
# Tehnici de programare

# Metoda backtracking

ovidiu.banias@upt.ro

# Utiliatate. Exemplificare


Se doreşte spargerea unui cifru (parolă) format din 4 cifre. Se presupune că există o funcție care primeşte ca parametru o combinație de 4 cifre şi returnează 0 (combinație incorectă) sau 1 (combinație corectă).

# Backtracking. Introducere

**Backtracking** = “to go back to an earlier point in a sequence”

**Utilitate** - rezolvarea problemelor cu următoarele proprietăți:

O soluție are forma unui vector

$$S_v = x_1, x_2, \ldots, x_n \,/\, x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n, v = \overline{1,\ldots}$$

Mulțimile $A_1, A_2, \ldots, A_n$ sunt finite având elemente aflate într-o relație de ordine bine stabilită

Se caută soluția/soluțiile valide în spațiul tuturor soluțiilor

Nu există altă rezolvare cu timp de rulare mai rapid

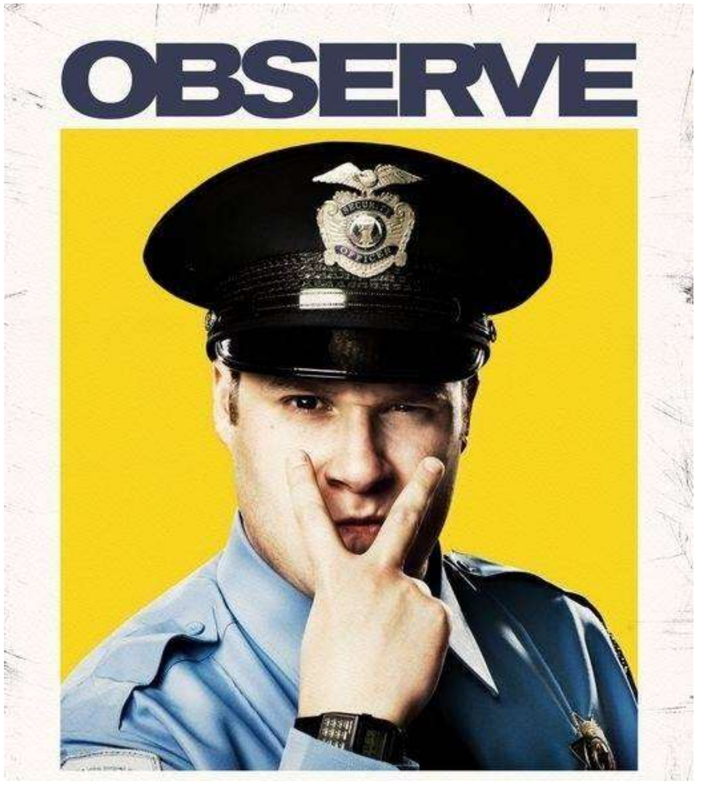
# Backtracking. Observații

Mulțimile $A_i \,/\, i = \overline{1,n}$ pot fi identice

$x_i \,/\, i = \overline{1,n}$ pot fi şi vectori

Numărul de elemente ale soluției $S$ poate fi sau nu cunoscut; depinde de fiecare problemă

Dacă se caută o singură soluție, algoritmul se opreşte forțat pentru a economisi timp



# Backtracking. Analiza complexității


**Produs cartezian**: Se doreşte spargerea unui cifru format din 4 cifre. Se presupune că există o funcție care primeşte ca parametru o combinație de 4 cifre şi returnează 0 (combinație incorectă) sau 1 (combinație corectă).

$$S_v = x_1, x_2, \ldots, x_n \,/\, x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n, v = \overline{1,\ldots}$$

$$n = 4, A \overset{(not)}{=} A_i \,/\, i = \overline{1,n} \Rightarrow S_v = x_1 x_2 x_3 x_4 \,/\, x_j \in \{0..9\}, j = \overline{1,4}$$

Generând produsul cartezian $S_v = A \times A \times A \times A$ se baleiază spațiul tuturor soluțiilor:

$$S_1 = (0,0,0,0)\ S_2 = (0,0,0,1)\ S_3 = (0,0,0,2) \ldots$$

$$S_{10} = (0,0,0,9) \ldots S_{100} = (0,0,9,9) \ldots$$

Rezolvare : adunare cu 1 în baza 10

# Backtracking. Analiza complexității

**Permutări**: Se doreşte generarea tuturor permutărilor de 4 elemente.

---

$$n = 4, A \overset{(not)}{=} A_i \,/\, i = \overline{1,n} \Rightarrow S_v = x_1x_2x_3x_4 \,/\, x_j \in \{1..4\},\ j = \overline{1,4}$$

Se poate folosi produsul cartezian

$$S_1 = (1,1,1,1)\ S_2 = (1,1,1,2)\ S_3 = (1,1,1,3)\ S_4 = (1,1,1,4)$$
$$S_5 = (1,1,2,1)\ S_6 = (1,1,2,2)\ S_7 = (1,1,2,3)\ldots$$

Câte soluții posibile? Câte valide?

# Backtracking. Cu alte cuvinte

- (Backtracking == Brute-force) ?
- Se generează toți candidații parțiali la “titlul” de soluție
- Candidații la soluție se construiesc pe vectori unidimensionali/bidimensionali
- Generarea candidațiilor se face în pași succesivi (înainte şi inapoi)
- După fiecare pas se poate face o validare pentru reducerea numărului căutărilor în spațiul soluțiilor
- Când s-a ajuns la o anumită dimensiune a vectorului, se verifică dacă candidatul parțial (vectorul) este sau nu o soluție
- Se alege soluția/soluțiile din candidații parțiali după criterii impuse de problemă

# Backtracking. Permutări.


**Generare permutări** mulțime de 4 elemente.

- proiectare algoritm generare permutări pornind de la un exemplu

**Pas 1**: Se utilizează un vector v cu 4 elemente.

- v[i] : elementul de pe poziția i din permutare

| | | | | |
|---|---|---|---|---|
| *v*: | **3** | **1** | **2** | **4** |
| index : | 1 | 2 | 3 | 4 |

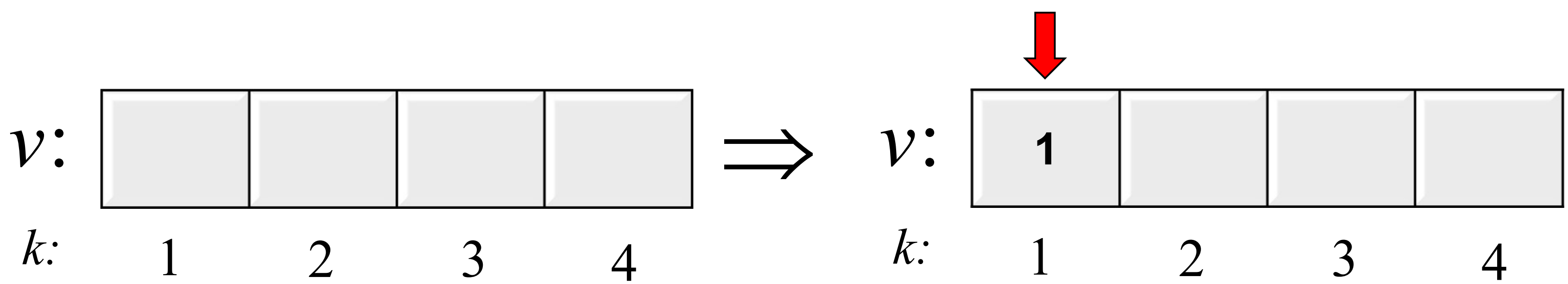
- v[3]=2 → elementul de pe poziția 3 din permutare este 2

# Backtracking. Permutări.

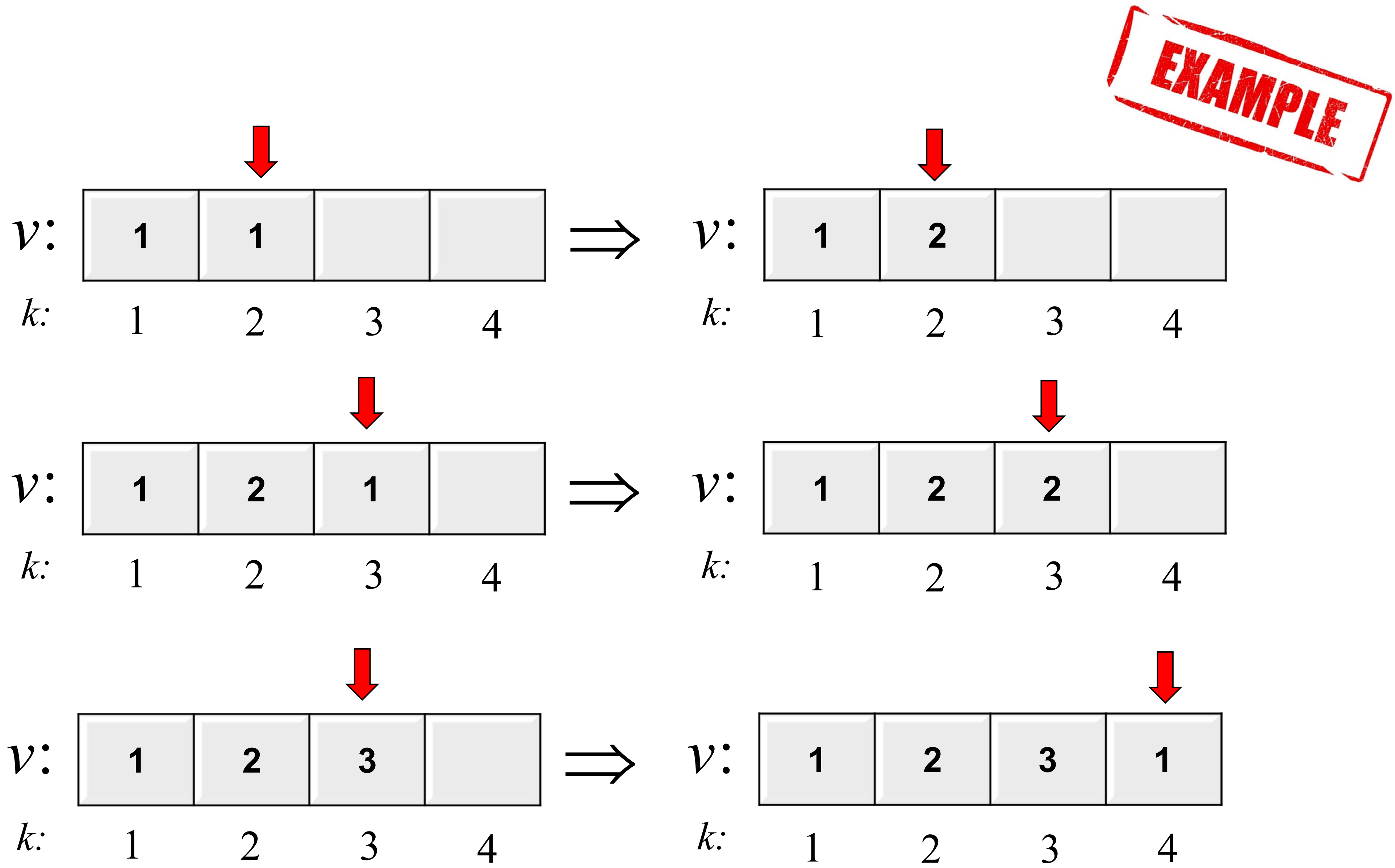

**Pas 2**: Se completează vectorul **v** de la stânga la dreapta cu valori **succesive** de la **1** la **n**.

- dacă s-a ajuns cu completarea până la un index k şi nu există duplicate până la indexul respectiv, se continuă completarea cu indexul k+1 (k<n).

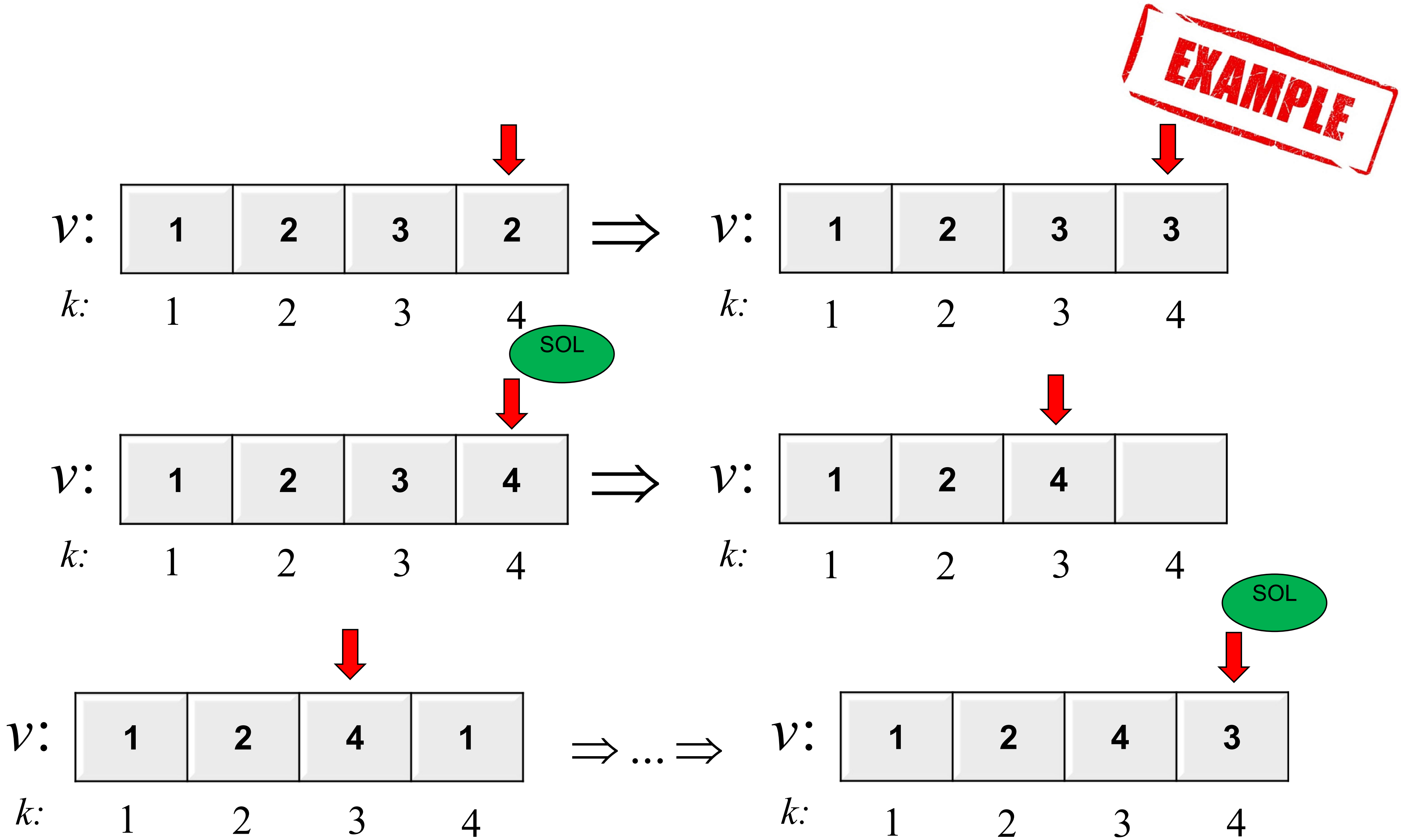
- dacă s-a ajuns cu completarea până la un index k şi nu se mai poate completa (s-a ajuns la valoarea n) şi există duplicate până la indexul respectiv, se continuă completarea cu indexul k-1(k>0).



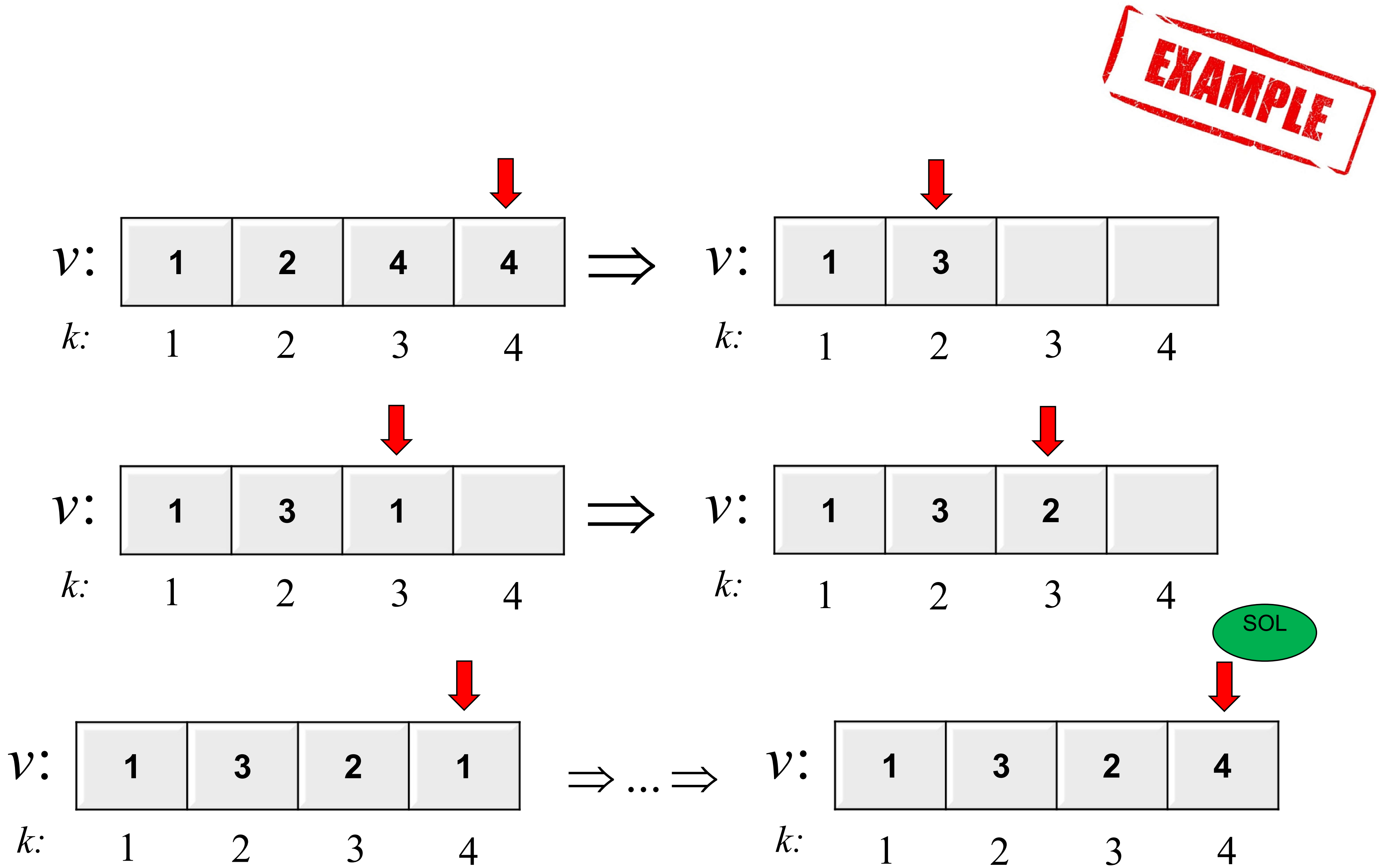
# Backtracking. Permutări.



# Backtracking. Permutări.



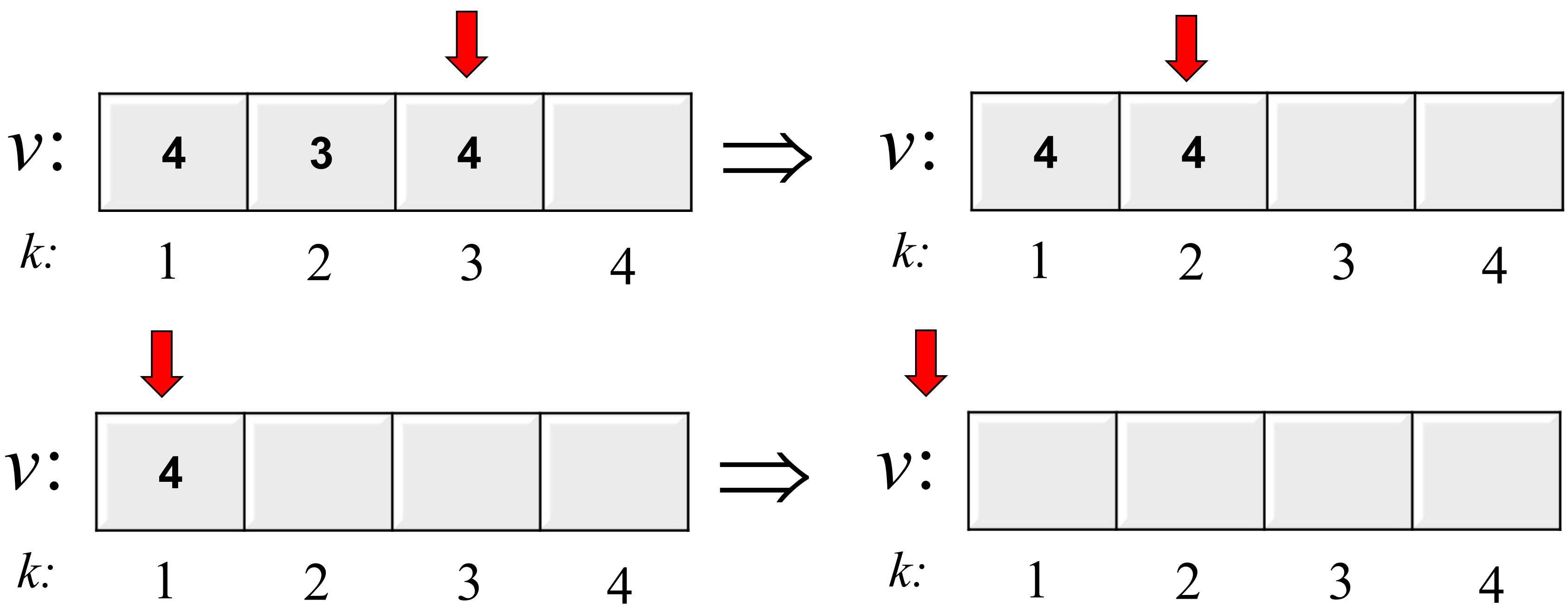
# Backtracking. Permutări.



# Backtracking. Permutări.


**Pas 3**: Se aleg soluțiile dintre candidați. Condiția este ca toate elementele vectorului să fie completate şi diferite între ele.

**Pas 4**: Se repetă algoritmul până când nu se mai îndeplineşte condiția: indexul **k** al vectorului **>0** (stiva este goală).



# Backtracking. Preambul implementare

- Pentru generarea tuturor soluțiilor se foloseşte o structură de date de tip stivă, **v**. Vârful stivei se notează cu **k**

- Algoritmul ciclează, adăugând/modificând/ştergând valori din vârful stivei
  - inițializează valoare element din vârful stivei – funcția **Init(k)**
  - modificare valoare element din vârful stivei – funcția **Successor(k)**
  - validarea elementului din vârful stivei – funcția **Valid(k)**
  - dacă elementul din vârful stivei este valid, putem avea un cantidat la soluție

    - funcția **Solution(k)**, iar în caz favorabil, afişare - **Print(k)**
  - 3 variante de poziționare în stivă :
    - Nu se modifică poziția – **k** rămâne neschimbat
    - Se urmăreşte adăugarea unui nou element în stivă – **k++**
    - Se coboară o poziție în stivă pentru că elementul curent din vârf nu mai satisface condițiile problemei– **k--**

# Backtracking. Implementare

## Funcția ***Init***

```
void Init(int k){ // k - vârful stivei

  v[k]=0; //iniţializează/resetează, valoarea din
     // vârful stivei
}
```

# Backtracking. Implementare

## Funcția ***Succesor***

```
int Succesor(int k){

    if (v[k]<n){   // se poate creşte valoarea din vârf
        v[k]++;    // se incrementează valoarea din vârf
        return 1; // funcţia a avut success
    }
    else     // nu se poate creşte valoarea din vârf
        return 0;
}
```

- Face următorul pas în căutarea candidatului în spațiul tuturor soluțiilor numai dacă condițiile problemei o permit – în cazul de față : valoarea curentă din vârful stivei este mai mică decât numarul maxim posibil n

- Returnează **1** sau **0** dacă s-a incrementat sau nu valoarea din vârful stivei

# Backtracking. Implementare

## Funcția ***Valid***

```
int Valid(k){

    for (i=1;i<k;i++) // verifică dacă elementul din
        if (v[i]==v[k]) return 0;  // vârf este diferit de
            // elemente precedente din stivă
    return 1;

}
```

- Se apelează doar în cazul în care funcția Successor a returnat valoarea **1**

- Verifică dacă valoarea curentă din vârful stivei (valoarea setată de către funcția Successor) respectă condițiile problemei – în cazul de față : elementele din stivă să fie diferite între ele

# Backtracking. Implementare

## Funcțiile ***Solution*** *și* ***Print***

```
int Solution(k){
    return (k==n);
}
void Print(){
    printf("%d : ",++m);
    for (i=1;i<=n;i++)
        printf("%d ",v[i]);
     printf("\n");
}
```

- Verifică condiția impusă de problemă ca valorile actuale din stivă (candidatul la soluție) să reprezinte o soluție, în cazul de față : să fie completate **n** elemente din stivă
- Validările intermediare asupra elementelor vectorului au fost făcute cu ajutorul funcției **Valid**.
- Se afişează elementele stivei (vectorul **v**)

# Backtracking. Implementare

## Rutina standard

```
void Back(int n){

    k=1; Init(k);

    while (k>0){ // cât timp stiva nu e vidă
        isS=0;isV=0;
        if (k<=n) // nu face sens depăşirea nivelului n în stivă
            do{         // repetă cât timp...
                isS=Succesor(k);
                if (isS) isV=Valid(k);
            } while (isS && !isV); // ...există succesor dar nu este valid
        if (isS) //este succesor si este valid
            if (Solution(k))  // verifică candidatul la soluţie
                Print();  // afişează soluţia
            else { // dacă nu este soluţie
                k++; Init(k); // creşte vârful stivei şi iniţializează
            }
        else  // nu există succesor pt. valoarea curentă din stivă
            k--; // -> se coboară o poziţie în stivă
    }
}
```

# Backtracking. Observații

- rutina standard de backtracking este de multe ori identică pt. 2 probleme diferite
- funcțiile **Init/Successor/Valid/Solution/Print** diferă în funcție de problemă
- codul poate fi restrâns, renunțând la cele 4 funcții
- lucrul iterativ cu stiva → **backtracking iterativ**
- rutina standard poate fi implementată şi recursiv → **backtracking recursiv** (mai uşor de implementat şi urmărit, necesită memorie suplimentară în Stivă)

# Backtracking. Exerciții

- Permutări
- Aranjamente
- Combinări
- Problema reginelor