1. ALGORITMI SI COMPLEXITATE

1.1. ALGORITMI

Definiţie: *Un algoritm este o procedură (o mulțime finită de reguli bine definite) care îndeplineşte un obiectiv precis. Algoritmul pleacă de la o stare inițială şi se termină într-o stare finală.*

Un exemplu simplu de algoritm este rețeta de bucătărie. Majoritatea algoritmilor sunt mai complecşi: au paşi care se repetă, necesită decizii. Mai mult, algoritmii pot fi compuşi pentru a crea alți algoritmi.

Inițial conceptul de algoritm a reprezentat o procedură pentru rezolvarea unei probleme matematice (ex. găsirea divizorului comun a două numere, înmulțirea a două numere).

În forma sa actuală, conceptul a fost formalizat de către Alan Turing şi Alonzo Church în lucrările lor "*Maşini Turing*" şi respectiv "*Lambda Calcul*".

Algoritmii sunt esențiali în procesarea informației în calculator deoarece un program de calculator este practic, un algoritm. Din acest punct de vedere, algoritmul poate fi considerat ca "*o secvență de operații care pot fi executate de către un sistem Turing complet*". Există definiții ale noțiunii de algoritmi care îi leagă de "*Maşina Turing*" ($1.2).

Definiţie: *Un algoritm este un proces de calcul definit de către o maşină Turing (Gurevich, 2000).*

Deoarece procesează informație, algoritmul citeşte date de la un dispozitiv şi poate înregistra (salva) date necesare altor procesări. Datele salvate sunt privite ca o parte a stării interne a algoritmului. În practică astfel de stări sunt înregistrate în "*structuri de date*". Un algoritm necesită şi date interne pentru operații specifice. Acestea se numesc "*tipuri de date abstracte*".

Un algoritm trebuie să se termine sau nu ?

Unii autori restricționează definiția algoritmului la o procedură care se termină. Alții consideră că algoritmii sunt proceduri care pot să se execute la infinit, o astfel de procedură numindu-se "metodă de calcul". În aceste cazuri se consideră că algoritmul trebuie să “genereze obiecte”.

Exemplu: Un algoritm care verifică dacă există mai mult de un zero într-o secvență infinită de cifre binare aleatoare trebuie să se execute la infinit.

1.1.1. Exprimarea algoritmilor

Algoritmii pot fi exprimați prin mai multe moduri:

- limbaj natural;
- pseudocod;
- scheme logice;
- limbaje de programare.

Exprimarea prin limbaj natural tinde să fie complicată şi, uneori, ambiguă.

Pseudocodul şi schemele logice sunt modalități structurate de exprimare a algoritmilor şi sunt independente de limbajul de programare.

În final, limbajul de programare este preferat deoarece reprezintă exprimarea algoritmului într-o formă ce poate fi executată de către calculator.

Exemplul 1: Un algoritm care găseşte cel mai mare număr dintr-o listă de numere.

Limbaj natural:

1. Admitem că primul *element* este cel mai mare.

2. Examinează fiecare din elementele rămase în listă şi dacă este mai mare decât "*cel mai mare*", atunci notează acest fapt (notează-l ca fiind "*cel mai mare*").

3. Ultimul element notat este "*cel mai mare*".

Pseudocod :

Intrare: O listă L de elemente nevidă

Ieşire: Cel mai mare element din lista L

cel_mai_mare $\leftarrow L_0$

for element în $L_{>1}$ **do**

if (element > cel_mai_mare) **then**

cel_mai_mare ← element

end_if

return cel_mai_mare

Exemplul 2: Algoritmul lui Euclid pentru aflarea celui mai mare divizor comun a două numere naturale.

Limbaj natural:

Fiind date două numere naturale a şi b, testează dacă b este zero. Dacă da, a este cel mai mare divizor comun. Dacă nu, repetă procesul utilizând pe b şi ce rămâne după împărțirea lui a la b.

Pseudocod:

function cmmdc(a, b)

if $b = 0$ **then return** a

else return cmmdc(b, a mod b)

end_if

sau

function cmmdc(a, b)

while $a \neq b$ **do**

if $a > b$ **then** $a \leftarrow a - b$

else $b \leftarrow b - a$

end_if

end_while

Vom demonstra corectitudinea acestui algoritm.

Fie $a,b \in N, a > b \Rightarrow a = qb + r$. Să arătăm că orice divizor al lui a şi b este divizor şi pentru r. Avem $r = a - qb$. Dacă d este un divizor al lui a şi b, fie $a = sd$, $b = td$. Rezultă $r = sd - qtd = (s - qt)d$. Deci d este divizor al lui r. Cum orice divizor al lui a şi b este divizor şi pentru r atunci şi cel maim mare divizor se referă şi la r. Este suficient să continuăm procesul cu numerele b şi r. Cum r este mai mic ca b în valoare absolută, vom găsi $r = 0$ într-un număr finit de paşi.

1.1.2. Clasificarea algoritmilor

În multe cazuri se doreşte a se recunoaşte, pentru un algoritm dat, câte resurse particulare sunt necesare (timp de execuție, memorie de stocare). Este vorba despre "*analiza algoritmilor*". Aceasta depinde de "*clasificarea algoritmilor*".

Clasificare funcție de implementare:

- *recursivi* (se invocă pe ei înşişi) / *iterativi* (au construcții repetitive);
- *logici* (controlează deducții logice);
- *seriali* (un singur procesor şi o singură instrucțiune executată la un moment dat) / *paraleli* (mai multe procesoare care executa instrucțiuni în acelaşi timp) ;
- *deterministici* (o decizie exactă la fiecare pas)/ *nedeterministici* (rezolvă problema plecând de la presupuneri);

Clasificare fucție de "design" (metodă), ex.:

- *împarte şi stăpâneşte* (împarte problema în una sau mai multe instanțe mai mici);
- *programare dinamică* (caută structuri optimale - o soluție optimală a problemei poate fi construită plecând de la soluțiile optimale ale subproblemelor);
- *metoda "Greedy"* (caută tot structuri optimale cu deosebirea că soluțiile la subprobleme nu trebuie să fie cunoscute la fiecare pas; alegerea *"Greedy"* se referă la *"ce arată"* mai bine la un moment dat).

Clasificare funcție de domeniul de studiu – algoritmii sunt:

de căutare, de sortare, numerici, algoritmi de grafuri, de geometrie computațională, de învățare automată, de criptografie, etc.

Clasificare funcție de complexitate

Unii algoritmi se încheie într-un timp linear, alții în timp exponențial, iar alții nu se încheie. Din acest punct de vedere algoritmii sunt clasificați în clase de echivalență bazate pe complexitate.

In continuare noi vom fi interesați de acest ultim tip de clasificare.

Pentru a formaliza conceptele legate de *"Complexitatea Algoritmilor"* este nevoie de un model computațional. Un astfel de model este *"Maşina Turing"*.

1.2.MASINA TURING

Definiția 1. *Un alfabet* Σ *este o mulțime nevidă şi finită de elemente, numite simboluri:* $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$.

Definiția 2. *Un cuvânt peste* Σ *este un* t *-uplu* $x = \langle \sigma_{i_1} \sigma_{i_2} .... \sigma_{i_k} \rangle$ *sau mai simplu* $\sigma_{i_1} \sigma_{i_2} .... \sigma_{i_k}$.

Definiția 3. *Un limbaj* L *peste* Σ *este o submulțime a lui* Σ^*. *Complementul lui* L *este* $L^c = \Sigma^* \setminus L$. *Cuvintele în* Σ^* *pot fi puse în ordine lexicografică. Astfel:*

– *pentru* $\forall n \in N$ *cuvintele de lungime n precedă cuvintele de lungime* $(n+1)$;

– *pentru orice lungime, ordinea este alfabetică.*

Maşina Turing constă din următoarele:

a. $k \geq 1$ benzi infinite în două direcții. Benzile sunt constituite dintr-un număr infinit de celule în ambele direcții. Fiecare bandă are o celulă distinctă, *"celula de start"* sau *"celula 0"*. În fiecare celulă a fiecărei benzi, se poate scrie un simbol al unui alfabet Σ. Se poate admite şi *"celula goală"*, care conține *"simbolul nul"* al alfabetului.

b. Fiecare bandă are un cap de citire/scriere şi fiecare pas al acestuia se face pe bandă. Deplasarea poate fi ***L****eft*, ***R****ight*, ***S****tay*.

c. O unitate centrală care este un automat finit. Stările acestuia sunt din mulțimea Q. Există o stare distinctă *"START"* şi una *"STOP"*. Unele maşini Turing iau în considerație o submulțime F a lui Q *ca* fiind mulțimea de stări finale. La fiecare pas, automatul se află

într-o stare q_i, având ca intrări k simboluri (de pe cele k benzi) pe care capetele de citire le citesc (simbolurile curente). Ieşirea este reprezentată de alte k simboluri pe care capetele le scriu pe benzile corespunzătoare (simbolurile noi) si o noua stare q_j. Fiecare capăt excută apoi o deplasare sânga, dreapta sau stă pe loc.

Obs: Pe o bandă, simbolul curent şi cel nou nu sunt neapărat diferite.
Rezumând, o maşină Turing deterministică cu k benzi ($k \geq 1$) este quint-uplul $\langle Q, \Sigma, I, q_0, F \rangle$, unde:

Q: mulțimea finită de stări;

Σ: mulțime finită – alfabetul (unde $\in \Sigma$ este simbolul nul);

I: o mulțime finită de elemente $\langle q, s, s', m, q' \rangle$, unde $q, q' \in Q^*$, $s, s' \in \Sigma^k$, $m \in \{L_{eft}, R_{ight}, S_{tay}\}^k$ mişcările pe cele k benzi.

q_0 este starea inițială. Vom mai considera şi $F \subset Q$ o mulțime de stări finale.

Quint-uplul $\langle q, s, s', m, q' \rangle$ este un pas (o instrucțiune) al (a) maşinii Turing unde:

q: starea curentă;

$s = (s_1, s_2, ...s_k)$: cele k simboluri curente luate în considerație pe cele k benzi;

$s' = (s_1', s_2', ..., s_k')$: cele k simboluri care trebuie scrise;

$m = (m_1, m_2, ..., m_k)$: mişcările pe cele k benzi.

q': starea nouă.

Enumeram câteva elemente care caracterizează o mşină Turing.
Intrarea maşinii Turing: cele k cuvinte de pe cele k benzi, scrise inițial începand cu celulele *"zero"*.
Ieşirea maşinii Turing: cele k cuvinte scrise pe benzi în momentul când maşina Turing se opeşte. În mod frecvent, suntem interesați doare de unul singur.
Starea globală a maşinii Turing o notăm cu S = {starea curentă a automatului, conținutul curent al celor k benzi, pozițiile curente ale celor k capete}.
Notăm cu S_0 starea globală inițială.
Calculul determinist al maşinii Turing T pentru intrarea x (unde $x = (x_1, ..., x_k)$, iar $x_1, ..., x_k$ sunt cuvinte) îl notăm cu $T(x)$ şi este secvența de stări globale care începe cu starea inițială S_0, iar fiecare stare globală urmează alteia în şir.

Rezultă că maşina Turing deterministă este caracterizată de faptul că are cel mult o intrare pentru orice combinație de simboluri şi stări.
Dacă x este un cuvânt de intrare, atunci spunem că procesul de calcul al maşinii Turing, pentru x, *se opreşte* dacă $T(x)$ este finită şi ultima stare este finală.

Cuvâtul x este *acceptat* de maşina Turing T dacă $T(x)$ este finită şi dacă ultima stare este o stare de acceptare. Vom considera două stări finale mai importante:

- q_A: starea de acceptare;
- q_R: starea de rejectare.

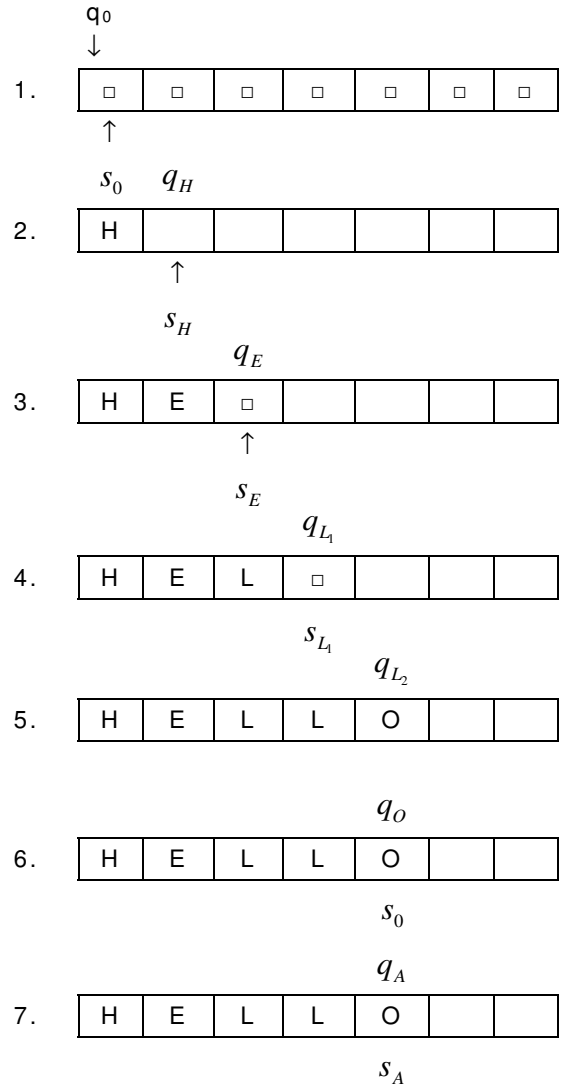
Pentru maşina Turing T notăm cu $L(T) = \{x \in \Sigma^* / T(x) \text{ se incheie cu } q_A\}$, limbajul acceptat de maşina T.

Maşina Turing *nedeterministă* este caracterizată de faptul că pentru orice stare q şi simbol s, oricare dintre stările din Q poate fi stare următoare. Unei stări globale S îi poate urma orice stare globală P.

Prezentăm câteva exemple de maşini Turing. Vom descrie instrucțiunile astfel: (stare_curenta, simbol_curent) → (stare_noua, simbol_nou, increment).

Exeplul 1. Maşină Turing care nu are la intrare niciun cuvânt iar la ieşire tipăreşte cuvântul “HELLO”.

Stare curentă	***Simbol curent***	***Stare nouă***	***Simbol nou***	***Increment***
q_0	□	q_H	H	+ 1
q_H	□	q_E	E	+ 1
q_E	□	q_{L_1}	L	+ 1
q_{L_1}	□	q_{L_2}	L	+ 1
q_{L_2}	□	q_O	O	0
q_O	0	q_A	O	0



Exemplul 2. Maşina Turing care determină cuvintele polindrom ale alfabetului $\{0,1\}$.

Stare curentă	***Simbol curent***	***Stare nouă***	***Simbol nou***	***Increment***

s_0	0	q_0	□	+ 1
q_0	0	q_0	0	+ 1
q_0	1	q_0	1	+ 1
q_0	□	q_{R_0}	□	− 1
s_0	1	q_1	□	+ 1
q_1	0	q_1	0	+ 1
q_1	1	q_1	1	+ 1
q_1	□	q_{R_1}	□	− 1
q_{R_0}	0	q_{REV}	□	− 1
q_{R_1}	1	q_{REV}	□	− 1
q_{R_0}	1	q_R	□	0
q_{R_1}	0	q_R	□	0
q_{REV}	0	q_{REV}	0	− 1
q_{REV}	1	q_{REV}	1	− 1
q_{REV}	□	s_0	□	+ 1
s_0	□	q_A	□	0

Mulțimea stărilor este $Q = \{q_0, s_0, q_R, q_A, q_{REV}, q_{R_1}, q_{R_0},\}$.

In următoarele exemple de calcul pentru această maşină, cuvântul *s* = 0110 va fi acceptat iar cuvântul *s* = 0111 va fi rejectat.

P1: s_0 , 0 1 1 0
↑

P2: q_0 , □ 1 1 0
↑

P3: q_0 , □ 1 1 0
↑

P4: q_0 , □ 1 1 0
↑

P5: q_0 , □ 1 1 0 □
↑

P6:	q_{R_0},	□	1	1	□	□
					↑	
P7:	q_{REV},	□	1	1	□	□
				↑		
P8:	q_{REV},	□	1	1	□	□
			↑			
P9:	q_{REV},	□	1	1	□	□
P10:	s_0,	□	1	1	□	□
			↑			
P11:	Q_1,	□	□	1	□	□
				↑		
P12:	Q_1,	□	□	1	□	□
					↑	
P13:	q_{R_1},	□	□	1	□	□
				↑		
P14:	q_{REV},	□	□	□	□	□
			↑			
P15:	s_0,	□	□	□	□	□
				↑		
P16:	q_A					
		Accept				

P1:	s_0,	0	1	1	1	
P2:	q_0,	□	1	1	1	
			↑			
P3:	q_0,	□	1	1	1	
				↑		
P4:	q_0,	□	1	1	1	
					↑	
P5:	q_0,	□	1	1	1	□
						↑
P6:	q_{R_0},	□	1	1	1	□
					↑	
P7:	q_R,	□	1	1	□	□
			Reject		↑	

In continuare dăm enunțul a două teoreme.

Teorema 1. *Pentru orice* $k \geq 1$ *şi orice alfabet* Σ*, există o maşină Turing cu* $(k+1)$ *benzi.*

Teorema 2. *Pentru orice maşină Turing* S *cu* k *benzi, există o maşină Turing* T *cu o bandă care înlocuieşte pe* S *în următorul sens: pentru orice cuvânt* $x \in \Sigma^*$*, maşina* S *se opreşte într-un număr finit de paşi la intrarea* x *dacă şi numai dacă* T *se opreşte la intrarea* x *şi, la oprire, acelaşi lucru este scris pe ultima bandă a lui* S *şi pe banda lui* T*. Mai mult, dacă* S *face* N *paşi, atunci:* T *face* $O\left(N^2\right)$ *paşi.*

O maşină Turing care este capabilă să simuleze orice altă maşină Turing, se numeşte *"Maşină Turing Universală"*.

Calculatoarele moderne, cu programe înregistrate sunt *"instanțe"* ale unei maşini Turing mai sofisticate numită *"Maşină Program cu acces aleator"* (Random Acces Stored Program Machine – RASP).

RASP înregistrează programul în memorie. Programul este o secvență finită de stări ale maşinii (numite şi instrucțiuni). RASP are un umăr infinit de registrii – celule de memorie care pot conține orice întreg.

RASP este caracterizată de *"adresarea indirectă"* – conținutul unui registru poate *"arăta"* spre adresa oricărui alt registru.

1.3. MĂSURAREA COMPLEXITĂȚII

Există două tipuri de măsură a complexității algoritmilor.

Măsura statică este bazată pe structura algoritmului.
Măsura dinamică este bazată pe calitatea algoritmului, intrările lui, comportamentul calculatorului în timpul execuției algoritmului.
De interes este măsura diamică.
În general, o măsură a complexității Φ este independentă de modelul computațional dacă, în raport cu o mulțime $\{P_i\}$ de algoritmi, are proprietățile:

1. $\forall i$ domeniul lui Φ_i este acelaşi cu al lui P_i, iar codomeniul este inclus în N, unde Φ_i este măsura lui P_i;

2. Există un predicat M, astfel încât: $M(i,x,m) \leftrightarrow \Phi_i(x) = m$, $\quad \forall i,\ x,\ m$.

Vom da o măsură a complexității din punct de vedere al timpului de execuție al unui algoritm. Vom folosi ca model computațional maşina Turing.

În general, când vorbim despre timpul de execuție al unui algoritm, ne gândim la numărul de paşi executați de procesul computațional (calculul determinist) asociat algoritmului. Când procesul de calcul nu se opreşte, considerăm timpul ca fiind nedefinit.

Definiție. *Analiza complexității determină timpul în care operațiile de bază ale unui algoritm sunt executate petru fiecare set de date de intrare.*

Sunt mai multe cazuri în care se determină complexitatea unui algoritm:

1. Cazul defavorabil, $W(n)$: în cât timp operațiile de bază sunt executate în cazul defavorabil.

2. Cazul cel mai bun, $B(n)$: în cât timp operațiile de bază sunt executate în cazul cel mai bun.

3. Fiecare caz, $T(n)$: în cât timp operațiile de bază sunt executate pentru fiecare caz.

4. Cazul mediu, $A(n)$: în cât timp operațiile de bază sunt executate în medie.

În continuare dăm câteva exemple de calcul a complexității.

Exemplul 1. Căutarea secvențială într-un tablou $s=[1,...,n]$:

```
begin
        locație ← 1
        while (locație ≤ n) and (s[locație] ≠ x) do
                locație ← locație + 1
        end_while
        if (locație > n) then
                locație ← 0  / * element negăsit */
        end_if
end
```

Să analizăm complexitatea în cele patru cazuri.

1. $W(n)=n+1 \Leftrightarrow$ elementul x nu este în tablou.

2. $B(n) = 1 \Leftrightarrow x$ se află în primul element.

3. $T(n)$ nu se calculează deoarece operațiile de bază nu sunt executate de același număr de ori pentru toate instanțele de dimensiune n.

4. Pentru $A(n)$:

Cazul 1. x este în tablou. Toate componentele tabloului au valori diferite. Rezultă că x poate fi găsit în fiecare dintre ele cu aceeași probabilitate, $\frac{1}{n}$. Rezultă că $A(n)=\sum_{k=1}^{n} k\cdot\frac{1}{n}=\frac{1}{n}\sum_{k=1}^{n} k=\frac{1}{n}\cdot\frac{n(n+1)}{2}=\frac{n+1}{2}$. $\frac{n+1}{2}$ este, practic, valoarea medie așteptată a comparațiilor.

Cazul 2. x se află sau nu în tablou. Fie *p* probabilitatea ca el să se afle în tablou. Probabilitatea ca el să se afle în unul din elemente este $\frac{p}{n}$, iar probabilitatea ca el să nu fie în tablou este $(1-p)$.

Dacă x se află pe poziția k, vom avea k treceri prin buclă, iar dacă x nu aparține tabloului, vom avea n treceri. Atunci

$$A(n)=\sum_{k=1}^{n} k\cdot\frac{p}{n}+n(1-p)=\frac{p}{n}\sum_{k=1}^{n} k+n(1-p)=$$

$$p\cdot\frac{n+1}{2}+n-np=\frac{p}{2}+\frac{p\cdot n}{2}+n-np=$$

$$=\frac{p}{2}+n-\frac{p\cdot n}{2}=\frac{p}{2}+n\left(1-\frac{p}{2}\right).$$

Pentru $p=1$ rezultă $A(n)=\frac{n+1}{2}$ și pentru $p=0{,}5$ rezultă $A(n)=3n+1$.

Exemplul 2. Sortarea unui tablou $s=[1,...,n]$ prin metoda interschimbării.

```
begin
        for i=1  to  n−1 do
            for  j=i+1  to  n do
                if (s[j]<s[i]) then
                        s[i] ↔ s[j]
                end_if
end
```

Notăm cu $T(n)$.numărul de pași executați de algoritm.

Operațiile de bază sunt: comparația $s[i]$ cu $s[j]$ şi schimbarea $s[i]$ cu $s[j]$.

Bucla exterioară se execută de $n-1$ ori.

La primul pas al buclei exterioare se vor executa $n-1$ paşi în bucla interioară.

La al doilea pas al buclei exterioare vor fi $n-2$ paşi în bucla interioară. Apoi $n-3$, $n-4$ paşi, etc.

Deci $T(n) = (n-1)+(n-2)+....+2+1 = \frac{n(n-1)}{2}$, iar complexitatea este $O(n^2)$.

Următoarele definiții reprezintă cazuri de mărginire a complexității.

Definiția 1. *Pentru o funcție* $f(n)$ *notăm cu* $O(f(n))$ *mulțimea funcțiilor* $g(n)$, *cu proprietatea că* $\exists\, c \in R$ *şi* $n_0 \in N$, *astfel încât* $g(n) \le c \cdot f(n)$, $\forall\, n \ge n_0$.

Exemplu. $f(n) = n^2$ şi $g(n) = n^2 + 10n$.

$n^2 + 10n \le c \cdot n^2$; $(1-c)n^2 + 10n \le 0$; $n[(1-c)n + 10] \le 0$; $n[(1-c)n + 10] \ge 0$

Pentru un $n_0 > \frac{10}{c-1}$, prima inegalitate este adevărată.

Definiția 2. *Pentru o funcție* $f(n)$ *notăm cu* $\Omega(f(n))$ *mulțimea funcțiilor* $g(n)$, *cu proprietatea că* $\exists c \in R$ *şi* $n_0 \in N_+$, *astfel încât* $g(n) \ge c \cdot f(n)$, $\forall\, n \ge n_e$.

Exemplul 1. $n_3 \in \Omega(n^2)$ pentru că $n^3 \ge 1 \cdot n$ pentru $\forall\, n \ge 1$.

Exemplul 2. $\frac{n(n+1)}{2} \in \Omega(n^2)$. Astfel,

$\frac{n(n+1)}{2} \ge \frac{n}{2} \cdot \frac{n+1}{2} \ge \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$ pentru $n \ge 2$.

Deci pentru $c = \frac{1}{4}$ şi $n_0 = 2$ avem îndeplinită condiția impusă în definiție.

Definiția 3. *Pentru o funcție* $f(n)$ *notăm cu* $\Phi(f(n)) = O(f(n)) \cap \Omega(f(n))$ *mulțimea funcțiilor* $g(n)$, *cu proprietatea că* $\exists c_1, c_2$ *şi* $n_0 \in N$ *astfel încât* $c_1 \cdot f(n) \le g(n) \le c_2 \cdot f(n)$, $\forall\, n \ge n_0$.

Exemple de clase de complexitate:

$O(n^2)$: $3\log(n)+8$, $4n^2$, $5n+7$, $6n^2+9$, $2n\log(n)$, $5n^2+2n$

$\Omega(n^2)$: $4n^2$, $4n^3+3n^2$, $6n^2+9$, $6n^6+n^4$, $5n^2+2n$, 2^n+4n

$\Phi(n^2)$: $4n^2$, $6n^2+9$, $5n^2+2n$

Prezentăm în continuare un algoritm de cautare care are o complexitate mai bună decât alogoritmul de cautare secvențială.

Căutarea binară în tabloul $s[1,...,n]$.

```
begin
    min ← 1; max ← n; locație ← 0;
    while (min ≤ max) and (locație = 0) do
            mijloc ← (min + max) div 2;
            if (x = s [mijloc]) then locație ← mijloc
            else if (x < s [mijloc]) then max ← mijloc−1
                 else min ← mijloc + 1
                 end_if
            end_if
```

```
        end_while
    end
```

Analiza complexității.

Considerăm cazul defavorabil $W(n)$. Presupunem că n este putere a lui 2, mai precis $n=2^k$. Prima instrucțiune din buclă înjumătățeşte dimensiunea tabloului de căutare. Aceasta se întâmplă de fiecare dată când se trece prin buclă.

Pentru cazul când x este mai mic sau mai mare decât toate elementele din listă vom arăta că $W(n) = \log(n)+1$.

Din înjumătățirea secvenței de căutare deducem:

$$W(n) = W\left(\frac{n}{2}\right) + 1$$

$$W(1) = 1$$

$$W(2) = W(1) + 1 = 2$$

$$W(4) = W(2) + 1 = 3$$

$$W(8) = W(4) + 1 = 3+1 = 4$$

$$W(16) = W(8) + 1 = 4+1 = 5$$

Să demonstrăm prin inducție că $W(n) = \log(n) + 1$.

Pentru $n = 1$: $W(1) = 1 = \log(1) + 1$

Presupunem $W(n) = \log(n) + 1$

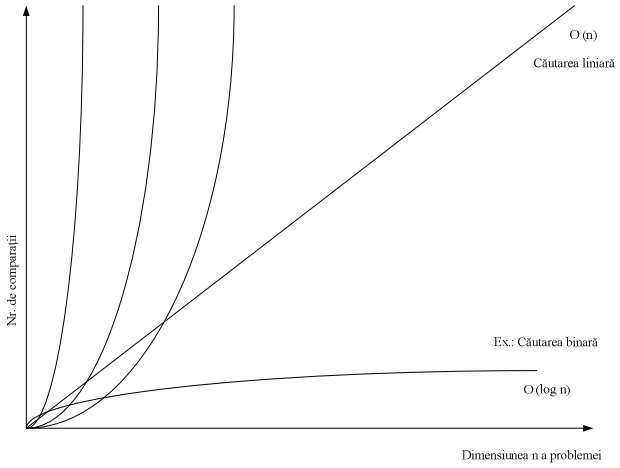
$W(2n) = W(n) + 1 = \log(n) + 1 + 1 = \log(n) + \log(2) + 1 =$
$\log(2n) + 1$

În cazul când $n \neq 2^k$ avem în general complexitatea $\lfloor \log(n)+1 \rfloor + 1 = O(\log(n))$.

Algoritmul de sortare binară este un exemplu de problemă *Divide-et-Impera* (**$Error! Reference source not found.**).

1.3.1. Tipuri de probleme şi complexitatea lor

$O(\log(n))$	apare când o problemă mai mare este rezolvată prin transformarea ei într-o problemă de dimesiune mai mică (tip logaritmic);
$O(n)$	apare când fiecare element al problemei necesită un mic effort (timp) de procesare (tip linear);
$O(n \lg(n))$	apare când o problemă este *"spartă"* în mai multe subprobleme rezolvabile independent, iar apoi sunt combinate soluțiile (tip linearitmic);
$O(n^2)$	apare când algoritmul procesează toate perechile de elemente ale unei mulțimi (tip quadratic);
$O(2^n)$	tip exponențial.



Primele trei curbe din stânga graficului de mai sus reprezintă în ordine (de la stânga la dreapta) complexitățile $O(2^n)$, $O(n^2)$ şi $O(n\lg(n))$.

Iată câteva cazuri practice de stabilire a complexității.

Cazul de rezolvare al unei singure instrucțiuni ce se execută o singură dată:

$O(1)$

Cazul de rezolvare:

for $i=1$ to n **do**

s;

unde s este $O(1)$. Rezultă complexitatea $n\cdot O(1)=O(n)$.

Cazul de rezolvare:

for $i=1$ to n **do**

for $j=1$ to n **do**

s;

unde s este $O(1)$. Rezultă complexitatea $n\cdot O(n)=O(n^2)$.

Cazul de rezolvare:

$h = 1$

while $(h \le n)$ **do**

s;

$h \leftarrow 2*h$

end_while

aici variabila h ia valorile 1,2,4,8..., până când depăşeşte pe n. Rezultă că sunt $1+\lfloor \log_2 n \rfloor$ treceri. Avem complexitatea $O(\log_2 n)$.

Cazul de rezolvare:

for $i=1$ to $n-1$ **do**

for $j=i$ to n **do**

s;

Bucla interioară se execută de *i* ori unde *i* = 1,2,...,*n*. Numărul de execuții este $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, deci complexitatea este $O(n^2)$.

Cazul de rezolvare:

```
h = n; j =1
while j < h do
   for i = 1 to n do
      s;
```

h ← h/2
end_while

Bucla interioară are complexitatea $O(n)$ iar bucla exterioară execută $\lfloor \log_2 n \rfloor$ iterații. Complexitatea este deci $O(n \log_2 n)$.

1.4. RECURSIVITATE

În general, spunem că un obiect sau o metodă este recursiv (ă) dacă se poate defini în funcție de el (ea) însuşi (însăşi). Mai precis, un astfel de obiect sau metodă, se poate defini prin:

– câteva cazuri simple sau metode simple;
– reguli care transformă cazurile complexe în cazuri simple.

Exemplu :

– orice părinte are un strămoş;
– părinții oricărui strămoş sunt de asemenea strămoşi ai persoanei considerate.

Cazul funcțiilor:

O funcție este definită recursiv dacă pornind de la anumite valori ale ei, se pot calcula alte valori prin autoapelarea funcției. Altfel spus, o funcție recursivă este parțial definită în termeni de ea însăşi.

Definiția unei funcții recursive trebuie să satisfacă *condiția de consistență*: valoarea funcției recursive trebuie să fie direct calculabilă sau calculabilă cu ajutorul unor valori direct calculabile.

În definirea unei funcții recursive trebuie să apară cel puțin o condiție de oprire din recursivitate.

Expresia unei fincții recursive este dată de o relație de recurență. Fie o secvență $\{t_n\}_{n \geq 1}$ şi $k \in N$. Dacă avem:

$$t_{n+k} = a_1 \cdot t_{n+k-1} + a_2 \cdot t_{n+k-2} + ... + a_k \, t_k$$

atunci secvența este o secvență recursivă de ordin k, iar relația precedentă este o relație recursivă de ordin k.

1.4.1. Exemple de funcții recursive

1. Formulă recursivă pentru algoritmul lui Euclid

Pentru $m, n \in N$, $m > n$ avem $cmmdc(m,n) = cmmdc(n, r_1)$, unde r_1 este restul împărțirii lui m la n. Deci $r_1 = m - n * \lfloor m/n \rfloor$.

Atunci $cmmdc\ (m,n) = cmmdc\ (n, m - n * \lfloor m/n \rfloor)$.

Algoritm Euclid scris ca funcție recursivă:

function $cmmdc\ (m,n)$
  if $(n = 0)$ **then** return m
    else
      return $cmmdc\ (n, m - n * \lfloor m/n \rfloor)$
    end_if
end_func

Apel*: răspuns* ← $cmmdc(a,b)$

Agoritmul Euclid scris ca procedură recursivă

procedure $cmmdc$ (in : m, n; out result)
  if $(n = 0)$ **then** result ← m
    else $cmmdc\left(n,\ m - n * \left[\frac{m}{n}\right];\ result\right)$

```
    end_if
end_proc.
```
Apel: $cmmdc(a, b\ ;\ raspuns)$

2. Funcția factorial

$fact : N \to N$

$$fact(n) = \begin{cases} 1, & daca\ n = 0 \\ n * fact\ (n-1), & daca\quad n \geq 1 \end{cases}$$

```
function  fact(n)
    if  (n=0)  then return  1
        else  return  n * fact(n-1);
    end_if
end_func.
```

Exemplu de calcul efectiv pentru funcția factorial:

```
factorial (3)
  factorial (2)
    factorial (1)
      factorial (0)  return  (0)
  return  2*1=2
    return  1*1=1
return  3*2=6
```

3. Funcția putere x^n

Pentru un întreg $x>0$ să se calculeze x^n, unde $n \geq 0$. Vom considera funcția $putere\ (x,n): N \times N \to N$. Evident $putere\ (x,0) = 1$. Atunci, expresia recurentă a funcției este:

$$putere\ (x,\ n) = \begin{cases} 1, & dacă\ n = 0 \\ x * putere\ (x,\ n-1), & dacă \quad n \geq 1 \end{cases}$$

Corectitudinea definiției rezultă din $x^0 = 1$ (pasul de verificare) şi $x^{n+1} = x \cdot x^n$ (pasul de inducție).

Afirmăm că funcția (1) are complexitatea $O\ (n)$.

Notăm cu $T(n)$ timpul de execuție a funcției putere.

$T\ (n) = T\ (n-1) + O\ (1)$, unde $O\ (1)$ este timpul necesar pentru a returna valoarea x.

$T\ (n) = T\ (n-1) + O\ (1) = T\ (n-2) + 2 * O\ (1) + ...$

$T\ (n) = T\ (0) + n * O\ (1)$.

Rezultă complexitatea $O\ (n)$.

Exprimând altfel funcția putere, vom obține un algoritm cu o complexitate mai bună.

Observăm că x^4 se poate calcula mai uşor dacă am calculat x^2, deoarece $x^4 = \left(x^2\right)^2$. La fel, $x^6 = \left(x^3\right)^2$. Notăm $\lfloor n/2 \rfloor = n\ div\ 2$

Considerăm $x^{n\ div\ 2} = putere\ (x, n\ div\ 2)$ şi notăm cu sqr ridicarea la puterea doi. Atunci

$$sqr\ (putere\ (x,\ n\ div\ 2)) = \begin{cases} x^{n-1}, & dacă\ n\ impar \\ x^n, & dacă\ n\ par \end{cases}$$

Atunci:

$$x^n = \begin{cases} x\ *\ sqr\ (putere\ (x,\ n\ div\ 2)), & dacă\ n\ impar \\ sqr\ (putere\ (x,\ n\ div\ 2)), & dacă\ n\ par \end{cases},$$

iar algoritmul complet este:

$$putere\ (x,\ n) = \begin{cases} 1, & dacă\ n = 0 \\ x * sqr\ (putere\ (x,\ n\ div\ 2)), & dacă\ n\ impar \\ sqr\ (putere\ (x,\ n\ div\ 2)), & dacă\ n\ par \end{cases}$$

Notăm cu $T(n)$ timpul de execuție pentru această formulă. Fie $n = 2^m$. Atunci:

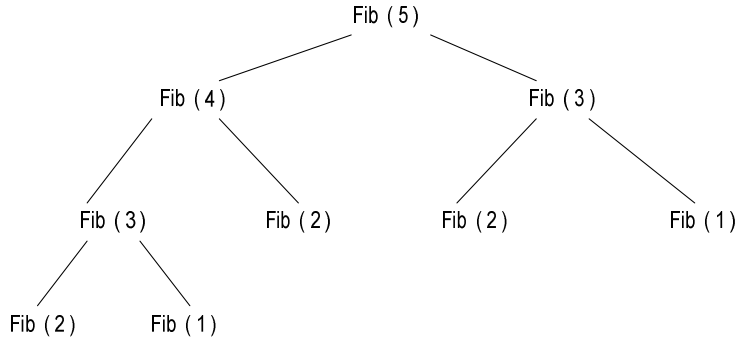
$$T(n) = \begin{cases} 1, & pentru\ n = 1 \\ T\left(\frac{n}{2}\right) + 1, & pentru\ n > 1 \end{cases}$$

$T(n) = T(2^m) = T(2^{m-1}) + 1 = T(2^{m-2}) + 2 = ... = T(2^0) + m = T(1) + m = m + 1 = \log_2 n + 1$ Rezultă complexitatea $O(\log_2 n)$.

4. Alte exemple de funcții recursive:

a) Fibonacci: $fib : N \to N$

$$fib(n) = \begin{cases} 1, & n = 1,\ n = 2 \\ fib\ (n-1) + (n-2), & n > 2 \end{cases}$$



b) Ackerman:

$$ack\ (m,n) = \begin{cases} n+1, & m = 0 \\ ack\ (m-1,\ 1), & n = 0 \\ ack\ (m-1,\ ack\ (m,\ n-1)), & rest \end{cases}$$

c) Manna – Punelli

$$mana\ (n) = \begin{cases} x-1, & x \geq 12 \\ f\ (f\ (x+2)), & x \in [0,\ 12) \end{cases}$$

1.4.2. Tipuri de recursie

Pentru exprimare, vom folosi funcțiile: $f(x)$, $g(x)$, $h(x)$, $i(x)$, $\varphi(u,v)$.

a) Recursia liniară

$$f\ (x) = \begin{cases} g\ (x), & dacă\ p\ (x) = true \\ \varphi\ (f\ (h\ (x)),\ i\ (x)), & altfel \end{cases}$$

Exemplu: $fact\ (n)$

b) Recursia neliniară de tip cascadă

$$f(x) = \begin{cases} g(x), & \text{dacă } p(x) = true \\ \varphi(f(h(x)), f(i(x))), & altfel \end{cases}$$

Exemplu: $fib(n) = \begin{cases} 1, & n = 1 \text{ sau } n = 2 \\ fib(n-1) + fib(n-2), & altfel \quad n > 2 \end{cases}$

c) Recursia neliniară de tip împachetat

$$f(x) = \begin{cases} g(x), & \text{dacă } p(x) = true \\ f(f(\ldots f(h(x))), \ldots), & altfel \end{cases}$$

Exemplu: $mana(n)$

1.4.3. Ecuații recurente

În multe cazuri, analiza complexității conduce la ecuații recurente şi la rezolvarea acestora. Prezentăm două tehnici generale pentru rezolvarea ecuațiilor recurente.

***Teorema* 1.** *Fie ecuația recurentă, omogenă şi liniară cu coeficienți constanți,* $a_0 t_n + a_1 \cdot t_{n-1} + \ldots + a_k t_{n-k} = 0$.

Dacă ecuația sa caracteristică, $a_0 r^k + a_1 \cdot r^{k-1} + \ldots + a_k r^0 = 0$, are k soluții distincte, $r_1, r_2, \ldots, r_k$, atunci singurele soluții ale ecuației recurente sunt de forma $t_n = c_1 r_1^n + c_2 r_2^n + \ldots + c_k r_k^k = 0$.

Exemplu: $t_n - 5t_{n-1} + 6t_{n-1} = 0$, $t_0 = 0$, $t_1 = 1$.

Ecuația caracteristică este $r^2 - 5r + 6 = 0$. Rezultă $r_1 = 3$, $r_2 = 2$. Soluția generală este $t_n = c_1 3^n + c_2 2^n$ şi din condițiile inițiale avem $t_n = 3^n - 2^n$.

***Teorema* 2.** *Fie* r *rădăcina de multiplicitate* m *pentru ecuația caracteristică a unei ecuații omogene, liniare şi recurente cu coeficienți constanți:*

$a_0 t_n + a_1 \cdot t_{n-1} + \ldots + a_k t_{n-k} = 0$

Atunci $t_n = r^n$, $t_n = nr^n$, $t_n = n^2 r^n, \ldots, t_n = n^{m-1} r^n$ *sunt toate soluții ale ecuații recurente. Fiecare dintre aceşti termeni trebuie introdus în soluția generală (prin adunare).*

Exemplu: Să se rezolve: $t_n - 7t_{n-1} + 15t_{n-2} + 9t_{n-3} = 0$, $t_0 = 0$, $t_1 = 1$, $t_2 = 2$.

Ecuația caracteristică este $r^3 - 7r^2 + 15r - 9 = 0$. Rezultă $r_1 = 1$, $r_{2,3} = 3$.

Soluția generală este $t_n = c_1 1^n + c_2 3^n + c_3 \cdot n \cdot 3^n$ şi din condiția inițială, selectăm soluția $t_n = -1 + 3^n - n3^{n-1}$.