**Metoda Backtracking - Prezentarea generală a metodei**

Metoda backtracking poate fi folosită în rezolvarea a diverse probleme. Este o metodă lentă, dar de multe ori este singura pe care o avem la dispoziție!

Introducere

Metoda backtracking poate fi aplicată în rezolvarea problemelor care respectă următoarele condiții:

* soluția poate fi reprezentată printr-un tablou x[]=(x[1], x[2], ..., x[n]), fiecare element x[i] aparținând unei mulțimi cunoscute Ai;
* fiecare mulțime Ai este finită, iar elementele ei se află într-o relație de ordine precizată – de multe ori cele n mulțimi sunt identice;
* se cer toate soluțiile problemei sau se cere o anumită soluție care nu poate fi determinată într-un alt mod (de regulă mai rapid).

Algoritmul de tip backtracking **are la bază un principiu extrem de simplu:** construiește vectorul x[] (numit **vector soluție**) pas cu pas , astfel:

La fiecare pas k, începând (de regulă) de la pasul 1, se prelucrează elementul curent x[k] al vectorului soluție:

* x[k] primește pe rând valori din mulțimea corespunzătoare Ak;
* la fiecare pas se verifică dacă configurația curentă a vectorului soluție poate duce la o soluție finală – dacă valoarea lui x[k] este corectă în raport cu x[1], x[2], … x[k-1]:
  + dacă valoarea nu este corectă, elementul curent X[k] primește următoarea valoare din Ak sau revenim la elementul anterior x[k-1], dacă X[k] a primit toate valorile din Ak **– se face un pas înapoi;**
  + dacă valoarea lui x[k] este corectă (avem o soluție parțială), se verifică existența unei soluții finale a problemei, existând 2 posibilități :
    - dacă configurația curentă a vectorului soluție x reprezintă soluție finală (de regulă) o afișăm;
    - dacă nu am identificat o soluție finală trecem la următorul element, x[k+1], și reluăm procesul pentru acest element **– se face un pas înainte**.

Pe măsură ce se construiește, vectorul soluție x[] reprezintă o **soluție parțială** a problemei. Când vectorul soluție este complet construit, avem o **soluție finală** a problemei.

### **Exemplu**

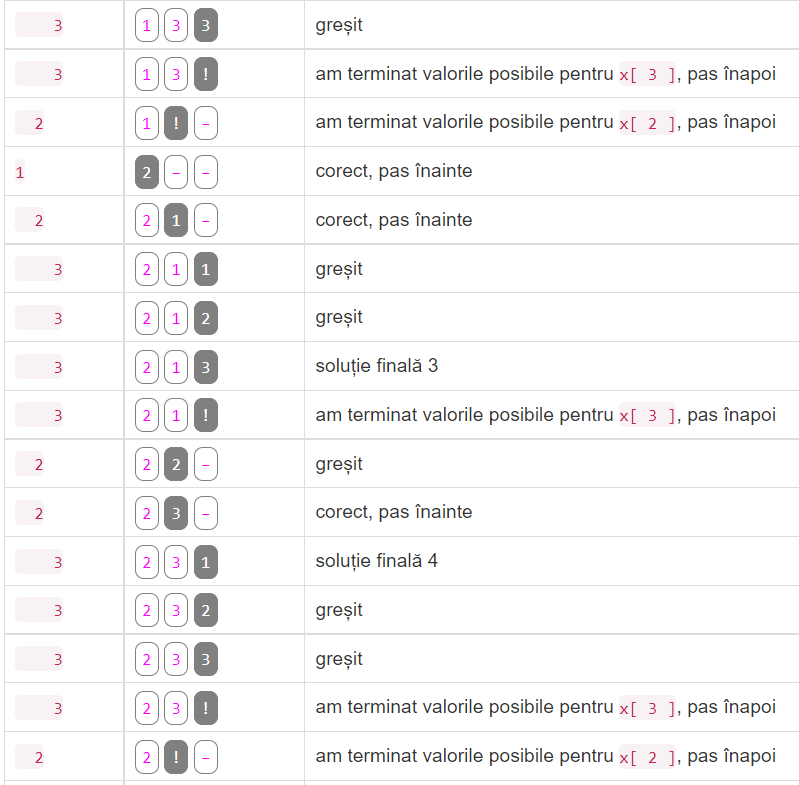
Să rezolvăm următoarea problemă folosind metoda backtracking, “cu creionul pe hârtie”: Să se afișeze permutările mulțimii *{1, 2, 3}*.

Se știe că un șir de numere reprezintă o permutare a unei mulțimi M dacă și numai dacă conține fiecare element al mulțimii M o singură dată. Altfel spus, în cazul nostru:

* are exact 3 elemente;
* fiecare element este cuprins între 1 și 3;
* elementele nu se repetă.

Pentru a rezolva problemă vom scrie pe rând valori din mulțimea dată și vom verifica la fiecare pas dacă valorile scrise duc la o permutare corectă**:(imaginea nu trebuie copiată, doar să parcurgeți ca să înțelegeți) !!!**







Pentru a putea realiza un algoritm backtracking pentru rezolvarea unei probleme trebuie să răspundem la următoarele întrebări:

1. **Ce memorăm în vectorul soluție x[]?** Uneori răspunsul este direct; de exemplu, la generarea permutărilor vectorul soluție reprezintă o permutare a mulțimii A={1,2,...,n}. În alte situații, vectorul soluție este o reprezentare mai puțin directă a soluției; de exemplu**,**[**generarea submulțimilor unei mulțimi**](https://www.pbinfo.ro/articole/137/generarea-submultimilor) folosind vectori caracteristici sau Problema reginelor.
2. **Ce valori poate lua fiecare element x[k] vectorului soluție și câte elemente poate avea x[]?** Altfel spus, care sunt mulțimile Ak. Vom numi aceste restricții **condiții externe**. Cu cât condițiile externe sunt mai restrictive (cu cât mulțimile Ak au mai puține elemente), cu atât va fi mai rapid algoritmul!
3. **Ce condiții trebuie să îndeplinească x[k] ca să fie considerat corect?** Elementul x[k] a primit o anumită valoare, în conformitate ce condițiile externe. Este ea corectă? Poate conduce la o soluție finală? Aceste condiții se numesc **condiții interne** și în ele pot să intervină doar x[k] și elementele x[1], x[2], …, x[k-1]. Elementele x[k+1], …, x[n] nu poti apărea în condițiile interne deoarece încă nu au fost generate!!!
4. **Am găsit o soluție finală?** Elementul x[k] a primit o valoare conformă cu condițiile externe, care respectă condițiile interne. Am ajuns la soluție finală sau continuăm cu x[k+1]?

**Exemplu.** Pentru problema [generării permutărilor](https://www.pbinfo.ro/articole/11234/generarea-permutarilor) mulțimii A={1,2,3,…,n}A={1,2,3,…,n}, condițiile de mai sus sunt:

1. Vectorul soluție conține o permutare a mulțimii AA;
2. **Condiții externe:** x[k]∈{1,2,3,…,n} sau x[k]=1,2,3,...,n , pentru k=1,2,....,n
3. **Condiții interne:** x[k]≠x[i]x[k]≠x[i], pentru i=1,2,3,....,k−1¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯i=1,k−1¯
4. **Condiții de existență a soluției:** k=n

## Algoritmul general

Metoda backtracking poate fi implementată iterativ sau recursiv. În ambele situații se se folosește o structură de deate de tip **stivă**. În cazul implementării iterative, stiva trebuie gestionată intern în algoritm – ceea ce poate duce la dificulăți în implementare. În cazul implementării recursive se folosește spațiu de memorie de tip stivă – STACK alocat programului; implementarea recursivă este de regulă mai scurtă și mai ușor de înțeles.

Următorul subprogram recursiv prezintă algoritmul la modul general:

* la fiecare apel BACK(k) se generează valori pentru elementul x[k] al vectorului soluție;
* instrucțiunea Pentru modelează condițiile externe;
* subprogramul OK(k) verifică condițiile interne
* subprogramul Solutie(k) verifică dacă configurația curentă a vectorului soluție reprezintă o soluție finală
* subprogramul Afisare(k) tratează soluția curentă a problemei – de exemplu o afișează!

**subprogram BACK(k)**

┌ pentru fiecare element i din A[k] executa

│ x[k] ← i

│ ┌ daca OK(k) atunci

│ │ ┌ daca Solutie(k) atunci

│ │ │ Afisare(k)

│ │ │ altfel

│ │ │ BACK(k+1)

│ │ └■

│ └■

└■

**Observații:**

* de cele mai multe ori mulțimile A sunt de forma A={1,2,3,….,n} sau A={1,2,3,….,m} sau A={a,a+1,a+2,….,b} sau o altă formă astfel încât să putem scrie instrucțiunea Pentru conform specificului limbajului de programare folosit – eventual folosind o structură repetitivă de alt tip! Dacă este necesar, trebuie realizate unele transformări încât mulțimile să ajungă la această formă!
* elementele mulțimii A pot fi in orice ordine. Contează însă ordinea în care le vom parcurge în instrucțiunea Pentru, deoarece în probleme este precizată de obicei o anumită ordine în care trebuie generate soluțiile:
  + dacă parcurgem elementele lui A în ordine crescătoare vom obține soluții în ordine lexicografică;
  + dacă parcurgem elementele lui A în ordine descrescătoare vom obține soluții în ordine invers lexicografică.
* în anumite probleme determinarea unei soluții finale nu conduce la întreruperea apelurilor recursive. Un exemplu este [generarea submulțimilor unei mulțimi](https://www.pbinfo.ro/articole/137/generarea-submultimilor). În acest caz algoritmul de mai sus poate fi modificat astfel:

daca OK(k) atunci

│ ┌ daca Solutie(k) atunci

│ │ Afisare(k)

│ └■

│ BACK(k+1)

└■

Trebuie să ne asigurăm că apelurile recursive se opresc!

**Următoarea secvență C++ oferă un șablon pentru rezolvarea unei probleme oarecare folosind metoda backtracking.**

#include <fstream>

using namespace std;

int x[10] ,n;

int Solutie(int k){

// x[k] verifică condițiile interne

// verificare dacă x[] reprezintă o soluție finală

return 1; // sau 0

}

int OK(int k){

// verificare conditii interne

return 1; // sau 0

}

void Afisare(int k)

{ // afișare/prelucrare soluția finală curentă

}

void Back(int k){

for(int i = A ; i <= B ; ++i) **//  În practică A și B vor avea valori specifice problemei:**

{

x[k]=i;

if( OK(k) )

if(Solutie(k))

Afisare(k);

else

Back(k+1);

}

}

int main(){

//citire date de intrare

Back(1);

return 0;

}

De multe ori condițiile de existență a soluției sunt simple și nu se justifică scrierea unei funcții pentru verificarea lor, ele putând fi verificate direct în funcția Back().

De cele mai multe ori, rezolvarea unei probleme folosind metoda backtracking constă în următoarele:

1. stabilirea semnificației vectorului soluție;
2. stabilirea condițiilor externe;
3. stabilirea condițiilor interne;
4. stabilirea condițiilor de existența a soluției finale;
5. completarea adecvată a șablonului de mai sus!

## Complexitatea algoritmului

**Algoritmii Backtracking sunt exponențiali.** Complexitatea depinde de la problemă la problemă dar este de tipul O(an).