Backtracking și elemente de combinatorică

Backtracking este o metodă de programare, adică un algoritm ce poate fi aplicat la o anume clasă de probleme. Algoritmul este destul de standard dar încurajăm depunerea de efort pentru buna înțelegere a modului de funcționare și nu memorarea template-ului de algoritm, pentru a ne putea ușor adapta de la o problemă la alta.

Să vedem care este clasa de probleme care se pot rezolva cu backtracking: soluția să fie sub formă de șir și se cer toate șirurile care îndeplinesc anumite proporietăți.

Un prim exemplu este generarea aranjamentelor de `m` elemente ale mulțimii {`1`, `2`, …, `n`}. Pentru că, așa cum spune și titlul, vom intersecta deseori backtracking cu combinatorica, vom stabili de pe acum exact ce înseamnă aranjamentele: șiruri de exact `m` elemente, distincte, cu valori naturale de la `1` la `n` fiind importantă ordinea elementelor din șir.

De exemplu, pentru `n=5` și `m=3`, câteva dintre șiruri ar fi:

1 2 3
1 2 4
1 2 5
1 3 2
1 3 4
1 3 5
1 4 5
1 5 2
. . .
5 4 3

Așa cum am spus, ordinea elementelor în soluție este importantă, în cazul aranjamentelor având `1 2 5` și `1 5 2` ca soluții distincte. După cum vedeți, noi ne propunem să generăm aceste șiruri și nu doar să stabilim numărul lor, cunoscut de la matematică prin formula: `n!/(n-m)!`

Un alt exemplu îl reprezintă generarea combinărilor de `m` elemente ale mulțimii {`1`, `2`, …, `n`}. Diferența față de aranjamente este că nu vom considera distincte două șiruri dacă elementele unuia se pot obține din cele ale celuilalt schimbându-le ordinea. Așadar, în cazul combinărilor, `1 2 5` și `1 5 2` reprezintă aceeași soluție. La implementare, pentru o clasă de șiruri care diferă doar prin permutarea elementelor vom stabili ca reprezentant pe acela cu elementele în ordine crescătoare. Așadar, se observă ușor că problema generării combinărilor este echivalentă cu aceea a generării de șiruri de lungime `m` cu elemente de la `1` la `n`, și cu elementele din fiecare șir *în prdine strict crescătoare*.

De exemplu, pentru `n=5` și `m=3`, șirurile generate de algoritmul de combinări ar fi:

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

De asemenea noi le vom genera, numărul lor fiind: `n!/(m!*(n-m)!)`.

Un al treilea exemplu clasic este acela al generării permutărilor de `n` elemente ale mulțimii {`1`, `2`, ...,`n`}. Asta înseamnă să așezăm în toate modurile, în ordine, toate numerele de la `1` la `n`, fiecare apărând o dată.

Pentru `n=3`, un algoritm de generare a permutărilor ar produce:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Numărul permutărilor cunoaștem că este `n!`.

Să facem o combinare a celor trei noțiuni pornind de la semnificația fiecăreia și să vedem ce relații mai stabilim între ele.

Observăm că permutările sunt de fapt aranjamente ale mulțimii {`1`, `2`, ...,`n`} pentru care `m` egal cu `n`. Aplicând acum formula aranjamentelor, vom avea `n!/(n-n)! = n!`, pentru că știm că `0!` este `1`.

Pornind de la semnificația combinărilor, ne gândim că orice combinare, dacă o permutăm în toate modurile, obținem un aranjament distinct. Dar o combinare de lungime `m` poate fi permutată în `m!` moduri, și atunci avem formula:

(Combinări de `n` luate câte `m`) `*` (permutări de `m`) = (Aranjamente de `n` luate câte `m`).

Dacă înlocuim în expresia de mai sus cu formulele scrise deja anterior pentru fiecare problemă în parte, obținem identitate.

Un al patrulea exemplu preliminar, și ultimul, este cel al generării elementelor produsului cartezian. Dacă avem `m` mulțimi, fiecare mulțime `i` fiind formată din {`1`, `2`, ... n_i}, un element al produsului cartezian are exact `m` elemente, cel de pe poziția `i` fiind un element oarecare extras din mulțimea a `i`-a. De exemplu, dacă avem `m=3` mulțimi și componența lor este {`1`, `2`, `3`}, {`1`, `2`}, {`1`, `2`}, toate elementele produsului cartezian al lor sunt șirurile:

1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2
3 1 1
3 1 2
3 2 1
3 2 2

Ca formulă de combinatorică, numărul de elemente este $n_1 * n_2 * \ldots * n_m$, adică produsul cardinalelor de la fiecare mulțime.

O concluzie care ne este de folos pentru ce avem de făcut în continuare este că toate cele patru probleme enunțate au soluții sub formă de vector și au mai multe soluții, care îndeplinesc anumite proprietăți. Tocmai asta vom încerca prin algoritmii de tip backtracking, să generăm soluții pentru astfel de probleme.

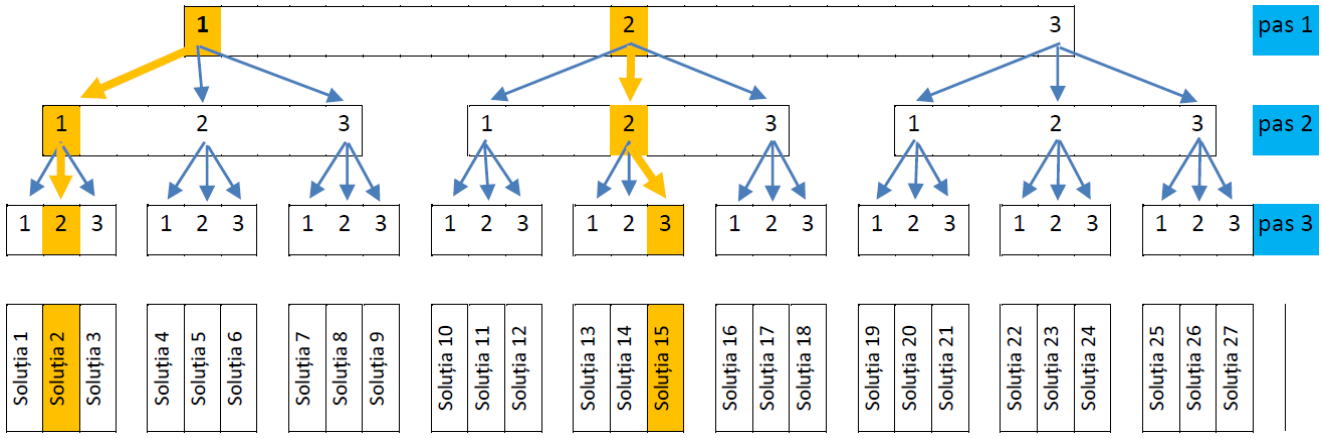
Așa cum suneam la început, problemele backtracking se rezolvă cu mici modificări față de un algoritm standard. Acesta se bazează pe recursivitatea cu mai multe autoapeluri și asta îl face să ajungă cu timp de executare mare în general.

Insist pe importanța bunei înțelegere a modului de funcționare ținând cont și de faptul că acesta nu este un lucru tocmai simplu, fiind vorba de recursivitate mai multe autoapeluri.

Să pornim de la următorul program:

```
#include <iostream>
using namespace std;
int n, m;
int x[30];
void backtrack(int pas) {
    if (pas == m+1) {
        for (int i=1;i<=m;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            x[pas] = i;
            backtrack(pas+1);
        }
}
int main () {
    cin>>n>>m;
    backtrack(1);
}
```

Îl vom analiza împreună cu următoarea figură.



Arborele este cel obținut în urma liniarizării algoritmului recursiv prezentat anterior pentru `n=3` și `m=3`. Pe ramura fara autoapel (`pas == m+1`) se ajunge cu `x` avand drept configurație fiecare lanț de sus în jos care se poate forma în arborele desenat (noi am îngroșat două dintre aceste lanțuri).

Încercăm să punctăm mai multe elemente prin care să înțelegem că se întâmplă asta.

- Pe linia `pas 1` (am numit-o așa pentru că este oglinda a ceea ce se întâmplă la apelul funcției în care parametrul are valoarea `1`) se găsesc toate valorile pe care le ia `x[1]`. Observăm că, pe rând, `x[1]` ia ca valori `1`, `2`, `3`, adică orice de la `1` la `n`.
- Pe linia `pas 2` avem toate valorile pe care le ia `x[2]`. Remarcati că `x[2]` ia valoarea `1` (de exemplu) de mai multe ori, adică pentru fiecare valoare pecare o ia `x[1]`. Adică dacă `x[1]` parcurge o singură dată setul de valori de la `1` la `n`, `x[2]` va parcurge acest set de mai multe ori.

- Mergând înainte cu analiza, pe linia `pas 3`, observăm că acolo și mai des se repetă valorile încercare. Adică, fiecare dntre `1`, `2`, ..., `n` ajung în `x[3]` pentru fiecare combinație posibilă (`x[1]`, `x[2]`). Este bine să facem un exercițiu, imaginându-se ce s-ar întâmpla dacă am mai avea un nivel, de cât de multe ori s-ar schimba și repeta valorile de acolo.

Pentru o mai bună înțelegere imaginați-vă un ceas electronic care afișează trei valori: ora, minutul, secunda. Pe parcursul unei zile ora parcurge o singură dată intervalul de valori posibile. Noi ne imaginăm ora ca fiind valoarea de la nivelul pas `1`, adică `x[1]`. Minutul (`pas 2`) își variază toate valorile posibile nu doar o dată, ci pntru fiecare oră diferită, iar secunda (`pas 3`) și mai des, pentru fiecare combinație posibilă oră, minut.

Cam așa funcționează și funcția recursivă prezentată mai sus.

Ea generează toate elementele produsulu carteziam a m mulțimi, fiecare având elemente de la `1` la `n`. La `pas = 1` considerăm că punem în `x[1]` fiecare valoare care poate fi extrasă din mulțimea `1`. La `pas = 2` se pune în `x[2]` fiecare valoare din mulțimea a doua (și am observat că ea se împerechează cu fiecare valoare extrasă din prima mulțime) etc.

Funcția are două lucruri de care depinde ceea ce se întâmplă: `n` și `m`.

- `m` reprezintă numărul de valori care se construiesc în `x`, adică lungimea unei soluții. Observăm că la o anume valoare a parametrului pas, noi generăm elementul corespunzător din soluție, apoi facem autoapel. Așadar depistăm finalizarea construcției unei soluții de lungime m după ce am făcut autoapelul pentru `pas=m`, adică atunci când ajungem cu pas la valoarea `m+1`.
- `n` reprezintă numărul de elemente din fiecare mulțime (noi am considerat cele m mulțimi ca fiind identice). Așadar forul care variaza între `1` și `n` fixează de fapt prin `i` toate valorile pe care le ia `x[pas]`, valoarea extrasă din mulțimea curentă. Un lucru de o deosebită importanță este că `i` este variabilă locală. Asta face ca la fiecare valoare a lui pas să rămână în stiva recurivității valoarea lui `i` la care a ajuns forul de la pasul respectiv iar când pe un pas mai mare forul (și totodată acel apel recursiv) se termină, se revine la pasul anterior și SE CONTINUĂ cu valoarea ce urmează celei incercate deja. Adică preia din stivă valoarea lui `i` de la acel pas și se continuă forul cu următoarea valoare. Pe de altă parte, observăm că de câte ori se avansează, forul se reia de la `1`.

Din ce am explicat mai sus putem deduce și că de câte ori pe un nivel s-au epuizat valorile de încercat, se revine la nivelul anterior și se face altă încercare (următoarea). De aici și denumirea algoritmului de backtracking, adică se face un pas înapoi când nu se mai poate.

O altă remarcă este că la trecerea la un pas superior (la autoapel, când la pasul curent s-a setat o valoare) se reiau de fiecare dată toate valorile posibile la acel pas, deci începând cu `1`. Așa se face ca pentru fiecare nouă valoare setată, pe pozițiile următoare ea se combină cu toate posibile.

Mai putem observa că `x` se comportă ca o stivă, adică o structură în care toate operațiile se fac la vârf. Ea merge în paralel cu stiva de autoapeluri, pas fiind vârful ei. O alegere la vârf, `x[pas]` face să urcăm un nivel în stivă (autoapel) și să facem acolo toate încercările posibile. Pe de altă parte, epuizarea valorilor de încercat în vârf face să coborâm un nivel și să încercăm acolo următoarea valoare.

Prezentăm încă două exemple în care variem puțin programul prezentat anterior.

1. Considerăm `m=3` și la pasul `1` valori în mulțime de la `0` la `23`, la pasul `2` și la pasul `3` valori în mulțime de la `0` la `59`. Nu vom mai merge așadar cu forul de la `1` la `n` la fiecare pas, ci în intervalele descrise mai sus. Vom folosi in vector `v` în care `v[1]=23`, `v[2]=59`, `v[3]=59` și la pasul pas vom merge de la `0` la `v[pas]`.

```
#include <iostream>
using namespace std;
int x[30];
int v[4];
void backtrack(int pas) {
    if (pas == 4) {
```

```
        for (int i=1;i<=3;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=0;i<=v[pas];i++) {
            x[pas] = i;
            backtrack(pas+1);
        }
}
int main () {
    v[1] = 23; v[2] = 59; v[3] = 59;
    backtrack(1);
}
```

Am îngroșat locurile din funcția recursivă unde au apărut modificări (la lungimea soluției și la intervalul de valori pentru fiecare element).

Efectul este să se afișeze toate momentele (oră, minut, secundă) de pe parcursul unei zile, în ordinea în care acestea se întâmplă.

0 0 0
0 0 1
0 0 2
...
0 0 59
0 1 0
0 1 1
0 1 2
...
23 59 59

2. Pentru un m dat, considerăm că fiecare dintre cele m mulțimi poate avea doar valorile 0 și 1. Așadar vom genera toate șirurile posibile de lungime `m` care au valori `0` și `1`.

```
#include <iostream>
using namespace std;
int x[30];
int m;
void backtrack(int pas) {
    if (pas == m+1) {
        for (int i=1;i<=m;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=0;i<=1;i++) {
            x[pas] = i;
            backtrack(pas+1);
        }
}
int main () {
    cin>>m;
    backtrack(1);
}
```

Pentru m=3 se obține:

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Algoritmul de mai sus este unul pe baza căruia putem afișa toate submulțimile mulțimii {1, 2, ..., m} dacă în loc să tipărim șirul de 0 și 1 tipărim indicii unde se află valoarea 1.

Adică putem schimba afișarea astfel:

```
for (int i=1;i<=m;i++)
    if (x[i] == 1)
        cout<<i<<" ";
cout<<"\n";
```

S-ar obține:

0 0 0	
0 0 1	3
0 1 0	2
0 1 1	2 3
1 0 0	1
1 0 1	1 3
1 1 0	1 2
1 1 1	1 2 3

Cele trei exemple prezentate sunt de produs cartezian al unor mulțimi. Analizând soluțiile în fiecare caz în parte observăm un lucru foarte important: acestea sunt generate în ordine lexicografică (aici nu ne referim la coloana a doua a tabelului aflat imediat mai sus ci doar la prima coloană, prima find ceea ce scoate algoritmul, a doua fiind o interpretare a fiecărei soluții generate).

Reamintim că la comparatea din punct de vedere lexicografic a două șiruri contează doar prima poziție unde în cele două șiruri valorile diferă, deci considerând că pe pozițiile anterioare valorile sunt identice, iar după poziția găsită nu mai contează cum sunt valorile.

Generarea în ordine lexicografică se întâmplă pentru că noi încercăm pe fiecare nivel valorile în ordine crescătoare și că odată un prefix generat acesta se ține de fapt fixat până când se generează toate soluțiile posibile cu el, acestea obținându-se încercând în ordine crescătoare valorile pe pozițiile următoare, deci tot lexicografic.

Revenind la exemplul inițial, cel cu m mulțimi cu elemente de la 1 la n fiecare, explicam mai sus că variabila i în care fixam valorile posibile de la un nivel trebuie să fie locală. Acum subliniem că x, vectorul soluție și variabilele care indică lungimea soluției (m) și setul de valori posibile la fiecare nivel (n) este comod să le ținem globale, ele fiind aceleași la fiecare autoapel.

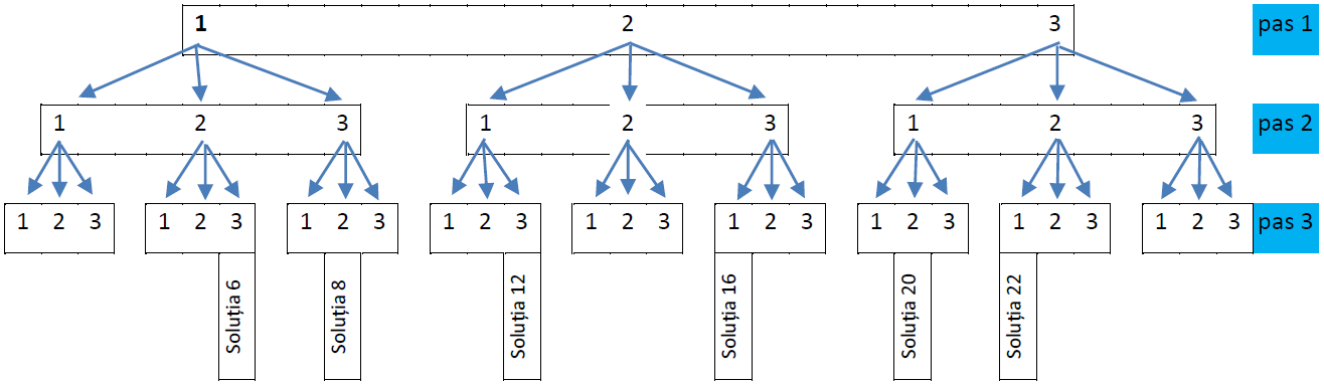
În exemplul de care aminteam, implementat în codul prezentat și al cărui desfășurător de autoapeluri apare în desenul de mai sus s-a generat produsul cartezian a m mulțimi, fiecare având elemente de la 1 la n. Altfel spus, s-au generat soluții de lungime m, fiecare element cu valori de la 1 la n și fără condiții suplimentare asupra elementelor unei soluții (orice combinație în condițiile de mai sus este permisă).

Soluțiile s-au generat în ordine lexicografică, adică întâi cele care încep cu valoarea minimă (1 în acest caz), apoi cele care încep cu al doilea minim etc. În general, pentru un prefix de soluție fixat, elementul de pe poziția următoare este generat testând valorile posibile în ordine crescătoare.

Numărul de soluții este n^m. Adică exponențial.

Cu lungimea m și valori de la 1 la n pentru fiecare element avem mai sus toate soluțiile posibile. În general problemele de backtracking mai au un element în plus: dintre toate soluțiile posibile se cer doar cele care îndeplinesc în plus anumite restricții. Mai departe să considerăm că dorim doar soluțiile cu elemente distincte.

Reformulăm așadar problema pe care ne vom concentra în continuare: să se genereze toate șirurile de lungime m, cu valori distincte cuprinse între 1 și n. Aceasta este de fapt echivalentă cu generarea aranjamentelor de câte m elemente ale mulțimii 1..n, deci șiruri în care contează și ordinea elementelor (Pentru n=5 și m=3 avem: 1 2 3, 1 2 4, 1 2 5, 1 3 2, 1 3 4, 1 3 5, 1 4 2, 1 4 3, 1 4 5, 1 5 2, 1 5 3, 1 5 4, 2 1 3, 2 1 4 ... 5 4 3).



În desenul de desfășurare a autoapelurilor pentru n=3 și m=3 prezentat ceva mai sus, dispar ramurile eliminate în cel aflate imediat mai sus.

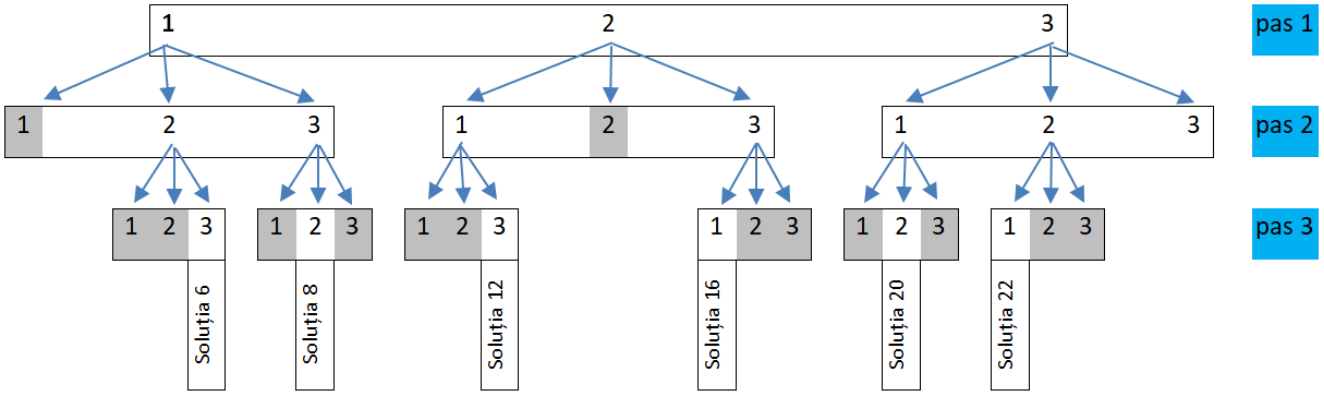
Pentru a obține acest lucru, la codul prezentat inițial pentru generarea produsului cartezian putem adăuga un test suplimentar chiar înainte de afișarea soluției: tipărim doar dacă elementele sunt distincte:

```
#include <iostream>
using namespace std;
int n, m;
int x[30];
void backtrack(int pas) {
    if (pas == m+1) {
        dist = 1;
        for (int i=1;i<m;i++)
            for (int j=i+1;j<=m;j++)
                if (x[i] == x[j])
                    dist = 0;
        if (dist == 1) {
            for (int i=1;i<=m;i++)
                cout<<x[i]<<" ";
            cout<<"\n";
        }
    } else
        for (int i=1;i<=n;i++) {
            x[pas] = i;
            backtrack(pas+1);
        }
}
```

```
int main () {
    cin>>n>>m;
    backtrack(1);
}
```

Am scris bold codul care face testul. Această posibilitate de a obține ce ne interesează generează totuși toate soluțiile produsului cartezian.

Am văzut că algoritmii care fac asta sunt de ordin exponențial și este foarte important să aducem orice optimizare pentru a câștiga cât mai mult timp la executare. Aici, de exemplu, observăm că dacă punem 1 pe prima poziție și pe a doua tot 1, încă de acum ne dăm seama că nu are sens să mai generăm valorile pe restul de poziții. Gândiți-vă că mai sus avem un exemplu pentru m=3, dar arborele putea fi mult mai înalt și am fi redus extrem de multe ramuri ale sale dacă ne-am fi oprit când apare primul duplicat în soluție, lucru echivalent cu a testa la fiecare moment dacă valoarea curentă este diferită de cele aflate deja în față. Cu acest test, arborele de autoapeluri ar arăta:



Așadar ideea de bază care se urmărește în această secțiune este de a impune un test suplimentar asupra elementului tocmai încercat în vârful stivei (adică în `x[pas]`) și dacă acest element nu trece testul, alături de cele generate deja în prefixul curent (`x[1]`, ..., `x[pas-1]`), atunci nu se mai face autoapel și se face altă alegere lapoziția curentă, pas.

Astfel, în scopul obținerii aranjamentelor, algoritmul de generare a produsului cartezian al `m` mulțimi, fiecare cu mulțimea de valori `1, 2, ... n,` se modifică astfel:

```
#include <iostream>
using namespace std;
int n, m;
int x[30];

int verif (int pas) {
    for (int i=1;i<pas;i++)
        if (x[pas]==x[i])
            return 0;
    return 1;
}

void backtrack(int pas) {
    if (pas == m+1) {
        for (int i=1;i<=m;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
```

```
            x[pas] = i;
            if (verif(pas) == 1)
                backtrack(pas+1);
        }
}
int main () {
    cin>>n>>m;
    backtrack(1);
}
```

Acesta, de mai sus, este algoritmul care generează aranjamentele, fără a se genera produsul cartezian, adică oprindu-ne dacă deja constatăm că prefixul curent nu poate fi al unei soluții.

Este esențial de remarcat ce trebuie gândit când scriem funcția verif: considerăm validate tot cu ea elementele plasate deja pe pozițiile `1`, `2`, … `pas-1` (căci s-a ajuns la pas doar trecându-se de verif când a fost rândul lor să fie în vârf) și doar pentru cel curent, `x[pas]`, testăm dacă trece condițiile alături de celelalte, din prefix. Adică noi am testat doar dacă `x[pas]` ar fi egal cu vreun element din fața sa.

Este acum momentul să dăm o generalizare pentru algoritmul de backtracking, adică un fel de template pe care să îl folosim la toate problemele, identificând punctele cheie, unde lucrurile pot să difere.

Pentru început se stabilesc trei lucruri (și le vom exemplifica pe algoritmul de generare de aranjamente, tocmai prezentat):

I. Care este **lungimea soluției** (în cazul de față `m`).
II. Ce **valori posibile** poate lua elementul din vârf, `x[pas]` (aici, de la `1` la `n`).
III. Ce **condiții de continuare** punem elementului tocmai încercat în vârf pentru ca el, alături de cele puse deja înainte, și validate la fel, să poată duce la soluție: conținutul funcției verif (aici, `x[pas]` să fie diferit de `x[i]`, cu `i` de la `1` la `pas-1`).

Am putea gândi astfel:

```
backtrack(pas) {
    dacă pas este mai mare cu 1 decât lungimea stabilită pentru soluție la (I)
        tipărește soluția (elementele din x de pe poziții de la 1 la
        valoarea stabilită la (I)).
    altfel {
        pentru fiecare valoare a lui x[pas]  stabilită la (II)
            dacă ea îndeplinește condițiile impuse la (III)
                backtrack(pas+1)
    }
}
```

Pentru condițiile de continuare, stabilite la III se recomandă scrierea unei funcții întrucât pot fi multe și așa le-am putea grupa separat. La scrierea acesteia am putea gândi:

```
int verif(int pas) {
    pentru fiecare motiv posibil să invalideze valoarea încercară la poziția pas
        return 0;
    return 1;
}
```

Altfel spus, dacă se scapă de tot ce poate invalida valoarea încercată, returnăm la final `1`.

Probleme rezolvate

1. Să se genereze toate permutările mulțimii 1, 2, ..., n. De exemplu, pentru n=4 s-ar afișa: 1 2 3 4, 1 2 4 3, 1 3 2 4, 1 3 4 2, 1 4 2 3, 1 4 3 2, 2 1 3 4, 2 1 4 3, ... 4 3 1 2, 4 3 2 1.

Rezolvare

I – soluția are lungimea n.

II – valorile posibile pentru x[i] sunt cuprinse între 1 și n.

III – condiții de continuare: x[pas] diferit de elementele din fața sa.

Așadar problema are, față de cea a generării aranjamentelor, o singură modificare: la afișare, m=n. De fapt aici avem un singur parametru, n.

```
#include <iostream>
using namespace std;
int n, m;
int x[30];
int verif (int pas) {
    for (int i=1;i<pas;i++)
        if (x[pas]==x[i])
            return 0;
    return 1;
}
void backtrack(int pas) {
    if (pas == n+1) {
        for (int i=1;i<=n;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            x[pas] = i;
            if (verif(pas) == 1)
                backtrack(pas+1);
        }
}
int main () {
    cin>>n>>m;
    backtrack(1);
}
```

2. Numim punct fix într-o permutare o poziție i unde x[i] = i. Dându-se n, să se genereze toate permutările de n care nu au puncte fixe. Pentru n=3, dintre cele 6 permutări posibile avem de afișat doar pe 2 3 1 și 3 1 2.

Rezolvare

Observăm așadar că trebuie în primul rând restrucția de la permutări (soluție de lungime n, valori posibile de la 1 la n, elemente distincte). Dintre toate permutările mai trebuie reduse însă. Și atunci cum facem asta? Unde apar noile restricții? Răspunsul: la condițiile de continuare. Dacă la poziția pas avem x[pas] == pas, returnăm 0. Doar asta se schimbă. Din acest motiv mai jos vom da numai funcția verif.

```
int verif (int pas) {
    if (x[pas] == pas)
        return 0;
```

```
    for (int i=1;i<pas;i++)
        if (x[pas]==x[i])
            return 0;
    return 1;
}
```

Să analizăm puțin. Unii oameni adaugă în `for`, la condiția de la `if`, `&& x[i] == i`. Noi observăm că nu este necesar să mai facem acest test la elementele din urmă întrucât, dacă testăm mereu doar ca elementul din vârf să nu fie "punct fix", avem mereu validate elementele dintre pozițiile `1` și `pas-1`.

Deducem că putem împărți condițiile de continuare în două:

- Unele care validează elementul curent cu toate din urmă (și aici am făcut asta pentru a verifica dacă elementele sunt distincte)
- Unele care fac validare a elementului curent doar local (fără for în urmă ci doar la poziția sa sau, vom vedea în continuare, cu unul, două elemente anterioare).

Observați și cum am respectat modul de gândire al lui verif prezentat anterior: am testat cele două situații care af face să eșueze verificarea și la final returnăm `1`, dacă ele trec.

Rămânem la această problemă să prezentăm și una dintre optimizările foarte importante care se fac deseori la problemele de backtracking. Observăm mai sus că funcția `verif` adaugă un factor `n` la complexitate (la fiecare element nou încercat e necesar să parcurgem înapoi pentru a testa dacă valorile rămân distincte).

În cazul problemelor ce testează elemente distincte și, vom vedea ulterior, și în alte vazuri, putem folosi următoarea tehnică: ținem un vector de frecvență, notat aici f, global în care avem marcate elementele folosite deja în prefixul validat din soluție. Acum, la încercarea unui nou element `x[pas]`, putem testa daca valoarea mai apare în față printr-un singur `if`, verificând dacă `x[pas]` este `1`.

Dacă trecem testul. deci lăsăm în soluție valoarea tocmai validată, înainte de a face autoapel, trebuie să o marcăm și pe ea în `f`. Însă, lucru extrem de important, după autoapel, simetric cu locul marcării, trebuie să redeclarăm disponibilă valoarea. Semnificația acestui lucru este că după autoapel forul continuă la pasul curent, se face acolo altă încercare și trebuie ca valoarea folosită anterior să fie lăsată liberă pentru a putea apoi fi utilizată la nivelele superioare. Soluția problemei este acum:

```
#include <iostream>
using namespace std;
int n, m;
int x[30], f[30];
int verif (int pas) {
    if (x[pas] == pas)
        return 0;
    return 1;
}
void backtrack(int pas) {
    if (pas == n+1) {
        for (int i=1;i<=n;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            if (f[i] == 0) {
                x[pas] = i;
                f[i] = 1;
                if (verif(pas) == 1)
                    backtrack(pas+1);
                f[i] = 0; ///!!!!
```

```
            }
        }
}
int main () {
    cin>>n>>m;
    backtrack(1);
}
```

Din raţiuni didactice, aici am păstrat funcţi `verif`. Noi cu vectorul de frecvenţă am rezolvat doar testarea ca elemenele să fie distincte. Dacă însă se impun şi alte condiţii soluţiilor, funcţia verif îşi are în continuare rostul ei. Evident că dacă nu se impun alte condiţii, funcţia verif poate lipsi. De exemplu, la generarea permutărilor, funcţia recursivă arată aşa:

```
void backtrack(int pas) {
    if (pas == n+1) {
        for (int i=1;i<=n;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            if (f[i] == 0) {
                x[pas] = i;
                f[i] = 1;
                backtrack(pas+1);
                f[i] = 0; ///!!!!
            }
        }
}
```

3. Problema turelor. Se cede afişarea tuturor modurilor de aplasa `n` ture, pe o tablă de şah `nXn` aşa încât oricare două să nu se atace. Cunoaştem că pe o tablă de şah două ture se atacă dacă sunt plasate pe aceeaşi linie sau pe aceeaşi coloană. De exemplu, pentru `n=3` avem soluţiile:

T		
	T	
		T

T		
		T
	T	

	T	
T		
		T

	T	
		T
T		

		T
T		
	T	

		T
	T	
T		

Rezolvare

La prima vedere problema este total atipică faţă de ce am studiat deja. Noi ştim să generăm soluţii sub formă de vector şi aici avem de-a face cu ceva în două dimensiuni. Facem însă următoarea observaţie: pe o linie nu pot fi două ture, deci pentru o linie am putea să memorăm coloana turei care se află pe acea linie. Ajungem aşadar la un vector acum în care `x[i]` reprezintă coloana turei care se află pe linia `i`. Adică în vectorul soluţie de la backtracking, indicii ar reprezenta linii de pe tabla de şah iar valorile ar reprezenta coloane. Ne-am asigurat până acum de linii distincte întrucât o componentă a unui vector poate avea o singură valoare şi cum aceasta reprezintă coloana corespunzătore liniei reprezentată de componentă, asigurăm unicitatea turelor pe linii. Acum ne mai rămâne doar să punem condiţia să nu avem două ture pe aceeaşi coloană. Se vede repede că asta este echivalent să impunem ca elementele din vector să fie distincte (valorile înseamnă indici de coloane). Aşadar problema noastră este absolut identică cu aceea de generare a permutărilor. Ne putem alege însă modul de afişase şi iată de putem obţine:

```
for (i=1;i<=n;i++)
    cout<<'(',i,',',x[i],')';
cout<<"\n";
```

```
(1,1)(2,2)(3,3)
(1,1)(2,3)(3,2)
(1,2)(2,1)(3,3)
```

	(1,2)(2,3)(3,1) (1,3)(2,1)(3,2) (1,3)(2,2)(3,1) Am afișat mai sus perechile de indici ocupați de turele de la fiecare soluție.

for (i=1;i<=n;i++) {
  for (j=1;j<=n;j++)
  if (x[i] == j)
  cout<<"T";
  else
  cout<<"*";
  cout<<"\n";
}
cout<<"\n";
	T**
	T
	**T
	
	T**
	**T
	T
	...
	Aici avem de-a face cu un mod de afișare mai prietenos.

4. Problema damelor. Se cere afișarea tuturor modurilor de aplasa `n` dame, pe o tablă de șah `nXn` așa încât oricare două să nu se atace. Cunoaștem că pe o tablă de șah două dame se atacă dacă sunt plasate pe aceeași linie, pe aceeași coloană sau pe aceeași diagonală. De exemplu, pentru `n=3` nu avem soluții, dar pentru `n=4` le avem pe următoarele două:

	D		
			D
D			
		D	

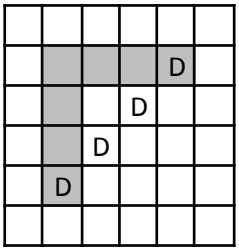
		D	
D			
			D
	D		

Rezolvare

Această problemă este foarte asămănătoare cu aceea a turelor, mulțimea soluțiilor sale fiind inclusă în mulțimea soluțiilor de la ture. Condiția suplimentară, aceea ca oricare două dame să nu se atace pe diagonală este:

$$|pas - i| \neq |x[pas] - x[i]|$$

Dacă două dame s-ar afla pe aceeași "diagonală" ar însemna că se formează un "triunghi dreptunghic isoscel" ca în figură.



Lungimile laturilor sale sunt date de diferența liniilor și a coloanelor pentru punctele pe care le ocupă. Noi am folosit aveste valori în modul pentru a prinde toate cazurile care pot apărea pentru dispunerea celor două puncte.

Vom păstra testul cu vectori de frecvență pentru elemente distincte și vom pune în verif testul pentru diagonale. Facem însă observația că este posibilă optimizarea cu vectori de frecvență și pentru testul de diagonale, dacă ținem cont că elementele de pe aceeași paralelă la diagonala principală a unei matrice pătratice au diferența indicilor constantă iar cele de pe aceeași paralelă la diagonala secundară au suma indicilor constantă. Lăsăm această optimizare ca exercițiu.

Iată un program complet care generează toate soluțiile de la problema damelor, afișate “prietenos”.

```
#include <iostream>
using namespace std;
int n;
int x[30], f[30];
int modul(int x) {
    return x > 0 ? x : -x;
}
int verif (int pas) {
    /// la pas-i nu am mai pus modul intrucat noi testam
    /// in urma si atunci sigur avem pas>i
    for (int i=1;i<pas;i++)
        if (pas-i == modul(x[pas]-x[i]))
            return 0;
    return 1;
}
void backtrack(int pas) {
    if (pas == n+1) {
        for (int i=1;i<=n;i++) {
            for (int j=1;j<=n;j++)
                if (x[i] == j)
                    cout<<"D";
                else
                    cout<<"*";
            cout<<"\n";
        }
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            if (f[i] == 0) {
                x[pas] = i;
                f[i] = 1;
                if (verif(pas))
                    backtrack(pas+1);
                f[i] = 0;
            }
        }
}
int main () {
    cin>>n;
    backtrack(1);
}
```

5. **Generarea combinărilor.** Revenim asupra problemei, despre care am mai discutat. Avem, în fond, de generat soluții de lungime `m` în care valorile sunt distincte, cuprinse între `1` și `n` și nu se consideră soluții distincte permutări ale aceluiași set de elemente. Problema este echivalentă cu a genera soluții cu elementele în ordine strict crescătoare. De exemplu, pentru `n=5` și `m=3` avem: `1 2 3, 1 2 4, 1 2 5, 1 3 4, 1 3 5, 1 4 5, 2 3 4, 2 3 5, 2 4 5, 3 4 5.`

Rezolvare

Observăm că față de problema generării aranjamentelor se impune condiția ca elementele să fie în ordine crescătoare. Acest lucru îl putem face în `verif`, impunând pentru `x[pas]` să fie strict mai mare ca `x[pas-1]`. În acest fel nu mai este necesar forul înapoi pentru a testa ca elementele să fie disticte (dacă ne asigurăm că fiecare e mai mare ca precedentul, atunci automat sunt și distincte).

```
#include <iostream>
using namespace std;
int n, m;
int x[30], f[30];
int verif (int pas) {
    if (pas > 1 && x[pas] <= x[pas-1])
        return 0;
    return 1;
}
void backtrack(int pas) {
    if (pas == m+1) {
        for (int i=1;i<=m;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            x[pas] = i;
            if (verif(pas))
                backtrack(pas+1);

        }
}
int main () {
    cin>>n>>m;
    backtrack(1);
}
```

Atenție la condiția din funcția de verificare, noi ne asigurăm că pas este cel puțin `2` pentru a exista și elementul din față, cu care comparăm.

Totuși, modalitatea de mai sus nu este cea mai bună de generare a combinărilor, chiar dacă ea reduce mult calculele drept consecință a condiției puse în `verif`. Noi observăm că odată un prefix generat bine (crescător), următorul element nu are sens să îl mai testăm de la `1` ci de la valoarea cu `1` mai mare decât ultima pusă deja. Acest lucru este suficient și îl putem controla direct din forul de pe ramura cu autoapelul, unde stabilim valorile posibile la pasul curent. Avem:

1	`for (int i=x[pas-1] + 1; i<=n; i++) {`
	`  x[pas] = i;`
	`  ...`

2	Pentru a ne asigura că forul de mai sus funcționează bine și când `pas=1`, trebuie să avem `x[0]=0`.
3	Nu ne mai rămân alte condiții de pus, noi generând valorile direct mai mari ca la pasul anterior și atunci putem renunța definitiv la funcția `verif`.

Iată deci algoritmul backtracking complet de generare a combinărilor care devine acum foarte compact și rapid.

```
#include <iostream>
using namespace std;
int n, m;
int x[30];
```

```
void backtrack(int pas) {
    if (pas == m+1) {
        for (int i=1;i<=m;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=x[pas-1]+1;i<=n;i++) {
            x[pas] = i;
            backtrack(pas+1);
        }
}
int main () {
    cin>>n>>m;
    x[0] = 0;
    backtrack(1);
}
```

6. **Generarea partițiilor unui număr.** Se pune problema să scriem în toate modurile un număr ca sumă de numere pozitive mai mici decât el. Se poate cere rezultatul în mai multe moduri:

a) Repetându-se termenii sumei și putându-se permuta. De exemplu, pentru n=5 primele soluții sunt: `1 1 1 1 1, 1 1 1 2, 1 1 2 1, 1 1 3, 1 2 1 1, 1 2 2, 1 3 1, 1 4, 2 1 1,` etc.

b) Repetându-se termenii sumei, dar fără a-i permuta. Adică dintre `1 1 1 2` și `1 1 2 1` și `1 2 1 1` și `2 1 1 1` afișăm doar o soluție. De pe acum observăm că asta este echivalent cu a genera soluțiile în ordine crescătoare, dar de data asta nu și strictă. La `n=6` am avea soluțiile: `1 1 1 1 1 1, 1 1 1 1 2, 1 1 1 3, 1 1 2 2, 1 1 4, 1 2 3, 1 5, 2 2 2, 2 4, 3 3, 6`.

c) Fără ca termenii să se repete și fără a-i putea permita. Pentru `n=6` avem soluțiile: `1 2 3, 1 5, 2 4, 6`.

Rezolvare

Ne concentrăm în primul rând asupra cerinței `a)`. Analizăm cele trei lucruri de stabilit la început:

I. Lungimea soluției nu mai este acum fixă.

II. Valorile posibile de la fiecare nivel din soluție sunt între `1` și `n`. Nu putem avea termeni mai mari decât valoarea de partiționat.

III. Condiții de continuare: trebuie ca suma prefixului deja generat să nu depășească valoarea `n` nicodată, iar dacă ea este chiar egală cu n vom tipări, fără să mai facem autoapel.

Vom scrie o funcție care calculează suma elementelor din `x` de pe poziții de la `1` la o valoare dată. O vom apela de două ori, pe de o parte în verif, ca să decidem dacă am sărit sau nu cu suma de `n` și apoi înainte de afișare pentru a testa dacă suma este chiar `n`. Acest test este cel prin care stabilim când ajungem la soluție pentru că aici nu ne mai putem folosi de comparare de pas cu lungime fixă.

Am mai putea modifica algoritmul și să afișăm după apelul funcției de sumă din `verif` dacă rezultatul este chiar `n`, dar noi suntem la început cu backtracking și nu dorim deocamdată să ne abatem mult de la standard, pentru a nu încurca lucrurile. Tot pentru standard am păstrat încă funcția `verif` dar am fi putut ca în locul ei să testăm direct dacă `suma(pas) <= n`.

```
#include <iostream>
using namespace std;
int n, m;
int x[30];
int suma(int pas) {
```

```
    int s = 0;
    for (int i=1;i<=pas;i++)
        s+=x[i];
    return s;
}
int verif(int pas) {
    if (suma(pas) <= n)
        return 1;
    return 0;
}
void backtrack(int pas) {
    if (suma(pas-1) == n) {
        for (int i=1;i<=pas-1;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++) {
            x[pas] = i;
            if (verif(pas))
                backtrack(pas+1);
        }
}
int main () {
    cin>>n;
    backtrack(1);
}
```

Modul de calcul al sumei prefixului, realizat la fiecare pas ca mai sus, a fost prezentat doar în scop didactic. De fapt noi putem face calculul sumei pe parcurs, într-o variabilă globală, pe care o mărim cu valoarea încercată în vârf înainte de autoapel și o micșorăm după cu aceeași valoare autoapel. Folosim același principiu ca la vectorul de frecvență global folosit la testul de elemente distincte. Secvența cheie este următoarea:

```
if (s + i <= n) {
    x[pas] = i;
    s += i;
    backtrack(pas+1);
    s -= i;
}
```

Observăm că la testul de soluție putem folosi direct variabila globală `s`. Această soluție reduce complexitatea cu un factor `n`.

Soluția completă este:

```
#include <iostream>
using namespace std;
int n, s;
int x[30];
void backtrack(int pas) {
    if (s == n) {
        for (int i=1;i<=pas-1;i++)
            cout<<x[i]<<" ";
        cout<<"\n";
    } else
        for (int i=1;i<=n;i++)
            if (s + i <= n){
```

```
                x[pas] = i;
                s += i;
                backtrack(pas+1);
                s -=i;
            }
}
int main () {
    cin>>n;
    backtrack(1);
}
```

Soluțiile `b)` și `c)` se obțin combinând ce este mai sus cu ideea de la combinări.

Astfel, pentru `b)`, forul de pe ramura cu autoapelul unde stabilim valorile de încercat la nivelul curent îl scriem:

```
for (int i=x[pas-1];i<=n;i++)
            if (s + i <= n){
```

În plus, înainte de apelul din main inițializăm `x[0] = 1`, pentru a putea construi corect valoarea de la pasul `1`. În felul acesta la pasul curent permitem valori mai mari decât la pasul anterior dar și egale.

Pentru c) pornim cu i de la `x[pas-1]+1` iar `x[0]` trebuie să fie la început `0`. Așa permitem, cum cere și enunțul, la un nivel, valori strict mai mari decât la cel anterior.

7. **Parantezări.** Dându-se un număr `n`, par și cel mult egal cu `20`, se cere afișarea tuturor șirurilor de `n` paranteze care sunt asociate corect (pbinfo.ro, `#344`). De exemplu, pentru `n=6` avem soluțiile:

```
((()))
(()())
(())()
()(())
()()()
```

Rezolvare

Observăm că vectorul soluție are lungimea `n` și la fiecare poziție sunt două valori posibile. Deci spațiul soluțiilor este inclus în cel al tuturor șirurilor de lungime `n` formate cu `2` valori distincte (să considerăm `0` pentru paranteză deschisă și `1` pentru paranteză închisă). Adică până acum am stabilit lungimea soluției ca fiind `n` și valorile posibile ca fiind `0` și `1`. Fără să punem condiții de continuare noi ajungem la o problemă pe care am mai amintit-o și am și rezolvat-o obținând toate submulțimile unei mulțimi. Acum să vedem care sunt condițiile de continuare speciifice acestei probleme.

- În primul rând trebuie ca la fiecare pas intermediar numărul de paranteze deschise să fie mai mare sau cel mult egal decât cel al parantezelor închise.
- La final cele două numere trebuie să fie egale, ambele cu `n/2`

În cod noi punem în `x` valori `0` și `1` iar în momentul tipăririi scriem pe ecran `'('` în loc de `0` și `')'` în loc de `1`.

Numărarea parantezelor deschise din prefixul curent o facem cu o variabilă globală pe care o incrementăm de câte ori punem paranteză deschisă și o decrementăm, simetric, la revenirea din apelurile la care am pus paranteză deschisă. Nu mai numărăm separat parantezele închise întrucât sunt restul, deci numărul lor se obține ca pas-numărul celor deschise.

Se mai observă în codul de mai jos că noi nu punem la final condiția ca să fie egale cele două numere de paranteze. Însă, pe parcurs, pe lângă testul ca acelea deschise să fie mereu cel puțin egale cu cele închise, mai punem și condiția ce acelea deschise să nu depășească `n/2`. Se deduce ușor, prin reducere la absurd, că dacă sunt mereu îndeplinite cele două condiții impuse mai sus, la final cele două valori nu pot fi altfel decât egale.

```
#include<fstream>
using namespace std;
int n,desc;
```

```
int x[21];
ifstream fin ("paranteze.in");
ofstream fout("paranteze.out");
void back(int pas){
    if(pas==n+1){
        for(int i=1;i<=n;i++){
            if (x[i] == 0)
                fout<<"(";
            else
                fout<<")";
        }
        fout<<"\n";
    }
    else{
        for(int i=0;i<=1;i++){
            x[pas] = i;
            if (i == 0)
                desc++;
            if (desc >= pas-desc && desc<=n/2)
                back(pas+1);
            if (i == 0)
                desc--;
        }
    }
}
int main(){
    fin>>n;
    back(1);
    return 0;
}
```

8. Fie n un număr natural. Să se determine toate posibilitățile de alegere a semnelor + și − pentru care $n = (+|-)\ 1^2 + (+|-)\ 2^2 + \ldots + (+|-)\ n^2$. De exemplu, pentru n=9 avem soluțiile:

```
- - + - + - + + -
+ - - + - - + - +
+ + - - + + - - +
+ + + + - + - - +
```

Dacă nu sunt soluții se va tipări IMPOSIBIL. (pbinfo.ro, 1357).

Rezolvare

În vectorul soluție vom memora semnul din fața fiecărui termen de la 1 la n. Codificăm minus cu 0 și plus cu 1. Așadar avem tot o probemă de generare de șiruri de lungime n cu 0 și 1. Folosim și aici o variabilă globală care ține valoarea expresiei formate cu prefixul curent.

```
#include <fstream>
#include <algorithm>
using namespace std;
int x[25], n,calcul,sol;
ifstream fin("plusminus.in");
ofstream fout("plusminus.out");
void back(int pas) {
    if (pas > n) {
        if (calcul==n){
            sol=1;
```

```
            for (int i=1;i<=n;i++)
                if (x[i]==0)
                    fout<<"- ";
                else
                    fout<<"+ ";
            fout<<"\n";
        }
        return;
    }
    for (int i=0;i<=1;i++) {
        x[pas] = i;
            if (i == 1)
                calcul += pas*pas;
            else
                calcul -= pas*pas;
            back(pas+1);
            if (i == 1)
                calcul -= pas*pas;
            else
                calcul += pas*pas;
    }
}
int main () {
    fin>>n;
    back(1);
    if (sol==0)
        fout<<"IMPOSIBIL";
    return 0;
}
```