned}$$

Αν υποθέσουμε τώρα ότι $n = b^k$ τότε θα έχουμε ότι $T(n/b^k) = T(1) = 1$ και για $i = k$ θα λάβουμε:

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

Επίσης $k = \log_b n$ και επομένως $a^k = a^{\log_b n} = n^{\log_b a}$. Στην περίπτωση της MERGE-SORT έχουμε $a = b = 2$ και $d(n) = n - 1 (= \Theta(n))$. Κάνοντας τις αντικαταστάσεις έχουμε:

$$\begin{aligned}
T(n) &= n + \sum_{j=0}^{k-1} 2^j(2^{k-j} - 1) \\
&= n + \sum_{j=0}^{k-1} (2^k - 2^j) \\
&= n + 2^k k - \frac{(2^k - 1)}{(2 - 1)} \\
&= n + n\log n - n + 1 \\
&= n\log n + 1
\end{aligned}$$

2.1.2 Binary Search

Ο αλγόριθμος της δυαδικής αναζητησης έχει ως είσοδο έναν ταξινομημένο πίνακα A και ψάχνει να βρει αν υπάρχει ένα στοιχείο a μέσα σ΄ αυτον τον πίνακα. Παρακάτω περιγράφουμε πως ο αλγόριθμος εκμεταλλεύεται την τεχνική Divide-and-Conquer:

Divide: Βρίσκουμε το μεσαίο στοιχείο του πίνακα και τον χωρίζουμε σε δύο υποπίνακες. Αν το μεσαίο στοιχείο είναι μικρότερο του στοιχείου που ψάχνουμε, τότε κρατάμε τον υποπίνακα με τα μικρότερα από το μεσαίο στοιχεία,

διαφορετικά κρατάμε τον υποπίνακα με τα μεγαλύτερα από το μεσαίο στοιχεία.

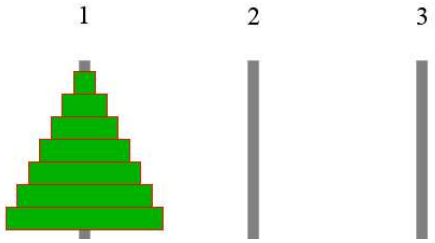
Conquer: Επαναλαμβάνουμε την παραπάνω διαδικασία στον επιλεγμένο υποπίνακα, μεχρίς ότου βρούμε το στοιχείο σαν μεσαίο η μεχρι να εχουμε κενούς υποπίνακες.

Combine: Δεν Υπάρχει βήμα Combine αφού με το πέρας όλων των αναδρομικών εκτελέσεων έχουμε τη λύση.

Η αναδρομική εξίσωση του αλγοριθμου είναι $T(n) = T(n/2) + c = O(logn)$. Οπότε, αντί να ψάχνουμε σειριακά σε όλο τον πίνακα σε $O(n)$ χρόνο χρησιμοποιώντας έναν αλγόριθμο Divide and Conquer ο χρόνος μειώνεται σε $O(logn)$!

2.1.3 Οι πύργοι του Hanoi

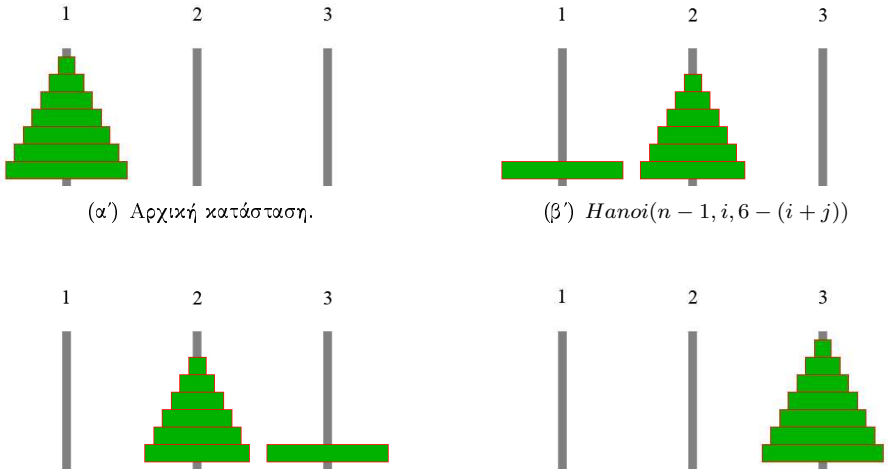
Ας εξετάσουμε ένα άλλο κλασσικό πρόβλημα στο οποίο χρησιμοποιείται η έννοια της αναδρομής. Πρόκειται για το πρόβλημα των πύργων του Hanoi το οποίο επινοήθηκε από τον γάλλο μαθηματικό Edouard Lucas το 1883. Στο πρόβλημα αυτό μας δίδονται 3 στύλοι 1,2,3 αντίστοιχα καθώς επίσης και n δίσκοι διαφορετικών μεγεθών οι οποίοι αρχικά είναι τοποθετημένοι στο στύλο 1, σχηματίζοντας έναν κώνο με τον πιο μεγάλο δίσκο κάτω και τον πιο μικρό πάνω, όπως φαίνεται στο σχήμα 2.4. Το ζητούμενο είναι να μεταφερθούν όλοι οι δίσκοι από το στύλο



Σχήμα 2.4: Οι πύργοι του Hanoi

1 στον στύλο 3, μετακινώντας μόνο ένα δίσκο τη φορά και προσέχοντας ποτέ να μην είναι ένας μικρός δίσκος κάτω από έναν μεγαλύτερο.

Ο αλγόριθμος ο οποίος επιλύει το πρόβλημα έχει ως εξής:



(α′) Αρχική κατάσταση.

(β′) $Hanoi(n-1, i, 6-(i+j))$

(γ′) Μετακίνηση του μεγάλου δίσκου από το στύλο i στον στύλο j.

(δ′) $Hanoi(n-1, 6-(i+j), j)$

Σχήμα 2.5: Σχηματική λειτουργία του αλγορίθμου.

HANOI (n, i, j)
1. **if** $n \geq 1$ **then**
2. HANOI$(n-1, i, 6-(i+j))$
3. Μεταφορά του μεγάλου δίσκου από τον στύλο i στον στύλο j
4. HANOI$(n-1, 6-(i+j), j)$
5. **end if**

Στο σχήμα 2.5 φαίνεται ο τρόπος λειτουργίας του αλγορίθμου.

Για να εξετάσουμε την πολυπλοκότητα του αλγορίθμου, γράφουμε την αναδρομική εξίσωση η οποία εκφράζει το χρόνο εκτέλεσης:

$$T(n) = \begin{cases} 0 & \text{if } n = 0, \\ 2T(n-1) + 1 & \text{if } n \geq 1. \end{cases}$$

Η επίλυση της παραπάνω αναδρομικής εξίσωσης θα γίνει με τη μέθοδο των *αθροιζομένων*

παραγόντων ως εξής:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \quad (\times 2^0) \\ T(n-1) &= 2T(n-2)+1 \quad (\times 2^1) \\ T(n-2) &= 2T(n-3)+1 \quad (\times 2^2) \\ &\dots \\ T(2) &= 2T(1)+1 \quad (\times 2^{n-2}) \\ T(1) &= 2T(0)+1 \quad (\times 2^{n-1}) \end{aligned}$$

Αθροίζοντας κατά μέλη λαμβάνουμε

$$T(n) = 2^n T(0) + (1 + \ldots + 2^{n-1}) = 2^n 0 + \frac{2^n - 1}{2-1}$$

και τελικά

$$T(n) = 2^n - 1, \qquad n \geq 0$$

Η γενική μορφή μιας αναδρομικής εξίσωσης όπως αυτή η οποία περιγράφει το χρόνο εκτέλεσης του αλγορίθμου Hanoi είναι η εξής:

$$a(n)T_n = b(n)T_{n-1} + c(n)$$

με T_0 σταθερά και $a(n), b(n), c(n)$ συναρτήσεις του n.

Άσκηση: Να επιλυθεί η αναδρομική εξίσωση $T(n) = T(n-1)+2$ με $T(1) = 1$.
Λύση:

$$\begin{aligned} T(n) &= 2 + T(n-1) \\ &= 2 + 2 + T(n-2) \\ &\dots \\ &= 2 + \ldots + 2 + T(1) \\ &= (n-1) \times 2 + 1 \\ &\leq 2n \end{aligned}$$

$$T(n) = \Theta(n)$$

2.1.4 Η ακολουθία Fibonacci

Ένα άλλο κλασσικό μαθηματικό πρόβλημα το οποίο εμπεριέχει την έννοια της αναδρομής είναι η ακολουθία Fibonacci. Η ακολουθία αυτή ορίζεται ως εξής:

$$T(n) = \begin{cases} 1 & \text{if } n = 0,\ n = 1, \\ T(n-1) + T(n-2) & \text{if } n > 1. \end{cases}$$

Πρακτικά κάθε αριθμός της ακολουθίας Fibonacci προκύπτει ως το άθροισμα των προηγουμένων δύο αριθμών της ακολουθίας. Για παράδειγμα οι 13 πρώτοι αριθμοί της ακολουθίας είναι: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233. Ένας επαναληπτικός αλγόριθμος ο οποίος μπορεί να χρησιμοποιηθεί για την εύρεση του $n-$οστού αριθμού της ακολουθίας είναι ο εξής:

FIBONACCI (n)
1. $f0 = 1$
2. $f1 = 1$
3. **for** $i = 3$ **to** n **do**
4. $\quad temp = f0 + f1$
5. $\quad f0 = f1$
6. $\quad f1 = temp$
7. **end for**
8. **return** f

Σχήμα 2.6: Επαναληπτικός Αλγόριθμος εύρεσης του n-οστού αριθμού Fibonacci

Για την εύρεση της ασυμπτωτικής πολυπλοκότητας του παραπάνω αλγορίθμου μπορούμε να παρατηρήσουμε ότι ο κύριος βρόγχος εκτελείται $n-2$ φορές, άρα η πολυπλοκότητα είναι της τάξης $O(n)$.

Μπορεί όμως να χρησιμοποιηθεί και ένας περισσότερο προφανής και πιο απλός αλγόριθμος ο οποίος βασίζεται στον αναδρομικό ορισμό της ακολουθίας.

Ο αλγόριθμος αυτός έχει ως εξής:

FIBONACCI (n)
1. $u = 1$
2. $f = 1$
3. **if** $n \leq 1$ **then**
4. $\quad$ **return** 1
5. **else**
6. $\quad$ **return** FIBONACCI$(n-1)$ + FIBONACCI$(n-2)$
7. **end if**

Σχήμα 2.7: Αναδρομικός Αλγόριθμος εύρεσης του n-οστού αριθμού Fibonacci

Ας θεωρήσουμε ένα παράδειγμα υπολογισμού κατά το οποίο ζητάμε να βρούμε τον 5ο αριθμό της ακολουθίας. Η εκτέλεση του αναδρομικού αλγορίθμου μπορεί να ιδωθεί ως ένα δένδρο αναδρομικών κλήσεων (άσκηση). Για την εύρεση της πολυπλοκότητας έχουμε:

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) \\ &= 1 + 1 + T(n-2) + T(n-3) + T(n-2) \quad (n \geq 3) \\ T(n) &\geq 2T(n-2) \\ &\geq 2^2 T(n-4) \\ &\dots \\ &\geq 2^{\frac{n}{2}} T(n - 2\frac{n}{2}) \end{aligned}$$

Άρα θα έχουμε

$$T(n) \geq 2^{\frac{n}{2}} T(0)$$

εάν n άρτιος και

$$T(n) \geq 2^{\frac{n-1}{2}} T(1)$$

εάν n περιττός, οπότε τελικά

$$T(n) \geq 2^{\frac{n}{2}},$$

δεδομένου ότι $T(0) = T(1) = 1$.

Παρατηρούμε λοιπόν ότι ο εκθετικός αλγόριθμος, αν και πολύ φυσικός στην σκέψη και εύκολος στην υλοποίηση έχει εκθετική πολυπλοκότητα. Το γεγονός αυτό οφείλεται στο ότι υπάρχει επανυπολογισμός αρκετών από τους ενδιάμεσους όρους, όπως μπορεί να φανεί και στο δένδρο υπολογισμών.

2.1.5 Ο αλγόριθμος Min-Max

Το τελευταίο πρόβλημα το οποίο θα θεωρήσουμε είναι το πρόβλημα της εύρεσης του μέγιστου και του ελάχιστου στοιχείου ενός πίνακα A με n στοιχεία. Ένας απλός επαναληπτικός αλγόριθμος για την επίλυση του προβλήματος είναι αυτός του σχήματος 2.8.

MinMax (A)
1. $min = a_1$
2. $max = a_1$
3. **for** $i = 2$ **to** n **do**
4. **if** $a_i > max$ **then**
5. $max = a_i$
6. **else if** $a_i < min$ **then**
7. $min = a_i$
8. **end if**
9. **end for**

Σχήμα 2.8: Αλγόριθμος εύρεσης του μέγιστου και του ελάχιστου στοιχείου ενός πίνακα

Ο παραπάνω αλγόριθμος στη χείριστη περίπτωση, όπου τα στοιχεία του πίνακα θα είναι ταξινομημένα σε φθίνουσα σειρά θα έχει ακριβή πολυπλοκότητα χρόνου $3(n-1)$, αν συμπεριλάβουμε και τη σύγκριση για να υλοποιήσουμε την επανάληψη for. Στην περίπτωση όπου τα στοιχεία του πίνακα είναι ταξινομημένα σε αύξουσα σειρά η ακριβής πολυπλοκότητα θα είναι $n-1$, άρα συνολικά θα έχουμε $\Theta(n)$.

Ας θεωρήσουμε τώρα όμως τον αναδρομικό αλγόριθμο του σχήματος 2.9 για την επίλυση του προβλήματος.

MinMax (A, i, j, min, max)

```
 1.  if i ≥ j − 1 then
 2.      if a_i < a_j then
 3.          max = a_j
 4.          min = a_i
 5.      else
 6.          max = a_i
 7.          min = a_j
 8.      end if
 9.  else
10.      m = ⌊(i + j)/2⌋
11.      MinMax(A, i, m, min_1, max_1)
12.      MinMax(A, m, j, min_2, max_2)
13.      min = fmin(min_1, min_2)
14.      max = fmax(max_1, max_2)
15.  end if
```

Σχήμα 2.9: Αναδρομικός Αλγόριθμος εύρεσης του μέγιστου και του ελάχιστου στοιχείου ενός πίνακα

Για την εύρεση της πολυπλοκότητας του αναδρομικού αλγορίθμου ας σημειώσουμε τα εξής:

- Ο έλεγχος $i \geq j-1$ καλύπτει τις δύο περιπτώσεις $i = j$ και $i = j-1$
- Οι συναρτήσεις fmax() και fmin() απαιτούν μία σύγκριση η κάθε μία για την εύρεση του μέγιστου ή του ελάχιστου (αντίστοιχα) μεταξύ δύο στοιχείων.

Η πολυπλοκότητα δίδεται από την παρακάτω αναδρομική σχέση:

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \text{ or } n = 1, \\ 2T(\frac{n}{2}) + 3 & \text{if } n > 2. \end{cases}$$

Για την επίλυσή της θα έχουμε (βλ. 2.1):

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + 3 \\
&= 2[2T(\frac{n}{2^2}) + 3] + 3 = 2^2T(\frac{n}{2^2}) + 2 \times 3 + 3 \\
&= 2^2[2T(\frac{n}{2^3}) + 3] + 2 \times 3 + 3 = 2^3T(\frac{n}{2^3}) + 2^2 \times 3 + 2 \times 3 + 3 \\
&= \ldots \\
&= 2^{k-1}T(\frac{n}{2^{k-1}}) + 2^{k-2} \times 3 + \ldots + 2 \times 3 + 3
\end{aligned}$$

και επομένως αν $n = 2^k$ έχουμε

$$\begin{aligned}
T(n) &= \frac{n}{2} \times T(2) + [\frac{2^{k-1} - 1}{2 - 1}] \times 3 \\
&= n + 3 \times 2^{k-1} - 3 \\
&= n + 3\frac{n}{2} - 3 \\
&= 5\frac{n}{2} - 3
\end{aligned}$$

Παρατηρούμε δηλαδή ότι στο συγκεκριμένο πρόβλημα η ακριβής πολυπλοκότητα του αναδρομικού αλγορίθμου είναι μικρότερη από την αντίστοιχη του επαναληπτικού, σε αντίθεση με το πρόβλημα των αριθμών Fibonacci που είδαμε προηγούμενα.

2.1.6 QuickSort

Ο Αλγόριθμος Quicksort είναι ένας αλγόριθμος ταξινόμησης που δημιουργήθηκε απο τον C. A. R. Hoare, με πολυπλοκότητα στη χείριστη περίπτωση $\Theta(n^2)$ σε είσοδο ενός πίνακα n αριθμών. Παρόλο που η πολυπλοκότητά του στη χείριστη περίπτωση είναι μεγάλη, ο Quicksort είναι συχνά η καλύτερη πρακτική επιλογή, γιατί είναι εκπληκτικά αποδοτικός στη μέση περίπτωση: Ο αναμενόμενος χρόνος είναι $\Theta(nlogn)$ και οι σταθεροί όροι που κρύβονται στο $\Theta(nlogn)$ είναι αρκετά μικροί. Ο αλγόριθμος έχει επίσης το πλεονέκτημα να ταξινομεί στον ίδιο πίνακα, χωρίς να χρειάζεται παραπάνω χώρο απ΄ οσο χρειάζεται η είσοδος.

Η QuickSort, οπώς και η MergeSort, ακολουθεί το παράδειγμα Divide-and-Conquer. Η διαδικασία Divide-and-Conquer 3 βημάτων για την ταξινόμηση ένος πίνακα $A[p \ldots r]$ είναι η εξής:

Divide: Διαμέριση του πίνακα $A[p \ldots r]$ σε δύο, πιθανώς κενούς, υποπίνακες $A[p \ldots q-1]$ και $A[q+1 \ldots r]$, τέτοιους ώστε κάθε στοιχείο του $A[p \ldots q-1]$ είναι μικρότερο ή ίσο από το $A[q]$ το οποίο με τη σειρά του είναι μικρότερο ή ίσο από κάθε στοιχείο του πίνακα $A[q+1 \ldots r]$. Διαμερίζοντας, βρίσκουμε τη θέση του $A[q]$ στον πίνακα $A[p \ldots r]$.

Conquer: Ταξινόμηση των δύο υποπινάκων $A[p \ldots q-1]$ και $A[q+1 \ldots r]$ με αναδρομικές κλήσεις της QuickSort.

Combine: Αφού οι υποπίνακες ταξινομούνται στον ίδιο πίνακα δε χρειάζεται παραπάνω εργασία για να συνδυάσουμε τις λύσεις με τους υποπίνακες. Ο συνολικός πίνακας $A[p \dots r]$ είναι ταξινομημένος μετά και την τελευταία αναδρομική κλήση της QuickSort.

Η διαδικασία του σχήματος 2.10 υλοποιεί την QuickSort.

QUICKSORT (A, p, r)
1. **if** $p < r$ **then**
2. $\quad q =$PARTITION(A, p, r)
3. $\quad$QUICKSORT $(A, p, q-1)$
4. $\quad$QUICKSORT $(A, q+1, r)$
5. **end if**

Σχήμα 2.10: Αλγόριθμος QUICKSORT

Το κλειδί του αλγορίθμου είναι η διαδικασία της διαμέρισης (Partition) η οποία επανατοποθετεί τα στοιχεία του πίνακα σύμφωνα με το στοιχείο περιστροφής (pivot) που έχει επιλεγεί: Τα μεγαλύτερα από αυτο μπαίνουν στα δεξιά του και τα μικρότερα στα αριστερά του, όπως φαίνεται στην υλοποίησή του στο σχήμα 2.11.

PARTITION (A, p, r)
1. $x = A[r]$
2. $i = p - 1$
3. **for** $j = p$ **to** $r - 1$ **do**
4. $\quad$ **if** $A[j] \leq x$
5. $\quad\quad i = i + 1$
6. $\quad\quad swap(A[i],\ A[j])$
7. $\quad$ **end if**
8. **end for**
9. $swap(A[i+1],\ A[r])$
10. return $i + 1$

Σχήμα 2.11: Διαδικασία διαμέρισης με βάση το οδηγό στοιχείο.

Ο χρόνος εκτέλεσης της QuickSort εξαρτάται από το αν η διαμέριση είναι ισορροπημένη ή όχι. Αυτό εξαρτάται από το στοίχειο που θα επιλεγεί ως οδηγό στοιχείο κατά τη διαμέριση. Αν η διαμέριση είναι ισορροπημένη, τότε η QuickSort εκτελείται ασυμπτωτικά το ίδιο γρήγορα με τη MergeSort. Αν η διαμέριση δεν είναι ισορροπημένη, τότε η QuickSort θα εκτελεστεί αργά.

Διαμέριση Χείριστης Περίπτωσης

Η χείριστη περίπτωση προκύπτει όταν η διαμέριση χωρίζει τον πίνακα μεγέθους n σε 2 υποπίνακες μεγέθους $n-1$ και 0 στοιχείων. Ας υποθέσουμε ότι αυτο συμβαίνει σε κάθε αναδρομική κλήση. Η διαμέριση κοστίζει $\Theta(n)$ χρόνο. Έτσι η αναδρομική εξίσωση για αυτή την περίπτωση είναι

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Επομένως στην χείριστη περίπτωση, όταν δηλαδή η διαμέριση είναι η χειρότερη δυνατή, ο χρόνος είναι $T(n) = \Theta(n^2)$. Αξίζει να σημειωθεί ότι η παραπάνω περίπτωση θεωρείται εκφυλισμένη και συμβαίνει μόνο όταν η είσοδος είναι ένας ήδη ταξινομημένος πίνακας.

Διαμέριση Βέλτιστης Περίπτωσης

Στον πιο ευμενή διαχωρισμό, η διαδικασία Partition δημιουργεί δύο υποπροβλήματα κάθε ένα με μέγεθος όχι μεγαλύτερο από $n/2$, αφού το ένα υποπρόβλημα θα είναι μεγέθους $\lfloor n/2 \rfloor$ και το άλλο μεγέθους $\lceil n/2 \rceil - 1$. Σ′ αυτη την περίπτωση η QuickSort εκτελείται πολύ πιο γρήγορα. Η αναδρομική εξίσωση είναι $T(n) \leq 2T(n/2) + \Theta(n)$, η οποία ανήκει στην 2η περίπτωση του Master Theorem και έχει λύση την $T(n) = O(nlogn)$.

Διαμέριση Μέσης Περίπτωσης

Θεωρούμε όλες τις θέσεις για τοποθέτηση του οδηγού στοιχείου v ισοπίθανες ($= \frac{1}{n}$). Ο μέσος αριθμός συγκρίσεων T_n για $n \geq 2$ υπολογίζεται ως εξής:

$$T_0 = T_1 = 0$$

και

$$T_n = n + 1 + \frac{1}{n} \sum_{k=1}^{n} (T_{k-1} + T_{n-k})$$

θεωρώντας τις συγκρίσεις μεταξύ των στοιχείων μόνο.
Λόγω συμμετρίας έχουμε:

$$T_n = n + 1 + \frac{2}{n} \sum_{k=1}^{n} T_{k-1} \Rightarrow nT_n = n^2 + n + 2 \sum_{k=1}^{n} T_{k-1}$$

Για $n-1$ έχουμε:

$$T_{n-1} = n + \frac{2}{n-1} \sum_{k=1}^{n-1} T_{k-1} \Rightarrow (n-1)T_{n-1} = n^2 - n + 2 \sum_{k=1}^{n-1} T_{k-1}$$

Αφαιρώντας έχουμε μετά την απλοποίηση:

$$nT_n = (n+1)T_{n-1} + 2n$$

Διαιρώντας με $n(n+1)$ παίρνουμε:

$$\frac{T_n}{n+1} = \frac{T_{n-1}}{n} + \frac{2}{n+1} = \frac{c}{3} + \sum_{k=3}^{n} \frac{2}{k+1}$$

Χρησιμοποιώντας την προσέγγιση:

$$\frac{T_n}{n+1} \simeq 2\sum_{k=1}^{n} \frac{1}{k} \simeq 2\int_1^n \frac{1}{x}dx = 2\ln n$$

παίρνουμε το αποτέλεσμα:

$$T_n = 1.38n\log n$$

2.2 Ασυμπτωτική προσέγγιση μερικών αθροισμάτων

Όταν μια συνάρτηση είναι μονότονη, τότε έχουμε τη δυνατότητα να φράξουμε το άθροισμά της από το ολοκλήρωμά της. Εάν επιπλέον η συνάρτηση μεταβάλλεται αργά, τότε το άθροισμά της και το ολοκλήρωμά της έχουν την *ίδια* τάξη μεγέθους ασυμπτωτικά.
Για κάθε μονότονη φθίνουσα συνάρτηση ισχύει η ακόλουθη σχέση (βλ. σχήμα 2.12):

$$\int_p^{q+1} f(x)dx \leq \sum_{i=p}^{q} f(i) \leq \int_{p-1}^{q} f(x)dx$$

Για να δούμε την χρησιμότητα της παραπάνω σχέσης ας θεωρήσουμε την συνάρτηση

$$H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$$

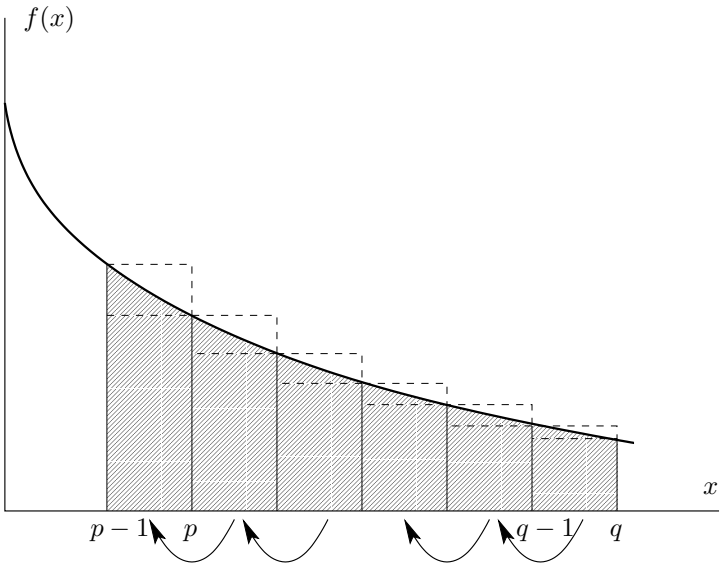
η οποία μας δίδει τον n-οστό αρμονικό αριθμό και ας εφαρμόσουμε την πιο πάνω σχέση. Εάν θέσουμε $f(x) = \frac{1}{x}$ τότε θα έχουμε

$$\int_2^{n+1} \frac{1}{x}dx \leq \sum_{i=2}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x}dx$$

και επομένως, για τον n-οστό αρμονικό αριθμό θα ισχύει

$$\ln(n+1) - \ln(2) + 1 \leq H_n \leq \ln(n) + 1$$

δηλαδή $H_n = \Theta(\ln n)$.



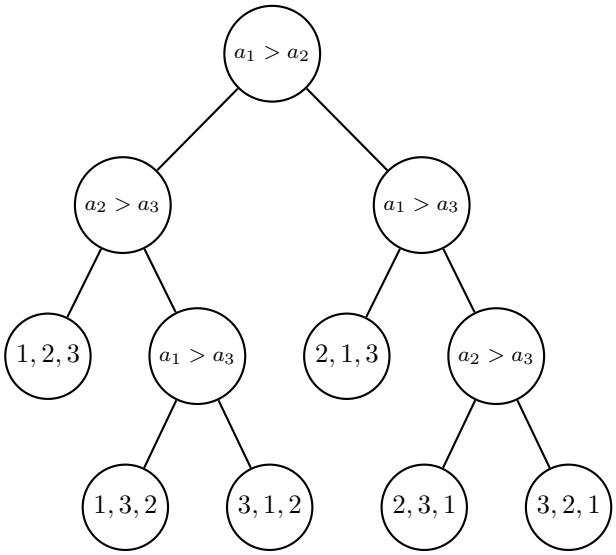
Σχήμα 2.12: Προσέγγιση μονότονης συνάρτησης από το ολοκλήρωμά της.

2.3 Δένδρα Απόφασης

Τα δένδρα απόφασης είναι κλασσικά δυαδικά δένδρα στα οποία ισχύουν μερικές ενδιαφέρουσες ιδιότητες. Κάθε εσωτερικός κόμβος ενός δένδρου απόφασης έχει ως ρόλο να θέτει μια ερώτηση όσο αφορά στη σύγκριση δύο στοιχείων. Το αριστερό παιδί του κόμβου αντιστοιχεί σε *αρνητική* απάντηση της ερώτησης. Το δεξιό παιδί του κόμβου αντιστοιχεί σε *θετική* απάντηση. Τα φύλλα ενός δένδρου απόφασης αντιπροσωπεύουν την μετάθεση που πρέπει να γίνει έτσι ώστε να επιτύχουμε τον ταξινομημένο πίνακα. Στο σχήμα 2.13 φαίνεται ένα παράδειγμα δένδρου απόφασης.

Κάθε δένδρο απόφασης το οποίο προορίζεται για την ταξινόμηση n στοιχείων έχει $n!$ φύλλα, όσες δηλαδή και οι δυνατές μεταθέσεις των n στοιχείων. Το ύψος h ενός τέτοιου δένδρου είναι $h \geq \lceil \log_2(n!) \rceil$. Αναπτύσσοντας ασυμπτωτικά το $n!$ από τον τύπο του Stirling θα έχουμε

$$n! = \sqrt{(2\pi n)} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + o(\frac{1}{n})\right)$$



Σχήμα 2.13: Παράδειγμα ενός δένδρου απόφασης.

και επομένως

$$\log_2(n!) = \log_2\left(\sqrt{(2\pi n)}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}+o(\frac{1}{n})\right)\right) =$$

$$n\log_2 n - n\log_2 e + \frac{1}{2}\log_2 n + \frac{1}{2}\log_2 2\pi + \log_2\left(1+\frac{1}{12n}+o\left(\frac{1}{n}\right)\right)$$

δηλαδή τελικά $h = \Omega(n \log n)$.

Όπως είναι προφανές, ακολουθώντας μια διαδρομή στο δένδρο και απαντώντας στις ερωτήσεις μπορούμε να έχουμε μια ταξινόμηση των n στοιχείων. Επομένως η ταξινόμηση n στοιχείων βασισμένη σε συγκρίσεις των στοιχείων ανά δύο, απαιτεί $\Omega(n \log n)$ συγκρίσεις. Συμπεραίνουμε ότι οι αλγόριθμοι ταξινόμησης με σωρό, με συγχώνευση και με διχοτομική εισαγωγή είναι βέλτιστοι αλγόριθμοι στην χείριστη περίπτωση. Αντίθετα ο αλγόριθμος QuickSort δεν είναι βέλτιστος στην χείριστη περίπτωση, αλλά είναι βέλτιστος προς την κατά μέσο όρο πολυπλοκότητα.

2.4 The Master Theorem

Στην παράγραφο αυτή θα περιγράψουμε μια μέθοδο γνωστή και ως *Master* μέθοδο ή αλλιώς *Master Theorem* η οποία μας βοηθά στο να επιλύουμε γρήγορα

αναδρομικές εξισώσεις της μορφής

$$T(n) = aT(\frac{n}{b}) + f(n)$$

όπου $a \geq 1$ και $b > 1$. Πριν υπεισέλθουμε στην περιγραφή και τις λεπτομέρειες της μεθόδου θα κάνουμε μερικές υπενθυμίσεις όσο αφορά σε βασικές σχέσεις που ισχύουν στις αναδρομικές εξισώσεις καθώς και σε θέματα συμβολισμού.

Για τη συνέχεια θα χρησιμοποιήσουμε τον συμβολισμό του παρακάτω πίνακα.

$$\begin{aligned}
\lg n + k &= (\lg n) + k \\
\lg(n) &= \log_2(n) \quad \text{(δυαδικός λογάριθμος)} \\
\ln n &= \log_e(n) \quad \text{(φυσικός λογάριθμος)} \\
\lg^k n &= (\lg n)^k \quad \text{(πολυλογάριθμος)} \\
\lg \lg n &= \lg(\lg n) \quad \text{(σύνθεση)}
\end{aligned}$$

Για όλους τους πραγματικούς αριθμούς $a > 0,\ b > 0,\ c > 0$ ισχύουν οι παρακάτω σχέσεις:

$$\begin{aligned}
a &= b^{\log_b a} \\
n &= 2^{\log_2 n} \\
\log_c(ab) &= \log_c a + \log_c b \\
\log_b a^n &= n \log_b a \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b \frac{1}{a} &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b c} &= c^{\log_b a}
\end{aligned}$$

Τέλος, υπενθυμίζουμε μια σειρά από σχέσεις - ιδιότητες οι οποίες ισχύουν και οι οποίες θα φανούν χρήσιμες στη συνέχεια.

- Μία συνάρτηση $f(n)$ είναι πολυωνυμικά φραγμένη αν $f(n) = O(n^k)$ για κάποια σταθερά k.

- Κάθε εκθετική συνάρτηση με βάση μεγαλύτερη του 1 αυξάνεται γρηγορότερα από κάθε πολυωνυμική συνάρτηση. Δηλαδή, έχουμε για a και b σταθερές, με $a > 1$, ότι:
$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \Rightarrow n^b = o(a^n).$$

- Η $f(n)$ είναι πολυλογαριθμικά φραγμένη αν $f(n) = O(\log^k n)$ για κάποια σταθερά k.

- Κάθε θετική πολυωνυμική συνάρτηση αυξάνεται γρηγορότερα από κάθε πολυλογαριθμική συνάρτηση. Δηλαδή,έχουμε για k και b θετικές σταθερές ότι:

$$\lim_{n\to\infty} \frac{\log^k n}{n^b} = 0 \Rightarrow \log^k(n) = o(n^b).$$

2.4.1 Το Master θεώρημα

Ας θεωρήσουμε την αναδρομική εξίσωση

$$T(n) = aT(\frac{n}{b}) + f(n)$$

όπου $a \geq 1$ και $b > 1$ και $f(n)$ μια ασυμπτωτική θετική συνάρτηση. Για να είναι καλώς ορισμένη η παραπάνω αναδρομική εξίσωση και επειδή ο λόγος $\frac{n}{b}$ μπορεί να μην είναι ακέραιος, με το $\frac{n}{b}$ εννοούμε $\lfloor \frac{n}{b} \rfloor$ ή $\lceil \frac{n}{b} \rceil$.

Θεώρημα 1. Η $T(n)$ φράσσεται ασυμπτωτικά ως εξής:

1. Εάν $f(n) = O(n^{\log_b a - \epsilon})$ για κάποια σταθερά $\epsilon > 0$ τότε $T(n) = \Theta(n^{\log_b a})$.
2. Εάν $f(n) = \Theta(n^{\log_b a})$ τότε $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Εάν $f(n) = \Omega(n^{\log_b a + \epsilon})$ για κάποια σταθερά $\epsilon > 0$ και εάν $af(\frac{n}{b}) \leq cf(n)$ για κάποια σταθερά $c < 1$ και όλα τα αρκετά μεγάλα n τότε $T(n) = \Theta(f(n))$.

Σε κάθε μία από τις τρεις περιπτώσεις του θεωρήματος συγκρίνουμε τη συνάρτηση $f(n)$ με τη συνάρτηση $n^{\log_b a}$. Διαισθητικά, η λύση της αναδρομικής εξίσωσης καθορίζεται από την μεγαλύτερη από τις δύο αυτές συναρτήσεις. Εάν, όπως στην περίπτωση 1, η συνάρτηση $n^{\log_b a}$ είναι η μεγαλύτερη τότε η λύση είναι $T(n) = \Theta(n^{\log_b a})$. Εάν συμβαίνει το αντίθετο, όπως στην περίπτωση 3, η λύση είναι $T(n) = \Theta(f(n))$. Τέλος, εάν όπως στην περίπτωση 2, οι δύο συναρτήσεις είναι της ίδιας τάξης μεγέθους, τότε απλά πολλαπλασιάζουμε με ένα λογαριθμικό παράγοντα και η λύση είναι $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Πέρα από την παραπάνω διαίσθηση αξίζει να παρατηρήσουμε ορισμένες τεχνικές λεπτομέρειες. Στην πρώτη περίπτωση του θεωρήματος, η συνάρτηση $f(n)$ δεν πρέπει απλά να είναι μικρότερη από την $n^{\log_b a}$, αλλά πρέπει να είναι *πολυωνυμικά* μικρότερη. Δηλαδή η $f(n)$ πρέπει να είναι ασυμπτωτικά μικρότερη από την $n^{\log_b a}$ κατά ένα παράγοντα n^ϵ για κάποιο σταθερό $\epsilon > 0$. Επίσης, στην τρίτη περίπτωση του θεωρήματος, η $f(n)$ πρέπει επίσης να είναι *πολυωνυμικά* μεγαλύτερη από την $n^{\log_b a}$, αλλά επίσης πρέπει να ικανοποιεί και την συνθήκη κανονικότητας $af(\frac{n}{b}) \leq cf(n)$. Αξίζει να σημειώσουμε ότι η τελευταία συνθήκη ικανοποιείται από τις περισσότερες πολυωνυμικά φραγμένες συναρτήσεις.

Πριν μελετήσουμε κάποια παραδείγματα εφαρμογής του θεωρήματος, είναι σημαντικό να κατανοήσουμε ότι οι τρεις περιπτώσεις δεν καλύπτουν όλες τις πιθανές εκδοχές για τη συνάρτηση $f(n)$. Μεταξύ των περιπτώσεων 1 και 2 υπάρχει η περίπτωση κατά την οποία η $f(n)$ είναι μικρότερη από την $n^{\log_b a}$, αλλά όχι πολυωνυμικά μικρότερη. Ομοίως, μεταξύ των περιπτώσεων 2 και 3 υπάρχει και η περίπτωση κατά την οποία η $f(n)$ είναι μεγαλύτερη από την $n^{\log_b a}$, αλλά όχι

πολυωνυμικά μεγαλύτερη. Εάν λοιπόν η συνάρτηση $f(n)$ συμπίπτει με κάποια από αυτές τις ενδιάμεσες περιπτώσεις, ή εάν δεν ικανοποιείται η συνθήκη κανονικότητας της περίπτωσης 3, το θεώρημα προφανώς δεν μπορεί να εφαρμοστεί.

2.4.2 Παραδείγματα

Για να χρησιμοποιήσουμε τη Master μέθοδο απλά αποφασίζουμε σε ποια από τις τρεις περιπτώσεις εμπίπτει η αναδρομική εξίσωση που καλούμαστε να επιλύσουμε.

1. $T(n) = 9T(\frac{n}{3}) + n$.
   Σε αυτή την εξίσωση έχουμε $a = 9$, $b = 3$ και $f(n) = n$. Συνεπώς $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Αφού λοιπόν $f(n) = O(n^{\log_3 9-\epsilon})$ όπου $\epsilon = 1$, μπορούμε να εφαρμόσουμε την περίπτωση 1 του θεωρήματος και να συμπεράνουμε ότι η λύση είναι $T(n) = \Theta(n^2)$.

2. $T(n) = T(\frac{2n}{3}) + 1$.
   Εδώ έχουμε $a = 1$, $b = \frac{3}{2}$ και $f(n) = 1$. Συνεπώς $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Οπότε αφού $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, εφαρμόζουμε την περίπτωση 2 και η λύση είναι $T(n) = \Theta(\lg n)$.

3. $T(n) = 3T(\frac{n}{4}) + n \log n$. Σε αυτή την εξίσωση έχουμε $a = 3$, $b = 4$ και $f(n) = n \log n$ και $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Δηλαδή $f(n) = \Omega(n^{\log_4 3+\epsilon})$ με $\epsilon \approx 0.2$. Επιπρόσθετα $af(\frac{n}{b}) = 3(\frac{n}{4}) \log(\frac{n}{4}) \leq \frac{3}{4} n \log n = cf(n)$ με $c = \frac{3}{4}$. Συνεπώς μπορούμε να εφαρμόσουμε την περίπτωση 3 και άρα η λύση είναι $T(n) = \Theta(n \log n)$.

4. $T(n) = 2T(\frac{n}{2}) + n \log n$.
   Εδώ έχουμε $a = 2$, $b = 2$ και $f(n) = n \log n$. Συνεπώς $n^{\log_b a} = n$. Αλλά ενώ $n \log n > n$ ο λόγος $\frac{n \log n}{n} = \log n$ είναι ασυμπτωτικά μικρότερος από το n^ϵ για κάθε θετική σταθερά ϵ. Κατά συνέπεια η $f(n)$ δεν είναι πολυωνυμικά μεγαλύτερη, κατά ένα παράγοντα n^ϵ, από την συνάρτηση $n^{\log_b a} = n$ και η περίπτωση 3 του θεωρήματος δεν μπορεί να εφαρμοστεί.

2.5 Αναδρομικές γραμμικές εξισώσεις

Οι αναδρομικές εξισώσεις χωρίζονται σε κατηγορίες ανάλογα με

1. τον τύπο της συνάρτησης f. Η συνάρτηση f μπορεί να είναι γραμμικός συνδυασμός των $T(p)$, με συντελεστές σταθερούς ή μεταβλητούς ή πωλυώνυμα $T(p)$ κ.τ.λ.

2. το σύνολο τιμών p που εμπλέκονται για τον υπολογισμό του $T(n)$. Πιο συγκεκριμένα έχουμε:

   - $T(n) = f(T(n-1))$ εξίσωση τάξης 1
   - $T(n) = f(T(n-1), \ldots, T(n-k))$ εξίσωση τάξης k για k φιξαρισμένο
   - $T(n) = f(\{T(p); \forall p < n\})$ πλήρης εξίσωση

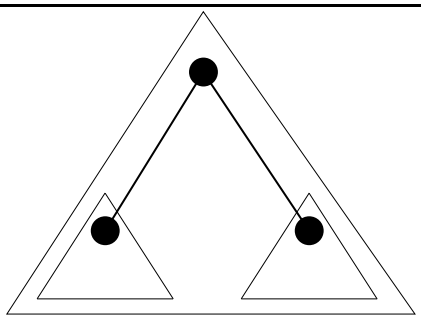
2.5.1 Γραμμικές αναδρομές τάξης k

Οι γραμμικές αναδρομές τάξης k είναι την μορφής

$$T(n) = f(n, T(n-1), \ldots, T(n-k)) + g(n)$$

όπου $k \geq 1$ σταθερά, f γραμμικός συνδυασμός των $T(i)$ για $n-k \leq i \leq n-1$ και g οποιαδήποτε συνάρτηση του n.
Παράδειγμα: Πλήρες δυαδικό δένδρο. Ο αριθμός κόμβων a_n του πλήρους δυαδικού δένδρου ύψους n είναι ίσος με τον αριθμό των κόμβων των δύο υποδένδρων ύψους $n-1$ συν 1 κόμβο ο οποίος είναι η ρίζα. Άρα (βλ. σχήμα 2.14)

$$a_n = 2a_{n-1} + 1 \ (n \geq 1, k = 1, a_0 = 1)$$



Σχήμα 2.14: Πλήρες δυαδικό δένδρο

2.5.2 Αναδρομές διαμερίσεων

Ο γενικός τύπος των αναδρομικών εξισώσεων αυτής της κατηγορίας είναι:

$$T(n) = aT(\frac{n}{b}) + d(n)$$

όπου a, b ακέραιες σταθερές και $d(n)$ οποιαδήποτε συνάρτηση. Τέτοιου τύπου αναδρομικές εξισώσεις προκύπτουν συνήθως από αναδρομικούς αλγορίθμους οι οποίοι χρησιμοποιούν διαμερίσεις.
Παράδειγμα: Ο αλγόριθμος Mergesort. Ο αριθμός των συγκρίσεων του αλγορίθμου Mergesort $T(n)$ δίδεται από την εξίσωση

$$T(n) = 2T(\frac{n}{2}) + n - 1 \quad n \geq 2$$
$$T(1) = 0$$

όπου $n-1$ είναι το κόστος της συγχώνευσης των δύο υποπινάκων.

2.5.3 Αναδρομές πλήρεις, γραμμικές ή πολυωνυμικές

Ο γενικός τύπος των αναδρομικών εξισώσεων αυτής της κατηγορίας είναι:

$$T(n) = f(n, T(n-1), T(n-2), \ldots, T(0)) + g(n)$$

όπου f μια συνάρτηση γραμμική ή πολυωνυμική των $T(i)$ και $g(n)$ οποιαδήποτε συνάρτηση.

Παράδειγμα: Αναδρομή πλήρης πολυωνυμική. Ο αριθμός δυαδικών δένδρων με n κόμβους δίδεται από την παρακάτω αναδρομική εξίσωση

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$
$$b_0 = 1$$

Παράδειγμα: Αναδρομή πλήρης γραμμική. Η κατά μέσο όρο πολυπλοκότητα της Quicksort δίδεται από την αναδρομική εξίσωση

$$C_n = (n+1) + \frac{2}{n} \sum_{k=1}^{n-1} C_k, \quad n \geq 2$$
$$C_0 = C_1 = 0$$

2.5.4 Παραδείγματα Γραμμικών Αναδρομικών Εξισώσεων

1. Να επιλυθεί η παρακάτω αναδρομική εξίσωση, (αριθμός κόμβων ενός πλήρους δυαδικού δέντρου ύψους n)

$$a_n = a_{n-1} + 2^n, \quad n \geq 1$$
$$a_0 = 1$$

Λύση:

$$a_n = a_{n-1} + 2^n$$
$$a_{n-1} = a_{n-2} + 2^{n-1}$$
$$a_{n-2} = a_{n-3} + 2^{n-2}$$
$$\ldots$$
$$a_2 = a_1 + 2^2$$
$$a_1 = a_0 + 2^1$$

Αθροίζοντας κατά μέλη λαμβάνουμε:

$$a_n = a_0 + \sum_{i=1}^{n} 2^i, \quad n \geq 1$$
$$a_n = 1 + \frac{2^{n+1} - 2}{2 - 1}$$
$$a_n = 2^{n+1} - 1$$

2. Να επιλυθεί η προηγούμενη αναδρομική εξίσωση με τη μέθοδο αθροιζόμενων παραγόντων.

Λύση:

$$\begin{aligned} a_n &= 2a_{n-1} + 1 \quad (\times 2^0) \\ a_{n-1} &= 2a_{n-2} + 1 \quad (\times 2^1) \\ a_{n-2} &= 2a_{n-3} + 1 \quad (\times 2^2) \\ &\dots \\ a_2 &= 2a_1 + 1 \quad (\times 2^{n-2}) \\ a_1 &= 2a_0 + 1 \quad (\times 2^{n-1}) \end{aligned}$$

Χρησιμοποιώντας τη μέθοδο των αθροιζομένων παραγόντων λαμβάνουμε:

$$\begin{aligned} a_n &= 2^n a_0 + \sum_{i=0}^{n-1} 2^i \\ a_n &= 2^n + \frac{2^n - 1}{2 - 1} \\ a_n &= 2^{n+1} - 1, \quad n \geq 0 \end{aligned}$$

2.5.5 Γραμμικές Αναδρομικές Εξισώσεις με σταθερούς συντελεστές

Η γραμμική εξίσωση τάξης k με σταθερούς συντελεστές είναι της μορφής:

$$u_n + a_1 u_{n-1} + a_2 u_{n-2} + \ldots + a_k u_{n-k} = b(n)$$

με a_i σταθερές. Η εξίσωση

$$u_n + a_1 u_{n-1} + a_2 u_{n-2} + \ldots + a_k u_{n-k} = 0$$

καλείται αντίστοιχη ομογενής εξίσωση της αναδρομικής.

Το σύνολο των λύσεων της ομογενούς εξίσωσης σχηματίζει έναν διανυσματικό χώρο διάστασης μικρότερης είτε ίσης του k και κάποιες λύσεις είναι της μορφής:

$$u_n = r^n.$$

Αν αντικαταστήσουμε στην εξίσωση παίρνουμε την χαρακτηριστική εξίσωση της ομογενούς εξίσωσης:

$$r^k + a_1 r^{k-1} + \ldots + a_k = 0$$

Αν οι k ρίζες της εξίσωσης είναι διαφορετικές $r_1, r_2, \ldots, r_k$ τότε η λύση της ομογενούς εξίσωσης είναι:

$$u_n = \lambda_1 r_1^n + \lambda_2 r_2^n + \ldots + \lambda_k r_k^n$$

με λ_i σταθερές που βρίσκουμε με την βοήθεια των αρχικών τιμών $u_1, \ldots, u_{k-1}$.

Παράδειγμα: Να επιλυθεί η παρακάτω αναδρομική εξίσωση:

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2$$
$$F_0 = 0$$
$$F_1 = 1$$

Λύση: Έχουμε ότι:

$$F_n - F_{n-1} - F_{n-2} = 0.$$

Αν θεωρήσουμε ότι $u_n = r^n$ τότε η χαρακτηριστική εξίσωση της ομογενούς εξίσωσης γράφεται:

$$r^n - r^{n-1} - r^{n-2} = 0 \Rightarrow$$
$$r^{n-2}(r^2 - r - 1) = 0 \Rightarrow$$
$$r^2 - r - 1 = 0.$$

Οι λύσεις της εξίσωσης αυτής είναι:

$$r_1 = \frac{1+\sqrt{5}}{2}$$
$$r_2 = \frac{1-\sqrt{5}}{2}$$

Συνεπώς η λύση της ομογενούς εξίσωσης είναι της μορφής:

$$F_n = \lambda_1 r_1^n + \lambda_2 r_2^n$$

από: $F_0 = 0 \Leftrightarrow \lambda_1 = -\lambda_2$ και $F_1 = 1 \Leftrightarrow \lambda_1 r_1 + \lambda_2 r_2 = 1 \Leftrightarrow \lambda_1 r_1 - \lambda_1 r_2 = 1$ άρα τελικά $\lambda_1 = \frac{1}{r_1 - r_2}$ και $\lambda_2 = \frac{1}{r_2 - r_1}$ δηλαδή

$$\lambda_1 = \frac{1}{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}} = \frac{1}{\sqrt{5}}$$

$$\lambda_2 = -\frac{1}{\sqrt{5}}$$

Τελικά, η λύση είναι:

$$F_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

2.5.6 Εξισώσεις μετατρεπόμενες σε γραμμικές αναδρομές

Για την μετατροπή εξισώσεων σε γραμμικές αναδρομικές εξισώσεις και την επιτυχή επίλυσή τους εργαζόμαστε ως εξής: Ας υποθέσουμε την αναδρομική εξίσωση

$$T(n) = aT(\frac{n}{b}) + d(n)$$

με $n \geq 2$, a, b σταθερές και $T(1) = 1$. Το αρχικό πρόβλημα διάστασης n προκύπτει από το συνδυασμό a υποπροβλημάτων διάστασης $\frac{n}{b}$, ενώ $d(n)$ είναι το κόστος δημιουργίας των υποπροβλημάτων και το κόστος κατασκευής της λύσης από τις υπολύσεις. Επιπρόσθετα, ας υποθέσουμε ότι $n = b^k$ και άρα η αρχική εξίσωση γίνεται

$$T(b^k) = aT(b^{k-1}) + d(b^k).$$

Εάν τώρα θέσουμε $t_k = T(b^k)$ η εξίσωση η οποία προκύπτει από την αρχική με αντικατάσταση είναι γραμμική:

$$t_k = at_{k-1} + d(b^k)$$

με $t_0 = 1$.

Ας θεωρήσουμε, επιπρόσθετα την εξίσωση:

$$a(n)u_n = b(n)u_{n-1} + c(n).$$

Αν ισχύει

$$f(n) = \frac{\prod_{i=1}^{n-1} a(i)}{\prod_{i=1}^{n} b(i)}$$

τότε

$$v_n = v_{n-1} + f(n)c(n)$$

με $v_n = a(n)f(n)u_n$ και χρησιμοποιώντας τη μέθοδο των αθροιζομένων παραγόντων θα έχουμε

$$u_n = \frac{1}{a(n)f(n)}(a(0)f(0)u_0 + \sum_{i=1}^{n} f(i)c(i))$$

και άρα

$$t_k = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

Τελικά, αφού $k = \log_b n$ και $a^{\log_b n} = n^{\log_b a}$ η αρχική εξίσωση γίνεται

$$T(n) = n^{\log_b a} + \sum_{j=0}^{(\log_b n)-1} a^j d(\frac{n}{b^j}).$$

Παράδειγμα: Να επιλυθεί η παρακάτω αναδρομική εξίσωση (βλ. 2.1.1 πολυπλοκότητα Merge Sort):

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

Λύση: Έχουμε $a = b = 2$ και $d(n) = n - 1$ άρα

$$\begin{aligned} T(n) &= n + \sum_{j=0}^{k-1} 2^j((\frac{n}{2^j}) - 1) \\ &= n + kn - (2^k - 1) \end{aligned}$$

και επειδή $k = \log_2 n$ τελικά

$$T(n) = n \log_2 n + 1.$$

Μία άλλη μέθοδος για την μετατροπή μιας αναδρομικής εξίσωσης σε γραμμική είναι ο πολλαπλασιασμός αυτής με ένα αθροιστικό παράγοντα s_n. Ας υποθέσουμε την εξίσωση

$$a_n T_n = b_n T_{n-1} + c_n.$$

Τότε θα έχουμε:

$$s_n a_n T_n = s_n b_n T_{n-1} + s_n c_n.$$

Διαλέγοντας το s_n έτσι ώστε $s_n b_n = s_{n-1} a_{n-1}$ και θέτοντας $u_n = s_n a_n T_n$ η εξίσωση γίνεται

$$u_n = u_{n-1} + s_n c_n$$

και συνεπώς

$$u_n = s_0 a_0 T_0 + \sum_{k=1}^{n} s_k c_k = s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k$$

και η λύση είναι

$$T_n = \frac{u_n}{s_n a_n} = \frac{s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k}{s_n a_n}$$

δηλαδή τελικά

$$T_n = \frac{1}{s_n a_n (s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k)}.$$

Παράδειγμα: Να επιλυθεί η παρακάτω αναδρομική εξίσωση (βλ. 2.1.3 πύργοι του Hanoi):

$$T_n = 2T_{n-1} + 1$$

Λύση: Έχουμε $u_n = 1$, $b_n = 2$ και $s_n = 2^{-n}$ άρα

$$\begin{aligned} T_n &= \frac{1}{2^{-n}(0 + \sum_{k=1}^{n} s_k c_k)} = 2^n(s_1 + \ldots + s_n) \\ &= 2^n - 1 \end{aligned}$$