Metoda
Programării Dinamice

Exemplu

Mulțime independentă de pondere maximă

Se consideră vectorul $w = (w_1,..., w_n)$.

Să se determine un subşir al lui w care nu conține elemente consecutive în w și are suma elementelor maximă

Exemplu

Pentru

$$w = (1, 4, 7, 5)$$

soluția este

$$(4, 5)$$

Mulțime independentă de pondere maximă

- **Problemă echivalentă**

Se consideră un graf de tip lanț cu $V = \{v_1, \ldots, v_n\}$.

Vârfurile grafului au asociate ponderile $w_1, \ldots,$ respectiv w_n.

Să se determine o mulțime independentă de vârfuri de pondere maximă

- ponderea unei mulțimi de vârfuri = suma ponderilor vârfurilor

Mulțime independentă de pondere maximă

- **Aplicații** – probleme de alocare de resurse cu evitarea interferenței (indicată prin muchii -> graf de conflicte)

  Ponderea asociată vârfurilor poate reprezenta cantitatea de date pe care stația trebuie să o transmită

  Problemă – transmiterea cantității maxime de date fără interferențe

- Nu se cunosc algoritmi polinomiali în cazul general (NP-completitudine)

- **Maximum Weighted Independent Set**

Mulțime independentă de pondere maximă

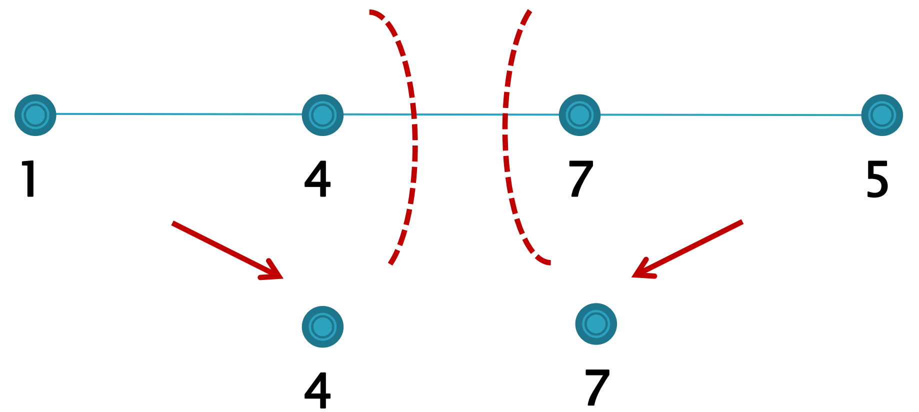
- **Abordare cu metoda Greedy**

La un pas – este adăugat în soluție vârful de pondere maximă neadiacent cu cele deja selectate

Soluția: $\{v_3, v_1\}$ cu ponderile $\{7, 1\}$ – **nu este optimă**

Mulțime independentă de pondere maximă

- **Abordare cu metoda Divide et Impera**



Incorect – reuniunea soluțiilor subproblemelor nu este o soluție posibilă (corectă) pentru problema inițială

Mulțime independentă de pondere maximă

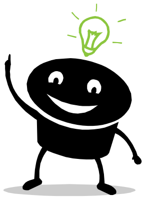
- **Căutarea exhaustivă a soluției optime**

Generarea tuturor soluțiilor posibile și determinarea celei optime – algoritm exponențial

Mulțime independentă de pondere maximă

Problema nu se poate rezolva folosind metode deja studiate

Mulțime independentă de pondere maximă



Analizăm structura unei soluții optime, evidențiind un element (primul/ultimul) al acesteia, pentru a determina subprobleme utile și relații de recurență

Mulțime independentă de pondere maximă

Fie $S \subseteq V = \{v_1,\ldots, v_n\}$ o soluție optimă

- Dacă $v_n \in S \Rightarrow S-\{v_n\}$ este soluție optimă pentru

  $G-\{v_n, v_{n-1}\}$

Mulțime independentă de pondere maximă

Fie $S \subseteq V = \{v_1,\ldots, v_n\}$ o soluție optimă

- Dacă $v_n \in S \Rightarrow S-\{v_n\}$ este soluție optimă pentru

  $G-\{v_n, v_{n-1}\}$

- Dacă $v_n \notin S \Rightarrow S$ este soluție optimă pentru

  $G-\{v_n\}$

Mulțime independentă de pondere maximă

Fie $S \subseteq V = \{v_1,\ldots, v_n\}$ o soluție optimă

- Dacă $v_n \in S \Rightarrow S-\{v_n\}$ este soluție optimă pentru

  $G-\{v_n, v_{n-1}\}$

- Dacă $v_n \notin S \Rightarrow S$ este soluție optimă pentru

  $G-\{v_n\}$

- **Dacă am ști deja soluțiile pentru grafurile $G-\{v_n, v_{n-1}\}$ și $G-\{v_n\}$, am putea determina S alegând dintre cele două situații cazul în care se obține soluția optimă**

Mulțime independentă de pondere maximă

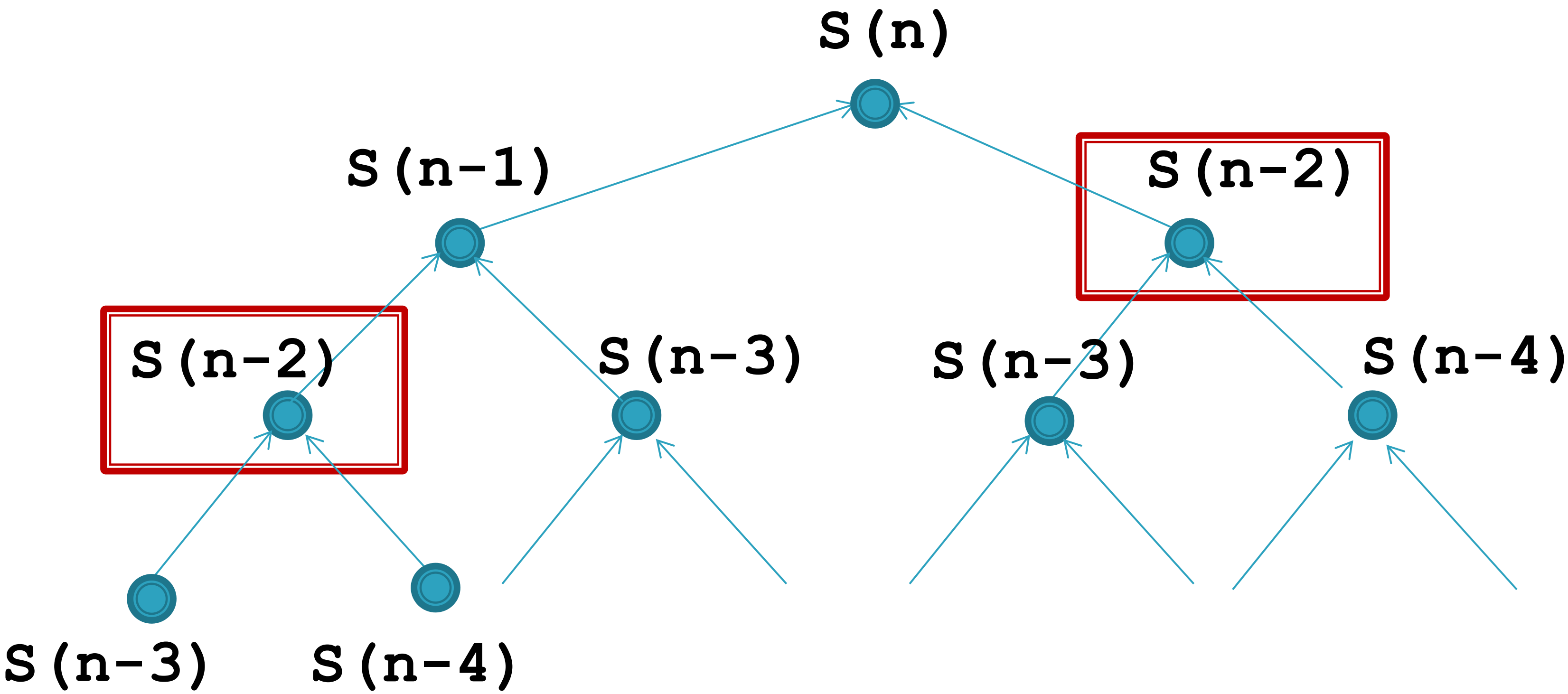
Recurență:

- Notăm $S(i)$ = ponderea maximă a unei mulțimi independete în graful indus de vârfurile $\{v_1, \ldots, v_i\}$

- **$S(n) = \max\{ S(n-2)+w_n, S(n-1)\}$**

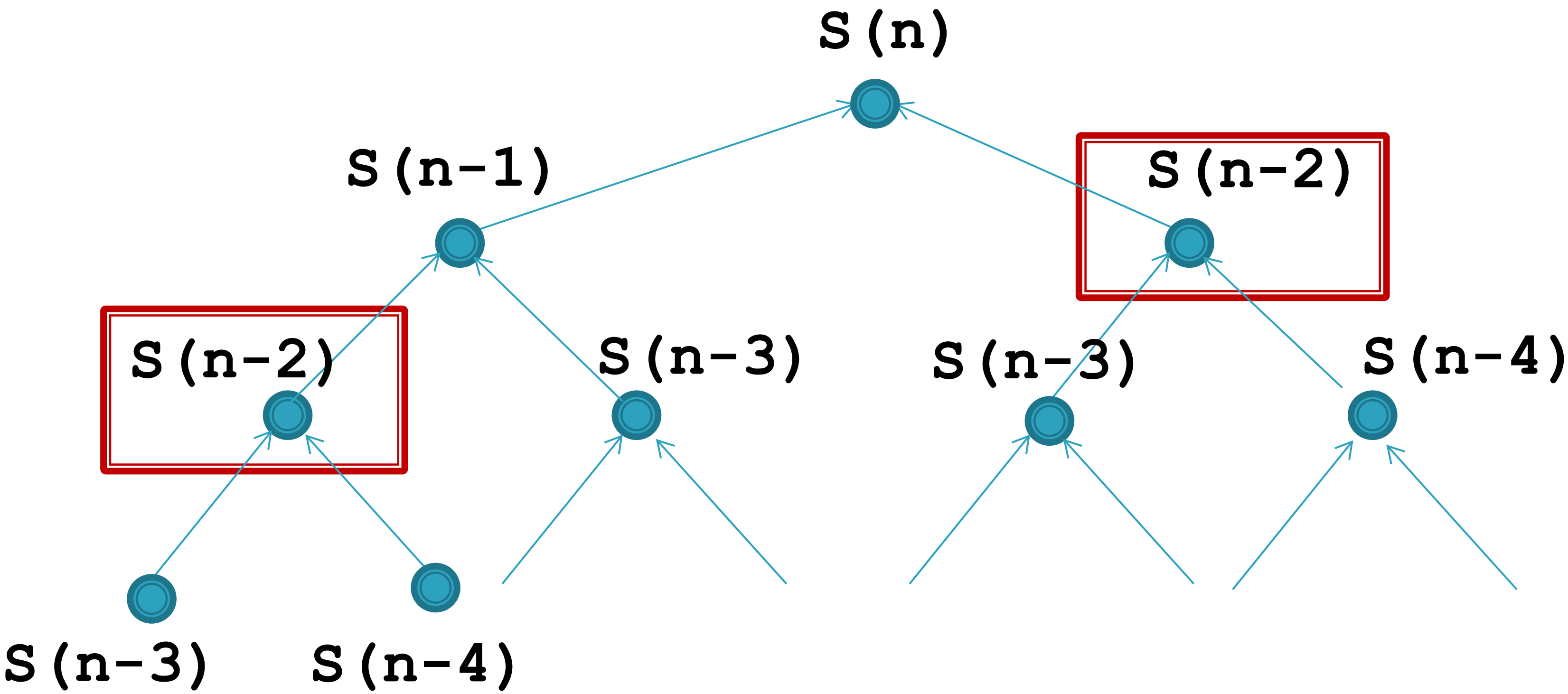
- **$S(1) = w_1$, $S(0) = 0$**

Mulțime independentă de pondere maximă



Subproblemele se repetă – algoritm exponențial

Mulțime independentă de pondere maximă



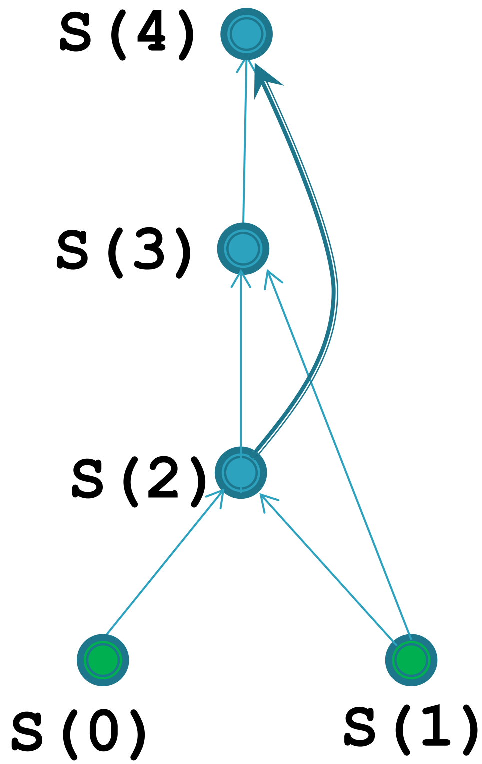
Putem evita rezolvarea unei subprobleme de mai multe ori?

Mulțime independentă de pondere maximă

Recurență:

- **$S(n) = \max\{ S(n-2)+w_n, S(n-1)\}$**
- **Memorăm într-un vector rezultatele subproblemelor deja rezolvate (memoizare)** $\Rightarrow$ o subproblemă va fi rezolvată o singură dată – algoritm O(n)

Mulțime independentă de pondere maximă



Mulțime independentă de pondere maximă

- **Variantă**– implementare iterativă a recurenței

  (bottom-up)

Implementare recursivă – memoizare

```
void sol(int n){
    if(n==0) {s[0]=0;return;}
    if(n==1) {s[1]=w[1]; return ;}
    if (s[n-1]==0) //nerezolvata
          sol(n-1);
    if (s[n-2]==0) //nerezolvata
          sol(n-2);
    s[n]= Math.max(s[n-2]+w[n],s[n-1]);
}
```

```
for(int i=0;i<=n;i++)
      s[i]=0; //initial-nerezolvate pt n>1
```

Implementare iterativă

```
int SolNerec(int n){
        s[0] = 0;
        s[1] = w[1];
        for(int i=2;i<=n;i++)
                s[i]= Math.max(s[i-2]+w[i],s[i-1]);
        return s[n];
 }

 int solNerecFaraVector(int n){ //similar Fibonacci
         if(n==0) return 0;
         int i,s0,s1,si;
         s0=0;  s1=w[1];
         for(i=2;i<=n;i++){
                si= Math.max(s0+w[i],s1);
                s0=s1; s1=si;
         }
         return s1;
  }
```

Mulțime independentă de pondere maximă

Cum putem determina și o submulțime optimă, nu doar ponderea?

Mulțime independentă de pondere maximă

Cum putem determina și o submulțime optimă, nu doar ponderea?

- Din relația de recurență putem deduce ce vârfuri au fost selectate în soluție

$$s[n]= \max\{ s[n-2]+w[n],\ s[n-1]\}$$

Mulțime independentă de pondere maximă

Cum putem determina și o submulțime optimă, nu doar ponderea?

- Din relația de recurență putem deduce ce vârfuri au fost selectate în soluție

$$s[n]= \max\{ s[n-2]+w[n],\ s[n-1]\}$$

  - Dacă $\mathbf{s[n] = s[n-2]+w[n]}$, vârful n se adaugă în soluție și problema se reduce la primele n-2 vârfuri
  - Dacă $\mathbf{s[n] = s[n-1]}$, nu se adaugă nici un vârf la soluție și problema se reduce la primele n-1 vârfuri

```
void afisRec(int s[],int n){
    if(n==0) return;
    if(n==1){
        System.out.println(n+" de pondere "+w[n]);
        return;
    }
    if(s[n]==s[n-2]+w[n]){
        afisRec(s,n-2);
        System.out.println(n+" de pondere "+w[n]);
    }
    else
        afisRec(s,n-1);
}
```

Mulțime independentă de pondere maximă

TEMĂ – rezolvaţi problema pentru un arbore oarecare O(n)

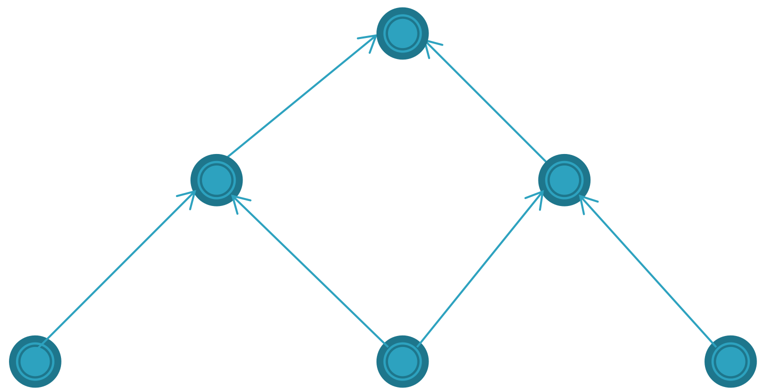
Concluzii

Dezavantaje ale metodelor deja studiate

- Greedy – nu furnizează mereu soluția optimă
- Alte exemple:
  - Problema rucsacului, cazul discret
  - Problema monedelor, cazul general

Dezavantaje ale metodelor deja studiate

- Divide et Impera – ineficientă dacă subproblemele se repetă



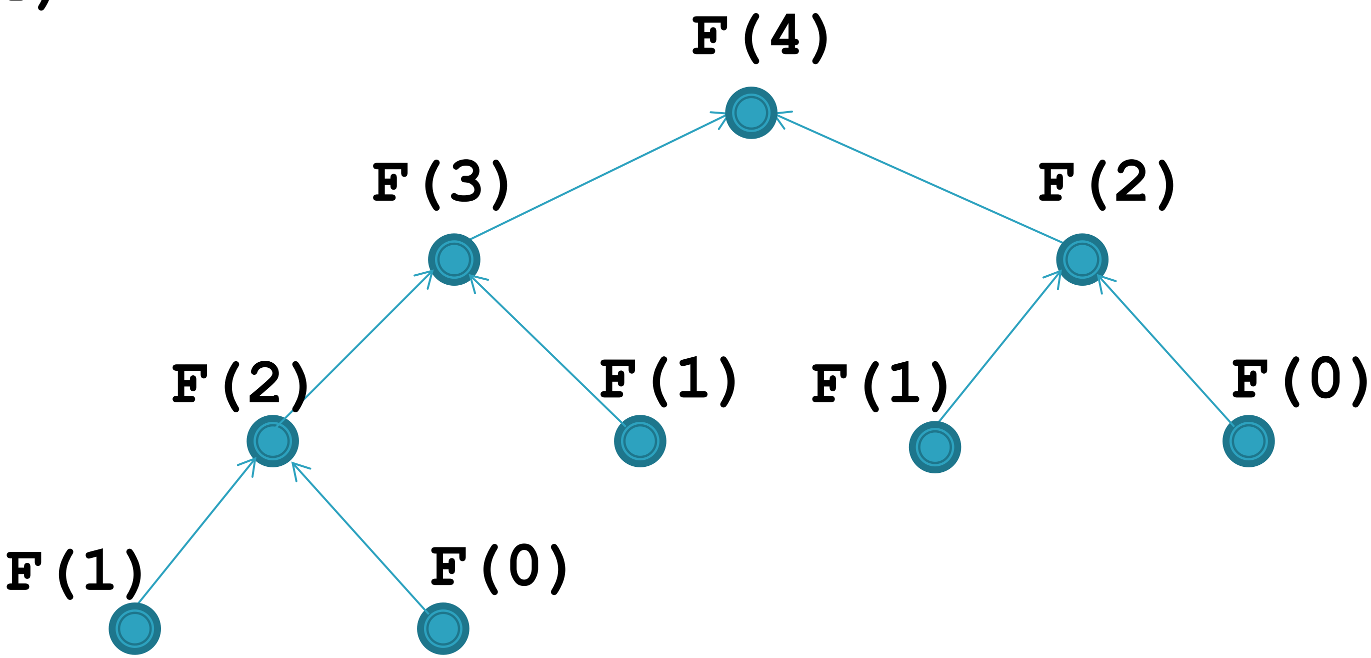
Dezavantaje ale metodelor deja studiate

- **Exemplu** – Calculăm numărul Fibonacci `F(n)`

  `F(n) = F(n-1)+F(n-2)`

  `F(0) = F(1) = 1`

- **`F(4)`**



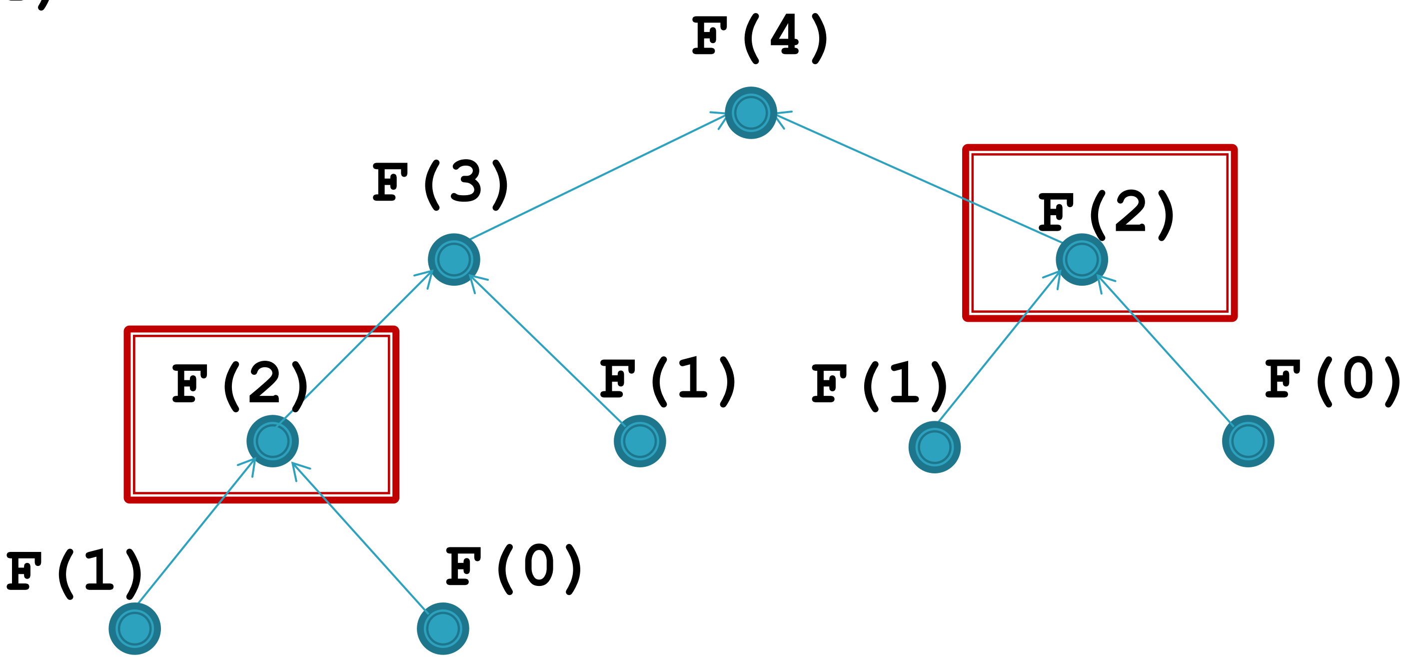
Dezavantaje ale metodelor deja studiate

- **Exemplu –** Calculăm numărul Fibonacci `F(n)`

  `F(n) = F(n-1)+F(n-2)`

  `F(0) = F(1) = 1`

- **`F(4)`**



Dezavantaje ale metodelor deja studiate

- **Soluții**
  - **reducere la subprobleme utile + relații de recurență**
  - **rezolvarea eficientă a subproblemelor**
    - recursiv cu memoizare (salvarea rezultatelor subproblemelor deja rezolvate)
    - algoritmi iterativi buttom-up

↓

Metoda programării dinamice

Prezentarea metodei

Cadru general

- Probleme care presupun rezolvarea de relații de recurență
- De obicei aceste relații se obțin din respectarea unui principiu de optimalitate (subprobleme optime)

Cadru general

- Fie A şi B două mulţimi oarecare (B = **N**, **Z**, **R**, {0,1} …)

  Fiecărui $x \in A$ **urmează să i se asocieze** o valoare $v(x) \in B$.

  $$\mathbf{x \in A \longrightarrow v(x) \in B}$$

Cadru general

- Fie A şi B două mulţimi oarecare (B = **N**, **Z**, **R**, {0,1} …)

  Fiecărui $x \in A$ **urmează să i se asocieze** o valoare $v(x) \in B$.

  $$\mathbf{x \in A \longrightarrow v(x) \in B}$$

  - v este **cunoscută** doar pe submulţimea $\mathbf{X \subset A}$

Cadru general

- Fie A şi B două mulţimi oarecare (B = **N**, **Z**, **R**, {0,1} ...)

  Fiecărui $x \in A$ **urmează să i se asocieze** o valoare $v(x) \in B$.

  $$\mathbf{x \in A \longrightarrow v(x) \in B}$$

  - v este **cunoscută** doar pe submulţimea $\mathbf{X \subset A}$

  - Pentru fiecare $\mathbf{x \in A \setminus X}$ avem relația

    $$\mathbf{v(x) = f_x(v(a_1),\ldots,v(a_k))}$$

    unde

    $$\mathbf{A_x = \{a_1,\ldots,a_k\}}$$

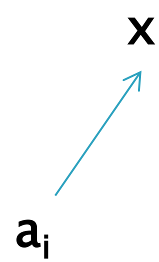
    este mulţimea elementelor din A de a căror valoare depinde v(x)

Cadru general

- Dat $z \in A$, se cere să se calculeze, **dacă este posibil**, valoarea $v(z)$ – **eficient**

Cadru general

- Putem reprezenta problema pe un *graf de dependenţe*. Vârfurile corespund elementelor din A, iar descendenţii unui vârf x sunt vârfurile din A_x.



valoarea lui x depide

direct de v(a_i)

- Problema are soluție numai dacă în graful de dependențe **nu există circuite accesibile din z**

Graf de dependenţe – Exemplu

- Calcul număr Fibonacci F_n

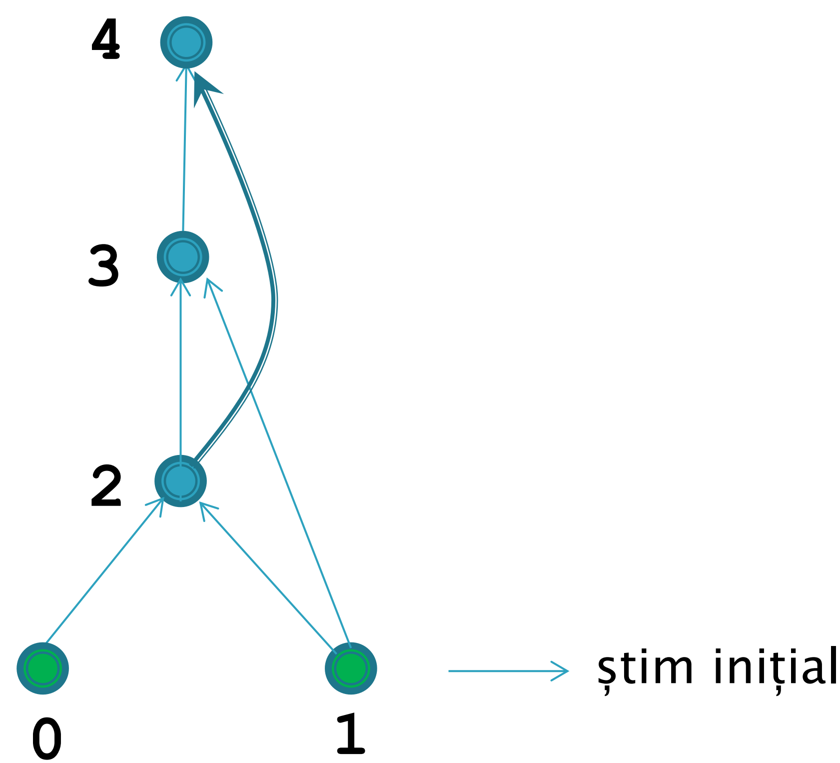
```
F(n) = F(n-1)+F(n-2)
F(0) = F(1) = 1
```

Graf de dependenţe

- Calcul număr Fibonacci F_n

$$F(n) = F(n-1)+F(n-2)$$

$$F(0) = F(1) = 1 \Longrightarrow X = \{0,\ 1\}$$



Graf de dependenţe

- **Alt exemplu**

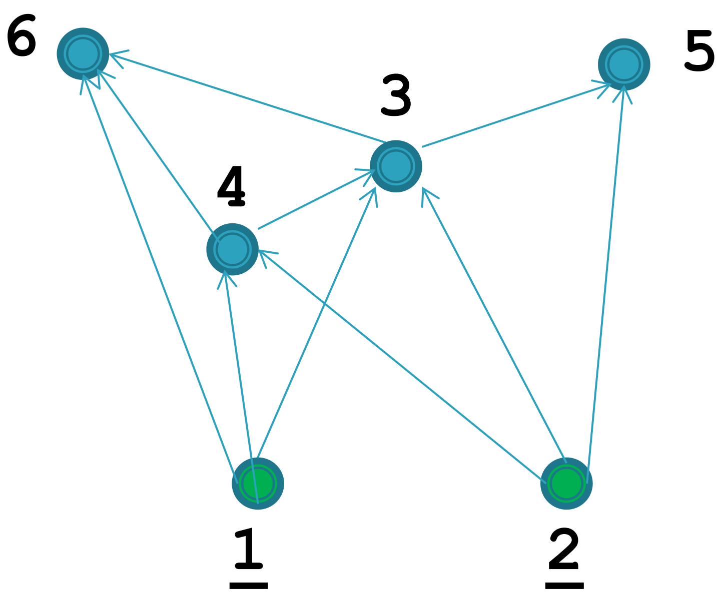
```
A = {1,2,...,5,6};
v(1) = v(2) = 1
v(3) = v(1) + v(2) + v(4)
v(4) = v(1) + v(2)
v(5) = v(2) + v(3)
v(6) = v(1) + v(3) + v(4)
```

`v(6) = ?`

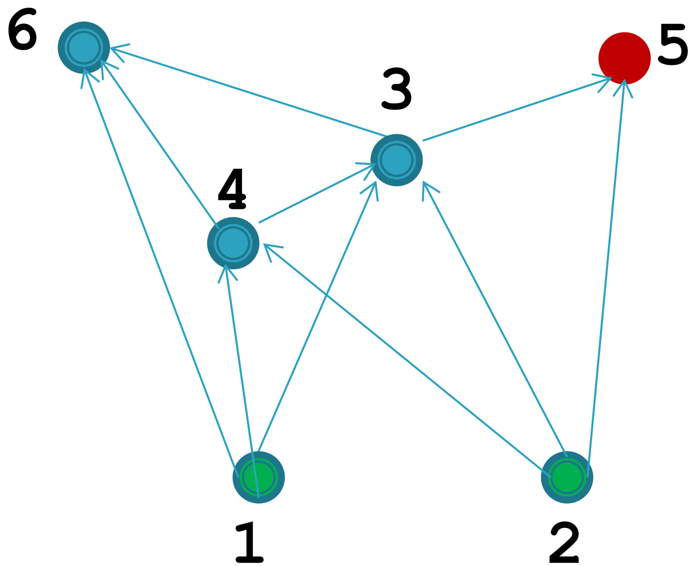
Graf de dependenţe

- **Alt exemplu**

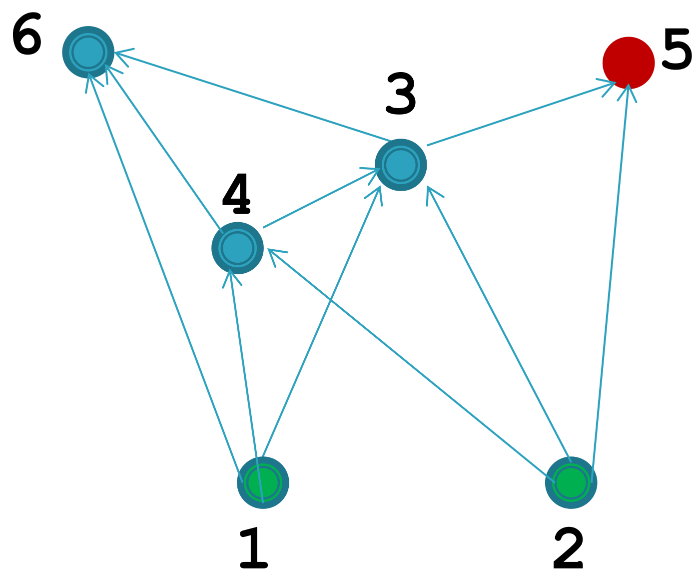
```
A = {1,2,...,5,6};
v(1) = v(2) = 1        ⟹ X = {1, 2}
v(3) = v(1) + v(2) + v(4)
v(4) = v(1) + v(2)
v(5) = v(2) + v(3)
v(6) = v(1) + v(3) + v(4)
```



$v(6) = ?$



$v(6) = ?$



nu intervine în calculul lui v(6)

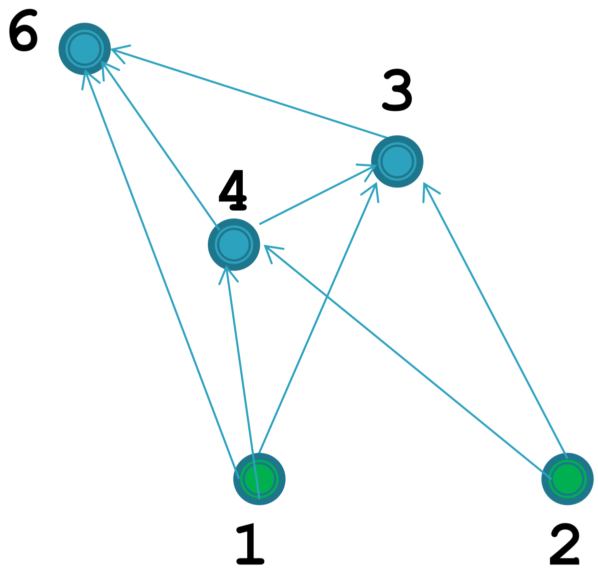
Cadru general

- Fie G_z graful indus de mulţimea vârfurilor **y de a căror valoare depinde v(z)** = pentru care există un drum de la y la z = **vârfuri observabile** din z

  G_z = **graful vârfurilor observabile** din z

- Problema presupune o **parcurgere a grafului G_z**

v(6) = ?



graful vârfurilor observabile din 6

Cadru general

- Ar fi bine dacă
  - am cunoaşte de la început G_z
  - forma acestui graf ar permite o **parcurgere mai simplă**, care să conducă la calcularea valorii v(z).

Cadru general

- Încercare de rezolvare cu metoda Divide et Impera.

```
procedure DivImp(x)
    for y∈Ax\X  {y de care x depinde direct}
        DivImp(y)
    calculează v(x) conform funcţiei fx
end;
```

Cadru general

- Încercare de rezolvare cu metoda Divide et Impera.

```
procedure DivImp(x)
    for y∈Ax\X  {y de care x depinde direct}
        DivImp(y)
    calculează v(x) conform funcţiei fx
end;
```

  - **algoritmul nu se termină pentru grafuri ciclice**
  - **valoarea unui vârf poate fi calculată de mai multe ori**

Cadru general

- **Soluție** – sortarea topologică pentru G_z $\Longrightarrow$ ordinea în care se calculează valorile v ale vârfurilor
  - **Ar fi mai bine dacă forma grafului ar permite o parcurgere mai simplă.**

Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**

Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**

Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**
  - În graf este pus în evidenţă un graf de vârfuri **observabile** din z, numit **PD-arbore de rădăcină z**, cu proprietățile

Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**
  - În graf este pus în evidenţă un graf de vârfuri **observabile** din z, numit **PD-arbore de rădăcină z**, cu proprietățile
    - ✓nu conţine circuite

Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**
  - În graf este pus în evidenţă un graf de vârfuri **observabile** din z, numit **PD-arbore de rădăcină z**, cu proprietățile
    - ✓ nu conţine circuite
    - ✓ mulțimea vârfurilor cu gradul intern 0 (ale căror valori nu depind de o altă valoare) este inclusă în X

Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**
  - În graf este pus în evidenţă un graf de vârfuri **observabile** din z, numit **PD-arbore de rădăcină z**, cu proprietățile
    - ✓ nu conţine circuite
    - ✓ mulțimea vârfurilor cu gradul intern 0 (ale căror valori nu depind de o altă valoare) este inclusă în X
    - ✓ **se poate așeza pe niveluri** (nivelul lui y = lungimea maximă a unui drum de la y la z)

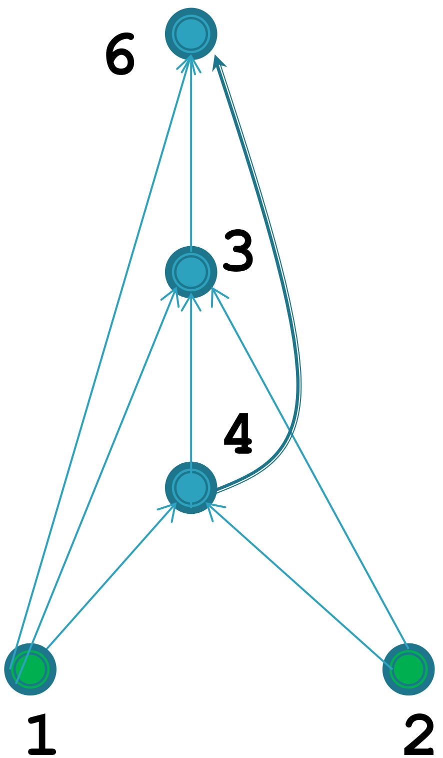
Metoda programării dinamice

- **Metoda programării dinamice constă în următoarele**:
  - Se asociază problemei un **graf de dependenţe**
  - În graf este pus în evidenţă un graf de vârfuri **observabile** din z, numit **PD-arbore de rădăcină z**
  - Se parcurge PD-arborele în postordine generalizată

```
procedure postord(x)
    for j∈A_x
        if viz[j]=false {diferența față de DI}
            postord(j)
    calculează v(x) conform funcţiei f_x;
    viz[x]←true
end
```

- Apel `postord(z)`

$v(6) = ?$



- Prin parcurgerea în postordine generalizată, vârfurile vor fi **sortate topologic**

Metoda programării dinamice

- **Generalizează metoda Divide et Impera** – dependenţele nu au forma unui arbore, ci a unui PD-arbore.

Metoda programării dinamice

- **Generalizează metoda Divide et Impera** – dependenţele nu au forma unui arbore, ci a unui PD-arbore.
- Este util să căutăm în PD-arbore regularităţi care să evite memorarea valorilor tuturor vârfurilor şi/sau să simplifice parcurgerea în postordine.

Exemplu – Fibonacci

- A = {0,...,n}, B = **N**
- **X = {0,1}** (ştim $F_0=0$; $F_1=1$)
- v(k) = F_k , deci
  - **v(k) = v(k-1)+ v(k-2)**

Exemplu – Fibonacci

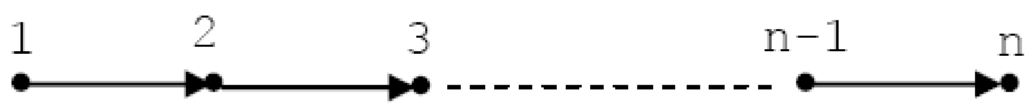
- $A = \{0,...,n\}, \quad B = \mathbf{N}$
- $\mathbf{X = \{0,1\}}$ (ştim $F_0=0$; $F_1=1$)
- $v(k) = F_k$, deci
  - $\mathbf{v(k) = v(k-1)+ v(k-2)}$
  - $A_k = \{k-1,k-2\}, \forall k\geq 2$
  - $f_k(a,b) = a + b, \forall k\geq 2$

Exemplu – Fibonacci , varianta 2

- $A = \{1,...,n\}$, $B = \mathbf{N} \times \mathbf{N}$
- $\mathbf{v(k) = (F_{k-1}, F_k)}$
- $v(1) = (0, 1)$

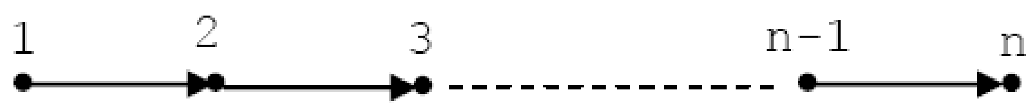
Exemplu – Fibonacci , varianta 2

- $A = \{1, \ldots, n\}, \quad B = \mathbf{N} \times \mathbf{N}$
- $\mathbf{v(k) = (F_{k-1}, F_k)}$
- $v(1) = (0, 1)$
  - $A_k = \{k-1\}, \ \forall k \geq 2$
  - $\mathbf{f_k(a,b) = (b, a+b)} \ \forall k \geq 2$
  - $\mathbf{v(k) = f_k(v(k-1))}$



Exemplu – Fibonacci , varianta 2

- $A = \{1,\ldots,n\}$, $B = \mathbf{N} \times \mathbf{N}$
- $\mathbf{v(k) = (F_{k-1}, F_k)}$
- $v(1) = (0, 1)$
  - $A_k = \{k-1\}$, $\forall k \geq 2$
  - $\mathbf{f_k(a,b) = (b, a+b)}$ $\forall k \geq 2$
  - $\mathbf{v(k) = f_k(v(k-1))}$




```
a←0; b←1
for i=2,n
    (a,b)←(b,a+b)
write(b)
```

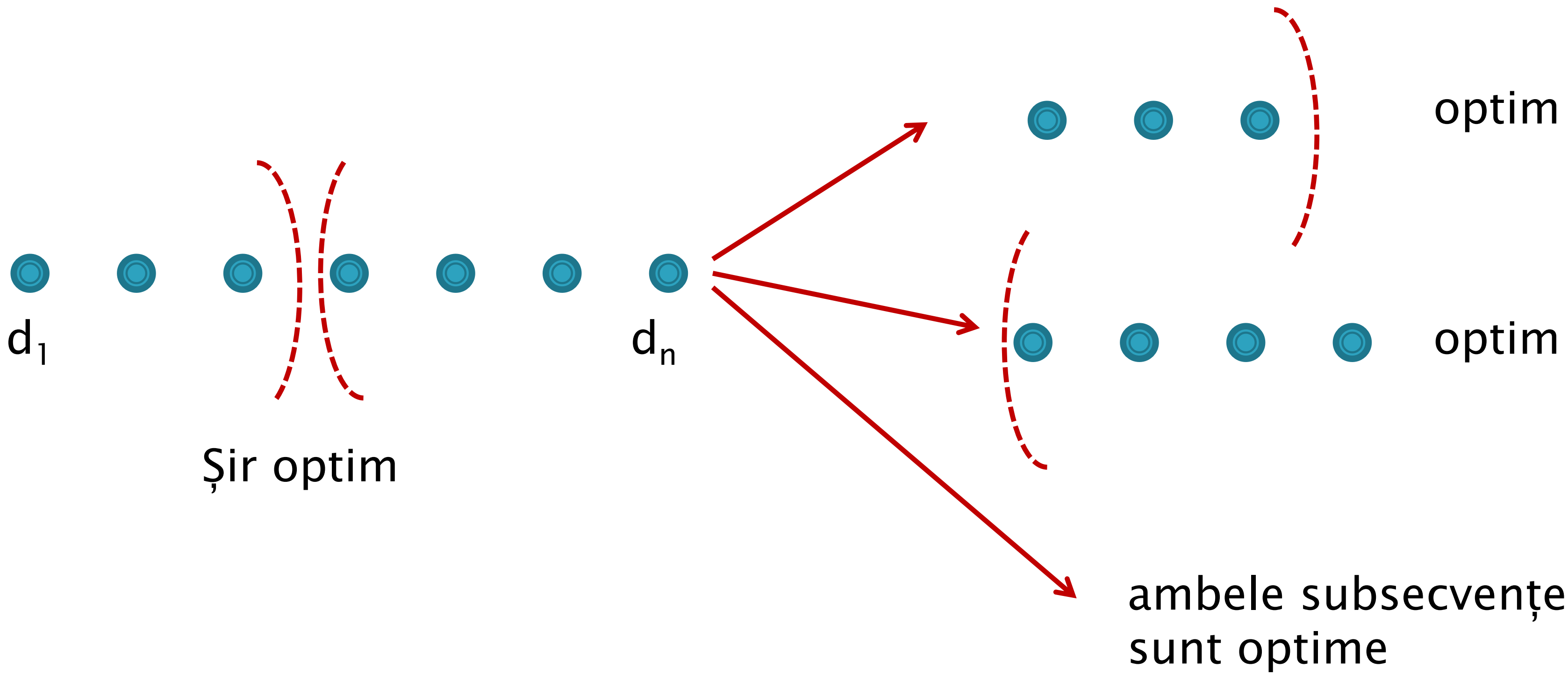
Metoda programării dinamice

- Se poate utiliza în **problemele de optim** – care verifică un **principiu de optimalitate**, din care se obţin relaţiile de calcul

Metoda programării dinamice

Fie soluția optimă $d_1, \ldots, d_n$

Principiul de optimalitate poate fi satisfăcut sub una din următoarele forme:



Metoda programării dinamice

Fie soluția optimă $d_1, ..., d_n$

Principiul de optimalitate poate fi satisfăcut sub una din următoarele forme:

(1) $d_1, d_2, ..., d_n$ optim $\Rightarrow$ $d_k, ..., d_n$ optim pentru subproblema corespunzatoare, $\forall\ 1 \le k \le n$

(2) $d_1, d_2, ..., d_n$ optim $\Rightarrow$ $d_1, ..., d_k$ optim, $\forall\ 1 \le k \le n$

(3) $d_1, d_2, ..., d_n$ optim $\Rightarrow$ $d_1, ..., d_k$ optim, $\forall\ 1 \le k \le n$

si

$d_k, ..., d_n$ optim, $\forall\ 1 \le k \le n$

Etape

- **Stabilirea subproblemelor utile** (de exemplu din principiul de optimalitate)
- **Cum putem rezolva problema iniţială folosind subproblemele**
- **Care subprobleme le putem rezolva direct**
- **Relațiile de recurență**
- **Ordinea de parcurgere a PD-arborelui (ordinea de calcul)**

Exemple

Subşir crescător de lungime maximă

Se consideră vectorul a = (a_1,..., a_n).

Să se determine lungimea maximă a unui subşir crescător din a şi un astfel de subşir de lungime maximă

<u>Exemplu</u>

Pentru

a = (8, 1, 7, 4, 6, 5, 11)

lungimea maximă este **4**, un subșir fiind

1, 4, 6, 11

Subşir crescător de lungime maximă

- **Longest Increasing Subsequence**
- **Înrudită cu problema determinării celui mai lung subșir comun a două șiruri** (Longest Common Subsequence)
- **Aplicații**
  - cautarea de tiparuri (patterns):

    baze de date mari

    bioinformatica
  - similitudini în genetică (ADN)

    sequence alignment

- Lavanya, B., Murugan, A.: Discovery of longest increasing subsequences and its variants using DNA operations. International Journal of Engineering and Technology 5(2), 1169–1177 (2013)

Subşir crescător de lungime maximă

Principiu de optimalitate:

Dacă

$$a_{i1},\ a_{i2},\ \dots,\ a_{ip},$$

este un subşir optim care începe pe poziția `i1`, atunci:

$$a_{i2},\ \dots,\ a_{ip}$$

este un subşir optim care începe pe poziția `i2`;

Mai general

$$a_{ik},\ \dots,\ a_{ip}$$

este un subşir optim care începe pe poziția `ik`.

Subşir crescător de lungime maximă

Principiu de optimalitate

↓

Subprobleme:

Calculăm pentru fiecare poziție i lungimea maximă a unui subşir crescător ce <u>**începe**</u> **pe poziția i** (cu elementul a_i)

Subşir crescător de lungime maximă

- **Subproblemă:**

  `lung[i]` = lungimea maximă a unui subşir crescător ce începe pe poziția i

Subşir crescător de lungime maximă

- **Subproblemă:**

  `lung[i]` = lungimea maximă a unui subşir crescător ce începe pe poziția i

- **Soluție problemă:**

Subşir crescător de lungime maximă

- **Subproblemă:**

lung[i] = lungimea maximă a unui subşir crescător ce începe pe poziția i

- **Soluție problemă:**

$$\mathbf{nr} = \max\{\mathbf{lung[i]} \mid \mathbf{i} = 1,2,\ldots,\mathbf{n}\}$$

Subşir crescător de lungime maximă

- **Subproblemă:**

  `lung[i]` = lungimea maximă a unui subşir crescător ce începe pe poziția i

- **Știm direct**

- **Relație de recurență**

- **Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)**

Subşir crescător de lungime maximă

- **Subproblemă:**

  `lung[i]` = lungimea maximă a unui subşir crescător ce începe pe poziția i

- **Știm direct** `lung[n] = 1`

- **Relație de recurență**

- **Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)**

Subşir crescător de lungime maximă

- **Subproblemă:**

  `lung[i]` = lungimea maximă a unui subşir crescător ce începe pe poziția i

- **Ştim direct** `lung[n] = 1`

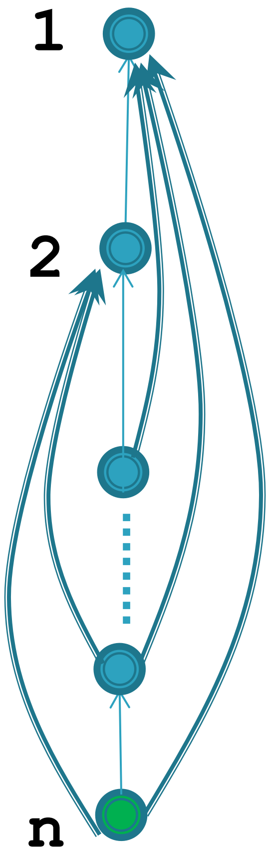
- **Relație de recurență**

  $$\texttt{lung[i]} = 1 + \max\{\texttt{lung[j]} \mid j>i,\ a_i<a_j\}$$

- **Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)**

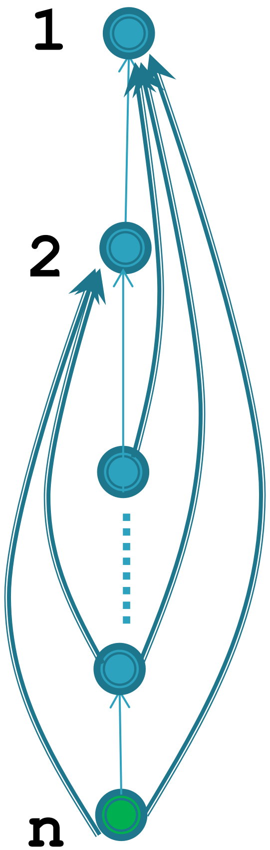
Subşir crescător de lungime maximă

- **Graful de dependenţe**



Subşir crescător de lungime maximă

- Grafful de dependenţe



- Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)

`i = n, n-1, … , 1`

Subşir crescător de lungime maximă

- Cum determinăm un subşir maxim?

Subşir crescător de lungime maximă

- Pentru a determina și **un subșir optim** putem memora în plus

`succ[i]` = indicele următorului element dintr-un subșir optim care începe pe poziția i (**n+1** dacă nu există)

= **indicele pentru care se realizează maximul în relaţia de recurenţă**

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :							
succ :							

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :							1
succ :							8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :							1
succ :							8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :						2	1
succ :						7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :						2	1
succ :						7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :					2	2	1
succ :					7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :					2	2	1
succ :					7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :				3	2	2	1
succ :				5	7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :				3	2	2	1
succ :				5	7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :			2	3	2	2	1
succ :			7	5	7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :			2	3	2	2	1
succ :			7	5	7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :		4	2	3	2	2	1
succ :		4	7	5	7	7	8

Subşir crescător de lungime maximă

	1	2	3	4	5	6	7
a:	8	1	7	4	6	5	11
lung :		4	2	3	2	2	1
succ :		4	7	5	7	7	8

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	2	4	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Subşir crescător de lungime maximă

	1	2	3	4	5	6	7
a:	8	1	7	4	6	5	11
lung :	2	**4**	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Soluţie: lung = 4

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	2	4	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Subşir: 1,

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	2	4	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Subşir: 1,

Subşir crescător de lungime maximă

	1	2	3	4	5	6	7
a:	8	1	7	4	6	5	11
lung :	2	4	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Subşir: 1, 4,

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	2	4	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Subşir: 1, 4, 6

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	2	**4**	2	3	2	2	1
succ :	7	4	7	5	7	7	8

Subşir: 1, 4, 6, 11

Subşir crescător de lungime maximă

Algoritm

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1; succ[n] = n+1; //stim
```

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1; succ[n] = n+1; //stim
for (int i=n-1;i>=1;i--){//ordine de calcul










}
```

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1; succ[n] = n+1; //stim
for (int i=n-1;i>=1;i--){//ordine de calcul
    succ[i]= n+1; lung[i]=1;
    //formula de recurenta









}
```

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1; succ[n] = n+1; //stim
for (int i=n-1;i>=1;i--){//ordine de calcul
    succ[i]= n+1; lung[i]=1;
    //formula de recurenta
    for(int j=i+1;j<=n; j++){




    }

}
```

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1; succ[n] = n+1; //stim
for (int i=n-1;i>=1;i--){//ordine de calcul
   succ[i]= n+1; lung[i]=1;
   //formula de recurenta
   for(int j=i+1;j<=n; j++){
     if((a[i]<a[j]) && (1+lung[j]>lung[i])){
          lung[i]= 1 + lung[j];
          succ[i] = j;
     }
   }

}
```

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1; succ[n] = n+1; //stim
for (int i=n-1;i>=1;i--){//ordine de calcul
    succ[i]= n+1; lung[i]=1;
    //formula de recurenta
    for(int j=i+1;j<=n; j++){
      if((a[i]<a[j]) && (1+lung[j]>lung[i])){
           lung[i]= 1 + lung[j];
           succ[i] = j;
      }
    }
    if(lung[i]> nr){nr = lung[i]; poz = i;}
}
```

```
//afisare subsir
for (int i=1;i<=nr;i++){
    System.out.print(a[poz]+" ");
    poz = succ[poz];
}
```

```
//afisare subsir
for (int i=1;i<=nr;i++){
   System.out.print(a[poz]+" ");
   poz = succ[poz];
}
```

- **Complexitate – $O(n^2)$**

```
//afisare subsir
for (int i=1;i<=nr;i++){
    System.out.print(a[poz]+" ");
    poz = succ[poz];
}
```

- **Complexitate – $O(n^2)$**

- **Temă O(n log n)**

Altă soluție

Principiu de optimalitate:

Dacă

$$a_{i1},\ a_{i2},\ \ldots,\ a_{ip},$$

este un subșir optim care se termină pe poziția `ip`, atunci

Altă soluție

Principiu de optimalitate:

Dacă

$$a_{i1},\ a_{i2},\ \dots,\ a_{ip},$$

este un subșir optim care se termină pe poziția `ip`, atunci

$$a_{i1},\ \dots,\ a_{ik}$$

este un subșir optim care se termină pe poziția `ik`.

Altă soluție

Principiu de optimalitate:

Dacă

$$a_{i1},\ a_{i2},\ \ldots,\ a_{ip},$$

este un subșir optim care se termină pe poziția `ip`, atunci

$$a_{i1},\ \ldots,\ a_{ik}$$

este un subșir optim care se termină pe poziția `ik`.

Subproblemă:

Calculăm pentru fiecare poziție i lungimea maximă a unui subşir crescător ce **se termină pe poziția i**

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :							
pred :							

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1						
pred :	0						

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1	1					
pred :	0	0					

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1	1	2				
pred :	0	0	2				

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1	1	2	2			
pred :	0	0	2	2			

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1	1	2	2	3		
pred :	0	0	2	2	4		

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1	1	2	2	3	3	
pred :	0	0	2	2	4	4	

Subşir crescător de lungime maximă

a:	8	1	7	4	6	5	11
	1	2	3	4	5	6	7
lung :	1	1	2	2	3	3	4
pred :	0	0	2	2	4	4	6

Subşir crescător de lungime maximă

Algoritm O(n log n)– Indicații

1. pentru i=1,n

Pentru fiecare lungime j=1..,n reținem

m[j] = poziția **celui mai mic element din şir** cu proprietatea că există un **subșir de lungime j care se termină cu el** (în subvectorul a[1..i])

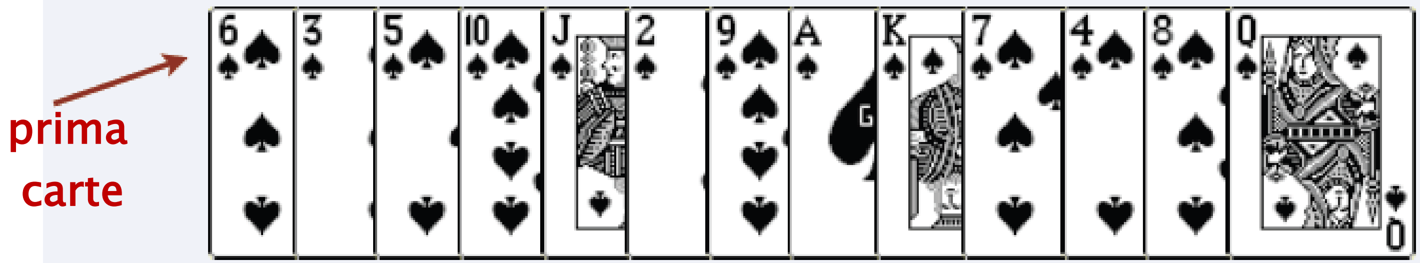
- **a[m[1]] ≤ a[m[2]] ≤ … ≤ a[m[n]]**
- La pasul i – căutăm binar cea mai mare lungime j cu

  a[m[j]] ≤ a[i]

  – dacă găsim j se actualizează m[j+1], altfel se actualizează m[1]

Subşir crescător de lungime maximă

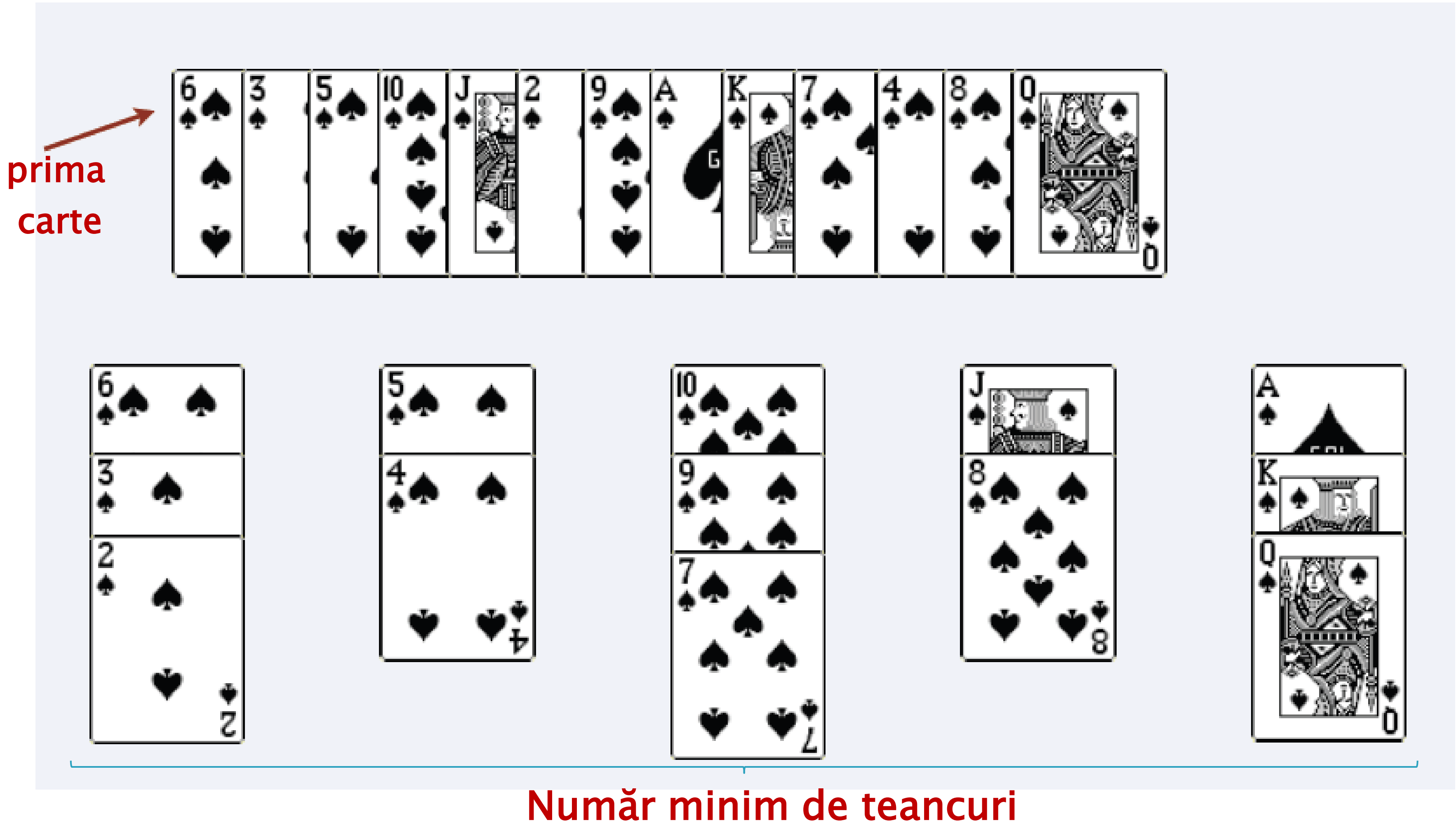
Patience solitaire / patience sort



https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/LongestIncreasingSubsequence.pdf

Subşir crescător de lungime maximă

Patience solitaire / patience sort



https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/LongestIncreasingSubsequence.pdf

Subşir crescător de lungime maximă

Patience solitaire

Algoritm: Greedy – cartea curentă este adăugată la cel mai din stânga teanc pe care se potrivește

− La fiecare pas, cărțile din topul fiecărui teanc formează un șir crescător

- Determinarea celui mai din stânga teanc pe care se potrivește cartea – cu **căutare binară**

– **O(n log n)**

Subşir crescător de lungime maximă

Patience solitaire / patience sort

6, 3, 5, 10, 12, 2, 9, 15, 14, 7, 4,8,13

Subşir crescător de lungime maximă

Patience solitaire / patience sort

3, 5, 10, 12, 2, 9, 15, 14, 7, 4,8,13

6

Subşir crescător de lungime maximă

Patience solitaire / patience sort

5, 10, 12, 2, 9, 15, 14, 7, 4,8,13

6

3

Subşir crescător de lungime maximă

Patience solitaire / patience sort

10, 12, 2, 9, 15, 14, 7, 4,8,13

6 5

3

Subşir crescător de lungime maximă

Patience solitaire / patience sort

12, 2, 9, 15, 14, 7, 4,8,13

6 5 10
3

Subşir crescător de lungime maximă

Patience solitaire / patience sort

2, 9, 15, 14, 7, 4,8,13

6 5 10 12
3

Subşir crescător de lungime maximă

Patience solitaire / patience sort

9, 15, 14, 7, 4,8,13

6	5	10	12
3			
2			

Subşir crescător de lungime maximă

Patience solitaire / patience sort

15, 14, 7, 4,8,13

6	5	10	12
3		9	
2			

Subşir crescător de lungime maximă

Patience solitaire / patience sort

14, 7, 4,8,13

6	5	10	12	15
3		9		
2				

Subşir crescător de lungime maximă

Patience solitaire / patience sort

7, 4,8,13

6	5	10	12	15
3		9		14
2				

Subşir crescător de lungime maximă

Patience solitaire / patience sort

4,8,13

6	5	10	12	15
3		9		14
2		7		

Subşir crescător de lungime maximă

Patience solitaire / patience sort

8,13

6	5	10	12	15
3	4	9		14
2		7		

Subşir crescător de lungime maximă

Patience solitaire / patience sort

13

6	5	10	12	15
3	4	9	8	14
2		7		

Subşir crescător de lungime maximă

Patience solitaire / patience sort

6	5	10	12	15
3	4	9	8	14
2		7		13

Subşir crescător de lungime maximă

Patience solitaire / patience sort

Evident: numărul minim de subșiruri descrescătoare în care se poate descompune un șir $\geq$ lungimea maximă a unui subșir crescător

Subşir crescător de lungime maximă

Patience solitaire / patience sort

Proprietate: numărul minim de subșiruri descrescătoare în care se poate descompune un șir =

lungimea maximă a unui subșir crescător

Subşir crescător de lungime maximă

Patience solitaire / patience sort

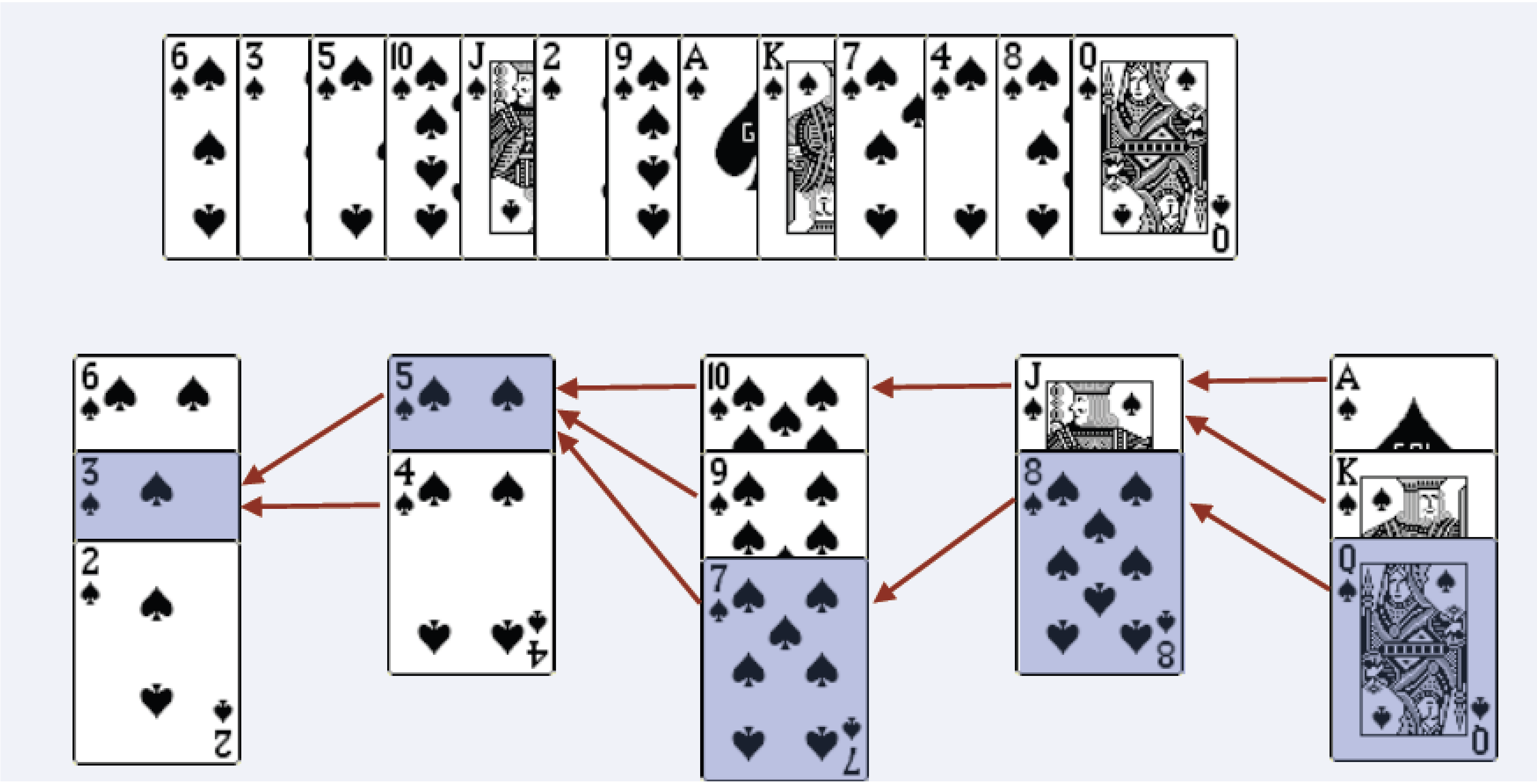
Proprietate: numărul minim de subșiruri descrescătoare în care se poate descompune un șir =

lungimea maximă a unui subșir crescător

- Pentru a memora un subșir crescător, memorăm la fiecare pas al algoritmului Greedy, pentru cartea curent adăugată o legătură de tip predecesor către vârful teancului anterior celui în care a fost adăugată
- Subșirul crescător se obține pornind de la ultima carte adăugată și urmând legătura predecesor

Subşir crescător de lungime maximă

Patience solitaire / patience sort



https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/LongestIncreasingSubsequence.pdf

Subşir crescător de lungime maximă

Patience sort

–după distribuirea cărților în teancuri alegem succesiv cartea cu cea mai mică valoare din vârful unui teanc și o adăugăm în șir $\longrightarrow$ obținem șirul de cărți ordonat crescător