Laborator 01: Divide et Impera

Responsabili:

- Darius-Florentin Neațu (2017-2021) [mailto:neatudarius@gmail.com]
- Radu Nichita (2021) [mailto:radunichita99@gmail.com]
- Cristian Olaru (2021) [mailto:cristianolaru99@gmail.com]
- Miruna-Elena Banu (2021) [mailto:mirunaelena.banu@gmail.com]
- Mara-Ioana Nicolae (2021) [mailto:maraioana9967@gmail.com]
- Ștefan Popa (2018-2020) [mailto:stefanpopa2209@gmail.com]

Autori:

- Radu Vișan (2018) [mailto:visanr95@gmail.com]
- Cristian Banu (2018) [mailto:cristb@gmail.com]
- Darius-Florentin Neațu (2018) [mailto:neatudarius@gmail.com]

Obiective laborator

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

Precizări inițiale

Toate exemplele de cod se găsesc pe pagina pa-lab::demo/lab01 [https://github.com/acs-pa/pa-lab/tree/main/demo/lab01].

Exemplele de cod apar încorporate și în textul laboratorului pentru a facilita parcurgerea cursivă a acestuia. ATENȚIE! Varianta actualizată a acestor exemple se găsește întotdeauna pe GitHub.

- Toate bucățile de cod prezentate în partea introductivă a laboratorului (înainte de exerciții) 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 să nu fie de ajuns pentru a compila codul.
- Vă rugam să compilați **DOAR** codul de pe GitHub. Pentru raportarea problemelor, contactați unul dintre maintaineri.
- Pentru orice problemă legată de conținutul acestei pagini, vă rugam să dați e-mail 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.”

Deci un algoritm D&I **împarte problema** în mai multe subprobleme similare cu problema inițială şi de dimensiuni mai mici, **rezolvă subproblemele** recursiv şi apoi **combină 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): rezolvă subproblemele recursiv; dacă dimensiunea sub-problemelor este mică, se rezolvă 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 $D(n)$, $S(n)$ şi $C(n)$ reprezintă complexitățile celor 3 pași descriși mai sus: divide, stăpânește, respectiv combină.

Probleme clasice

MergeSort

Enunț

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 găsiți algoritmul MergeSort scris în 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 găsi o implementare în 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 temporală** : $T = O(n * log(n))$
  - Ce înseamnă această metrică? Măsoara efectiv **timpul de execuție** al algoritmului (nu include citiri, afișări etc).
- **complexitate spatiala** : $S = O(n)$
  - Ce înseamnă această metrica? Măsoara efectiv **memoria suplimentară** folosită de algoritm (în acest caz ne referim strict la buffer-ul temporar).

Rețineți cele două convenții despre complexități de mai sus. Le vom folosi pentru restul semestrului.

Binary Search

Enunț

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

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

Rezolvare

BinarySearch (Căutare Binară), 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 temporală** : $T = O(log(n))$
  - se deduce din recurența $T(n) = T(n/2) + O(1)$
- **complexitate spațială** : $S = O(1)$
  - nu avem structuri de date complexe auxiliare
  - atragem atenția că acest algoritm se poate implementa și **recursiv**, caz în care complexitatea spațiala devine $O(log(n))$, întrucât salvăm pe stivă $O(log(n))$ parametri (întregi, referințe)

Turnurile din Hanoi

Enunț

Se consideră 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 așezat deasupra unui disc de dimensiune mai mare decât a sa.

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

Soluție

Pentru rezolvarea problemei folosim următoarea strategie [9] [http://www.mathcs.org/java/programs/Hanoi/index.html]:

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

Ideea din spate este că avem mereu o singura sursă si o singură destinație să atingem un scop. Întotdeauna a 3-a tija va fi considerată auxiliară și poate fi folosită 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 temporală** : $T(n) = O(2^n)$
  - se deduce din recurența $T(n) = 2 * T(n-1) + O(1)$
- **complexitate spațială** : $S(n) = O(n)$
  - la un moment dat, nivelul maxim de recursivitate este n

ZParcurgere

Enunț

Gigel are o tablă pătratică de dimensiuni $2^n * 2^n$. Ar vrea să scrie pe pătrățelele tablei numere naturale cuprinse între 1 si $2^n * 2^n$ conform unei parcurgeri mai deosebite pe care o numește **Z-parcurgere**.

O Z-parcurgere vizitează recursiv cele patru cadrane ale tablei în ordinea: **stânga-sus**, **dreapta-sus**, **stânga-jos**, **dreapta-jos**.

La un moment dat, Gigel ar vrea să știe ce **număr de ordine** trebuie să scrie conform Z-parcurgerii pe anumite pătrațele date prin coordonatele lor (x, y). Gigel incepe umplerea tablei **întotdeauna** din colțul din stânga-sus.

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

Răspuns: 4

Explicație: Matricea arată ca în exemplul următor.

1	2
3	4

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

Răspuns: 13

Explicație: Matricea arată ca în exemplul următor.

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

Soluție

Analizând modul în care se **completează** tabloul/matricea din enunț, observăm că la fiecare etapa împărțim matricea (**problema**) în 4 submatrici (**4 subprobleme**). De asemenea, șirul de numere pe care dorim să il punem în matrice se împarte în 4 secvente, fiecare corespunzând unei submatrici.

Observăm astfel că problema suportă **descompunerea** în **subprobleme disjuncte** și cu **structura similara**, ceea ce ne face să ne gândim la o soluție cu Divide et Impera.

Complexitate

- **complexitate temporală** : $T = O(n)$
  - $\log_4(2^n) = \frac{1}{2}\log_2(2^n) = \frac{1}{2}n$
- **complexitate spatială** : $S = O(n)$
  - stocăm parametri pentru recursivitate
  - solutia se poate implementa si iterativ, caz in care $S = O(1)$; deoarece dimensiunile spațiului de căutare sunt $2^n * 2^n$, n este foarte mic ($n <= 15$), de aceea o solutie iterativa nu aduce nici un câștig **efectiv** în această situație.

Concluzii

Divide et impera este o tehnică folosită pentru a realiza soluții pentru o anumita clasă de probleme: acestea contin **subprobleme disjuncte** și cu **structură similară**. 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].

Exerciții

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab01 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab01].

Count occurrences

Se dă un șir sortat **v** cu **n** elemente. Găsiți numărul de elemente egale cu **x** din șir.

$n = 6$ si $x = 10$

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

Răspuns: 2

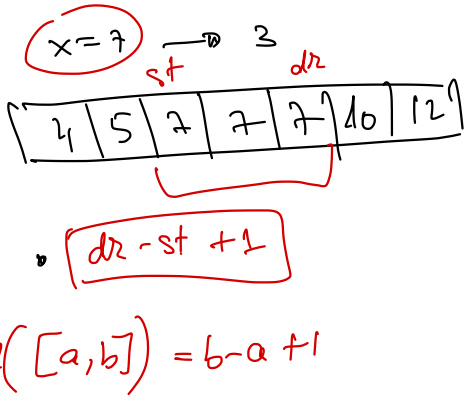
Explicație: 10 apare de 2 ori in sir.

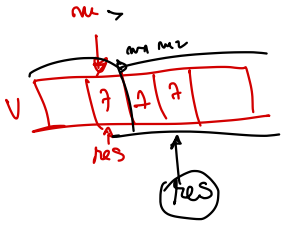
Task-uri:

- Aceasta problemă este deja rezolvată. Pentru a vă acomoda cu scheletul, va trebui sa faceți câțiva pași:
  - Rulați comanda *./check.sh* si cititi cum se foloseste checker-ul.
  - Rulați 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 următoarea sursă în folderul corespunzător. Rulați comanda anterioară. Observați mesajele afișate când ați 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:
    int n, x;
    vector<int> v;

    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();
    }
};

// [ATENTIE] NU modifica functia main!
int main() {
    // * se aloca un obiect Task pe heap
    // (se presupune ca e prea mare pentru a fi alocat pe stiva)
    // * se apeleaza metoda solve()
    // (citire, rezolvare, printare)
    // * se distruge obiectul si se elibereaza memoria
    auto* task = new (std::nothrow) Task{}; // hint: cppreference/nothrow
    if (!task) {
        std::cerr << "new failed\n";
        return -1;
    }
    task->solve();
    delete task;
    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();
        }
}
```

- Înțelegeți soluția oferită împreună cu asistentul vostru.
- Care este complexitatea soluției (timp + spațiu)? De ce?

SQRT

Se dă un număr real **n**. Scrieți un algoritm de complexitate **O(log n)** care să calculeze $sqrt(n)$ cu o precizie de 0.001.

$n = 0.25$

Răspuns: orice valoare intre 0.499 si 0.501 (inclusiv)

Pentru a putea trece testele, trebuie sa afișați rezultatul cu **cel puțin** 4 zecimale.

ZParcurgere

Rezolvați problema ZParcurgere folosind scheletul pus la dispoziție. Enunțul și explicațiile le găsiți în partea de seminar.

Exponențiere logaritmică

Se dau două numere naturale **base** și **exponent**. Scrieți un algoritm de complexitate $O(log(exponent))$care să calculeze $base^{exponent}\ \%\ MOD$.

Întrucât expresia $base^{exponent}$ este foarte mare, dorim să aflăm doar **restul** împărțirii lui la un număr **MOD**.

Proprietăți matematice necesare:

- $(a + b)\ \%\ MOD = ((a\ \%\ MOD) + (b\ \%\ MOD))\ \%\ MOD$
- $(a\ *\ b)\ \%\ MOD = ((a\ \%\ MOD)\ *\ (b\ \%\ MOD))\ \%\ MOD$

Atenție la înmulțire! Rezultatul **temporar** poate provoca un overflow.

Soluții:

- **C++**: $a * b \Rightarrow 1LL * a * b$
- **Java**: $a * b \Rightarrow 1L * a * b$

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

Răspuns: 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.

Testați soluția voastră de la **Exponentiere logaritmica** pe infoarena, la problema ClassicTask [https://infoarena.ro/problema/classictask] (trebuie să modificați numele fișierelor).

Identificați problema și modificați sursa astfel încât să luați punctaj maxim.

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.

Puteți rezolva această problemă pe infoarena [https://infoarena.ro/problema/fractal].

Puteți rezolva această problemă pe infoarena [https://www.infoarena.ro/problema/hanoi].

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.

Puteți rezolva această problemă pe leetcode [https://leetcode.com/problems/k-closest-points-to-origin/]

Puteți rezolva această problemă pe leetcode [https://leetcode.com/problems/merge-k-sorted-lists/]

Referințe

[0] Chapter **Divide-and-Conquer**, “Introduction to Algorithms”, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

pa/laboratoare/laborator-01.txt · Last modified: 2021/03/10 23:06 by radu.nichita