Laborator 01: Divide et Impera

Responsabili:

- Darius Neațu
- Ștefan Popa

Autori:

- Radu Vișan (2018)
- Cristian Banu (2018)
- Darius Neațu (2018)

Obiective laborator

- Înțelegerea conceptului teoretic din spatele descompunerii unei probleme
- Rezolvarea de probleme abordabile folosind conceptul de Divide et Impera

Precizari initiale

> Toate exemplele de cod se gasesc in demo-lab01.zip.
>
> Exemplele de cod apar incorporate si in textul laboratorului pentru a facilita parcurgerea cursiva a acestuia.

- Toate bucatile de cod prezentate in partea introductiva a laboratorului (inainte de exercitii) au fost testate. Cu toate acestea, este posibil ca din cauza mai multor factori (formatare, caractere invizibile puse de browser etc) un simplu copy-paste sa nu fie de ajuns pentru a compila codul.
- Va rugam sa incercati si codul din arhiva **demo-lab01.zip**, inainte de a raporta ca ceva nu merge. :D
- Pentru orice problema legata de continutul acestei pagini, va rugam sa dati email unuia dintre responsabili.

Importanţă – aplicaţii practice

Paradigma Divide et Impera stă la baza construirii de algoritmi eficienți pentru diverse probleme:

- Sortări (ex: MergeSort [1], QuickSort [2])
- Înmulțirea numerelor mari (ex: Karatsuba [3])
- Analiza sintactică (ex: parsere top-down [4])
- Calcularea transformatei Fourier discretă (ex: FFT [5])

Un alt domeniu de utilizare a tehnicii divide et impera este programarea paralelă pe mai multe procesoare, sub-problemele fiind executate pe mașini diferite.

Prezentarea generală a problemei

O descriere a tehnicii D&I: "Divide and Conquer algorithms break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem." [7]

Deci un algoritm D&I **împarte problema** în mai multe subprobleme similare cu problema inițială și de dimensiuni mai mici, **rezolva subproblemele** recursiv și apoi **combina soluțiile** obținute pentru a obține soluția problemei inițiale.

Sunt trei pași pentru aplicarea algoritmului D&I:

- **Divide**: împarte problema în una sau mai multe *probleme similare de dimensiuni mai mici*.
- **Impera** (stăpânește): rezolva subprobleme recursiv; dacă dimensiunea sub-problemelor este mica se rezolva iterativ.
- **Combină**: combină soluțiile subproblemelor pentru a obține soluția problemei inițiale.

Complexitatea algoritmilor D&I se calculează după formula:

$$T(n) = D(n) + S(n) + C(n),$$

unde $D(n)$, $S(n)$ și $C(n)$ reprezintă complexitățile celor 3 pași descriși mai sus: divide, stăpânește, respectiv combină.

Probleme clasice

MergeSort

Enunt

Sortarea prin interclasare (MergeSort [1]) este un algoritm de sortare de vectori ce folosește paradigma D&I:

- **Divide**: împarte vectorul inițial în doi sub-vectori de dimensiune n/2.
- **Stăpânește**: sortează cei doi sub-vectori recursiv folosind sortarea prin interclasare; recursivitatea se oprește când dimensiunea unui sub-vector este 1 (deja sortat).
- **Combina**: Interclasează cei doi sub-vectori sortați pentru a obține vectorul inițial sortat.

Pseudocod

Mai jos gasiti algoritmul MergeSort scris in pseudocod.

Pseudocod

Implementare in C++

Complexitate

Complexitatea algoritmului este dată de formula: $T(n) = D(n) + S(n) + C(n)$, unde $D(n) = O(1)$, $S(n) = 2 * T(n/2)$ și $C(n) = O(n)$, rezulta $T(n) = 2 * T(n/2) + O(n)$.

Folosind teorema Master [8] găsim complexitatea algoritmului: $T(n) = O(n * log(n))$.

- **complexitate temporala** : $T = O(n * log(n))$
  - Ce inseamna aceasta metrica? Masoara efectiv **timpul de executie** al algoritmului (nu include citiri, afisari etc).
- **complexitate spatiala** : $S = O(n)$
  - Ce inseamna aceasta metrica? Masoara efectiv **memoria suplimentara** folosita de algoritm (in acest caz ne referim strict la buffer-ul temporar).

> Retineti cele doua conventii despre complexitati de mai sus. Le vom folosi pentru restul semestrului.

Binary Search

Enunt

Se dă un **vector sortat crescător** ($v[1]$, $v[2]$, ..., $v[n]$) ce conține valori reale distincte și o valoare x.

Sa se găsească la ce **poziție** apare x în vectorul dat.

Rezolvare

BinarySearch (Cautare Binara), se rezolva cu un algoritm D&I:

- **Divide**: împărțim vectorul în doi sub-vectori de dimensiune n/2.
- **Stăpânește**: aplicăm algoritmul de căutare binară pe sub-vectorul care conține valoarea căutată.
- **Combină**: soluția sub-problemei devine soluția problemei inițiale, motiv pentru care nu mai este nevoie de etapa de combinare.

Pseudocod

```
BinarySearch(v, left, right, x) {             // functia returneaza pozitia x pe care se afla numarul
    // vom cauta cat timp intervalul de cautare nu a fost inca epuizat (are cel putin un element)
    while (left <= right) {
        mid = (left + right) / 2          // mijlocul intervalului de cautare

        if (x == v[mid]) return mid;      // elementul cautat este cel din mijloc
        if (x < v[mid])  right = mid - 1; // elementul cautat este mai mic decat cel din mijloc, 
        if (x > v[mid])  left  = mid + 1; // elementul cautat este mai mare decat cel din mijloc,
    }

    return -1;                            // in acest punct ajungem daca si numai daca x nu a fos
}
```

Complexitate

- **complexitate temporala** : $T = O(log(n))$
  - se deduce din recurenta $T(n) = T(n/2) + O(1)$
- * **complexitate spatiala** : $S = O(1)$
  - nu avem structuri de date complexe auxiliare
  - atragem atentia ca acest algoritm se poate implementa si **recursiv**, caz in care complexitatea spatiala devine $O(log(n))$, intrucat salvam pe stiva $O(log(n))$ parametri (intregi, referinte)

Turnurile din Hanoi

Enunt

Se considera 3 tije S (**sursa**), D (**destinatie**), aux (**auxiliar**) și n discuri de dimensiuni distincte ($1, 2, ..., n$ - ordinea crescatoare a dimensiunilor) situate inițial toate pe tija S în ordinea $1, 2, ..., n$ (de la vârf către bază).

Singura operație care se poate efectua este de a selecta un disc ce se află în vârful unei tije și plasarea lui în vârful altei tije astfel încât să fie așezat deasupra unui disc de dimensiune mai mare decât a sa.

Sa se găsească un algoritm prin care se mută toate discurile de pe tija S pe tija D (problema turnurilor din Hanoi).

Solutie

Pentru rezolvarea problemei folosim următoarea strategie [9]:

- mutam primele $n - 1$ discuri de pe tija S pe tija aux folosindu-ne de tija D.
- mutam discul n pe tija D.
- mutam apoi cele $n - 1$ discuri de pe tija aux pe tija D folosindu-ne de tija S.

Ideea din spate este ca avem mereu o singura sursa si o singura destinatie sa atingem un scop. Intotdeauna a 3-a tija va fi considerata auxiliara si poate fi folosita pentru a atinge scopul propus.

Algoritm

```
// muta n discuri de pe tija S pe tija D folosind tija aux
Hanoi(n, S, D, aux) {
    if (n >= 1) {
        Hanoi(n - 1, S, aux, D);   // mut n-1 discuri de pe sursa (S) pe auxiliar (aux)
                                   // in aceasta subproblema sursa este S, destinatia este aux, inte

        Muta_disc(S, D);           // acum pot muta direct discul n de pe sursa (S) pe destinatie (D)

        Hanoi(n - 1, aux, D, S);   // mut n-1 discuri de pe sursa (aux, aici sunt ele momentan) pe de
                                   // in aceasta subproblema, S este auxiliar, intrucat este tija lib
    }
}
```

Complexitate

- **complexitate temporala** : $T(n) = O(2^n)$
  - se deduce din recurenta $T(n) = 2 * T(n - 1) + O(1)$
- * **complexitate spatiala** : $S(n) = O(n)$
  - la un moment dat, nivelul maxim de recursivitate este n

ZParcurgere

Enunt

Gigel are o tabla patratica de dimensiuni $2^n * 2^n$. Ar vrea sa scrie pe patratelele tablei numere naturale cuprinse intre 1 si $2^n * 2^n$ conform unei parcurgeri mai deosebite pe care o numeste **Z-parcurgere**.

O Z-parcurgere viziteaza recursiv cele patru cadrane ale tablei in ordinea: **stanga-sus**, **dreapta-sus**, **stanga-jos**, **dreapta-jos**.

La un moment dat Gigel ar vrea sa stie ce **numar de ordine** trebuie sa scrie conform Z-parcurgerii pe anumite patratele date prin coordonatele lor (x, y). Gigel incepe umplerea tablei **intotdeauna** din coltul din stanga-sus.

Exemplu 1

Exemplu 2

Solutie

Analizand modul in care se **completeaza** tabloul/matricea din enunt, observam ca la fiecare etapa impartim matricea (**problema**) in 4 submatrici (**4 subprobleme**). De asemenea, sirul de numere pe care dorim sa il punem in matrice se imparte in 4 secvente, fiecare corespunzand unei submatrici.

Observam astfel ca problema suporta **descompunerea** in **subprobleme disjuncte** si cu **structura similara**, ceea ce ne face sa ne gandim la o solutie cu Divide et Impera.

Complexitate

- **complexitate temporala** : $T = O(n)$
  - $\log_4(2^n) = \frac{1}{2}\log_2(2^n) = \frac{1}{2}n$
- * **complexitate spatiala** : $S = O(n)$
  - stocam parametri pentru recursivitate
  - solutia se poate implementa si iterativ, caz in care $S = O(1)$; deoarece dimensiunile spatiului de cautare sunt $2^n * 2^n$, n este foarte mic ($n <= 15$), de aceea o solutie iterativa nu aduce nici un castig **efectiv** in aceasta situatie

Concluzii

Divide et impera este o tehnică folosită pentru a realiza soluții pentru o anumita clasa de probleme: acestea contin **subprobleme disjuncte** si cu **structura similara**. In cadrul acestei tehnici se disting trei etape: divide, stăpânește și combină.

Mai multe exemple de algoritmi care folosesc tehnica divide et impera puteți găsi la [11].

Exercitii

> In acest laborator vom folosi scheletul de laborator din arhiva skel-lab01.zip.

Count occurrences

Se da un sir sortat **v** cu **n** elemente. Gasiti numarul de elemente egale cu **x** din sir.

Exemplu 1

Task-uri:

- Aceasta problema este deja rezolvata. Pentru a va acomoda cu scheletul, va trebui sa faceti cativa pasi:
  - Rulati comanda ./check.sh si cititi cum se foloseste checker-ul.
  - Rulati comanda necesara pentru a rula task-ul 1. Sursa nu implementeaza corect algoritmul si returneaza valori default. Din acest motiv primiti mesajul **WRONG ANSWER**.
  - Copiati urmatoarea sursa in folderul corespunzator. Rulati comanda anterioara. Observati mesajele afisate cand ati rezolvat corect un task.

Solutie C++

Solutie Java

- Intelegeti solutia oferita impreuna cu asistentul vostru.
- Care este complexitatea solutiei (timp + spatiu)? De ce?

SQRT

Se da un numar real **n**. Scrieti un algoritm de complexitate **O(log n)** care sa calculeze $sqrt(n)$ cu o precizie de 0.001.

Exemplu 1

> Pentru a putea trece testele, trebuie sa afisati rezultatul cu **cel putin** 4 zecimale.

ZParcurgere

Rezolvati problema ZParcurgere folosind scheletul pus la dispozitie. Enuntul si explicatiile le gasiti in partea de seminar.

Exponentiere logaritmica

Se dau doua numere naturale **base** si **exponent**. Scrieti un algoritm de complexitate $O(log(exponent))$ care sa calculeze $base^{exponent}$ $\% \ MOD$.

> Intrucat expresia $base^{exponent}$ este foarte mare, dorim sa aflam doar **restul** impartirii lui la un numar **MOD**.
>
> Proprietati matematice necesare:
>
> - $(a + b) \% MOD = ((a \% MOD) + (b \% MOD)) \% MOD$
> - $(a * b) \% MOD = ((a \% MOD) * (b \% MOD)) \% MOD$
>
> Atentie la inmultire! Rezultatul **temporar** poate provoca un overflow. Solutii:
>
> - **C++**: $a * b \Rightarrow 1LL * a * b$
> - **Java**: $a * b \Rightarrow 1L * a * b$

Exemplu 1

Bonus

Inversiuni

ClassicTask

Extra

Statistici de ordine

Missing number

Fractal

SSM

Secventa descrescatoare

Table of Contents