PROGRAMARE ȘI STRUCTURI DE DATE

CURS 7

Lect. dr. Oneț - Marian Zsuzsanna

Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

În cursul 5 și 6

- Containere
  - TAD Listă
  - TAD Stivă
  - TAD Coadă
  - TAD Dicționar
  - TAD MultiDicționar
- Containere ordonate
  - Colecția ordonată
  - Mulțimea ordonată
  - Lista ordonată
  - Coadă cu priorități
  - Dicționar ordonat
  - MultiDicționar ordonat

Cuprins

Complexități

- Sunt foarte multe probleme care pot fi rezolvate în mai multe feluri. Pentru a compara soluțiile și a determina care este mai bună se folosește *complexitatea* algoritmului.

Complexități

- Sunt foarte multe probleme care pot fi rezolvate în mai multe feluri. Pentru a compara soluțiile și a determina care este mai bună se folosește *complexitatea* algoritmului.
- În general sunt considerate două criterii importante când măsurăm performața unui algoritm: *timpul* necesar pentru a obține soluția problemei și *spațiul* - *memoria* utilizată pentru stocarea datelor necesare. Astfel, există *complexitate de timp* și *complexitate de spațiu*.

Complexitatea de timp

- Deși se numește *complexitate de timp*, acest tip de complexitate nu se măsoară în milisecunde, secunde, minute, etc., pentru că aceste valori depind foarte mult de calculatorul pe care rulăm codul.
- Pentru a calcula complexitatea de timp al unui algoritm, este necesar să considerăm în primul rând numărul de operații elementare efectuate de algoritm.
- Operațiile elementare sunt citirea, afișarea, atribuirea, comparația, adunarea, etc.

Exemplu

- Să considerăm o problemă mai simplă (decât problema cu bețe): *Să calculăm și să afișăm suma primelor n numere naturale*.
- De exemplu, pentru n = 4 soluția este 1+2+3+4 = 10, iar pentru n = 10 soluția este 55.

Exemplu 2 - Prima implementare

- Prima variantă de implementare este să folosim un ciclu (**pentru** sau **câttimp**), pentru a calcula suma, adunând pe rând fiecare element.

```
subalgoritm suma1(n: întreg) este:
    suma, i: întreg
    suma = 0
    i = 1
    câttimp i <= n execută
        suma = suma + i
        i = i + 1
    sf_câttimp
    scrie "Suma este: " + suma
sf_subalgoritm
```

Exemplu 2 - A doua implementare

- O altă soluție este să ne aducem aminte că există o formulă pentru a calcula suma primelor n numere:

$$\frac{n * (n + 1)}{2}$$

Exemplu 2 - A doua implementare

- O altă soluție este să ne aducem aminte că există o formulă pentru a calcula suma primelor n numere:

$$\frac{n * (n + 1)}{2}$$

subalgoritm suma2(n: întreg) **este:**
    suma: întreg
    suma = n * (n+1) / 2
    scrie "Suma este" + suma
sf_subalgoritm

Complexitatea de timp II

- Câte operații elementare sunt efectuate de prima variantă de soluție?

```
subalgoritm suma1(n: întreg) este:
    suma, i: întreg
    suma = 0
    i = 1
    câttimp i <= n execută
        suma = suma + i
        i = i + 1
    sf_câttimp
    scrie "Suma este: " + suma
sf_subalgoritm
```

Complexitatea de timp II

- Câte operații elementare sunt efectuate de prima variantă de soluție?

```
subalgoritm suma1(n: întreg) este:
    suma, i: întreg
    suma = 0
    i = 1
    câttimp i <= n execută
        suma = suma + i
        i = i + 1
    sf_câttimp
    scrie "Suma este: " + suma
sf_subalgoritm
```

- De foarte multe ori numărul de operații depinde de datele de intrare, în cazul nostru de n. Să calculăm numărul de operații pentru $n = 5$.

Numărul de operații pentru suma1

- definirea variabilelor suma și i - 2 operații
- suma $=$ 0 - o operație
- i $=$ 1 - o operație
- în ciclul **câttimp** avem următoarele operații:
  - i $<=$ 5 - o operație (dar se execută de 6 ori: pentru valorile 1, 2, 3, 4, 5, 6 ale variabilei i)
  - suma $=$ suma $+$ i - 2 operații (adunare și atribuire) (dar se execută de 5 ori: pentru valorile 1, 2, 3, 4, 5 ale variabile i)
  - i $=$ i $+$ 1 - 2 operații (adunare și atribuire) (dar se execută de 5 ori: pentru valorile 1, 2, 3, 4, 5 ale variabilei i)
- afișarea - o operație
- În total pentru n $=$ 5, vom avea $2 + 1 + 1 + 6 + 10 + 10 + 1 = 31$ operații

Numărul de operații pentru suma1

- Câte operații vom avea pentru numărul n = 10?

Numărul de operații pentru suma1

- Câte operații vom avea pentru numărul n $= 10$?
- Vom avea $2 + 1 + 1 + 11 + 20 + 20 + 1 = 56$ operații

Numărul de operații pentru suma1

- Câte operații vom avea pentru numărul n $= 10$?
- Vom avea $2 + 1 + 1 + 11 + 20 + 20 + 1 = 56$ operații
- Câte operații vom avea pentru numărul n $= 30$?

Numărul de operații pentru suma1

- Câte operații vom avea pentru numărul n $= 10$?
- Vom avea $2 + 1 + 1 + 11 + 20 + 20 + 1 = 56$ operații
- Câte operații vom avea pentru numărul n $= 30$?
- Vom avea $2 + 1 + 1 + 31 + 60 + 60 + 1 = 156$ operații

Numărul de operații pentru suma1

- Câte operații vom avea pentru numărul n $= 10$?
- Vom avea $2 + 1 + 1 + 11 + 20 + 20 + 1 = 56$ operații
- Câte operații vom avea pentru numărul n $= 30$?
- Vom avea $2 + 1 + 1 + 31 + 60 + 60 + 1 = 156$ operații
- Câte operații avem pentru numărul n?

Numărul de operații pentru suma1

- Câte operații vom avea pentru numărul n = 10?
- Vom avea $2 + 1 + 1 + 11 + 20 + 20 + 1 = 56$ operații
- Câte operații vom avea pentru numărul n = 30?
- Vom avea $2 + 1 + 1 + 31 + 60 + 60 + 1 = 156$ operații
- Câte operații avem pentru numărul n?
- Vom avea 2 + 1 + 1 + n+1 + 2*n + 2*n + 1 = 5*n + 6 operații. Numărul de operații pentru un algoritm care are date de intrare de mărime n, se notează cu T(n). Deci, în cazul algoritmului *suma1* avem $T(n) = 5 * n + 6$

Numărul de operații pentru suma2

- Câte operații sunt efectuate de algoritmul suma2 pentru n = 5?

subalgoritm suma2(n: întreg) **este:**
  suma: întreg
  suma = n * (n+1) / 2
  scrie "Suma este" + suma
sf_subalgoritm

Numărul de operații pentru suma2

- Câte operații sunt efectuate de algoritmul suma2 pentru n = 5?

subalgoritm suma2(n: întreg) **este:**
  suma: întreg
  suma = n * (n+1) / 2
  scrie "Suma este" + suma
sf_subalgoritm

- 6 operații

Numărul de operații pentru suma2

- Câte operații sunt efectuate de algoritmul suma2 pentru n = 5?

subalgoritm suma2(n: întreg) **este:**
  suma: întreg
  suma = n * (n+1) / 2
  scrie "Suma este" + suma
sf_subalgoritm

- 6 operații
- Și pentru n = 10? n = 30? numărul *n*?

Numărul de operații pentru suma2

- Câte operații sunt efectuate de algoritmul suma2 pentru n = 5?

subalgoritm suma2(n: întreg) **este:**
  suma: întreg
  suma = n * (n+1) / 2
  scrie "Suma este" + suma
sf_subalgoritm

- 6 operații
- Și pentru n = 10? n = 30? numărul *n*?
- Tot 6 operații. Indiferent de ce număr avem ca parametru, algortimul suma2 va efectua 6 operații, deci T(n) = 6.

Complexitatea de timp

- T(n) - adică numărul de operații - nu este complexitatea algoritmului. Ceea ce ne interesează pentru a calcula complexitatea este *cum se modifică valoarea T(n), dacă se modifică valoarea lui n?* Cât de repede crește valoarea T(n), dacă crește valoarea lui n?

Complexitatea de timp

- T(n) - adică numărul de operații - nu este complexitatea algoritmului. Ceea ce ne interesează pentru a calcula complexitatea este *cum se modifică valoarea T(n), dacă se modifică valoarea lui n?* Cât de repede crește valoarea T(n), dacă crește valoarea lui n?
- Pentru algoritmul suma1 am calculat că $T(n) = 5 * n + 6$. Cât va fi $T(n)$ dacă dublăm n-ul?

Complexitatea de timp

- T(n) - adică numărul de operații - nu este complexitatea algoritmului. Ceea ce ne interesează pentru a calcula complexitatea este *cum se modifică valoarea T(n), dacă se modifică valoarea lui n?* Cât de repede crește valoarea T(n), dacă crește valoarea lui n?
- Pentru algoritmul suma1 am calculat că $T(n) = 5 * n + 6$. Cât va fi $T(n)$ dacă dublăm n-ul?
- $T(2 * n) = 10 * n + 6$, adică $T(2 * n) \approx 2 * T(n)$

Complexitatea de timp

- T(n) - adică numărul de operații - nu este complexitatea algoritmului. Ceea ce ne interesează pentru a calcula complexitatea este *cum se modifică valoarea T(n), dacă se modifică valoarea lui n?* Cât de repede crește valoarea T(n), dacă crește valoarea lui n?
- Pentru algoritmul suma1 am calculat că $T(n) = 5 * n + 6$. Cât va fi $T(n)$ dacă dublăm n-ul?
- $T(2 * n) = 10 * n + 6$, adică $T(2 * n) \approx 2 * T(n)$
- Pentru algoritmul suma2 am calculat că $T(n) = 6$ indiferent de valoarea lui n. Adică, $T(2 * n) = T(n)$

Complexitatea de timp

- Când vorbim de complexități, nu ne interesează numărul exact de operații, ci doar termenul de gradul maxim din $T(n)$ (de aceea, în general nici nu calculăm T(n) exact, ci încercăm să găsim doar termenul de gradul maxim din T(n)) .
- Încercăm să căutăm o funcție care, la valori mari pentru n, aproximează valoarea lui T(n). De aceea, acest tip de analiză se numește *analiză asimptotică*.

Complexitatea de timp

- Să presupunem că avem un algoritm cu următorul număr de operații: $T(n) = n^2 + 3 * n + 20$

n	n^2	T(n)	$\frac{T(n)}{n^2}$
1	1	24	24
5	25	60	2.4
7	49	90	1.84
10	100	150	1.5
100	10000	10320	1.032
200	40000	40620	1.0155

- Cu cât crește valoarea lui n, cu atât devine mai mică diferența dintre valoarea $T(n)$ și valoarea n^2. Astfel, pentru valori mari pentru n, putem spune că $T(n) \approx n^2$

Complexitatea de timp

- Există 3 notații folosite pentru a descrie complexitatea unui algoritm: *notația O-mare*, *notația Ω-mare* și *notația Θ-mare*

Notația O – *mare* I

- Notația O-mare desemnează **marginea asimptotică superioară** a unei funcții.
- **Fie o funcție $g(n)$. Mulțimea de funcții $O(g(n))$ este definită în modul următor:**

$$O(g(n)) = \{f(n) : \text{ exista constante pozitive } c \text{ si } n_0 \text{ astfel incat } 0 \leq f(n) \leq c * g(n) \text{ pentru orice } n \geq n_0\} \quad (1)$$

- Pentru a indica apartenența unei funcții $f(n)$ la mulțimea $O(g(n))$, vom utiliza notația $f(n) = O(g(n))$ sau $f(n) \in O(g(n))$

Notația O — *mare* II

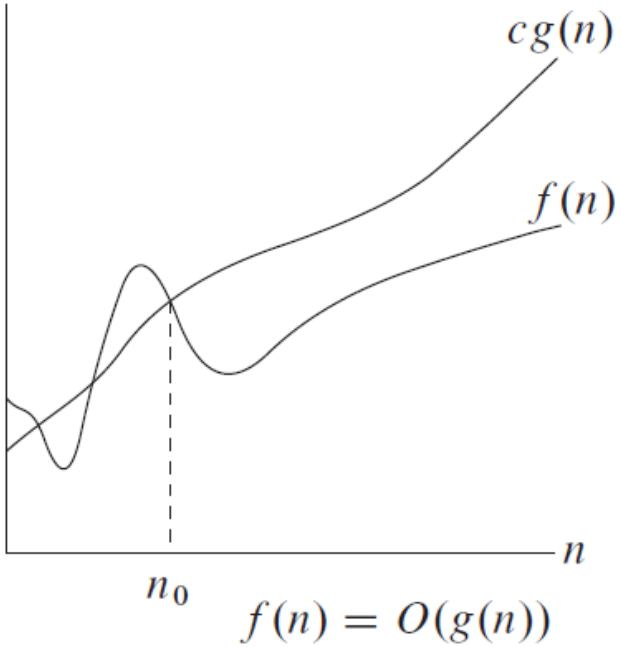

Figure: Sursă: Cormen et. al: Introduction to Algorithms, 3rd Edition, MIT Press, 2009

Notația O – *mare* III

- Să considerăm $f(n) = n^4 + 2n^2 + 10n + 500$ și $g(n) = n^5$.
- Pe baza definiției, $f(n) = O(g(n))$ dacă putem găsi constante pozitive c și n_0 astfel încât $0 \leq n^4 + 2n^2 + 10n + 500 \leq c * n^5$ pentru orice $n \geq n_0$
- Inegalitatea nu trebuie să fie adevărată pentru toate valorile posibile ale lui n. Pot exista valori mici, pentru care inegalitatea să nu fie adevărată, dar trebuie să arătăm că există o limită, n_0, și inegalitatea este adevărată pentru orice punct mai mare ca n_0.
- Există mai multe alegeri *bune* pentru c și n_0. Să alegem $c = 2$

Notația O – *mare* IV

n	$f(n)$	2 * g(n)
1	513	2
3	629	486
4	828	2048
5	1225	6250
7	3069	33614

- Pe baza valorilor din tabel, putem conclude că pentru orice $n_0 \geq 4$, vom avea $0 \leq n^4 + 2n^2 + 10n + 500 \leq 2n^5$
- **Obs.** Dacă alegeam o valoare mai mare pentru c, atunci probabil inegalitatea era adevărată de la o valoare mai mică pentru n_0, dar acest lucru nu contează: trebuie să găsim un singur c pentru care trebuie să existe un n_0, nu contează cât de mare/mic este.

Notația O – *mare* V

- Deci pentru funcția $f(n) = n^4 + 2n^2 + 10n + 500$ avem $f(n) = O(n^5)$.
- Să considerăm $g(n) = n^4$. Este $f(n) = O(n^4)$?

Notația O – *mare* VI

- Pentru a demonstra că $f(n) = O(n^4)$ trebuie să căutăm constante pozitive c și n_0, astfel încât $0 \leq n^4 + 2n^2 + 10n + 500 \leq c * n^4$ pentru orice $n \geq n_0$
- Să alegem $c = 3$. În acest caz avem următoarele valori:

n	$f(n)$	$3 * n^4$
1	513	3
3	629	243
4	828	768
5	1225	1875
7	3069	7203

- Pe baza valorilor din tabel, putem conclude că pentru orice $n_0 \geq 5$, vom avea $0 \leq n^4 + 2n^2 + 10n + 500 \leq 3n^4$

Notația O – *mare* VII

- Să considerăm $g(n) = n^3$. Este $f(n) = O(n^3)$?

Notația O – *mare* VIII

- Pentru a demonstra că $f(n) = O(n^3)$ trebuie să căutăm constante pozitive c și n_0, astfel încât $0 \leq n^4 + 2n^2 + 10n + 500 \leq c * n^3$, pentru orice $n \geq n_0$.

- Aceată inegalitate nu mai poate fi demonstrată. Indiferent cât de mare ar fi valoarea lui c, va exista un punct p, când $f(p)$ devine mai mare decât $c * p^3$.

Notația O – *mare* IX

- Să considerăm $f(n) = 3n^3 + 20n^2 + 10n + 500$ și $g(n) = n^3$. Este $f(n) = O(g(n))$?
- Pe baza definiției, $f(n) = O(g(n))$ dacă putem găsi constante pozitive c și n_0 astfel încât
  $0 \leq 3 * n^3 + 20n^2 + 10n + 500 \leq c * n^3$ pentru orice $n \geq n_0$
- E clar că pentru valori de $c \leq 3$ inegalitatea nu este adevărată. Să luăm $c = 4$.

Notația O – *mare* X

n	$3 * n^3 + 20n^2 + 10 * n + 500$	$4 * n^3$
1	533	4
5	1425	500
10	5600	4000
15	15275	13500
20	32700	32000
25	60125	62500

- Pe baza valorilor din tabel, putem conclude că pentru orice $n_0 \geq 25$, vom avea $0 \leq 3 * n^3 + 20 * n^2 + 10n + 500 \leq 4n^3$
- Deci pentru funcția $f(n) = 3 * n^3 + 20 * n^2 + 10n + 500$ avem $f(n) = O(n^3)$.

Notația O – *mare* XI

- Am văzut că pentru $f(n) = n^4 + 2 * n^2 + 10n + 500$:
  - $f(n) = O(n^5)$,
  - $f(n) = O(n^4)$
  - $f(n) \neq O(n^3)$.
- Puteți spune alte funcții $g(n)$ pentru care $f(n) = O(g(n))$?

Notația O – *mare* XII

- Funcția suma1 avea $T(n) = 5 * n + 6$
- Găsiți o funcție $g(n)$ pentru care $T(n) = O(g(n))$.
- Puteți găsi mai multe valori pentru $g(n)$?

Notația O – *mare* XIII

- Pentru orice funcție $g(n)$ mai mare sau egal cu n (de ex. n, n^2, n^3, 2^n, etc.) avem $T(n) = O(g(n))$

Notația O – *mare* XIV

- Funcția suma2 avea $T(n) = 6$
- Găsiți o funcție $g(n)$ pentru care $T(n) = O(g(n))$
- Puteți găsi mai multe valori pentru $g(n)$?

Notația O – *mare* XV

- Pentru orice funcție $g(n)$ mai mare sau egal cu 1 (de ex. 1, n, n^2, n^3, 2^n, etc.) avem $T(n) = O(g(n))$

Notația O – *mare* XVI

- Am văzut că pentru o funcție $f(n)$ pot exista mai multe funcții $g(n)$, astfel încât $f(n) = O(g(n))$. În general, dacă folosim notația O-mare, încercăm să găsim **cea mai mică valoarea posibilă** pentru $g(n)$.
- Deci, vom spune că $T(n) = 5 * n + 6$ aparține lui $O(n)$ și $T(n) = 6$ aparține lui $O(1)$.

Notatia O-mare - concluzii

- Să presupunem că $T(n)$ este numărul de operații elementare pentru un algoritm
- Să presupunem că $t(n)$ este termenul de gradul maxim din $T(n)$, fără orice constantă
- $T(n)$ aparține lui $O(g(n))$, dacă $g(n) \geq t(n)$
- Exemple:
  - $7n^3 + 10 \in O(n^3)$
  - $\frac{n^2}{2} + 4n + 1 \in O(n^2)$
  - $n * log_2 n + 4n \in O(n * log_2 n)$
  - $n * log_2 n + 4n \in O(n^2)$

Notația Ω – *mare* I

- Notația Ω-mare desemnează **marginea asimptotică inferioară** a unei funcții.
- **Fie o funcție $g(n)$. Mulțimea de funcții $\Omega(g(n))$ este definită în modul următor:**

$$\Omega(g(n)) = \{f(n) : \text{ exista constante pozitive } c \text{ si } n_0 \text{ astfel incat } 0 \leq c * g(n) \leq f(n) \text{ pentru orice } n \geq n_0\} \quad (2)$$

- Pentru a indica apartenența unei funcții $f(n)$ la mulțimea $\Omega(g(n))$, vom utiliza notația $f(n) = \Omega(g(n))$ sau $f(n) \in \Omega(g(n))$

Notația Ω — *mare* II

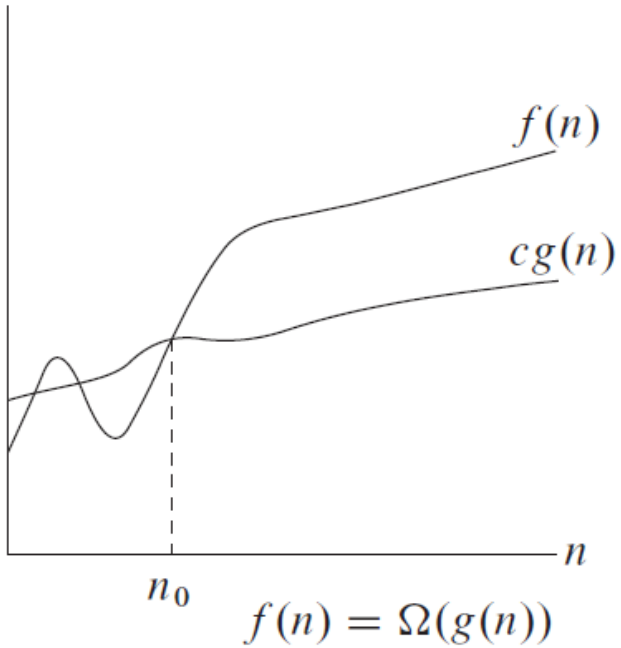

Figure: Sursă: Cormen et. al: Introduction to Algorithms, 3rd Edition, MIT Press, 2009

Notația Ω – *mare* III

- Să considerăm $f(n) = n^4 + 2n^2 + 10n + 500$
- Găsiți valori pentru funcția $g(n)$, astfel încât $f(n) = \Omega(g(n))$

Notația Ω — *mare* IV

- Pentru orice funcție $g(n)$ mai mică sau egală cu n^4 (1, n, n^2, n^3 și n^4, etc.) avem $f(n) = \Omega(g(n))$
- Și în cazul $T(n) = 5 * n + 6$?
- Și la $T(n) = 6$?

Notația Ω – *mare* V

- Am văzut că pentru o funcție $f(n)$ pot exista mai multe funcții $g(n)$, astfel încât $f(n) = \Omega(g(n))$. În general, dacă folosim notația Ω-mare, încercăm să găsim **cea mai mare valoarea posibilă** pentru $g(n)$.
- Deci, vom spune că $T(n) = 5 * n + 6$ aparține lui $\Omega(n)$ și $T(n) = 6$ aparține lui $\Omega(1)$.

Notația Θ-mare I

- **Fie o funcție $g(n)$. Mulțimea de funcții $\Theta(g(n))$ este definită în modul următor:**

$$\Theta(g(n)) = \{f(n) : \textit{ exista constante pozitive } c_1, c_2 \textit{ si } n_0 \textit{ astfel incat } c_1 * g(n) \leq f(n) \leq c_2 * g(n) \textit{ pentru orice } n \geq n_0\} \quad (3)$$

- Pentru a indica apartenența unei funcții $f(n)$ la mulțimea $\Theta(g(n))$, vom utiliza notația $f(n) = \Theta(g(n))$ sau $f(n) \in \Theta(g(n))$

Notația Θ-mare II

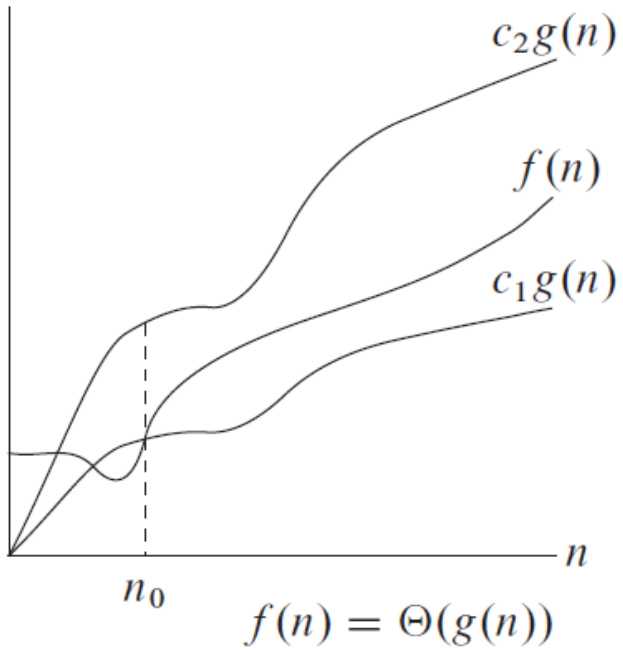

Figure: Sursă: Cormen et. al: Introduction to Algorithms, 3rd Edition, MIT Press, 2009

Notația Θ-mare III

- Să considerăm funcțai suma1, care avea $T(n) = 5 * n + 6$. Găsiți o valoare pentru funcția $g(n)$, astfel încât $f(n) = \Theta(g(n))$
- Să considerăm funcțai suma2, care avea $T(n) = 6$. Găsiți o valoare pentru funcția $g(n)$, astfel încât $f(n) = \Theta(g(n))$

Notația Θ-mare IV

- Pentru funcția suma1 avem $T(n) = \Theta(n)$
- Pentru funcția suma2 avem $T(n) = \Theta(1)$

Notația Θ-mare V

- $O(g(n))$ este limită superioară pentru funcția $f(n)$, $\Omega(g(n))$ este limita inferioară, $\Theta(g(n))$ este valoarea exactă.
- Am văzut că pentru o funcție $f(n)$, pot exista mai multe funcții $g(n)$, astfel încât $f(n) = O(g(n))$ sau $f(n) = \Omega(g(n))$, dar există o singură funcție pentru care $f(n) = \Theta(n)$
- Când acest lucru este posibil, vom folosi notația Θ pentru complexitatea unui algoritm. Dacă nu putem folosi notația Θ, vom folosi notația O, și încercăm să găsim cea mai mică valoare la notația O.

Complexități uzuale

- Câteva complexități des întâlnite în practică
  - $T(n) = \Theta(1)$ - complexitate **constantă**
  - $T(n) = \Theta(log_2 n)$ - complexitate **logaritmică**
  - $T(n) = \Theta(n)$ - complexitate **liniară**
  - $T(n) = \Theta(n^2)$ - complexitate **pătratică**
  - $T(n) = \Theta(n^k)$ - complexitate **polinomială**
  - $T(n) = \Theta(2^n)$ - complexitate **exponențială**

Ghid pentru a calcula complexitatea unui algoritm I

- **Cicluri** - Numărul de operații pentru un ciclu: numărul de operații din interiorul ciclului, înmulțit cu numărul de repetiții pentru ciclu.

```
m = 0
n = 1
pentru i = 0, n, 1 execută
    m = m + 2
    n = n + 2 * m
sf_pentru
```

Ghid pentru a calcula complexitatea unui algoritm II

- Practic, numărul de operații pentru orice ciclu (unde incrementarea se face cu 1) poate fi scris ca o sumă:

$$2 + \sum_{i=0}^{n-1} 5 = 2 + 5 * n$$

- Deci $T(n) = 2 + 5 * n = \Theta(n)$

Ghid pentru a calcula complexitatea unui algoritm III

- **Cicluri imbricate** - Se analizează de la ciclul din exterior spre ciclul din interior. Numărul de operații este produsul numerelor de operații din cicluri.

```
k = 1
pentru i = 0, n, 1 execută
    pentru j = 0, n, 1, execută
        k = k + i + j
    sf_pentru
sf_pentru
```

Ghid pentru a calcula complexitatea unui algoritm IV

- $$1 + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = 1 + \sum_{i=0}^{n-1} n = 1 + n^2$$
- Deci $T(n) = n^2 + 1 = \Theta(n^2)$
- Nu trebuie să considerăm separat că avem 3 instrucțiuni în ciclul pentru cu j (2 adunări și o atribuire), dacă e vorbă de un număr constant de operații, putem scrie doar 1.

Ghid pentru a calcula complexitatea unui algoritm V

- **Instrucțiuni consecutive** - Se adună numărul de operații pentru fiecare instrucțiune

```
x = 0
pentru i = 0, n, 1 execută
    x = x + 1
sf_pentru
m = x
pentru i = 0, n, 1 execută
    pentru j = 1, i, 1 execută
        m = m + j - i
    sf_pentru
sf_pentru
```

Ghid pentru a calcula complexitatea unui algoritm VI

- $$1+\sum_{i=0}^{n-1}1+1+\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1=2+n+\sum_{i=0}^{n-1}i=2+n+\frac{n*(n-1)}{2}$$
- $$T(n)=\frac{n^2}{2}+\frac{n}{2}+2=\Theta(n^2)$$

Ghid pentru a calcula complexitatea unui algoritm VII

- **Instrucțiunea dacă** - Se adună numărul de operații pentru condiție și maximul dintre numărul de operații dintre ramura *dacă* și *altfel*

```
rez = 0
dacă x ≤ 0 atunci
    rez = x * (-1)
altfel
   pentru i = 1, x+1, 1 execută
      rez = rez + x
   sf_pentru
sf_dacă
```

Ghid pentru a calcula complexitatea unui algoritm VIII

- $$1 + 1 + max(1, \sum_{i=1}^{x} 1) = 2 + max(1, x) = 2 + x$$
- $T(x) = x + 2 = \Theta(x)$
- În general complexitățile sunt sub forma $\Theta(n)$, sau $O(n)$, dar dacă avem un algoritm unde datele de intrare nu sunt notate cu n, putem folosi altă valoare, de ex: $\Theta(x)$

Caz favorabil, defavorabil, mediu I

- Am văzut că numărul de operații efectuate de un algoritm ($T(n)$) de multe ori depinde de mărimea datelelor de intrare (notată de obicei cu valoarea n).
- De exemplu, dacă am o funcție care calculează suma elementelor dintr-un tablou, numărul de operații depinde de lungimea tabloului (numărul de elemente din tablou).
- Câteodată numărul de operații nu depinde doar de mărimea datelor de intrare, ci și de datele exacte.

Caz favorabil, defavorabil, mediu II

- Să considerăm funcția următoare, care caută poziția unui număr dat într-un tablou și returnează -1 dacă tabloul nu conține elementul respectiv.

Caz favorabil, defavorabil, mediu III

```
funcție PozitieElement(tablou: întreg[], n: întreg, elem: întreg) este:
//n este numărul de elemente din tablou
    rezultat, index: întreg
    gasit: boolean
    rezultat = -1
    gasit = fals
    index = 0
    cât timp gasit = fals ȘI index < n execută
        dacă tablou[index] == elem atunci
            rezultat = index
            gasit = adevărat
        sf_dacă
        index = index + 1
    sf_cât timp
    returnează rezultat
sf_funcție
```

Caz favorbil, defavorabil, mediu IV

- Să presupunem că tabloul transmis ca parametru are lungimea $n = 5$ și căutăm elementul 13. Câte operații vor fi efectuate de algoritm?
- Trebuie să determinăm de câte ori se execută ciclul *câttimp*.
  - Dacă tablou[0] este 13 (de exemplu în tabloul [13, 1, 5, 3, 6]), ciclul *câttimp* se execută o singură dată și avem aproximativ $6 + 6 + 1 = 13$ operații
  - Dacă tablou[0] nu e 13 și tablou[1] este 13 (de exemplu în tabloul [5, 13, 9, 10, 12]) ciclul *câttimp* se execută de 2 ori și avem aproximativ $6 + 11 + 1 = 18$ operații
  - Dacă tablou[0] și tablou[1] nu e 13 și tablou[2] este 13 (de exemplu în tabloul [3, 7, 13, 1, 2]), ciclul *câttimp* se execută de 3 ori și avem aproximativ $6 + 15 + 1 = 22$ operații
  - etc...

Caz favorabil, defavorabil, mediu V

- În cazul funcției *PozitieElement*, complexitatea algoritmului nu depinde doar de lungimea tabloului, ci și de elemente exacte din tablou. În aceste situații putem vorbi despre:
  - Complexitate în caz favorabil - cel mai mic număr de operații
  - Complexitate în caz defavorabil - cel mai mare număr de operații
  - Complexitate în caz mediu - numărul mediu de operații
- **Cazul favorabil, defavorabil și mediu se referă la un n fixat!** Nu spunem că avem caz favorabil când tabloul e scurt și caz defavorabil când e lung.
- Nu orice algoritm are caz favorabil, defavorbil și mediu separat, sunt algoritmi care tot timpul execută același număr de operații pentru un n fixat.

Cazul favorabil

- Cazul favorabil este cea mai bună situație posibilă, reprezintă datele de intrare pentru care algoritmul va executa un număr cât mai mic (minimul posibil) de operații.

Cazul favorabil

- Cazul favorabil este cea mai bună situație posibilă, reprezintă datele de intrare pentru care algoritmul va executa un număr cât mai mic (minimul posibil) de operații.
- În cazul funcției *PozitieElement* cazul favorabil este când primul element din tablou este cel căutat. În acest caz algoritmul va executa aproximativ 13 operații (6 operații de inițializare, condiția din câttimp și din dacă, 2 atribuiri în dacă, incrementarea indexului, condiția din câttimp și atribuirea finală).
- Numărul de operații în acest caz favorabil nu depinde de lungimea tabloului (adică în caz favorabil avem 13 operații indiferent de numărul de elemente din tablou), deci complexitatea în caz favorabil este $\Theta(1)$

Cazul defavorabil

- Cazul defavorabil este situația cea mai defavorabilă, datele de intrare pentru care algoritmul va executa numărul maxim de operații posibile.

Cazul defavorabil

- Cazul defavorabil este situația cea mai defavorabilă, datele de intrare pentru care algoritmul va executa numărul maxim de operații posibile.
- În cazul funcției *PozitieElement* cazul defavorabil este când tabloul nu conține elementul căutat. În acest caz va trebui să parcurgem și să verificăm toate elementele din tablou. E ca și cum nu ar exista variabila *găsit* și am avea un ciclu *pentru*.
- În cazul defavorabil parcurgem toate elementele, deci complexitatea este proporțională cu lungimea tabloului, deci $\Theta(n)$

Cazul mediu I

- La cazult mediu, teoretic, ar trebuie să luăm toate cazurile posibile, să numărăm pentru fiecare caz câte operații sunt efectuate, și să facem media. Dar acest lucru e imposibil în general.
- Cazul mediu se calculează cu o formulă:

$$\sum_{I \in D} P(I) * E(I)$$

- D reprezintă *domeniul* problemei, adică toate datele de intrare posibile (pentru *PozitieElement* asta înseamnă toate tablourile de numere întregi cu n elemente).

Cazul mediu II

- I reprezintă un caz posibil pentru date de intrare (pentru *PozitieElement* când vorbim de I nu ne gândim la un tablou concret, ci la situații posibile: tablou unde primul element este cel căutat, tablou unde al 2-lea element este cel căutat, ..., tablou unde ultimul element este cel căutat, tablou care nu conține elementul căutat)
- $P(I)$ reprezintă *probabilitatea* să avem ca data de intrare pe I. De cele mai multe ori probabilitățile sunt considerate egale, adică e la fel de probabil să am tablou unde primul element e cel căutat ca situația în care elementul căutat e pe poziția 2, etc.
- $E(I)$ este numărul de operații care vor fi efectuate pentru datele de intrare I.

Cazul mediu III

- Și acum să vedem concret pentru funcția *PozitieElement*:
  - Fixăm lungimea tabloului la n.
  - Pentru I există următoarele posibilitați: tablou unde primul număr e cel căutat, tablou unde al 2-lea număr este cel căutat, ..., tablou unde ultimul element este cel căutat, tablou unde nu există numărul căutat.
  - Sunt în total $n+1$ situații posibile, deci probabilitatea $P(I)$ este $\frac{1}{n+1}$.
  - Pentru a calcula valoarea $E(I)$ putem considera doar numărul de câte ori este executat ciclul *câttimp*, pentru fiecare I (în loc să încercăm să calculăm numărul exact de operații).

$$\sum_{I \in D} P(I) * E(I) = \frac{1}{n+1} * 1 + \frac{1}{n+1} * 2 + ... + \frac{1}{n+1} * n + \frac{1}{n+1} * n =$$

$$\frac{1}{n+1} * (1+2+3+...+n+n) = \frac{1}{n+1} * \frac{n*(n+1)}{2} + \frac{n}{n+1} = \Theta(n)$$

Cazul mediu IV

- Am calculat că avem:
  - Caz favorabil $\Theta(1)$
  - Caz defavorabil $\Theta(n)$
  - Caz mediu $\Theta(n)$
- Complexitatea totală este maximul dintre aceste valori, adică n. Dar pentru a arăta că există situații când complexitatea este mai mică, vom folosi notația O – *mare*. Dacă toate cele 3 valori erau la fel, atunci complexitatea totală era exprimată cu Θ.
- Deci, complexitatea totală pentru funcția *PozitiePar* este $O(n)$

Complexitatea de timp - Concluzii

- Complexitatea unui algoritm depinde de mărimea datelor de intrare.
- Câteodată complexitatea depinde nu doar de mărimea datelor de intrare ci și de valoarea lor exactă. În aceste situații vorbim de caz favorabil, defavorabil și mediu.
- Notația Ω – *mare* nu prea este folosită în practică
- În general, preferăm să folosim notația Θ – *mare*
- Folosim notația O – *mare* în 2 situații:
  - Nu pot exprima complexitatea cu Θ (nu pot calcula valoarea exactă), dar pot să dau o limită superioară
  - Am caz favorabil, defavorabil și mediu, și aceste 3 cazuri au complexități diferite
- Dacă folosim notația O – *mare*, încercăm să estimăm cât mai precis valoarea complexității

Formule utile

- $$\sum_{i=0}^{n} 1 = 1 + 1 + 1 + ... + 1 = n + 1$$
- $$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + ... + n = \frac{n * (n+1)}{2}$$
- $$\sum_{i=0}^{n} i^2 = 0^2 + 1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n * (n+1) * (2n+1)}{6}$$
- $$\sum_{i=0}^{n} i^3 = 0^3 + 1^3 + 2^3 + 3^3 + ... + n^3 = \frac{n^2 * (n+1)^2}{4}$$

Formule utile

- $$\sum_{i=0}^{n} i^4 = 0^4 + 1^4 + ... + n^4 = \frac{n * (n+1) * (2n+1) * (3n^2 + 3n - 1)}{30}$$
- $$\sum_{i=0}^{n} p^i = 1 + p^1 + p^2 + ... + p^n = \frac{p^{n+1} - 1}{p - 1}$$

Complexitatea de spațiu I

- Complexitatea de spațiu măsoară spațiul de memorie de care are nevoie un algoritm pentru a stoca variabilele folosite pe parcursul algoritmului. În general vorbim de *complexitate de spațiu extra*, adică nu calculăm spațiul ocupat de parametri, doar ceea ce trebuie în plus.
- Se folosesc aceleași notații ca în cazul complexității de timp
- De exemplu, complexitatea de spațiu pentru algoritmul *PozitieElement* este $\Theta(1)$ - tot timpul avem de reținul doar 3 variabile (*rezultat*, *gasit* și *index*) indiferent de mărimea tabloului.

Complexitatea de spațiu II

- De multe ori există un compromis între complexitatea de timp și spațiu: un algoritm cu o complexitate de timp mai bună, s-ar putea să aibă complexitate de spațiu mai mare, decât un alt algoritm care are o complexitate de timp mai slabă.

Complexitatea pentru exemplul cu bete

- Temă de gândire până la cursul 9: gândiți-vă ce complexitate au funcțiile pentru cele 3 variante de a rezolva problema cu bețe.

Exemple

```
public static int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i * i; j++) {
            sum = sum + 1;
        }
    }
    return sum;
}
```

- **Obs:** Nu trebuie să înțelegem ce face subalgoritmul pentru a calcula complexitatea.

Exemple

- Putem rescrie (aproape) orice ciclu **pentru** ca o suma:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{j<i^2} 1 = \sum_{i=0}^{n-1} i^2 = \frac{(n-1) * n * (2n-1)}{6} \in \Theta(n^3)$$

Exemple

- Putem rescrie (aproape) orice ciclu **pentru** ca o suma:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{j<i^2} 1 = \sum_{i=0}^{n-1} i^2 = \frac{(n-1) * n * (2n-1)}{6} \in \Theta(n^3)$$

- **Obs 1:** Nu trebuie să calculăm expresia exactă, când vedem cât va fi termenul de gradul maxim, ne putem opri cu calcule.
- **Obs 2:** Două cicluri **pentru** unul în altul, nu înseamnă automat complexitate n^2.

Exemple

```
public static int sum(int n){
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            sum = sum + 1;
        }
    }
    return sum;
}
```

Exemple

- Dacă în ciclul *pentru*, variabila se incrementează cu 1, putem folosi suma:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1}(i-1) = \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 =$$

$$= \frac{(n-1)*n}{2} - (n-1) \in \Theta(n^2)$$

Exemple

```
public static int mistery(int n) {
    int r = 0;
   for (int i = 1; i <= n; i++) {
      for (int j = i+1; j <= n; j++) {
         for (int k = i + j - 1; k <= n; k++) {
            r = r + 1;
         }
      }
   }
   return r;
}
```

Exemple

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sum_{k=i+j-1}^{n} 1 = \sum_{i=1}^{n}\sum_{j=i+1}^{n}(n-(i+j-1)+1) =$$

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

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

- Continuăm separat cu cei 4 termeni, și la final, când avem o experisie pentru fiecare, revenim la expresia de aici

Exemple - Primul termen

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} n = \sum_{i=1}^{n}(n-(i+1)+1) * n =$$

$$= \sum_{i=1}^{n}(n-i)n = \sum_{i=1}^{n}(n^2 - i * n) = \sum_{i=1}^{n} n^2 - \sum_{i=1}^{n} i * n =$$

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

$$= \frac{2*n^3}{2} - \frac{n^3+n^2}{2} = \frac{n^3-n^2}{2}$$

Exemple - Al 2-lea termen

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} i = \sum_{i=1}^{n}(n-(i+1)+1)*i =$$

$$= \sum_{i=1}^{n}(n-i)*i = \sum_{i=1}^{n} n*i - \sum_{i=1}^{n} i^2 =$$

$$= n*\sum_{i=1}^{n} i - \frac{n*(n+1)*(2n+1)}{6} = n*\frac{n*(n+1)}{2} - \frac{(n^2+n)*(2n+1)}{6} =$$

$$= \frac{3n^3+3n^2}{6} - \frac{2n^3+n^2+2n^2+n}{6} = \frac{n^3-n}{6}$$

Exemple - Al 3-lea termen

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} j = \sum_{i=1}^{n} \frac{n*(n+1)}{2} - \frac{i*(i+1)}{2} =$$

$$\sum_{i=1}^{n} \frac{n^2+n-i^2-i}{2} = \frac{1}{2}*(\sum_{i=1}^{n} n^2 + \sum_{i=1}^{n} n - \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i) =$$

$$\frac{1}{2}*(n*n^2 + n*n - \frac{n*(n+1)*(2n+1)}{6} - \frac{n*(n+1)}{2}) =$$

$$= \frac{1}{2}*(n^3 + n^2 - \frac{2n^3+3n^2+n}{6} - \frac{n^2+n}{2}) =$$

$$= \frac{1}{2}*(\frac{6n^3+6n^2-2n^3-3n^2-n-3n^2-3n}{6}) = \frac{4n^3-4n}{12} = \frac{n^3-n}{3}$$

Exemple - Al 4-lea termen

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 2 = \sum_{i=1}^{n}(n-(i+1)+1)*2 = \sum_{i=1}^{n} 2*(n-i) =$$

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

$$= 2n^2 - 2*\frac{n*(n+1)}{2} = 2n^2 - n^2 - n = n^2 - n$$

Exemple - Înapoi la expresia anterioară

$$\frac{n^3-n^2}{2}-\frac{n^3-n}{6}-\frac{n^3-n}{3}+n^2-n=$$

$$=\frac{3n^3-3n^2-n^3+n-2n^3+2n+6n^2-6n}{6}=\frac{3n^2-3n}{6}\in\Theta(n^2)$$

- **Obs:** În acest exemplu, termenul de gradul 3 (n^3) dispare din rezultat, deci doar faptul că pe prima linie avem n^3 nu înseamnă automat că vom avea complexitate $\Theta(n^3)$.

Exemple

```
public static void function(int n) {
   for (int i = 1; i <= n/3; i++) {
      for (int j = 1; j <= n; j = j + 4) {
         System.out.println(*);
      }
   }
}
```

Exemple

- În al doilea ciclu **pentru**, j nu se incrementează cu 1, deci nu putem scrie acel ciclu **pentru** sub forma unei sume.
- În primul ciclu **pentru**, i merge de la 1, până la $\frac{n}{3}$, și la fiecare pas i crește cu 1. În total, ciclul se repetă de $\frac{n}{3}$ ori.
- În al doilea ciclu **pentru**, j merge de la 1, până la n, și la fiecare pas crește cu 4. Valorile lui j vor fi:
  - 1, 5, 9, 13, 17, 21, etc.
- Dacă j crește cu 1, ciclul se repetă de n ori. Dacă j crește cu 4, ciclul se repetă de $\frac{n}{4}$ ori.
- Cele 2 cicluri fiind imbricate, înmulțim valorile. $T(n) \in \Theta(n^2)$

Exemple

```
public static void s1(n){
   for (int i = 1; i <= n; i++) {
      int j = n;
      while (j > 0) {
         j = j/2;
         System.out.println(*);
      }
   }
}
```

Exemple

- Când vedem ciclu **câttimp**, trebuie să verificăm dacă avem o situație când avem caz favorabil, defavorabil, mediu.
- Aici nu avem caz favorabil, defavorabil și mediu, dar nici nu putem rescrie **câttimp** ca o sumă (La fiecare iterație j se împarte la 2, nu putem scrie acest lucru ca o sumă).
- Ciclul **pentru** se repetă de n ori.
- Valoarea lui j tot timpul pornește de la n, și la fiecare iterație se înjumătățește.
  - Dacă j e inițial 100, vom avea: 100, 50, 25, 12, 6, 3, 1, 0
  - Dacă j e inițial 500, vom avea: 500, 250, 125, 62, 31, 15, 7, 3, 1, 0

Exemple

- Numărul de iterații pentru ciclul **câttimp** este $log_2 n$ ($log_b X$, arată de câte ori pot împărții în mod repetat numărul X la b până ajung la 0).
- Complexitatea este: $T(n) \in \Theta(n * log_2 n)$

Exemple

```
public static long factorial(int n) {
    if (n <= 1) {
       return 1;
    } else {
       return n * factorial (n-1);
    }
}
```

Exemple

- *factorial* este funcție recursivă (adică pentru a calcula *factorial(n)*, trebuie să calculăm *factorial(n-1)*.
- La funcții recursive, calculăm altfel complexitatea decât la funcții nerecursive.
- Pornim de la relația de recurență (adică scriem valoarea lui $T(n)$ în funcție de valoarea lui $T(n-1)$.

$$T(n) = \begin{cases} 1 & \text{dacă } n <= 1 \\ \text{T(n-1)} + 1 & \text{altfel} \end{cases}$$

Exemple

- Deocamdată ne ocupăm cu ramura recursivă

$$T(n) = T(n-1) + 1$$
$$T(n-1) = T(n-2) + 1$$
$$T(n-2) = T(n-3) + 1$$
$$\ldots$$
$$T(2) = T(1) + 1$$

- Acum adunăm toate aceste ecuații. O mare parte dintre termeni ($T(n-1)$, $T(n-2)$, ... $T(2)$) apar și în partea stângă și în partea dreaptă, deci se simplifică.

$$T(n) = T(1) + 1 + 1 + 1 \ldots + 1$$

Exemple

- $T(1)$ este 1 (prima ramură din relația recursivă)
- Câte $+1$ avem?
  - Avem câte un $+1$ pentru fiecare ecuație. Șiavem în total $n-1$ ecuații.

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

- Deci $T(n) \in \Theta(n)$
- **Obs:** La funcții nerecursive, nu are importanță dacă avem un $+1$ sau nu (pentru că oricum avem în general termeni mai mari). La funcții recursive trebuie să avem acest termen, altfel nu vom avea rezultatul corect.

Exemple

```
public static void recursiveFun4(int n, int m, int o) {
    if (n<=0) {
      System.out.println(m + o);
    } else {
      recursiveFun4(n-1, m+1, o);
      recursiveFun4(n-1, m, o+1);
    }
}
```

Exemple

```
public static void recursiveFun4(int n, int m, int o) {
    if (n<=0) {
      System.out.println(m + o);
    } else {
      recursiveFun4(n-1, m+1, o);
      recursiveFun4(n-1, m, o+1);
    }
}
```

- Începem din nou cu formula de recurență

$$T(n) = \begin{cases} 1 & \text{dacă } n <= 0 \\ 2*T(n-1) + 1 & \text{altfel} \end{cases}$$

Exemple

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

- Vrem din nou ca la final să adunăm ecuațiile, și vrem să dispară majoritatea termenelor. Pentru asta ne trebuie ca acum să scriem cât e $2 * T(n-1)$

$$2 * T(n-1) = 2^2 * T(n-2) + 2$$
$$2^2 * T(n-2) = 2^3 * T(n-3) + 2^2$$
$$2^3 * T(n-3) = 2^4 * T(n-4) + 2^3$$
$$\dots$$
$$2^{n-2} * T(2) = 2^{n-1} * T(1) + 2^{n-2}$$
$$2^{n-1} * T(1) = 2^n * T(0) + 2^{n-1}$$

Exemple

- Adunăm ecuațiile:

$$T(n) = 2^n * T(0) + 1 + 2 + 2^2 + 2^3 + ... + 2^{n-1}$$

$$T(n) = 1 + 2 + 2^2 + 2^3 + ... + 2^{n-1} + 2^n$$

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

Exemple

```
public staticint recursiveFun5(int n) {
   for (i = 0; i < n; i += 2) {
      // do something
   }
   if (n <= 0) {
      return 1;
 } else {
       return 1 + recursiveFun5(n/5);
   }
}
```

Exemple

```
public staticint recursiveFun5(int n) {
   for (i = 0; i < n; i += 2) {
      // do something
   }
   if (n <= 0) {
      return 1;
 } else {
       return 1 + recursiveFun5(n/5);
   }
}
```

$$T(n) = \begin{cases} 1 & \text{dacă } n <= 0 \\ T(n/5) + n & \text{altfel} \end{cases}$$

Exemple

- Când apelul recursiv se face pentru n împărțit la un număr c, înainte de a începe calculele, notăm valoarea lui n ca fiind c^k.
- În cazul nostru, presupunem că $n = 5^k$

$$T(5^k) = T(5^{k-1}) + 5^k$$
$$T(5^{k-1}) = T(5^{k-2}) + 5^{k-1}$$
$$T(5^{k-2}) = T(5^{k-3}) + 5^{k-2}$$
$$\ldots$$
$$T(5^2) = T(5^1) + 5^2$$
$$T(5^1) = T(5^0) + 5$$

Exemple

- Adunăm ecuațiile

$$T(5^k) = T(1) + 5 + 5^2 + ... + 5^{k-2} + 5^{k-1} + 5^k$$

$$T(5^k) = 1 + 5 + ... + 5^{k-1} + 5^k = \frac{5^{k+1} - 1}{5 - 1}$$

$$\frac{5^k * 5 - 1}{4} = \frac{5n - 1}{4} \in \Theta(n)$$