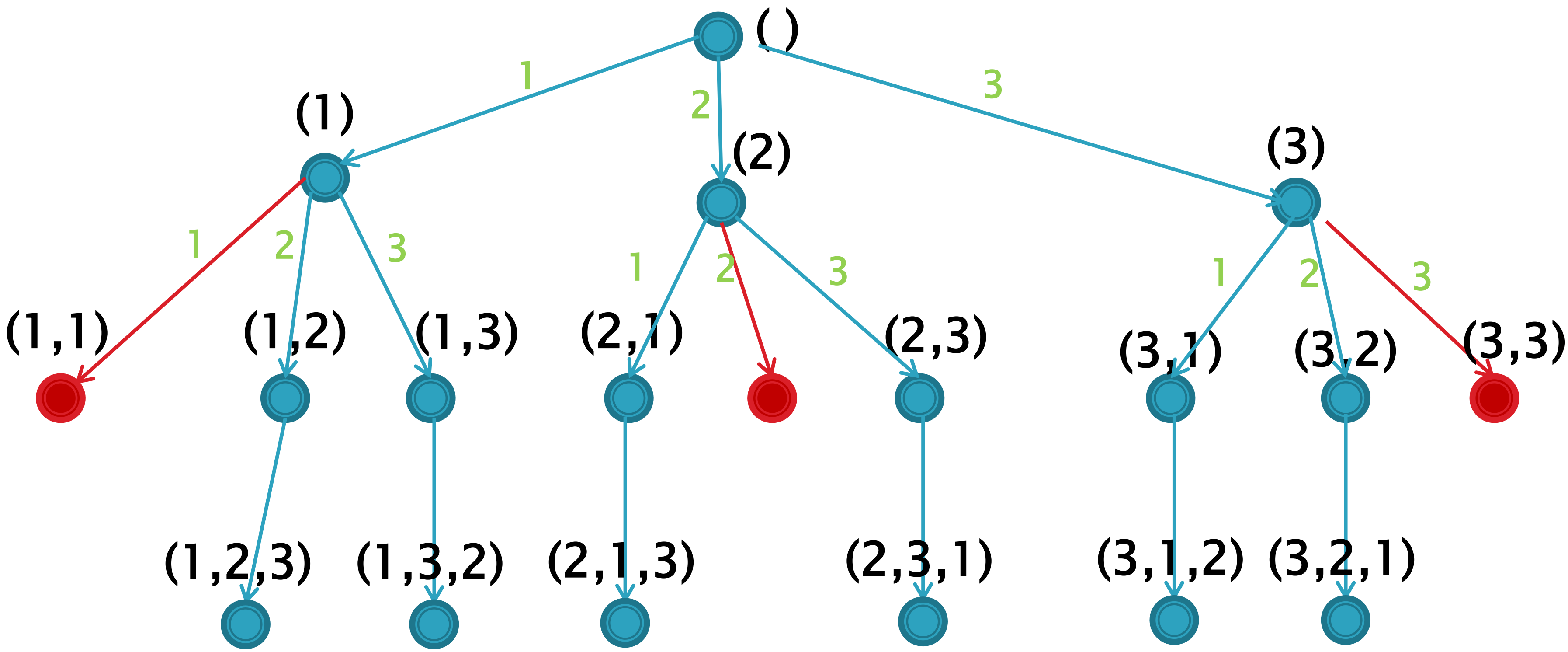
Metoda Backtracking

Cadru

- **Exemplu** – Generarea permutărilor
- Căutare mai “inteligentă” în spaţiul în care se găsesc soluţiile posibile (printre soluţii candidat)
- **Configuraţiile 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 ⇒ algoritm exhaustiv (brute force), care consideră toate soluţiile candidat

- Mai rapid – **limitarea parcurgerii** arborelui prin determinarea de configuraţii care **nu mai trebuie explorate (nu pot conduce către soluţii** dorite)

- De exemplu **(1,1) nu poate fi extins la o permutare**

Metoda Backtracking

Metoda Backtracking

- Soluțiile se construiesc **element cu element**, trecând prin configurații **corespunzătoare soluțiilor parțiale ("incomplete")**

- Aceste configuraţii 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**:

  $X = X_1 \times \ldots \times X_n$ = spaţiul soluţiilor candidat

- p :X $\rightarrow$ {0,1} este o **proprietate** definită pe X pe care trebuie să o verifice soluția – numită condiţie 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 micşorarea timpului de calcul – prin **evitarea generării unor soluţii care nu satisfac anumite** **condiţiile de continuare**

Metoda Backtracking

<u>Condiţii de continuare</u> **pentru soluția parțială $y=x_1...x_k$** notate **cont**(x,k) = condiții de **continuare a completării soluției** (a parcurgerii subarborelui de rădăcină x)

- **Condiţiile de continuare**
  - **rezultă de obicei din condițiile interne (finale)**
  - sunt <u>**strict necesare**</u>, **ideal fiind să fie şi suficiente**
  - sunt importante pentru micșorarea timpului de executare

Metoda Backtracking

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

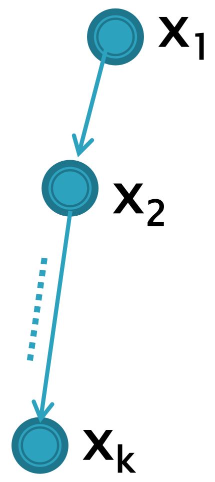
Metoda Backtracking

Așadar:

- Vectorul soluție $x = (x_1, x_2,\ldots,x_n) \in X$ este **construit progresiv**, începând cu prima componentă.
- Se avansează cu o valoare pentru x_k dacă este satisfăcută **condiţia de continuare** cont(x,k)

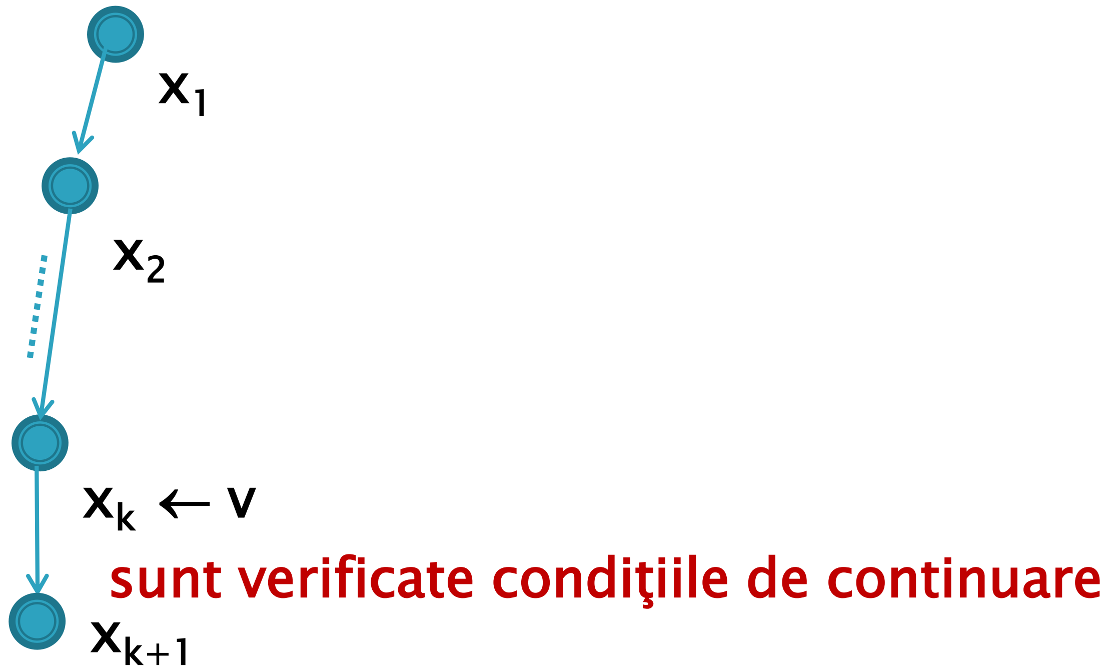
Metoda Backtracking

- Cazuri posibile pentru x_k:



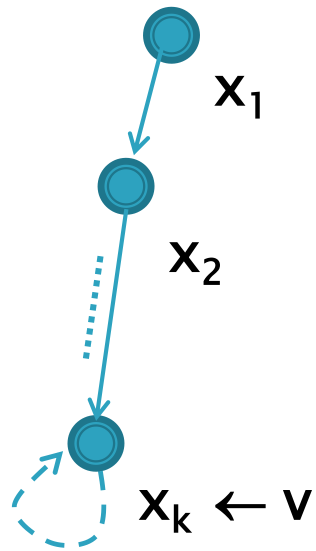
Metoda Backtracking

- Atribuie o valoare $v \in X_k$ lui x_k şi avansează (sunt verificate condiţiile de continuare)



Metoda Backtracking

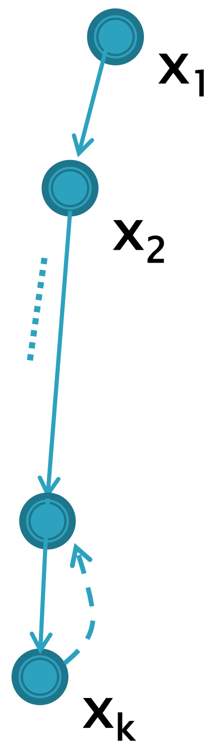
- Încercare eşuată (atribuie o valoare $v \in X_k$ lui x_k pentru care nu sunt verificate condiţiile de continuare)



nu sunt verificate condiţiile de continuare

Metoda Backtracking

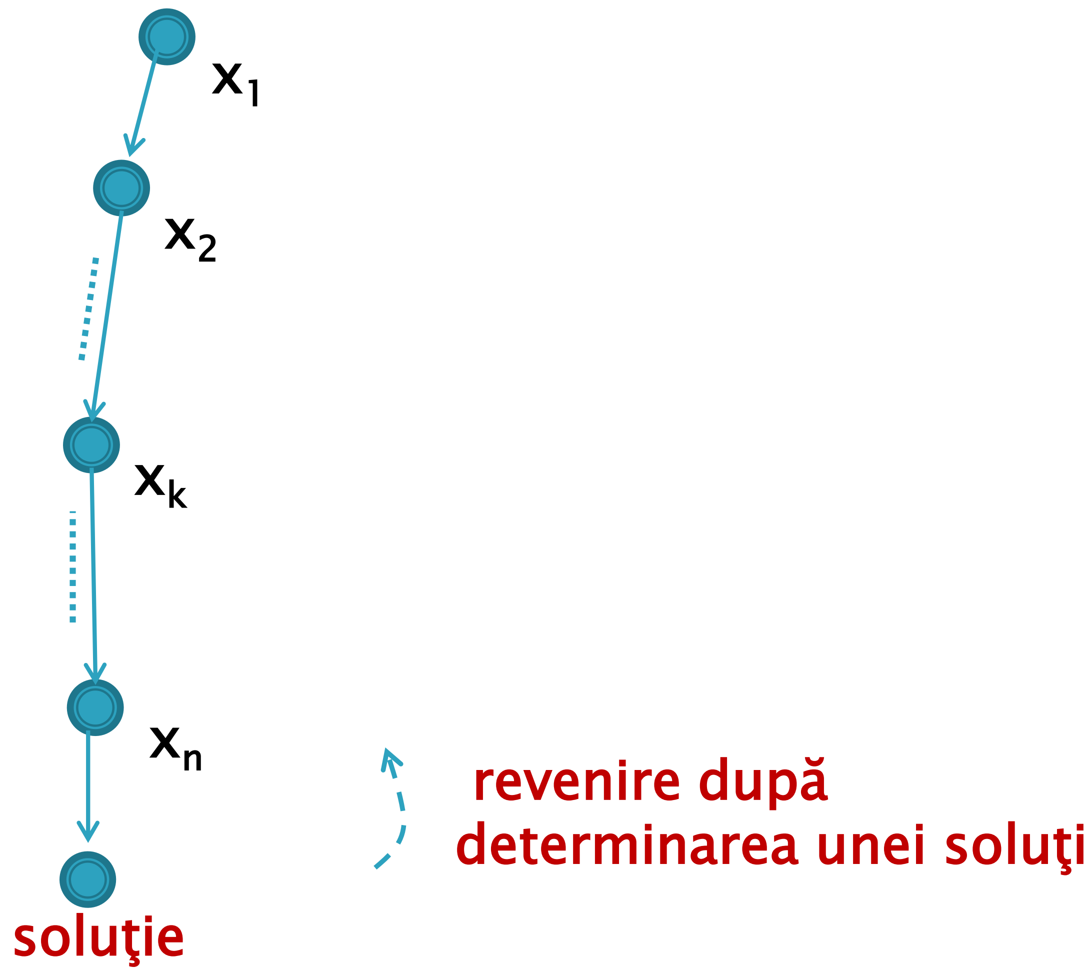
- Revenire – nu mai există valori $v \in X_k$ neconsiderate



nu mai există valori pentru x_k neconsiderate

Metoda Backtracking

- Revenire după determinarea unei soluţii



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

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **`back(1)`**

```
procedure back(k)
  if k=n+1
      retine_sol(x) {revenire dupa solutie}
  else









end.
```

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

```
procedure back(k)
  if k=n+1
       retine_sol(x) {revenire dupa solutie}
  else
       for i=pk,…,uk {valori posibile}
          xk←i;                      {atribuie}




end.
```

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **back(1)**

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

Varianta nerecursivă

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
        retine_sol(x); k←k-1;{revenire după o sol.}
```

Varianta nerecursivă

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
        retine_sol(x); k←k-1;{revenire după o sol.}
   else
       if xk<uk  {mai sunt valori în Xk}
           xk←xk+1;
```

Varianta nerecursivă

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
       retine_sol(x); k←k-1;{revenire după o sol.}
   else
      if xk<uk  {mai sunt valori în Xk}
          xk←xk+1;
          if cont(x,k)
              k←k+1;    { atribuie şi avansează }
          else               { încercare eşuată }
```

Varianta nerecursivă

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
       retine_sol(x); k←k-1;{revenire după o sol.}
   else
      if xk<uk  {mai sunt valori în Xk}
          xk←xk+1;
          if cont(x,k)
              k←k+1;    { atribuie şi avansează }
          else                { încercare eşuată }
      else xk←pk-1; k←k-1;     { revenire }
```

Observații

- În procedura retine_sol este posibil sa fie nevoie de testarea condițiilor finale, dacă pe parcurs condițiile de continuare nu au fost suficiente cât să garanteze obținerea unei soluții corecte

- Metoda poate fi folosită și în:
  - **probleme de optim** – atunci in retine_sol memorăm cea mai bună soluție generată până acum conform criteriului de optim
  - **probleme de numărare** – atunci in retine_sol contorizăm soluțiile generate până acum

Exemple

Exemple

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

Permutări

- Permutările mulțimii {1,2,… ,n}

Permutări

- Reprezentarea soluţiei

- Condiţii interne (finale)

- Condiţii de continuare (!!pentru x_k)

Permutări

- Reprezentarea soluţiei

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

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

- Condiţii interne (finale)

- Condiţii de continuare (!!pentru x_k)

Permutări

- **Reprezentarea soluţiei**

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

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

- **Condiţii interne (finale)**

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

- **Condiţii de continuare (!!pentru x_k)**

Permutări

- **Reprezentarea soluţiei**

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

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

- **Condiţii interne (finale)**

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

- **Condiţii de continuare (!!pentru x_k)**

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

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
```

```python
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] => 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,... ,n} (contează ordinea)

```
n = 5
m = 3:
     1 2 3
     1 2 4
     1 2 5
     1 3 2 ...
```

Aranjamente

- **Reprezentarea soluţiei**

  $x = (x_1, x_2, \ldots, x_m)$, unde

  $x_k \in \{1,2,\ldots,n\}$ $(p_k = 1,\ u_k = n)$.

- **Condiţii interne (finale)** – ca la permutări

  $x_i \neq x_j$ pentru orice $i \neq j$.

- **Condiţii de continuare (!!pentru x_k)**

  $x_i \neq x_k$ pentru orice $i \in \{1,2,\ldots,k-1\}$

```
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("n = "))
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,... ,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 soluţiei**

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

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

- **Condiţii interne (finale)** – crescător=>distincte

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

- **Condiţii 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 inutil, 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 multimii
{{1,...,{n}}}-metoda 2")
back(1)
```

Submulțimi

- ## Submulțimile mulțimii {1,2,... ,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} -> vectorul [0,1,1,0,1]

Generare de submulțimi = generare de șiruri binare de lungime n

Submulțimi

- **Reprezentarea soluţiei**

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

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

- **Condiţii interne (finale)**

- **Condiţii de continuare (!!pentru x_k)**

```python
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 multimii {{ {','.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 = $\{x_1, x_2, \ldots, x_k\}$ cu

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

n = 3

1 + 1 + 1

1 + 2

3

Partițiile unui număr natural n

- **Reprezentarea soluţiei**

  $x = \{x_1, x_2, \ldots, x_k\}$, unde

  $x_i \in \{1, \ldots, n\}$

- **Condiţii interne (finale)**

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

  Pentru unicitate: $x_1 \leq x_2 \leq \ldots \leq x_k$

- **Condiţii de continuare**

  $x_{k-1} \leq x_k \Longrightarrow x_k \in \{x_{k-1}, \ldots, n\}$

  $x_1 + x_2 + \ldots + x_k \leq n$

```
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)
```

```python
#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)
```

Subşiruri crescătoare lungime maximă

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

Subşiruri crescătoare lungime maximă

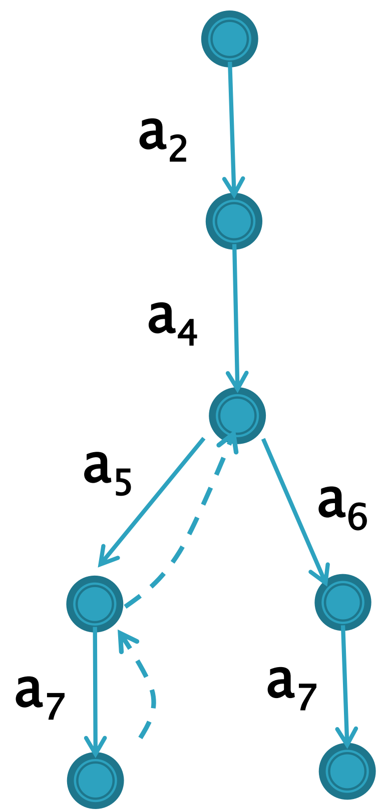
- Un subşir crescător de lungime maximă **începe cu o poziţie i** cu **`lung[i] = lmax`**

- Următorul element după $\mathbf{a_i}$ este un $\mathbf{a_j}$ cu proprietăţile
  - i<j
  - $\mathbf{a_j > a_i}$
  - **`lung[j] = lung[i]-1`**

  = **acei a_j posibili succesori ai lui i, pentru care se realizează egalitate în relaţia de recurenţă** (în PD se reţine în succ[i] un singur astfel de j)

Subşiruri 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



1, 4, 6, 11

1, 4, 5, 11

Subşiruri crescătoare lungime maximă

- **Reprezentarea soluţiei**

  $x = \{x_1, x_2, \ldots, x_{lmax}\}$, unde

  $x_k \in \{1, \ldots, n\}$ poziţie din vectorul a

- **Condiţii interne (finale)**

  $ax_1 < ax_2 < \ldots < ax_{lmax}$

- **Condiţii de continuare**

  $lung[a[x_1]] = lmax$

  $x_{k-1} < x_k$, $ax_{k-1} < ax_k$, $lung[x_k]=lung[x_{k-1}]-1$

```python
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)
```