

Metoda Backtracking

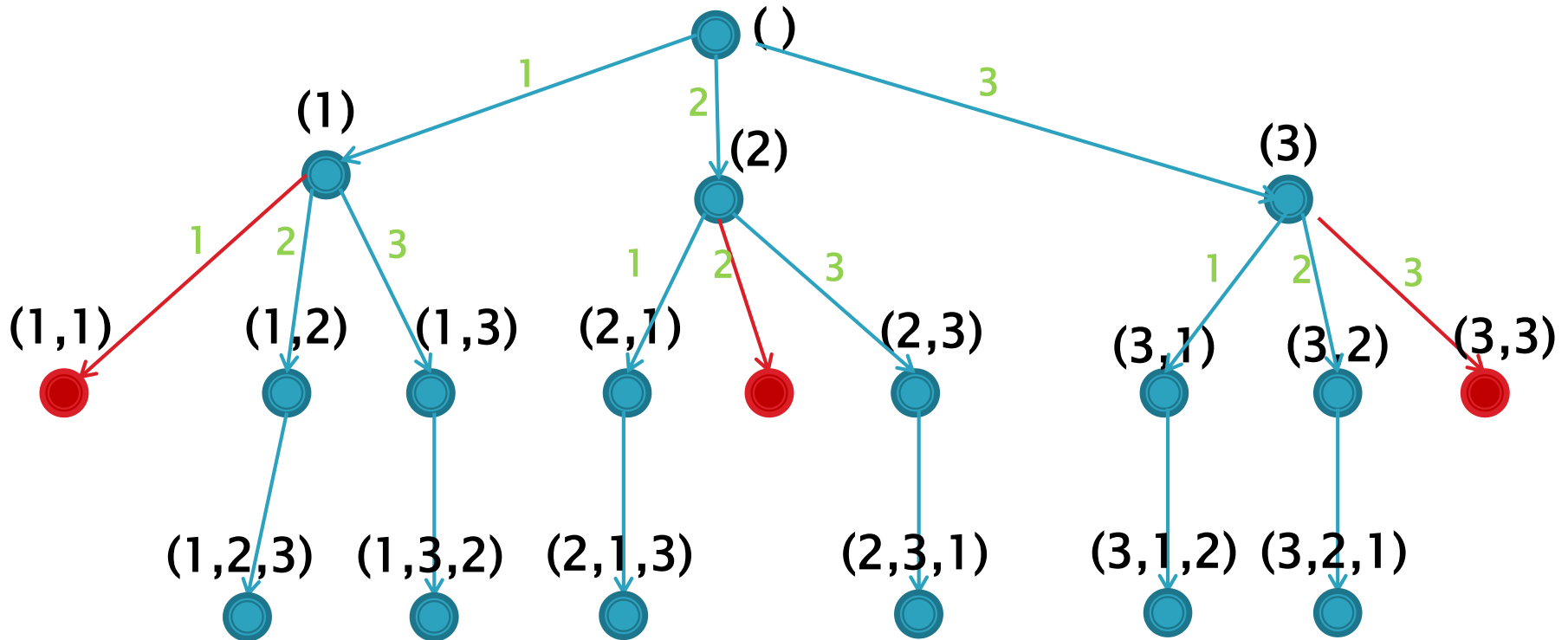


Cadru

- ▶ **Exemplu – Generarea permutărilor**
- ▶ Căutare mai “inteligentă” în spațiul în care se găsesc soluțiile posibile (printre soluții candidat)
- ▶ **Configurațiile prin care se trece în procesul de căutare – structură arborescentă (soluție construită element cu element)**

Exemplu

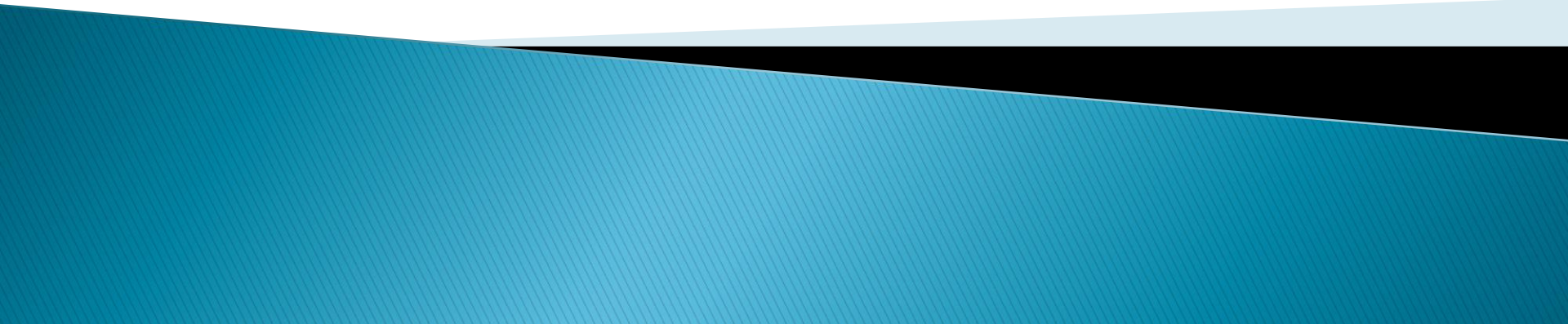
- ▶ Permutări $\{1, 2, 3\}$



Cadru

- ▶ Parcurgerea completă a arborelui \Rightarrow algoritm exhaustiv (brute force), care consideră toate soluțiile candidat
- ▶ Mai rapid – limitarea parcurgerii arborelui prin determinarea de configurații care **nu mai trebuie explorate** (nu pot conduce către soluții dorite)
- ▶ De exemplu **(1,1) nu poate fi extins la o permutare**

Metoda Backtracking



Metoda Backtracking

- ▶ Probleme computaționale + cu constrângeri (constraint satisfaction problems) – puzzle
- ▶ Combinatorică
- ▶ Căutare IA

Metoda Backtracking

- ▶ Soluțiile se construiesc **element cu element**, trecând prin configurații **corespunzătoare soluțiilor parțiale (“incomplete”)**
- ▶ Aceste configurații se pot **testa** dacă **nu** pot fi completate până la o soluție posibilă → **condiții de continuare**

Cadru posibil

- ▶ Soluția se poate reprezenta sub formă de **vector**:

$$X = X_1 \times \dots \times X_n = \text{spațiul soluțiilor candidat}$$

- ▶ $p : X \rightarrow \{0, 1\}$ este o **proprietate** definită pe X pe care trebuie să o verifice soluția – numită condiție internă (finală) pentru x
- ▶ **Căutăm un element $x \in X$ cu proprietatea $p(x)$**

Metoda Backtracking

- ▶ Generarea tuturor elementelor produsului cartezian X nu este acceptabilă.
- ▶ Metoda backtracking încearcă micșorarea timpului de calcul – prin evitarea generării unor soluții care nu satisfac anumite **condițiile de continuare**

Metoda Backtracking

Condiții de continuare pentru soluția parțială $y = x_1 \dots x_k$
notate **cont_k**(x) = condiții de continuare a completării
soluției (a parcurgerii subarborelui de rădăcină x)

- Condițiile de continuare
 - rezultă de obicei din condițiile interne (finale)
 - sunt strict necesare, **ideal fiind să fie și suficiente**
 - sunt importante pentru micșorarea timpului de executare

Metoda Backtracking

- ▶ Vectorul soluție $x = (x_1, x_2, \dots, x_n) \in X$ este **construit progresiv**, începând cu prima componentă.

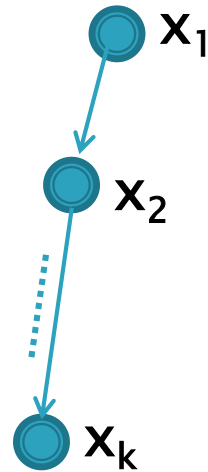
Metoda Backtracking

Așadar:

- ▶ Vectorul soluție $x = (x_1, x_2, \dots, x_n) \in X$ este **construit progresiv**, începând cu prima componentă.
- ▶ Se avansează cu o valoare pentru x_k dacă este satisfăcută **condiția de continuare** $\text{cont}(x_1, \dots, x_k)$

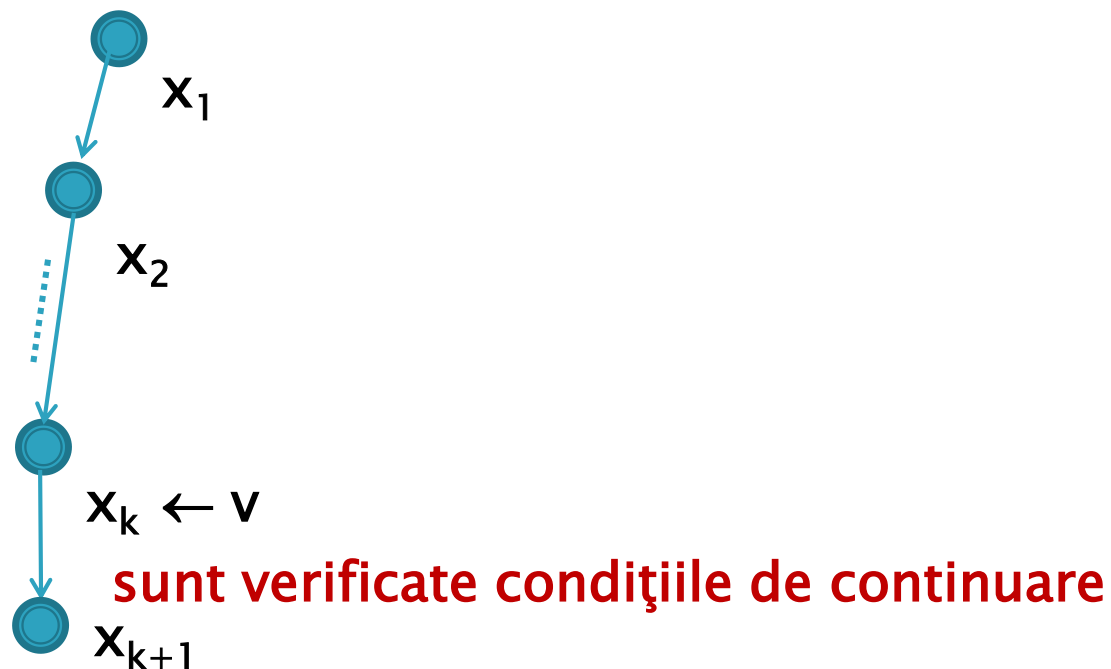
Metoda Backtracking

- Cazuri posibile pentru x_k :



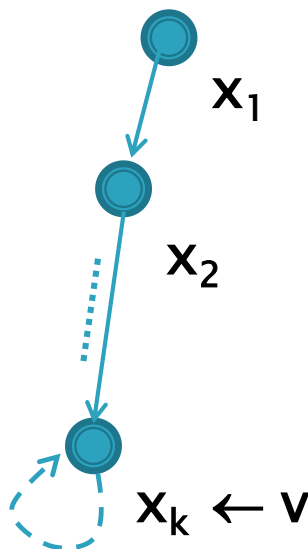
Metoda Backtracking

- Atribuire o valoare $v \in X_k$ lui x_k și avansează (sunt verificate condițiile de continuare)



Metoda Backtracking

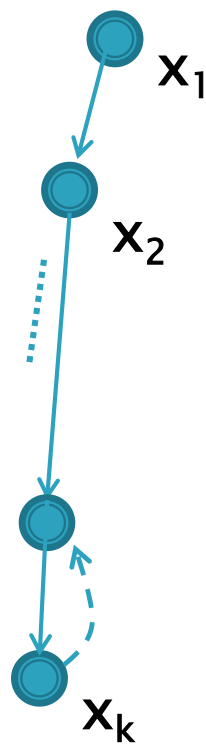
- ❑ Încercare eșuată (atribuie o valoare $v \in X_k$ lui x_k pentru care nu sunt verificate condițiile de continuare)



nu sunt verificate condițiile
de continuare

Metoda Backtracking

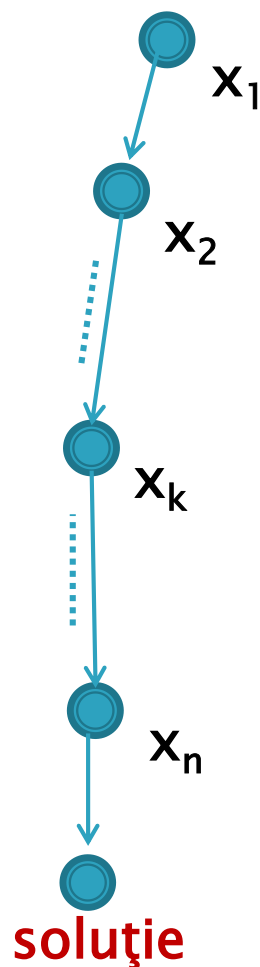
□ Revenire – nu mai există valori $v \in X_k$ neconsiderate



nu mai există valori pentru x_k
neconsiderate

Metoda Backtracking

- Revenire după determinarea unei soluții



revenire după
determinarea unei soluții

Cazul $X_i = \{p_i, p_i + 1, \dots, u_i\}$

- ▶ $X_i = \{p_i, p_i+1, \dots, u_i\}$
- ▶ Apelul inițial este: **back(1)**

```
procedure back(k)
```

```
  if k=n+1
```

```
    retsol(x) {revenire dupa solutie}
```

```
  else
```

```
end.
```

- ▶ $X_i = \{p_i, p_i+1, \dots, u_i\}$
- ▶ Apelul inițial este: **back(1)**

```
procedure back(k)
```

```
  if k=n+1
```

```
    retsol(x) {revenire dupa solutie}
```

```
  else
```

```
    for (i=pk; i<=uk; i++) {valori posibile}
```

```
      xk←i; {atribuie}
```

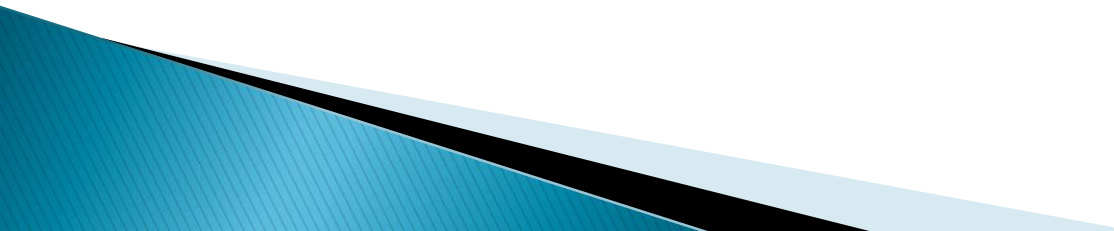
```
end.
```

- ▶ $X_i = \{p_i, p_i+1, \dots, u_i\}$
- ▶ Apelul inițial este: **back(1)**

```
procedure back(k)
  if k=n+1
    retsol(x) {revenire dupa solutie}
  else
    for (i=pk; i<=uk; i++) {valori posibile}
      xk ← i; {atribuie}
      if cont(x1, ..., xk)
        back(k+1); {avanseaza}
        {revenire din recursivitate}
  end.
```

Exemple

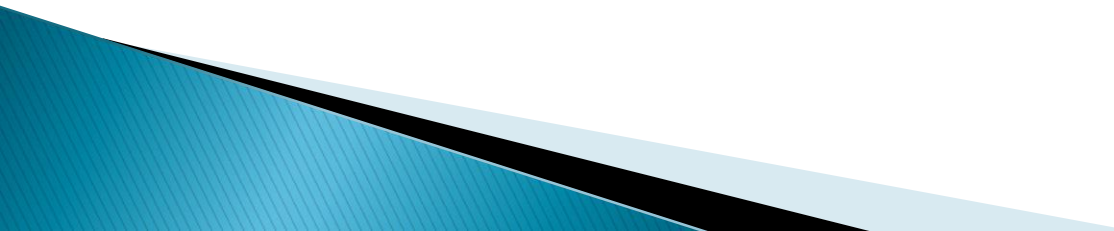
Exemple

- ▶ Permutări, combinări, aranjamente
 - ▶ Produs cartezian
 - ▶ Submulțimi
 - ▶ Toate subșirurile crescătoare de lungime maximă
 - ▶ Partițiile unui număr n
 - ▶ Toate descompunerile unei sume folosind monede date
- 

Permutări

- ▶ Permutările mulțimii $\{1, 2, \dots, n\}$

Permutări

- ▶ **Reprezentarea soluției**
 - ▶ **Condiții interne (finale)**
 - ▶ **Condiții de continuare (!!pentru x_k)**
- 

Permutări

- ▶ **Reprezentarea soluției**

$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (p_k = 1, u_k = n) .$

- ▶ **Condiții interne (finale)**

- ▶ **Condiții de continuare (!!pentru \mathbf{x}_k)**

Permutări

► Reprezentarea soluției

$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (\mathbf{p}_k = 1, \mathbf{u}_k = n) .$

► Condiții interne (finale)

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice $i \neq j$.

► Condiții de continuare (!!pentru \mathbf{x}_k)

Permutări

► Reprezentarea soluției

$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (\mathbf{p}_k = 1, \mathbf{u}_k = n) .$

► Condiții interne (finale)

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice $i \neq j$.

► Condiții de continuare (!!pentru \mathbf{x}_k)

$\mathbf{x}_i \neq \mathbf{x}_k$ pentru orice $i \in \{1, 2, \dots, \mathbf{k}-1\}$

Permutări, $n=3$



1

1 1

1 2

1 2 1

1 2 2

1 2 3

1 2 3 soluție

1 3

1 3 1

1 3 2

1 3 2 soluție

1 3 3

2

2 1

2 1 1

2 1 2

2 1 3

2 1 3 soluție

etc

```

def back(k):
    if k == n+1: #a completat deja cele n elemente din x
        print(*x[1:],sep=", ")
    else:
        for i in range(1, n+1): #xi valori intre 1 si n
            x[k] = i
            if x[k] not in x[:k]: #conditia de continuare
                #mai bine x.index(x[k],0,k)
                back(k+1)

n = int(input("n = "))
# o soluție s va avea n elemente
x = [0]*(n+1)
print(f"Permutările de lungime {n}:")
back(1)

```

Permutări – Variante

- ▶ **Permutări cu exact un punct fix:**
 - testăm la final fiecare permutare obținută dacă are un punct fix
 - **în plus** testăm la condițiile de continuare să nu avem deja mai mult de un punct fix

```

def nr_puncte_fixe(x,n):
    k=0
    for i in range(1,n+1):
        if x[i]==i:
            k+=1
    return k
def back(k):
    if k==n+1:
        if nr_puncte_fixe(x,n)==1:
            print(*x[1:],sep=", ")
    else:
        for i in range(1, n+1):
            x[k] = i
            if x[k] not in x[:k] and nr_puncte_fixe(x,k)<=1:
                back(k+1)

```


Permutări – Variante

▶ Anagramele unui cuvânt

- generăm permutări x ale indicilor – le corespund anagrame ale cuvântului

$s = are$

$x = [1, 3, 2] \Rightarrow \text{anagrama } s[1]s[3]s[2] = aer$

▶ Anagramele distincte ale unui cuvânt

- o soluție: le memorez în set
- TEMĂ – fără a le memora

```
def back(k):
    if k==n+1:
        rez.add("".join([s[x[i]-1] for i in range(1,k)]))
    else:
        for i in range(1, n+1):
            x[k] = i
            if x[k] not in x[:k]:
                back(k+1)
```

```
s=input("cuvant ")
n=len(s)
# o soluție s va avea n elemente
x = [0]*(n+1)
print(f"Anagrame:")
rez=set()
back(1)
print(rez)
```

Aranjamente

- ▶ Aranjamente de m elemente ale mulțimii $\{1, 2, \dots, n\}$ (contează ordinea)

$n = 5$

$m = 3$:

1 2 3

1 2 4

1 2 5

1 3 2 ...

Aranjamente

► Reprezentarea soluției

$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m)$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (p_k = 1, u_k = n) .$

► Condiții interne (finale) – ca la permutări

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice $i \neq j$.

► Condiții de continuare (!!pentru \mathbf{x}_k)

$\mathbf{x}_i \neq \mathbf{x}_k$ pentru orice $i \in \{1, 2, \dots, k-1\}$

```
def back(k):
    if k == m+1:
        print(*x[1:], sep=", ")
    else:
        for i in range(1, n+1):
            x[k] = i
            if x[k] not in x[:k]:
                back(k+1)

n = int(input("n = "))
m = int(input("m = "))
x = [0]*(m+1)
print(f"Aranjamente de lungime {m} ale multimii {{1,...,{n}}}:")
back(1)
```

Combinări

- Combinări de m elemente ale mulțimii $\{1, 2, \dots, n\}$ (submulțimi cu m elemente)

$n = 5$

$m = 3$:

1 2 3

1 2 4

1 2 5

1 3 4

1 3 5 ...

Combinări

► Reprezentarea soluției

$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m)$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (p_k = 1, u_k = n) .$


► Condiții interne (finale) – crescător=>distincte

$x_i < x_j$ pentru orice $i < j$.

► Condiții de continuare (!!pentru x_k)

$\mathbf{x}_{k-1} < \mathbf{x}_k$

```
def back(k):  
    if k==m+1:  
        print(*x[1:],sep=", ")  
    else:  
        for i in range(1, n+1):  
            x[k] = i  
            if x[k] > x[k-1]:  
                back(k+1)
```



Genereaza valori inutile, am putea porni direct cu prima valoare pentru $x[k]$ egală cu $x[k-1]+1$


```

def back(k):
    if k == m+1:
        print(*x[1:], sep=", ")
    else:
        for i in range(x[k-1]+1, n+1):
            x[k] = i
            back(k+1)

n = int(input("n = "))
m = int(input("m = "))
# o soluție s va avea m elemente
x = [0]*(m+1)
print(f"Toate combinarile (submultimi) de {m} ale multimei
{{1,...,{n}}}-metoda 2")
back(1)

```

Submulțimi

► Submulțimile mulțimii $\{1, 2, \dots, n\}$

O submulțime - asociat un vector caracteristic v cu n elemente 0/1 ($v_i = 0 \Leftrightarrow i$ nu aparține submulțimii)

$n = 5: \{1, 2, 3, 4, 5\}$

Submulțimea $\{2, 3, 5\} \rightarrow$ vectorul $[0, 1, 1, 0, 1]$

Generare de submulțimi = generare de șiruri binare de lungime n

Submulțimi

- ▶ **Reprezentarea soluției**

$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m)$, unde

$\mathbf{x}_k \in \{0, 1\}$

- ▶ **Condiții interne (finale)**

- ▶ **Condiții de continuare (!!pentru \mathbf{x}_k)**

```

def back(k):
    if k==n+1:
        print("submultime:",end=" ")
        for i in range(1,n+1):
            if x[i]==1:
                print(v[i-1],end=" ")
        print()
    else:
        for i in range(0,2):
            x[k] = i
            back(k+1)

s = input("multime:")
v=[int(x) for x in s.split()]
n=len(v)
x = [0]*(n+1)
print(f"Submultimile multimi {{ {'','.join(s.split())} }}:")
back(1)

```

Submulțimi

- ▶ **Toate submulțimi de sumă M (M dat) ale unei mulțimi**
 - condiție finală : suma elementelor din submulțime = M
 - condiție de continuare: suma elementelor deja adăugate la submulțime să nu depășească M

```

def suma(x,n):
    s=0
    for i in range(1, n + 1):
        if x[i] == 1:
            s=s+v[i - 1]
    return s

def back(k):
    if k==n+1:
        if suma(x,n)==M:
            for i in range(1,n+1):
                if x[i]==1:
                    print(v[i-1],end=" ")
            print()
    else:
        for i in range(0,2):
            x[k] = i
            if suma(x,k)<=M:
                back(k+1)

```

Submulțimi

- ▶ **Recomandare** – nu recalculăm sume de câte ori mai adaug un element

```

def back2(k):
    global s
    if k==n+1:
        if s==M:
            for i in range(1,n+1):
                if x[i]==1: print(v[i-1],end=" ")
            print()
        else:
            for i in range(0,2):
                x[k] = i
                if x[k] == 1: s = s + v[k - 1]
                if s <= M:
                    back2(k + 1)
                if x[k] == 1: s = s - v[k - 1]
s=input("multime:"); v=[int(x) for x in s.split()]
M=int(input("M=")); n=len(v)
x = [0]*(n+1)
print(f"Submultimile de suma {M} - metoda 2:")
s = 0
back2(1)

```


Partițiile unui număr natural n

- ▶ **Exemplu:** Dat un număr natural n , să se genereze toate partițiile lui n ca sumă de numere pozitive

Partiție a lui $n = \{x_1, x_2, \dots, x_k\}$ cu

$$x_1 + x_2 + \dots + x_k = n$$

$$n = 3$$

$$1 + 1 + 1$$

$$1 + 2$$

$$3$$

Partițiile unui număr natural n

► Reprezentarea soluției

$$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}, \text{ unde}$$
$$\mathbf{x}_i \in \{1, \dots, n\}$$

► Condiții interne (finale)

$$\mathbf{x}_1 + \mathbf{x}_2 + \dots + \mathbf{x}_k = n$$

Pentru unicitate: $\mathbf{x}_1 \leq \mathbf{x}_2 \leq \dots \leq \mathbf{x}_k$

► Condiții de continuare

$$\mathbf{x}_{k-1} \leq \mathbf{x}_k \longrightarrow \mathbf{x}_k \in \{\mathbf{x}_{k-1}, \dots, n\}$$

$$\mathbf{x}_1 + \mathbf{x}_2 + \dots + \mathbf{x}_k \leq n$$

```
def back1(k):  
    global s  
    if sum(x[1:k])==n:  
        print(*x[1:k],sep="+")  
    else:  
        for i in range(x[k-1], n+1):  
            x[k] = i  
            if sum(x[1:k+1])<=n:  
                back1(k+1)
```

```
n = int(input("n = "))  
x = [0]*(n+1)  
print(f"Descompuneri distincte:")  
x[0]=1  
s=0  
back1(1)
```

```
#varianta- fara a recalcula sum
def back(k):
    global s
    if s==n:
        print(*x[1:k],sep="+")
    else:
        for i in range(x[k-1], n+1):
            x[k] = i
            s = s + x[k]
            if s<=n:
                back(k+1)
            s = s - x[k]
n = int(input("n = "))
x = [0]*(n+1)
print(f"Descompuneri distincte:")
x[0]=1
s=0
back(1)
```

Subșiruri crescătoare lungime maximă

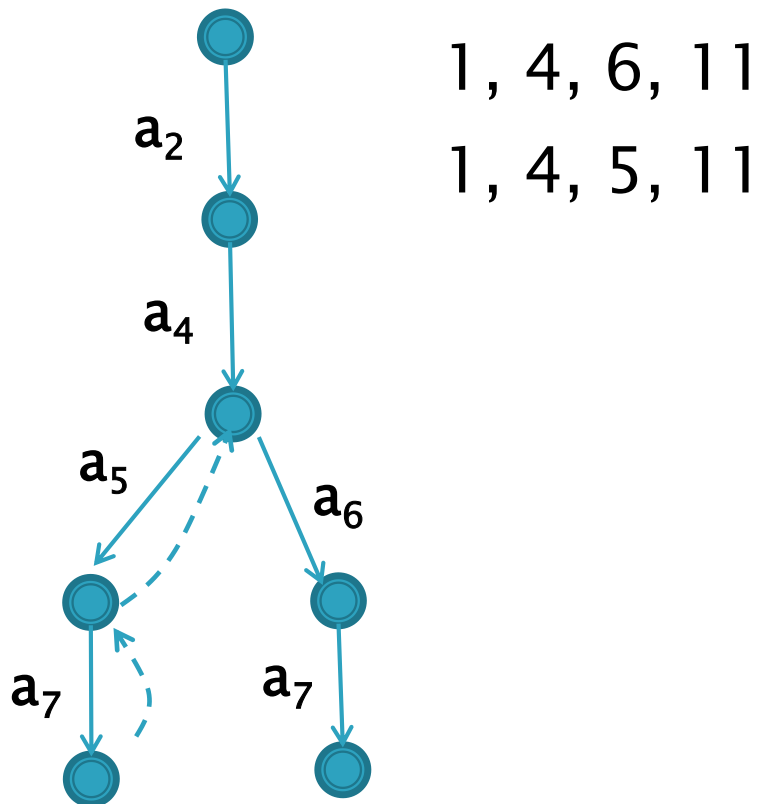
- ▶ Fie vectorul $a=(a_1,\dots,a_n)$. Să se determine **toate subșirurile crescătoare de lungime maximă**

Subșiruri crescătoare lungime maximă

- ▶ Un subșir crescător de lungime maximă **începe** cu o poziție i cu $\text{lung}[i] = \text{lmax}$
 - ▶ Următorul element după a_i este un a_j cu proprietățile
 - $i < j$
 - $a_j > a_i$
 - $\text{lung}[j] = \text{lung}[i] - 1$
- = **acei a_j posibili succesori ai lui i , pentru care se realizează egalitate în relația de recurență** (în PD se reține în $\text{succ}[i]$ un singur astfel de j)

Subșiruri crescătoare 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



Subșiruri crescătoare lungime maximă

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{l_{\max}}\}$, unde

$\mathbf{x}_k \in \{1, \dots, n\}$ poziție din vectorul a

► Condiții interne (finale)

$$a_{\mathbf{x}_1} < a_{\mathbf{x}_2} < \dots < a_{\mathbf{x}_{l_{\max}}}$$

► Condiții de continuare

$$\text{lung}[a[\mathbf{x}_1]] = l_{\max}$$

$$\mathbf{x}_{k-1} < \mathbf{x}_k, \quad a_{\mathbf{x}_{k-1}} < a_{\mathbf{x}_k}, \quad \text{lung}[\mathbf{x}_k] = \text{lung}[\mathbf{x}_{k-1}] - 1$$


```

def afisToateSolutiile():
    for i in range(0,n):
        if lung[i] == l_max:
            x[1] = i
            back(2)

def back(k):
    if k == l_max + 1:
        for i in range(1,k):
            print(a[x[i]], end=" ")
        print()
    else:
        for j in range(x[k-1]+1,n):
            x[k]=j
            if a[x[k-1]] < a[x[k]] and lung[x[k-1]]== 1+lung[x[k]]:
                back(k+1)

```

