COMPLEXITATEA ALGORITMILOR
- STUDIU -

Autor: Secită Neli

Un ***algoritm*** este un ansamblu complet şi finit de operaţii cunoscute, care sunt executate într-o succesiune logică bine stabilită şi au scopul de a transforma un ansamblu finit de valori de intrare într-un ansamblu finit de ieşire într-un timp finit.

Un ***algoritm*** este ***eficient*** dacă foloseşte puţine resurse ale calculatorului şi anume:

- ***Memoria internă.*** În memoria internă se alocă spaţiu atât pentru datele folosite de algoritm, cât şi pentru codul executabil al programului (instrucţiunile în cod maşină).
- ***Procesorul.*** Timpul de utilizare a procesorului depinde de timpul necesar pentru executarea algoritmului.

În ciuda progreselor tehnologice spectaculoase, trebuie elaboraţi algoritmi eficienţi (care au codul maşină scurt şi sunt rapizi) pentru rezolvarea problemelor moderne din ce in ce mai complexe. În general, se urmăreşte ca în urma analizei să rezulte o dependentă între timpul de execuţie şi dimensiunea datelor de prelucrat.

Criteriile faţă de care se stabileşte eficienţa algoritmilor sunt:

- Complexitatea spaţiului
- Complexitatea timp

COMPLEXITATEA SPAŢIU

Complexitatea spaţiu este dimensiunea spaţiului de memorie utilizat de program, spaţiu care este de două tipuri:

- ***constant*** - folosit pentru memorarea codului programului executabil, a constantelor, variabilelor simple şi a structurilor de date de dimensiune constantă alocate static. Dimensiunea spaţiului este independentă de volumul şi tipul datelor de intrare;
- ***variabil*** - folosit pentru structurile de date alocate dinamic, precum şi pentru apelurile de proceduri şi funcţii. Dimensiunea acestui spaţiu depinde şi de tipul şi volumul datelor de intrare, precum şi de instanţa problemei de rezolvat (starea la un moment dat al execuţiei: dacă folosesc sau nu structuri dinamice de date, dacă apelează sau nu subprograme).

Pentru a folosi cât mai puţină memorie trebuie ca în gândirea algoritmului să avem în vedere următoarele:

- Alegerea corectă a tipului de dată pentru fiecare variabilă de memorie folosită în algoritm.
- Rezolvarea problemei folosind cât mai puţine variabile de memorie.

Atunci când i se atribuie unei date un tip de dată, data capătă mai multe atribute care determină domeniul de definiţie intern al ei (mulţimea în care poate lua valori data). Tipul de dată ales influenţează calitatea programului, deoarece el determină dimensiunea zonei de memorie alocată, algoritmul de codificare şi operatorii admişi pentru prelucrare. Din această cauză, la alegerea tipului de dată trebuie să se facă în două moduri analiza datei:

- ***Logic (la nivelul conceptual).*** Analiza se face pornind de la enunţul problemei şi constă în identificarea domeniului de definiţie extern al datei. De exemplu, în enunţul problemei se precizează că trebuie prelucrat un număr întreg pozitiv, cu valori cuprinse între 0 şi 200. Acesta este domeniul de definiţie extern al datei.
- ***Fizic (la nivelul reprezentării ei în memoria internă)***. Analiza se face pornind de la tipurile de date implementate în limbajul de programare, fiecare tip de dată având un domeniu de definiţie intern al datei. Să presupunem că în limbajul de programare sunt implementate următoarele trei tipuri de date întregi: ***tipul 1*** cu domeniul de definiţie [-128, 127] reprezentat pe un octet, ***tipul 2*** cu domeniul de definiţie [0, 255] reprezentat pe un octet şi ***tipul 3*** cu domeniul de definiţie [-32768, 32767] reprezentat pe doi octeţi. În urma analizei fizice trebuie ales tipul de dată adecvat. ***Regula*** este: ***se alege tipul de dată care consumă cea mai puţină memorie, astfel încât domeniul de definiţie extern al datei să fie inclus în domeniul de definiţie intern al datei.*** Pentru exemplul prezentat se va alege *tipul 2* de dată deoarece numai *tipul 2* şi *tipul 3* respectă condiţia de incluziune a domeniilor de definiţie ($[0,200] \subset [0,255]$ şi $[0,200] \subset [-32768, 32767]$, dar $[0,200] \not\subset [-128,127]$), iar *tipul 2* ocupă mai puţin spaţiu de memorie (1 octet) decât *tipul 3* (2 octeţi).

Problema pare neimportantă atunci când algoritmul foloseşte câteva variabile de memorie. Dar pentru rezolvarea problemelor complexe se pot folosi structuri de date în care se memorează foarte multe date elementare (de la câteva zeci până la milioane de date elementare). În acest caz este important dacă tipul de dată elementară este ales corect. Fiecare octet de care nu are nevoie o dată elementară poate să însemne un consum inutil de milioane de octeţi pentru structura de date.

Progresele tehnologice realizate (crearea unor memorii cu capacităţi de stocare din ce în ce mai mari) duc la scăderea importanţei complexităţii spaţiu.

COMPLEXITATEA TIMP

În general, timpul de execuţie al unui program este durata (exprimată în secunde, milisecunde, nanosecunde etc.) necesară ca programul să preia datele de intrare, să efectueze anumite operaţii asupra lor şi să furnizeze datele de ieşire. Acest timp depinde întotdeauna de calculatorul pe care se execută programul. Un program care sortează un milion de numere va avea un anumit timp de execuţie pe un calculator dotat cu un procesor Pentium II, care diferă de timpul de execuţie pe un calculator dotat cu un procesor Pentium 4.

Pentru calculatoarele de ultimă generaţie vom avea timpi de execuţie diferiţi care vor depinde, în primul rând, de arhitectura şi frecvenţa procesorului. Considerând din nou exemplul programului care sortează un milion de numere, vom obţine timpi de execuţie diferiţi pe calculatoare din familii diferite. Programul nu va rula în acelaşi timp pe calculatoare cu procesoare Celeron, Pentium 4, Itanium, Athlon XP, Athlon 64 sau Opteron.

Dacă avem procesoare din aceeaşi familie, vom observa diferenţe vizibile între un procesor cu frecvenţa de 2 GHz şi un procesor cu frecvenţa de 3 GHz.

Dacă avem aceeaşi frecvenţă, vom observa, din nou, diferenţe între un Pentrium 4 şi un Athlon XP.

De fapt, chiar dacă avem acelaşi tip de procesor şi aceeaşi frecvenţă, pot apărea unele diferenţe de timp datorate altor factori cum ar fi: sistemul de operare, cantitatea de memorie, rata de transfer a discului etc.

Aşadar, măsurarea exactă a timpului de execuţie nu este un criteriu valid pentru a descrie timpul de execuţie al unui algoritm.

S-ar putea crede că un algoritm care are nevoie de *t* secunde pentru a rula pe un calculator al cărui procesor are frecvenţa de 1 GHz va avea nevoie de *t/2* secunde pentru a rula pe un calculator al cărui procesor are frecvenţa de 2 GHz. Din nefericire, o astfel de presupunere nu este validă, deoarece dublarea frecvenţei nu înseamnă dublarea performanţelor.

Nu vom determina timpul de execuţie al programului pe calculator, ci vom realiza ***analiza complexităţii a algoritmului*** din următoarele motive:

- timpul de execuţie al programului depinde de numărul exact de operaţii elementare efectuate de algoritmul ce stă la baza conceperii programului şi rezolvării problemei (algoritmul are comportări diferite pentru seturi de date de intrare diferite);
- timpul de execuţie depinde de tipul calculatorului pe care se execută programul (programul este "rapid" pe calculatoare puternice, dar "lent" pe calculatoare mai slabe);
- timpul de execuţie al programului depinde şi de limbajul de programare în care este implementat (acelaşi program scris în limbajul C++ s-ar putea să fie mai rapid decât implementarea sa în Pascal);
- din considerente practice, nu putem testa pe calculator programe pentru cazuri oricât de mari, care ar dura ore, zile şi chiar ani;
- se pot obţine informaţii empirice asupra comportării în timp (duratei) a programelor numai pentru seturi particulare de date de intrare, nu pentru situaţii generale.

Pentru a înţelege analiza complexităţii unui algoritm vom defini următoarele noţiuni.

Se numeşte **operaţie elementară** acea operaţie sau grup de operaţii care are durata (timpul) de execuţie independentă de datele de intrare ale problemei.

Exemple de operaţii elementare: o adunare, o scădere, o înmulţire, o împărţire, o comparaţie, o atribuire, 10 adunări, 10 scăderi, etc., pot constitui o operaţie elementară, dar un grup de n adunări nu poate reprezenta o operaţie elementară, deoarece durata lui de execuţie depinde de volumul datelor de intrare (n).

Costul unei operaţii elementare este dat de timpul necesar execuţiei acestei operaţii elementare. Timpul unei operaţii elementare este fix. Fiecare operaţie elementară are alt timp de execuţie.

Operaţia de bază este o operaţie elementară sau o succesiune de operaţii elementare a căror execuţie nu depind de valorile datelor de intrare.

Eficienţa unui algoritm reprezintă timpul de calcul estimat prin numărul de execuţii ale operaţiei elementare.

Eficienţa se notează cu $T(n)$ şi este o funcţie care depinde de dimensiunea datelor de intrare.

ETAPELE ANALIZEI COMPLEXITĂŢII

Se stabileşte dimensiunea problemei

Se stabileşte operaţia de bază

Se verifică dacă numărul de execuţii ale operaţiei de bază depinde doar de dimensiunea problemei

DA

NU

Se determină numărul de execuţii ale operaţiei de bază

Se analizează:

- **Cazul cel mai favorabil**
- **Cazul cel mai defavorabil**
- **Cazul mediu (dacă este posibil)**

Se stabileşte clasa de complexitate a algoritmului

Exemplu:

Dacă într-un algoritm se execută $2 \cdot n^2 + 5 \cdot n + 3$ operaţii elementare, eficienţa sa va fi $T(n) = 2 \cdot n^2 + 5 \cdot n + 3$.

Ordinul de complexitate al unui algoritm îl reprezintă timpul de calcul estimat prin ordinul de mărime al numărului de execuţii ale operaţiei de bază.

Ordinul de complexitate se notează cu $O(f(n))$, unde $f(n)$ reprezintă termenul determinant al numărului de execuţii ale operaţiei de bază – termenul determinant din funcţia $T(n)$.

Notaţii:
n - ordinul de mărime al datelor de intrare în algoritm
T(n) – complexitatea timp al algoritmului
O – ordinul de mărire al complexităţii timp
f(n) – o funcţie dependentă de numărul de operaţii timp

Exemplu

Pentru determinarea ordinului de complexitate se porneşte de la funcţia $T(n)$ - în exemplul nostru $T(n) = 2 \cdot n^2 + 5 \cdot n + 3$ - şi se execută următoarele operaţii:

1. Coeficienţii termenilor din expresia $T(n)$ se reduc la valoarea 1. În exemplu, $f(n) = n^2 + n + 1$
2. Se păstrează din expresie termenul determinat. În exemplu, $f(n) = n^2$, deoarece pentru valori foarte mari ale lui n valoarea n+1 este neglijabilă faţă de cea a lui n^2.

Aşadar dacă eficienţa unui algoritm este dată de funcţia $T(n) = 2 \cdot n^2 + 5 \cdot n + 3$, spunem că acest algoritm are ordinul de complexitate $O(n^2)$.

Astfel în funcţie de ordinul de complexitate, există următoarele tipuri de algoritmi:

Ordin de complexitate	Tipul algoritmului
$\boldsymbol{O(n)}$	**Algoritm liniar**
$\boldsymbol{O(n^m)}$	**Algoritm polinomial**. Dacă m=2, algoritmul este pătratic, iar dacă m=3, algoritmul este cubic.
$\boldsymbol{O(k^n)}$	**Algoritm exponenţial**. De exemplu: 2^n , 3^n etc. Algoritmul de tip $O(n!)$ este tot de tip exponenţial deoarece: $1 \times 2 \times 3 \times \ldots \times n > 2 \times 2 \times 2 \times \ldots \times 2 = 2^{n-1}$

$O(logn)$	**Algoritm logaritmic**
$O(nlog\,n)$	**Algoritm liniar logaritmic**

$$O(1) \leq O(logn) \leq O(n) \leq O(n\,logn) \leq O(n^2) \leq O(n^3) \leq O(n^k) \leq O(k^n) \leq O(n!)$$

Tipul algoritmului este foarte important deoarece de el depinde timpul de execuţie şi implicit eficienţa algoritmului. Pentru exemplificare să presupunem că pentru executarea unei operaţii de bază sunt necesare 10^{-9} secunde (adică într-o secundă se execută un miliard de operaţii de bază), iar dimensiunea datelor de intrare este *n*. În următorul tabel putem compara timpii de execuţie necesari pentru diferite tipuri de algoritmi şi pentru diferite dimensiuni ale datelor de intrare (valori ale lui *n*).

$O($	**n=10**	**n=20**	**n=30**	**n=40**	**n=50**
$log_2 n$	3,32	4,32			
n	10^{-8}s	$2 \times 10^{-8} s$	$3 \times 10^{-8} s$	$4 \times 10^{-8} s$	$5 \times 10^{-8} s$
n^2	10^{-7}s	$4 \times 10^{-7} s$	$9 \times 10^{-7} s$	$16 \times 10^{-7} s$	$26 \times 10^{-7} s$
n^3	10^{-6}s	$8 \times 10^{-6} s$	$27 \times 10^{-6} s$	$64 \times 10^{-6} s$	$125 \times 10^{-6} s$
2^n	$2^{10} \times 10^{-9} s \approx$ s	$2^{20} \times 10^{-9}$	$2^{30} \times 10^{-9} s \approx 1s$	$2^{40} \times 10^{-9} s \approx 10^3 \approx$	$2^{50} \times 10^{-9} s \approx 10^6 \approx$
3^n	$3^{10} \times 10^{-9} s \approx$	$3^{20} \times 10^{-9}$	$3^{30} \times 10^{-9} s \approx 2 \times$	$3^{40} \times 10^{-9} s \approx 1{,}2 \times$	$3^{50} \times 10^{-9} s \approx 7 \times 10$

Concluzii

1. Cei mai rapizi algoritmi sunt cei logaritmici
2. Dacă pentru rezolvarea aceleiaşi probleme există algoritmi polinomiali şi exponenţiali, se preferă cei polinomiali.
3. Dacă pentru rezolvarea aceleiaşi probleme există mai mulţi algoritmi polinomiali, se preferă cel cu gradul mai mic.

Timpul de execuţie al unui algoritm nu depinde doar de dimensiunea datelor de intrare, ci şi de configurarea acestora. De exemplu, pentru a sorta un şir de numere "aproape sortat" vom avea nevoie, în principiu, de mai puţin timp decât pentru un şir „sortat în ordine inversă".

Noţiunea de timp de execuţie T(n) poate avea mai multe semnificaţii:

✶ **Cazul cel mai defavorabil** corespunde **timpului maxim de execuţie**. În cazul unui algoritm de sortare ascendent, acest timp este dat (în general) de sortarea unui vector având elementele în ordine descrescătoare.

✶ **Cazul mediu** corespunde **timpului mediu de execuţie**, adică raportul dintre suma timpului necesar pentru toate seturile de date posibile şi numărul de seturi. Timpul mediu în cazul unui algoritm de sortare se bazează pe un vector de elemente generat aleatoriu.

✶ **Cazul cel mai favorabil** corespunde **timpului minim de execuţie**. Pentru un algoritm de sortare, acest caz este dat (in general) de un vector deja sortat.

De obicei, în analiza algoritmilor se utilizează cazul cel mai defavorabil. Există trei motive pentru această alegere.

În primul rând, timpul de execuţie în cel mai defavorabil caz reprezintă o limită superioară pentru timpul de execuţie corespunzător oricărei configuraţii de dimensiune dată. Este garantat faptul, că pentru nici o configuraţie, nu vom avea un timp de execuţie mai mare.

În al doilea rând, în cazul unor algoritmi cazul cel mai defavorabil apare relativ frecvent. De exemplu, pentru căutarea unei valori într-un vector, cazul cel mai defavorabil apare atunci când valoarea nu există, iar o astfel de situaţie este destul de frecventă.

În al treilea rând, în marea majoritate a cazurilor, cazul mediu este "aproape la fel de nefavorabil" ca şi cel mai nefavorabil caz. De exemplu, pentru a determina maximul unui şir, indiferent ce metodă aplicăm, va trebui întotdeauna să parcurgem toate elementele şirului.

Totuşi, în anumite situaţii, timpul de execuţie pentru cazul mediu se poate dovedi util. Cea mai mare dificultate întâmpinată în momentul în care se încearcă determinarea acestui timp este dată de faptul că, în marea majoritate a cazurilor, nu vom putea determina timpii de execuţie pentru toate cazurile pentru ca apoi să calculăm o medie. În principiu, în acest scop putem considera că orice configuraţie a datelor de intrare are aceleaşi şanse să apară ca şi oricare alta. Din nefericire, în practică, această presupunere se dovedeşte deseori falsă.

BIBLIOGRAFIE:

1. Donald E. Knuth – Arta programării calculatoarelor, Volumul III , Editura Teora, 2001
2. Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest – Introducere în algoritmi, Editura Agora Cluj, 2008.