Calcularea complexității algoritmilor recursivi

30 octombrie 2022

Cuprins

1 Analiza complexității algoritmilor recursivi

Spre deosebire de un algoritm iterativ, unde putem număra operațiile efectuate în fiecare buclă, un algoritm recursiv necesită o analiză mai atentă. Un algoritm recursiv poate fi definit prin relația:

$$T(n) = cost\_recursiv(n) + cost\_nerecursiv(n)$$

Costul recursiv al unui apel se referă la câte noi apeluri recursive lansează un anumit apel și la dimensiunea parametrilor acestor noi apeluri, în timp ce costul nerecursiv reprezintă orice altceva mai face un apel (alte operații).

Relații de recurență uzuale:

1. Divide et impera: $T(n) = b \cdot T\left(\frac{n}{c}\right) + f(n)$
2. Chip and conquer: $T(n) = T(n-c) + f(n)$ [1]
3. Chip and be conquered: $T(n) = b \cdot T(n-c) + f(n)$ [2]

În primul rând, pentru un algoritm recursiv, trebuie definită o funcție care să descrie timpul său de rulare. Având funcția, putem aplica diverse metode pentru a determina complexitatea algoritmului.

[1] Exemplu: Algoritmul Quickselect.
[2] Exemplu: Recurența rezolvării problemei Turnurilor din Hanoi.

Vom exemplifica aceste metode prin studiul algoritmului Mergesort[1, p. 31,34].

Algoritm 1 Pseudocod Merge

```
1: funcție MERGE(A, p, q, r)
2:     n_1 = q − p + 1
3:     n_2 = r − q
4:     fie L[1..n_1 + 1] și R[1..n_2 + 1] două tablouri
5:     pentru i ← 1, n_1 execută
6:         L[i] = A[p + i − 1]
7:     final pentru
8:     pentru j ← 1, n_2 execută
9:         R[j] = A[q + j]
10:    final pentru
11:    L[n_1 + 1] = ∞
12:    R[n_2 + 1] = ∞
13:    i = j = 1
14:    pentru k ← p, r execută
15:        dacă L[i] ≤ R[j] atunci
16:            A[k] = L[i]
17:            i = i + 1
18:        altfel
19:            A[k] = R[j]
20:            j = j + 1
21:        final dacă
22:    final pentru
23: final funcție
```

Algoritm 2 Pseudocod Merge-Sort

```
1: funcție MERGE-SORT(A, p, r)
2:     dacă p ≤ r atunci
3:         q = ⌊(p + r)/2⌋
4:         MERGE-SORT(A, p, q)
5:         MERGE-SORT(A, q + 1, r)
6:         MERGE(A, p, q, r)
7:     final dacă
8: final funcție
```

Se poate observa că algoritmul *Merge* parcurge o singură dată elementele celor doi vectori, având complexitatea $\Theta(n)$. Pentru a calcula complexitatea algoritmului *MergeSort*, vom determina întâi o funcție caracteristică acestuia. Pentru a simplifica analiza complexității pseudocodul 2, vom presupune că că n, dimensiunea problemei, este o putere a lui 2. [1, cap. 2.3.2, subcap. *Analysis of merge sort*] (Algoritmul funcționează corect pentru orice n.)

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

În această formulă, n reprezintă dimensiunea vectorului primit la intrarea. Complexitatea algoritmului depinde de numărul de apeluri recursive (în cazul

nostru, 2) și de complexitatea fiecărui apel (în cazul nostru dată de apelul funcției *Merge*, având complexitatea $\Theta(n)$).

2 Metoda iterației

Recurența originală: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n;\ T(1) = c$

1. Explicităm termenul recursiv: $T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}$
2. Înmulțim cu factorul asociat: $2 \cdot T\left(\frac{n}{2}\right) = 4 \cdot T\left(\frac{n}{4}\right) + c \cdot n$
3. Repetăm până intuim forma generală pentru termenul liber.

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) && +c \cdot n \\
2 \cdot T\left(\frac{n}{2}\right) &= 4 \cdot T\left(\frac{n}{4}\right) && +c \cdot n \\
4 \cdot T\left(\frac{n}{4}\right) &= 8 \cdot T\left(\frac{n}{8}\right) && +c \cdot n \\
&\dots \\
2^k \cdot T\left(\frac{n}{2^k}\right) &= 2^{k+1} \cdot T\left(\frac{n}{2^{k+1}}\right) && +c \cdot n \\
&\dots \\
2^{h-1} \cdot T\left(\frac{n}{2^{h-1}}\right) &= 2^h \cdot \underbrace{T\left(\frac{n}{2^h}\right)}_{T(1)} && +c \cdot n
\end{aligned}
$$

4. Pentru a afla adâncimea apelurilor recursive, h, trebuie egalat $T(1)$ cu forma generală găsită $T\left(\frac{n}{2^h}\right)$:

$$\frac{n}{2^h} = 1 \Rightarrow h = \log_2(n)$$

5. Explicităm pentru cazul de bază $T(1)$: $2^h \cdot T(1) = n \cdot c$
6. Adunăm toți termenii liberi:

$$T(n) = h \cdot c \cdot n + n \cdot c = (\log_2(n) + 1) \cdot c \cdot n$$

7. $T(n) \in \Theta(n \cdot \log(n))$

3 Metoda arborelui de recurență

În fig. 1 este exemplificată această metodă pentru *MergeSort*. În fiecare nod al arborelui punem complexitatea termenului liber. În partea dreaptă, sunt adunate valorile din nodurile de pe acel nivel. Suma valorilor de pe fiecare nivel va reprezenta complexitatea întregului algoritm.

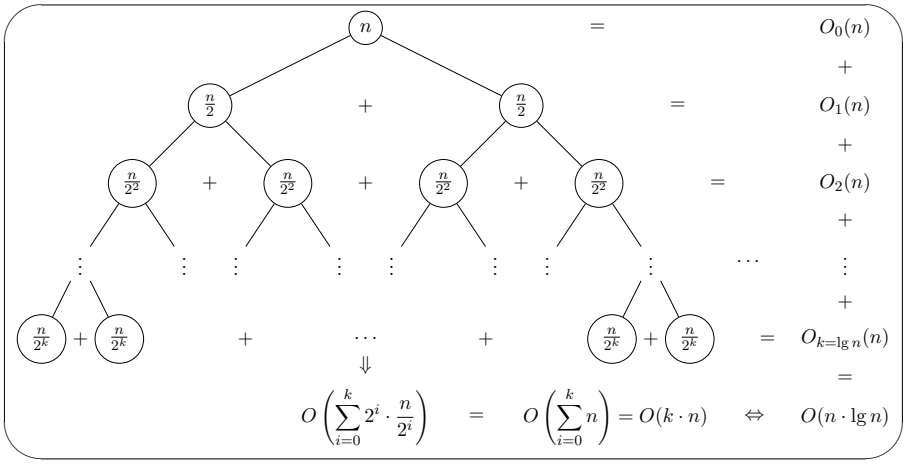

Figura 1: Arborele de recurență pentru MergeSort (aici $\lg n$ denotează $\log_2 n$)

Această metodă este foarte asemănătoare cu metoda iterației, în ambele cazuri fiind nevoiți să „ghicim" forma generală a sumei. O demonstrație formală ar necesita și o demonstrație prin inducție a aceste forme. Această reprezentare grafică poate fi mai intuitivă în unele cazuri, mai ales când metoda iterației este greu de aplicat.

4 Exerciții

1. $T(n) = 2T(n-1) + 1$
2. $T(n) = T(n-2) + \Theta(n)$
3. $T(n) = T(n-k) + \Theta(n)$, k const
4. $T(n) = T(n-a) + T(a) + \Theta(cn), a \geq 1$ și $c > 0$ const
5. $T(n) = 2T(n/3) + n \log n$
6. $T(n) = 7T(n/2) + n^2$
7. $T(n) = 2T(\sqrt{n}) + 1$
8. $T(n) = 2T(\sqrt{n}) + \log n$
9. $T(n) = 2T(\sqrt{n}) + \log(\log(n))$
10. $T(n) = T(n/5) + T(4n/5) + \Theta(n)$

11. $T(n) = T(n/5) + T(7n/10 + 6) + \Theta(n)$
12. $T(n) = T(n/2) + T(n/4) + n^2$
13. $T(n) = T(n/2 + \log(n)) + n$
14. $T(n) = 4T(n/2) + n^2/\log n$

Referințe

[1] Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C.: Introduction to Algorithms. 3rd edn. MIT Press and McGraw-Hill, USA (2009)