# TEMA 0- Proiectarea Algoritmilor

Negoescu Bianca-Maria, 321CD

Facultatea de Automatica si Calculatoare

Universitatea Politehnica Bucuresti

Aprilie 2021

# CONTENTS

# 1. DIVIDE ET IMPERA

## 1.1. Problema Placilor

### 1.1.1. Enunt

Avand o placa de dimensiuni n x n, unde n este de forma $2^k$, k > = 1, cu o celula lipsa (de dimensiunea 1 x 1). Umpleti placa folosind placi in forma de L. O tigla in forma de L este un patrat de 2 x 2, din care lipseste o celula de dimensiunea $1 \times 1$.

### 1.1.2. Descrierea solutiei

Aceasta problema poate fi rezolvata folosind Divide et Impera.
n este dimensiunea placii, iar p pozitia placii lipsa

1. Best Case: daca n = 2, un patrat de dimensiuni 2 x 2 cu o celula lipsa este o placa si poate fi umplut cu o singura placa.
2. Se pune o placa in forma de L in centru astfel incat sa nu acopere subpatratul de dimensiuni n / 2 * n / 2 care are un patrat lipsa. Acum toate cele patru subpatrate cu dimensiunea n / 2 x n / 2 au o celula lipsă (o celula care nu are trebuie completata).
3. Se rezolva problema recursiv pentru urmatoarele patru subpatrate.

### 1.1.3. Prezentarea algoritmului de rezolvare (Pseudocod)

**int** recursiveTile (n, x, y)
**if** n = 2 **then**
-Plaseaza o singura placa sub forma de L
return 0;
**else**
-Itereaza prin matrice si gaseste locatia (coordonatele) placii lipsa
-Verifica in ce cadran se afla si plaseaza o placa in forma de L astfel incat sa acopere toate cadranele mai putin cel in care se afla patratica lipsa
-Imparte recursiv in 4 subcadrane:
recursiveTile(n / 2, x, y + n / 2);
recursiveTile(n / 2, x, y);
recursiveTile(n / 2, x + n / 2, y);
recursiveTile(n / 2, x + n / 2, y + n / 2);
return 0;

### 1.1.4. Complexitate

Relatia de recurenta este urmatoarea:
T(n) = 4T(n/2) + C, unde C este o constanta
Recurenta se poate rezolva cu Master, de unde rezulta o complexitate temporala de $O(n^2)$.

### 1.1.5. Eficienta algoritmului

O complexitate temporala de $O(n^2)$ nu este foarte buna, algoritmul putand fi imbunatatit la o complexitate de O(n).

### 1.1.6. Exemplificarea aplicarii algoritmului

Input : Fie matricea B cu n = 4
Apelul initial din main se va face pentru coordonatele (x, y) = (0, 0), iar inainte de apel se va initializa B[0][0] cu -1.
-1 | 0 | 0 | 0
0 | 0 | 0 | 0
0 | 0 | 0 | 0
0 | 0 | 0 | 0
recursiveTile (4, 0, 0)
Se cauta primul element diferit de 0, la noi este B[0][0] = -1, care se gasese in primul cadran, deci vom plasa o placa L pe pozitiile B[2][1], B[2][2] si B[1][2]
-1 | 0 | 0 | 0
0 | 0 | 1 | 0
0 | 1 | 1 | 0
0 | 0 | 0 | 0
Apoi vin cele 4 apeluri recursive:
recursiveTile(2,0,2)
recursiveTile(2,0,0)
recursiveTile(2,2,0)
recusriveTile(2,2,2)
Prima data se intra in primul apel recursiv

- recursiveTile(2,0,2)

  n = 2, x = 0, y = 2
  n = 2 (A)
  cnt++ => cnt = 2
  B[0][2] ≠ 0(*A*) => *B*[0][2] = 2
  *B*[0][3] ≠ 0(*A*) => *B*[0][3] = 2

$B[1][2] \neq (F)$
$B[1][3] \neq (A)$ => $B[1][3] = 2$

-1 | 0 | 2 | 2
0 | 0 | 1 | 2
0 | 1 | 1 | 0
0 | 0 | 0 | 0

- recursiveTile(2,0,0)
  n = 2, x = 0 , y = 0
  n = 2 (A)
  cnt++ => cnt = 3
  B[0][0] ≠ 0($F$)
  $B[0][1] \neq 0(A)$ => $B[0][1] = 3$
  $B[1][0] \neq (A)$ => $B[1][0] = 3$
  $B[1][1] \neq (A)$ => $B[1][1] = 3$

  -1 | 3 | 2 | 2
  3 | 3 | 1 | 2
  0 | 1 | 1 | 0
  0 | 0 | 0 | 0

- recursiveTile(2,2,0)
  n = 2, x = 2 , y = 0
  n = 2 (A)
  cnt++ => cnt = 4
  B[2][0] ≠ 0($A$) => $B[2][0] = 4$
  $B[2][1] \neq 0(F)$
  $B[3][0] \neq (A)$ => $B[3][0] = 4$
  $B[3][1] \neq (A)$ => $B[3][1] = 4$

  -1 | 3 | 2 | 2
  3 | 3 | 1 | 2
  4 | 1 | 1 | 0
  4 | 4 | 0 | 0

- recursiveTile(2,2,2)
  n = 2, x = 2 , y = 2
  n = 2 (A)
  cnt++ => cnt = 5

B[2][2] ≠ 0(*F*)
*B*[2][3] ≠ 0(*A*) => *B*[2][3] = 5
*B*[3][2] ≠ (*A*) => *B*[3][2] = 5
*B*[3][3] ≠ (*A*) => *B*[3][3] = 5

Output:
-1 | 3 | 2 | 2
3 | 3 | 1 | 2
4 | 1 | 1 | 5
4 | 4 | 5 | 5

# 2. TEHNICA GREEDY

## 2.1. Algoritm Greedy pentru Fractiile Egiptene

### 2.1.1. Enunt

Fiecare fractie pozitiva poate fi reprezentata ca o suma de fractii unitate unice. O fractie este fractie unitate daca numaratorul este 1 si numitorul este un numar pozitiv intreg. Spre exemple, 1/3 este o fractie unitate. O astfel de reprezentare se numeste fractie egipteana, deoarece erau folosite de catre egipteni.

### 2.1.2. Descrierea solutiei

Pentru un numar dat de forma nr/dr, unde dr > nr, mai intai se gaseste cea mai mare varianta de fractie unitate, dupa care se repeta pentru partea ramasa.

### 2.1.3. Prezentarea algoritmului de prezentare (Pseudocod)

**void** printEgyptian (int nr, int dr)
**if** nr == 0 **or** dr == 0 **return**
**if** dr%nr == 0 **afiseaza** 1/ (dr/nr)
**if** nr%dr == 0 **afiseaza** nr/dr
**if** nr > dr {
**afiseaza** nr/dr + **printEgyptian(nr%dr, dr)**
**return** }
int n = dr/nr + 1
**afiseaza** 1/n + **printEgyptian(nr*n-dr, dr*n)**

### 2.1.4. Complexitate

- Daca nr < dr, vor fi maxim "nr" intrari in apelul recursiv, rezultand o complexitate temporala de O(nr).
- Daca nr > dr, vor fi vor fi (nr % dr) intrari in apelul recursiv, rezultand o complexitate temporala de O(nr % dr), dar "nr" fiind mai mic decat "dr", nr % dr = nr, deci complexitatea va fi tot O(nr).

### 2.1.5. Posibilitatea de obtinere a optimului global

Algoritmul Greedy functioneaza corect deoarece o fractie este intotdeauna redusa la o forma unde numitorul este mai mare decat numaratorul si numaratorul nu divide numitorul. Pentru astfel de forme reduse, apelul recursiv se face pentru un numarator redus. Astfel, apelul recursiv scade din ce in ce mai mult numaratorul pana ce acesta ajunge sa fie 1.
Datorita corectitudinii sale, rezulta ca in urma alegerilor optime locale din cadrul fiecarui apel recursiv, se va obtine solutia optima global. De aici reiese ca **problema are proprietatea alegerii locale**.
Deoarece am concluzionat ca solutia finala globala este una optima, fiindca se bazeaza pe solutiile locale, este evident ca solutia optima a problemei contine solutiile optime ale subproblemelor. De aici rezulta ca problema are **proprietatea de substructura optima**.

### 2.1.6. Exemplificarea aplicarii algoritmului

Input: Sa se calculeze 6/14
nr = 6, dr = 14
printEgyptian (6, 14)
n = 14/6 + 1 = 2 + 1 = 3
**afiseaza** 1/3 + **printEgyptian(6 * 3 - 14, 14 * 3)**
printEgyptian(4, 42)
nr = 4, dr = 42
n = 42/4 + 1 = 11
**afiseaza** 1/11 + **printEgyptian(11 * 4 - 42, 42 * 11)**
printEgyptian(2, 462)
462 % 2 = 0 (A) => **afiseaza** 2/462 = 1/231
Output :
1/3 + 1/11 + 1/231

# 3. TEHNICA PROGRAMARII DINAMICE

## 3.1. Coin Change

### 3.1.1. Enunt

Se da o valoare N. Daca vrem sa dam rest pentru N centi si avem o rezerva nelimitata din fiecare S = S1, S2, ..., Sm monede, in cate modalitati putem da rest? Ordinea monedelor nu conteaza.
Spre exemplu, N = 4 si S = 1,2,3.
Avem 4 solutii: 1,1,1,1,1,1,2,2,2,1,3.
Deci, outputul ar trebui sa fie 4.
Pentru N = 10, exista 5 solutii: 2,2,2,2,2, 2,2,3,3, 2,2,6, 2,3,5 and 5,5. Deci outputul va fi 5.

### 3.1.2. Descrierea solutiei

Pentru a determina numarul total de solutii, putem imparti setul de solutii in 2 seturi.

- Solutii care nu contin moneda Sm.
- Solutii care contin cel putin o moneda Sm.

Fie count(S[], m, n) functia care numara cate solutii exista. Aceasta poate fi scrisa ca suma dintre count(S[], m-1, n) si count(S[], m, n-Sm). Prin urmare, problema are **proprietatea de substructura optima** intrucat problema poate fi solutionata folosind solutiile subproblemelor.

### 3.1.3. Prezentarea algoritmului de functionare (Pseudocod)

**int** count (int S[], int m, int n )
**if** n == 0 **return** 1
**if** n < 0 **return** 0
**if** m <= 0 **and** n >= 1 **return** 0
return **count**(S, m - 1, n) + **count**(S, m, n - S[m - 1])

### 3.1.4. Complexitate

Algoritmul de mai sus calculeaza aceleasi subprobleme de mai multe ori. Daca reprezentam arborele de recurenta, observam ca numeroase subprobleme sunt apelate mai mult de o singura data. Din moment ce subproblemele sunt reapelate, problema are **proprietatea de suprapunere a subproblemelor**. Asadar, coin change problem are ambele proprietati ale programarii dinamice.
Precum orice alta problema tipica de programare dinamica, recalcularea acelorasi subprobleme poate fi evitata prin constructia temporara a unei matrici in maniera bottom up. Complexitatea temporala este: O(n*m).
In cazul cel mai defavorabil, arborele recursiv are inaltimea "n", iar algoritmul rezolva doar "n" subprobleme deoarece se retin in memoria cache solutiile anterior calculate. Fiecare subproblema este calculata cu "m" iteratii. De aici rezulta complexitatea temporala O(n*m).

### 3.1.5. Relatia de recurenta

Relatia de recurenta s-a obtinut pe baza celor doua cazuri mentionate si in cadrul sectiunii "Descrierea solutiei":

- Cazul 1: moneda nu este luata
  Cand moneda $S_i$ nu este luata, numarul de modalitati de a forma restul pentru "n" este exact acelasi ca inainte ca moneda $S_i$ sa fie luata in considerare.
  Asta inseamna ca pentru acest caz apelul recursiv este de forma count(S, m - 1, n).

- Cazul 2: moneda este luata.
  Cand moneda $S_i$ este luata, ii cheltuim valoarea, deci trebuie sa scadem valoarea ei din suma initiala, adica din n. Pentru acest caz, apelul recursiv este de forma count(S, m, n - S[m - 1]).

Din cele 2 cazuri rezulta formula finala de recurenta: count(S, m - 1, n) + count(S, m, n - S[m - 1]).

### 3.1.6. Exemplificarea aplicarii algoritmului

Input: S = {1,2,3}, n = 5
C({1,2,3}, 5) = C({1,2,3}, 2) + C({1,2}, 5) = 1 + 3 = 4
C({1,2,3}, 2) = C({1,2,3}, -1) + C({1,2}, 2) = 0 + 1 = 1
C({1,2}, 5) = C({1,2}, 3) + C({1}, 5) = 2 + 1 = 3
C({1,2,3}, -1) = 0
C({1,2}, 2) = C({1,2},0) + C({1},2) = 1
C({1,2}, 3) = C({1,2},1) + C({1},3) = 1 + 1 = 2

C({1}, 5) = C({1}, 4) + C({}, 5) = 1 + 0 = 1
C({1}, 4) = C({1}, 3) + C({}, 4) = 1
=> numar solutii = 4

# 4. TEHNICA BACKTRACKING

## 4.1. m Coloring Problem

### 4.1.1. Enunt

Se da un graf neorientat si un numbar m. Sa se determine daca graful poate fi colorat cu cel mai mult m culori astfel incat 2 noduri adiacente sa nu fie colorate cu aceeasi culoare.

### 4.1.2. Descrierea solutiei problemei

Ideea este de a atribui culori una cate una la varfuri diferite, incepand de la varful 0. Inainte de a atribui o culoare, se verifica siguranta luand in considerare culorile deja atribuite vârfurilor adiacente, adica se verifica dac varfurile adiacente au sau nu aceeasi culoare. Daca exista o atribuire de culoare care nu incalca conditiile, se marcheaza atribuirea de culoare ca parte a solutiei. Daca nu este posibila nicio atribuire de culoare, se revine la inceput si se returneaza fals.

### 4.1.3. Prezentarea algoritmului de prezentare (Pseudocod)

Fie V- numarul de noduri
color[] - vectorul de culori
i - indexul curent
G - graful
m - numarul maxim de culori care pot fi folosite

**bool** mcoloring( G, m, color[], v)
**if** i == V **afiseaza** configuratia culorilor in color[]
**atribuie** o culoare unui nod de la 1 la m
pentru culoarea atribuita **apeleaza** functia de verificare
**mcoloring**(G, m, color[], v + 1)
**if** un apel recursiv a returnat true, se da **break** din loop si se returneaza **true**
daca niciun apel recursiv nu a returnat true, se returneaza **false**

### 4.1.4. Complexitate

In total vor fi $m^V$ combinatii de culori. De aici rezulta o complexitate temportala de $O(m^V)$. Complexitatea pentru limita superioara a timpului ramane la fel, insa cea pentru

timpul mediu va fi mai mica.

### 4.1.5. Exemplificarea aplicarii algoritmului

Input n = 4 si m = 3
Matricea de adiacenta :
1 | 1 | 0 | 1
1 | 1 | 1 | 1
0 | 1 | 1 | 1
1 | 1 | 1 | 1
mcoloring(1) => color[1] = 1
mcoloring(2) => color[2] = 1 **coliziune** cu color[1] = 1
=> color[2] = 2
mcoloring(3) => color[3] = 1
Se verifica culorile vecinilor 2 si 4, nu exista nicio coliziune, este in regula
mcoloring(4) => color[4] = 1 **coliziune** cu color [1]
color[4] = 2 **coliziune** cu color[2]
color[4] = 3 este in regula
Output:
color[] = 1, 2, 1, 3

# 5. BIBLIOGRAFIE


- Tiling Problem
- Egyptian Fractions
- Coin Change Problem
- m Coloring Problem