Clase de algoritmi. Recapitulare

CAPITOLUL X

Cuprins

Clasificarea algoritmilor

În capitolele anterioare am discutat și analizat numeroși algoritmi pentru soluționarea unor tipuri diferite de probleme

În general înainte de a rezolva o problemă nouă, tendința generală este de a căuta probleme similare pentru care cunoaștem deja soluții, de aici apare și nevoie de a tipurile de probleme și de algoritmi

Există mai multe criterii după care putem clasifica algoritmii:

- Metoda de implementare
- Metoda de proiectare
- Clasa de complexitate
- Domeniul de aplicabilitate, etc.

Clasificarea algoritmilor

Clasificarea după metoda de implementare:

- Recursivitate vs. iterație

Un algoritm recursiv este acela care se apelează pe el însuși în mod repetat până când o condiție de bază este satisfăcută. Este folosit adesea în limbaje funcționale ca C, C++, Java, etc.

Un algoritm iterativ folosește construcții de tip bucle sau alte structuri de tip cozi, de exemplu pentru a rezolva problema.

- Procedural vs. declarativ

Limbajele de programare declarative sunt orientate pe descrierea cerinței, solicitării (ex. SQL).

In limbajele procedurale, trebuie specificați toți pașii de rezolvare (C, PHP, etc.)

- Serial vs. paralel

Algoritmii paraleli nu au făcut subiectul acestui curs. Până acum am studiat doar algoritmi secvențiali

Clasificarea algoritmilor

Clasificarea după metoda de implementare (continuare):

- Deterministic vs. nedeterministic

Algoritmii determiniști rezolvă problema printr-un proces predefinit

Algoritmii nedeterminiști ”ghicesc” soluția folosindu-se de diferite euristici (nu fac subiectul acestui curs).

- Exact vs. aproximativ

Câteodată este dificil a găsi soluția optimă, pe care o numim exactă în acest caz. În cazul în care ne putem mulțumi cu soluții apropiate, putem folosi algoritmi care determină soluții aproximative.

Clasificarea algoritmilor

Clasificarea după metoda de proiectare:

- Algoritmi de tip Greedy

Algoritmi cu diferite etape

Deciziile se iau fără a analiza situația în perspectivă, se selectează optime locale (minime/maxime locale)

- Divide et Impera (Divide and Conquer)

Algoritmii împart problema în subprobleme mai simple de același tip, aplică recursivitatea pe fiecare subproblemă și combină rezultatele

- Programare dinamica

Diferența între programarea dinamică și Divide et Impera constă prin faptul că în acest caz subproblemele depind una de alta și se suprapun

- Programare liniară

Se axează pe maximizarea/minimizarea unor funcții liniare a valorilor de intrare

Clasificarea algoritmilor

Clasificarea după clasa de complexitate:

- Clasele de complexitate s-au discutat în capitolul de analiza al algoritmilor
- Ex:
  - Liniară – O(n)
  - Polinomială $O(n^c)$, unde c este o constantă
  - Exponențială $O(2^n)$
  - Logaritmică O(log n)
  - Etc.

Clasificarea domeniul de aplicabilitate/ aria de cercetare:

Ex.: Inteligență artificială, Criptografie, Compresie de date, etc.

Algoritmi de tip Greedy

Algoritmii de tip greedy, se desfășoară în etape. În fiecare etapă se ia o decizie care implică cea mai bună variantă la momentul respectiv

În fiecare moment se va alege optimul local presupunând că soluția va duce și la un optim global.

Ex: Sortarea prin selecție (selection sort), alege la fiecare pas valoarea minimă locală.

Algoritmii de tip Greedy nu vor da întotdeauna soluția optimă per ansamblu. De exemplu problemele studiate care se rezolvau cu metoda Backtracking nu conțin neapărat în soluția finală doar minime/maxime locale.

Divide et impera

După cum am văzut algoritmii de tip Greedy nu dau întotdeauna soluția optimă.

Unele din problemele pentru care algoritmii de tip Greedy nu sunt optimi, se pot rezolva cu tehnica Divide et impera (Divide and conquer).

Să ne amintim!

Divide et impera funcționează prin împărțirea recursivă a problemei în subprobleme mai simple de același tip, până se ajunge la probleme atât de simple în cât se pot rezolva direct. Soluțiile obținute pentru aceste subprobleme se combină pentru a furniza soluția completă pentru problema originala. Ex. Căutare binară, Quicksort, Mergesort

Cei trei pași sunt:

1. Divide problema în subprobleme mai simple de același tip
2. Rezolvă în mod recursiv fiecare din subprobleme
3. Combină rezultatele obținute în mod corespunzător

Divide et impera

Avantajele metodei Divide et impera:

- Rezolvarea relativ facilă a unor probleme complicate (ex. Turnul din Hanoi)
- Paralelismul – pentru că fiecare subproblemă se rezolvă independent, algoritmii pot fi implementați pe sisteme multiprocesor
- Accesul la memorie – de obicei folosesc în mod eficient memoria cache a sistemului

Dezavantajele metodei

- Dezavantajele legate de recursivitate – timp lent, folosirea stivei
- În unele cazuri o abordare directă poate fi mai simplă – ex. adunarea a n numere se face mai simplu folosind o buclă

Programare dinamică

Programarea dinamică nu se referă la o tehnică de codare, ci se referă la o clasă de algoritmi. Termenul programare în acest context nu este legat de codare, ci se referă la umplerea unor tabele.

Tehnica de programare dinamică se aseamănă cu tehnica divide et impera, dar diferă prin faptul că subproblemele nu mai sunt independente, ci se pot suprapune.

Folosind o tabelă în care se memorează rezultatele subproblemelor deja rezolvate, tehnica reduce semnificativ complexitatea.

Programarea dinamică = Recursivitate + Memorare

Recursivitate – rezolvarea problemelor într-o manieră recursivă

Memorare – stocarea valorilor deja calculate într-o tabelă

Programare dinamică

Proprietățile de bază ale unei probleme care poate fi rezolvată prin programare dinamică:

- Substructură optimă – soluția optimă a problemei conține soluțiile optime ale subproblemelor care o alcătuiesc
- Suprapunerea subproblemelor – o soluție recursivă conține un număr mic de subprobleme distincte repetate de un număr mare de ori

Diferența între programarea dinamică și simpla recursivitate directă constă în memorarea apelurilor recursive. Dacă subproblemele sunt distincte, atunci memorarea nu are sens. În acest caz nu ne folosim de programarea dinamică.

Programare dinamică

Abordări:

1. De sus în jos (top-down)

2. De jos în sus (bottom-up)

Top-down

În acest caz problema este spartă în subprobleme. Fiecare subproblemă este rezolvată și soluția memorată într-o tabelă. Înainte de a rezolva o nouă subproblemă se verifică dacă nu există deja valoarea calculată pentru aceasta.

Bottom-up

Se calculează funcția începând de la cazurile cele mai simple, apoi se crește încet numărul de variabile de intrare/ valoarea valorilor de intrare. Pentru cazurile mai complexe se folosesc valorile pentru cazurile simple deja calculate.

Programare dinamică

Exemplu:

Seria Fibonacci, definită pe numere naturale

$$Fib(n)\begin{cases} 0, n = 0 \\ 1, n = 1 \\ Fib(n-1) + Fib(n-2), n: 1 \end{cases}$$

Varianta **recursivă**

```
int RecFib(unsigned int n)
{ if (n==0) return 0;
   if (n==1) return 1;
   return RecFib(n-1)+RecFib(n-2);
}
```

Conform ecuației de recurență, obținem

$T(n) = T(n-1) + T(n-2) + 1 \Rightarrow O(2^n)$

Cum ajută memorarea?

Ex: Fib (5), deși are 2^5 apeluri, avem doar 5 valori distincte calculate de mai multe ori.

fib(5)
fib(4) + *fib*(3)
(*fib*(3) + *fib*(2)) + (*fib*(2) + *fib*(1))
((*fib*(2) + *fib*(1)) + (*fib*(1) + *fib*(0))) + ((*fib*(1) + *fib*(0)) + *fib*(1))
(((*fib*(1) + *fib*(0)) + *fib*(1)) + (*fib*(1) + *fib*(0))) + ((*fib*(1) + *fib*(0)) + *fib*(1))

Programare dinamică

Exemplu:

Seria Fibonacci, definită pe numere naturale

$$Fib(n)\begin{cases} 0, n = 0 \\ 1, n = 1 \\ Fib(n-1) + Fib(n-2), n > 1 \end{cases}$$

Varianta prin **programare dinamică**

```
int fib[MAX]
//abordarea de jos in sus
int PDFib(unsigned int n)
{
 if ((n==0)||(n==1)) return n;
 fib[0] =0;
 fib[1] =1;
 for(int i=2; i<= n; i++)
    fib[i]=fib[i-1]+fib[i-2];
  return fib[n];
} //Complexitate O(n)
```

Programare dinamică

Exemplu:

Seria Fibonacci, definită pe numere naturale

$$Fib(n)\begin{cases} 0, n = 0 \\ 1, n = 1 \\ Fib(n-1) + Fib(n-2), n > 1 \end{cases}$$

Varianta prin **programare dinamică**

```
int fib[MAX]={0,1};

//abordarea de sus in jos

int PDFib2(unsigned int n)

{
 if ((n==0)||(n==1)) return n;
 if (fib[n]!=0) return fib[n]; //daca
avem deja calculat
 else fib[n]=PDFib2(n-1)+PDFib2(n-2);
 return fib[n];
} //Complexitate O(n)
```

Concluzii

Să ne amintim! Unii algoritmi au o rată de creștere mai lentă, alții mai accelerată. Aceasta se reflectă în clasa de complexitate a algoritmului, care determină și categoria de dificultate a problemei

Complexitatea timpului de rulare	Nume clasă complexitate	Exemplu	Categorie de probleme
O(1)	Constantă	Adăugarea unui element într-o stivă	Probleme ușoare
O(log n)	Logaritmică	Căutarea binară	
O(n)	Liniară	Căutarea liniară	
O(n log n)	Liniară logaritmică	Heap sort	
$O(n^2)$	Pătratică	Bubble sort	
$O(n^3)$	Cubică	Înmulțirea matricilor	
$O(2^n)$	Exponențială	Turnurile din Hanoi	Probleme complexe
O(n!)	Factorială	Permutările unui șir de caractere	

Exerciții

Ex1: Fie un set de n intervale S={($start_i$, $stop_i$), 1<= i <= n}. Să se determine subsetul maxim S' a lui S, astfel încât să nu existe două intervale din S' care să se suprapună.

Ex2: Fie un tablou de întregi A[0…n-1] ce conține valori unice în ordine crescătoare. Să se verifice dacă există un index i astfel încât A[i]=i, folosind un algoritm de tip divide et impera care are o complexitate O(log n).

Ex3: Pentru un tablou de întregi, să se scrie un program care găsește o subsecvență A(i)...A(j) pentru care suma elementelor este maximă, folosind programarea dinamică.