

Tema 1 – Proiectarea Algoritmilor

Popa Stefania

Grupa 323b

Cuprins

	PAGINA
1. Tehnica Divide et Impera.....	2
2. Tehnica Greedy.....	6
3. Tehnica Programarii Dinamice.....	9
4. Tehnica Backtracking.....	13
5. Analiza Comparativa	17


1. Tehnica Divide et Impera


1.1 *Enuntul problemei*

Gasiti elementul minim si cel maxim dintr-un vector dat cu n elemente.

1.2 *Descrierea solutiei problemei*


Rezolvarea problemei este realizata in 3 pasi:

 **Divide:** Impartim vectorul in 2 parti egale, deci in acelasi timp, impartim problema in 2 subprobleme de dimensiuni egale.

 **Impera:** Aflam minimul si maximul pentru fiecare dintre cele 2 parti ale vectorului, in mod recursiv (pana se ajunge la cazurile de baza).

(Avem doua cazuri de baza: 1) vectorul are un singur element, acesta fiind si minimul si maximul

2) vectorul are doua elemente, caz in care le comparam pentru a afla care este minimul si care este maximul)

 **Combina:** Comparam elementele de maxim si minim din cele 2 jumatati pentru a afla maximul si minimul vectorului initial.

(Pentru a retine maximul si minimul folosim un vector auxiliar maxMin[2])

1.3 *Prezentarea algoritmului de rezolvare a problemei (pseudocod)*

```

int[] findMinMax (int X[], int l, int r)
{
    int max, min
    if (l == r)
    {
        max = X[l]
        min = X[l]
    }
    else if (l + 1 == r)
    {
        if (X[l] < X[r])
        {
            max = X[r]
            min = X[l]
        }
        else
        {
            max = X[l]
            min = X[r]
        }
    }
    else
    {
        int mid = l + (r - 1)/2
        int lminMax[2] = findMinMax(X, l, mid)
        int rminMax[2] = findMinMax(x, mid+1, r)
        if (lminMax[0] > rminMax[0])
            max = lminMax[0]
        else
            max = rminMax[0]
        if (lminMax[1] < rminMax[1])
            min = lminMax[1]
        else
            min = rminMax[1]
    }

    int maxMin[2] = {max, min}
    return maxMin
}

```

1.4 Complexitatea temporală a algoritmului

Problema este împartită în două subprobleme de dimensiune $n/2$. La fiecare pas „Combina”, efectuăm 2 operații de comparație. Cazurile de bază sunt $n=2$ (\rightarrow 1 comparație) și $n=1$ (nici o comparație).

Asadar, relația de recurență a complexității algoritmului este:

$$T(n) = T(n/2) + T(n/2) + 2 = 2T(n/2) + 2, \quad \text{unde } T(2) = 1 \text{ și } T(1) = 0$$

Deci, complexitatea temporală a algoritmului este $O(n)$.

1.5 Analiza succintă asupra eficienței algoritmului propus

Această abordare a problemei are aceeași complexitate temporală ca în cazul rezolvării clasice, cu „*brute force*”, prin parcurgerea vectorului.

Totuși, rezolvarea cu Divide et Impera are un avantaj, și anume faptul că efectuează un număr mai mic de comparații decât „*brute force*”:

- *Brute force*: $2(n-1)$ comparații \rightarrow worst case
- *Divide et Impera*: $3n / 2 - 2$ comparații \rightarrow $n = \text{putere a lui } 2$

1.6 Exemplificarea aplicării algoritmului

Utilitatea problemei propuse este înțeleasă încă din enunțul acesteia. Câteva exemple interesante ar fi: găsirea extremităților dintr-o listă cu diverse date, precum profitul unei firme pe parcursul mai multor ani sau observarea evoluției/involuției în rândul numărului de exemplare ale unei specii de animale.

Pentru urmatorul exemplu simplu: $X[4] = \{3,1,6,4\}$, pasii algoritmului sunt:

$\text{findMinMax}(X, 0, 3) \rightarrow \text{mid} = 1$

$\rightarrow \text{findMinMax}(X, 0, 1) \rightarrow l+1 = r \text{ caci } 0+1 = 1 \rightarrow$

$l_{\text{max}} = 3, l_{\text{min}} = 1 \rightarrow l_{\text{MinMax}}[2] = \{3, 1\}$

$\rightarrow \text{findMinMax}(X, 2, 3) \rightarrow l+1 = r \text{ caci } 2+1 = 3 \rightarrow$

$r_{\text{max}} = 6, r_{\text{min}} = 4 \rightarrow r_{\text{MinMax}}[2] = \{6, 4\}$

$\rightarrow l_{\text{MinMax}}[0] < r_{\text{MinMax}}[0] \Rightarrow \text{MAX} = 6 \text{ si}$

$l_{\text{MinMax}}[1] < r_{\text{MinMax}}[1] \Rightarrow \text{MIN} = 1$

Sursa: <https://www.enjoyalgorithms.com/blog/find-the-minimum-and-maximum-value-in-an-array>

2. Tehnica Greedy

2.1 *Enuntul problemei*





Sunt secretara unei persoane de afaceri foarte ocupate si trebuie sa-i planific activitatile in mod eficient astfel incat sa apuce sa termine cat mai multe dintre acestea in timpul T pe care il are la dispozitie.

Astfel, avem un vector A de numere reale, unde fiecare element indică timpul necesar pentru finalizarea unei activitati. Scopul meu este sa calculez numărul maxim de lucruri pe care le poate face în timpul limitat pe care îl are la dispozitie .

2.2 *Descrierea solutiei problemei*

Solutia expusa este reprezentata de un algoritm cu prelucrare initială a multimii A , prelucrare care, de fapt, stabileste de la început ordinea în care sunt extrase elementele din A la fiecare pas.

Pasii de rezolvare sunt urmatorii:

-  Sortam crescator vectorul A .
(Pentru a reusi sa efectueze cat mai multe activitati, trebuie sa inceapa cu cele care necesita cat mai putin timp)
-  Selectam pe rand, in ordinea din vectorul sortat, cate o activitate.
-  Adaugam la timpul total, timpul corespunzator activitatii curente.
-  Daca noul timp total nu depaseste timpul T , adaugam 1 la numarul de activitati planificate.

→ Repetam ultimii trei pasi cat timp durata totala de efectuare a activitatilor nu depaseste timpul T .

2.3 *Prezentarea algoritmului de rezolvare a problemei (pseudocod)*

T = timpul in care trebuie sa ne incadram

N = nr de activitati totale

```
int schedule(int A, int T, int N)
{
    int noOfActivities = 0
    int currTime = 0
    ascendingSort(A)
    for(int i =0; i<N; i++)
    {
        currTime+ = A[i]
        if(currTime >T)
            break;
        noOfActivities++;
    }
    return noOfActivities;
}
```

2.4 *Complexitatea temporală a algoritmului*

Pentru pasul de sortare avem complexitate $O(n \cdot \log n)$, iar pentru cel de selectarea a fiecărei activitati, avem in worst case, $O(n)$.

De aici rezulta o complexitate totala de $O(n \cdot \log n)$, care nu ar mai putea fi imbunatatita in acest caz.

2.5 Analiza succinta asupra posibilitatii de obtinere a optimului global

In cazul acestei abordari, cu prelucrare initiala a multimii A, algoritmul duce la o solutie corecta, ce reprezinta chiar optimul global.

Algoritmul respecta cele doua proprietati esentiale pentru obtinerea optimului global: „*Proprietatea de alegere de tip Greedy*” si „*Proprietatea de substructură optimă*”.

2.6 Exemplificarea aplicarii algoritmului

Luam urmatorul exemplu: trebuie sa planificam programul in functie de 5 activitati cu urmatorii timpi necesari corespunzatori : $A = [3, 1, 6, 4, 8]$ si timpul $T = 17$ disponibil.

$\text{ascendingSort}(A) \Rightarrow A = [1, 3, 4, 6, 8]$

$\text{currTime} = 0, \text{noOfActivities} = 0$

$\text{currTime} = 0 + A[0] = 0+1 = 1 < T \Rightarrow \text{noOfActivities} = 0+1 = 1$

$\text{currTime} = 1 + A[1] = 1+3 = 4 < T \Rightarrow \text{noOfActivities} = 1+1 = 2$

$\text{currTime} = 4 + A[2] = 4+4 = 8 < T \Rightarrow \text{noOfActivities} = 2+1 = 3$

$\text{currTime} = 8 + A[3] = 8+6 = 14 < T \Rightarrow \text{noOfActivities} = 3+1 = 4$

$\text{currTime} = 14 + A[4] = 14+8 = 22 > T$. Deci, $\text{noOfActivities} = 4$

Sursa: *inspirat din*

<https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/>

3. Tehnica Programarii Dinamice

3.1 Enunțul problemei

„Drumul de suma maxima in triunghi”: Fie un triunghi format din n linii.

Determinați cea mai mare sumă de numere aflate pe un drum între numărul de pe prima linie și un număr de pe ultima linie. Fiecare număr din acest drum este situat sub precedentul, la stânga sau la dreapta acestuia.

(Un astfel de triunghi ar arata cam asa:

			7			
		3		8		
	8		1		0	
2		7		4		4
4	5	2		6	5	

)

3.2 Descrierea solutiei problemei

Algoritmul de rezolvare se realizeaza in urmatoorii pasi:

- ✚ Vom reține triunghiul într-o matrice pătratică T , de ordin n , sub diagonala principală.

Pentru exemplul de mai sus, matricea va arata asa:

7				
3	8			
8	1	0		
2	7	4	4	
4	5	2	6	5

- ✚ Subproblemele problemei date constau în determinarea sumei maxime care se poate obține din numere aflate pe un drum între numărul $T[i][j]$, până la un număr de pe ultima linie, fiecare număr din acest drum fiind situat sub precedentul, la stânga sau la dreapta sa.
- ✚ Pentru a reține soluțiile subproblemelor, vom utiliza o matrice suplimentară S , pătratică de ordin n , cu semnificația:
 $S[i][j]$ = suma maximă ce se poate obține pe un drum de la $T[i][j]$ la un element de pe ultima linie, respectând condițiile problemei.
 →Soluția problemei va fi $S[1][1]$.

✚ Relația de recurență care caracterizează substructura optimă a problemei este: $S[i][j] = T[i][j] + \max\{ S[i+1][j], S[i+1][j+1] \}$
 $S[n][i] = T[n][i], \forall i \in \{1, 2, \dots, n\}$

3.3 *Prezentarea algoritmului de rezolvare a problemei (pseudocod)*

```
int triangle( int *T, int n)
{
    int S[n+1][n+1] = {0}
    for (i=1; i<=n; i++)
        S[n][i]=T[n][i]
    for (i=n-1; i>=1; i--)
        for (j=1; j<=i; j++)
            if (S[i+1][j] < S[i+1][j+1])
                S[i][j] = T[i][j]+S[i+1][j+1])
            else
                S[i][j] = T[i][j]+S[i+1][j]

    return S[1][1];
}
```

3.4 *Complexitatea temporală și spațială a algoritmului*

Complexitatea temporală:

- Primul for $\rightarrow O(n)$
- for2 + for3 $\rightarrow (n-1) + (n-2) + \dots + 1 = n(n-1)/2$

Deci, complexitatea temporală a algoritmului este $O(n^2)$.

Complexitatea spațială:

- $n, i, j \rightarrow$ fiecare câte 4 biti
- matricea $S \rightarrow (4n)^2$ biti
- matricea $T \rightarrow (4n)^2$ biti

Deci, complexitatea spațială a algoritmului este $O(n^2)$.

3.5 Explicarea modului in care a fost obtinuta relatia de recurenta

Pentru rezolvarea problemei, trebuie sa rezolvam subproblemele ce constau in determinarea sumei maxime care se poate obtine din numere aflate pe un drum între numărul $T[i][j]$, până la un număr de pe ultima linie.

Plecand dintr-un element $T[i][j]$, putem merge la $T[i+1][j]$ (corepunzator elementului de sub $T[i][j]$, partea stanga) sau la $T[i+1][j+1]$ (corepunzator elementului de sub $T[i][j]$, partea dreapta).

Pentru gasirea sumei maxime, trebuie sa alegem calea ce ne ofera elemente cat mai mari, deci dintre cele doua posibilitati, va trebui sa o alegem, dupa comparare, pe cea mai mare.

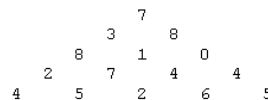
Fac precizarea ca pentru ca toate acestea sa fie posibile, drumul de suma maxima de la un element de pe ultima linie la ultima linie este chiar elementul respectiv, adica $S[n][i] = T[n][i]$, $\forall i \in \{1, 2, \dots, n\}$.

Astfel, am obtinut relatia de recurenta:

$$S[i][j] = T[i][j] + \max\{ S[i+1][j], S[i+1][j+1] \}$$

3.6 Exemplificarea aplicarii algoritmului

Voi prezenta pasii algoritmului pentru exemplul de mai sus:



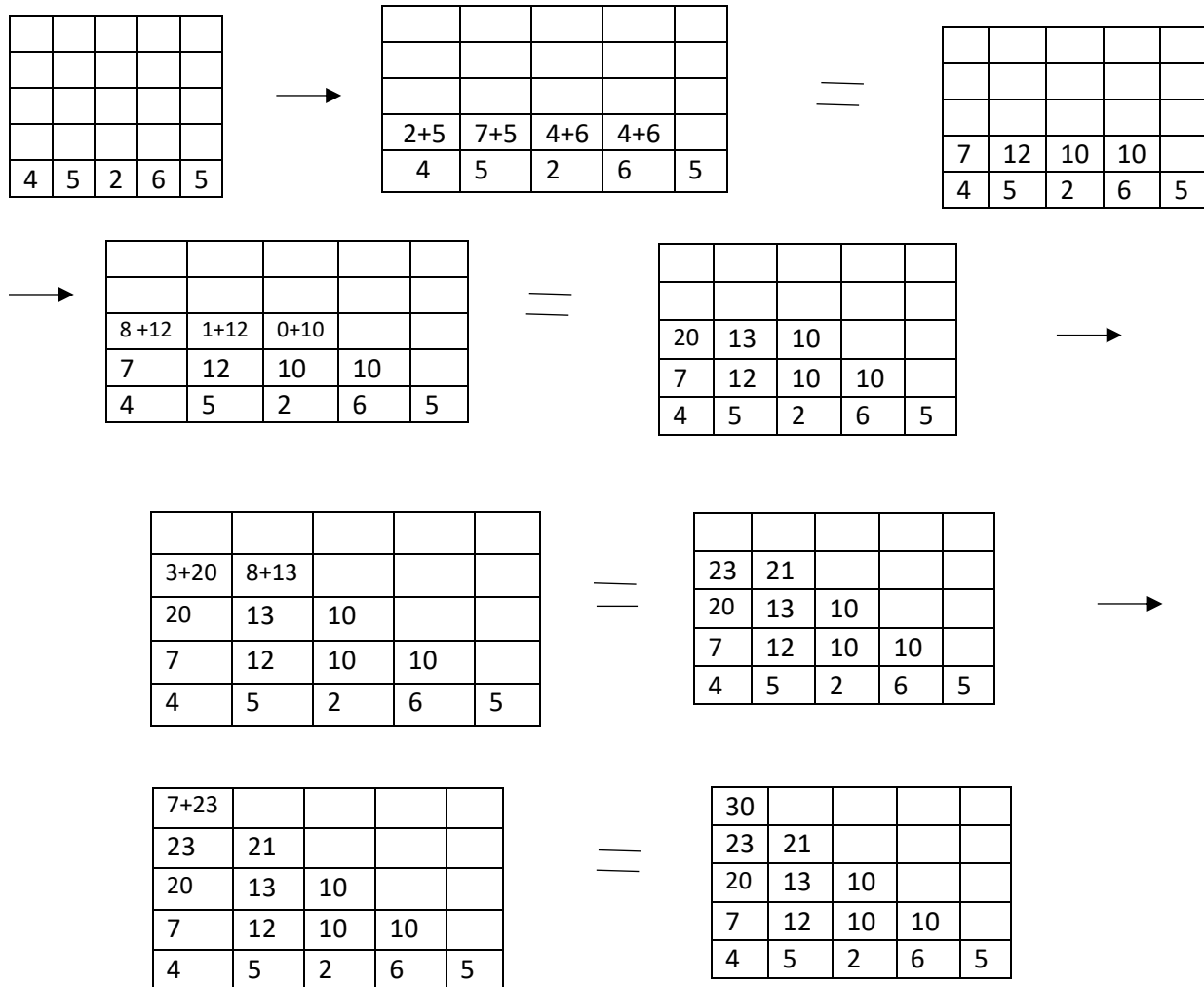
Matricea T corespunzatoare acestuia este :

7				
3	8			
8	1	0		
2	7	4	4	
4	5	2	6	5

Matrice S in care vom prelucra sumele pentru obtinerea rezultatului final este initial plina de 0:

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Transformările prin care trece aceasta, urmand pasii algoritmului pana la solutia finala sunt:



Deci, drumul cu suma maxima pornind din varf pana la ultima linie este $S[1][1] = 30$.

Sursa: <https://muhaz.org/programare-dinamica-descriere-probleme-clasice-probleme-propus.html>

4.Tehnica Backtracking

4.1 *Enuntul problemei*

„Jocul Fazan”: Se citesc dintr-un fisier numerele naturale **n** si **m** si apoi **n** cuvinte distincte cu cel mult 15 litere.

Sa se afiseze toate secvente de cate **m** cuvinte dintre cele citite care sa respecte conditiile jocului "fazan".

4.2 *Descrierea solutiei problemei*

Implementarea algoritmului este realizata in cativa pasi, folosind metoda Backtracking:

- ✚ Implementam regula jocului „Fazan” prin care primele doua litere ale cuvintului curent trebuie sa coincida cu ultimele doua ale cuvintului precedent.
- ✚ Introducem aceasta regula in schema metodei Backtracking. Deci, solutiile finale vor fi construite prin incercari de alegeri de variante. Cand conditia „Fazan” nu va fi respectata, vom reveni la ultima decizie luata si vom incerca alegerea unei alte variante.
- ✚ Cand solutia devine completa, adica atunci cand am ales **m** cuvinte ce respecta regula, o vom afisa.

→ Pentru a tine evidenta indicilor numerelor introduse pana in momentul curent in solutie, ne vom folosi de vectorul boolean `chosen[100]`, iar pentru retinerea in ordine a indicilor elementelor ce alcatuiesc o solutie, ne vom folosi de vectorul `index[100]`.

4.3 *Prezentarea algoritmului de rezolvare a problemei (pseudocod)*

```

char C[100][16] = matricea in care retinem cuvintele date
bool chosen[100] = {false}
int index[100] = {0}

void afisare()
{
    for(int i=1;i<=m;i++)
        scrie C[index[i]]
}

int fazan(char cuv1[], char cuv2[])
{
    if(cuv1[strlen(cuv1)-2]==cuv2[0] && cuv1[strlen(cuv1)-1]==cuv2[1])
        return 1
    return 0
}

void backtracking(int k)
{
    for(int i=1;i<=n;i++)
        if(!chosen[i])
        {
            index[k]=i
            chosen[i]=true
            if(k==1 || fazan(C[index[k-1]], C[index[k]]))
                if(k==m) afisare()
                else backtracking(k+1)
            chosen[i]=0
        }
}

backtracking(1);

```

4.4 Complexitatea temporală a algoritmului

Complexitatea temporală a algoritmului este exponențială.

Avem de format o soluție compusă din m elemente, fiecare poziție putând lua valori dintr-o mulțime de n elemente a.i. condiția Fazan să fie respectată.

Pentru numere n și m mari, complexitatea devine foarte mare.

4.5 Analiza succintă asupra eficienței algoritmului propus

Algoritmul nu este unul eficient, prin prisma folosirii tehnicii Backtracking, care nu este deloc optimă, aceasta fiind o metodă ce ar trebui folosită pentru rezolvarea unei probleme doar dacă nu a fost găsită o altă modalitate de rezolvare care să aibă o complexitate temporală mai bună.

4.6 Exemplificarea aplicării algoritmului

Considerăm ca ni se dau cuvintele: [„real”, „nasture”, „cadou”, „alina”]

$chosen[4] = \{false\}$, $index[4] = \{0\}$, $k = 1$

backtracking(1)

$i = 1 \rightarrow$ intra în if $\rightarrow index[1] = 1$, $chosen[1] = true$, $k = 1$ rezulta:

\rightarrow backtracking(2) $k = 2$

$i = 1$, $chosen[1] = true \rightarrow$ nu intra în if

$i = 2 \rightarrow$ intra în if $\rightarrow index[2] = 2$, $chosen[2] = true$, $fazan(real, nasture) = false$

rezulta $chosen[2] = false$

$i = 3 \rightarrow$ intra în if $\rightarrow index[2] = 3$, $chosen[3] = true$, $fazan(real, cadou) = false$

rezulta $chosen[3] = false$

$i = 4 \rightarrow$ intra în if $\rightarrow index[2] = 4$, $chosen[4] = true$, $fazan(real, alina) = true$

rezulta:

backtracking(3), k = 3

i = 1 , chosen[1] = true → nu intra in if

i = 2 → intra in if → index[3] = 2, chosen[2] = true, fazan(alina, nasture) = true,

k = 3 = m rezulta **afisare(real, alina, nasture)**

chosen[2] = false

i = 3 → intra in if → index[3] = 3, chosen[3] = true, fazan(alina, cadou) = false

rezulta chosen[3] = false

i = 4 , chosen[4] = true → nu intra in if

Ne intoarcem la pasul ramas la backtracking(2) si facem chosen[4] = false, i = 2 si astfel punem pe prima pozitie cuvantul nasture. In aceeaasi maniera se continua pana la sfarsit, gasind solutiile {(alina nasture real), (nasture real alina), (real alina nasture)}.

Sursa: inspirat din <https://info.mcip.ro/?cap=Backtracking&prob=818>



5. Analiza Comparativa

5.1

Pentru o analiza comparativa cat mai explicita, vom porni de la o problema rezolvata anterior, la punctul 3, prin tehnica „**Programarii Dinamice**” : problema drumului de suma maxima intr-un triunghi.

De aceasta data, vom prezenta o rezolvare a problemei, folosind tehnica „**Divide et Impera**”.

Pasii implementarii sunt urmatoarii:

-  Pornim de la elementul de pe prima linie si impartim problema in 2 subprobleme de dimensiuni egale (subprobleme corespunzatoare celor 2 variante pe care putem mereg pornind dintr-un element):
 - determinam suma maxima ce poate fi obtinuta alegand calea prin elementul de sub cel curent ($T[i+1][j]$)
 - determinam suma maxima ce poate fi obtinuta alegand calea prin elementul de sub cel curent, partea dreapta ($T[i+1][j+1]$)
-  Alegem maximul dintre cele doua variante si il adunam la elementul curent, pentru a afla suma maxima corespunzatoare pozitiei curente.

(Acesti doi pasi sunt repetati pana ce ajungem pe ultima linie, cand ne oprim si compunem solutia finala)

Pseudocodul pentru aceasta implementare este:

```
int triangle(int i, int j)
{
    if(i < n)
    {
        int max1 = triangle(i+1, j)
        int max2 = triangle(i+1, j+1)
        return T[i][j] + max(max1, max2)
    }
    return 0;
}
```

Complexitatea temporală a acestui algoritm este exponențială ($O(2^n)$), calculându-se cu relația de recurență dedusă din pseudocod: $T(n) = 2 \cdot T(n-1) + \Theta(1)$.

Ne amintim că prin tehnica Programării Dinamice, obținem o complexitate temporală mult mai bună, $O(n^2)$, asigurând abordarea prin Divide et Impera este una foarte ineficientă. Cauza acestei probleme este suprapunerea subproblemelor, ceea ce duce la calcularea aceluiași valori de mai multe ori.

5.2

În următorul tabel am comparat cele 4 tehnici de programare, subliniind diverse aspecte și încercând să evidențiez anumite avantaje dar și dezavantaje.

Printre aspectele menționate se numără caracteristici ale fiecărei tehnici în parte, dar și precizarea tipurilor de probleme pentru care fiecare dintre acestea poate fi aplicată.

Divide et Impera	Greedy	Programare Dinamica	Backtracking
Pentru probleme ce se pot imparti în mai multe probleme similare problemei initiale, dar de dimensiune mai mica, pe care sa le putem rezolva în mod recursiv si apoi sa le combinam pentru a crea o solutie a problemei initiale	Aplicată, în general, în cazul problemelor de optimizare pentru care atunci când dorim să construim solutia trebuie să determinăm o secvență de actiuni selectate din mai multe variante posibile.	Programarea dinamica este o tehnica de programare care se poate aplica problemelor care constau în rezolvarea unei relatii de recurenta.	Se foloseste în cazul problemelor a caror solutie este un vector $v = (v_1; v_2; \dots; v_n)$, unde fiecare variabila v_i are un domeniu de definitie finit D_i , iar elementele din multimea domeniului se afla într-o relatie de ordine bine stabilita.
Subproblemele sunt independente unele de celelalte		Subproblemele sunt interdependente	
	Nu există o garanție pentru obținerea unei soluții optime.	Este garantat că va genera o soluție optimă folosind Principiul optimității.	
Recursiva	Nerecursiva	Nerecursiva	Recursiva
	Nu genereaza toate solutiile posibile pt o problema		Genereaza toate solutiile posibile pt o problema
Rezolvarea unor probleme prin combinarea solutiilor unor subprobleme de dimensiune mai mica.		Rezolvarea unor probleme prin combinarea solutiilor unor subprobleme de dimensiune mai mica.	
Rezolva o problema de fiecare data când o vor întâlni în cadrul unei descompuneri.		Rezolvă subproblemele o singură dată și apoi stochează → memoizare	

5.3

Divide et Impera

Avantaje

- ✓ Problemele dificilele pote fi rezolvate cu ușurință.
- ✓ Împarte întreaga problemă în subprobleme, astfel încât poate fi rezolvată paralel, folosind multiprocesare.
- ✓ Utilizează eficient memoria cache fără a ocupa mult spațiu
- ✓ Reduce complexitatea în timp a problemei

Dezavantaje

- Implică recursivitate care este uneori lentă
- Eficiența depinde de implementarea logicii

Greedy

Avantaje

- ✓ A lua întotdeauna cea mai bună alegere disponibilă este de obicei ușor. De obicei, necesită sortarea opțiunilor.
- ✓ A lua în mod repetat următoarea cea mai bună alegere disponibilă este de obicei o muncă liniară. Dar nu uitați de costul sortării opțiunilor.
- ✓ Mult mai ieftin decât căutarea exhaustivă. Mult mai ieftin decât majoritatea altor algoritmi.

Dezavantaje

- Uneori, algoritmi lacomi nu reușesc să găsească soluția optimă la nivel global, deoarece nu iau în considerare toate datele.
- Alegerea făcută de un algoritm lacom poate depinde de alegerile pe care le-a făcut până acum, dar nu este conștient de alegerile viitoare pe care le-ar putea face.

Programare Dinamica

Avantaje

- ✓ Poate obtine atat optimul local cat si cel global din punct de vedere al solutiei.
- ✓ Accelerează procesarea, deoarece sunt utilizate referințele care erau calculate anterior.
- ✓ Deoarece este o tehnică de programare recursivă, reduce liniile de cod din program.

Dezavantaje

- Este necesară multă memorie pentru a stoca rezultatul calculat al fiecărei subprobleme, fără a putea garanta că valoarea stocată va fi folosită sau nu.
- De multe ori, valoarea de ieșire este stocată fără a fi folosită vreodată în următoarele subprobleme în timpul execuției. Acest lucru duce la utilizarea inutilă a memoriei.
- În programarea dinamică funcțiile sunt numite recursiv. Acest lucru menține memoria stivei în continuă creștere.

Backtracking

Obs: Tehnica Backtracking ar trebui sa fie folosita pentru rezolvarea unei probleme doar daca nu a fost gasita o alta modalitate de rezolvare care sa aiba o complexitate temporala mai buna.

Avantaje

- ✓ Codul este foarte intuitiv. Este aproape ca și cum un copil mic încearcă să rezolve problema.
- ✓ Este ușor de inteles si implementat.

Dezavantaje:

- De cele mai multe ori exista algoritmi mai optimi pentru o problema.
- Foarte ineficient în timp în multe cazuri (când factorul de ramificare este mare)
- Complexitate spatiala mare