Tema 1 - PA

Craciun Alexandru-Andrei

Universitatea Politehnica Bucuresti,Facultatea de Automatica si Calculatoare
alexandru.craciun01@stud.acs.upb.ro

1 Tehnica Divide et Impera

1.1 Enuntul problemei

Se citeste un vector cu n elemente numere naturale. Sa se calculeze suma elementelor vectorului folosind divide et impera. [1]

1.2 Descrierea solutiei problemei

Fie vectorul v cu n elemente. Incepem de la ultimul element, pe pozitia n, si suma finala este egala cu primul numar + suma tuturor elementelor de dupa el. Astfel spargem problema principala in 1 si n - 1 probleme.
Astfel avem problema :

$$sumaVector(v, index) = \begin{cases} \text{v[index]} & \text{if } index == 0 \\ \text{v[index] + sumaVector(v, index - 1)} & \text{otherwise} \end{cases}$$

1.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 1 Sume Elementelor unui Vector

```
1: procedure SUMAVECTOR(V, INDEX)
2:     if index = 0 then
3:         return v[index]
4:     else
5:         return v[index] + sumaVector(v, index - 1)
```

1.4 Aprecierea complexitătii algoritmului propus

Complexitatea problemei este : T(n) = T(n - 1) + O(1).
Folosind recurenta $\Longrightarrow$ T(n) = T(1) + (n - 1) $\cdot$ O(1). Stiind ca T(1) are complexitatea O(1) $\Longrightarrow$ T(n) are complexitatea O(n).

1.5 Analiză succintă asupra eficientei algoritmului propus

Din complexitatea de mai sus eu presupun ca algoritm este optim. Totusi aceasta problema se mai poate rezolva printr-un for care are aceasi complexitate ca rezolvarea de mai sus.

1.6 Exemplu

Fie v $=\{1, 2, 3\}$ si n = 3
I. Avem problema sumaVector(v, 2). Aceasta returneaza 3(v[2]) + sumaVector(v,1)
II. Avem problema sumaVector(v, 1). Aceasta returneaza 2(v[1]) + sumaVector(v,0)
III. Avem problema sumaVector(v, 0). Cum indexul este 0 aceasta returneaza 1(v [0]).
Prin apelul recursiv suma este: 3 + sumaVector(v,1) = 3 + 2 + sumaVector(v,0) = 3 + 2 + 1 = 6

2 Tehnica Greedy

2.1 Enuntul problemei

Subsir crescator maximal. Se citeste un numar n si apoi un sir de n numere intregi.Aflati numarul de elemente cu proprietate ca formeaza un sir cu elementele in ordine crescatoare. Problema a fost modificata. [2]

2.2 Descrierea solutiei problemei

La fiecare iteratie se verifica daca numarul urmator respecta proprietatea din enunt. Daca da atunci maximul este incrementat. In caz contrar se verifica se salveaza maximul intr-o alta variabila si se reseteaza maximul temporar.

2.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 2 Maximul sirului de elemente crescatoare

```
1: procedure maxSir(v)
2:     max ← 1, finalMax ← 1
3:     for i ← 0, v.size − 1 do
4:         if v[i + 1] >= v[i] then
5:             max ← max + 1
6:         else
7:             if finalMax <= max then
8:                 finalMax ← max
9:             max ← 1
10:    if finalMax <= max then
11:        return max
12:    return finalMax
```

2.4 Aprecierea complexitătii algoritmului propus

Fiindca in algoritmul prezentat mai sus este un for, complexitatea acestuia este O(n).

2.5 Exemplu

Fie v =$\{1, 6, 9, 2, 4\}$ astfel v.size = 5
I.6 este mai mare fata de 1 astfel max = max + 1 = 2.
II.9 este mai mare fata de 6 astfel max = max + 1 = 3.
III.2 este mai mic fata de 9 astfel finalMax = max = 3 si max este resetat.
IV.4 este mai mare fata de 2 astfel max = max + 1 = 2.
V. La sfarsit max este mai mic fata de finalMax deci se returneaza finalMax = 3.

3 Tehnica Programării Dinamice

3.1 Enuntul problemei

O livada este impartita in N x N zone. In fiecare zona creste cate un pom. Din fiecare pom cade pe jos o cantitate de fructe.
In zona stanga sus se afla un arici care vrea sa ajunga in zona dreapta jos. Ariciul se poate deplasa doar pe doua directii: in jos sau spre dreapta.
Determinati cantitatea maxima de fructe pe care le poate aduna ariciul prin deplasarea din pozitia initiala in cea dorita. [3]

3.2 Descrierea solutiei problemei

Avem o matrice auxiliara solution de dimensiunea N x N. Stiind sigur ca in pozitia finala solution = livada. Matricea solution se completeaza din dreapta-jos pana in coltul din staga-sus.Fiecare valoare din matrice este egala cu suma dintre valoarea maxima dintre pasii care se pot face si numarul de fructe aflate pe pozitia respectiva in livada.

3.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 3 Arici Flamand

```
1: procedure MAXIMFRUCTE(LIVADA, SOLUTION,POZI, POZJ)
2:     for i ← pozi, 0 do
3:         for j ← pozj, 0 do
4:             if i = posi && j = posj then
5:                 continue
6:             else if i = posi then
7:                 solution[i][j] ← livada[i][j] + solution[i][j + 1]
8:             else if j = posj then
9:                 solution[i][j] ← livada[i][j] + solution[i + 1][j]
10:            else
11:                solution[i][j] ← livada[i][j]+max(solution[i+1][j], solution[i][j+1])
```

3.4 Aprecierea complexitătii algoritmului propus

Deoarece este completata fiecare valoarea din tabela de solutii, complexitatea este O(n^2).

3.5 Explicarea modului în care a fost obtinută relatia de recurentă

I.Cazul de baza.

Pozitia (i, j) este pozitiaFinala, astfel solution[i][j] = livada[i][j].
II.Cazul general.

Daca i = iFinal atunci solution[i][j] = livada[i][j] + solution[i][j+1]
Daca j = jFinal atunci solution[i][j] = livada[i][j] + solution[i+1][j]
Altfel solution[i][j] = livada[i][j] + max(solution[i+1][j], solution[i][j+1])
III.Recurenta finala.

$$solution(i,j) = \begin{cases} \text{livada[i][j]} & \text{if}(i,j) == pozitiaFinala \\ \text{livada[i][j] + max(solution[i+1][j], solution[i][j+1])} & \text{otherwise} \end{cases}$$

3.6 Exemplu

Fie livada:
0 4 1
0 1 1
1 0 1
Si pozitie finala: (2, 2)
Astfel construim matricea auxiliara solution astfel:
I. Incepem din pozitia finala $\Longrightarrow$ solution[2][2] = livada[2][2] = 1
II. Calculam maxim vecinilor pozitiei anterioare $\Longrightarrow$
solution[1][2] = livada[1][2] + solution[2][2] (deoarece singura miscare posibila este la dreapta) = 0 + 1 = 1
solution[2][1] = livada[2][1] + solution[2][2] (deoarece singura miscare posibila este in jos) = 1 + 1 = 2
III. Astfel calculam pentru toate elementele din matricea solution
IV. Cazul in care se poate face o miscare in jos si la dreapta atunci se ia maximul dintre cele 2 valori.
De exemplu:
solution[1][1] = livada[1][1] + max(solution[2][1], solution[1][2]) = 1 + max(1, 2) = 1 + 2 = 3

4 Tehnica Backtracking

4.1 Enuntul problemei

O tabla de sah se citeste ca o matrice N x N in care pozitiile libere au valoarea 0, iar piesele adverse sunt marcate prin valoarea -1. Pe prima linie pe coloana j se afla un pion. Sa se determine drumul pe care poate ajunge pionul pe ultima linie luand un numar maxim de piese. Pionul aflat in pozitia i,j se poate deplasa astfel: - in pozitia i+1,j daca e libera - in pozitia i+1, j-1 daca e piesa in aceasta pozitie - in pozitia i+1, j+1 daca e piesa in aceasta pozitie. [4]

4.2 Descrierea solutiei problemei

La fiecare pas al pionului se verifica daca exista inamici pe pozitii avantajoase(in diagonala) sau daca se poate muta in fata(nu exista pion advers in fata). Astfel se creaza cate o ramura pentru fiecare caz in fiecare. Cand se ajunge la sfarsitul tablei se printeaza drumul pionului folosind un contor. Cand se ajunge intr-o pozitie imposibile(nu exista pioni adversi in pozitii avantajoase si exista un adversar in fata pionului) se termina ramura respectiva.

4.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 4 Drumul pionului

```
1: procedure SOLVEPAWN(TABLE, POSI, POSJ,PASI
2:     if posi = table.size − 1 then
3:         table[posi][posj] ← pasi
4:         printTable
5:         table[posi][posj] ← 0
6:         return
7:     table[posi][posj] ← pasi
8:     if posj > 0 && table[posi + 1][posj − 1] = −1 then
9:         solvePawn(table,posi + 1, posj - 1, contor + 1)
10:        table[posi + 1][posj − 1] ← −1
11:    if posj < size − 1 && table[posi + 1][posj + 1] = −1 then
12:        solvePawn(table,posi + 1, posj + 1, contor + 1)
13:        table[posi + 1][posj + 1] ← −1
14:    if table[posi + 1][posj] = −1 then
15:        table[posi][posj] ← 0
16:        return
17:    solvePawn(table, posi + 1, posj, contor)
18:    table[posi][posj] ← 0
19:    return
```

4.4 Aprecierea complexitătii algoritmului propus

Complexitatea problemei este : O(n) deoarece se face parcurgerea doar pe liniile tablei de sah.

4.5 Analiză succintă asupra eficientei algoritmului propus

Algoritmul prezinta toate drumurile posibile ale pionului deoarece la fiecare pas verifica ce drum se poate alege si se creaza cate o ramura pentru fiecare drum posibil. Se verifica pe rand fiecare miscare a pionului si in cazul in care miscarea verificata este posibila se creaza ramura si se traverseaza aceasta pana la sfarsit. La final se revine la starea din care am pornit si se verifica urmatoarea miscare. Nu cred ca se poate aplica vreo varianta de optimizare.

4.6 Exemplu

Fie tabla de sah:
0 P 0
-1 0 0

0 -1 0

I.initial pionul ataca adversarul de pe pozitia (1, 0)
II.Ataca pionul de pe pozitia (2,1)
III. S-a ajuns la sfarsitul tablei, astfel tabla este:
0 1 0
2 0 0
0 3 0

IV.Se revine la pasul 2 si pionul avanseaza o pozitie (2,0)
V. S-a ajuns la sfarsitul tablei, astfel tabla este:
0 1 0
2 0 0
3 -1 0

VI. Se revenie la pasul 1 si pionul avanseaza o pozitie (1,1)
VII. Nu exista o miscare posibila(nu exista un adversar aflat in pozitie de atac si pozitia din fata este blocata).

5 Analiza comparativa

5.1 Problema aleasa

Am ales problema descrisa la programarea dinamica, problema cu livada si ariciul. Pentru aceasta am sa compar programarea dinamica si tehnica greedy.

5.2 Algoritm tehnica greedy

Algorithm 5 Arici Flamand Greedy

```
1: procedure MAXIMFRUCTE(LIVADA)
2:     max ← 0, starti ← 0, startj ← 0
3:     while starti! = livada.size − 1 || startj! = livada.size − 1 do
4:         max ← max + livada[starti][startj]
5:         if starti = livada.size − 1 then
6:             startj ← startj + 1
7:         else if startj = livada.size − 1 then
8:             starti ← starti + 1
9:         else
10:            if livada[starti + 1][startj] >= livada[starti][startj + 1] then
11:                starti ← starti + 1
12:            else
13:                startj ← startj + 1
```

Algortimul de mai sus calculeaza maximul in functie de urmatoarele 2 miscari pe care poate sa le faca ariciul(Astfel calculeaza in functie de maximul local al problemei).

5.3 Avantaje si dezavantaje

I.Programarea dinamica ofera un raspuns mai bun fata de tehnica greedy.
Fie livada:
0 4 2 2
0 1 1 0
1 0 3 0
4 2 1 0
Tehnica greedy ofera maximul ca fiind 8, fata de programarea dinamica care ofera rezultatul ca fiind 10.
II.Tehnica greedy are o complexitate mai mica fata de programarea dinamica.
Tehnia greedy: Pentru fiecare dimensiune a matricii ariciul trb sa faca 2 pasi astfel complexitatea este $O(2 \cdot (n - 1)) = O(n)$
Programarea dinamica: Trebuie completata fiecare valoarea din tabela de solutii astfel complexitatea este $O(n^2)$.

5.4 Exemple

Fie livada precizata anterior:
0 4 2 2
0 1 1 0
1 0 3 0
4 2 1 0

Tehnica greedy:
I.Ariciul vede ca maximul dintre urmatoarele 2 miscari este 4 astfel noua lui pozitie este (0, 1) si maximul este 4.
II.Ariciul vede ca maximul dintre urmatoarele 2 miscari este 2 astfel noua lui pozitie este (0, 2) si maximul este 6.
III.Ariciul vede ca maximul dintre urmatoarele 2 miscari este 2 astfel noua lui pozitie este (0, 3) si maximul este 8.
IV. Fiindca ariciul a ajuns la sfarsitul livezii, acesta merge in jos pana la sfarsit. Maximul este la final 8.

Programarea Dinamica:
I. Incepem din pozitia finala $\Longrightarrow$ solution[3][3] = livada[3][3] = 0
II. Calculam maxim vecinilor pozitiei anterioare $\Longrightarrow$
solution[2][3] = livada[2][3] + solution[3][3] (deoarece singura miscare posibila este la dreapta) = 0 + 1 = 1
solution[3][2] = livada[3][2] + solution[3][3] (deoarece singura miscare posibila este in jos) = 0 + 0 = 0

III. Astfel calculam pentru toate elementele din matricea solution
IV. Cazul in care se poate face o miscare in jos si la dreapta atunci se ia maximul dintre cele 2 valori.
De exemplu:
solution[0][1] = livada[0][1] + max(solution[1][1], solution[0][2]) = 1 + max(10, 5) = 0 + 10 = 10
Maxim final este 10.

References

1. https://info.mcip.ro/?cap=Divide%20et%20Impera&prob=205
2. https://info.mcip.ro/?cap=Programare%20dinamica&prob=405
3. https://info.mcip.ro/?cap=Programare%20dinamica
4. https://info.mcip.ro/?cap=Programare%20dinamica&prob=750