1) DIVIDE ET IMPERA

ALGORITMUL LUI STRASSEN DE INMULTIRE A MATRICELOR

Enunt: Fie A si B doua matrice patratice. Aflati C = A*B produsul lor.

Descrierea solutiei: Cu totii stim algoritmul naiv de inmultire a matricelor ce trece pe rand prin liniile matricei A, apoi pe coloanele matricei A, respectiv liniile matricei B, iar in final pe coloanele din matricea B, astfel reiesind un algoritm de complexitate O(n^3).

Insa, Volker Strassen a reusit sa arate ca exista un algoritm mai eficient, a carui performanta este remarcata mai ales la inmultirea matricelor de dimensiuni medii si mari.

Toata ideea vine din a inmulti 2 matrice ca blocuri si nu ca intreg, si anume:

Fie A = $\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$ si B = $\begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$

unde: A si B apartin $\mathbb{R}^n$

A11, A12, A21, A22 apartin $\mathbb{R}^{\frac{n}{2}}$

B11, B12, B21, B22 apartin $\mathbb{R}^{\frac{n}{2}}$

Astfel produsul A x B devine: $\begin{bmatrix} A11 * B11 + A12 * B21 & A11 * B12 + A12 * B22 \\ A21 * B11 + A22 * B21 & A21 * B12 + A22 * B22 \end{bmatrix}$ (1)

Aplicand aceeasi idee si noului produs obtinem urmatoarea relatie de recurenta:

$$T(n) = 8 * T(n/2) + O(n^2); \qquad (2)$$

unde 8 vine de la numarul de inmultiri de matrici ce trebuie realizate la fiecare pas, iar O(n^2) vine de la adunarea matricelor rezultate (avem de 4 ori cate 2 matrice ce trebuiesc adunate, fiecare adunare avand o complexitate de O(n^2);

Analizand relatia de recurenta (2), obtinem: a=8, b=2, iar f(n) = O(n^2)

Reiese usor ca ne aflam in cazul I al teoremei Master:

O(n^2) ∈ O($n^{\log_2 8}$)

O(n^2) ∈ O(n^3), lucru adevarat pentru ca orice f(n) ∈ O(n^2) apartine si multimii O(n^3), in care O(n^2) este inclusa.

Dar aflandu-ne in cazul I => ca recurenta noastra T(n) ∈ O(n^3), deci in continuare nu rezolvam aproape nimic din punct de vedere al complexitatii.

In acest moment, putem include ideea lui Strassen de a reduce numarul de inmultiri realizate in cadrul unui pas de la 8 la 7. Iata ce notatii se fac pentru a realiza acest lucru:

p1 = A11 * (B12 – B22)

p2 = (A11 + A12) * B22

p3 = (A21 + A22) * B11

p4 = A22 * (B21- B11)

p5 = (A11 + A22) * (B11 + B22)

p6 = (A12 – A22) * (B21 + B22)

p7 = (A11 – A21) * (B11 + B12)

Asadar, in urma inmultirii matricelor A si B, optinem matricea C, care in final se poate descompune la rezultatul prezentat in (1):

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} * \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

Complexitate:

De aceasta data, noua relatie de recurenta va arata in felul urmator:

$$T(n) = 7 * T(n/2) + O(n^2); \qquad (3)$$

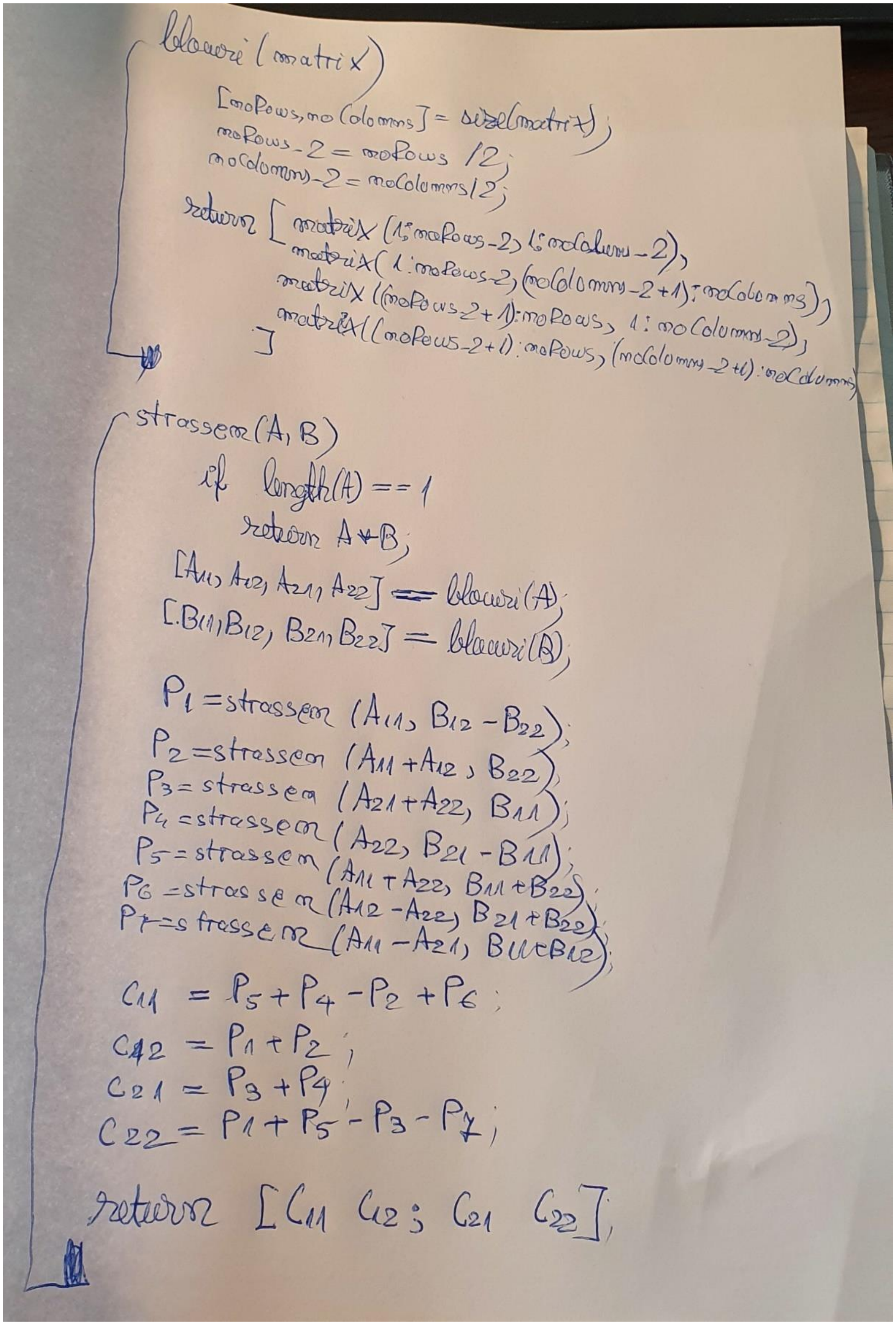
Analizand relatia de recurenta (2), obtinem: a=7, b=2, iar f(n) = O(n^2)

Reiese usor ca ne aflam in cazul I al teoremei Master:

O(n^2) ∈ O($n^{\log_2 7}$) ; intuim acest lucru intrucat $n^2 < n^{\log_2 7}$ ($2 < \log_2 7$)

O(n^2) ∈ O($n^{\log_2 7}$), lucru adevarat pentru ca orice f(n) ∈ O(n^2) apartine si multimii O($n^{\log_2 7}$), in care O(n^2) este inclusa.

Iata pseudocodul pentru algoritmul lui Strassen de inmultire a matricelor:



```
blocuri(matrix)
    [noRows, noColomns] = size(matrix);
    noRows_2 = noRows /2;
    noColomns_2 = noColomns/2;
    return [ matrix(1:noRows_2, 1:noColomns_2),
             matrix(1:noRows_2, (noColomns_2+1):noColomns),
             matrix((noRows_2+1):noRows, 1:noColomns_2),
             matrix((noRows_2+1):noRows, (noColomns_2+1):noColomns)
           ]

strassen(A, B)
    if length(A) == 1
        return A*B;
    [A11, A12, A21, A22] = blocuri(A);
    [B11, B12, B21, B22] = blocuri(B);

    P1 = strassen (A11, B12 - B22);
    P2 = strassen (A11 + A12, B22);
    P3 = strassen (A21 + A22, B11);
    P4 = strassen (A22, B21 - B11);
    P5 = strassen (A11 + A22, B11 + B22);
    P6 = strassen (A12 - A22, B21 + B22);
    P7 = strassen (A11 - A21, B11 + B12);

    C11 = P5 + P4 - P2 + P6;
    C12 = P1 + P2;
    C21 = P3 + P4;
    C22 = P1 + P5 - P3 - P7;

    return [C11 C12; C21 C22];
```

Fig. 1 Pseudocodul pentru algoritmul lui Strassen de inmultire a matricelor.

Un exemplu de aplicare a algoritmului Strassen de la care se poate generaliza foarte usor este cel din fig. 2:

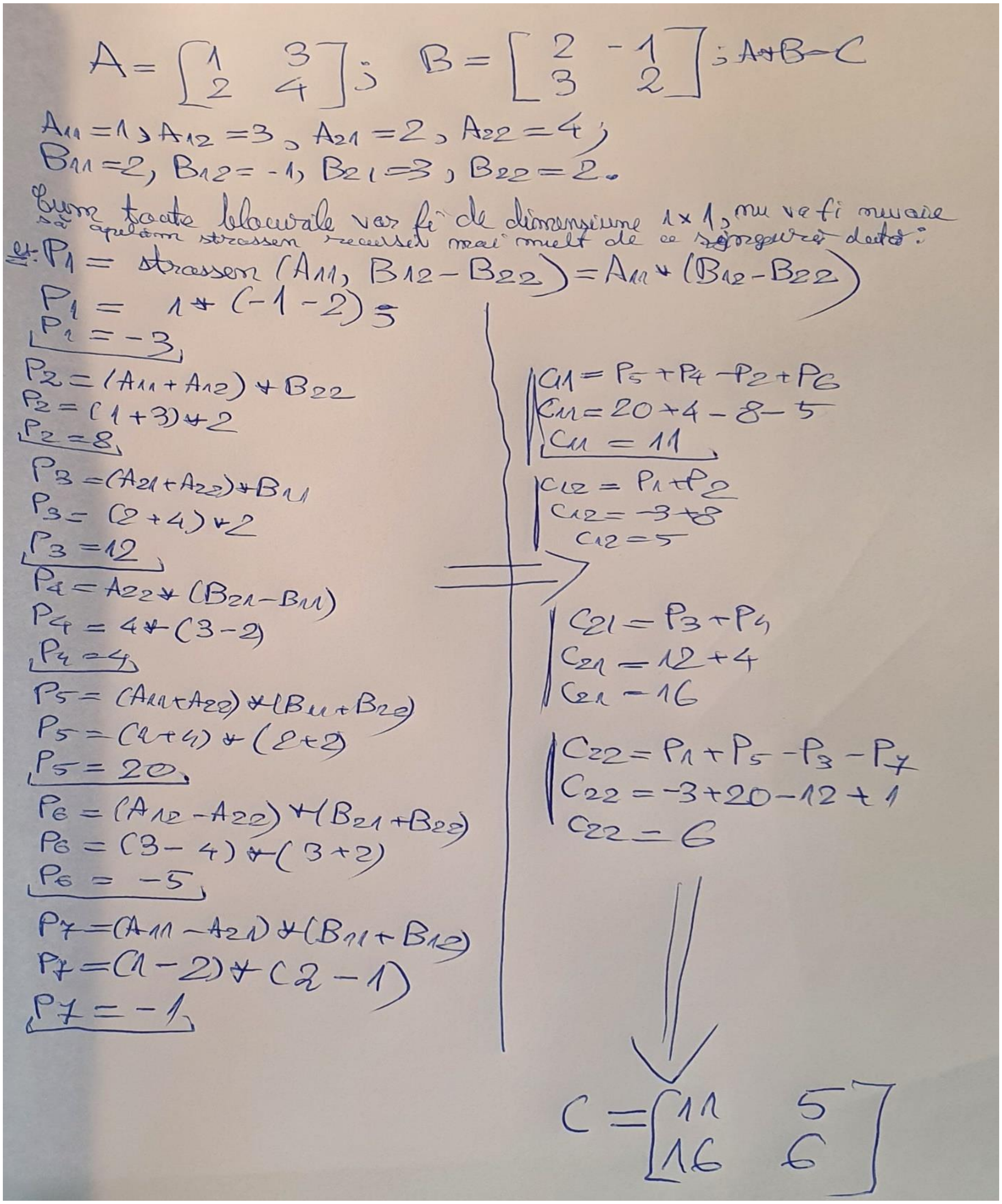


Fig.2 Exemplu de aplicare al algoritmului Strassen pe 2 matrice de dimensiune 2x2.

Ce merita mentionat este ca nu vorbim de cel mai eficient algoritm de inmultire a matricelor, in sensul ca pentru matrice nu de dimensiune foarte mare (n < 100), algoritmul naiv poate avea performante mai bune in ceea ce priveste timpul de executie. [2]

Acest lucru vine de la constantele ce sunt utilizate in formula de recurenta ce reiese din algoritmul lui Strassen (fiecare suma si diferenta intre blocuri adauga cate un O(n^2)).

Stiva de apeluri devine destul de mare pentru matrice de dimensiuni mari, astfel incat vom consuma foarte multa memorie.

Erorile sunt mai mari in cazul algoritmului lui Strassen decat in cazul algoritmului naiv din cauza preciziei limitate a calculului pe elemente ce apartin $\mathbb{R} \setminus \mathbb{Z}$. [1]

Un algoritm foarte eficient in problema multiplicarii matricelor este Coppersmith Winograd, a carui complexitate este O($n^{2.375477}$). [3]

2) Greedy

Problema montarii rafturilor

Enunt: Se da un perete de lungime w si doua lungimi de rafturi, m si n. Gaseste numarul de rafturi de fiecare tip ce ar trebui folosite pentru a minimiza spatiul ramas liber pe perete, cat si lungimea spatiului ramas liber. Raftul mai mare este mai ieftin, deci acesta este preferat. In orice caz, prioritatea este de a minimiza spatiul ramas liber pe perete, iar costul rafturilor este pe plan secund.

Descrierea solutiei: La fiecare iteratie vom incerca sa adaugam un raft mai lung pana cand lungimea rafturilor lungi depaseste lungimea peretelui. In cazul iteratiilor in care obtinem un spatiu mai mic liber in urma adaugarii noului raft lung, retinem numarul de rafturi lungi adaugate pana la acel pas, numarul de rafturi scurte necesare completarii peretelui si spatiul care ramane liber.

Pseudocodul problemei prezentate este ilustrat in fig. 3:

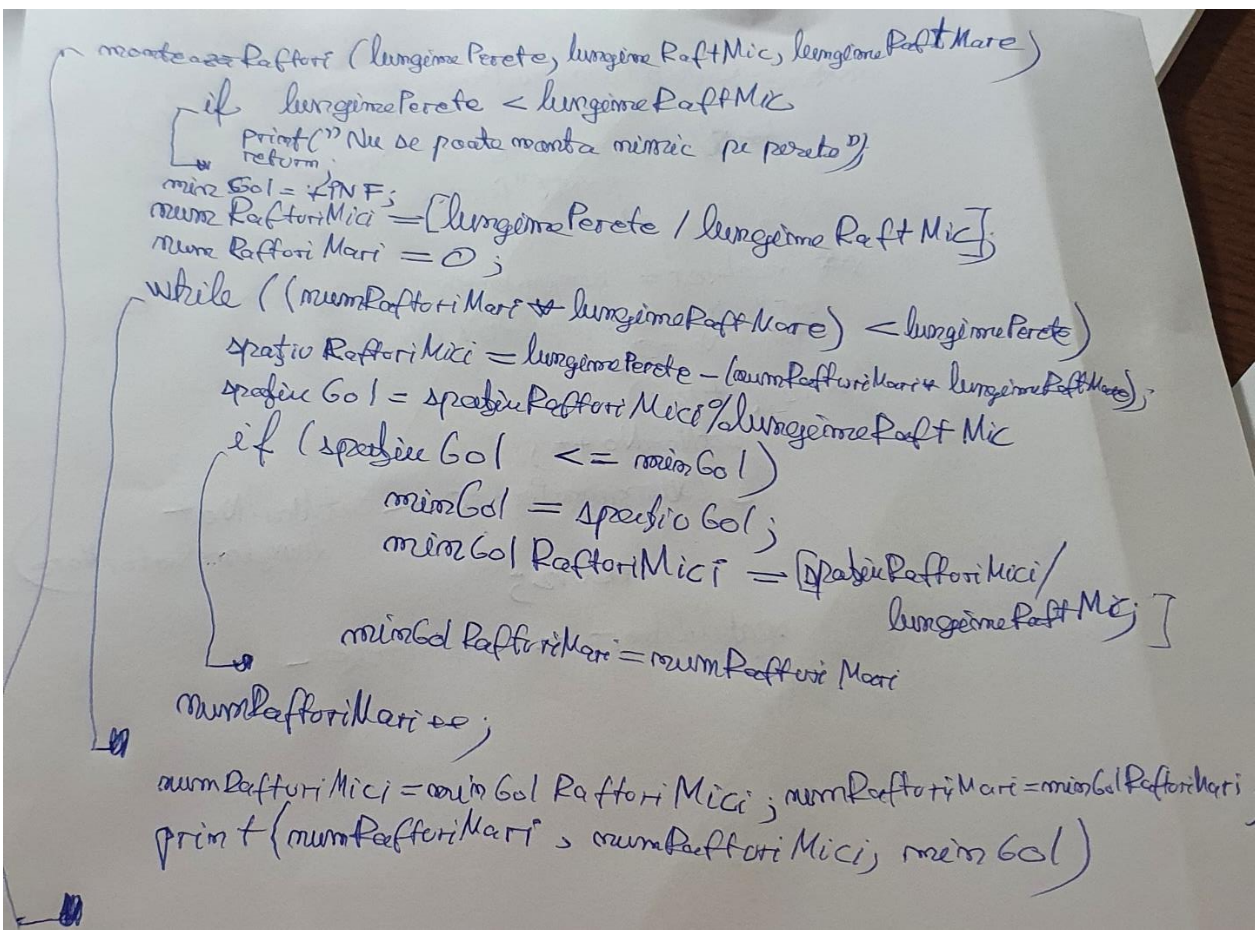


Fig. 3 Pseudocodul pentru problema montarii rafturilor.

Analiza complexitatii:

Este lesne de remarcat ca in afara buclei while avem doar operatii constante, deci complexitate $\theta(1)$. In cadrul buclei while sunt realizate tot operatii constante, deci din nou $\theta(1)$ de [lungimePerete/lungimeRaftMare] + 1 ori (asa cum se poate observa si in Fig. 4), motiv pentru care iese imediat o complexitate totala deci θ([lungimePerete/lungimeRaftMare]).

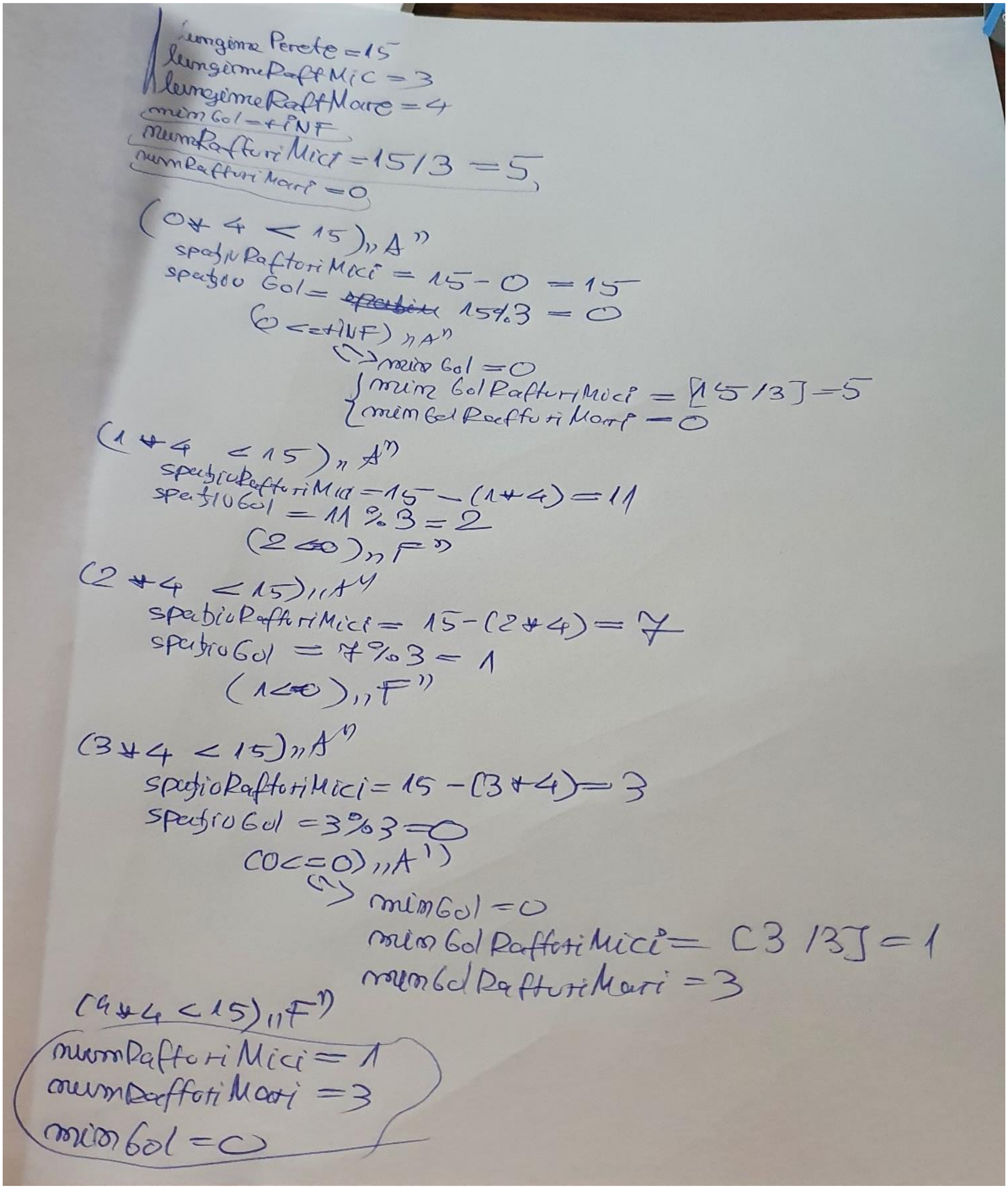
Posibilitatea de obtinere a optimului global:

Proprietatea de alegere de tip Greedy: Sa presupunem ca optimul global este diferit de cel obtinut in urma alegerilor succesive de optim local. Asta inseamna ca exista o combinatie de rafturi care sunt puse in asa fel incat se obtine un spatiul liber pe perete mai mic decat in urma solutiei generate prin algoritmul de tip Greedy. Acest lucru este fals, intrucat in cadrul algoritmului sunt verificate pentru fiecare numar de rafturi mari fixat la fiecare iteratie, numarul maxim de rafturi mici ce mai pot fi adaugate pentru a minimiza spatiul ramas liber. Deci nu exista o combinatie mai buna decat cea rezultata in urma algoritmului de tip Greedy.

Proprietatea de substructura optimala: Avand in vedere ca la fiecare pas verificam daca noul spatiu ramas neocupat pe perete este mai mic sau egal decat spatiul minim ramas neocupat de pana atunci, obtinerea noului optim local este direct infleuntata de vechiul optim local. Daca presupunem prin reducere la absurd ca la un pas nou obtinem o noua solutie optima (spatiu ramas liber) mai mare decat la pasul anterior, o sa ne aflam intr-o contradictie, intrucat acest lucru e imposibil, asa cum reiese si din linia „if (spatiuGol < minGol)" din figura 3.

Din cele 2 paragrafe prezentate mai sus, reiese ca algoritmul indeplineste cele doua proprietati pentru a fi catalogat drept Greedy.

Am creat un exemplu sugestiv pentru a ilustra modul de functionare al algoritmului. Iata-l in Fig. 4 de mai jos:



lungime Perete = 15
lungimeRaftMic = 3
lungimeRaftMare = 4
minGol = +INF
numRafturiMici = 15/3 = 5
numRafturiMari = 0

(0*4 < 15) "A"
spatiuRafturiMici = 15 - 0 = 15
spatiu Gol = ~~spatiu~~ 15%3 = 0
(0 <= +INF) "A"
=> minGol = 0
{ minGolRafturiMici = [15/3] = 5
{ minGolRafturiMari = 0

(1*4 < 15) "A"
spatiuRafturiMici = 15 - (1*4) = 11
spatiuGol = 11 % 3 = 2
(2 <= 0) "F"

(2*4 < 15) "A"
spatiuRafturiMici = 15 - (2*4) = 7
spatiuGol = 7%3 = 1
(1 <= 0) "F"

(3*4 < 15) "A"
spatiuRafturiMici = 15 - (3*4) = 3
spatiuGol = 3%3 = 0
(0 <= 0) "A"
=> minGol = 0
minGolRafturiMici = [3/3] = 1
minGolRafturiMari = 3

(4*4 < 15) "F"

numRafturiMici = 1
numRafturiMari = 3
minGol = 0

Fig. 4 Exemplu de aplicare al algoritmului Greedy pentru rezolvarea problemei montarii ratfturilor.

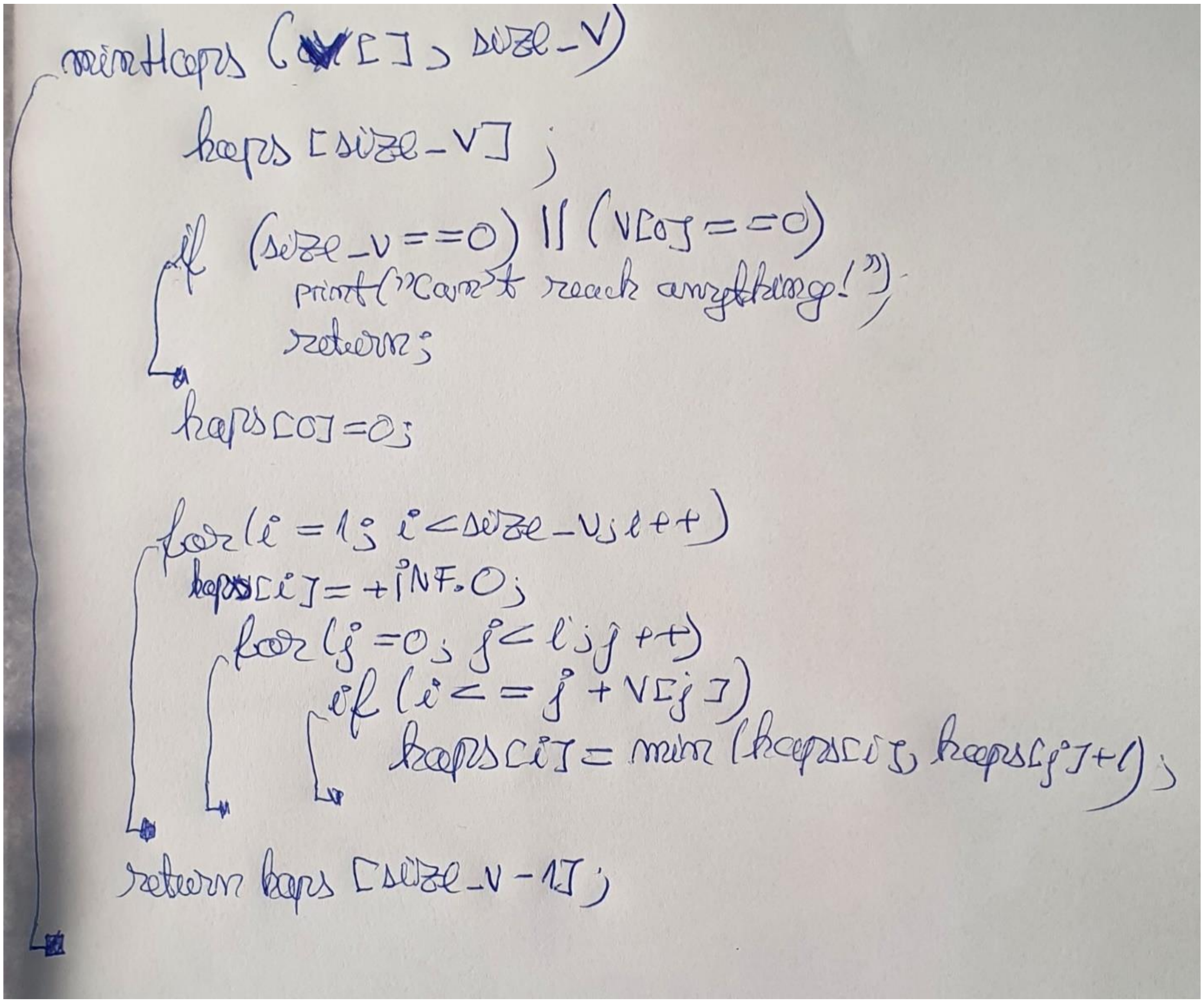
3) Programare Dinamica

Problema hopurilor minime

Enunt: Se da un vector cu n numere intregi in care fiecare element reprezinta numarul maxim de hopuri ce pot fi realizate inainte de la acel element. Scrieti o functie care sa returneze numarul minim de hopuri pentru a ajunge la finalul vectorului, pornind de la primul element.

Obs: Daca un element este 0, nu se poate realiza deplasarea prin acesta.

Descrierea solutiei: Cream un vector de hopuri, unde fiecare element, hopuri[i], reprezinta numarul minim de hopuri ce se pot realiza pentru a ajunge de la inceputul vectorului dat ca input pana la vector[i]. Vom trece pe rand prin fiecare element din vectorul de hopuri si vom verifica functie de elementele anterioare pozitiei sale, daca este accesibil din ele. Se va considera ca elementul este accesibil daca exista un alt element anterior de la care se pot realiza un numar maxim de hopuri corespondent valorii sale din vectorul de input pentru a ajunge la elementul curent.



```
minHops (v[], size_v)
    hops[size_v];
    if (size_v == 0) || (v[0] == 0)
        print("Can't reach anything!");
        return;
    hops[0] = 0;
    for(i = 1; i < size_v; i++)
        hops[i] = +INF.0;
        for(j = 0; j < i; j++)
            if (i <= j + v[j])
                hops[i] = min(hops[i], hops[j]+1);
    return hops[size_v - 1];
```

Fig. 5 Pseudocod pentru problema hopurilor minime.

Complexitate: Dupa cum se poate observa in algoritmul prezentat in Fig. 5, avem 2 foruri imbricate**.** Pentru fiecare i, avem i * (operatii constante) care se executa. Ignoram operatiile constante si ramanem, doar cu i care ia valori de la 1 la n. Facand o simpla suma 1+2+..+n, obtinem o complexitate polinomiala de ordinul n^2. Deci, in cadrul acestui algoritm *complexitatea temporala* este $\theta(n^2)$.

Complexitatea spatiala este data de vectorul auxiliar de hopuri, de dimensiune egala cu vectorul de input, deci este $\theta(n)$.

Despre relatia de recurenta: Pentru acest algoritm, vom folosi formula generala hops[i] = min(hops[i], hops[j] + 1), unde j = range(0:i). Daca gasim un numar de hopuri realizate pentru a ajunge la elementul j din vectorul de input care adaugat cu 1 este mai mic decat hopurile realizate pana la elementul i pana in momentul curent, actualizam aferent (bineinteles daca elementul i este accesibil din elementul j)

Mai jos, in Fig. 6 si 7 atasez 2 poze cu aplicarea algoritmului pe un input pentru a intelege modul de functionare al acestuia:

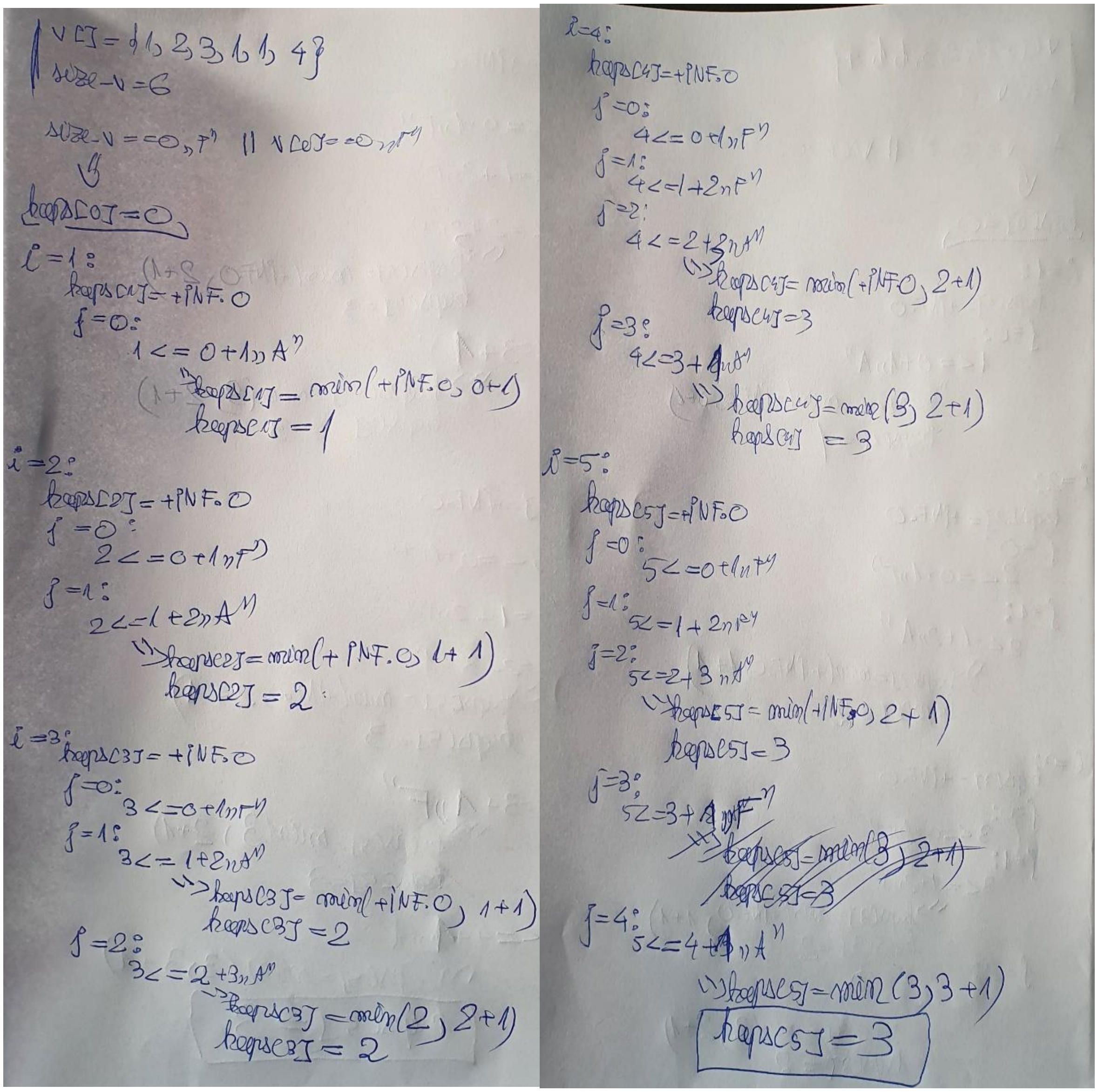


Fig. 6 (stanga) si 7 (dreapta) Exemplificarea aplicarii algoritmului hopurilor minime.

4) Tehnica Backtracking

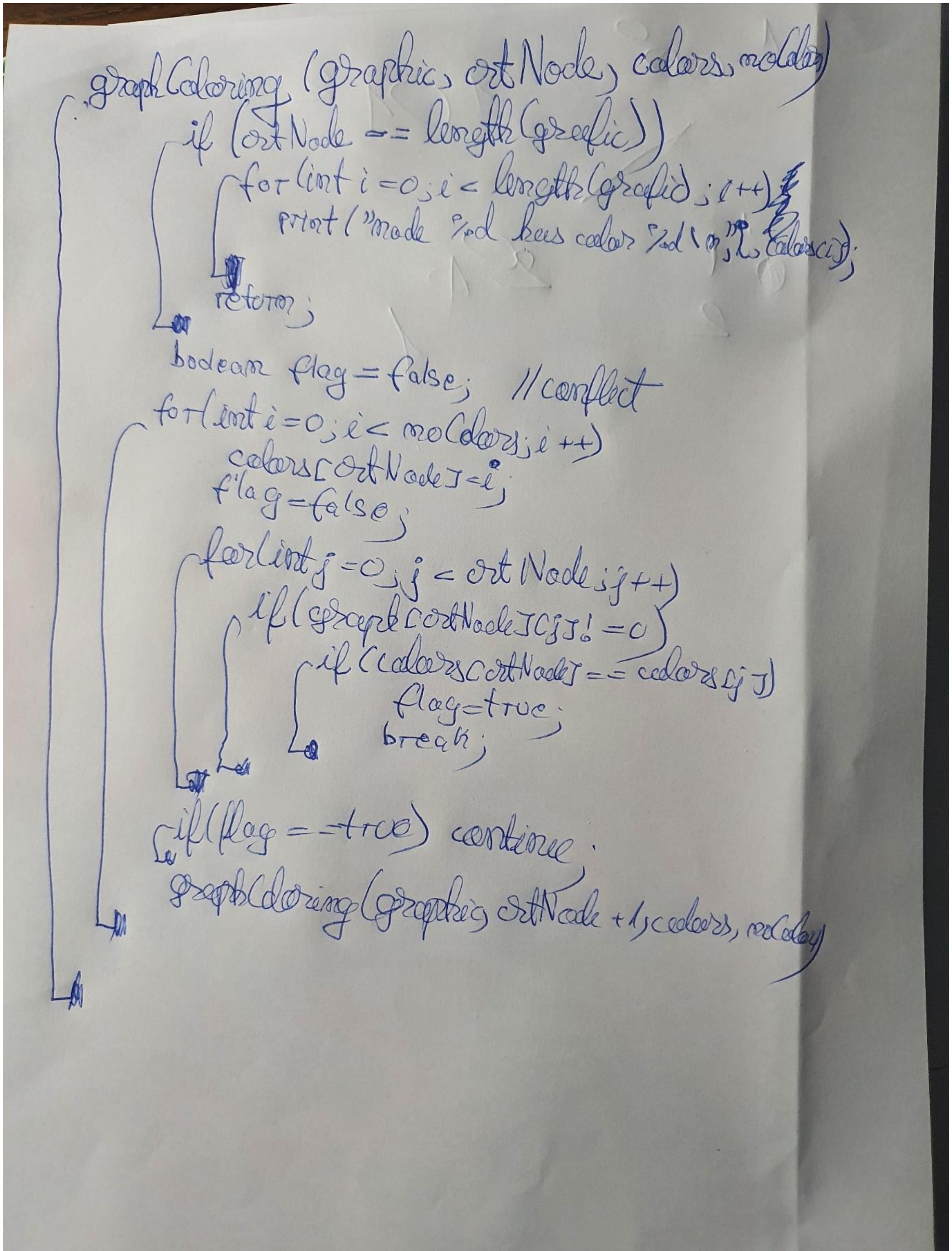
Problema colorarii grafurilor

Enunt: Sa se genereze toate combinatiile de culori posibile pentru nodurile unui graf neorientat, astfel incat oricare doua noduri adiacente sa nu aiba aceeasi culoare. Se dau m culori disponibile pentru colorare.

Rezolvarea problemei: Vom incerca pe rand toate atribuirile valide de culori pentru nodurile din graf. Dandu-se graful reprezentat ca o matrice, pentru nodul curent, crtNode sa-i spunem, ii vom atribui o culoare pe rand, si vom verifica pentru fiecare culoare linia corespunzatoare sa din matrice cu toate nodurile adiacente pentru verificarea atribuirii si stabilirea daca aceasta este valida. In caz ca atribuirea este valida, putem trece la nodul urmator (randul urmator din matrice) si verifica acelasi lucru pana la solutie, altfel se incearca o noua atribuire pana cand se ajunge sa nu mai avem atribuiri posibile.

Am atasat o poza cu pseudocodul pentru problema colorarii grafurilor, vezi Figura 8.

Posibile optimizari: In ceea ce priveste optimizarile, se poate observa si din fig. 8 ca nu voi verifica pentru nodul curent, decat nodurile carora le-au fost deja atribuite culori. Nu are sens sa verificam si nodurile care nu au fost colorate inca, ar fi ineficient. Mai mult decat atat, putem sa distribuim procesul de colorare pe diverse procesoare, spre exemplu daca m=8, iar |V| = 4, in ipoteza in care avem la dispozitie 4 procesoare, putem sa incepem colorarea primului nod pe primul procesor cu culorile 1 si, ulterior 5, pe al doilea cu culorile 2 si, ulterior 6, pe al treilea culorile 3 si, ulterior 7, respectiv pe al patrulea culorile 4 si, ulterior 8. Asa cum iese si din [6], se poate aplica BFS, pentru a reduce complexitatea temporala la O(|V| +|E|).



```
graphColoring (graphic, ortNode, colors, noColors)
  if (ortNode == length(grafic))
    for(int i=0; i< length(grafic); i++)
      print("node %d has color %d\n", i, colors[i]);
    return;
  boolean flag=false; //conflict
  for(int i=0; i< noColors; i++)
    colors[ortNode]=i;
    flag=false;
    for(int j=0; j< ortNode; j++)
      if(graph[ortNode][j]!=0)
        if((colors[ortNode]==colors[j])
          flag=true;
          break;
    if(flag==true) continue;
    graphColoring(graphic, ortNode+1, colors, noColors)
```

Fig. 8 Pseudocodul pentru algoritmul de rezolvare a problemei colorarii grafurilor.

Aprecierea complexitatii:

O($m^{|V|}$), pentru complexitatea temporala, intrucat pentru fiecare nod in parte, incercam sa verificam daca exista vreo solutie cu orice culoare.

O(|V|) pentru complexitatea spatiala, din cauza ca pe stiva de executie se vor afla maxim |V| + 1 apeluri recursive

Exemplu sugestiv:

Am trecut prin pseudocodul prezentat mai sus cu un exemplu pentru care am ales dimensiunea grafului egala cu 4, iar numarul de culori egale cu 3. Am preferat sa fac o reprezentare grafica, intrucat este mai usor de urmarit. Iata rezolvarea:

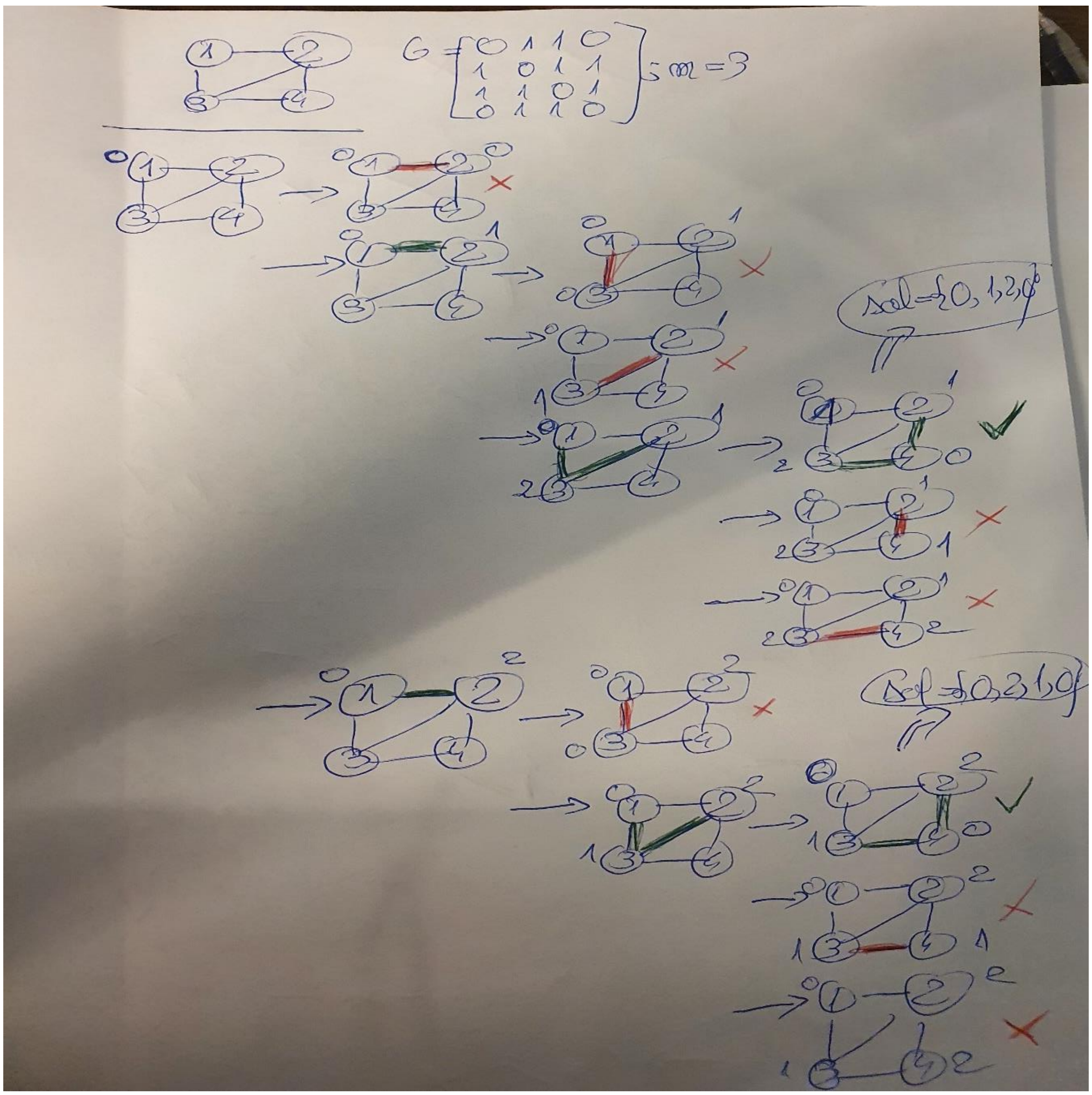


Fig. 9 Aplicare a algoritmului de m-colorare. (1)

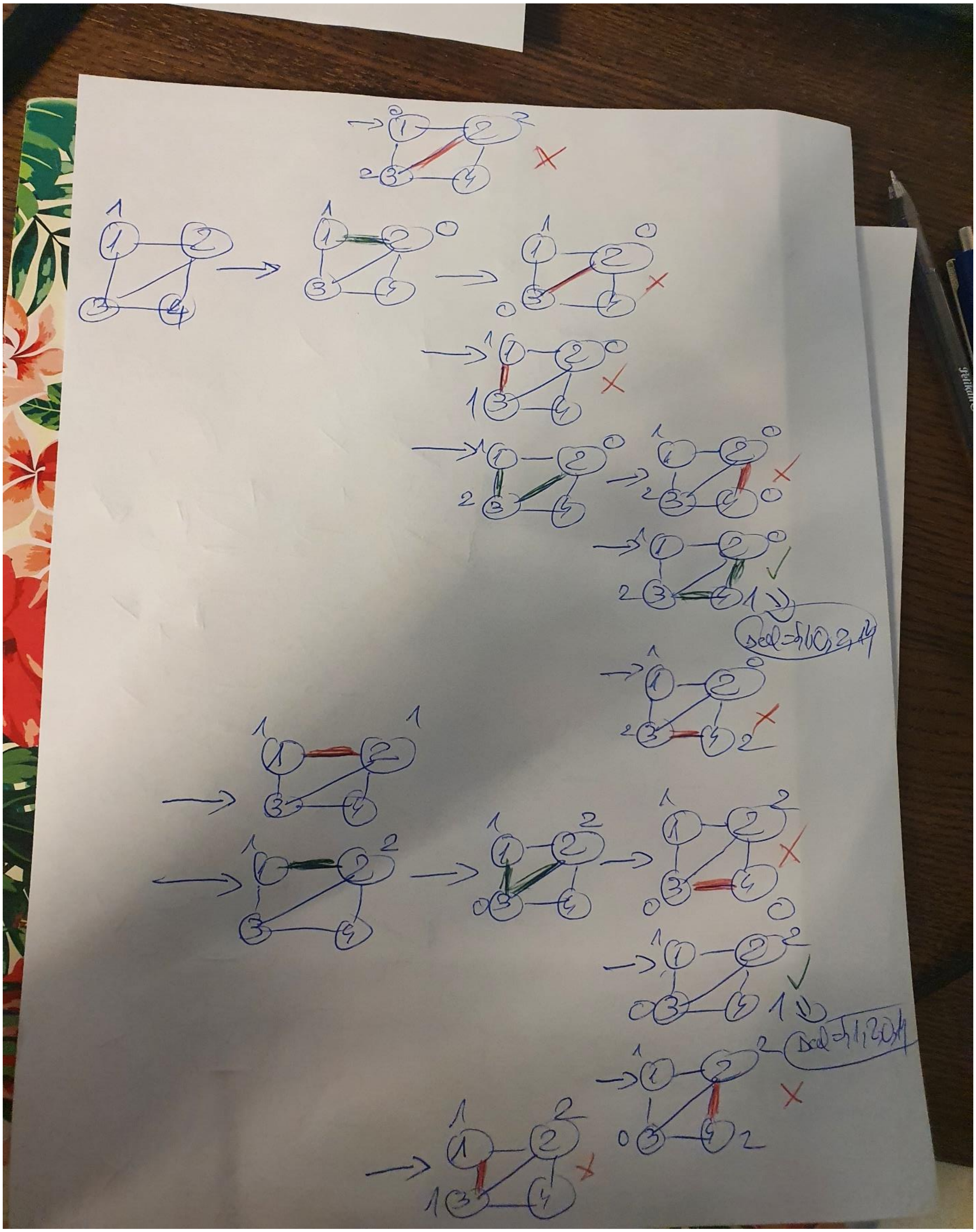


Fig. 10 Aplicare a algoritmului de m-colorare. (2)

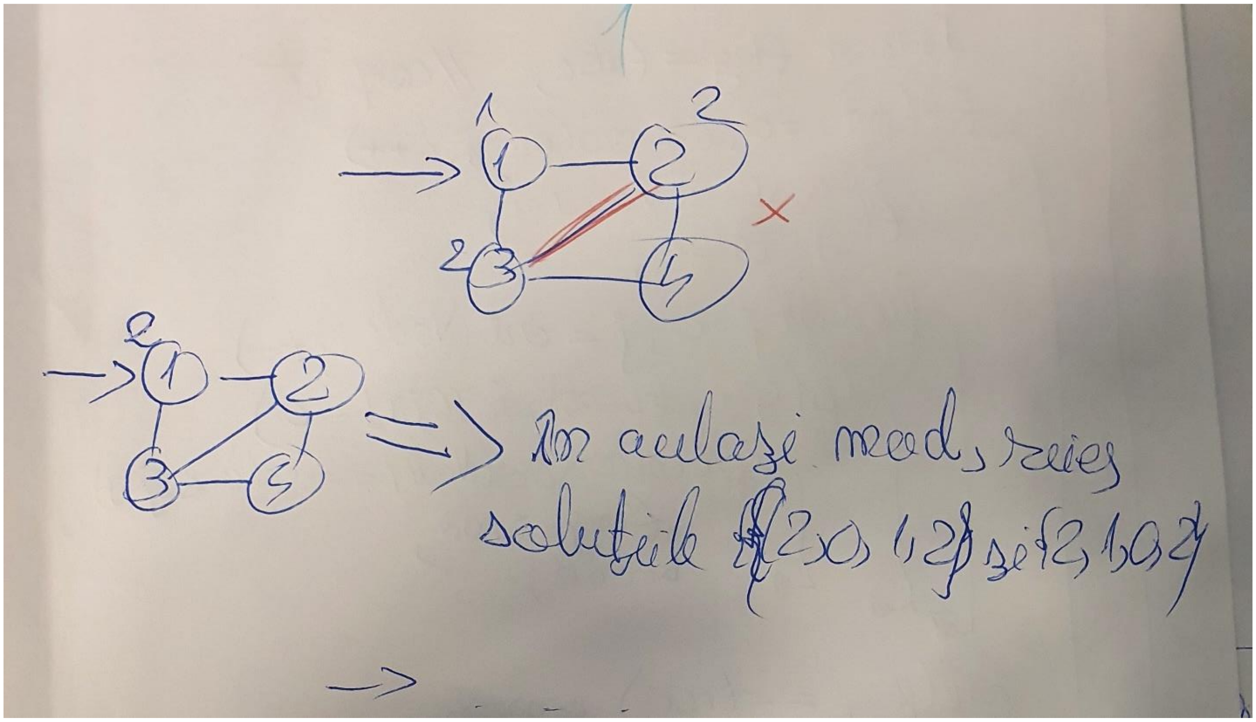


Fig.11 Aplicare a algoritmului de m-colorare. (3)

Comparatie intre cele patru tehnici de programare

	Divide et Impera	Greedy	Programare Dinamic	Backtracking
Tip problema comuna rezolvata	Problema ce poate fi impartita in mai multe subprobleme de aceeasi natura, care pot fi rezolvate recursiv; Solutiile subproblemelor trebuie sa poata fi combinate pentru a determina rezultatul final al problemei initiale.	Probleme ce au proprietatea **alegerii de tip Greedy** (folosind solutia optima locala, se ajunge la solutia de optim global), dar si proprieteatea de **substructura optima** (solutie optima a problemei initiale contine solutii optime ale subproblemelor)	Probleme ce pot fi descompuse in subprobleme ce necesita aceleasi calcule, astfel rezultatele obtinute anterior putand fi reutilizate.	Probleme de optimizare, dar si de enumerare (daca scopul final este de a alege intre mai multe raspunsuri posibile).
Avantaje	De obicei, genereaza algoritmi eficienti; Faciliteaza paralelismul, astfel diverse procesoare pot efectua calculele necesare diferitelor subprobleme. Acces rapid la memorie, se foloseste memoria cache.	Relativ usor de implementat de cele mai multe ori; Chiar daca nu genereaza cea mai buna solutie, poate fi generata o solutie satisfacatoare.	Scade semnificativ timpul de executie, evitand recalcularea unor solutii. Spre deosebire de un algoritm de tip Greedy, aici vom obtine solutia optima	Rezultatul/Rezultatele final(e) este/sunt cu certitudine cel(e) optim(e); Permite optimizari de tipul paralelizarii (diferite pozitii de start ale problemei pe diverse procesoare).
Dezavantaje	Apelurile recursive determina o stiva de executie foarte mare a programului; Se intampla des ca diverse subprobleme sa apara de mai multe ori in cadrul aceleiasi probleme, astfel ar trebui folosita o forma de memorizare.	Este adeseea foarte dificil de demonstrat ca algoritmul folosit este si corect.	Memorie suplimentara de multe ori proportionala cu marimea datelor de intrare. Adesea, multe valori calculate anterior nu sunt utilizate si folosesc spatiul de stocare inutil.	Complexite spatiala foarte mare, intrucat folosim stiva de executie a programului, iar apelurile recursive ajung sa fie in numar mare; Ineficient, intrucat verifica toate situatiile posibile (adesea, exista algoritmi mai eficienti pentru a rezolva problema data)

Exemple de probleme pentru care pot fi aplicate mai multe tehnici de programare:

Pentru problema gasirii submultimii de suma maxima, analizand intre [7] si [8], observam o diferenta de complexitate temporala intre algoritmi si anume θ(nlogn) pentru Divide et impera, respectiv θ(n) pentru programare dinamica. Este usor de remarcat ca algoritmul prezentat in [7] foloseste mai mult spatiu, din cauza apelurilor recursive. La fel si ca pot exista situatii in care se va calcula o suma de mai multe ori din aceleasi numere aflate si in alte pozitii ale vectorului.

Pentru bine cunoscuta problema a colorarii grafurilor, putem folosi un algoritm Greedy de aproximare, cum este prezentat in [9], dar putem verifica daca este posibila o astfel de colorare prin backtracking, cum reiese din [10]. Se observa o complexitate temporala de O(V^2+E) in cazul algoritmului Greedy si una de O($m^{|V|}$) in cazul folosirii tehncii de backtracking, unde m reprezinta numarul de culori. Asa cum am prezentat si in tabela de mai sus, algoritmul de tip Greedy este mult mai usor de implementat, cat si mai eficient, insa nu asigura ca se va ajunge la o solutie corecta, pe cand cel de Backtracking, desi mai lent, va returna solutia/solutiile optima/optime.

Referinte Divide Et Impera:

[1]: https://www.geeksforgeeks.org/strassens-matrix-multiplication/

[2]: https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344

[3]: https://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm

Referinte Greedy:

[4]: https://www.geeksforgeeks.org/fitting-shelves-problem/

Referinte:Programare Dinamica:

[5]: https://leetcode.com/problems/jump-game-ii

Referinte Backtracking:

[6]: https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/

Referinte comparatii algoritmi:

[7]: https://www.geeksforgeeks.org/maximum-subarray-sum-using-divide-and-conquer-algorithm/

[8]: https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

[9}: https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/

[10]: https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/