Mirel COȘULSCHI

Bunicilor mei,
Constanța și Gheorghe Constantinescu

Mirel COȘULSCHI

ALGORITMICA GRAFURILOR ȘI APLICAȚII

Ediția a III-a

Editura Universitaria
Craiova, 2020

Referenți științifici:
Prof.univ.dr. Ion Iancu
Lect.univ.dr. Petre Băzăvan

Descrierea CIP a Bibliotecii Naţionale a României
COȘULSCHI, MIREL
Algoritmica grafurilor și aplicații / Mirel Coșulschi. – Ediția a 3-a.
Craiova: Universitaria, 2020
Conţine bibliografie
ISBN 978-606-14-1660-8

51

Prefaţă

Cu toată creşterea de popularitate a tutorialelor video, a cursurilor online, a prezentărilor PowerPoint, cărţile tipărite constituie, încă, principalul mijloc de răspândire al cunoştinţelor. În acest sens, vine să îşi susţină cauza şi cartea de faţă, ca modalitate de a deschide un zăvor ferecat pentru cei care bat la porţile cunoaşterii.

Lucrarea de faţă prezintă capitole de algoritmica grafurilor ce abordează aproape toate aspectele importante din domeniu. Primele elemente de teoria grafurilor datează cu mult înaintea apariţiei calculatoarelor, grafurile devenind o parte importantă a informaticii, odată cu dezvoltarea acesteia ca ştiinţă. Un graf poate fi utilizat pentru modelarea unui sistem distribuit, văzut sub forma unui număr de componente de calcul autonome, interconectate prin intermediul unei reţele de comunicaţii şi care cooperează în vederea realizării unor scopuri comune. Reţelele mobile ad-hoc, reţelele de senzori fără fir (wireless), structurile grid sau cloud sunt instanţe ale unui astfel de sistem distribuit, fiind întâlnite foarte des în ultimul timp. În aceeaşi idee menţionăm că Facebook este un exemplu de graf social sau Web-ul este un exemplu de multi-graf orientat.

Pe lângă conceptele teoretice, cititorul interesat de subiect va găsi peste 90 de algoritmi prezentaţi în lucrarea structurată pe 10 capitole. O parte din algoritmi sunt introduşi mai întâi la nivel conceptual şi, mai apoi, prezentaţi în detaliu în pseudo-cod. Mulţi algoritmi sunt analizaţi din punct de vedere al complexităţii, punându-se astfel la dispoziţia unui programator un criteriu de evaluare a unui algoritm în vederea alegerii sale pentru rezolvarea unei probleme sau situaţii. Bibliografia bogată conţine referinţe către lucrări importante din domeniu.

Materialul este prezentat într-o înlănţuire *teorie-algoritm- implementare-exemple-aplicaţii*, abordarea autorului înclinându-se spre metode de rezolvare a unor probleme cu caracter practic, al căror grad de dificultate variază de la simplu la dificil.

Cartea reprezintă un material de referinţă ce vine în sprijinul studenţilor care urmează cursul de algoritmica grafurilor la facultăţile de informatică, calculatoare, automatică, telecomunicaţii şi informatică economică, însă poate fi consultată în egală măsură de către elevii de liceu la profilul informatică sau de cititorul interesat de diferitele aspecte de implementare ale algoritmilor prezentaţi. De asemenea, prin topicile avansate prezentate în lucrare, constituie un material util studenţilor de la cursurile masterale şi doctorale.

Mulţumirile autorului se îndreaptă către referenţii ştiinţifici ai acestei lucrări pentru sugestiile şi observaţiile pertinente pe marginea manuscrisului.

Craiova, septembrie, 2020 *Mirel Coşulschi*

Capitolul 1

Introducere în algoritmi

Definiţia 1.1 *Algoritmul constituie o reprezentare finită a unei metode de calcul ce permite rezolvarea unei anumite probleme. Se poate spune că un algoritm reprezintă o secvenţă finită de operaţii, ordonată şi complet definită, care, pornind de la datele de intrare, produce rezultate.*

Termenul de *algoritm* îi este atribuit matematicianului persan Abu Ja'far Mohammed ibn Musa al-Khowarizmi (sec. VIII-IX), care a scris o carte de matematică cunoscută în traducere latină sub titlul de "*Algorithmi de numero indorum*", iar apoi ca "*Liber algorithmi*", unde termenul de "*algorithm*" provine de la "*al-Khowarizmi*", ceea ce literal înseamnă "*din orasul Khowarizm*". Matematicienii din Evul Mediu înţelegeau prin algoritm o regulă (sau o mulţime de reguli) pe baza căreia se efectuau calcule aritmetice: de exemplu în secolul al XVI-lea, algoritmii se foloseau la înmulţiri sau înjumătăţiri de numere.

Fiecare propoziţie ce face parte din descrierea unui algoritm este, de fapt, o comandă ce trebuie executată de cineva, acesta putând fi o persoană sau o maşină de calcul. De altfel, un algoritm poate fi descris cu ajutorul oricărui limbaj, de la limbajul natural şi până la limbajul de asamblare al unui calculator. Denumim *limbaj algoritmic* un limbaj al cărui scop este acela de a descrie algoritmi.

Algoritmul specifică succesiuni posibile de transformări ale datelor. Un *tip de date* poate fi caracterizat printr-o mulţime de valori ce reprezintă domeniul tipului de date şi o mulţime de operaţii definite peste acest domeniu. Tipurile de date pot fi organizate în următoarele categorii:

1. *tipuri de date elementare* (de exemplu tipul *întreg*, tipul *real*) - valorile sunt unităţi atomice de informaţie;

2. *tipuri de date structurate* (de exemplu tipul *tablou*, tipul *înregistrare*) - valorile sunt structuri relativ simple rezultate în urma combinaţiei unor valori elementare;

3. *tipuri de date structurate de nivel înalt* (de exemplu *stiva*) - se pot descrie independent de limbaj iar valorile au o structură mai complexă.

Un algoritm trebuie să posede următoarele *trăsături caracteristice*:

1. *claritate* - la fiecare pas trebuie să specifice operaţia pe care urmează să o efectueze algoritmul asupra datelor de intrare;

2. *corectitudine* - rezultatele trebuie să fie corecte;

3. *generalitate* - algoritmul trebuie să ofere soluţia nu numai pentru o singură problemă ci pentru o întreagă clasă de probleme;
4. *finitudine* - algoritmul trebuie să se termine într-un timp finit;
5. *eficienţă* - un algoritm poate fi utilizat numai în situaţia în care resursele de calcul necesare acestuia sunt în cantităţi rezonabile, şi nu depăşesc cu mult posibilităţile calculatoarelor la un moment dat.

Un *program* reprezintă implementarea unui algoritm într-un limbaj de programare.
Studiul algoritmilor cuprinde mai multe aspecte:

- *elaborare* - activitatea de concepere a unui algoritm are şi un caracter creativ, din această cauză nefiind posibilă deprinderea numai pe cale mecanică. Pentru a facilita obţinerea unei soluţii la o problemă concretă se recomandă folosirea tehnicilor generale de elaborare a algoritmilor la care se adaugă în mod hotărâtor intuiţia programatorului;
- *exprimare* - implementarea unui algoritm intr-un limbaj de programare se poate face utilizând mai multe stiluri: *programare structurată*, *programare orientată pe obiecte* etc.;
- *validare* - verificarea corectitudinii algoritmului prin metode formale;
- *analiză* - stabilirea unor criterii pentru evaluarea eficienţei unui algoritm pentru a-i putea compara şi clasifica.

Un model de reprezentare al memoriei unei maşini de calcul este acela al unei structuri liniare compusă din celule, fiecare celulă fiind identificată printr-o adresă şi putând păstra o valoare corespunzătoare unui anumit tip de dată. Accesul la celule este facilitat de variabile. O *variabilă* se caracterizează prin:

- *un identificator* - un nume ce referă variabila;
- *o adresă* - desemnează o locaţie de memorie;
- *un tip de date* - descrie tipul valorilor memorate în celula de memorie asociată.

1.1 Limbajul pseudocod

Limbajul natural nu permite o descriere suficient de riguroasă a algoritmilor, de aceea, pentru reprezentarea acestora se folosesc alte modalităţi de descriere precum:

- *scheme logice*;
- *limbajul pseudocod*.

În continuare vom prezenta pricipalele construcţii din cadrul limbajului pseudocod.

Intrări/ieşiri

Citirea datelor de intrare se poate realiza prin intermediul enunţului `Input`:
1: **Input** {lista variabile}
Afişarea rezultatelor este reprezentată cu ajutorul instrucţiunii `Output`:
1: **Output** {lista de valori}

Instrucţiunea de atribuire

Este instrucţiunea cel mai des utilizată într-un algoritm şi realizează încărcarea unei variabile (locaţii de memorie) cu o anumită valoare. Are următoarea sintaxă:

1: $<$ *variabila* $>\leftarrow<$ *expresie* $>$

unde $<$ *expresie* $>$ este o expresie aritmetică sau logică.

Se evaluează expresia $<$ *expresie* $>$ iar rezultatul se atribuie variabilei $<$ *variabila* $>$, memorându-se în locaţia de memorie asociată. Această variabilă trebuie să fie de acelaşi tip de dată cu expresia sau un tip de dată care să includă şi tipul expresiei.

O expresie este constituită din *operanzi* şi *operatori*. *Operanzii* pot fi variabile şi valori constante, iar *operatorii* pot fi:

- *operatori aritmetici* - $+$ (adunare), $-$ (scădere), $*$ (înmulţire), / (împărţire), $\hat{}$ (ridicare la putere), *div* (câtul împărţirii întregi), *mod* (restul împărţirii întregi);
- *operatori relaţionali* - $=$ (egal), $\neq$ (diferit), $<$ (strict mai mic), $\leq$ (mai mic sau egal), $>$ (strict mai mare), $\geq$ (mai mare sau egal);
- *operatori logici* - OR sau $\vee$ (disjuncţie), AND sau $\wedge$ (conjuncţie), NOT sau $\neg$ (negaţie).

Cea mai simplă expresie este formată dintr-o *variabilă* sau o *constantă* (operand). Expresiile mai complicate se obţin din operaţii efectuate între variabile şi constante. La scrierea expresiilor trebuie să se ţină cont de faptul că, în momentul evaluării lor, în primul rând se vor evalua expresiile din paranteze, iar operaţiile se execută în ordinea determinată de priorităţile lor.

Enunţuri de ramificare

```
1: if <expresie-booleana> then
2:     <instructiune1>
   [
3: else
4:     <instructiune2>
   ]
5: end if
```

```
1: if (a mod 2 = 0) then
2:     Output {"Numarul este par"}
3: else
4:     Output {"Numarul este impar"}
5: end if
```

Enunţul `if ... then ... else` evaluează mai întâi expresia booleană pentru a determina unul din cele două drumuri pe care le poate lua execuţia algoritmului. Ea poate include opţional o clauză `else`.

Dacă $<$*expresie-booleana*$>$ se evaluează la valoarea de adevăr `true`, atunci se execută $<$*instructiune1*$>$ şi se continuă cu următoarea instrucţiune după `if`. Dacă $<$*expresie-booleana*$>$ are valoarea de adevăr `false`, atunci se execută $<$*instructiune2*$>$. $<$*instructiune1*$>$ şi $<$*instructiune2*$>$ sunt instrucţiuni compuse ce pot să conţină, la rândul lor, o altă instrucţiune `if`.

Exemplul 1.1 *Un algoritm simplu este cel ce rezolvă ecuaţia de gradul I, $ax+b=0$, $a, b \in \mathbb{R}$ (algoritmul 1). Acesta se bazează pe rezolvarea matematică a ecuaţiei de gradul I:*

1. *dacă $a = 0$, atunci ecuaţia devine $0 \cdot x + b = 0$. Pentru cazul în care $b \neq 0$ avem $0 \cdot x + b = 0$, egalitate ce nu poate fi satisfăcută pentru nicio valoare a lui $x \in \mathbb{R}$ sau $\mathbb{C}$. Spunem, în acest caz, că ecuaţia este incompatibilă. Dacă $b = 0$, avem $0 \cdot x + 0 = 0$, relaţia fiind adevărată pentru orice valoare a lui $x \in \mathbb{R}$. Spunem că ecuaţia este compatibil nedeterminată.*

2. dacă $a \neq 0$, ecuaţia are o singură soluţie, $x_1 = -\frac{b}{a} \in \mathbb{R}$.

Algoritm 1 Algoritm pentru rezolvarea ecuaţiei de gradul I

```
1: Input {a, b}
2: if (a = 0) then
3:     if (b = 0) then
4:         Output { "Ecuatie compatibil nedeterminata" }
5:     else
6:         Output { "Ecuatie incompatibila" }
7:     end if
8: else
9:     x ← −b/a
10:    Output { "Solutia este:", x }
11: end if
```

Având acest algoritm drept model să se realizeze un algoritm pentru rezolvarea ecuaţiei generale de gradul al II-lea, $ax^2 + bx + c = 0$, unde $a, b, c \in \mathbb{R}$.

Enunţuri repetitive

Enunţurile repetitive permit descrierea unor prelucrări ce trebuie efectuate în mod repetat, în funcţie de poziţia condiţiei de continuare existând două variante de structuri repetitive: *structura repetitivă condiţionată anterior* şi *structura repetitivă condiţionată posterior*.

Structura repetitivă condiţionată anterior `while` (sau instrucţiune de ciclare cu test iniţial) are următoarea sintaxă:

```
1: while <expresie-booleana> do
2:     <instructiune>
3: end while
```

```
1: sum ← 0
2: i ← 1
3: while (i ≤ n) do
4:     sum ← sum + i
5:     i ← i + 1
6: end while
```

Cât timp <*expresie-booleana*> este *adevarată*, se execută <*instructiune*>; dacă <*expresie-booleana*> este *falsă* chiar la prima evaluare, atunci <*instructiune*> nu ajunge să fie realizată niciodată. Acest comportament este opus celui corespunzător structurii repetitive condiţionată posterior `repeat`, unde <*instructiune*> este executată cel puţin o dată. (Dacă o expresie are valoarea de adevăr `true` spunem atunci că expresia este *adevărată*; dacă o expresie are valoarea de adevăr `false` spunem atunci că expresia nu este adevărată - este falsă).

Exemplul 1.2 *Vom realiza un algoritm pentru calculul câtului şi restului împărţirii a două numere întregi, prin scăderi succesive (a se vedea algoritmul 2).*

Mai întâi, se iniţializează câtul cu valoarea zero (linia 3) şi restul cu valoarea deîmpărţitului (linia 4). Apoi, atâta timp cât restul este mai mare decât valoarea împărţitorului (liniile 5 - 8), vom incrementa câtul cu o unitate, şi decrementa restul cu valoarea împărţitorului. La final, sunt afişate valorile câtului şi restului.

Un caz particular de structură repetitivă condiţionată anterior este `for`, utilizată în cazul în care o instrucţiune sau un grup de instrucţiuni trebuie să se repete de 0 sau mai multe ori - numărul de repetiţii fiind cunoscut înainte de începerea sa. *Enunţurile repetitive bazate pe instrucţiunile `while` şi `repeat` sunt mult mai potrivite în cazul în care condiţia de terminare trebuie reevaluată în timpul ciclării (atunci când numărul de repetiţii nu este cunoscut apriori).*

Algoritm 2 Algoritm pentru calculul împărţirii întregi

```
1: Input {a, b}
2: if ((a > 0) ∧ (b > 0)) then
3:     cat ← 0
4:     rest ← a
5:     while (rest ≥ b) do
6:         rest ← rest − b
7:         cat ← cat + 1
8:     end while
9:     Output {cat, rest}
10: else
11:     Output {"Numerele trebuie sa fie strict pozitive!"}
12: end if
```

Instrucţiunea `for` are următoarea sintaxă:

```
1: for <Var> ← <Expresie1>, <Expresie2>,
       Step < p > do
2:     <instructiune>
3: end for
```

```
1: sum ← 0
2: for i ← 1, n do
3:     sum ← sum + i
4: end for
```

Comportamentul enunţului repetitiv `for` se descrie cel mai bine prin comparaţie cu enunţul repetitiv `while`. Astfel secvenţa următoare de limbaj pseudocod

```
1: for v ← e1, e2 STEP p do
2:     < instructiune >
3: end for
```

este echivalentă cu secvenţa ce conţine enunţul repetitiv `while`:

```
1: v ← e1
2: while (v ≤ e2) do
3:     <instructiune>
4:     v ← v + p
5: end while
```

e_1 reprezintă valoarea iniţială, e_2 limita finală, iar p pasul de lucru: la fiecare pas, valoarea variabilei v este incrementată cu valoarea variabilei p, valoarea ce poate fi atât pozitivă cât şi negativă. Dacă $p \geq 0$ atunci v va lua valori în ordine crescătoare, iar dacă $p < 0$ atunci v va primi valori în ordine descrescătoare.

În cazul în care pasul de incrementare este 1, atunci el se poate omite din enunţul repetitiv `for`, variabila contor fiind incrementată automat la fiecare repetiţie cu valoarea 1. Dacă pasul de incrementare p are valoarea 1 atunci avem următoarele situaţii posibile pentru construcţia `for`:

- $Expresie1 > Expresie2$: <instructiune> nu se execută niciodată;
- $Expresie1 = Expresie2$: <instructiune> se execută exact o dată;
- $Expresie1 < Expresie2$: <instructiune> se execută de $Expresie2 - Expresie1 + 1$ ori.

Exemplul 1.3 *Un caz destul de frecvent întâlnit în practică este cel în care se cere determinarea elementului de valoare maximă, respectiv minimă dintr-un şir de elemente. Acest şir de elemente poate fi păstrat într-o structură de date elementară, cunoscută sub numele de vector. Un vector este un caz particular de matrice având o singură linie şi n coloane.*

Algoritm 3 Algoritm pentru calculul elementului de valoare minimă dintr-un şir

1: **Input** $\{N, x_1, x_2, \ldots, x_N\}$
2: $min \leftarrow x_1$
3: **for** $i \leftarrow 2, N$ **do**
4: **if** $(min > x_i)$ **then**
5: $min \leftarrow x_i$
6: **end if**
7: **end for**
8: **Output** $\{ min \}$

Prezentăm în continuare algoritmul pentru determinarea elementului de valoare minimă ce aparţine unui şir de numere naturale (a se vedea algoritmul 3).

În acest algoritm se observă utilizarea enunţului repetitiv `for`*, întâlnit, în general, în situaţiile în care se cunoaşte apriori numărul de paşi pe care trebuie să-l realizeze instrucţiunea repetitivă.*

Mai întâi se iniţializează variabila min cu valoarea elementului x_1, presupunând că elementul de valoare minimă este primul element din cadrul vectorului X (linia 2). În continuare vom compara valoarea variabilei min cu valoarea fiecărui element din şir (liniile 3–7): dacă vom găsi un element a cărui valoare este mai mică decât cea a minimului calculat până la momentul curent (linia 4), vom reţine noua valoare în variabila min (linia 5).

Structura repetitivă condiţionată posterior `repeat ... until` prezintă următoarea sintaxă:

```
1: repeat
2:     <instructiune>
3: until <expresie-booleana>
```

```
1: sum ← 0
2: i ← 1
3: repeat
4:     sum ← sum + i
5:     i ← i + 1
6: until (i > n)
```

Atâta timp cât <*expresie-booleana*> este falsă, se execută <*instructiune*>. Se observă faptul că instrucţiunile dintre `repeat` şi `until` se vor executa cel puţin o dată.

Exemplul 1.4 *Algoritmul 4, prezentat în continuare, calculează suma primelor n numere naturale, $S = 1 + 2 + 3 + \ldots + n$.*

Notăm cu S_n suma primelor n numere naturale ($S_n = 1 + 2 + 3 + \ldots + n$). Aceasta se poate calcula într-o manieră incrementală astfel:

$$\begin{aligned} S_1 &= 1 \\ S_2 &= 1 + 2 = S_1 + 2 \\ S_3 &= (1 + 2) + 3 = S_2 + 3 \\ S_4 &= (1 + 2 + 3) + 4 = S_3 + 4 \\ &\ldots \\ S_n &= (1 + 2 + \ldots + n - 1) + n = S_{n-1} + n \end{aligned}$$

Observaţia 1.5 *1. Algoritmul 4 utilizează enunţul repetitiv cu test final,* `repeat ... until`*.*

Algoritm 4 Algoritm pentru calculul sumei primelor n numere naturale (prima variantă)

```
1: Input {N}
2: S ← 0
3: i ← 1
4: repeat
5:     S ← S + i
6:     i ← i + 1
7: until (i > N)
8: Output {S}
```

2. *Enunţurile $S \leftarrow 0$ şi $i \leftarrow 1$ au rolul de a atribui valori iniţiale variabilelor S şi i. Întotdeauna variabilele trebuie iniţializate corect din punctul de vedere al algoritmului de calcul. Variabila ce reţine rezultatul unui proces de însumări succesive se va iniţializa întotdeauna cu valoarea 0, valoare ce nu va influenţa rezultatul final. O variabilă ce păstrează rezultatul unei operaţii (sumă sau produs) se va iniţializa cu elementul neutru faţă de operaţia respectivă.*

3. *Atribuirile $i = i + 1$ şi $S = S + i$ (liniile 5 şi 6) sunt lipsite de sens din punct de vedere algebric. Însă, atunci când ne referim la un sistem de calcul electronic, aceste operaţii se referă la valoarea curentă şi la cea anterioară a unei variabile. De asemenea, trebuie să se facă o distincţie clară între operaţia de atribuire şi cea de egalitate: operaţia de atribuire ('$\leftarrow$') face ca valoarea curentă a variabilei din stânga operatorului de atribuire să fie iniţializată cu valoarea expresiei din dreapta acestuia (în expresia '$k \leftarrow i + 2$' variabila k primeşte valoarea rezultată în urma evaluării expresiei '$i + 2$'), pe când operaţia de egalitate verifică dacă valoarea elementului din stânga operatorului '$=$' (sau a expresiei aflate în partea stângă a operatorului) este egală cu valoarea expresiei din dreapta acestuia (de exemplu rezultatul evaluării expresiei $k = i + 2$) are valoarea de adevăr `true` dacă valoarea variabilei 'k' este egală cu valoarea rezultată în urma evaluării expresiei '$i + 2$' şi are valoarea de adevăr `false` în caz contrar).*

   Prin urmare, în cazul expresiei $i \leftarrow i+1$, valoarea curentă a variabilei i devine valoarea anterioară a aceleiaşi variabile incrementată cu o unitate.

Suma primelor n numere naturale se constituie în suma termenilor unei progresii aritmetice ce se poate calcula direct cu formula sumei, astfel $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. Având în vedere această identitate, algoritmul de calcul a sumei primelor n numere naturale se simplifică foarte mult (a se vedea algoritmul 5).

Algoritm 5 Algoritm pentru calculul sumei primelor n numere naturale (a doua variantă)

```
1: Input {N}
2: S ← n · (n + 1)/2
3: Output {S}
```

Proceduri şi funcţii

Procedurile sunt subrutine ale căror instrucţiuni se execută ori de câte ori acestea sunt apelate prin numele lor.

Apelarea procedurilor se face în unitatea de program apelantă prin numele procedurii, primit în momentul definirii, urmat de lista parametrilor actuali. Această listă trebuie să corespundă ca număr şi tip cu lista parametrilor formali, în situaţia în care există o listă de parametri formali în antetul subrutinei. Definiţia unei proceduri are următoarea sintaxă:

1: **procedure** <NUME>(<lista_parametri_formali>)
2: <instructiune>
3: **end procedure**

`lista_parametri_formali` (opţională) simbolizează o listă de identificatori (parametri) ce permite transferul datelor între *subrutina apelantă* şi *subrutina apelată*. Aceşti parametri se specifică prin nume (identificator) urmat, eventual, de tipul de dată al parametrului. Mai mulţi parametri de acelaşi tip pot fi grupaţi, folosindu-se drept separator virgula.

De fapt, lista parametrilor formali poate fi descrisă mai detaliat astfel:

1: **procedure** <NUME>(<PFI; PFO>)
2: **end procedure**

unde

`PFI` = lista parametrilor formali de intrare.

`PFO` = lista parametrilor formali de ieşire.

Enunţul de apel al unei proceduri are următoarea sintaxă:

1: **CALL** <nume>(PAI; PAO)

`PAI` - lista parametrilor actuali de intrare, ce corespund parametrilor formali de intrare. Corespondenţa se face de la stânga la dreapta şi trebuie să fie de acelaşi tip cu parametrii formali.

`PAO` - lista parametrilor actuali de ieşire.

Exemplul 1.6 *Să se realizeze un algoritm care determină cel mai mare divizor comun a două numere naturale.*

Reamintim câteva rezultate teoretice ce vor fi folositoare pentru înţelegerea procesului de calcul al algoritmului ce va fi prezentat.

Teorema 1.7 (Teorema împărţirii cu rest) *Pentru două numere întregi a şi b, cu $b \neq 0$, $\exists$ două numere întregi q şi r, unice, astfel încât:*

$$a = b \cdot q + r, 0 \leq r < |b|$$

unde a se numeşte deîmpărţitul, b împărţitorul, q câtul iar r restul.

Definiţia 1.2 *Cel mai mare divizor comun (notat `cmmdc`) a două numere naturale a şi b este un număr natural d cu proprietăţile:*

1. *$d|a$, $d|b$ (d este un divizor comun al lui a şi b);*
2. *$\forall d' \in \mathbb{N}$ astfel încât $d'|a$, $d'|b$ avem $d'|d$ (oricare alt divizor comun al lui a şi b, îl divide şi pe d).*

Observaţia 1.8

1. *$cmmdc(a, 0) = a$.*
2. *Dacă $cmmdc(a, b) = 1$ se spune că numerele a şi b sunt prime între ele.*
3. *Între cel mai mic multiplu comun şi cel mai mare divizor comun există următoarea relaţie:*

$$cmmmc(a, b) = \frac{a \cdot b}{cmmdc(a, b)}. \tag{1.1}$$

Metoda de calcul pentru a obţine cel mai mare divizor comun a două numere a şi b prin împărţiri succesive, cunoscută sub numele de algoritmul lui Euclid, se poate descrie astfel:

Se împarte a la b şi se reţine restul r. Dacă r este nul atunci cel mai mare divizor comun este b. În caz contrar, se împarte b la r şi se păstrează noul rest. Calculele se continuă, folosindu-se ca deîmpărţit vechiul împărţitor, iar, ca împărţitor, ultimul rest obţinut, până când restul obţinut devine nul.

Operaţiile anterioare pot fi transpuse prin intermediul următoarelor formule matematice:

$$\begin{aligned}
a &= bq_1 + r_1 && 0 \le r_1 < b \\
b &= r_1 q_2 + r_2 && 0 \le r_2 < r_1 \\
r_1 &= r_2 q_3 + r_3 && 0 \le r_3 < r_2 \\
&\dots \\
r_k &= r_{k+1} q_{k+2} + r_{k+2} && 0 \le r_{k+2} < r_{k+1} \\
&\dots \\
r_{n-2} &= r_{n-1} q_n + r_n && 0 \le r_n < r_{n-1} \\
r_{n-1} &= r_n q_{n+1} + r_{n+1} && 0 \le r_{n+1} < r_n
\end{aligned}$$

Aceste împărţiri nu se constituie într-un proces infinit, deoarece secvenţa de numere naturale $r_1, r_2, \dots, r_{n+1}, \dots$ este strict descrescătoare ($r_1 > r_2 > \dots > r_n > \dots$) şi mărginită inferior de 0. Rezultă că $\exists p \in \mathbb{N}$ astfel încât $r_p \neq 0$, şi $r_{p+1} = 0$.

Algoritm 6 Algoritmul lui Euclid pentru calculul cmmdc a două numere

```
1:  procedure CMMDC(x, y; b)
2:      a ← x
3:      b ← y
4:      if (b = 0) then
5:          b ← a
6:      else
7:          r ← a mod b
8:          while (r ≠ 0) do                ▷ Procesul se încheie atunci când r ia valoarea 0
9:              a ← b
10:             b ← r
11:             r ← a mod b
12:         end while
13:     end if
14: end procedure
```

Dacă analizăm întregul proces de calcul, se observă faptul că deîmpărţitul este împărţitorul de la etapa anterioară, iar împărţitorul este restul de la etapa anterioară. De asemenea, trebuie menţionat că în acest proces de calcul, câtul nu participă în mod activ.

Algoritmul 6 este un foarte bun exemplu de utilizare a instrucţiunii de ciclare cu test iniţial, `while`. Din analiza algoritmului reiese faptul că, mai întâi, se efectuează o evaluare a restului r (calculul său, liniile 7 şi 11), după care se testează condiţia egalităţii acestuia cu valoarea 0 (linia 8).

Funcţiile au aceeaşi sintaxă ca şi procedurile:

```
1: function <NUME>(<parametri>)
2:     <instructiune>
3: end function
```

Funcţiile pot fi apelate prin numele lor, ca termen al unei expresii.

1.2 Elemente de analiza algoritmilor

Gradul de dificultate al unei probleme P poate fi pus in evidenţă prin timpul de execuţie al algoritmului corespunzător şi/sau prin spaţiul de memorie necesar. Timpul de execuţie în cazul cel mai defavorabil ne dă durata maximă de execuţie a unui algoritm. Timpul mediu de execuţie se obţine prin însumarea timpilor de execuţie pentru toate mulţimile de date de intrare urmată de raportarea la numărul acestora.

Definiţia 1.3 *Timpul de execuţie în cazul cel mai defavorabil al unui algoritm A este o funcţie $T_A : \mathbb{N} \longrightarrow \mathbb{N}$ unde $T_A(n)$ reprezintă numărul maxim de instrucţiuni executate de către A în cazul unor date de intrare de dimensiune n.*

Definiţia 1.4 *Timpul mediu de execuţie al unui algoritm A este o funcţie $T_A^{med} : \mathbb{N} \longrightarrow \mathbb{N}$ unde $T_A^{med}(n)$ reprezintă numărul mediu de instrucţiuni executate de către A în cazul unor date de intrare de dimensiune n.*

Fiind dată o problemă P, o funcţie $T(n)$ se spune că este o *margine superioară* dacă există un algoritm A ce rezolvă problema P iar timpul de execuţie în cazul cel mai defavorabil al algoritmului A este cel mult $T(n)$.

Fiind dată o problemă P, o funcţie $T(n)$ se spune că este o *margine pentru cazul mediu* dacă există un algoritm A ce rezolvă problema P iar timpul mediu de execuţie al algoritmului A este cel mult $T(n)$.

Fiind dată o problemă P, o funcţie $T(n)$ se spune că este o *margine inferioară* dacă orice algoritm A ce rezolvă problema P va avea cel puţin un timp $T(n)$ pentru unele date de intrare de dimensiune n, atunci când n tinde la infinit ($n \to \infty$).

Definiţia 1.5 *Fiind dată o funcţie $g : \mathbb{N} \longrightarrow \mathbb{R}$ vom nota cu $O(g(n))$ mulţimea:*
$O(g(n)) = \{f(n) | \ \exists c > 0,\ n_0 \geq 0$ *a.î.* $0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0\}$.

Vom spune despre f că nu creşte în mod sigur mai repede decât funcţia g. Pentru a indica faptul că o funcţie $f(n)$ este un membru al mulţimii $O(g(n))$, vom scrie $f(n) = O(g(n))$, în loc de $f(n) \in O(g(n))$.

Observaţia 1.9 *Vom prezenta câteva proprietăţi ale lui $O(\cdot)$:*

- $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$.
  De exemplu pentru funcţia $f(n) = 7 \cdot n^5 - n^2 + 3 \cdot \log n$ aplicând regula valorii maxime vom avea: $O(f(n)) = O(\max\{7 \cdot n^5, n^2, 3 \cdot \log n\})$ adică $O(f(n)) = O(n^5)$.
- $O(\log_a n) = O(\log_b n)$*;*
- $f(n) = O(f(n))$ *(reflexivitate);*
- $f(n) = O(g(n))$ *şi* $g(n) = O(h(n))$ *atunci* $f(n) = O(h(n))$ *(tranzitivitate).*

Definiţia 1.6 *Fiind dată o funcţie $g : \mathbb{N} \longrightarrow \mathbb{R}$ vom nota cu $\Theta(g(n))$ mulţimea:*
$\Theta(g(n)) = \{f(n) | \ \exists c_1, c_2 > 0,\ \exists n_0 \geq 0$ *a.î.* $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0\}$.

Spunem că $g(n)$ este o margine *asimptotic tare* pentru $f(n)$.

Definiţia 1.7 *Fiind dată o funcţie $g : \mathbb{N} \longrightarrow \mathbb{R}$ vom nota cu $\Omega(g(n))$ mulţimea:*
$\Omega(g(n)) = \{f(n) | \ \exists c > 0,\ n_0 \geq 0$ *a.î.* $0 \leq c \cdot g(n) \leq f(n),\ \forall n \geq n_0\}$.

La fel cum O furnizează o delimitare *asimptotică superioară* pentru o funcţie, Ω furnizează o delimitare *asimptotică inferioară* pentru aceasta.

Teorema 1.10 *Pentru orice două funcţii $f(n)$ şi $g(n)$, avem $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ şi $f(n) = \Omega(g(n))$.*

Definiţia 1.8 *Fiind dată o funcţie $g : \mathbb{N} \longrightarrow \mathbb{R}$ vom nota cu $o(g(n))$ mulţimea:* $o(g(n)) = \{f(n) |\ \forall c > 0,\ \exists n_0 > 0 \text{ a.î. } 0 \leq f(n) < c \cdot g(n),\ \forall n \geq n_0\}$.

Observaţia 1.11 $f(n) = o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Definiţia 1.9 $f(n) \in \omega(g(n)) \iff g(n) \in o(f(n))$
$\omega(g(n))$ este o mulţime ce se defineşte astfel:
$\omega(g(n)) = \{f(n) |\ \forall c > 0,\ \exists n_0 > 0 \text{ a.î. } 0 \leq c \cdot g(n) < f(n),\ \forall n \geq n_0\}$

Observaţia 1.12

1. $f(n) = \Theta(g(n)), g(n) = \Theta(h(n)) \Longrightarrow f(n) = \Theta(h(n))$ *(tranzitivitate);*
2. $f(n) = \Theta(f(n))$ *(reflexivitate);*
3. $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$ *(simetrie).*

Definiţia 1.10 *O funcţie f are o creştere exponenţială dacă $\exists c > 1$ a.î. $f(x) = \Omega(c^x)$ şi $\exists d$ a.î. $f(x) = O(d^x)$. O funcţie f este polinomială de gradul d dacă $f(n) = \Theta(n^d)$ şi $f(n) = O(n^{d'}),\ \forall d' \geq d$.*

Teorema 1.13

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c & \Rightarrow g(n) \in \Theta(f(n)), c > 0 \\ 0 & \Rightarrow g(n) \in o(f(n)) \\ \infty & \Rightarrow f(n) \in o(g(n)). \end{cases} \tag{1.2}$$

Exemplul 1.14

- *Pentru $x \in \mathbb{R}_+^*$ avem $x^n \in o(n!)$.*

  Deoarece

  $$\frac{x^n}{n!} < \frac{x^n}{\underbrace{x^4 \ldots x^4}_{\lceil n/2 \rceil ori}} \leq \frac{x^n}{(x^4)^{n/2}} = \frac{x^n}{x^{2n}} = (\frac{1}{x})^n, \tag{1.3}$$

  rezultă că

  $$\lim_{n\to\infty} \frac{x^n}{n!} = 0. \tag{1.4}$$

- $\log n \in o(n)$.

  $$\lim_{x\to\infty} \frac{\log x}{x} = \lim_{x\to\infty} \frac{(\log x)'}{(x)'} = \lim_{x\to\infty} \frac{\frac{1}{x \ln 2}}{1} = 0.$$

- *Pentru $a, b > 1,\ a, b \in \mathbb{R}^*$ avem $\log_a n \in \Theta(\log_b n)$.*

Calcularea cu exactitate a timpului de execuţie al unui program oarecare se poate dovedi o activitate dificilă. De aceea, în practică se utilizează estimări încercându-se eliminarea constantelor şi simplificarea formulelor ce intervin în cadrul evaluării.

Dacă două părţi ale unui program, P_1 şi P_2, au timpii de execuţie corespunzători $T_1(n)$ şi $T_2(n)$, unde $T_1(n) = O(f(n))$ şi $T_2(n) = O(g(n))$, atunci programul $P_1 \oplus P_2$ va avea timpul

de execuţie $T_1(n) + T_2(n) = O(max(f(n), g(n)))$ (*regula sumei*), unde $\oplus$ reprezintă operaţia de concatenare.

Prin aplicarea acestei reguli, rezultă faptul că, timpul necesar execuţiei unui număr finit de operaţii este, neluând în considerare constantele, caracterizat în principal de către timpul de execuţie al operaţiei cele mai costisitoare.

Cea de-a doua regulă, *regula produsului*, spune că, fiind date două funţii $f(n)$ şi $g(n)$ astfel încât $T_1(n) = O(f(n))$ şi $T_2(n) = O(g(n))$, atunci avem $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$.

Pe baza *regulilor produsului şi sumei* putem face următoarele reduceri:

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$$

Prezentăm în continuare câteva reguli generale pentru evaluarea complexităţii unui algoritm:

- timpul de execuţie al unei instrucţiuni de atribuire, citire sau afişare a unei variabile este $O(1)$.

- timpul de execuţie al unei secvenţe de instrucţiuni este proporţional cu timpul instrucţiunii care durează cel mai mult.

- timpul de execuţie al unei instrucţiuni de decizie (`if`) este timpul executării instrucţiunilor de pe ramura aleasă plus timpul necesar evaluării condiţiei.

- timpul de execuţie pentru o instrucţiune de ciclare este suma, după numărul de paşi pe care îi realizează instrucţiunea de ciclare, dintre timpul necesar executării corpului instrucţiunii plus timpul necesar evaluării condiţiei.

  1: **for** $i \leftarrow 1, n$ **do**
  2: $\quad A(i)$
  3: **end for**

  Dacă instrucţiunii compuse $A(i)$ îi corespunde un timp de execuţie constant t, ce nu depinde de i, atunci timpul corespunzător întregii instrucţiuni de ciclare `for` anterioare este:

  $$\sum_{i=1}^{n} t = t \sum_{i=1}^{n} 1 = t \cdot n = O(n) \tag{1.5}$$

  În cazul general, timpul de execuţie al instrucţiunii compuse $A(i)$ depinde de pasul i, notat cu t_i. Astfel timpul total corespunzător instrucţiunii `for` este $\sum_{i=1}^{n} t_i$.

- Instrucţiunile de ciclare al căror număr de paşi depinde de îndeplinirea unei condiţii (`while`) sau de neîndeplinirea acesteia (`repeat ... until`), sunt mult mai dificil de analizat deoarece nu există o metodă generală de a afla cu exactitate numărul de repetări al corpului instrucţiunii. De exemplu, pentru fragmentul următor

  1: $i \leftarrow l$
  2: **while** $(a_i < x)$ **do**
  3: $\quad \ldots$ ▷ calcule ce pot modifica, direct sau indirect, valoarea lui a_i
  4: $\quad i \leftarrow i + 1$
  5: **end while**

  nu se poate preciza de câte ori se va ajunge la realizarea instrucţiunii de incrementare din interiorul instrucţiunii `while`. În marea majoritate a cazurilor se utilizează elemente de teoria probabilităţilor în vederea obţinerii unei estimări a numărului de repetări al corpului instrucţiunii.

Exemplul 1.15 *De exemplu*

```
1: for i ← 1, n do
2:     instructiune1
3: end for
```

are timpul de execuţie $O(n)$.

```
1: for i ← 1, n do
2:     for j ← 1, m do
3:         instructiune2
4:     end for
5: end for
```

Timpul de execuţie pentru secvenţa de cod ce conţine două instrucţiuni de ciclare imbricate este $O(n \cdot m)$.

Exemplul 1.16 *Să considerăm un alt exemplu: se dă un şir* A *cu* n *numere reale, şi se doreşte calcularea elementelor unui şir* B *astfel încât să avem* $b_i = \frac{1}{i} \cdot \sum_{j=1}^{i} a_j$, *pentru* $i = \overline{1, n}$.

```
1: for i ← 1, n do
2:     s ← 0
3:     for j ← 1, i do
4:         s ← s + a_j
5:     end for
6:     b_i ← s/i
7: end for
8: return
```

Dacă notăm cu o constantă c_x, *timpul necesar pentru efectuarea unei operaţii atomice, vom obţine: costul efectuării liniei 1 este* $c_1 \cdot n$, *al liniei 2 este* $c_2 \cdot n$, *al liniei 3 este* $c_3 \cdot i$, *al liniei 4 este* $c_4 \cdot i$, *al liniei 6 este* $c_5 \cdot n$ *iar al liniei* **??** *este* c_6.

$$\begin{aligned} T(n) &= c_1 n + c_2 n + \sum_{i=1}^{n} c_3 i + \sum_{i=1}^{n} c_4 i + c_5 n + c_6 = c_1 n + c_2 n + c_3 \sum_{i=1}^{n} i + c_4 \sum_{i=1}^{n} i + c_5 n + c_6 \\ &= \frac{n(n+1)}{2}(c_3 + c_4) + n(c_1 + c_2 + c_5) + c_6 \\ &= \frac{1}{2}(c_3 + c_4)n^2 + \frac{1}{2}(c_3 + c_4)n + n(c_1 + c_2 + c_5) + c_6 \\ &= \frac{1}{2}(c_3 + c_4)n^2 + [\frac{1}{2}(c_3 + c_4) + c_1 + c_2 + c_5]n + c_6 = n^2 \cdot p + n \cdot q + c_6. \end{aligned}$$

De obicei nu se efectuează o analiză atât de detaliată a algoritmului, dar se încearcă o evaluare a blocurilor principale, cum ar fi instrucţiunile de ciclare, atribuindu-le direct valori corespunzătoare complexităţii timp, atunci când este posibil:

$$T(n) = O(n^2) + O(n) + O(1) = O(n^2). \tag{1.6}$$

Vom modifica algoritmul anterior astfel încât să reducem numărul de calcule efectuate:

```
1: s ← 0
2: for i ← 1, n do
3:     s ← s + a_i
4:     b_i ← s/i
5: end for
```

6: ***return***

Pentru această variantă de lucru, complexitatea timp este următoarea:

$$T(n) = O(1)(\textit{linia 1}) + O(n)(\textit{liniile 2–4}) + O(1)(\textit{linia 6}) = O(n) \tag{1.7}$$

În urma analizei de complexitate, putem concluziona faptul că cea de-a doua variantă a algoritmului rulează într-un timp liniar.

1.2.1 Metoda substituţiei

Metoda substituţiei presupune mai întâi ghicirea (estimarea) formei soluţiei pentru relaţia de recurenţă şi apoi, demonstrarea prin inducţie matematică, a corectitudinii soluţiei alese.

Metoda poate fi aplicată în cazul unor ecuaţii pentru care forma soluţiei poate fi estimată. Să considerăm următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} 1 & , \text{ dacă } n = 1 \\ 2T(\frac{n}{2}) + n & , \text{ în rest.} \end{cases} \tag{1.8}$$

Vom presupune că soluţia acestei relaţii de recurenţă este $T(n) = O(n \log n)$ (notăm $\log_2 n = \log n$). Folosind metoda *inducţiei matematice*, vom încerca să demonstrăm inegalitatea:

$$T(n) \leq cn \log n. \tag{1.9}$$

Presupunem mai întâi, că inecuaţia (1.9) are loc pentru $\forall k < n$, inclusiv pentru $\frac{n}{2}$, adică $T(\frac{n}{2}) \leq c \cdot \frac{n}{2} \cdot \log(\frac{n}{2})$.

Vom demonstra că inecuaţia (1.9) este îndeplinită şi pentru n: $T(n) = 2T(\frac{n}{2}) + n$.

$$\begin{aligned} T(n) &\leq 2(c\frac{n}{2}\log\frac{n}{2}) + n = cn\log\frac{n}{2} + n \\ &= cn\log n - cn\log 2 + n = cn\log n - cn + n \\ &\leq cn\log n. \end{aligned} \tag{1.10}$$

Metoda inducţiei matematice presupune să arătăm că soluţia respectă şi cazurile particulare ($T(1) = 1$). Pentru $n = 1$ vom avea $T(1) = c \cdot 1 \cdot \log 1 = 0$ ceea ce contrazice relaţia $T(1) = 1$.

Această situaţie poate fi rezolvată dacă considerăm că $T(n) = cn \log n, \forall n \geq n_0$ (n_0 este o constantă).

Din $T(2) = 4$ şi $T(3) = 5$ vom alege valoarea parametrului c astfel încât să fie îndeplinite inegalităţile $T(2) \leq 2c \log 2$ şi $T(3) \leq 3c \log 3$. Se observă că orice valoare a lui $c \geq 2$ satisface inegalităţile anterioare.

1.2.2 Schimbarea de variabilă

Această metodă presupune realizarea unei schimbări de variabilă în vederea simplificării formulei sau pentru regăsirea unei formule deja cunoscută.

Să considerăm ecuaţia $T(n) = 2T(\sqrt{n}) + \log n$. Dacă înlocuim pe n cu 2^m obţinem:

$$T(2^m) = 2T(2^{\frac{m}{2}}) + \log 2^m = 2T(2^{\frac{m}{2}}) + m. \tag{1.11}$$

Notăm cu $S(m) = T(2^m)$ şi înlocuind în (1.11), obţinem o nouă relaţie de recurenţă:

$$S(m) = 2S(\frac{m}{2}) + m. \tag{1.12}$$

Se observă că această relaţie are o formă similară cu cea din formula (1.8).

Soluţia relaţiei (1.12) este:

$$S(m) = O(m \log m) \Rightarrow T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n). \quad (1.13)$$

În final se obţine faptul că $T(n) = O(\log n \log \log n)$.

1.2.3 Metoda iterativă

Metoda iterativă este mult mai eficientă deoarece nu presupune '*ghicirea*' soluţiei, variantă ce conduce deseori la rezultate greşite şi timp pierdut. De exemplu, să considerăm următoarea formulă de recurenţă:

$$T(n) = \begin{cases} 1 & , \text{ dacă } n = 1 \\ T(\frac{n}{2}) + 1 & , \text{ în rest.} \end{cases} \quad (1.14)$$

Pentru rezolvarea acestei recurenţe vom substitui succesiv pe n cu $\frac{n}{2}$:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= T(\frac{n}{4}) + 2 \\ &= T(\frac{n}{8}) + 3 \\ &\dots \\ &= T(\frac{n}{2^k}) + k. \end{aligned} \quad (1.15)$$

Prin urmare atunci când $1 = \frac{n}{2^k} \Rightarrow k = \log n$, vom avea

$T(n) = 1 + k = 1 + \log n \Rightarrow T(n) = O(\log n)$.

Să analizăm o altă formulă de recurenţă:

$$T(n) = \begin{cases} 1 & , \text{ dacă } n = 1 \\ 2T(\frac{n}{2}) + n & , \text{ în rest.} \end{cases} \quad (1.16)$$

Înlocuind succesiv pe n cu $\frac{n}{2}$ vom obţine următoarea serie de identităţi:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2(2T(\frac{n}{2}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + 2n \\ &= 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n = 8T(\frac{n}{8}) + 3n \\ &\dots \\ &= 2^k T(\frac{n}{2^k}) + kn. \end{aligned} \quad (1.17)$$

Deoarece substituţia se opreşte atunci când $\frac{n}{2^k} = 1$, adică pentru $k = \log n$, vom avea:

$$T(n) = 2^k + kn = n + n \log n = O(n \log n). \quad (1.18)$$

1.2.4 Teorema master

Metoda master pune la dispoziţie o variantă de rezolvare a unor recurenţe de forma

$$T(n) = aT(\frac{n}{b}) + f(n), \tag{1.19}$$

unde $a \geq 1, b > 1$ sunt constante, iar $f(n)$ este o funcţie asimptotic pozitivă. Formula de recurenţă (1.19) descrie timpul de execuţie al unui algoritm ce presupune descompunerea unei probleme de dimensiune n în a subprobleme, fiecare având dimensiunea datelor de intrare $\frac{n}{b}$. $f(n)$ reprezintă costul asociat împărţirii datelor de intrare cât şi costul combinării rezultatelor celor a subprobleme.

Teorema 1.17 (Teorema master) *[24] Fie $a \geq 1$ şi $b > 1$ două constante, $f(n)$ o funcţie asimptotic pozitivă, şi $T(n)$ definită de relaţia de recurenţă*

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

unde $\frac{n}{b}$ va fi $\lfloor \frac{n}{b} \rfloor$ sau $\lceil \frac{n}{b} \rceil$.

Atunci $T(n)$ este mărginită asimptotic după cum urmează:

1. *dacă $f(n) = O(n^{\log_b a - \epsilon})$ pentru o constantă $\epsilon > 0$, atunci $T(n) = \Theta(n^{\log_b a})$;*
2. *dacă $f(n) = \Theta(n^{\log_b a} \log^k n)$, atunci $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ (de obicei $k = 0$);*
3. *dacă $f(n) = \Omega(n^{\log_b a + \epsilon})$ pentru o constantă $\epsilon > 0$, şi dacă $af(\frac{n}{b}) \leq cf(n)$ pentru o constantă $c < 1$ şi oricare număr n suficient de mare, atunci $T(n) = \Theta(f(n))$.*

Corolarul 1.18 *Pentru o funcţie f de tip polinomial unde $f(n) = cn^k$ avem:*

1. *dacă $a > b^k$ atunci $T(n) = O(n^{\log_b a})$;*
2. *dacă $a = b^k$ atunci $T(n) = O(n^k \log n)$;*
3. *dacă $a < b^k$ atunci $T(n) = O(n^k)$.*

Cu alte cuvinte, dacă

$$T(n) \leq aT(\frac{n}{b}) + O(n^k),$$

atunci avem

$$T(n) = \begin{cases} O(n^k \log n) & , \text{ dacă } a = b^k \\ O(n^k) & , \text{ dacă } a < b^k \\ O(n^{\log_b a}) & , \text{ dacă } a > b^k. \end{cases}$$

Exemplul 1.19

- *Fie $T(n) = 2T(\frac{n}{2}) + n$. Avem $a = 2, b = 2, k = 1, f(n) = n^k$. Aplicând corolarul pentru cazul $a = b^k$, se obţine soluţia $T(n) = O(n^k \log n) = O(n \log n)$.*
- *Pentru $T(n) = 9T(\frac{n}{3}) + n$, avem $a = 9, b = 3, k = 1$. Aplicând corolarul pentru cazul $a > b^k$, se obţine soluţia $T(n) = O(n^{\log_3 9}) = O(n^2)$.*
- *Fie $T(n) = 4T(\frac{n}{2}) + n^3$. Avem $a = 4, b = 2, k = 3, f(n) = n^3$. Aplicând corolarul pentru cazul $a < b^k$, se obţine soluţia $T(n) = O(n^3)$.*
- *Fie $T(n) = 2^n T(\frac{n}{2}) + n^n$. Nu se poate aplica Teorema master deoarece $a = 2^n$ nu este constant.*

- *Fie* $T(n) = 2T(\frac{n}{2}) + n \log n$. *Atunci avem* $a = 2, b = 2, k = 1, f(n) = \Theta(n^{\log_2 2} \log^k n)$, *şi aplicând Teorema master, cazul al doilea, obţinem:* $T(n) = \Theta(n \log^2 n)$.
- *Fie* $T(n) = \frac{1}{2}T(\frac{n}{2}) + \frac{1}{n}$. *Nu se poate aplica Teorema master deoarece* $a = \frac{1}{2} < 1$.
- *Fie* $T(n) = \sqrt{2}T(\frac{n}{2}) + \log n$. *Aplicând Teorema master pentru* $a = \sqrt{2}, b = 2, f(n) = O(n^{\frac{1}{2}-\epsilon})$ *obţinem:* $T(n) = \Theta(\sqrt{n})$.
- *Fie* $T(n) = 3T(\frac{n}{4}) + n \log n$. *Aplicând Teorema master pentru* $a = 3, b = 4, f(n) = \Omega(n^{\log_4 3+\epsilon})$ *obţinem:* $T(n) = \Theta(n \log n)$.
- *Fie* $T(n) = 16T(\frac{n}{4}) - n \log n$. *Nu se poate aplica Teorema master deoarece* $f(n) = -n \log n$ *nu este o funcţie asimptotic pozitivă.*

Exemplul 1.20 ***Analiza acţiunilor***

`Deschiderea` *unei acţiuni la o anumită dată calendaristică reprezintă numărul maxim de zile consecutive (până la acea dată) în care preţul acţiunii a fost mai mic sau egal cu preţul din ziua respectivă. Fie* p_k *preţul unei acţiuni în ziua* k *iar* d_k *deschiderea calculată în aceeaşi zi.*

În continuare se prezintă un exemplu de calcul al deschiderii unei acţiuni pentru un număr de 7 zile:

p_0	p_1	p_2	p_3	p_4	p_5	p_6
9	*6*	*3*	*4*	*2*	*5*	*7*
d_0	d_1	d_2	d_3	d_4	d_5	d_6
1	*1*	*1*	*2*	*1*	*4*	*6*

Algoritm 7 Algoritm de calcul al deschiderii unei acţiuni (prima variantă)

```
1: procedure ComputeSpan1(p, n; d)
2:     for k ← 0, n − 1 do
3:         j ← 1
4:         while ((j < k) ∧ (p_{k−j} ≤ p_k)) do
5:             j ← j + 1
6:         end while
7:         d_k ← j
8:     end for
9:     return
10: end procedure
```

Algoritmul 7 calculează deschiderea unei acţiuni pentru un interval mai lung de timp pe baza evoluţiei preţurilor acesteia. Timpul de execuţie al acestui algoritm este $O(n^2)$.

Vom prezenta un alt mod de calcul al deschiderii unei acţiuni folosind o stivă (a se vedea algoritmul 8).

Stiva este o structură de date de tip container deoarece ea depozitează elemente de un anumit tip. Pentru a putea să operăm asupra unei colecţii de elemente păstrate într-o stivă, se definesc următoarele operaţii, în afară de operaţiile fundamentale `push` *şi* `pop`*:*

- `push(x:object)` – *adaugă obiectul* x *în vârful stivei.*
- `pop():object` – *elimină şi întoarce obiectul din vârful stivei; dacă stiva este goală avem o situaţie ce generează o excepţie.*

- `peek():object` – *întoarce valoarea obiectului din vârful stivei fără a-l extrage.*
- `size():integer` – *întoarce numărul de obiecte din stivă.*
- `isEmpty():boolean` – *întoarce* `true` *dacă stiva nu conţine nici un obiect.*
- `isFull(): boolean` – *întoarce* `true` *dacă stiva este plină.*
- `init(capacitate:integer)` – *iniţializează stiva.*

Algoritm 8 Algoritm de calcul al deschiderii unei acţiuni (a doua variantă)

```
1: procedure ComputeSpan2(p, n; d)
2:     call init(S)
3:     for k ← 0, n − 1 do
4:         ind ← 1
5:         while (isEmpty(S) = false) ∧ (ind = 1) do
6:             if (p_k ≥ p_peek(S)) then
7:                 call pop(S)
8:             else
9:                 ind ← 0
10:            end if
11:        end while
12:        if (ind = 1) then
13:            h ← −1
14:        else
15:            h ← peek(S)
16:        end if
17:        d_k ← k − h
18:        call push(S, k)
19:    end for
20:    return
21: end procedure
```

Timpul de execuţie al noului algoritm este $O(n)$.

1.3 Exerciţii

1. (a) Să se determine dacă un număr natural este simetric sau nu.
   (b) Să se determine toate cifrele distincte dintr-un număr natural.
   (c) Să se determine reprezentarea în baza 2 a unui număr natural.
   (d) Să se determine forma corespunzătoare în baza 10 a unui număr reprezentat în baza 2.
   (e) Să se elimine dintr–un număr natural toate cifrele de forma $3k + 1$ şi să se afişeze numărul rezultat.
   (f) Pentru un număr natural dat să se construiască din cifrele acestuia cel mai mare număr prin amplasarea mai întâi a cifrelor impare şi apoi a cifrelor pare.
2. (a) Să se determine toţi divizorii unui număr natural.