Laborator 01: Divide et Impera

Responsabili:

- Radu Vișan [mailto:visanr95@gmail.com]
- Cristian Banu [mailto:cristb@gmail.com]
- Darius Neațu [mailto:neatudarius@gmail.com]

Obiective laborator

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

Precizari initiale

Toate exemplele de cod se gasesc in demo-lab01.zip.

Acestea apar incorporate si in textul laboratorului pentru a facilita parcurgerea cursiva a laboratorului.

- 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] [http://www.sorting-algorithms.com/merge-sort], QuickSort [2] [http://www.sorting-algorithms.com/quick-sort])
- Înmulțirea numerelor mari (ex: Karatsuba [3] [http://en.wikipedia.org/wiki/Karatsuba_algorithm])
- Analiza sintactică (ex: parsere top-down [4] [http://en.wikipedia.org/wiki/Top-down_parser])
- Calcularea transformatei Fourier discretă (ex: FFT [5] [http://en.wikipedia.org/wiki/Fast_Fourier_transform])

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 şi de dimensiuni mai mici, **rezolva sub-problemele** recursiv şi apoi **combina soluțiile** obţinute 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 sub-problemelor pentru a obține soluția problemei inițiale.

Complexitatea algoritmilor D&I se calculează după formula:

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

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

Probleme clasice

MergeSort

Enunt

Sortarea prin interclasare (MergeSort [1] [http://www.sorting-algorithms.com/merge-sort]) 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.

```
MergeSort(v, start, end)                // v – vector, start – limită inferioră, end – limită superioară
        if (start == end) return;       // condiția de oprire
        mid = (start + end) / 2;        // etapa divide
        MergeSort(v, start, mid);       // etapa stăpânește
        MergeSort(v, mid + 1, end);
        Merge(v, start, end);           // etapa combină

Merge(v, start, end)                    // interclasare sub-vectori
        // indecsi
        mid = (start + end) / 2;
        i = start;
        j = mid + 1;
        k = 1;

        tmp = buffer temporar in care incap (end - start + 1) numere
        while (i <= mid && j <= end)
                if (v[i] <= v[j]) tmp[k++] = v[i++];
                             else tmp[k++] = v[j++];

        while (i <= mid)
                tmp[k++] = v[i++];

        while (j <= end)
                tmp[k++] = v[j++];

        copy(v[start..end], tmp[1..k-1]);
```

Mai jos puteti gasi o implementare in C++.

```
#include <bits/stdc++.h>
using namespace std;

vector<int> v;

void merge_halves(int left, int right) {
        int mid = (left + right) / 2;
        vector<int> aux;
        int i = left, j = mid + 1;

        while (i <= mid && j <= right) {
                if (v[i] <= v[j]) {
                        aux.push_back(v[i]);
                        i++;
                } else {
                        aux.push_back(v[j]);
                        j++;
                }
        }

        while (i <= mid) {
                aux.push_back(v[i]);
                i++;
        }

        while (j <= right) {
                aux.push_back(v[j]);
                j++;
        }

        for (int k = left; k <= right; k++) {
                v[k] = aux[k - left];
        }
}


void merge_sort(int left, int right) {
        if (left >= right) return ;
        int mid = (left + right) / 2;

        merge_sort(left, mid);
        merge_sort(mid + 1, right);
        merge_halves(left, right);
}
```

```
int main() {
        random_device rd;
        for (int i = 0; i < 10; i++) {
                v.push_back(rd() % 100);
        }

        cout << "Vectorul initial: ";
        for (int i = 0; i < v.size(); i++) {
                cout << v[i] << " ";
        }
        cout << "\n";

        merge_sort(0, v.size() - 1);

        cout << "Vectorul sortat: ";
        for (int i = 0; i < v.size(); i++) {
                cout << v[i] << " ";
        }
        cout << "\n";

        return 0;
}
```

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] [http://people.csail.mit.edu/thies/6.046-web/master.pdf] 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 x (oricare pozitie) sa
       // 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, ne mutam in prima jumatate
            if (x > v[mid])  left  = mid + 1; // elementul cautat este mai mare decat cel din mijloc, ne mutam in a doua jumatate
       }

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

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**) şi n discuri de dimensiuni distincte ($1, 2, \ldots, n$ - ordinea crescătoare a dimensiunilor) situate inițial toate pe tija S în ordinea $1, 2, \ldots, n$ (de la vârf către baza).

Singura operație care se poate efectua este de a selecta un disc ce se află în vârful unei tije şi plasarea lui în vârful altei tije astfel încât să fie aşezat 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] [http://www.mathcs.org/java/programs/Hanoi/index.html]:

- 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, intermediarul este D

        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 destinatie (D - scop final)
                                   // in aceasta subproblema, S este auxiliar, intrucat este tija libera
    }
}
```

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.

$n = 1$ si $(x, y) = (2, 2)$

Raspuns: 4

Explicatie: Matricea arata ca in exemplul urmator.

1	2
3	4

$n = 2$ si $(x, y) = (3, 3)$

Raspuns: 13

Explicatie: Matricea arata ca in exemplul urmator.

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

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 dimensinile 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 solutii pentru o anumita clasa de probleme: acestea contin **subprobleme disjuncte** si cu **structura similara**. În 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] [http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf].

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.

$n = 6$ si $x = 10$

i	1	2	3	4	5	6
v	1	2	4	10	10	20

Raspuns: 2

Explicatie: 10 apare de 2 ori in sir.

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.

Sursa **main.cpp** asociata cu task 1 este mai jos.

```
#include <bits/stdc++.h>
using namespace std;

class Task {
public:
        void solve() {
                read_input();
```

```
                print_output(get_result());
        }

private:
        void read_input() {
                ifstream fin("in");
                fin >> n;
                for (int i = 0, e; i < n; i++) {
                        fin >> e;
                        v.push_back(e);
                }
                fin >> x;
                fin.close();
        }

        int find_first() {
                int left = 0, right = n - 1, mid, res = -1;
                while (left <= right) {
                        mid = (left + right) / 2;
                        if (v[mid] >= x) {
                                res = mid;
                                right = mid - 1;
                        } else {
                                left = mid + 1;
                        }
                }
                return res;
        }

        int find_last() {
                int left = 0, right = n - 1, mid, res = -1;
                while (left <= right) {
                        mid = (left + right) / 2;
                        if (v[mid] <= x) {
                                res = mid;
                                left = mid + 1;
                        } else {
                                right = mid - 1;
                        }
                }
                return res;
        }

        int get_result() {
                int first = find_first();
                int last = find_last();
                if (first == -1 || last == -1) {
                        return 0;
                }
                return last - first + 1;
        }

        void print_output(int result) {
                ofstream fout("out");
                fout << result;
                fout.close();
        }

        int n, x;
        vector<int> v;
};

int main() {
        Task task;
        task.solve();
        return 0;
}
```

Sursa **Main.java** asociata cu task 1 este mai jos.

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class Main {
        static class Task {
                public final static String INPUT_FILE = "in";
                public final static String OUTPUT_FILE = "out";

                int n;
                int[] v;
                int x;

                private void readInput() {
                        try {
                                Scanner sc = new Scanner(new File(INPUT_FILE));
                                n = sc.nextInt();
                                v = new int[n];
                                for (int i = 0; i < n; i++) {
                                        v[i] = sc.nextInt();
                                }
```

```
                        x = sc.nextInt();
                        sc.close();
                } catch (IOException e) {
                        throw new RuntimeException(e);
                }
        }

        private void writeOutput(int count) {
                try {
                        PrintWriter pw = new PrintWriter(new File(OUTPUT_FILE));
                        pw.printf("%d\n", count);
                        pw.close();
                } catch (IOException e) {
                        throw new RuntimeException(e);
                }
        }

        private int findFirst() {
                int left = 0, right = n - 1, mid, res = -1;
                while (left <= right) {
                        mid = (left + right) / 2;
                        if (v[mid] >= x) {
                                res = mid;
                                right = mid - 1;
                        } else {
                                left = mid + 1;
                        }
                }
                return res;
        }

        private int findLast() {
                int left = 0, right = n - 1, mid, res = -1;
                while (left <= right) {
                        mid = (left + right) / 2;
                        if (v[mid] <= x) {
                                res = mid;
                                left = mid + 1;
                        } else {
                                right = mid - 1;
                        }
                }
                return res;
        }

        private int getAnswer() {
                int first = findFirst();
                int last = findLast();
                if (first == -1 || last == -1) {
                        return 0;
                }
                return last - first + 1;
        }

        public void solve() {
                readInput();
                writeOutput(getAnswer());
        }
    }

    public static void main(String[] args) {
            new Task().solve();
    }
}
```

- 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.

$n = 0.25$

Raspuns: orice valoare intre 0.499 si 0.501 (inclusiv)

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**: $a * b \Rightarrow 1L * a * b$

$base = 2$, $exponent = 10$, $MOD = 5$

Raspuns: 4

Explicatie: $2^{10}\ \%\ 5 = 4$

Bonus

Se da un sir S de n numere intregi. Sa se detemine cate inversiuni sunt in sirul dat. Numim inversiune o pereche de indici $1 <= i < j <= n$ astfel incat $S[i] > S[j]$

Exemplu: in sirul `{0 1 9 4 5 7 6 8 2}` sunt 12 inversiuni.

Aceasta problema nu are schelet.

Extra

Se dă un vector de numere întregi neordonate. Scriind o funcție de partitionare, folosiți **Divide et Impera** pentru

- a determina a k-lea element ca mărime din vector
- a sorta vectorii prin QuickSort

Puteti testa problema partition aici [https://leetcode.com/problems/kth-largest-element-in-an-array/description/]. Problema QuickSort [https://en.wikipedia.org/wiki/Quicksort] (chiar si MergeSort) poate fi testata aici [https://infoarena.ro/problema/algsort].

Exemplu: pentru vectorul `{0 1 2 4 5 7 6 8 9}`, al 3-lea element ca ordine este 2, iar vectorul sortat este {0 1 2 4 5 6 7 8 9}

Se dau $n - 1$ numere naturale distincte intre 0 si $n - 1$. Scriind o functie de partitionare, determinati numarul lipsa.

Exemplu: pentru $n = 9$ si vectorul 019457682, numarul lipsa este 3.

Puteti rezolva aceasta problema pe infoarena [https://infoarena.ro/problema/fractal].

Puteti rezolva aceasta problema de aici [https://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-03#ssm].

Pentru problema SSM vom studia o solutie potrivita in lab03. Puteti sa incercati sa o rezolvati cu Divide et Impera pentru a exersa cele invatate astazi.

Dându-se N numere întregi sub forma unei secvenţe de numere strict crescătoare, care se continuă cu o secvenţă de întregi strict descrescătoare, se doreşte determinarea punctului din întregul şir înaintea căruia toate elementele sunt strict crescătoare, şi dupa care, toate elementele sunt strict descrescătoare. Considerăm evident faptul că acest punct nu există dacă cele N numere sunt dispuse într-un şir fie doar strict crescător, fie doar strict descrescător.