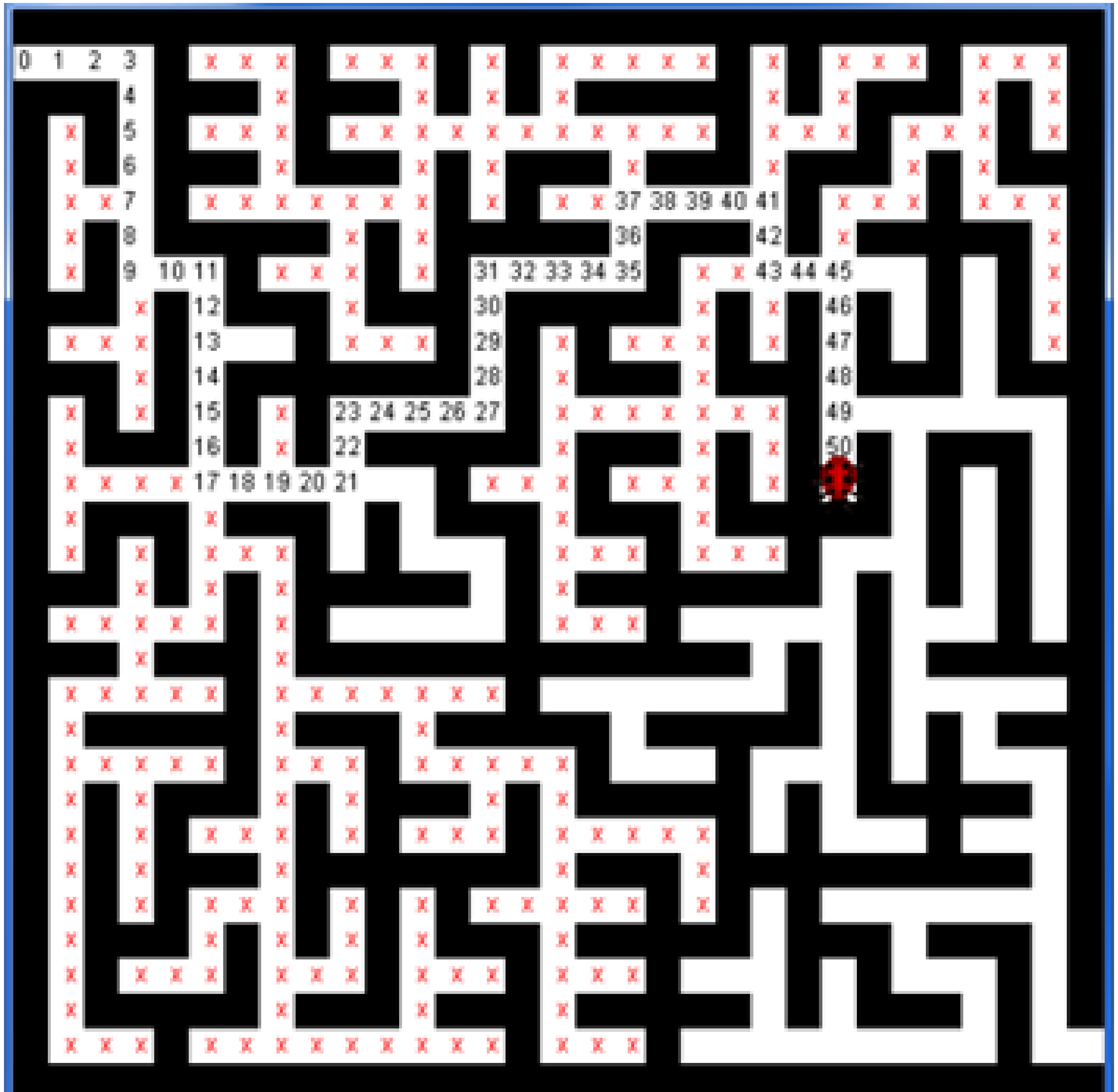
Facultatea de Matematică și Informatică

Lecții de pregătire – Admitere 2020

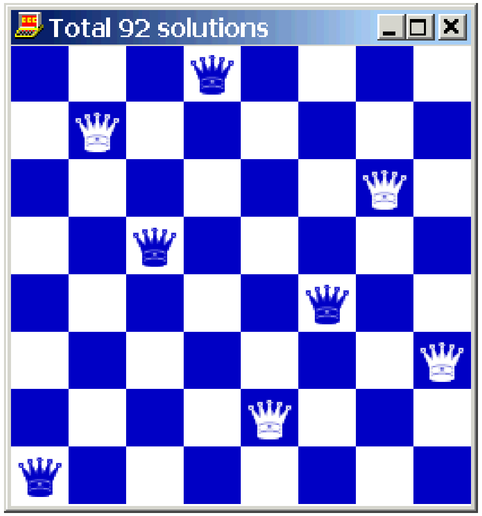
Rezolvarea problemelor folosind metoda backtracking

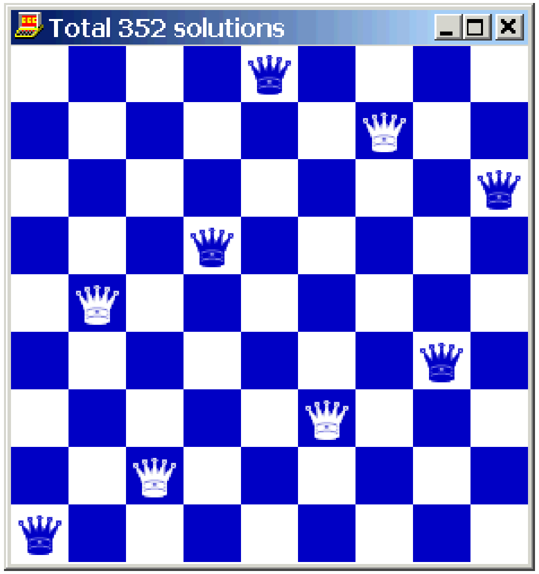
Florentin Ipate

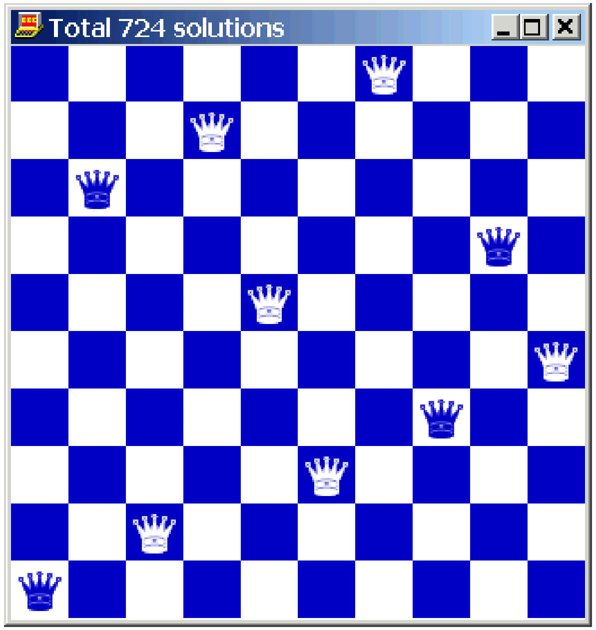
Exemplu: ieșirea din labirint



Exemplu: aranjarea a n regine







Exemplu: rezolvarea unui sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Exemplu: rezolvarea unui sudoku

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Agenda

- Metoda Backtracking: prezentare generală
  - Varianta nerecursivă
  - Varianta recursivă
- Exemple
  - Generarea permutărilor, aranjamentelor, combinărilor
  - Problema celor *n* regine
  - Problema colorării hărţilor
  - Şiruri de paranteze ce se închid corect
  - Partiţiile unui număr natural
- Probleme de backtracking date la admitere

Metoda Backtracking

- Se foloseşte în cazul problemelor a căror soluţie este un vector $x = (x_1, x_2, ..., x_n)$ unde fiecare x_i apartine unei ***mulţimi finite*** A_i, elementele mulțimilor A_i aflându-se într-o ***relaţie de ordine bine stabilită.***
- Între componentele x_i ale vectorului sunt precizate anumite relaţii numite ***condiţii interne.***
- Mulţimea $A = \prod_{i=1}^{n} A_i$ se numeşte ***spaţiul soluţiilor posibile***.
- Soluţiile posibile care satisfac condiţiile interne se numesc ***soluţii rezultat***.
- Generarea tuturor elementelor produsului cartezian nu este acceptabilă (căutare exhaustivă într-un spaţiu de dimensiuni mari).

Metoda Backtracking

- Dimensiunea spaţiului soluţiilor posibile

  $\mathbf{A_1 \times A_2 \times \ldots \times A_n}$ are $|\mathbf{A_1}| \times |\mathbf{A_2}| \times \ldots \times |\mathbf{A_n}|$ elemente
- Metoda backtracking încearcă micşorarea timpului de calcul, realizând o ***căutare sistematică*** în spaţiul soluţiilor posibile.
- Vectorul x este construit progresiv, începând cu prima componentă. Se avansează cu o valoare x_k dacă este satisfăcută ***condiţia de continuare.***
- ***Condiţiile de continuare rezultă de obicei din condiţiile interne.*** Ele sunt strict necesare, ideal fiind să fie şi suficiente.

Metoda Backtracking

- Backtracking = parcurgerea limitată (conform condiţiilor de continuare) în adâncime a unui arbore.
- Spaţiul soluţiilor este organizat ca un arbore
  - un vârf este **viabil** dacă sunt şanse să se găsească o soluţie explorând subarborele cu rădăcina în acel vârf
  - sunt exploraţi numai subarborii cu rădăcini viabile

Backtracking nerecursiv

```
k←1;
cât timp k > 0 repetă
  dacă  ( k = n+1 ) atunci  {am găsit o soluţie}
     prelucrează(x1,..., xn); {afişează soluţie}
     k←k-1; {revenire după găsirea soluţiei}
  altfel
    dacă ( ∃v ∈ Ak netestat) atunci
        xk←v;     { atribuie }
      dacă (x1,...,xk îndeplineşte cond. continuare)
            atunci
                  k←k+1;     { avansează }
      sfârşit dacă
    altfel
       k←k-1;       { revenire }
    sfârşit dacă
sfârşit cât timp
```

Pentru cazul particular A_i ={1, 2,…,*n*}, ∀ i=1,...,*n*, algoritmul se rescrie astfel:

```
xi ← 0, ∀i=1,...,n
k←1;
cât timp k > 0 repetă
  dacă  ( k = n+1 ) atunci {am găsit o soluţie}
     prelucrează(x1,..., xn); {afişează soluţie}
     k←k-1; {revenire după găsirea soluţiei}
  altfel
    dacă (xk<n) atunci
        xk← xk + 1;     { atribuie }
      dacă (x1,...,xk îndeplineşte cond. continuare)
            atunci
                 k←k+1;     { avansează }
      sfârşit dacă
    altfel
       xk←0; k←k-1;        { revenire }
    sfârşit dacă
sfârşit cât timp
```

Backtracking recursiv

Descriem varianta recursivă pentru **cazul particular** $A_i = \{1, 2, \ldots, n\}$, $\forall$ i=1,...,n. Apelul iniţial este **backrec(1).**

```
procedura backrec(k)
dacă (k=n+1) atunci  {am găsit o soluţie}
  prelucrează(x1,..., xn); {afişează soluţie}
altfel
  i←1;
  cât timp i<=n repetă {toate valorile posibile }
     xk←i;
     dacă (x1,...,xk îndeplineşte cond. continuare)
        atunci backrec(k+1);
     sfârşit dacă
     i←i+1;
  sfârşit cât timp
sfârşit dacă
sfârşit procedura
```

Exemple

1. Generarea permutărilor, aranjamentelor, combinărilor
2. Problema celor *n* regine
3. Colorarea hărţilor
4. Şiruri de paranteze ce se închid corect
5. Partiţiile unui număr natural

6. Generarea unor numere cu proprietăţi specificate
7. Generarea produsului cartezian
8. Generarea tuturor soluţiilor unei probleme, pentru a alege dintre acestea o soluţie care minimizează sau maximizează o expresie

Generarea permutărilor

Se citeşte un număr natural *n*. Să se genereze toate permutările mulţimii $\{1, 2, \ldots, n\}$.

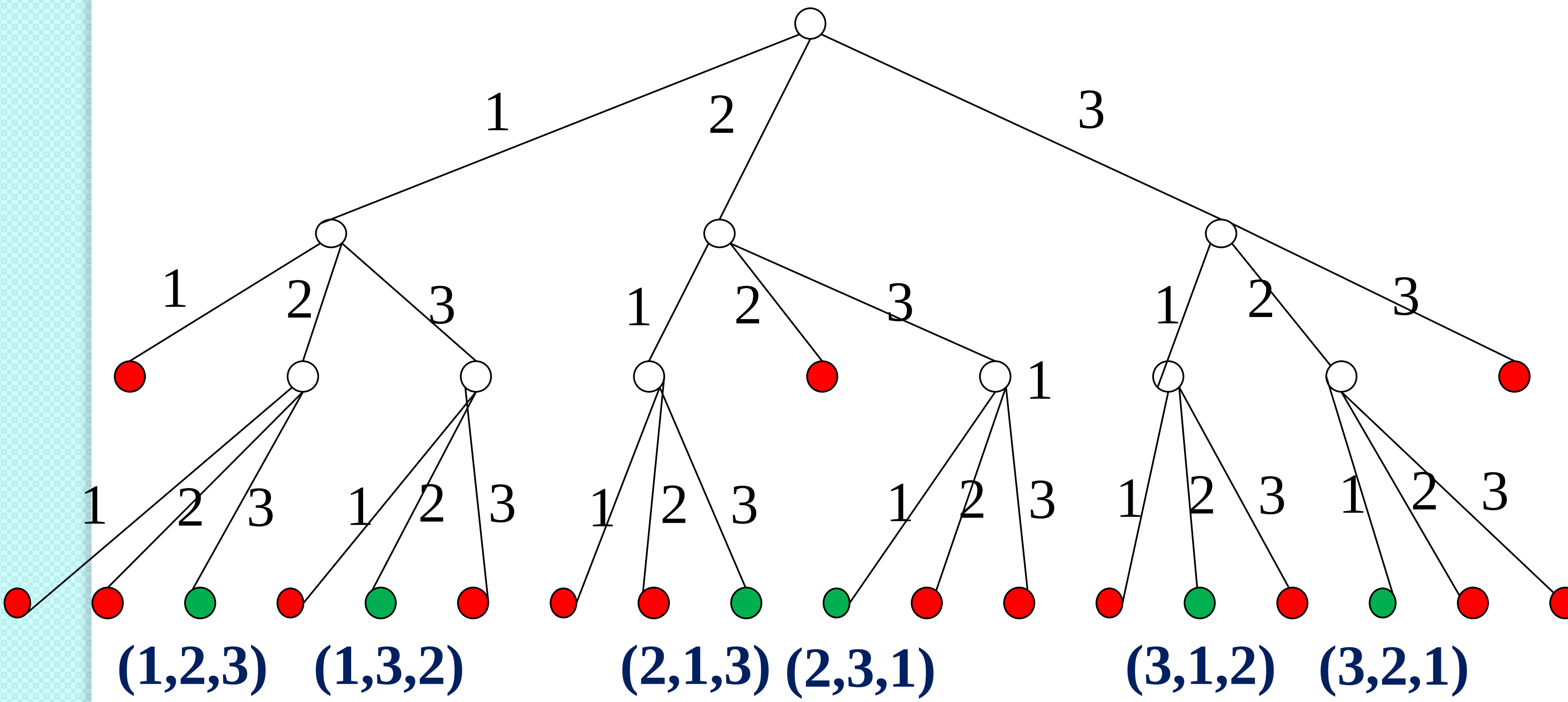
- Reprezentarea soluţiei ?
- Condiţii interne ?
- Condiţii de continuare ?

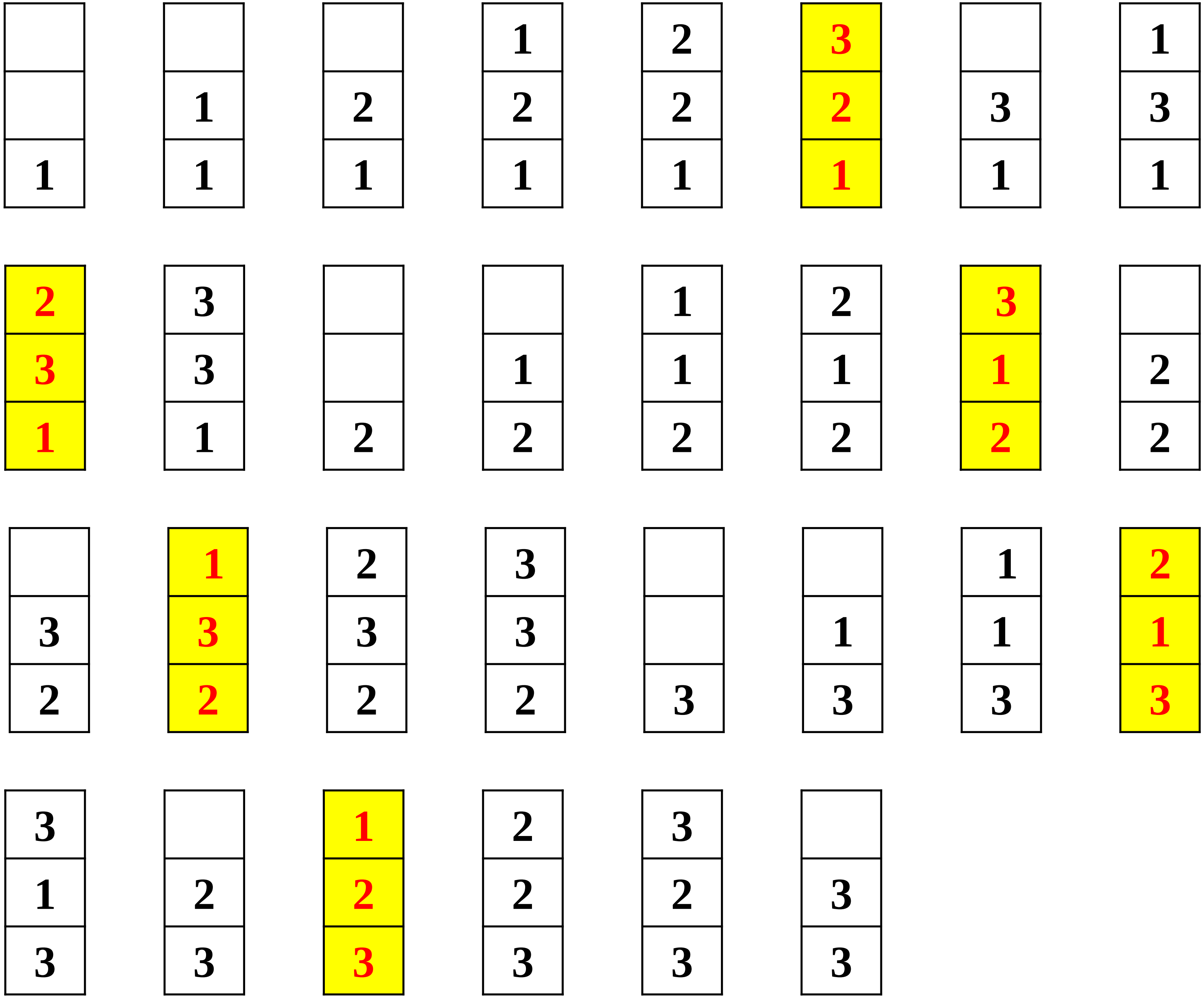
- ✓ $x = (x_1, x_2, \ldots, x_n)$
- ✓ Condiţii interne: pentru orice $i \neq j$, $x_i \neq x_j$
- ✓ Condiţii de continuare: pentru orice $\mathbf{i<k}$, $x_i \neq x_k$

Exemplu: *n* = 3

$= \{1,2,3\} \times \{1,2,3\} \times \{1,2,3\}$

$x = (x_1, x_2, x_3)$




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

<table>
<tr><td>

```pascal
type stiva = array[1..20] of
    integer;
var n,nrsol:integer;
```

</td><td>

```c
int n, nrsol = 0, x[20];
```

</td></tr>
<tr><td>

```pascal
procedure tipar(x:stiva);
var i:integer;
begin
 nrsol := nrsol+1;
 write('Solutia ',nrsol,': ');
 for i:=1 to n do
   write(x[i],' ');
 writeln;
end;
```

</td><td>

```c
void tipar(int x[])
{
 nrsol++;
 printf("Solutia %d:",
    nrsol);
 for(int i=0; i<n; i++)
     printf(" %d ", x[i]);
 printf("\n");
}
```

</td></tr>
<tr><td>

```pascal
function cont(x:stiva;
k:integer):boolean;
var i:integer;
begin
 cont:=true;
 for i:=1 to k-1 do
    if (x[i]=x[k]) then
        cont:=false;
end;
```

</td><td>

```c
int cont(int x[], int k)
{
   for(int i=0;i<k;i++)
        if (x[i]==x[k])
             return 0;
   return 1;
}
```

</td></tr>
</table>

<pre>procedure back();
var k,i:integer;
begin
 for i:=1 to n do x[i]:=0;
 k:=1;
 while k>0 do
   begin
   if k=n+1 then begin
      tipar(x);
      k:=k-1;
      end
   else
      if x[k]<n then
        begin
         x[k]:=x[k]+1;
         if cont(x,k) then
                 k:=k+1;
         end
      else begin
           x[k]:=0;
           k:=k-1;
           end;
   end;
end;</pre>
	<pre>void back()
	{
	   int k = 0;
	   for(int i=0; i<n; i++)
	           x[i]=0;
	   while(k > -1)
	   {
	     if (k==n) {
	           tipar(x);
	           k--; }
	     else
	       if(x[k]<n)
	       {
	          x[k]++;
	          if (cont(x,k))
	              k++;
	       }
	      else {
	          x[k]=0;
	         k--;
	       }
	   }
	}</pre>

```pascal
procedure backrec(k:integer);
var i:integer;
begin
  if k=n+1 then
      tipar(x)
  else
     for i:=1 to n do
         begin
         x[k]:=i;
         if cont(x,k) then
                backrec(k+1);
         end
end;
```

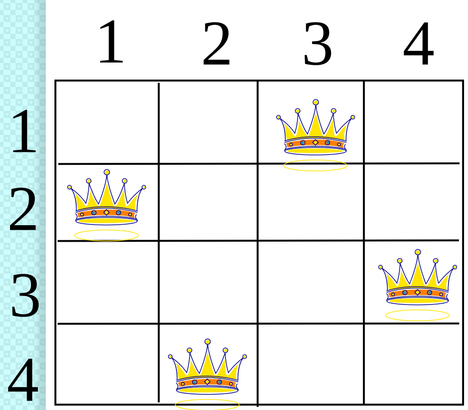
```c
void backrec(int k)
{
  if(k==n)
     tipar(x);
  else
   for(int i=1;i<=n ;i++)
   {
      x[k]=i;
      if (cont(x,k))
           backrec(k+1);
   }
}
```

```pascal
BEGIN
 write('Dati n: '); readln(n);
 nrsol := 0;
 back();
   { sau }
   { backrec(1); }
 readkey;
END.
```

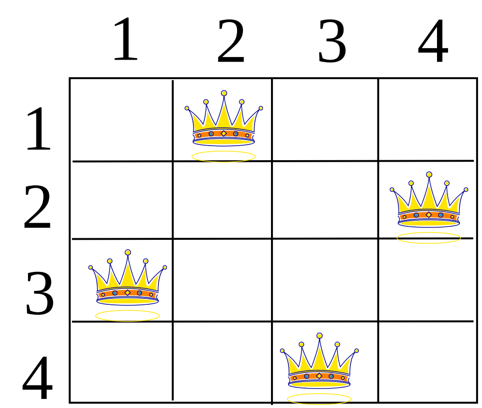
```c
int main()
{
    printf("Dati n:");
    scanf("%d",&n);
    back();
      // sau
      // backrec(0);
    return 0;
}
```

Problema celor *n* regine

Fiind dată o tablă de şah de dimensiune *n* x *n*, se cer toate soluţiile de aranjare a *n* regine astfel încât să nu atace (să nu se afle două regine pe aceeaşi linie, coloană sau diagonală).



- Reprezentarea soluţiei ?
- Condiţii interne ?
- Condiţii de continuare ?



✓ $x = (x_1, x_2, \ldots, x_n)$, x_k = coloana pe care este plasată regina de pe linia k

✓ Condiţii interne: pentru orice $i \neq j$, $x_i \neq x_j$ și $|x_i - x_j| \neq |j-i|$.

✓ Condiţii de continuare: pentru orice **i<k**, $x_i \neq x_k$ și $|x_i - x_k| \neq k-i$.

Problema celor *n* regine

Algoritmul este acelaşi ca la permutări, se modifică doar condiţiile de continuare

Pascal	C

`function cont(x:stiva;`
`k:integer):boolean;`
`var i:integer;`
`begin`
` cont:=true;`
` for i:=1 to k-1 do`
`  if (x[i]=x[k]) or`
`    (abs(x[k]-x[i])=abs(k-i))`
`  then`
`     cont:=false;`
`end;`
	`int cont(int x[], int k)`
	`{`
	
	` for(int i=0;i<k;i++)`
	`  if ((x[i]==x[k]) ||`
	`    (abs(x[k]-x[i])== k-i))`
	`        return 0;`
	` return 1;`
	
	`}`

Generarea aranjamentelor

Se citesc două numere naturale *n* şi *p*. Să se genereze toate submulţimile mulţimii {1,2, ..., *n*} de *p* elemente. Două mulţimi cu aceleaşi elemente, la care ordinea acestora diferă, sunt considerate diferite.

- Reprezentarea soluţiei ?
- Condiţii interne ?
- Condiţii de continuare ?

n=3, p=2

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

- ✓ $x = (x_1, x_2, \ldots, x_p)$
- ✓ Condiţii interne: pentru orice $i \neq j$, $x_i \neq x_j$
- ✓ Condiţii de continuare: pentru orice $\mathbf{i<k}$, $x_i \neq x_k$

Algoritmul este acelaşi ca la permutări, se modifică doar dimensiunea stivei

<pre>procedure tipar(x:stiva);
var i:integer;
begin
 nrsol := nrsol+1;
 write('Solutia ',nrsol,': ');
 for i:=1 to p do
   write(x[i],' ');
 writeln;
end;</pre>
	<pre>void tipar(int x[])
	{
	 nrsol++;
	 printf("Solutia %d:",
	    nrsol);
	 for(int i=0; i<p; i++)
	      printf(" %d ", x[i]);
	 printf("\n");
	}</pre>

<pre>procedure back();
var k,i:integer;
begin
 for i:=1 to p do x[i]:=0;
 k:=1;
 while k>0 do
   begin
   if k=p+1 then begin
      …</pre>
	<pre>void back()
	{
	   int k = 0;
	   for(int i=0; i<p; i++)
	           x[i]=0;
	   while(k > -1){
	     if (k==p) { tipar(x);
	                 k--; }
	     else
	        …</pre>

Generarea combinărilor

Se citesc două numere naturale *n* şi *p*, *n*≥*p*. Să se genereze toate submulţimile mulţimii {1,2, ..., *n*} având *p* elemente. Două mulţimi cu aceleaşi elemente, la care ordinea acestora diferă, sunt considerate egale.

- Reprezentarea soluţiei ?
- Condiţii interne ?
- Condiţii de continuare ?

n=4, p=3

1 2 3

1 2 4

1 3 4

2 3 4

- ✓ $x = (x_1, x_2, \ldots, x_p)$
- ✓ Condiţii interne: pentru orice $i<j$, $x_i<x_j$
- ✓ Condiţii de continuare: pentru orice $k>1$, $\mathbf{x_{k-1}<x_k}$

Algoritmul este acelaşi ca la aranjamente, se modifică în plus funcţia de continuare

<pre>function cont(x:stiva;
k:integer):boolean;
var i:integer;
begin
 cont:=true;
 if k>1 then
    if x[k]<=x[k-1] then
        cont:=false;
end;</pre>
	<pre>int cont(int x[], int k)
	{
	
	 if (k>0)
	    if (x[k] <= x[k-1])
	        return 0;
	 return 1;
	}</pre>

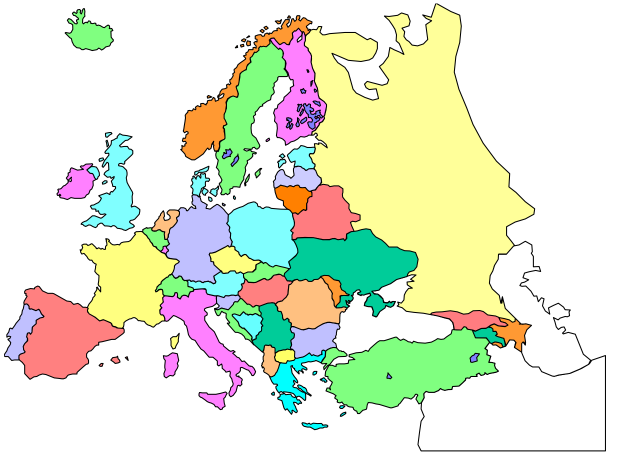
<pre>procedure back();
var k,i:integer;
begin
 for i:=1 to p do x[i]:=0;
 k:=1;
 while k>0 do
   begin
   if k=p+1 then begin
      …</pre>
	<pre>void back()
	{
	   int k = 0;
	   for(int i=0; i<p; i++)
	           x[i]=0;
	   while(k > -1){
	     if (k==p) { tipar(x);
	                 k--; }
	     else
	       …</pre>

Întrebare

- Este importantă evitarea situaţiei în care se tipăreşte de două ori o mulţime, din cauză că este scrisă în altă ordine: {1,2,4} vs. {1,4,2}.
- Ce valori poate să ia elementul de pe nivelul k al stivei?
  a) între 1 si n
  b) între k si p
  c) între k si n-p+k
  d) între x[k-1]+1 si n-p+k

Colorarea hărţilor

Se consideră o hartă. Se cere colorarea ei folosind cel mult 4 culori, astfel încât oricare două ţări vecine să fie colorate diferit.



- Reprezentarea soluţiei ?
- Condiţii interne ?
- Condiţii de continuare ?

Problema matematică: orice hartă poate fi colorată utilizând cel mult 4 culori (rezultat demonstrat în 1976 de către Appel and Haken – unul dintre primele rezultate obţinute folosind tehnica demonstrării asistate de calculator)

Colorarea hărţilor

- **Reprezentarea soluţiei**: $x = (x_1, x_2,\dots, x_n)$, unde x_k = culoarea curentă cu care este colorată ţara k, $x_k \in \{1,2,3,4\}$
- **Condițiile interne**: $x_i \neq x_j$ pentru orice două țări vecine i și j.
- **Condiții de continuare**: $x_i \neq x_k$ pentru orice țară $i \in \{1,2,\dots,k-1\}$ vecină cu ţara k.
- Folosim o matrice **a** pentru a memora relaţia de vecinătate dintre ţări:

$$a_{ij} = \begin{cases} 1, \text{ dacă țările i și j sunt vecine} \\ 0, \text{ altfel} \end{cases}$$

```pascal
function cont(x:vector;
k:integer):boolean;
 var i:integer;
 begin
 cont:=true;
 for i:=1 to k do
  if((x[i]=x[k]) and
      a[k,i]=1)) then
         cont:=false;
end;
```

```c
int cont(int x[], int k){


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

```pascal
procedure backrec(k:integer);
var i:integer;
begin
  if k=n+1 then
      tipar(x)
  else
    for i:=1 to 4 do
         begin
         x[k]:=i;
         if cont(x,k) then
                backrec(k+1);
         end
end;
```

```c
void backrec(int k){


if(k==n)
    tipar(x);
 else
  for(int i=1;i<=4 ;i++)
   {
     x[k]=i;
     if (cont(x,k))
         backrec(k+1);
    }
}
```

Generarea şirurilor de n paranteze ce se închid corect

Se citeşte de la tastatură un număr natural n, $n \leq 30$. Să se genereze şi să se afişeze pe ecran toate combinaţiile de n paranteze rotunde care se închid corect.

- De exemplu, pentru $n = 4$ se obţin următoarele combinaţii:

(( )), ( )( )

- Pentru $n = 6$ se obţin combinaţiile:

((( ))), (()()), ( )( )( ), ( )(( )), (( ))( )

- Există soluţii ⇔ *n* este par.
- **Reprezentarea soluţiei**: $x = (x_1, x_2,\ldots, x_n)$, unde $x_k \in$ {'(', ')'}
- Notăm **dif = nr$_($ - nr$_)$** la pasul k
- **Condiţii interne (finale)**
  - dif = 0
  - dif ≥ 0 pentru orice secvenţă **{$x_1, x_2,\ldots, x_k$}**
- **Condiţii de continuare**
  - dif ≥ 0 → doar **necesar**
  - **dif ≤ n – k** → şi **suficient**
- **Observaţie**. În implementarea următoare backtracking-ul este **optimal**: se avansează dacă şi numai dacă suntem siguri că vom obţine cel puţin o soluţie. Cu alte cuvinte, **condiţiile de continuare nu sunt numai necesare, dar şi suficiente**.

```pascal
procedure backrec(k:integer);
 begin
  if(k=n+1) then
    tipar(x)
  else
   begin
     x[k]:='(';
     dif:=dif+1;
     if(dif <= n-k+1) then
          backrec(k+1);
     dif:=dif-1;
     x[k]:=')';
     dif:=dif-1;
     if(dif >= 0) then
          backrec(k+1);
     dif:=dif+1;
   end;
end;
```

```c
void backrec(int k)
 {
  if(k==n)
        tipar(x);
  else
   {
        x[k]='(';
             dif++;
             if (dif <= n-k)
                   backrec(k+1);
             dif--;
             x[k]=')';
             dif--;
             if (dif >= 0)
                    backrec(k+1);
             dif++;
   }
 }
```

```pascal
if(n mod 2=0) then
 begin
   dif:=0;
   backrec(1);
 end;
```

```c
if (n%2==0)
     {
        dif=0;
        backrec(0);
     }
```

Partiţiile unui număr natural

Dat un număr natural n, să se genereze toate partițiile lui n ca sumă de numere pozitive ($x_1, x_2, \ldots, x_k \in \{1,2,\ldots,n\}$ cu proprietatea $x_1 + x_2 + \ldots + x_k = n$).

De exemplu, pentru $n = 4$, partițiile sunt

1+1+1+1, 1+1+2, 1+3, 2+2, 4

Pentru $n = 6$, partițiile sunt

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

- **Reprezentarea soluţiei**: $x = (x_1, x_2, \ldots, x_k)$ (stivă de lungime variabilă!), unde $x_i \in \{1,2,\ldots,n\}$
- **Condiţii interne (finale)**
  - $x_1 + x_2 + \ldots + x_k = n$
  - **pentru unicitate:** $x_1 \leq x_2 \leq \ldots \leq x_k$
- **Condiţii de continuare**
  - $x_{k-1} \leq x_k$ (avem $x_k \in \{x_{k-1}, \ldots, n\}$)
  - $x_1 + x_2 + \ldots + x_k \leq n$

```
begin
 k:=1; s:=0;
 while(k>=1) do
  begin
  if(x[k]<n) then
    begin
     x[k]:=x[k]+1; s:=s+1;
     if (s<=n)
       then begin
         if(s=n) then begin
                    tipar(x,k);
                    s:=s-x[k];
                    k:=k-1;
                    end
            else begin
                  k:=k+1;
                  x[k]:=x[k-1]-1;
                  s:=s+x[k];
                  end;
       end
     else begin
           s:=s-x[k];
           k:=k-1;
           end;
    end;
  end;
end;
```

```
void back(){
 int k=0, s=0;
 while(k>=0){
  if(x[k]<n){
    x[k]++; s++;
    if(s<=n) { //cond.cont.
      if(s==n){ // solutie
        tipar(x,k);
        s=s-x[k]; k--;
          //revenire
      }
      else
       { //avansare
        k++;
        x[k]=x[k-1]-1;
        s+=x[k];
       }
    }
   else {//revenire
     s=s-x[k]; k--;
   }
  }
 }
}
```

Probleme date la admitere care se pot rezolva cu metoda backtracking

1. Generarea unor numere cu proprietăţi specificate
2. Generarea produsului cartezian
3. Generarea tuturor soluţiilor unei probleme, pentru a alege dintre acestea o soluţie care minimizează sau maximizează o expresie

Problemă (admitere 2012)

Utilizând metoda backtracking se generează toate numerele cu câte trei cifre impare, cifre care aparţin mulţimii {7,8,1,6,2,3}. Primele 4 soluţii generate sunt, în această ordine: 777, 771, 773, 717. **Cea de a 8-a soluţie generată este:**

a) 737 b) 788 c) 717 d) 731

{7,8,1,6,2,3}
Solutia 1: 777
Solutia 2: 771
Solutia 3: 773
Solutia 4: 717

{7,8,1,6,2,3}
Solutia 5: 711
Solutia 6: 713
Solutia 7: 737
Solutia 8: 731

Algoritmul este asemănator cu cel de la aranjamente

`var cif: array[1..6] of integer = (7,8,1,6,2,3);`	`int n=6, p=3, nrsol = 0, x[3];` `int cif[]={7,8,1,6,2,3};`

`procedure tipar(x:stiva);`
`var i:integer;`
`begin`
` nrsol := nrsol+1;`
` write('Sol. ',nrsol,': ');`
` for i:=1 to p do`
`   write(cif[x[i]],' ');`
` writeln;`
`end;`
	`void tipar(int x[])`
	`{`
	`nrsol++;`
	`printf("Sol. %d:", nrsol);`
	`for(int i=0; i<p; i++)`
	` printf("%d", cif[x[i]-1]);`
	`  printf("\n");`
	`}`

`function cont(x:stiva;`
`k:integer):boolean;`
`begin`
` cont:=true;`
` if cif[x[k]] mod 2 = 0  then`
`        cont:=false;`
`end;`
	`int cont(int x[], int k)`
	`{`
	`  if (cif[x[k]-1]%2==0)`
	`       return 0;`
	`  return 1;`
	`}`

Problemă (admitere 2006)

Se cunosc numărul natural $n \geq 1$ şi două tablouri $X = (x_1, x_2, \ldots, x_n)$ şi $Y = (y_1, y_2, \ldots, y_n)$ cu elemente cifre în baza 10. Spunem că $X \leq Y$ dacă $x_i \leq y_i$ pentru orice $i \in \{1, 2, \ldots, n\}$. Să se scrie proceduri/funcţii care să afişeze :

i) Valoarea 1 sau 0, după cum $X \leq Y$ sau nu.

ii) Toate tablourile Z cu elemente cifre în baza 10 astfel încât $X \leq Z \leq Y$, precum şi numărul lor, în ipoteza că $X \leq Y$.

Reformulare. Se dau vectorii şi cu proprietatea Să se genereze toţi vectorii cu proprietatea

`a:array[1..4] of integer=(1,8,1,1);`
`b:array[1..4] of integer=(2,9,2,9);`
	`int n=4, nrsol = 0, x[4];`
	`int a[]={1,8,1,1},`
	`    b[]={2,9,2,9};`

```
procedure back();
var k,i:integer;
begin
 for i:=1 to n do
     x[i]:=a[i]-1;
 k:=1;
 while k>0 do  begin
   if k=n+1 then
      begin
      tipar(x); k:=k-1;
      end
   else
      if x[k]<b[k] then
         begin
          x[k]:=x[k]+1; k:=k+1;
          end
      else begin
            x[k]:=a[k]-1; k:=k-1;
            end;
   end;
end;
```

```
void back()
{
 int  k = 0;
 for(int i=0; i<n; i++)
       x[i]=a[i]-1;
 while(k > -1){
  if (k==n) {
       tipar(x);
       k--;
  }
  else
    if (x[k]<b[k]){
         x[k]++; k++;
    }
    else
       {
        x[k]=a[k]-1;
        k--;
       }
   }
}
```

<pre>procedure
backrec(k:integer);
var i:integer;
begin
  if k=n+1 then
      tipar(x)
  else
     for i:=a[k] to b[k] do
         begin
         x[k]:=i;
         backrec(k+1);
         end
end;</pre>
	<pre>void backrec(int k){
	
	if(k==n)
	     tipar(x);
	else
	 for(int i=a[k];i<=b[k];i++){
	      x[k]=i;
	      backrec(k+1);
	 }
	}</pre>

Numărul vectorilor cu proprietatea cerută este

$$\prod_{i=1}^{n} (b[i] - a[i] + 1)$$

Problemă (admitere 2011, enunţ modificat)

Se dă un vector *v* de *n* elemente egale cu 1. Prin partiţie a vectorului *v* înţelegem o împărţire a vectorului în subvectori, astfel încât fiecare element al vectorului *v* apare exact o dată într-unul dintre subvectori. Pentru fiecare partitie a vectorului *v* în *p* subvectori $v_{11}, \dots, v_{1n_1}, v_{21}, \dots, v_{2n_2}, \dots, v_{p1}, \dots, v_{pn_p}$, se calculează produsul sumelor elementelor din fiecare subvector al partiţiei, adică $\prod_{i=1}^{p} n_i$.

a) Să se citească *n*, *p* de la tastatură şi să se scrie un program care determină **cel mai mare produs** calculat în acest fel pentru toate partiţiile în *p* subvectori ale vectorului *v*.

b) Există o soluţie la punctul a) care să nu calculeze toate produsele posibile? Justificaţi.

Întrebare

- Se dau $n = 11$, p=3. Care este cel mai mare produs $\prod_{i=1}^{p} n_i$ pentru toate partiţiile în p subvectori ale vectorului v?
  a) 21
  b) 30
  c) 48
  d) 64
- **Observaţie:** $n_1 + n_2 + \ldots + n_p = n$.

Exemplu

- $n = 11, p = 3$: $n_1 + n_2 + n_3 = 11$
- [[1,1,1] , [1,1,1,1] , [1,1,1,1]]
- $n_1 = 3, n_2 = 4, n_3 = 4$, $n_1 \cdot n_2 \cdot n_3 = 48$

n_1	n_2	n_3	Produs
1	1	9	9
1	2	8	16
1	3	7	21
…	…	…	…
3	3	5	45
3	4	4	48

`var n,p,nrsol,prod_max:integer;` `nrsol := 0;  prod_max := 0;`	`int n, p, nrsol = 0,` `x[20], prod_max=0;`

`procedure tipar(x:stiva);`
`var i,prod:integer;`
`begin`
` prod := 1;`
` nrsol := nrsol+1;`
` write('Solutia ',nrsol,': ');`
` for i:=1 to p do begin`
`    write(x[i],' ');`
`    prod:=prod*x[i];`
`   end;`
` write(' are produsul ',prod);`
` writeln;`
` if prod > prod_max then`
`       prod_max := prod;`
`end;`
	`void tipar(int x[])`
	`{`
	`int prod = 1;`
	`nrsol++;`
	`printf("Sol %d: ", nrsol);`
	`for(int i=0; i<p; i++)`
	` {`
	`  printf("%d ", x[i]);`
	`  prod = prod*x[i];`
	` }`
	`printf(" are produsul %d`
	`\n", prod);`
	`if (prod > prod_max)`
	`    prod_max = prod;`
	`}`

```pascal
function cont(x:stiva;
k:integer):boolean;
var s,i:integer;
begin
 s := 0; cont:=true;
 if k>1 then
    if x[k]<x[k-1] then
        cont:=false;
 for i:=1 to k do s:=s+x[i];
 if s>n then
    cont:=false;
 end;
```

```c
int cont(int x[], int k)
{
 int s = 0;

 if (k>0)
   if (x[k] < x[k-1])
       return 0;

 for (int i=0; i<=k; i++)
        s =  s + x[i];
 return (s<=n);
}
```

```pascal
function solutie(x:stiva;
k:integer):boolean;
var s,i:integer;
begin
 s := 0; solutie:=true;
 if k<>p+1 then
    solutie := false;
 for i:=1 to p do
    s := s + x[i];
 if s <> n then
    solutie:=false;
end;
```

```c
int solutie(int x[], int k)
{


 int s = 0;
 if (k!=p)
   return false;
 for (int i=0; i<p; i++)
        s =  s + x[i];
 return (s==n);
}
```

```pascal
procedure back();
var k,i:integer;
begin
 for i:=1 to p do x[i]:=0;
 k:=1;
 while k>0 do
   begin
   if solutie(x,k) then
    begin
      tipar(x);
      k:=k-1;
    end
   else
      if x[k]<n then
        begin
         x[k]:=x[k]+1;
         if cont(x,k) then
               k:=k+1;
         end
      else begin
            x[k]:=0;
            k:=k-1;
            end;
   end;
end;
```

```c
void back(){

 int  k = 0;
 for(int i=0; i<p; i++)
           x[i]=0;
 while(k > -1)
  {
    if (solutie(x,k))
        {
          tipar(x);
          k--;
        }
    else
     if(x[k]<n)
       {
        x[k]++;
        if (cont(x,k))
             k++;
       }
     else {
            x[k]=0;
             k--;
          }
  }
}
```

Există o soluţie la punctul a) care să nu calculeze toate produsele posibile? Justificaţi!

Pentru *n* şi *p* daţi notăm cu *c*, *r* câtul şi restul împărţirii lui *n* la *p*. Elementele ce vor maximiza produsul sunt:

$$\underbrace{c,\ c,\ \ldots,\ c,}_{\text{de } p\text{-}r \text{ ori}}\quad \underbrace{c+1,\ \ldots,\ c+1}_{\text{de } r \text{ ori}}$$

Întrebare

Se dau n = 26, p=4. Care este cel mai mare produs $\prod_{i=1}^{p} n_i$ pentru toate partiţiile în p subvectori ale vectorului v?

a) 1375
b) 1820
c) 1764
d) 1728

Exerciţii propuse

1. Generarea tuturor submulţimilor mulţimii $\{1,2, ..., n\}$.
2. Generarea submulţimilor unei mulţimi cu elemente arbitrare $\{a_1,a_2, ..., a_n\}$.
3. Generarea partiţiilor unei mulţimi.
4. Se dă o sumă s şi n tipuri de monede având valorile a_1, a_2, ..., a_n lei. Se cer toate modalităţile de plată a sumei s utilizând tipurile de monede date.
5. Să se determine numerele A de n cifre, $A = \overline{a_1 a_2 \ldots a_n}$ cu $a_i \in \{1,2\}, i = 1..n,$ care sa fie divizibile cu 2^n.

Vă mulţumesc!

Succes la examenul de admitere!