

Tehnici de programare

Greedy; Divide and conquer

În general există mai multe metode prin care putem aborda o problemă. Printre aceste metode sunt: **greedy, divide and conquer, backtracking, dinamică, stocastică**, etc. Aceste metode furnizează un cadru, o modalitate de a concepe algoritmi. Din acest punct de vedere le putem considera ca fiind algoritmi de a crea algoritmi sau, cu alte cuvinte, *metaalgoritmi*. În continuare vom discuta despre unele dintre aceste metode, care sunt abordările specifice, la ce gen de probleme se potrivesc, care sunt plusurile și minusurile lor, toate exemplificate cu ajutorul unor exemple.

Greedy (Iacom)

Greedy este o metodă simplă de a concepe algoritmi, care utilizează următoarea abordare: **dacă avem mai multe posibilități, întotdeauna o alegem pe cea care ne apropie cel mai mult de rezultatul dorit.**

Tocmai din cauza acestei abordări, de a alege la fiecare pas ceea ce *pare* a fi cel mai favorabil, metoda se numește *greedy* (Iacom). Deoarece această abordare este simplă, este ușor să o folosim la conceperea unor algoritmi. În general algoritmii Greedy sunt ușor de implementat și au rezultate satisfăcătoare. Există o clasă de probleme pentru care metoda Greedy conduce la rezultate optimale, și anume atunci când *soluția optimă globală se compune întotdeauna din soluțiile optime locale*, de la fiecare pas. În acest gen de probleme se încadrează găsirea *arborelui de acoperire minimă a unui graf*, pentru care avem algoritmii *Prim* și *Kruskal*, de tip Greedy.

La modul general, Greedy nu găsește soluția optimă și nici măcar nu garantează că se va găsi o soluție, chiar dacă aceasta există. De exemplu, să considerăm că ne găsim în Paris și dorim să vizităm turnul Eiffel. Vedem turnul pe deasupra clădirilor, dar, tot din cauza clădirilor, nu putem vedea în totalitate niciun drum până la el. Dacă aplicăm metoda Greedy, vom proceda în felul următor: alegem întotdeauna strada care se îndreaptă cel mai mult în direcția turnului și mergem pe ea. Când ajungem la o intersecție, din nou alegem strada care se îndreaptă cel mai mult către turn. Conform acestei abordări, există următoarele posibilități:

- drumul ales de noi este într-adevăr cel mai scurt și ajungem cel mai repede la destinație - în acest caz am găsit soluția optimă
- drumul găsit este mai lung decât alte drumuri (de exemplu dacă strada care părea că duce chiar către destinație face ulterior un ocol larg, ceea ce ne abate de la direcția noastră), dar în final tot reușim să ajungem la destinație - am găsit o soluție, dar ea nu este optimă
- drumul ales se oprește într-o fundătură din care nu mai putem ieși, deoarece Greedy nu face reveniri la pașii anteriori - nu am găsit nicio soluție, deși ea există

Exemple de abordări Greedy:

- dorim să umplem o cutie cu niște obiecte: alegem întotdeauna obiectul cel mai mare care trebuie pus în cutie (deci cel care ne duce cel mai aproape de umplerea cutiei), până când nu mai avem obiecte sau cutia se umple - vom ajunge întotdeauna la o soluție și în plus soluția necesită un număr minim de operații, dar este posibil ca spațiul din cutie să nu fie folosit în mod optim
- avem de sortat crescător un vector: iterăm de la stânga la dreapta, și pentru fiecare poziție alegem elementul minim din vectorul care începe de la iterator (deci elementul care ne apropie cel mai mult de soluție), element pe care îl vom pune la poziția curentă - acesta este algoritmul de sortare cu două bucle *for*, destul de des folosit. Chiar dacă este unul dintre cei mai ineficienți algoritmi de sortare, implementarea sa este simplă.

Exemplul 1: Se cere o valoare reală cu maxim 2 zecimale, care reprezintă o sumă de lei. Să se afișeze cum poate fi această sumă plătită cu bancnotele și monedele aflate în circulație, astfel încât numărul total de bancnote și de monede să fie minim:

```
#include <stdio.h>

// tip de bancnota sau moneda
typedef struct{
    int val;                                // valoarea in bani
    char *nume;
}Tip;

#define NTIPURI    10

Tip tipuri[NTIPURI]={{50000,"500 lei"},{20000,"200 lei"},{10000,"100 lei"},{5000,"50 lei"},{1000,"10 lei"},
    {500,"5 lei"},{100,"1 leu"},{50,"50 bani"},{10,"10 bani"},{1,"1 ban"}};

int main()
{
    float valLei;
    printf("valoare: ");scanf("%g",&valLei);
    int valBani=(int)(valLei*100);
    int iTip=0;                                // indexul curent in tipuri
    while(valBani){                            // cat timp mai sunt bani de platit
        int n=valBani/tipuri[iTip].val;        // numarul de unitati (bancnote sau monede) cu acea valoare
        if(n){
            printf("%d x %s\n",n,tipuri[iTip].nume);
            valBani-=n*tipuri[iTip].val;
        }
        ++iTip;
    }
    return 0;
}
```

Această problemă face parte din clasa celor care pot fi rezolvate optimal prin metoda Greedy, deoarece soluția optimă globală este compusă din optimele locale. Într-adevăr, dacă pentru o anumită valoare nu am alege numărul maxim de unități superioare care se încadrează în suma de plătit, atunci ar însemna că vom avea nevoie de mai multe unități inferioare cu care să acoperim acea sumă, deci ne depărtăm de optim.

Conform metodei Greedy, alegem varianta cu care ajungem cât mai repede la soluție. În acest caz, ajungem cel mai repede dacă folosim cea mai mare bancnotă sau monedă care se încadrează în suma de plătit. Pentru aceasta, am sortat în mod descrescător bancnotele și monedele, pentru a începe cu cele cu valoarea mai mare. Împărțirea prin care se află valoarea lui n (numărul de unități care se încadrează în valoarea rămasă), are loc în numere întregi, deci întotdeauna vom obține un număr întreg de unități.

Divide and conquer (D&C - divide și cucerește)

Această metodă, care mai este denumită și cu numele latin *Divide et impera*, utilizează următoarea abordare: **se descompune problema în subprobleme independente, care se rezolvă fiecare separat. În final, dacă este cazul, urmează o etapă de combinare a rezultatelor obținute de la rezolvările independente.**

Un aspect foarte important în această abordare este *independența subproblemelor*. Aceasta înseamnă că rezolvarea unei subprobleme nu trebuie să influențeze rezolvarea alteia. Dacă rezolvările unor subprobleme sunt *interdependente*, atunci nu mai vorbim de D&C ci de *programare dinamică*.

Fie următorul exemplu: o localitate este despărțită în două de un râu peste care există un singur pod. Noi ne aflăm la o locație dintr-o parte a râului și dorim să ajungem la o altă locație din cealaltă parte a râului. Din cauză că există doar un singur pod, este evident că drumul pe care îl alegem ca să străbatem prima parte a localității trebuie neapărat să ne ducă la acel pod, indiferent de locația la care dorim să ajungem în cealaltă parte. Analogic, orice drum pe care trebuie să-l parcurgem în cealaltă jumătate a localității, trebuie neapărat să înceapă de la pod. Pe baza acestor observații simple, putem descompune problema găsirii drumului în două subprobleme independente: găsirea unui drum până la pod și găsirea unui drum de la pod la destinație. Rezolvările acestor subprobleme nu depind în niciun fel una de cealaltă. Mai mult decât atât, dacă am fi două persoane, am putea fiecare dintre noi să rezolvăm în mod independent câte o subproblemă, tocmai din cauză că ele nu se influențează în niciun fel una pe cealaltă.

Abordările de tip D&C sunt în special importante în contextul microprocesoarelor cu mai multe nuclee (cores). Dacă avem un algoritm D&C, putem rezolva simultan subprobleme folosind toate nucleele disponibile (multithreading). Chiar dacă rulat într-un singur fir de execuție (thread), algoritmul D&C este mai lent decât alte abordări, atunci când este executat în paralel, el le va depăși ca performanțe. Există și posibilitatea de a distribui într-o rețea de calculatoare rezolvarea subproblemelor (distributed computing), măbind astfel și mai mult capacitățile computaționale. De exemplu, proiectul *Folding@home* utilizează rețele de calculatoare pentru simularea dinamicii proteinelor. O altă posibilitate este utilizarea GPU (Graphics Processing Unit) pentru sarcini computaționale generale. Un GPU poate avea mii de nuclee care pot fi utilizate simultan, ceea ce în anumite situații duce la o accelerare foarte mare a rezolvării.

Există multe probleme care se pretează la o abordare de tip D&C:

- algoritmi de tip *ray tracing*, în care se emite câte o rază de lumină prin fiecare pixel al ecranului și se calculează toate reflexiile și refracțiile ei
- algoritmii de sortare *quicksort* și *merge sort*
- calculul unor fractali (ex: Mandelbrot)
- rezolvarea unor probleme cu structuri de date gen arbori, în care se poate acționa simultan pe toți subarborii (copiii) unui nod

Exemplul 2: Se cere un număr n și coeficienții unui polinom de grad n . Apoi se cere un număr m și coeficienții unui polinom de grad m . Toți coeficienții sunt numere întregi. Gradul unui polinom este maxim 100. Se cere să se calculeze polinomul produs al celor două polinoame de intrare. Calculul fiecărui coeficient al produsului se va face în așa fel încât să poată fi executat pe un calculator separat, presupunând că sunt disponibile toate datele necesare:

```
#include <stdio.h>

#define MAX 100    // gradul maxim al polinoamelor sursa

int a[MAX],b[MAX],c[2*MAX];
int n,m;

// citeste un polinom de grad n
void citire(int *v,int n)
{
    int i;
    for(i=n;i>=0;i--){
        printf("x^%d=",i);
        scanf("%d",&v[i]);
    }
}

// afiseaza un polinom de grad n
void afisare(int *v,int n)
{

```

```

int i;
for(i=n;i>=0;i--){
    if(i!=n)putchar('+');
    printf("%dx^%d",v[i],i);
}
putchar('\n');
}

// calculeaza coeficientul de grad k al inmultirii polinoamelor a si b
void coeficient(int k)
{
    int i;
    for(i=0;i<=n;i++){
        int j=k-i;
        if(j>=0&&j<=m)c[k]+=a[i]*b[j];
    }
}

int main(){
    printf("n=");scanf("%d",&n);citire(a,n);
    printf("m=");scanf("%d",&m);citire(b,m);
    afisare(a,n);
    afisare(b,m);
    int k;
    for(k=0;k<=n+m;k++)coeficient(k);
    afisare(c,n+m);
    return 0;
}

```

Programul este destul de simplu: se citesc coeficienții polinoamelor de intrare în vectorii a și b , iar apoi se calculează rezultatul și se afișează. Partea mai interesantă este funcția *coeficient*, care calculează coeficientul de grad k al rezultatului. Dacă analizăm această funcție, constatăm că ea nu modifică decât $c[k]$, adică doar coeficientul gradului k din rezultat. Orice alți coeficienți din rezultat rămân neschimbați. În acest fel devine posibil să calculăm în paralel fiecare coeficient al rezultatului, având nevoie pentru aceasta doar de datele de intrare și fără să intervenim asupra altor coeficienți.

Pentru comparație, mai jos este o variantă de înmulțire care nu se pretează la procesare distribuită, deoarece din înmulțiri rezultă coeficienți de grade diferite, care trebuie adunați la gradele corespunzătoare $c[i+j]$. Astfel, dacă programul ar rula simultan pe mai multe calculatoare, ar trebui ca fiecare actualizare de coeficienți să fie transmisă tuturor celorlalte calculatoare, deoarece și acestea au nevoie de ea:

```

// varianta care nu se preteaza la procesare distribuită
void inmultire()
{
    int i,j;
    for(i=0;i<=n;i++){
        for(j=0;j<=m;j++){
            c[i+j]+=a[i]*b[j];
        }
    }
}

```

Dacă analizăm cele două variante, constatăm că al doilea algoritm este mai simplu, deoarece nu are *if*-ul interior. Din acest motiv, dacă executăm ambele programe folosind un singur fir de execuție, a doua variantă va fi mai

rapidă. Dacă folosim mai multe fire de execuție, prima variantă va fi mai rapidă, în special pentru polinoame de grad mare.

Aplicații propuse

Notă: pentru fiecare aplicație, specificați într-un scurt comentariu la începutul programului ce metodă de abordare ați folosit și de ce rezolvarea se încadrează în această metodă

Aplicația 11.1: Se cere un $n < 100$, iar apoi coordonatele reale (x, y) a n puncte din plan. Fiecare punct reprezintă poziția unde va trebui dată o gaură într-o placă. Bormașina se află inițial la locația $(0, 0)$. Să se afișeze ordinea de dat găuri, astfel încât întotdeauna bormașina să fie mutată la punctul cel mai apropiat de ea.

Aplicația 11.2: Se cere un n strict pozitiv și patru valori reale: ma , mb , m , d . ma și mb reprezintă masele a două corpuri situate la distanța d unul de celălalt. m este masa unui corp care pornește din a și ajunge în b , din n pași egali. Să se calculeze în fiecare punct al traseului lui m forța F de atracție gravitațională care acționează asupra lui. F va fi cu semn: negativ înseamnă că m este atras către ma , iar pozitiv către mb . Formula atracției gravitaționale dintre două corpuri m_1 și m_2 , situate la distanța d unul de celălalt este: $F = G * m_1 * m_2 / d^2$, unde $G = 6.674e-11$. Masele sunt exprimate în kilograme, distanțele în metri, iar forțele în newtoni.

Aplicația 11.3: Definim *reducerea* unei matrici cu elemente întregi ca fiind valoarea calculată astfel: dacă matricea nu are niciun element, reducerea este 0. Dacă matricea are un element, reducerea este valoarea acelui element. Pentru orice alte cazuri, matricea se subîmparte în 4 matrici prin tăierea ei în cruce, prin mijlocul matricii. Reducerea va fi maximul reducerilor celor două matrici superioare, minus minimul reducerilor celor două matrici inferioare. Să se calculeze reducerea unei matrici de dimensiuni m, n citită de la tastatură.

Aplicația 11.4: La un campionat iau parte n jucători, fiecare definit prin *nume* (max 15 caractere) și *valoare* (int). Jucătorii sunt distribuiți în m grupe, n divizibil cu m . Distribuția jucătorilor în grupe se face după valoarea lor, astfel încât cei mai valoroși m jucători să fie fiecare în altă grupă, apoi următorii m cei mai valoroși rămași să fie și ei în grupe diferite și tot așa, până când toți jucătorii sunt distribuiți. Să se afișeze împărțirea jucătorilor pe grupe, pentru valori citite dintr-un fișier.

Aplicația 11.5: Să se scrie o funcție care primește un număr întreg $0 \leq n < 1000$ și îl afișează în formă literară. Să se apeleze funcția pentru numere introduse de la tastatură, până la apariția valorii 1000. Exemple: $108 \rightarrow$ o sută opt; $520 \rightarrow$ cinci sute douazeci; $16 \rightarrow$ șaisprezece