Proiectarea Algoritmilor

Tema 0

Facultatea de Automatică şi Calculatoare
Universitatea Politehnică Bucureşti

Duţu Alin Călin

23 aprilie 2021

Cuprins

1 Divide et Impera **4**

1.1 Enunţ 4
1.2 Descrierea soluţiei problemei 4
1.3 Prezentarea algoritmului de rezolvare a problemei 5
1.4 Aprecierea complexităţii algoritmului propus 6
1.5 Analiza succintă asupra eficienţei algoritmului propus 7
1.6 Exemplificarea aplicării algoritmului propus pentru un exemplu sugestiv . . 7
1.6.1 Exemplu de input 7
1.6.2 Aplicarea algoritmului 8

2 Greedy **10**

2.1 Enunţ 10
2.2 Descrierea soluţiei problemei 10
2.3 Prezentarea algoritmului de rezolvare a problemei 11
2.4 Aprecierea complexităţii algoritmului propus 11
2.5 Analiză succintă asupra posibilităţii de obţinere a optimului global 12
2.6 Exemplificarea aplicării algoritmului propus pentru un exemplu sugestiv . . 12
2.6.1 Exemplu de input 12
2.6.2 Aplicarea algoritmului 13

3 Programare Dinamică **15**

3.1 Enunţ 15
3.2 Descrierea soluţiei probemei 15
3.3 Prezentarea algoritmului de rezolvare a problemei 16
3.4 Aprecierea complexităţii algoritmului propus 17
3.5 Explicarea modului în care a fost obţinută relaţia de recurenţă 17
3.6 Exemplificarea aplicării algoritmului propus pentru un exemplu sugestiv . . 17
3.6.1 Exemplu de input 17
3.6.2 Aplicarea algoritmului 19

4 Backtracking **20**
4.1 Enunţ 20
4.2 Descrierea soluţiei probemei 20
4.3 Prezentarea algoritmului de rezolvare a problemei 21
4.4 Aprecierea complexităţii algoritmului propus 22
4.5 Analiză succintă asupra eficienţei algoritmului propus 23
4.6 Exemplificarea aplicarii algoritmului propus pentru un exemplu sugestiv . . 23
4.6.1 Exemplu de input 23
4.6.2 Aplicarea algoritmului 24

5 Analiză comparativă **26**
5.1 Divide et Impera 26
5.2 Greedy 26
5.3 Programare dinamică 27
5.4 Backtracking 27

Bibliografie **27**

Capitolul 1

Divide et Impera

1.1 Enunţ

Se dă un pachet de 52 de cărţi de joc. Pachetul nu conţine Jokeri. Se citesc din fisierul "Pachet carti.in" cele 4 tipuri de cărţi într-o ordine aleatoare urmată de numărul şi tip-ul unei cărţi din pachet. Toate cărţile vor fi aranjate în ordine crescătoare şi în ordinea tipurilor aşa cum este dată din fisierul de intrare. As-ul va fi considerat numărul 11. Să se găsească poziţia cărţii citite de la input.

Pentru această problemă avem următoarele notaţii: ir - inimă roşie; in - inimă neagră; rb - romb; tr - treflă.

1.2 Descrierea soluţiei problemei

Problema dată se poate rezolva folosind un algortim de căutare. Având avantajul că toate cărţile de un tip sunt sortate putem apela la o căutare binară, însă tipurile nu sunt sortate si nici nu pot fi sortate în vreun fel pentru a putea aplica căutarea binară peste tot pachetul, aşa că se va recurge la un compromis.

La începutul programului se vor citi datele de intrare şi se va forma un vector de structuri. Pentru această metodă se va folosi structura carte ce conţine un int pentru numărul cărţii şi un string pentru tipul cărţii. După formarea vectorului se va apela

funcţia Find_Card care va căuta într-o primă etapă tipul cărţii şi apoi va căuta după număr. Funcţia primeşte vectorul, dar şi indicii de început şi sfârşit ai vectorului, verifică mijlocul vectorului şi împarte problema în două sau împarte pachetul în două părţi cu caracteristici diferite. Odată cu găsirea tipului, algoritmul ajunge la eficienţa cea mai bună întrucât această bucată de vector este ordonată crescător. Aplicându-se algoritmul de căutare binară ajungem rapid la cartea căutată şi în final se va afişa poziţia găsită.

1.3 Prezentarea algoritmului de rezolvare a problemei

```
structura{
        numar;
        tip;
} carte

int Find_card(pachet, carte, start, end)
        daca start > finish
                daca pachet[start] == carte
                        returneaza start;

                altfel returneaza -1;

        mid = (start + finish) / 2;

        daca pachet[mid] == carte
                returneaza mid;

        daca tipul lui carte == tip pachet[start] si tip pachet[finish]
                daca numar carte < numar pachet[mid]
                        returneaza Find_card(pachet, carte, start, mid-1);

                returneaza Find_card(pachet, carte, mid + 1, finish);

        left = Find_card(pachet, carte, start, mid - 1);
        right = Find_card(pachet, carte, mid + 1, finish);
```

```
        daca left diferit de -1
                returneaza left;

        returneaza right;


int main()
        citeste n
        pentru i de la 1 la n
                citeste type

                pentru j de la 0 la 13
                        pachet[j + i * 13] = carte(j + 2, type);

        citeste searched;
        scrie Find_Card(pachet, searched, 0 , 51) + 1;
        returneaza 0;
```

1.4 Aprecierea complexităţii algoritmului propus

Complexitatea algoritmului dat pentru un singur pachet de 52 de cărţi este O(1) deoarece avem un număr fix de cărţi, deci un număr fix de tipuri de cărţi, astfel vom avea un număr finit de intrucţiuni executate. Pentru mai multe pachete de cărţi puse toate într-un singur loc algoritmul va avea complexitatea O(n) deoarece când se intră în Find_cards întâi se va aplica căutarea binară pe tot pachetul până când se va ajunge la un set cu tipul respectiv, ceea ce va scoate o complexitate O(n), iar căutarea pe un singur set va scoate O(1) deoarece setul are un număr fix de cărţi. Dacă set-urile nu ar avea valori fixe algortimul ar scoate o complexitate O(log m) unde m este numărul de cărţi dintr-un set deoarece s-ar fac face căutare binară pe un şir crescător. Deci, complexitatea în cel mai rău caz este O(n).

1.5 Analiza succintă asupra eficienţei algoritmului propus

Acest algoritm, desi este eficient din punct de vedere al timpului de execuţie, complexitatea spaţială ar fi O(n) deoarece s-ar genera câte două apeluri de funcţii pentru fiecare tip de carte până la găsirea tipului cărţii. Modalitatea prin care s-ar putea scăpa de această complexitate este folosirea unei formule matematice, dar asta ar însemna să nu folosim metoda Divide et Impera. Deci, algoritmul folosit pentru rezolvarea acestei probleme este optim în măsura în care se cere în cerinţă rezolvarea doar cu Divide et Impera.

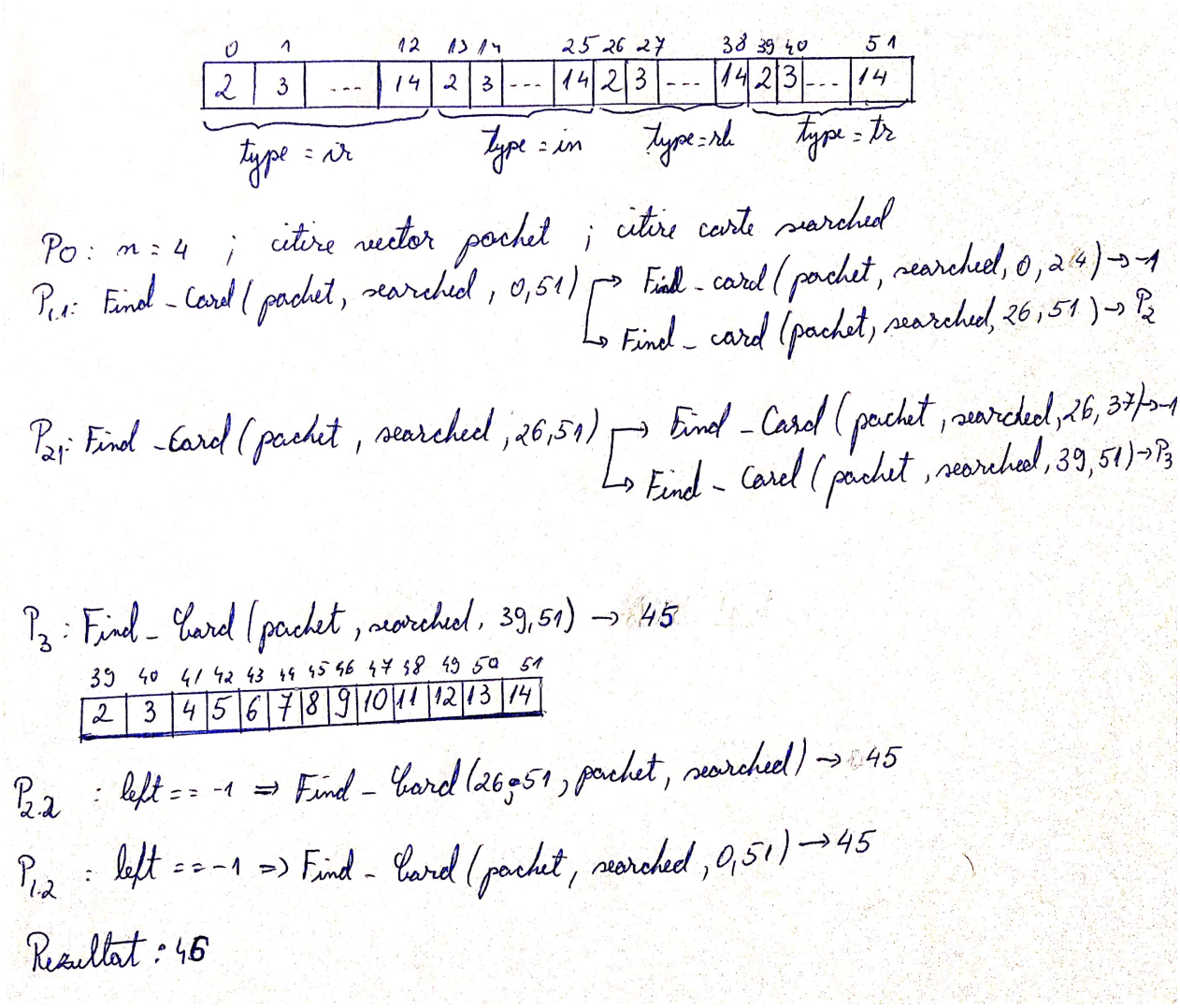
1.6 Exemplificarea aplicării algoritmului propus pentru un exemplu sugestiv

1.6.1 Exemplu de input

```
Pachet carti.in
4
ir
in
rb
tr
8 tr

Rezultatul: 46
```

1.6.2 Aplicarea algoritmului



Pasul 0: Se citeşte din fişierul de intrare n şi cele n tipuri de cărţi, se creează vectorul pachet şi se citesc datele cărţii căutate (8 de treflă). Se apelează funcţia Find_Card pe tot pachetul.

Pasul 1.1: Se verifică dacă mijlocul pachetului este cartea căutată şi în acest caz nu este. Tipul cărţii de la indicele start nu este treflă, deci se va apela de două ori Find_Card, odată pe cărţile de la poziţiile 0-24, apel care va returna -1 pentru că acea bucată nu conţine 8-ul de treflă si încă odata pe cărţile de la 26 la 37.

Pasul 2.1: Se verifică mijlocul pachetului şi observam ca din nou nu se potriveşte, se

mai observă că tipul cărţii de la indicele start nu este treflă, deci se apelează din nou Find_Card de 2 ori. Primul apel va returna -1 deoarece toate cărţile acoperite de funcţie, adică 26-37 sunt de romb. Al doilea apel va acoperi toate cărtile de treflă, adică 39-51.

Pasul 3: Se verifică mijlocul şi se constată o potrivire între cartea căutată şi cartea de la poziţia 45. Se va returna 45.

Pasul 2.2: Apelul din variabila left nu conţine soluţia, deci va returnat -1 care condiţionează programul să returneze 45.

Pasul 1.2: Partea stângă a pachetului nu conţine cartea căutată ceea ce înseamnă că partea dreaptă o conţine. Se returnează 45.

Se va scrie în output rezultatul funcţiei Find_Card + 1, adică 46 pentru că numărătoarea in program a început de la 0 şi în realitate ea începe de la 1.

Capitolul 2

Greedy

2.1 Enunţ

Se citesc 3 numere naturale S, n şi e cu următoarele semnificaţii: S este o sumă de bani care trebuie plătită folosind bancnote care au valori puterile lui e de la 1 la e^n. Să se afişeze la final numărul de bancnote folosite. Datele se vor citi din fişierul Euro.in.

Sursa problemei -> [1]

2.2 Descrierea soluţiei problemei

Pentru rezolvarea acestei probleme se poate opta pentru o variantă scurtă şi simplă. Se încearcă aflarea valorii maxime a unei bancnote printr-o iterare de la 0 până la puterea maximă. După aflarea valorii maxime va trebui să se aplice un concept Greedy foarte simplu pentru minimizarea numărului de bancnote. Se plăteşte mai întâi cât se poate cu bancnote de valoare maximă, apoi când nu se mai poate, se va plăti restul sumei cu bancnote a căror valoare e din ce în ce mai mică până când suma este plătită. Se poate afla foarte usor numărul de bancnote printr-o împărtire a sumei la valoarea maximă, numărul fiind dat de cât. Restul va fi plătit cu bancnote mai mici. După ce s-a efectuat plata se va afişa rezultatul.

2.3 Prezentarea algoritmului de rezolvare a problemei

```
int main()
        citire sum;
        citire max_pow;
        citire baza;

        max = 1;
        nr = 0;

        pentru i de la 0 la max_pow - 1
                daca(max * baza > sum)
                        inchide for-ul;

                max = max * baza;

        cat timp sum > 0
                nr = nr + sum / max;
                sum = sum % max;
                max = max / baza;

        scrie nr;
```

2.4 Aprecierea complexităţii algoritmului propus

Din punct de vedere temporal, programul conţine un for de la 0 la max_pow care va face în cel mai rău caz max_pow pasi, deci O(max_pow) şi un while care depinde de sumă. Numărul de iteraţii va depinde de puterea bazei la valoarea maximă pe care o poate avea o bancnotă, adică va depinde de numărul de tipuri de bancnote pe care le avem la dispoziţie, deci avem complexitate O($log_{baza}max$). Astfel complexitatea finală va fi O(max_pow + $log_{baza}max$.

2.5 Analiză succintă asupra posibilităţii de obţinere a optimului global

Algoritmul respectă proprietatea de substructură optimală deoarece soluţia subproblemei, adică numărul de bancnote pentru un tip ajută al rezolvarea problemei, adică numărul total de bancnote. De asemenea, algoritmul respectă si proprietatea de alegere de tip Greedy deoarece nu se calculează numărul bancnotelor pe baza valorilor anterior calculate.

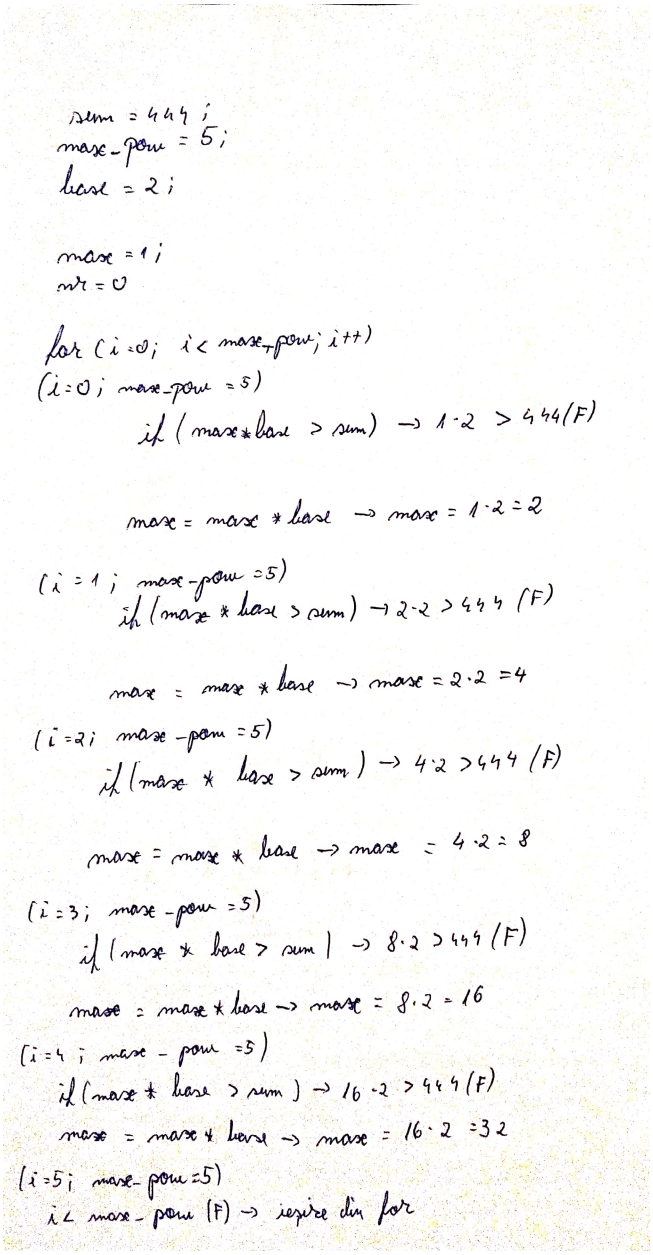
2.6 Exemplificarea aplicării algoritmului propus pentru un exemplu sugestiv

2.6.1 Exemplu de input

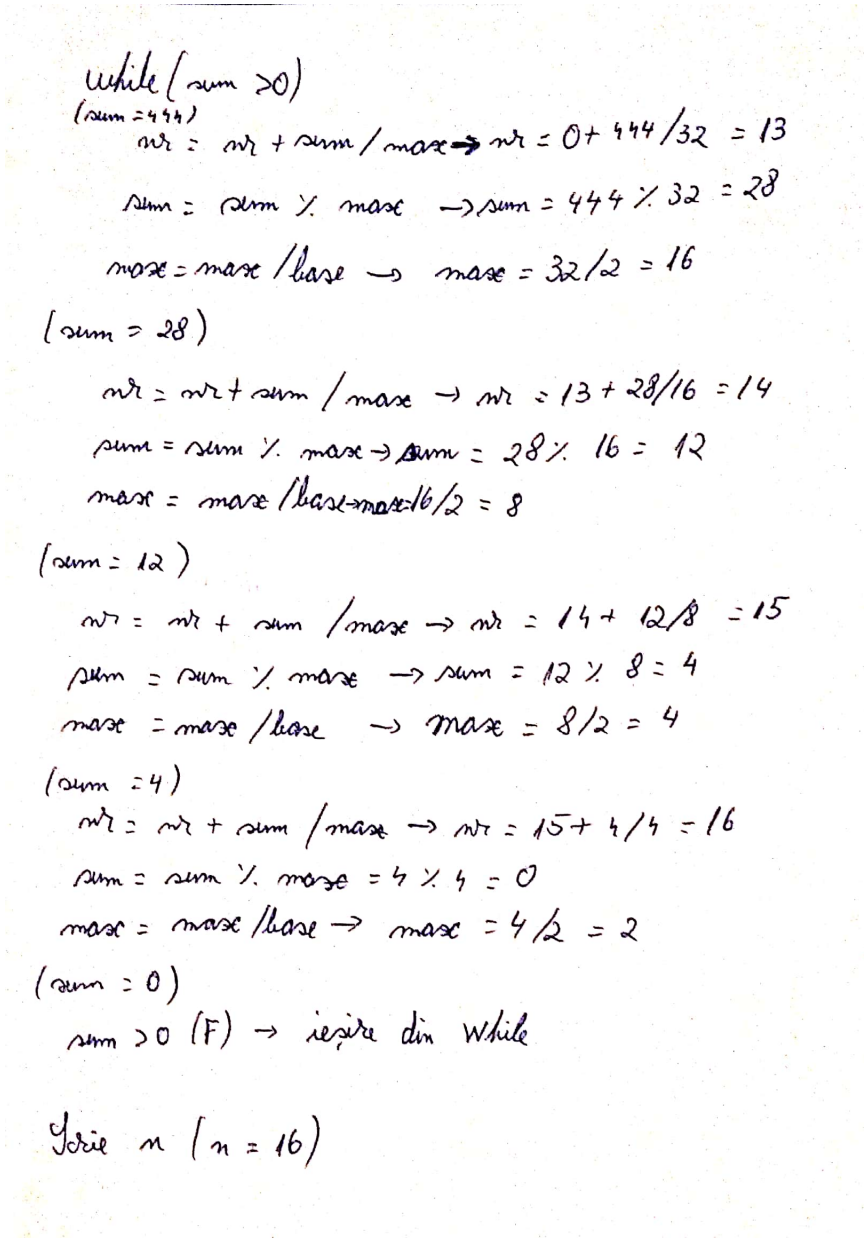
```
Euro.in
444 5 2

Rezultat: 16
```

2.6.2 Aplicarea algoritmului



Algoritmul începe prin citirea parametrilor şi intrarea în for care stabileşte valoarea maximă a unei bancnote în funcţie de puterea maximă şi în funcţie de sumă. Practic, fiecare iteraţie din for va verifica următoarea putere a bazei, astfel va ieşi puterea cea mai mare.



După găsirea maximului se va itera prin while-ul care va scoate numărul de bancnote. Aici se va împărţi suma la valoarea maximă şi va folosi: câtul ca să numere bancnotele folosite şi restul câtului pentru restul de plată. La următoarea iteraţie se va număra cât din restul rămas se poate plăti cu bancnote cu următoarea valoare maximă si tot aşa. În final, după ce se plăteşte toată suma, while-ul se încheie şi se afişează numărul de bancnote.

Capitolul 3

Programare Dinamică

3.1 Enunţ

Pentru a prepara o ciorbă bună pentru soţul ei, Rodica se duce la piaţă pentru a cumpăra ingredientele necesare. Ea are o listă "Ingrediente.in" cu numărul vânzătorilor de legume disponibili şi cu suma pe care o deţine pe prima linie. Pe următoarele n linii sunt ofertele fiecărui vânzător sub forma unei perechi de tip (kg, preţ). Rodica doreşte să afle folosind tehnica Programării Dinamice care este numărul maxim de kilograme pe care le poate cumpăra cu toţi banii pe care îi are.

3.2 Descrierea soluţiei probemei

Pentru a utiliza tehnica Programării dinamice, în această problemă, trebuie să stabilim dimensiunile tabelei. Astfel liniile vor fi reprezentate de vânzători deoarece cu aceste "obiecte" vom lucra, iar coloanele vor fi reprezentate de suma cheltuită pentru că avem la dispoziţie o sumă maximă, o restricţie. Cu acest tabel putem lua toate posibilităţile şi aşa vom găsi soluţia cea mai bună. Astfel, în fiecare câmp al tabelei vom verifica care din cele 3 cazuri va fi cel mai bun pentru acea situaţie. Primul caz este când soluţia precedentă este cea mai bună şi atunci vom prelua acea soluţie. Al doilea caz este când oferta vânzătorului este mai rentabilă decât soluţia precedentă, iar al 3-lea caz este o combinaţie între cele două. În final, soluţia se va regăsi pe câmpul de pe ultima linie şi ultima coloană.

3.3 Prezentarea algoritmului de rezolvare a problemei

```
structura{
        greutate;
        pret;
} Ingredient;

int main()
        citeste n;
        citeste S;

        pentru i de la 1 la n
                citeste vanzator[i].greutate;
                citeste vanzator[i].pret;

        pentru i de la 0 la n
                pentru j de la 0 la S
                        daca i = 0 sau j = 0
                                continua;

                        daca j < vanzator[i].pret
                                dp[i][j] = dp[i - 1][j];
                                continua;

                        daca dp[i - 1][j] > vanzator[i].greutate
                                dp[i][j] = dp[i - 1][j];

                        altfel dp[i][j] = vanzator[i].greutate;

                        daca(dp[i][j] < dp[i-1][j - vanzator[i].pret]
                                + vanzator[i].greutate)

                                dp[i][j] =  dp[i-1][j - vanzator[i].pret]
                                                + vanzator[i].greutate;

        scrie dp[n][S];
```

3.4 Aprecierea complexităţii algoritmului propus

Din punct de vedere al complexităţii temporale algoritmul va executa n * S paşi deoarece avem citirile care au O(1), avem citirea vectorului de vânzători care este intr-un for de la 1 la n, ceea ce inseamnă O(n), avem apoi iterarea prin tabela dinamică care conţine două for-uri, unul de la 0 la n şi unul de la 0 la S şi în interiorul lor instructiuni care au complexitate O(1), deci avem O(S*n). Astfel vom avea O(S * n + n) = O(n * (S + 1)) = O(n * S).

În program există variabile care au o complexitate spaţiala de O(1), un vector care conţine structuri cu 2 câmpuri care are complexitate O(2 * n) şi o tabelă de dimensiuni n+1 şi S+1 care va avea o complexitate spaţiala O((n + 1) * (S + 1)) = O(n * S). Apeluri recursive nu există în acest program, deci complexitatea spaţială rămâne O(n * 2) pentru S <= 2 şi O(n * S) pentru S > 2.

3.5 Explicarea modului în care a fost obţinută relaţia de recurenţă

În principiu, dacă la linia i avem vânzătorul cu oferta (kg, pret) vom avea pe această linie de la prima coloană până la coloana pret soluţia precedentă, adică dp[i - 1][nr. coloană]. Apoi avem de verificat care din cele 3 soluţii este cea mai bună. Se va compara soluţia de la linia precedentă cu oferta vânzătorului actual şi se va obtine o valoare maximă. Se verifică apoi dacă este mai rentabilă cumularea greutăţii ofertei actuale cu soluţia de la linia trecută dar scăzând din coloana actuală costul actualului vânzător.

3.6 Exemplificarea aplicării algoritmului propus pentru un exemplu sugestiv

3.6.1 Exemplu de input

`Ingrediente.in`

```
5 30
3 9
1 3
2 5
5 15
4 10

Rezultat: 11
```

3.6.2 Aplicarea algoritmului

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
2	0	0	0	1	1	1	1	1	1	3	3	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
3	0	0	0	1	1	2	2	2	3	3	3	3	4	4	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	0	0	0	1	1	2	2	2	3	3	3	3	4	4	5	5	5	6	6	6	7	7	7	8	8	8	8	9	9	10	10
5	0	0	0	1	1	2	2	2	3	3	4	4	4	5	5	6	6	6	7	7	7	7	8	8	9	9	9	10	10	10	11

Figura 3.1: Tabelul PD

După citirea şi iniţializarea tuturor vectorilor urmează construirea soluţiei folosind tabela PD. Ştiind că tabela este iniţializată la 0 nu avem nimic de modificat la linia şi coloana cu indicele 0.

Astfel, pe prima linie avem până la costul ofertei adică coloana 9 soluţia precedentă şi apoi vom avea doar 3 kilograme până la final deoarece nu sunt alte oferte care să o precede pe aceasta. La a doua linie, la fel, pâna la coloana 3 soluţia precedentă. De la coloana 3 la 8 oferta actuală este cea mai bună, deci se va trece 1 kilogram, de la 9 la 11 soluţia precedentă este cea mai bună, iar de la 12 până la final cumularea primelor două oferte va fi cea mai optimă opţiune.

Linia a 3-a va avea până la coloana a 4-a soluţia precedentă, coloanele 5 - 7 vor avea oferta actuală, iar de la 8 încolo se vor face cumulări de soluţii. Linia a 4-a de la 0 la 19 se iau soluţiile precedente şi de la 20 încolo se vor cumula ofertele, iar la linia 5 de la 0 la 9 se va lua soluţia precedentă, coloanele 10, 11 şi 12 vor lua oferta actuală şi de la 13 până la finalul liniei se vor face cumulări de oferte.

În final putem extrage soluţia de la ultima linie şi ultima coloană a tabelei dp.

Capitolul 4

Backtracking

4.1 Enunţ

Se citeşte din fisierul plata.in care are pe prima linie numărul tipurilor de bancnote urmat de o sumă, pe a doua linie valorile bancnotelor, iar pe a treia linie numărul de bancnote disponibile pentru fiecare valoare citită de la linia 2. Să se genereze modalităţile de plată sub formă de submulţimi a sumei cu bancnotele disponibile. Submulţimile trebuie să fie în ordine crescătoare şi să nu se repete.

4.2 Descrierea soluţiei probemei

Pentru a genera toate combinaţiile posibile, dar având anumite restricţii se poate apela fără ezitări la Backtracking. Algoritmul are 3 funcţii. Prima este main în care se citesc toate datele şi se crează vectorul de bancnote în care se va insera câte o bancnotă. A doua este backtracking care creează un vector de soluţii, se iterează prin vectorul de domeniu sau vectorul de bancnote şi se verifică dacă valoarea din domeniu e mai mare decât ultima valoare din soluţie (dacă valoarea există). Dacă este, atunci se adaugă acel element în soluţie, îl scoate din domeniu şi apoi prin recursivitate se creează toate combinaţiile posibile care au acelaşi început. Odată ce s-a ajuns la o soluţie se verifică dacă este validă, adică dacă suma soluţiei este exact suma căutată. Dacă da, aici intervine partea cu restricţii. Se va verifica dacă solution se regăseşte deja printre soluţiile găsite utilizând a treia funcţie, is_not_in_sols. Dacă soluţia e nouă atunci ea va fi adăugată la vectorul de soluţii. În final se va afişa vectorul de soluţii.

4.3 Prezentarea algoritmului de rezolvare a problemei

```
int is_not_in_sols(solution, sols)
        pentru i de la 0  la marimea lui sols - 1
                daca (marimea lui sols(i) = marimea lui solution)
                        same = 1;

                        pentru j  de la 0  la marimea lui sols(i) - 1
                                daca (sols(i)(j) diferit de solution(j))
                                        same = 0;
                                        iesi din for

                        daca (same = 1)
                                returneaza 0;

        returneaza 1;


void bactracking(domain, solution, sols, sol_sum, sum)
        daca (sol_sum > sum)
                iesi din functie;

        daca (sol_sum = sum)
                daca(is_not_in_sols(solution, sols)  = 1)
                        adauga solution in sols;
                        iesi din functie;

        pentru i de la 0 la marimea lui domain - 1
                daca solution are elemente
                   si domain(i) < ultimul element al lui solution
                        continua;

                new_domain = domain;
                scoate domain(i) din new_domain;

                new_solution = solution;
```

```
        adauga domain(i) in solution;

        new_sol_sum = sol_sum + domain(i);

        backtracking(new_domain, new_solution, sols, new_sol_sum, sum);


int main()
        citeste n;
        citeste S;

        citeste types;

        pentru i de la 0 la n - 1
                citeste times;
                pentru j de la 0 la times -1
                        adauga types(i) in domain;

        backtracking(domain, solution, sols, 0, S);

        pentru i de la 0 la marimea lui sols
                pentru j de la 0 la marimea lui sols(i)
                        scrie sols(i)(j);

                scrie pe urmatoarea linie

        returneaza 0;
```

4.4 Aprecierea complexităţii algoritmului propus

Din punct de vedere temporal algoritmul conţine un for la citire de complexitate O(n * max(times)) şi un for pentru scriere de complexitate O(sol.size * sols(i).size). În funcţia is_not_in_sols algoritmul iterează prin toate solutiile lui sols deci avem complexitate O(sol.size * sols(i).size), iar în funcţia backtracking, pentru că avem un for care trece prin tot domeniul şi un apel recursiv care va genera toate combinaţiile posibile,

vom avea complexitate O(m * m!) unde m este numărul de bancnote. Comparând cele 3 complexităţi pe cazul cel mai rău ar rezulta O(n * max(times)) < O(sol.size * sols(i).size) ≤ O(m * m!) pentru că generarea soluţiilor ia cel mai mult timp şi pentru că numărul soluţiilor va fi evident mai mic decât numărul iteraţiilor de backtracking. În concluzie complexitatea temporală este O(m * m!), unde m este numărul de bancnote disponibile.

Din punct de vedere spaţial, programul are un vector care stochează tipurile de bancnote, deci O(n) şi un vector domain care stochează toate bancnotele, una câte una, deci O(m), unde m este numărul de bancnote disponibile.

Astfel pentru acest program avem complexitate temporală O(m * m!) şi complexitate spaţială O(m), unde m este numărul de bancnote.

4.5 Analiză succintă asupra eficienţei algoritmului propus

Algoritmul nu are foarte multe optimizări. Singura optimizare este în momentul în care se alege din domeniu o valoare şi se impune restricţia pentru multimi crescătoare. Dacă elementul din domeniu e mai mic decât ultimul element din soluţie, atunci se trece la următoarea iteraţie, acest lucru eliminând o bună parte din combinaţii. In plus se mai pot aplica optimizările din compilator cu O1, O2 sau O3, dar altfel consider acest algoritm ca fiind eficient pentru această problemă.

4.6 Exemplificarea aplicarii algoritmului propus pentru un exemplu sugestiv

4.6.1 Exemplu de input

```
plata.in
4 40
1 5 10 50
5 3 3 1
```

```
Rezultat:
1 1 1 1 1 5 5 5 10 10
1 1 1 1 1 5 10 10 10
5 5 10 10 10
```

4.6.2 Aplicarea algoritmului

Algoritmul va primi datele de intrarea si va forma vectorul domain cu toate bancnotele disponibile si apoi se va apela funcţia backtracking. Acesta va fi parcursul:

Pasul 1: Solution = {1}; Domain = {1, 1, 1, 1, 5, 5, 5, 10, 10, 10, 50}
Pasul 2: Solution = {1, 1}; Domain = {1, 1, 1, 5, 5, 5, 10, 10, 10, 50}
Pasul 3: Solution = {1, 1, 1}; Domain = {1, 1, 5, 5, 5, 10, 10, 10, 50}
Pasul 4: Solution = {1, 1, 1, 1}; Domain = {1, 5, 5, 5, 10, 10, 10, 50}
Pasul 5: Solution = {1, 1, 1, 1, 1}; Domain = {5, 5, 5, 10, 10, 10, 50}
Pasul 6: Solution = {1, 1, 1, 1, 1, 5}; Domain = {5, 5, 10, 10, 10, 50}
Pasul 7: Solution = {1, 1, 1, 1, 1, 5, 5}; Domain = {5, 10, 10, 10, 50}
Pasul 8: Solution = {1, 1, 1, 1, 1, 5, 5, 5}; Domain = {10, 10, 10, 50}
Pasul 9: Solution = {1, 1, 1, 1, 1, 5, 5, 5, 10}; Domain = {10, 10, 50}
Pasul 10: Solution = {1, 1, 1, 1, 1, 5, 5, 5, 10, 10}; Domain = {10, 50}
Pasul 11: Solution = {1, 1, 1, 1, 1, 5, 5, 5, 10, 10, 10}; Domain = {50}
-> suma bancnotelor din Solution este 40, deci se adaugă Solution in sols.

Urmează un set de pasi unde soluţiile vor fi identice cu prima soluţie până când:

Pasul 12: Solution = {1, 1, 1, 1, 1, 5}; Domain = { 10, 10, 10, 50}
Pasul 13: Solution = {1, 1, 1, 1, 1, 5, 10}; Domain = {10, 10, 50}
Pasul 14: Solution = {1, 1, 1, 1, 1, 5, 10, 10}; Domain = {10, 50}
Pasul 15: Solution = {1, 1, 1, 1, 1, 5, 10, 10, 10}; Domain = {50}
-> suma bancnotelor din Solution este 40, deci se adaugă Solution in sols.

Urmează din nou un set de soluţii care se vor repeta până când:

Pasul 16: Solution = {5}; Domain = {5, 10, 10, 10, 50}
Pasul 17: Solution = {5, 5}; Domain = {10, 10, 10, 50}
Pasul 18: Solution = {5, 5, 10}; Domain = {10, 10, 50}
Pasul 19: Solution = {5, 5, 10, 10}; Domain = {10, 50}
Pasul 20: Solution = {5, 5, 10, 10, 10}; Domain = {50}

-> suma bancnotelor din Solution este 40, deci se adaugă Solution in sols.

Până la finalul execuţiei metoda backtracking nu va mai găsi alte combinaţii. În final se vor afişa în output soluţiile:

1 1 1 1 1 5 5 5 10 10

1 1 1 1 1 5 10 10 10

5 5 10 10 10

Capitolul 5

Analiză comparativă

5.1 Divide et Impera

Tehnica Divide et Impera aduce multe avantaje mai ales la sortări deoarece cu ajutorul acestei tehnici s-a ajuns să se creeze algoritmi de sortare foarte eficienţi cum ar fi Merge Sort care are cea mai bună complexitate pe toate cele 3 cazuri dintre toţi algoritmii de sortare, O(n * log n). De asemenea, această metodă a introdus şi algoritmul de căutare binară care efectuat pe un vector sortat va avea o complexitate O(log n) care este printre cele mai bune din toată lista de algoritmi de căutare, însă dacă nu este sortat, atunci căutarea binară va avea O(n) care este asemănătoare cu complexitatea căutării liniare. De asemenea, tot în acest caz este mai puţin eficient ă din punct de vedere al timpului de execuţie căutarea binară dacă se apelează împreună cu o funcţie de sortare faţă de alţe metode cum ar fi chiar căutarea liniară. Astfel, Divide et Impera este o tehnică eficientă, însă doar pentru anumite probleme şi în anumite condiţii.

5.2 Greedy

Tehnica Greedy îmbină eficienţa cu simplitatea rezolvării unui algoritm folosind "scurtături" pentru a construi cât mai rapid soluţiile. De exemplu, în Problema Rucsacului unde avem un număr de obiecte care au o greutate şi o valoare vom aplica tehnica Greedy prin compararea obiectelor cu rezultatul câtului dintre preţul si greutatea obiectelor. Un avantaj este că vom obţine o soluţie mult mai rapidă şi mai eficentă, însă nu este şi cea mai corectă soluţie. Pe anumite input-uri acest raţionament nu scoate soluţia cea mai optimă ceea ce este un dezavantaj pentru tehnica Greedy. Pentru a rezolva acest neajuns putem folosi Programare Dinamică.

5.3 Programare dinamică

Programarea dinamică are la bază un concept cu totul diferit faţă de Greedy, construieşte soluţia actuală după soluţiile precedente folosind un vector sau o matrice unde sunt reţinute toate soluţiile. Avantajul faţă de Greedy este că soluţia creată va fi mereu cea mai optimă, însă un mare dezavantaj îl constituie crearea vectorului/matricei de soluţii care pentru input-uri mari au o dimensiune foarte mare consumând astfel timp şi resurse. Pe de altă parte există algoritmi eficienţi care au la bază programare dinamică. Dacă avem o recurenţa, de exemplu, în loc să folosim un vector în care să reţinem soluţiile, putem folosi tehnica de exponenţiere pe matrice pentru recurenţe liniare care o complexitate excelentă, O(log n) care scoate un timp excelent, mult mai bun decât soluţia clasică. Astfel, Programarea Dinamică este în general folosită eficient la problemele care se rezolvă cu recurenţe sau cu soluţii care depind de cele precedente.

5.4 Backtracking

Metoda Backtracking este folosită pentru a crea toate soluţiile posibile având anumite restricţii de care să putem ţine cont. Este folosit în general în problemele unde trebuiesc generate mulţimi sau submulţimi pe baza unui domeniu dat şi care să respecte restricţiile impuse. Avantajul acestei metode este eficienţa în comparaţie cu metoda brute-force care generează toate posibilităţile si deabia după le selectează, însă chiar şi aşa complexitatea algoritmilor care au la bază Backtracking este foarte mare. Faţă de celelalte metode Backtraking este mai mult ca o ultimă alternativă, ceva ce se poate folosi eficient doar dacă nu există alte posibilităţi şi din acest punct de vedere se poate considera o metodă eficientă.

Bibliografie

[1] Muresan Vasile Ciprian. Sursa problema Greedy - Probleme de Informatică(Ultima accesare: 23.04.2021). https://info.mcip.ro/?cap=Greedy&prob=592.