Mirel COŞULSCHI

ALGORITMI FUNDAMENTALI. Proiectare şi implementare

Mirel COŞULSCHI

ALGORITMI FUNDAMENTALI

Proiectare şi implementare

Editura UNIVERSITARIA
Craiova, 2015

Editura PRO UNIVERSITARIA
Bucureşti, 2015

Referenți științifici:
Prof.univ.dr. Costin Bădică
Lect.univ.dr. Petre Băzăvan

Descrierea CIP a Bibliotecii Naţionale a României
COŞULSCHI, MIREL
Algoritmi fundamentali : proiectare şi implementare / Mirel Coşulschi. - Craiova : Universitaria ; Bucureşti : Pro Universitaria, 2015
Bibliogr.
ISBN 978-606-14-0881-8
ISBN 978-606-26-0154-6

004.421(075.8)
004.65(075.8)

Design coperta: Adrian Gabroveanu

Prefaţă

Alături de înţelegerea şi deprinderea unui limbaj de programare, un bun specialist în informatică trebuie să stăpânească limbajul algoritmic, să poată estima complexitatea unui algoritm şi să aibă o cultură de specialitate solidă în ceea ce priveşte algoritmii fundamentali. Acest demers publicistic subscrie ideii expusă mai sus.

Lucrarea de faţă abordează tematici privitoare la complexitatea algoritmilor, recursivitate, combinatorică, tehnici de sortare şi căutare, metode generale de elaborare a algoritmilor (metoda Greedy, metoda Divide-et-Impera, metoda Backtracking şi metoda Programării dinamice). Materialul a fost prezentat de autor studenţilor specializării Informatică de la Facultatea de Matematică şi Ştiinţe ale Naturii (în trecut Facultatea de Matematică şi Informatică) a Universităţii din Craiova, în cadrul laboratoarelor şi cursurilor de specialitate. Materialul poate fi consultat în egală măsură de către elevii de liceu la profilul informatică, de studenţii altor facultăţi ce au în programă studiul limbajului algoritmic, sau de specialiştii care lucrează în industria IT ca programatori. Din carte nu lipsesc topici avansate care se adresează studenţilor înscrişi la studiile de master sau doctorat.

Pe lângă conceptele teoretice, cititorul interesat de subiect va găsi peste 140 de algoritmi prezentaţi în cadrul materialului ce este structurat 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. Bibliografia bogată conţine referinţe către lucrări importante din domeniu.

Autorul este recunoscător referenţilor ştiinţifici ai acestei lucrări pentru sugestiile şi observaţiile pertinente pe marginea manuscrisului.

Craiova, ianuarie, 2015 *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, ..., x_N }
2: min ← x_1
3: for i ← 2, N do
4:     if (min > x_i) then
5:         min ← 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`*.*

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