**Subalgoritm back (k)**  
pentru fiecare valoare i din multimea Sk execută  
xk←i  
dacă X respectă condiţiile interne atunci  
dacă X este solutie atunci   
afisează X  
altfel  
apelează back(k+1)  
sfdacă  
sfdacă  
sfpentru

În funcţie de problema concretă, în algoritmul descris mai sus se vor modifica doar instrucţiunea *pentru*, condiţiile interne şi cele de soluţie, structura algoritmului păstrându-se.

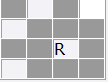
**Probleme de generare. Oportunitatea utilizării metodei backtracking**

Problemele care se rezolvă prin metoda backtracking pot fi împărţite în mai multe grupuri de probleme cu rezolvări asemănătoare, in funcţie de modificările pe care le vom face în algoritm. Principalele grupuri de probleme sunt:  
a) probleme în care vectorul soluţie are lungime fixă şi fiecare element apare o singură dată în soluţie;  
b) probleme în care vectorul soluţie are lungime variabilă şi fiecare element poate să apară de mai multe ori în soluţie;  
c) probleme în plan, atunci când spaţiul în care ne deplasăm este un tablou bidimensional.  
Vom prezenta în cele ce urmează câteva probleme care pac parte din primul grup. Cele mai cunoscute sunt:

* generarea permutărilor unei mulţimi
* generarea aranjamentelor unei mulţimi
* generarea submulţimilor unei mulţimi
* generarea submulţimilor cu m elemente ale unei mulţimi (combinări)
* generarea produsului cartezian a n mulţimi
* generarea tuturor secvenţelor de n (par) paranteze care se închid corect.
* colorarea ţărilor de pe o hartă astfel încât oricare două ţări vecine să aibă culori diferite
* aranjarea a n regine pe o tablă de şah de dimensiune n fără ca ele să se atace.

Toate problemele din acest grup au particularitatea că soluţia se obţine atunci când vectorul soluţie ajunge să conţină un anumit număr de elemente.  
Vom exemplifica modul de lucru al metodei backtracking pentru problema damelor.

***Aranjarea reginelor***. Dându-se o tablă de şah de dimensiune nxn (n>3) să se aranjeze pe ea n regine fără ca ele să se atace. Reamintim că o regină atacă linia, coloana şi cele 2 diagonale pe care se află. În figura de mai jos celulele colorare mai închis sunt atacate de regina poziţionată unde indică litera “R”.



În algoritmul de mai sus avem de particularizat următoarele:  
Instrucţiunea pentru fiecare valoare i din mulţimea Sk execută va fi înlocuită cu o instrucţiune ***pentru*** care parcurge toate valorile de la 1 până la n.  
Condiţia de a putea plasa o regină pe poziţia k este un pic mai complicată şi presupune verificarea ca să nu se atace cu nici una dintre celelalte k-1 regine deja plasate pe tabla. Dacă pe poziţia k din vectorul X punem o valoare ea va reprezenta coloana pe care se plasează pe tablă regina k. Condiţiile devin astfel:   
x[i]¹x[k] şi |k-i|¹|x[k]-x[i]| cu i de la 1 la k-1 şi |x| reprezentând modului lui x.  
Condiţia de soluţie este simplă şi presupune plasarea corectă a tuturor celor n regine.  
Programele Pascal si C++ rezultate prin implementarea algoritmului descris mai sus următoarele:

|  |  |
| --- | --- |
| **Varianta Pascal** | **Varianta C/C++** |
| var x:array[1..100] of byte; n:byte; nrsol:word;  procedure scriesol; var i,j:byte; begin inc(nrsol); writeln('Solutia a',nrsol,'este'); for i:=1 to n do begin writeln; for j:=1 to n do if x[j]=i then write('X',' ') else write('O',' '); end; end;  function potcont(k:byte):boolean; var i:byte; atac:boolean; begin atac:=false; for i:=1 to k-1 do if(x[i]=x[k]) or (k-i=abs(x[k]-x[i])) then atac:=true; potcont:=not atac; end;  procedure back(k:byte); var i:byte; begin for i:=1 to n do  begin x[k]:=i; if potcont(k) then if k=n then scriesol else back(k+1);  end; end;  begin read(n); nrsol:=0; back(1); writeln(nrsol,'solutii'); end. | #include<iostream.h> #include<math.h>  int x[100],n,nrsol;  void scriesol () { int i,j; nrsol++; cout<<"Solutia a "<<nrsol<<" este"; for(i=1;i<=n;i++) { cout<<endl; for(j=1;j<=n;j++) if (x[j]==i) cout<<"X "; else cout<<"O "; } }  int potcont(int k) { int i; for(i=1;i<=k-1;i++) if (x[i]==x[k] || k-i==abs(x[k]-x[i])) return 0; return 1; }  void back(int k) { int i; for(i=1;i<=n;i++) { x[k]=i; if (potcont(k)) if (k==n) scriesol(); else back(k+1); } }  void main() { cin>>n; nrsol=0; back(1); cout<<nrsol<<" solutii"; } |

Din al doilea grup fac parte probleme a căror condiţie de soluţie nu se mai pune asupra numărului de elemente din vectorul X, ci asupra elementelor din soluţie. Exemple:

* partiţiile unui număr natural n
* partiţiile unei mulţimi
* plata unei sumei cu monede de valori date

***Partiţiile unui număr natural***. Fie n>0, natural. Să se scrie un program care să afişeze toate partiţiile unui număr natural n.  
Numim partiţie a unui număr natural nenul n o mulţime de numere naturale nenule {p1, p2, …, pk} care îndeplinesc condiţia p1+p2+ …+pk = n.  
Ex: pt n = 4 programul va afişa:  
4 = 1+1+1+1  
4 = 1+1+2  
4 = 1+3  
4 = 2+2  
4 = 4  
Observaţii: - lungimea vectorului soluţie cel mult n;

* există posibilitatea ca soluţiile să se repete;
* condiţia de final este îndeplinită atunci când suma elementelor vectorului soluţie este n.

Am menţionat mai sus că vom folosi doi parametri, unul pentru poziţia în vectorul soluţie şi un al doilea în care avem sumele parţiale la fiecare moment. Avem determinată o soluţie atunci când valoarea celui de-al doilea parametru este egală cu n.  
În această situaţie la fiecare plasare a unei valori în vectorul sol valoarea celui de al doilea parametru se măreşte cu elementul ce se plasează în vectorul soluţie. Apelul procedurii back din programul principal va fi back(1, 0).  
Există şi posibilitatea de a apela procedura back din programul principal back(1, n) şi valoarea celui de al doilea parametru se decrementează cu elementul ce se plasează în vectorul sol, iar o soluţie avem când acest parametru este zero. Indiferent care modalitate este aleasă acest al doilea parametru ne permite să optimizăm puţin programul în sensul că putem considera nişte condiţii de continuare mai strânse.

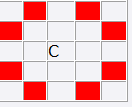
|  |  |
| --- | --- |
| **Varianta Pascal** | **Varianta C/C++** |
| program Partitii\_nr\_natural; var n, ns: byte; sol: array[1..20] of byte; procedure afis(l: byte); var i: byte; begin inc(ns); write 'Solutia ', ns, ' : '); for i:=1 to l do write(sol[i]:3); writeln; end;  procedure back(i, sp: byte); var j: byte; begin if sp = n then afis(i-1) else for j:=1 to n-sp do if (j>=sol[i-1]) then begin sol[i]:=j; back(i+1, sp+j) end; end;  begin read(n); ns:=0; back(1,0); writeln(ns,'solutii');  end. | #include<iostream.h> int n, ns,sol[20];  void afis(int l) { int i; ns++; cout<<"Solutia nr. "<< ns<<" : "; for(i=1;i<=l;i++) cout<<sol[i]<<" "; cout<<endl; }  void back(int i, int sp) { int j; if (sp==n) afis(i-1); else for(j=1;j<=n-sp;j++) if (j>=sol[i-1]) { sol[i]=j; back(i+1, sp+j); } }  void main() { cin>>n; ns=0; back(1,0); cout<<ns<<" solutii"; } |

Problemele în plan necesită parcurgerea unui tablou unidimensional, iar cele mai cunoscute sunt:

* parcurgerea tablei de şah cu un cal, fără a trece de două ori prin aceeaşi poziţie
* găsirea ieşirii dintr-un labirint

Problemele care se rezolvă prin metoda backtracking în plan au ca cerinţă deplasarea în tablou, pe linii, coloane sau diagonale sau prin săritură (de obicei săritura calului) dintr-un punct în alt punct al tabloului sau pe frontieră (prima linie sau coloană, ultima linie sau coloană) eventual respectând anumite condiţii de deplasare. Dacă ne găsim într-un anumit punct iar cerinţa este de a ne deplasa în unul din cei opt vecini ai lui vom utiliza pentru acest lucru două cicluri for de la –1 la 1 cu care valori vom modifica coordonata punctului curent. Dacă deplasarea este permisă numai pe linii condiţia de respectat este ca suma în valoare absolută pentru cei doi indici să fie 1, iar pentru deplasarea pe diagonală 2. De asemenea se mai impune condiţia de a nu părăsi tabloul, lucru care îl vom respecta testând coordonatele noului punct să aparţină mulţimii [1..nrlinii] şi [1..nrcol].

***Săritura calului***. Fiind dată o tablă de şah de dimensiunea nxn şi un cal în colţul stânga sus al acesteia, se cere să se afişeze toate posibilităţile de mutare a acestei piese de şah astfel încât să treacă o singură dată prin fiecare pătrat al tablei. O soluţie va fi afişată ca o matrice nxn în care sunt numerotate săriturile calului.  
Exemplu, pentru n=5, o soluţie este  
1 14 9 20 23  
10 19 22 15 8  
5 2 13 24 21  
18 11 4 7 16  
3 6 17 12 25  
Pentru rezolvarea acestei probleme vom codifica direcţiile de deplasare pentru că ar fi ineficient să scriem două cicluri for de la –2 la 2 cu cele 25 de variante de deplasare din care valide sunt doar opt. De asemenea aici spre deosebire de celelalte probleme tratate la aplicarea metodei backtracking în plan nu vom folosi un vector soluţie, ci vom scrie săriturile în tablou urmărind ca la revenire să refacem poziţiile ocupate pentru a nu se lua blocaje. În figura de mai jos sunt prezentate cele 8 mutări posibile pentru un cal.



|  |  |
| --- | --- |
| **Varianta Pascal** | **Varianta C/C++** |
| const dx:array[1..8] of -2..2=(-1,1,2,2,1,-1,-2,-2); dy:array[1..8] of -2..2=(-2,-2,-1,1,2,2,1,-1); var a:array[1..10,1..10] of integer; n:integer; f:text;  procedure cit; var i,j :integer; begin readln(n); for i:=1 to n do for j:=1 to n do a[i,j]:=0; end;  procedure afis; var i,j:integer; begin for i:=1 to n do begin for j:=1 to n do write(f,a[i,j]:3); writeln(f); end; writeln(f); end;  FUNCTION inside (i,j:integer):boolean; begin inside:=(i in [1..n])and (j in [1..n]) end;  procedure back(i,j,pas:integer); var k,inou,jnou:integer; begin a[i,j]:=pas; if pas=n\*n then afis else for k:=1 to 8 do begin inou:=i+dx[k]; jnou:=j+dy[k]; if inside(inou,jnou) and (a[inou,jnou]=0) then back(inou,jnou,pas+1); end; a[i,j]:=0; end;  begin assign(f,'cal.txt'); rewrite(f); cit; back(1,1,1); close(f); end. | #include<fstream.h> const int dx[8]={-1,1,2,2,1,-1,-2,-2}; const int dy[8]={-2,-2,-1,1,2,2,1,-1}; int a[10][10],n; ofstream f("cal.out");  void afis() { int i,j; for(i=1;i<=n;i++) { for(j=1;j<=n;j++) f<<a[i][j]<<" "; f<<endl; } f<<endl; }  int inside(int i,int j) { return i>=1 && i<=n && j>=1 && j<=n; }  void back(int i, int j, int pas) { int k,inou,jnou; a[i][j]=pas; if (pas==n\*n) afis(); else for(k=0;k<8;k++) { inou=i+dx[k]; jnou=j+dy[k]; if (inside(inou,jnou) && a[inou][jnou]==0) back(inou,jnou,pas+1); } a[i][j]=0; }  void main() { cin>>n;; back(1,1,1); } |

La aceste categorii de probleme se adaugă şi cele de optim, care presupun alegerea soluţiei optime dintre cele generate.

De asemenea, problemele de combinatorică se pot rezolva folosind metoda backtracking. Soluţiile cerute se pot reprezenta ca vectori de forma X = (x1, ..., xn)ÎS unde S = S1 x ... x Sn, unde mulţimile S1, ...,Sn sunt mulţimi finite. Soluţiile se generează element cu element şi trebuie să respecte o serie de reguli, în funcţie de problema de generare concretă.  
Algoritmii de tip backtracking prezentaţi în capitolul anterior vor fi folosiţi pentru a genera permutări, aranjamente, etc. Pentru fiecare probleme, însă, putem face modificări pentru a creşte eficienţa algoritmului de rezolvare sau pentru a-l simplifica. Pentru fiecare algoritm vom identifica particularităţile sale şi condiţiile care trebuie puse asupra vectorului soluţie.   
Algoritmii şi programele prezentate mai jos folosesc mulţimi de forma {1,2,…,n} pentru a simplifica lucrurile. Trecerea la o mulţime generală se face setul de simplu deoarece generând indicii, practic se generează mulţimea, şi astfel, oricărui vector soluţie X generat pe baza indicilor i se poate asocia o soluţie dintr-o mulţime oarecare.   
De asemenea, este bine să numărăm câte soluţii generează fiecare program şi să verificăm aceste numere cu ajutorul formulelor cunoscute de la matematică.

***Generarea permutărilor***  
Se dă o mulţime cu n elemente A={a1,a2,…,an}. Se cere să se genereze si să se afişeze toate permutările ei. Altfel spus, se cere să se afişeze toate modurile în care se pot amesteca elementele mulţimii A.  
Folosim pentru generare mulţimea {1,2,…,n}. Condiţiile care se pun sunt următoarele:

* Fiecare element din vectorul X se ia din {1,2,…,n};
* Un element Xk este valid dacă el nu a fost plasat anterior în vectorul soluţie X;
* Când am generat n elemente cu condiţiile de mai sus, atunci avem o soluţie.

Se pot identifica mai multe modalităţi de a verifica dacă elementul Xk a fost plasat deja în vectorul soluţie. Cele mai importante două sunt:

* parcurgerea elementelor deja generate pentru a verifica daca Xk apare sau nu între ele;
* folosirea unui vector cu n elemente în care vom avea valori 0 sau 1 corespunzătoare elementelor mulţimii iniţiale. Valoarea 1 va preciza faptul că elementul de pe poziţia corespunzătoare a fost plasat anterior în vectorul soluţie, iar valoarea 0 că nu.

Corespunzător acestor două moduri de a verifica dacă un element a mai fost sau nu plasat în vectorul soluţie, avem 2 moduri de generare a permutărilor.

***Generarea aranjamentelor***  
Generăm aranjamentele unei mulţimi atunci când ne se cer toate modurile de a alege m elemente distincte dintre cele n ale mulţimii (m<n).  
Această problemă se rezolvă foarte uşor folosind metodele de generarea permutărilor. Singurele modificări presupun citirea numărului m, modificarea condiţiei de soluţie, care va fi k=m în loc de k=n şi a numărului de elemente afişate.

***Generarea combinărilor***  
Generăm combinărilor unei mulţimi presupune o condiţie suplimentară faţă de permutări sau aranjamente. Acest lucru se datorează faptului că generarea combinărilor presupune alegerea în ordine strict crescătoare a elementelor care compun vectorul soluţie.   
Astfel, condiţia de continuare, sau de validare a unui element este aceea că el trebuie să fie strict mai mare decât cel plasat anterior. În acest mod asigurăm faptul că elementele nu se vor repeta şi că vor fi generate în ordine strict crescătoare. Trebuie, însă, să avem grijă să nu punem această condiţie si asupra primului element din vectorul soluţie, deoarece el nu are cu cine să fie comparat.   
O optimizare a algoritmului de generare a combinărilor se poate obţine pornind instrucţiunea for pentru plasarea unui element de la valoare următoare valorii generate anterior. Trebuie să avem grijă la prima poziţie, care nu are element anterior. Am putea iniţializa X0 cu 0. Astfel nu mai trebuie să verificăm dacă elementul Xk este mai mare ca Xk-1.

***Generarea produsului cartezian***  
Se consideră n mulţimi A1, A2, ... , An de forma {1,2..,an}. Să se genereze produsul cartezian al acestor mulţimi.  
Am considerat mulţimile de forma {1,2..,an} pentru a simplifica problema, în special la partea de citire si afişare, algoritmul de generare rămânând nemodificat.  
Identificăm următoarele particularităţi şi condiţii:

* Fiecare element Xk din vectorul soluţie aparţine unei mulţimi {1,2..,ak}. Aici este o diferenţă faţă de algoritmii anteriori în care fiecare element din soluţie era luat din aceeaşi mulţime şi trebuie să avem grijă la acest fapt când scriem programul.
* Nu există condiţii între elementele vectorului soluţie.
* Obţinem soluţia când s-au generat n valori.

***Generarea submulţimilor unei mulţimi***  
Generarea submulţimilor unei mulţimi A cu n elemente se poate face cu ajutorul algoritmului de generare a combinărilor, apelându-l repetat cu valorile 1, 2, ..., n pentru a genera submulţimile cu un element, apoi cele cu două elemente, apoi cu 3 elemente etc.   
Această modalitate de rezolvare este şi mai complicată şi mai puţin eficientă decât următoarea, care se bazează pe generarea produsului cartezian {0,1}n. Această a doua metodă este eficientă deoarece generează 2n soluţii, adică exact atâtea câte submulţimi are o mulţime cu n elemente.  
Aşadar, generăm toate combinaţiile de lungime n cu valorile 0 şi 1. Pentru fiecare combinaţie parcurgem soluţia X şi afişăm elementele din mulţimea A cărora le corespund valori 1 în X. Astfel, pentru combinaţia 001011 vom afişa elementele de pe poziţiile 3, 5 şi 6 din mulţimea iniţială.

***Generarea partiţiilor unei mulţimi***  
Generăm partiţiilor unei mulţimi presupune împărţirea mulţimii în mulţimi nevide şi disjuncte care reunite să dea întreaga mulţime. Putem, ca şi în cazurile anterioare, să considerăm mulţimea {1,2,…,n}. Construim un vector soluţie în care pentru fiecare element vom trece submulţimea în care îl vom include. Această submulţime mai este numită şi clasă.   
Algoritmul generează pe rând toate modalităţile de a împărţi elementele mulţimii {1,2,…,n} folosind mai întâi o singură clasă, apoi două, ş.a.m.d. până la n clase.