Structuri de algoritmi și date
(curs 4 - S2)

08.10.2024

Capitolul IV. Analiza algoritmilor

Un algoritm reprezintă o metodă pentru a obține rezultatul dorit.

Un algoritm constă din:
- Un **set de date inițiale** care abstractizează contextul problemei de rezolvat
- Un **set de relații de transformare** care sunt operate pe baza unor reguli al căror conținut și a căror succesiune reprezintă însăși substanța algoritmului
- Un **set de rezultate** preconizate sau informații finale, care se obțin de regulă trecând printr-un șir de informații intermediare

Proprietățile unui algoritm:
- **Generalitate** – un algoritm nu rezolvă doar o anume problemă ci o clasă generică de probleme de același tip
- **Finitudine** – informația finală se obține din cea inițială trecând printr-un număr finit de transformări
- **Unicitate** – transformările și ordinea în care ele se aplică sunt univoc determinate de regulile algoritmului

Consecință: Ori de câte ori se aplică același algoritm asupra aceluiași set de date inițiale se obțin aceleași rezultate

Scopul: analiza performanței algoritmilor

Un pas important in analiza algoritmilor este determinarea resurselor(de timp si spatiu) necesare pentru rularea acestuia.

Algoritmii se compara in termen de eficienta.

La ce servește analiza algorimilor?

Permite precizarea **predictivă** a comportamentului algoritmilor

Prin analiză pot fi **comparați** diferiți algoritmi

Cum putem compara doi algoritmi?

Empiric, implementând ambii algoritmi pe un sistem de calcul, rulând programele cu aceleași seturi de date de intrare, măsurând resursele (de timp și spațiu) utilizate și comparând rezultatele

Printr-o **analiză asimptotică**

Dezavantajele folosirii metodei **empirice**:
- Efortul investit în implementarea unor algoritmi care nu vor fi folosiți
- Pot aparea diferențe de performanță din modul cum este scris codul, independente de diferențele de performanță ale algoritmilor
- Seturile de date de intrare pot favoriza un algoritm în detrimentul altuia
- Se poate ca niciunul din algoritmii comparați să nu se încadreze în bugetul de resurse

Analiza asimptotică ⟶ o metodă de estimare

O **resursă** critică pentru un program este **timpul de rulare**

În practică nu putem să ne exăm exclusiv pe această resursă, ci trebuie să ținem cont și de spațiul de memorie (cea operativă și cea de stocare)

Există o serie de factori care influențează timpii de rulare (frecvența CPU, limbajul de programare, mediul de compilare și rulare, etc.), cu toate acestea ei nu sunt relevanți pentru compararea performanței algoritmilor

Analiza algoritmilor se bazează de regulă pe **ipoteze**:

- Sistemele de calcul sunt considerate convenționale, adică ele execută câte o singură instrucțiune la un moment dat
- Timpul total de execuție al algoritmului rezultă din însumarea timpilor instrucțiunilor individuale care îl alcătuiesc

Exemplul 1:

Considerăm un algoritm simplu de găsire a valorii maxime într-un tablou unidimensional:

```
// Returneaza pozitia valorii maxime din vectorul "A" de dimensiune "n"
int largest(int A[], int n) {
   int index_max = 0; // in variabila se retine pozitia valorii maxime
   for (int i=1; i<n; i++)      // pentru fiecare element din vector
      if (A[index_max] < A[i])       // daca A[i] este mai mare
      index_max= i;            // retinem pozitia
   return index_max;           // returneaza pozitia valorii maxime
}
```

Complexitate: $O(n)$

Timp: $T(n) = c \cdot n$

```
for(int i = 0; i < n; i++){
      a[i] = i + 1; //avem o(n)
}

for(int i = 0; i < 100; i++){
      a[i] = i + 1; //avem o(1)
}
```

$\rightarrow T(100) = 100c = c \Rightarrow O(1)$

```
for(int i = 0; i < n; i++){
      for(j = 0; j < i; j++){
            a[i] = j + 1; //avem o(n^2)
            }
}
```

În cazul precedent dimensiunea datelor de intrare este n, numărul de elemente din tablou

Timpul de rulare nu depinde de valoarea maximă sau de valorile din tablou ci de dimensiunea acestuia

Asfel putem exprima timpul de rulare T ca o funcție de n (T(n))

Dacă considerăm constant timpul necesar unei comparații, atunci timpul total este:

$$T(n) = cn$$

Exemplul 2:

Considerăm o funcție care returnează prima valoare dintr-un tablou:

```
int first(int A[], int n) {
   return A[0];         // returneaza prima valoare
}
```

$\rightarrow O(1) \rightarrow$ complexitate

În acest caz timpul de rulare nu mai depinde de dimensiunea tabloului ci este constant

$$T(n) = c$$

Exemplul 3:

Considerăm următoarea funcție:

```
int summ(int A[], int n) {
   sum = 0;
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
      sum+=A[j];
   return sum;
   }
```

Complexitate $O(n^2)$

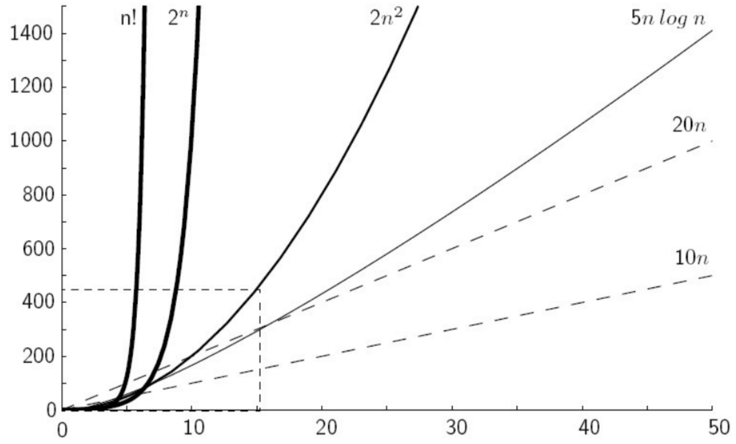
Considerând că timpul necesar pentru o operație de adunare este constant, timpul de rulare a funcției precedente este:

$$T(n) = cn^2$$

Rata de creștere a timpului de rulare al unui algoritm reprezintă costul de timp în funcție de dimensiunea datelor de intrare

Funcția de timp poate fi constantă, poate crește liniar, pătratic, logaritmic etc.

Rata de creștere a timpului de rulare ne ajută să comparăm diferiți algoritmi



Observăm că pentru o valoara de intrare destul de mare valorile timpilor de execuție sunt influențate mai mult de forma funcției de creștere (pe ordinul de mărime) decât de constantele care apar în funcții

Astfel, în analiza algoritmilor ne axăm pe determinarea ordinului de mărime al timpului de executie al unui program, nu pe aflarea timpilor exacti de executie

Ordinul de mărime se determină prin studiul **eficienței asimptotice**

Cel mai **favorabil**, cel mai **defavorabil** caz și cazul **mediu**
- Există și funcții pentru care nu avem o variație a dimensiunii datelor de intrare și totuși timpul de rulare diferă, de data aceasta în funcție de valorile datelor de intrare (ex. factorial (n))
- Ex: În cazul unei căutări secvețiale într-un tablou, elementul căutat se poate afla pe prima poziție (cel mai favorabil caz), pe ultima pozitie (cel mai defavorabil caz). In medie algoritmul de căutare secvențială face n/2 verificări (acesta este considerat cazul mediu)

Când analizăm un algoritm ce caz ar trebui luat in considerare?
- De obicei se consideră cel mai **defavorabil** caz
- În unele situații se consideră și cazul mediu

Avantajelul analizării cazului cel mai **defavorabil**
- Determinăm o limită superioară pentru performanța acestuia (in toate celelalte cazuri, algoritmul se comportă mai bine decât pentru cazul cel mai defavorabil). Are o deosebită aplicabilitate în siteme timp real (ex. control trafic aerian, sisteme anti-racheta).

Dezavantaj:
- Nu constituie o analiză reprezentativă pentru cazurile în care costurile de execuție trebuie agregate (ex. însumate)

Avantajul analizării cazului **mediu**
- Constituie o analiză reprezentativă pentru cazurile în care costurile de execuție trebuie agregate (ex. însumate)

Dezavantaje:
- Nu este întotdeauna posibilă determinarea cazului mediu
- Distribuția datelor joacă un rol esențial în determinarea cazului mediu

Notatii asimptotice

În **studiul eficienței asimptotice**, ne interesează cu precădere **limita** la care tinde timpul de execuție al algoritmului odată cu creșterea nelimitată a dimensiunii intrării

În **analiza asimptotică** se folosesc o serie de notații

Notația Θ (theta)

fct. $g(n) \longrightarrow \Theta(g(n)) \longrightarrow$ o mulțime de funcții

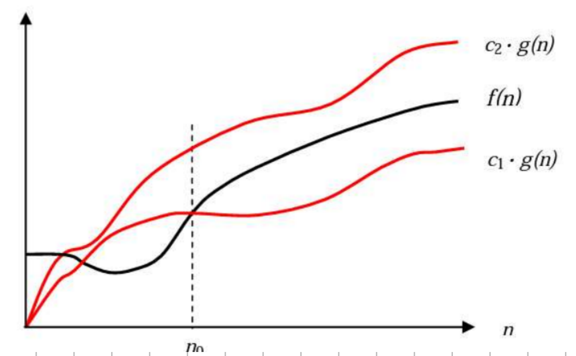
$$\Theta(g(n)) = \{ f(n) : \exists \text{ const} > 0 \ (c_1, c_2, n_0) \text{ a.î. } 0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \ \forall n \ge n_0 \}$$

Se poate spune că $f(n) \in \Theta(g(n))$

$\iff \exists\, c_1, c_2$ a.î. $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$, n suficient de mare

Cu alte cuvinte funcția f(n) este mărginită inferior de $c_1g(n)$ și superior de $c_2g(n)$

Se spune ca g(n) este o margine **asimptotică strânsă** ("asymptotically tight bound") a lui f(n).



$f(n) = \Theta(g(n))$

În practică, determinarea lui Θ în cazul unei expresii polinomiale, se realizează luând în considerare termenii de ordinul cel mai mare și neglijând restul termenilor

Exemplu:

$$\overbrace{\frac{1}{2} \cdot n^2 - 3 \cdot n}^{f(n)} = \Theta(n^2)$$

Constantele c_1 , c_2 și n_0 trebuie determinate astfel încât, pentru orice $n \ge n_0$ să fie valabilă relația:

$$c_1 \cdot n^2 \le \frac{1}{2} \cdot n^2 - 3 \cdot n \le c_2 \cdot n^2$$

Egalând c_1=1/4, c_2=1/2 și n_0=12 se poate verifica relația

Deoarece o funcţie polinominală de grad zero este o **constantă**, despre orice funcţie constantă se poate spune ca este $\Theta(n^0)$ sau $\Theta(1)$.

Deşi acesta este un abuz de interpretare (deoarece *n* nu tinde la infinit), prin **convenţie** $\Theta(1)$ desemnează fie o ***constantă*** fie o ***funcţie constantă*** în raport cu o variabilă.

$\Rightarrow$ pentru timp constant $\Theta(1) = \Theta(n^0)$

$O(1)$ sau $O(n^0)$

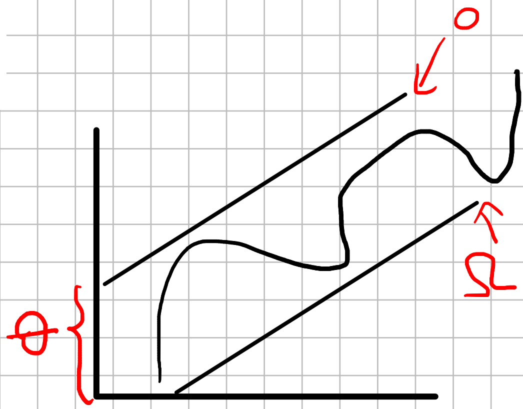
$\nexists$ alte constante în $O(\ldots)$

for(k = 1; k <= n; k *= 2) $\longrightarrow O(\log_2 n)$

Notația O (O mare)

Notația O desemnează **marginea asimptotică superioară** a unei funcții. Pentru o funcţie dată f(n), se defineşte O(g(n)) ca şi mulţimea de funcţii:

$$O(g(n)) = \{f(n) : \exists \text{ constantele pozitive } c \text{ şi } n_0 \text{ astfel încât } 0 \le f(n) \le c \cdot g(n), \text{pentru } \forall\, n \ge n_0\}$$



Notația O:
- se utilizează pentru a desemna o **margine superioară** a unei funcții în interiorul unui factor constant
- reprezintă o **limită superioară de creștere** a unei funcții

Faptul că f(n) este Θ(g(n)) implică că f(n) = O(g(n)) deoarece notaţia Θ este mai puternică decât notaţia O. Formal acest lucru se precizează prin relaţia:

$$\Theta(g(n)) \subseteq O(g(n))$$

Deoarece s-a demonstrat faptul că orice funcţie pătratică $a \cdot n^2 + b \cdot n + c$, $a > 0$ este $\Theta(n^2)$, rezultă ca această funcţie este implicit şi $O(n^2)$

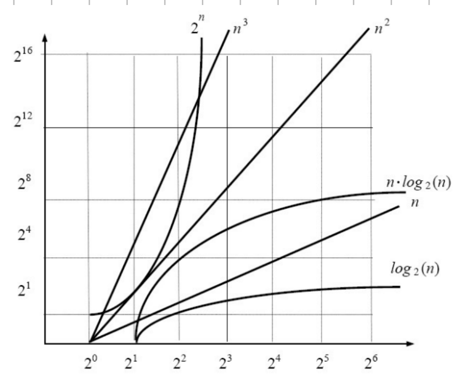
Notaţia O este de obicei cea mai utilizată în aprecierea **timpului de execuţie al algoritmilor** respectiv a **performanţei** acestora

Uneori ea poate fi estimată direct din **inspectarea structurii algoritmului**, spre exemplu existenţa unei bucle duble conduce de regulă, la o margine de ordinul $O(n^2)$

Deoarece notaţia O descrie o **margine superioară** şi atunci când este utilizată, ea mărgineşte **cazul cel mai defavorabil** de execuţie al unui algoritm

Prin implicaţie, ea **mărgineşte superior** comportamentul algoritmului în aceeaşi măsură pentru **orice** altă intrare

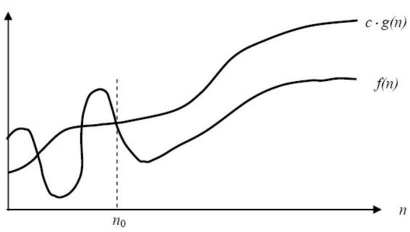
Propoziția **nu** este valabilă și pentru **Θ**



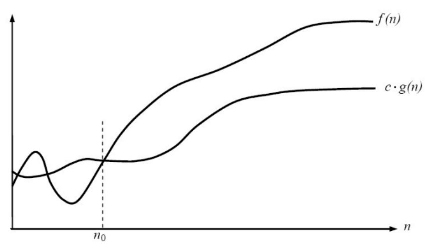
Ordine de mărime ale notației O

Notația Ω (Omega mare)
- Notaţia Ω precizează o **margine asimptotică inferioară.** Pentru o funcţie dată f(n), prin Ω(g(n)) se precizează mulţimea funcţiilor:

$$\Omega(g(n)) = \{f(n) : \exists \text{ constantele pozitive } c \text{ şi } n_0 \text{ astfel încât } 0 \le c \cdot g(n) \le f(n), \text{pentru } \forall\, n \ge n_0\}$$



f(n) = O(g(n))



f(n) = Ω(g(n))

Teoremă: Pentru oricare două funcţii f(n) şi g(n) , f(n) = (g(n))Θ dacă şi numai dacă f(n) = O(g(n)) şi f(n) = Ω(g(n))

Deoarece notaţia Ω descrie o **limită inferioară**, atunci când este utilizată pentru a mărgini cazul cel mai favorabil de execuţie al unui algoritm, prin implicaţie ea **mărgineşte inferior** orice intrare arbitrară a algoritmului.

Notația o (o mic)
- Marginea asimptotică superioară desemnată prin notaţia O, poate fi din punct de vedere asimptotic strânsă sau lejeră (laxă).
- Pentru desemnarea unei **margini asimptotice lejere** se utilizează notaţia o (o mic).

Principala diferenţă dintre notaţiile O şi o rezidă în faptul că în cazul *f(n)* = O(*g(n)*), marginea $0 \le f(n) \le c \cdot g(n)$ este valabilă pentru **anumite constante** $c > 0$, în timp ce *f(n)* =o(*g(n)*), marginea $0 \le g(n) < c \cdot g(n)$ este valabilă pentru **orice constantă** $c > 0$

În notaţia o, funcţia *f(n)* devine nesemnificativă în raport cu *g(n)* când *n* tinde la infinit

$$f(n) = o(g(n)) \text{implică} \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Notația ω (omega mic)
- Prin analogie, notaţia ω este pentru notaţia Ω ceea ce este o pentru O.
- Cu alte cuvinte notaţia ω precizează o **margine asimptotică inferioară lejeră**.
- În relaţia *f(n) = ω(g(n)), f(n)* devine arbitrară în raport cu *g(n)* atunci când *n* tinde la infinit.

$$f(n) = \omega(g(n)) \text{implică} \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Proprietăți ale notațiilor asimptotice:

Tranzitivitate:
- $f(n) = \Theta(g(n))$ şi $g(n) = \Theta(h(n))$ implică $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ şi $g(n) = O(h(n))$ implică $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ şi $g(n) = \Omega(h(n))$ implică $f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$ şi $g(n) = o(h(n))$ implică $f(n) = o(h(n))$
- $f(n) = \omega(g(n))$ şi $g(n) = \omega(h(n))$ implică $f(n) = \omega(h(n))$

Reflexivitate:
- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Simetrie:
- $f(n) = \Theta(g(n))$ dacă și numai dacă $g(n) = \Theta(f(n))$

Simetrie transpusă
- $f(n) = O(g(n))$ dacă și numai dacă $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ dacă și numai dacă $g(n) = \omega(f(n))$

Reguli de simplificare:
1. Dacă f(n) este în O(g(n)) și g(n) este în O(h(n)), atunci f(n) este în O(h(n))
2. Dacă f(n) este în O(kg(n)), pentru orice constantă k>0, atunci f(n) este în O(g(n))
3. Dacă f1(n) este în O(g1(n)) și f2(n) este în O(g2(n)), atunci f1(n)+f2(n) este în O(max(g1(n),g2(n)))
4. Dacă f1(n) este în O(g1(n)) și f2(n) este în O(g2(n)), atunci f1(n)·f2(n) este în O(g1(n)·g2(n))

Prima regulă spune că dacă o funcție g(n) reprezintă o limită superioară pentru funcția de cost f(n), atunci orice limită superioară a funcției g(n) reprezintă o limită superioară și pentru f(n)

O proprietate similară este valabilă și pentru notația Ω: dacă o funcție g(n) reprezintă o limită inferioară pentru funcția de cost f(n), atunci orice limită inferioară a funcției g(n) reprezintă o limită inferioară și pentru f(n)

A doua regulă spune că putem ignora orice constantă multiplicativă din ecuație când folosim notația O.

Regula este valabilă și pentru notațiile Ω și Θ

A treia regulă spune că dacă avem două părți de program care rulează secvențial, trebuie să luăm în considerare partea cea mai costisitoare

Regula este valabilă și pentru notațiile Ω și Θ

A patra regulă este folosită pentru a analiza bucle simple în program. Dacă o acțiune se repetă de un număr de ori și acțiunea are același cost de fiecare dată, atunci costul total este costul unei acțiuni multiplicat cu numărul de repetiții ale acelei acțiuni. → c

Regula este valabilă și pentru notațiile Ω și Θ

Compararea funcțiilor
- Fiind date două funcții f(n) și g(n), ale căror rată de creștere este exprimată sub formă de ecuații. Am vrea să determinăm care din ele crește mai repede.
- Putem afla care din ele are rata de creștere mai mare aflând următoarea limită

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

- Dacă limita tinde la infinit, atunci f(n) este în Ω(g(n)), deoarece f(n) crește mai repede
- Dacă limita tinde la zero, atunci f(n) este în O(g(n)), deoarece g(n) crește mai repede
- Dacă limita tinde spre o constantă, alta decât zero, atunci f(n) = Θ(g(n)), deoarece ambele cresc cu aceeași rată

$f(n) = n^2 + n$

$g(n) = n^3$

$\Downarrow$

cazul 2

Aprecierea timpului de execuție

Cu ajutorul notației **O** (O mare) se poate aprecia **ordinul timpului de execuție** al unui algoritm

Se face sublinierea că aprecierea se referă atât la ordinul timpului de execuţie al unui **algoritm abstract** cât şi cel al unui **program** real rezultat din implementarea respectivului algoritm.

Timpul efectiv de execuţie al unui program, **depinde** de mai mulţi factori cum ar fi:
- (1) **Dimensiunea şi natura** datelor de intrare.
- (2) **Caracteristicile sistemului de calcul** pe care se rulează programul.
- (3) **Eficienţa codului** produs de compilator.

Notaţia O mare permite eliminarea factorilor care nu pot fi controlaţi, cum ar fi spre exemplu (2) şi (3) enumeraţi mai sus, concentrându-se asupra comportării algoritmului independent de program.

În general un algoritm a cărui complexitate temporală este $O(n^2)$ va rula ca şi program în $O(n^2)$ unităţi de timp indiferent de limbajul sau sistemul de calcul utilizat.

În aprecierea timpului de execuţie se porneşte de la **ipoteza simplificatoare** deja enunţată, că fiecare instrucţie utilizează în medie aceeaşi cantitate de timp

Instrucţiunile care **nu pot fi încadrate** în această medie de timp sunt:
- Instrucţiunea **IF**
- Secvenţele **repetitive** (buclele)
- **Apelurile** de funcţii.

Presupunând pentru moment că apelurile de funcţii se ignoră, se consideră **cel mai defavorabil caz** şi se adoptă prin convenţie următoarele **simplificări**:
- Se presupune că o instrucţiune IF va consuma întotdeauna timpul necesar **execuţiei ramurii celei mai lungi**, dacă nu există raţiuni contrare justificate;
- Se presupune că întotdeauna instrucţiunile din interiorul unei bucle se vor executa de **numărul maxim** de ori permis de condiţia de control

Ex1:

Atribuirea unei valori pentru o variabilă de tip întreg

a = b;

Pentru că timpul pentru executarea instrucțiunii de atribuire este constant, avem Θ (1) => O(1)

Ex2:

Considerăm o buclă for:

```
int sum=0;
for (int i =0; i<n;i++)
        sum+=n;
```

Prima linie este Θ(1), implicit și O(1).

Bucla se repetă de n ori. Durata execuției celei de-a treia linii este constantă. Din regula de estimare a buclelor => Θ(n) pentru întregul cod.

Ex3:

Considerăm mai multe bucle:

```
sum = 0;
for (i=1; i<=n; i++)     // Prima bucla
    for (j=1; j<=i; j++)     // O bucla dubla
        sum++;
for (k=0; k<n; k++)     // A doua bucla
    A[k] = k;
```

> const. } $c_3 \cdot i$ de n ori

$\Rightarrow c_3 \sum_{i=1}^{n} i = c_3 \frac{n(n+1)}{2} =$

$= c_3 \frac{n^2+n}{2}$

$\longrightarrow \Theta(n^2)$

Acest exemplu are trei secvențe:
- O atribuire
- Două bucle

sum=0 ult. for

Prima secvență este c1 = Θ(1) și ultima c2*n = Θ(n), a doua secvență se calculează astfel:
- sum++ necesită timp constant
- For-ul interior se execută de i ori => c3*i, iar cel exterior de n ori
- Costul total = $c_3 \cdot \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ = Θ(n^2), din regula de insumare avem Θ(n^2) pentru întregul cod

Nu toate buclele duble sunt **$\Theta(n^2)$**

Ex4:

```
sum1 = 0;
for (k=1; k<=n; k*=2) // se repetă de log n ori
      for (j=1; j<=n; j++) // se repetă de n ori
            sum1++;
```

$$\sum_{k=1}^{\log_2 n} n \Rightarrow \Theta(n \cdot \log_2 n)$$

Presupunem că n este putere a lui 2.

Primul for din bucla dublă se execută de log n + 1

Al doilea for se execută de n ori

Costul total = $\sum_{i=0}^{\log n} n$ = Θ(n log n)

Buclele while și do – while se analizează într-un mod asemănător cu bucla for

Pentru instrucțiunile if și switch se determină ramura ce reprezintă cazul cel mai defavorabil și se calculează timpul de execuție pentru acea ramură

Determinarea timpului de execuție pentru o subrutină recursivă poate fi dificilă

De obicei se determină o relație de recurență pentru calcularea timpului de execuție

Ex1: funcția factorial
- T(n) = T(n-1) + c, pentru n>1, T(1) = c

S-a demonstrat în cursul anterior că timpul de execuție este direct proporțional cu n => Θ(n)

Ex2: Căutarea binară
- T(n) = T(n/2) + 1 pentru n > 1; T(1) = 1.

  => T(n) = log n => Θ(n)

relația de recurență

caz de bază

Ex3: Parametrii multiplii
- Fie o figură cu P pixeli, care pot avea un cod de culoare intre 0 și C-1. Să se afle numărul de pixeli de fiecare culoare, apoi sa se sorteze culorile în funcție de numărul de pixeli de fiecare culoare

```
for (i=0; i<C; i++) // initializare
   count[i] = 0;
for (i=0; i<P; i++) // Parcurgere pixeli
   count[value(i)]++; /* incrementarea contorului pentru
culoarea value(i) */
sort(count, C); //functia de sortare
```

Ex3: Parametrii multiplii
- În exemplul anterior vectorul count are dimensiunea C
- Pentru prima buclă avem Θ(C), pentru a doua avem Θ(P)
- Pentru funcția de sortare, costul depinde de algoritmul implementat. Să presupunem că folosim o sortare cu Θ(n log n) => Θ(C log C) pentru sortarea colorilor
- Aceasta ne duce la un cost total de Θ (P+C log C). ?
- Putem să considerăm Θ (C log C)? Depinde de relația intre C și P, dacă C este mult mai mic, P are o influență mai mare decât C log C. Niciuna dintre variabile nu poate fi ignorată în acest caz

Aprecierea costului de memorie

Pe lângă timp, spațiul (de memorie) este o altă resursă pentru care ne interesează costul

Metodele de analiză pentru costul spațiului de memorie ocupat sunt similare cu cele pentru analiza costului de timp

În timp ce costurile de timp sunt determinate pentru un algoritm ce folosește un anumit tip de structuri de date, costurile de spațiu de memorie sunt în mod normal determinate pentru structurile de date efective

Analiza asimptotică se aplică și pentru spațiul de memorie în același mod ca pentru timpii de rulare

Ex1: Care sunt costurile de spațiu de memorie pentru reținerea a n valori întregi, dacă fiecare întreg ocupa c octeți.

Dacă fiecare întreg ocupa c octeți, atunci n întregi ocupa c*n octeți => $\Theta(n)$

Proiectarea algoritmilor presupune de cele mai multe ori un compromis între spațiu și timp

Ex1: Prin compresia datelor reducem spațiul utilizat, dar creștem costul de timp prin adăugarea de timpi suplimentari pentru conpresie/decompresie

Ex2: Un tabel de valori pentru o funcție reduce timpii necesari recalculării acelor valori, dar solicită spațiu pentru memorarea valorilor

În plus, accesul la spațiul de stocare extern include costuri de timp suplimentare

Profilarea unui algoritm

Presupunem că un algoritm a fost conceput, implementat, testat şi depanat pe un sistem de calcul ţintă.

Ne interesează de regulă profilul performanţei sale, adică timpii precişi de execuţie ai algoritmului pentru diferite seturi de date, eventual pe diferite sisteme ţintă.

Pentru aceasta sistemul de calcul ţintă trebuie să fie dotat cu un ceas intern şi cu funcţii sistem de acces la acest ceas

Se presupune un algoritm implementat în forma unui program numit Algoritm(X: Intrare,Y: Iesire)unde X este intrarea iar Y ieşirea.

Pentru a construi profilul algoritmului este necesar să fie concepute:

- (1) **Seturile de date** de intrare a căror dimensiune creşte între anumite limite, pentru a studia comportamentul algoritmului în raport cu dimensiunea intrării.
- (2) Seturile de date de intrare care în principiu se referă la **cazurile extreme** de comportament.
- (3) O funcție cu ajutorul căreia poate fi construit **profilul algoritmului** în baza seturilor de date anterior amintite.

Funcția Profil poate fi utilizată în mai multe scopuri funcţie de obiectivele urmărite.

- (1) Evidenţierea performanţei intrinseci a unui algoritm precizat.
- (2) Evidenţierea performanţei relative a doi sau mai mulţi algoritmi diferiţi care îndeplinesc aceeaşi sarcină.
- (3) Evidenţierea performanţei relative a două sau mai multe sisteme de calcul.

Concluzii

Facem diferența între limite superioare și limite inferioare, când nu cunoaștem cu exactitate rata de creștere a funcției cost

Folosim notația Θ, când vrem să indicăm că nu sunt diferențe majore între ratele de creștere ale limitelor inferioară și superioară pentru o funcție cost dată

Funcțiile O, Ω și Θ nu determină costurile de timp efective, ci determină limite de creștere a unei funcții de cost