INFORMATICĂ · CLASA A XI-A · METODA BACKTRACKING · LECȚIA 6

Backtracking (1): principiul metodei și generarea permutărilor

Durată: 60 min Nivel: mate-info Înainte: lecțiile xi-03

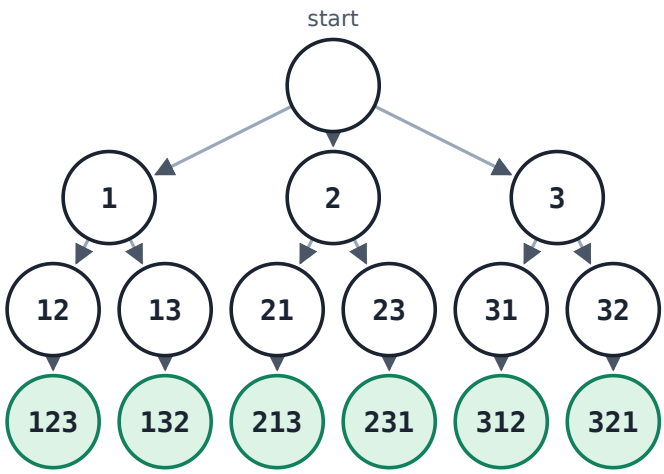
CE VEI ȘTI DUPĂ LECȚIE

- Înțelegi când se folosește backtracking: generarea tuturor soluțiilor care respectă niște condiții.
- Cunoști schema generală: vectorul soluție x, funcțiile valid, solutie, afisare, back(k).
- Generezi permutările mulțimii {1, 2, ..., n} în ordine lexicografică.
- Rezolvi problema celor n regine.

1. Ideea: construim soluția pas cu pas

Ai un lacăt cu 3 cifre și ai uitat codul. Încerci 000, 001, 002, ... Dacă știi că „cifrele sunt distincte", nu mai încerci 112: renunți imediat la combinațiile care încalcă regula. Aceasta este metoda **backtracking** (în română: „căutare cu revenire"): generăm **toate** soluțiile unei probleme (sau găsim una) construindu-le **element cu element**, `x[1], x[2], …, x[n]`. La fiecare pas încercăm toate valorile posibile; dacă o valoare încalcă o condiție, renunțăm la ea și **ne întoarcem** să încercăm alta.

Exemple: toate permutările, toate submulțimile, toate modurile de a plăti o sumă, toate drumurile într-un labirint.



nivelul k alege x[k]; valorile repetate sunt tăiate de valid

Arborele de căutare

Așa arată căutarea pentru permutările lui {1, 2, 3}: fiecare nivel alege valoarea pentru `x[k]`, iar frunzele (verzi) sunt soluțiile, în ordinea 123, 132, 213, 231, 312, 321 (adică 3! = 6).

DE CE AȘA?

De ce nu generăm toate șirurile de n numere și apoi alegem permutările? Pentru n = 10 ar fi 10^{10} șiruri, față de 10! = 3 628 800 permutări. Backtracking verifică **în timp ce construiește** (funcția `valid`) și nu mai continuă pe ramurile care nu pot duce la soluție. Se numește „tăiere" a arborelui (pruning).

2. Schema generală

```cpp
#include <iostream>
using namespace std;

int x[20], n;

bool valid(int k)                  // x[1..k] poate face parte dintr-o permutare?
{
    for (int i = 1; i < k; i++)
        if (x[i] == x[k]) return false;     // valoarea x[k] a mai apărut
    return true;
}

void afisare()
{
    for (int i = 1; i <= n; i++) cout << x[i] << ' ';
    cout << '\n';
}

void back(int k)                   // completăm poziția k
{
    for (int v = 1; v <= n; v++)           // toate valorile posibile pentru x[k]
    {
        x[k] = v;
        if (valid(k))
        {
            if (k == n) afisare();         // soluție completă
            else back(k + 1);              // trecem la poziția următoare
        }
    }
}

int main()
{
    cin >> n;
    back(1);
    return 0;
}
```

DE CE AȘA?

De ce nu „ștergem" `x[k]` după încercare? La următoarea iterație a lui `for`, `x[k]` primește oricum o valoare nouă, iar pozițiile mai mari se rescriu când ajungem la ele. „Întoarcerea" se face automat: când `back(k + 1)` se termină (a încercat toate valorile), revenim în `back(k)`, care încearcă următoarea valoare pentru `x[k]`. Stiva de apeluri *este* memoria căutării.

Urmărire (n = 3, primele pași; × = `valid` returnează fals):

k	v încercat	x[1..k]	valid?	acțiune
1	1	1	da	`back(2)`
2	1	1 1	×	următorul v
2	2	1 2	da	`back(3)`
3	1	1 2 1	×	
3	2	1 2 2	×	
3	3	1 2 3	da	k = n: **afișăm 1 2 3**
2	3	1 3	da	`back(3)` … (se continuă: 1 3 2)

Intrare / ieșire

Intrare	Ieșire

`3`	`1 2 3`
	`1 3 2`
	`2 1 3`
	`2 3 1`
	`3 1 2`
	`3 2 1`

`1`	`1`

Varianta cu vector de „folosit”

Verificarea din `valid` parcurge toate pozițiile anterioare (O(n)). Un vector `folosit[v]` o face în O(1):

```cpp
#include <iostream>
using namespace std;

int x[20], n;
bool folosit[20];
int cnt;

void back(int k)
{
    for (int v = 1; v <= n; v++)
        if (!folosit[v])
        {
            x[k] = v;
            folosit[v] = true;
            if (k == n) cnt++;              // aici ai putea afișa x[1..n]
            else back(k + 1);
            folosit[v] = false;             // „eliberăm” valoarea la întoarcere
        }
}

int main()
{
    cin >> n;
    back(1);
    cout << cnt;
    return 0;
}
```

DE CE AȘA?

De ce trebuie `folosit[v] = false` după apel? Când ne întoarcem, valoarea `v` nu mai este în `x[1..k]` : trebuie să poată fi aleasă din nou pe alte ramuri. Fără această linie, după prima soluție nu s-ar mai genera nimic. Aceasta este „revenirea” (backtrack) propriu-zisă.

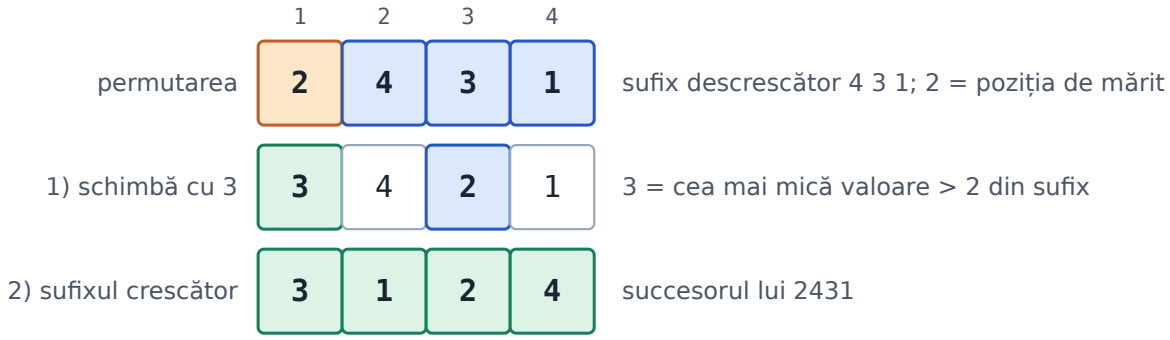
3. Cum generezi de mână, ca la Bac

LA BAC

Grilele de backtracking de la Bac sună așa: „Utilizând metoda backtracking se generează permutările … . Primele trei soluții sunt …, …, … . Care este a 7-a soluție?” sau „Care soluție se generează imediat după 2431?”. Trebuie să poți genera soluțiile **de mână**, în ordine. Reținerea ordinii: este cea **lexicografică** (ca în dicționar).

Succesorul unei permutări (cea care urmează imediat):

1. de la dreapta, urcă cât timp elementele sunt **descrescătoare** (sufixul nu se mai poate mări);
2. elementul din stânga sufixului este poziția de mărit;
3. schimbă-l cu cea mai mică valoare din sufix mai mare decât el;
4. așază sufixul în ordine **crescătoare**.



Succesorul

Pentru `2431` : sufixul descrescător este `431` ; poziția de mărit are valoarea 2; cea mai mică valoare din sufix mai mare ca 2 este 3; după schimb avem `3421` ; sufixul `421` se pune crescător: `124` . Rezultat: **3124**.

```
#include <iostream>
#include <algorithm>
using namespace std;

int p[20], n;

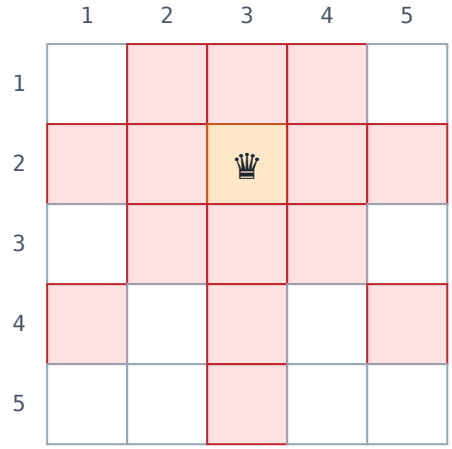
int main()
{
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> p[i];
    int i = n - 1;
    while (i >= 1 && p[i] > p[i + 1]) i--;        // urcăm cât sufixul e descrescător
    if (i == 0) { cout << "nu exista"; return 0; }  // permutarea n n-1 ... 1 este ultima
    int j = n;
    while (p[j] < p[i]) j--;                       // cea mai mică valoare din sufix >
p[i]
    swap(p[i], p[j]);
    reverse(p + i + 1, p + n + 1);                 // sufixul devine crescător
    for (int t = 1; t <= n; t++) cout << p[t] << ' ';
    return 0;
}
```

DE CE AȘA?

De ce funcționează? Sufixul descrescător este cea mai mare aranjare posibilă a elementelor lui: nu există nicio permutare „mai mare” cu aceeași poziție de început. Trebuie deci să mărim elementul de dinaintea lui, cu cel mai mic pas (următoarea valoare mai mare), iar apoi restul trebuie să fie cât mai mic: crescător. **De ce `reverse` ?** După schimb, sufixul este încă descrescător, iar inversarea unui șir descrescător îl face crescător.

4. Problema celor n regine

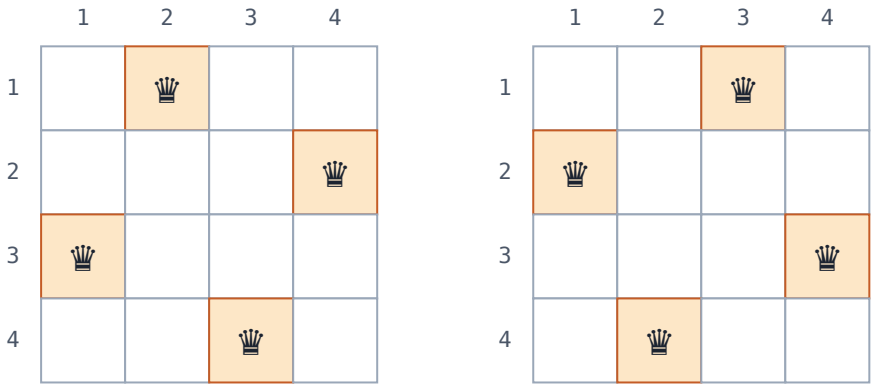
Așezăm `n` regine pe o tablă `n × n` astfel încât să nu se atace (nu pe aceeași linie, coloană sau diagonală). Punem regina `k` pe linia `k` ; `x[k]` = coloana ei. Astfel liniile sunt automat distincte; trebuie verificate doar coloanele și diagonalele.



coloană: x[i] = x[k]; diagonală: |x[i] − x[k]

Atacul reginei

Două regine `(i, x[i])` și `(k, x[k])` se atacă pe diagonală dacă diferența de linii este egală cu diferența de coloane: `|x[i] - x[k]| == k - i`.



Soluțiile pentru n = 4

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int x[20], n, cnt;

bool valid(int k)
{
    for (int i = 1; i < k; i++)
        if (x[i] == x[k] || abs(x[i] - x[k]) == k - i)    // aceeași coloană sau aceeași diagonală
            return false;
    return true;
}

void back(int k)
{
    for (int v = 1; v <= n; v++)
    {
        x[k] = v;
        if (valid(k))
        {
            if (k == n)
            {
                cnt++;
                if (n <= 4)                                // afișăm soluțiile doar pentru table mici
                {
                    for (int i = 1; i <= n; i++) cout << x[i] << ' ';
                    cout << '\n';
                }
            }
            else back(k + 1);
        }
    }
}

int main()
{
    cin >> n;
    back(1);
    cout << cnt;
    return 0;
}
```

DE CE AȘA?

De ce `abs(x[i] - x[k]) == k - i`? Mergând de la linia `i` la linia `k` (`k - i` rânduri mai jos), o regină ar trebui să se deplaseze pe orizontală tot cu `k - i` coloane ca să fie pe diagonala dintâi: la stânga sau la dreapta, deci diferența *în modul*. Condiția este în `valid` și taie ramuri întregi: pentru n = 8 sunt doar 92 de soluții, dar $8^8 \approx 16{,}7$ milioane de șiruri posibile.

Urmărire (n = 4, începutul căutării; × = invalid):

k	x[1..k]	motiv
1	1	valid
2	1 1	× aceeași coloană
2	1 2	× diagonală (diferență de coloane 1 = diferență de linii 1)
2	1 3	valid
3	1 3 1, 1 3 2	× coloană / diagonală
3	1 3 3, 1 3 4	× coloană / diagonală
…		întoarcere la `x[2]`: 1 4 … apoi la `x[1]` = 2

Intrare / ieșire

Intrare	Ieșire

`4`	`2 4 1 3`
	`3 1 4 2`
	`2`

`6`	`4` (numărul de soluții)

n	1	2	3	4	5	6	8
soluții	1	0	0	2	10	4	92

5. Greșeli frecvente

Greșeala	Efect	Corect
`valid(k)` care verifică și pozițiile > k	verifică valori nescrise	`for (i = 1; i < k; i++)`
uiți `folosit[v] = false` la întoarcere	se generează doar prima soluție	eliberează valoarea
apelezi `back(k)` în loc de `back(k + 1)`	recursie infinită	`back(k + 1)`
afișezi când `k == n - 1`	soluții incomplete	`k == n`
condiția de diagonală fără `abs`	pierzi soluții	`abs(x[i] – x[k]) == k – i`
pentru n mare afișezi toate soluțiile	timp enorm	numără, afișează doar câteva

6. La BAC

> **LA BAC**
>
> La Subiectul I apar 2–3 grile de backtracking: generare de mână (primele soluții, a k-a soluție, succesorul unei soluții, câte soluții) și câteodată cerința „care condiție trebuie pusă în `valid`?". Învață tiparul: *alegi o valoare pentru x[k]; verifici; treci la k + 1; la întoarcere încerci altă valoare*. La Subiectul II/III se cere rar programul; se cer mai ales rezultatele.

7. Exersează pe hârtie

Răspunsurile sunt la final: încearcă întâi singur.

1. Scrie toate permutările mulțimii {1, 2, 3, 4} care încep cu 3, în ordine lexicografică.

2. Care este permutarea generată imediat după `1432`? Dar după `3421`?

3. A câta permutare a mulțimii {1, 2, 3, 4} (în ordine lexicografică) este `2143`?

4. Câte permutări ale mulțimii {1, 2, 3, 4, 5} au 1 pe prima poziție?

5. Pentru `x = (2, 1, 2)`, ce returnează `valid(3)` pentru permutări? Explică.

6. Verifică pentru n = 4 că `x = (2, 4, 1, 3)` este o soluție la problema reginelor: pentru `k = 4`, calculează `|x[i] − x[4]|` și `4 − i` pentru `i = 1, 2, 3`.

7. Câte soluții are problema reginelor pentru n = 2? Dar pentru n = 3? Justifică.

8. Câte permutări trebuie generate pentru n = 10? Cât durează dacă fiecare se generează în 1 µs?

9. Cum modifici programul din secțiunea 2 ca să genereze permutările în ordine lexicografică **inversă**?

10. (BAC) Se generează, cu backtracking, permutările mulțimii {1, 2, 3, 4}. Primele trei soluții sunt 1234, 1243, 1324. Care este a 7-a soluție?

8. Mai departe (pentru cine vrea mai mult)

`next_permutation` din STL. În C++ ai deja funcția: `do { afisare(); } while (next_permutation(p + 1, p + n + 1));` (pornești cu vectorul sortat). Funcționează exact ca algoritmul de mână din secțiunea 3.

Permutări cu repetiții. Pentru cuvântul `aab` (litere care se repetă) sunt 3 anagrame distincte, nu 6. Soluție: numără aparițiile fiecărei litere și alege „litera următoare" doar dintre cele disponibile; număr de anagrame = n! / (n_1! · n_2! · …).

Optimizarea backtracking-ului. Pentru regine, în loc de `valid` cu o buclă, se păstrează trei vectori booleeni (coloane, diagonale `i + j`, diagonale `i − j`) și se verifică în O(1). Cu ei, n = 12 (14 200 de soluții) se rezolvă într-o fracțiune de secundă.

Limite. Backtracking-ul explorează un număr uriaș de variante (n! sau 2^n). Se folosește când n este mic (≤ ~12–20). Pentru n mare se caută alte metode: programare dinamică, greedy, euristici.

Rezumat

- Backtracking = construcție pas cu pas + renunțare la valorile invalide (+ întoarcere automată prin recursivitate).
- Schema: `back(k)` încearcă toate valorile pentru `x[k]`; dacă `valid(k)`: soluție (`k == n`) → afișare, altfel `back(k + 1)`.
- Permutări: valori distincte; n! soluții; succesorul se află de la dreapta.
- Regine: `x[k]` = coloana reginei de pe linia k; `valid` verifică coloana și diagonala.
- Cu `folosit[]` verificarea devine O(1), dar trebuie eliberat la întoarcere.

Probleme de lucru (pbinfo)

Începe cu acestea, în ordine: sunt alese să urmeze pașii din lecție.

- [ ] `#123` **permutari** — ușoară
  https://www.pbinfo.ro/probleme/123/permutari
- [ ] `#124` **permutari1** — ușoară
  Ordine invers lexicografică: bucla de la n la 1.
  https://www.pbinfo.ro/probleme/124/permutari1
- [ ] `#1281` **regine1** — ușoară
  https://www.pbinfo.ro/probleme/1281/regine1

Temă (pbinfo)

Trimite soluțiile pe pbinfo și notează punctajul obținut. Bifează ce ai terminat.

- [ ] `#3150` **permutari-pfp** — ușoară
  https://www.pbinfo.ro/probleme/3150/permutari-pfp
- [ ] `#3156` **perm-min-max** — ușoară
  https://www.pbinfo.ro/probleme/3156/perm-min-max
- [ ] `#3154` **anagrame-pfv** — ușoară
  https://www.pbinfo.ro/probleme/3154/anagrame-pfv
- [ ] `#3161` **permutari5** — ușoară
  https://www.pbinfo.ro/probleme/3161/permutari5
- [ ] `#2205` **permrep** — ușoară
  https://www.pbinfo.ro/probleme/2205/permrep

Pentru cei care vor mai mult

Probleme mai grele sau de concurs, pentru cine vrea să meargă dincolo de BAC.

- [ ] #1354 **varf** — ușoară
  https://www.pbinfo.ro/probleme/1354/varf

Grile din subiectele de BAC (pbinfo)

Exerciții de tip grilă din variantele de Bacalaureat, rezolvate online pe pbinfo, cu verificare automată a răspunsului.

- [ ] #414 **Exercițiul #414**
  se generează permutările mulțimii {1,2,3,4}: primele trei soluții date, care urmează
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=414
- [ ] #388 **Exercițiul #388**
  permutările cuvântului info: primele trei soluții și a câta este o soluție dată
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=388
- [ ] #396 **Exercițiul #396**
  procesul de generare a permutărilor mulțimii {1,2,...,n}: ce se întâmplă în timpul generării
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=396
- [ ] #403 **Exercițiul #403**
  în câte permutări ale mulțimii {I, N, F, O} vocalele apar pe poziții consecutive
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=403
- [ ] #410 **Exercițiul #410**
  generare în ordine lexicografică inversă a permutărilor
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=410
- [ ] #447 **Exercițiul #447**
  permutări de 4 obiecte: primele soluții date, care urmează
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=447
- [ ] #464 **Exercițiul #464**
  în câte permutări ale mulțimii {I, N, F, O} litera I apare pe prima poziție
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=464
- [ ] #468 **Exercițiul #468**
  numărul de permutări ale celor n elemente ale unei mulțimi
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=468

Citește și exersează în plus

- Metoda backtracking — prezentare generală https://www.pbinfo.ro/articole/16597/metoda-backtracking
- Generarea permutărilor https://www.pbinfo.ro/articole/11234/generarea-permutarilor
- VisuAlgo — Recursion & Backtracking (N-Queens, permutări) — animații pentru arborele de căutare și pentru problema celor n regine https://visualgo.net/en/recursion
- N-Queens interactiv — istoricul problemei celor 8 regine și numărul soluțiilor pentru fiecare n (engleză) https://en.wikipedia.org/wiki/Eight_queens_puzzle

- CSES — Chessboard and Queens — varianta cu pătrate interzise; bun pentru exersat după lecție (engleză) https://cses.fi/problemset/task/1624

Răspunsuri

Exersează pe hârtie

1. `3124`, `3142`, `3214`, `3241`, `3412`, `3421`.

2. După `1432`: sufixul descrescător este `432`, poziția de mărit are valoarea 1; cea mai mică valoare mai mare ca 1 din sufix este 2; după schimb `2431`, sufixul `431` se pune crescător: **`2134`**. După `3421`: sufixul descrescător este `421`, poziția de mărit are valoarea 3; cea mai mică valoare mai mare ca 3 este 4; după schimb `4321`, sufixul `321` crescător: **`4123`**.

3. A **8-a**: cele 6 permutări care încep cu 1 (1234 … 1432) sunt primele; apoi vin `2134` (a 7-a) și `2143` (a 8-a).

4. 4! = **24** (celelalte patru valori se permută pe pozițiile 2–5).

5. `false`: `x[1] = 2 = x[3]`, deci valoarea 2 a mai apărut (într-o permutare fiecare valoare apare o singură dată).

6. Pentru `k = 4`, `x[4] = 3`:

i	x[i]	diferența coloanelor (valoare absolută)	4 − i	atac?
1	2	1	3	nu
2	4	1	2	nu
3	1	2	1	nu

Coloanele sunt distincte și nicio diferență de coloane nu este egală cu `k − i`, deci regina 4 nu atacă pe nimeni. Verificările analoage pentru `k = 2, 3` reușesc, deci este soluție.

7. `n = 2`: **0** (a doua regină ar fi pe aceeași coloană sau pe diagonala primei). `n = 3`: **0**. Dacă regina 1 e pe coloana 1, regina 2 poate fi doar pe coloana 3 (coloana 1 e ocupată, coloana 2 e pe diagonală), iar regina 3 nu mai are loc (coloanele 1 și 3 sunt ocupate, coloana 2 este pe diagonala reginei 2). Cazul coloanei 3 este simetric; dacă regina 1 e pe coloana 2, regina 2 nu are nicio coloană liberă. (Pentru `n = 1` există o soluție: regina singură.)

8. 10! = **3 628 800** permutări; la 1 µs fiecare, circa **3,6 secunde**.

9. Bucla pentru valori merge descrescător: `for (int v = n; v >= 1; v--)`.

10. 2134 (primele 6 încep cu 1: 1234, 1243, 1324, 1342, 1423, 1432; a 7-a este prima cu 2: 2134).