

TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA" (continuare)

5. Sortarea rapidă (Quicksort)

C.A.R. Hoare - 1961

Quicksort are complexitatea $\mathcal{O}(n \log_2 n)$ dacă elementele tabloului nu sunt deja sortate. În caz contrar, complexitatea degenerază în $\mathcal{O}(n^2)$!!!

Ideea algoritmului:

Un element al unui tablou se numește *pivot* dacă toate elementele aflate în stânga sa sunt mai mici decât el și toate aflate în dreapta sa sunt mai mari.

Exemplu:

$t = (12, 7, 15, 29, 54, 109, 88, 73)$ – inițial

$t = (7, 12, 15, 29, 54, 73, 88, 109)$ – sortat crescător

Proprietățile unui pivot:

- În tabloul sortat crescător, un pivot rămâne pe aceeași poziție!!!
- Un tablou este sortat crescător dacă și numai dacă orice element este pivot!!!

Presupunem că tabloul t este sortat =>

$$t[0] \leq t[1] \leq t[2] \leq \dots \leq t[n-2] \leq t[n-1]$$

=> orice element $t[i]$ este pivot!

Presupunem că orice element este pivot =>

- $t[0]$ este pivot => $t[0] \leq \{t[1], \dots, t[n-1]\}$
- $t[1]$ este pivot => $t[0] \leq t[1] \leq \{t[2], \dots, t[n-1]\}$
- $t[2]$ este pivot => $t[0] \leq t[1] \leq t[2] \leq \{t[3], \dots, t[n-1]\}$
-
- $t[n-2]$ este pivot => $t[0] \leq t[1] \leq \dots \leq t[n-2] \leq t[n-1]$

=> tabloul t este sortat crescător!

Metoda Quicksort presupune pivotarea primului element din secvența curentă și reluarea acestui proces pentru secvențele din stânga și dreapta sa.

t = (40, 30, 20, 50, 0, 10, 90, 70)

**Indicele elementului
pe care îl pivotez**

i = 0

Indicele activ

j = 5

j = 6

j = 7

0	1	2	3	4	5	6	7
40	30	20	50	0	10	90	70

Elementul care trebuie pivotat

i = 0	i = 1	i = 2	i = 3		j = 5	j = 6	j = 7
0	1	2	3	4	5	6	7
10	30	20	50	0	40	90	70

i = 0	i = 1	i = 2	i = 3	j = 4	j = 5	j = 6	j = 7
0	1	2	3	4	5	6	7
10	30	20	40	0	50	90	70

i = 0	i = 1	i = 2	i = 3	i = 4	j = 5	j = 6	j = 7
0	1	2	3	4	5	6	7
10	30	20	0	40	50	90	70

0	1	2	3	4	5	6	7
10	30	20	0	40	50	90	70

Elementul a devenit pivot!

Reapelăm Quicksort pentru cele două secvențe!

$t = (12, 7, 15, 9, 54, 9, 88, 73)$
 $t = (9, 7, 15, 9, 54, 12, 88, 73)$
 $t = (9, 7, 9, 12, 54, 15, 88, 73)$

$t = (9, 7, 9, 12, 54, 15, 88, 73)$
 $t = (9, 7, 9, 12, 54, 15, 88, 73)$
 $t = (7, 9, 9, 12, 54, 15, 88, 73)$

$t = (7, 9, 9, 12, 54, 15, 88, 73)$
 $t = (7, 9, 9, 12, 15, 54, 88, 73)$

$t = (7, 9, 9, 12, 15, 54, 88, 73)$
 $t = (7, 9, 9, 12, 15, 54, 73, 88)$

```

#include <fstream>
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int pivot(int v[] , int st , int dr)
{
    //di = valoarea cu care se incrementează indicele i
    //dj = valoarea cu care se incrementează indicele j
    int i = st , j = dr , di = 0 , dj = -1, aux;

    while(i < j)
    {
        if(v[i] >= v[j])
        {
            aux = v[i], v[i] = v[j], v[j] = aux;
            aux = di, di = -dj, dj = -aux;
        }

        i = i + di;
        j = j + dj;
    }

    return i;
}

```

```

void qsort(int v[] , int st , int dr)
{
    int p;

    if(st < dr)
    {
        p = pivot(v , st , dr);
        qsort(v , st , p-1);
        qsort(v , p+1 , dr);
    }
}

int main()
{
    int i , n;

    srand(time(NULL));

    n = 1 + rand() % 20;

    int *v = new int[n];

    fstream f;
    f.open("sortare.txt" , ios::out | ios::app);

    f << n << endl;

    for(i = 0; i < n; i++)
    {
        v[i] = -100 + rand() % 201;
        f << v[i] << " ";
    }

    f << endl;

    qsort(v , 0 , n-1);

    for(i = 0; i < n; i++)
        f << v[i] << " ";

    f << endl << endl;

    f.close();

    return 0;
}

```

TEHNICA DE PROGRAMARE "BACKTRACKING"

Problemă: Să se genereze toate permutările de ordin n (numărul lor este $n!$).

Exemplu: există $3! = 6$ permutări de ordin $n = 3$:

1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1

O rezolvare de tip forță-brută pentru $n = 6$ presupune generarea tuturor posibilelor soluții, adică a tuplurilor de forma $p = (p_1, p_2, p_3, p_4, p_5, p_6)$ cu $p_1, \dots, p_6 \in \{1, 2, \dots, 6\}$, și selectarea celor care sunt permutări, adică au toate valorile diferite între ele ($p_1 \neq p_2 \neq \dots \neq p_6$).

1, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 2
.....
1, 1, 1, 1, 1, 6
1, 1, 1, 1, 2, 1
.....
1, 1, 6, 6, 6, 6
1, 2, 1, 1, 1, 1
.....
1, 2, 3, 4, 5, 5
1, 2, 3, 4, 5, 6
.....
6, 6, 6, 6, 6, 5
6, 6, 6, 6, 6, 6

Se observă foarte ușor faptul că se vor genera și testa $6^6 = 46656$ tupluri, din care doar $6! = 720$ vor fi permutări, deci eficiența acestei metode este foarte mică - în jurul unui procent de **1.5%**! Eficiența scăzută a acestei metode este indusă de faptul că se generează multe tupluri inutile, care nu sunt sigur permutări. De exemplu, se vor genera toate tuplurile de forma $p = (1, 1, p_3, p_4, p_5, p_6)$, adică $6^4 = 1296$ de tupluri inutile deoarece $p_1 = p_2$, deci, evident, aceste tupluri nu pot fi permutări!

O rezolvare de tip Backtracking presupune generare progresivă a soluțiilor, evitând generarea unor tupluri inutile, astfel (vom ține cont de faptul că $p_1, \dots, p_6 \in \{1, 2, \dots, 6\}$):

- $p = (1)$ - este o soluție parțială (componentele sale sunt diferite între ele, deci poate fi o permutare), dar nu este o soluție a problemei (nu are 6 componente), astfel că trebuie să adăugăm cel puțin încă o componentă;
- $p = (1, 1)$ - nu este o soluție parțială (componentele sale sunt egale, deci nu vom obține o permutare indiferent de ce valori vom adăuga în continuare), astfel că nu are

sens să adăugăm încă o componentă, ci vom genera următorul tuplu tot cu două componente;

- $p = (1,2)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1,2,1) \\ p = (1,2,2) \end{array} \right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1,2,3,1) \\ p = (1,2,3,2) \\ p = (1,2,3,3) \end{array} \right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3,4)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1,2,3,4,1) \\ p = (1,2,3,4,2) \\ p = (1,2,3,4,3) \\ p = (1,2,3,4,4) \end{array} \right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3,4,5)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1,2,3,4,5,1) \\ p = (1,2,3,4,5,2) \\ p = (1,2,3,4,5,3) \\ p = (1,2,3,4,5,4) \\ p = (1,2,3,4,5,5) \end{array} \right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3,4,5,6)$ – este o soluție parțială care este și soluție a problemei, deci o memorăm sau o prelucrăm (de exemplu, o afișăm), după care vom încerca generarea următorul tuplu. Deoarece ultima componentă are valoarea 6 (ultima posibilă), înseamnă că am epuizat toate valorile posibile pentru aceasta, deci o vom elimina și vom genera următorul tuplu format doar din 5 componente;
- $p = (1,2,3,4,6)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1,2,3,4,6,1) \\ p = (1,2,3,4,6,2) \\ p = (1,2,3,4,6,3) \\ p = (1,2,3,4,6,4) \end{array} \right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3,4,6,5)$ – este o soluție parțială care este și soluție a problemei, deci o memorăm sau o prelucrăm, după care vom genera următorul tuplu;
- $p = (1,2,3,4,6,6)$ – nu este o soluție parțială, dar ultima componentă are valoarea maxim posibilă 6, deci o vom elimina și vom genera următorul tuplu format doar din 5 componente;
- $p = (1,2,3,4,6)$ – ultima componentă are valoarea maxim posibilă 6, deci o vom elimina și vom genera următorul tuplu format doar din 4 componente;
- $p = (1,2,3,5)$ – este o soluție parțială, dar nu este o soluție a problemei;
-
- $p = (6,5,4,3,2,1)$ – este o soluție parțială care este și ultima soluție a problemei, deci o memorăm sau o prelucrăm, după care vom încerca generarea următorului tuplu. Se

observă cu ușurință faptul că, pe rând, nu vom mai găsi niciun tuplu convenabil, deci algoritmul se va termina.

Exemplu:

$$n = 8$$

$p = (4, 1, 2, 3, 6, 5, 8, 7) \rightarrow$ ultima permutare afișată

$p = (4, 1, 2, 3, 6, 5, 8, 8)$

$p = (4, 1, 2, 3, 6, 5, 8)$

$p = (4, 1, 2, 3, 6, 5)$

$p = (4, 1, 2, 3, 6, 6)$

$p = (4, 1, 2, 3, 6, 7)$

$p = (4, 1, 2, 3, 6, 7, 1)$

$p = (4, 1, 2, 3, 6, 7, 2)$

$p = (4, 1, 2, 3, 6, 7, 3)$

$p = (4, 1, 2, 3, 6, 7, 4)$

$p = (4, 1, 2, 3, 6, 7, 5)$

$p = (4, 1, 2, 3, 6, 7, 5, 1)$

$p = (4, 1, 2, 3, 6, 7, 5, 2)$

... ..

$p = (4, 1, 2, 3, 6, 7, 5, 8) \rightarrow$ următoarea permutare afișată (după cea de mai sus)

Forma generală a unui algoritm de tip Backtracking

Vom începe prin a preciza faptul că majoritatea problemele de generare pot fi formalizate astfel: "Fie mulțimile nevide A_1, A_2, \dots, A_n și un predicat $P: A_1 \times \dots \times A_n \rightarrow \{0, 1\}$. Să se genereze toate tuplurile de forma $S = (s_1, s_2, \dots, s_n)$ pentru care $s_1 \in A_1, \dots, s_n \in A_n$ și $P(s_1, s_2, \dots, s_n) = 1$ ". Practic, predicatul P cuantifică o proprietate pe care trebuie să o îndeplinească componentele tuplului S (o soluție a problemei) sub forma unei funcții de tip boolean ($0 = \text{false}$ și $1 = \text{true}$).

De exemplu, problema generării permutărilor poate fi formalizată în acest mod considerând $A_1 = \dots = A_n = \{1, 2, \dots, n\}$ și $P(s_1, s_2, \dots, s_n) := (s_1 \neq s_2) \wedge (s_2 \neq s_3) \wedge \dots \wedge (s_{n-1} \neq s_n)$, unde prin \wedge am notat operatorul logic AND.

Exemplu:

Să se afișeze toate numerele formate din 5 cifre și care au suma cifrelor egală cu 30.

$$S = (s_1, s_2, \dots, s_5)$$

$$A_1 = \{1, 2, \dots, 9\}$$

$$A_2 = \dots = A_5 = \{0, 1, 2, \dots, 9\}$$

$$P(s_1, s_2, \dots, s_5) := s_1 + s_2 + \dots + s_5 = 30$$

În continuare, vom prezenta câteva notații, definiții și observații:

- pentru orice componentă s_k vom nota cu \min_k cea mai mică valoare posibilă a sa, iar cu \max_k pe cea mai mare;
- într-un tuplu (s_1, s_2, \dots, s_k) , componenta s_k se numește *componentă curentă* (asupra sa se acționează în momentul respectiv);
- *condițiile de continuare* reprezintă condițiile pe care trebuie să le îndeplinească tuplul curent (s_1, s_2, \dots, s_k) astfel încât să aibă sens extinderea sa cu o nouă componentă s_{k+1} sau, altfel spus, există valori pe care le putem adăuga la el astfel încât să obținem o soluție $S = (s_1, s_2, \dots, s_k, s_{k+1}, \dots, s_n)$ a problemei;
- *condițiile de continuare* se deduc din predicatul P și sunt neapărat necesare, fără a fi întotdeauna și suficiente;
- tuplul curent (s_1, \dots, s_k) este o *soluție parțială* dacă el îndeplinește condițiile de continuare;
- orice *soluție* a problemei este implicit și soluție parțială, dar trebuie să mai îndeplinească și alte condiții suplimentare.

Folosind observațiile anterioare, forma generală a unui algoritm de tip Backtracking, implementat folosind o funcție recursivă este următoarea:

```
//k reprezintă indicele componentei curente s[k] dintr-un
//tablou unidimensional s indexat de la 1
void bkt(k)
{
    int v;

    //parcurgem toate valorile posibile v pentru s[k]
    for(v = min_k; v <= max_k; v = v + 1)
    {
        //atribuim componentei curente s[k] valoarea v
        s[k] = v;

        //dacă s[1],...,s[k] este soluție parțială
        if(s[1],...,s[k] este soluție parțială)
            //dacă s[1],...,s[k] este o soluție
            if(s[1],...,s[k] este soluție)
                prelucrăm soluția curentă s[1],...,s[k]
            //s[1],...,s[k] este soluție parțială, dar nu este
            //soluție, deci adăugăm o nouă componentă s[k+1]
            else
                bkt(k+1);
    }
}
```


k=1	k=2	k=3
<pre> int v; for(v = 1; v <= 6; v++) { s[1] = 3 //v = 3 bkt(2) } </pre>	<pre> int v; for(v = 1; v <= 6; v++) { s[2] = 4 //v = 4 s[2] = 5 //v = 5 bkt(3) } </pre>	<pre> int v; for(v = 1; v <= 6; v++) { s[3] = 6 //v = 6 } </pre>

Referitor la algoritmul general de Backtracking prezentat mai sus trebuie făcute câteva observații:

- bucățile de cod scrise cu roșu trebuie particularizate pentru fiecare problemă;
- am considerat tabloul s indexat de la 1, ci nu de la 0, pentru a permite o scriere naturală a unor condiții în care se utilizează indicii tabloului;
- \min_k și \max_k se deduc din semnificația componentei $s[k]$ a unei soluții;
- testarea faptului că $s[1], \dots, s[k]$ este soluție parțială se realizează, de obicei, în cadrul unei funcții (în exemplele pe care le vom prezenta ulterior în limbajul C vom utiliza o funcție `int solp(int k)`);
- pentru a testa că $s[1], \dots, s[k]$ este soluție vom ține cont de faptul că $s[1], \dots, s[k]$ este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar pe cele suplimentare lor;
- dacă $s[1], \dots, s[k]$ nu este soluție parțială, atunci nu vom adăuga o nouă componentă $s[k+1]$ prin apelul recursiv `bkt(k+1)`, deci instrucțiunea `for` va continua și componentei curente $s[k]$ i se va atribui următoarea valoare posibilă v , dacă aceasta există, iar în cazul în care aceasta nu există, instrucțiunea `for` corespunzătoare componentei curente $s[k]$ se va termina și, implicit, apelul funcției `bkt` corespunzător, deci se va reveni la componenta anterioară $s[k-1]$.

De exemplu, pentru a genera toate permutările de ordin n , observațiile de mai sus se particularizează, astfel:

- $s[k]$ reprezintă un element al permutării, deci $\min_k=1$ și $\max_k=n$;
- $s[1], \dots, s[k-1], s[k]$ este soluție parțială dacă valoarea componentei curente $s[k]$ nu a mai fost utilizată anterior, adică $s[k] \neq s[i]$ pentru orice $i \in \{1, 2, \dots, k-1\}$. Se observă faptul că această condiție este dedusă din predicatul P (care impune ca toate cele n valori $s[1], \dots, s[n]$ dintr-o permutare de ordin n să fie distincte) și este neapărat necesară (dacă $s[1], \dots, s[k]$ nu sunt distincte, atunci, indiferent de ce valori am atribui celorlalte $n-k$ componente $s[k+1], \dots, s[n]$ nu vom obține o permutare de ordin n), fără a fi și suficientă (dacă valorile $s[1], \dots, s[k]$ sunt distincte nu înseamnă că ele formează o permutare de ordin n , ci trebuie impusă condiția suplimentară $k=n$);

- pentru a testa că $s[1], \dots, s[k]$ este soluție vom ține cont de faptul că $s[1], \dots, s[k]$ este soluție parțială, deci nu vom retesta condițiile de continuare ($s[k] \neq s[i]$ pentru orice $i \in \{1, 2, \dots, k-1\}$), ci doar condiția suplimentară $k=n$.

Aplicând observațiile anterioare în forma generală a algoritmului de Backtracking, obținem următoarea implementare în limbajul C a algoritmului de generare a tuturor permutărilor mulțimii $\{1, 2, \dots, n\}$:

```
#include <iostream>
#include <iomanip>

using namespace std;

//nrsol = numarul solutiilor gasite (initializat automat cu 0)
int s[101], n, nrsol;

//verifică dacă s[1],...,s[k] este o soluție parțială sau nu
bool solp(int k)
{
    for(int i = 1; i < k; i++)
        //daca valoarea s[k] a mai fost folosita anterior,
        //adica pentru una dintre componentele s[1],...,s[k-1],
        //atunci s[1],...,s[k] NU este o soluție parțială
        if(s[i] == s[k])
            return false;

    return true;
}

//k reprezintă indicele componentei curente s[k] dintr-un
//tablou unidimensional s indexat de la 1
void bkt(int k)
{
    int v;

    //parcurgem toate valorile posibile v pentru s[k]
    //mink = 1, maxk = n
    //succ(v) = v + 1
    for(v = 1; v <= n; v++)
    {
        //atribuim componentei curente s[k] valoarea v
        s[k] = v;

        //dacă s[1],...,s[k] este soluție parțială
        if(solp(k) == true)
            //verific dacă s[1],...,s[k] este o soluție
            //știind că s[1],...,s[k] este o soluție partiala,
            //deci verific doar conditiile suplimentare!!!
            //in acest caz, trebuie sa avem k == n, adica
            //sa avem n numere diferite
            if(k == n)
            {
                //avem o solutie s[1],...,s[k] pe care o prelucram,
                //adica o afisam (in acest caz)
```

```

        cout << ++nrsol << ". ";
        for(int i = 1; i <= k; i++)
            cout << s[i] << ", ";
        cout << "\b\b " << endl;
    }
    else
        //s[1],...,s[k] este soluție parțială, dar nu este
        //soluție finală, deci adăugăm o nouă componentă
s[k+1]
        bkt(k+1);
    }
}

int main()
{
    cout << "n = ";
    cin >> n;

    // "pornesc" algoritmul de backtracking
    // începând cu prima componentă
    bkt(1);

    return 0;
}

```

Așa cum am menționat deja, complexitatea minimă a acestui algoritm este $\mathcal{O}(n!)$.

2. Generarea aranjamentelor

Aranjamentele cu m elemente ale unei mulțimi cu n elemente ($m \leq n$) reprezintă toate tuplurile care se pot forma utilizând m elemente distincte dintre cele n ale mulțimii. Numărul lor se notează cu A_n^m și este dat de formula $\frac{n!}{(n-m)!}$. De exemplu, numărul aranjamentelor cu $m = 3$ elemente ale unei mulțimi cu $n = 5$ elemente este $A_5^3 = 60$, o parte a lor fiind: $(1,2,3), (1,3,2), \dots, (3,2,1), \dots, (1,3,5), \dots, (5,3,1), \dots, (3,4,5), \dots, (5,4,3)$.

Se observă foarte ușor faptul că singura diferență față de generarea permutărilor o constituie lungimea unei soluții, care în acest caz este m în loc de n . De fapt, pentru $m = n$, aranjamentele unei mulțimi sunt chiar permutările sale!

La fel ca și în cazul generării permutărilor, putem aproxima complexitatea minimă a acestui algoritm prin numărul soluțiilor pe care le va afișa, deci prin $\mathcal{O}(A_n^m)$.

3. Generarea combinărilor

Combinările cu m elemente ale unei mulțimi cu n elemente ($m \leq n$) reprezintă toate submulțimile cu m elemente ale unei mulțimi cu n elemente. Numărul lor se notează cu C_n^m și este dat de formula $\frac{n!}{m!(n-m)!}$. De exemplu, numărul tuturor submulțimilor cu $m = 3$ elemente ale unei mulțimi cu $n = 5$ elemente este $C_5^3 = 10$, toate aceste submulțimi fiind: $\{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\}, \{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}$ și $\{3,4,5\}$.

Spre deosebire de tupluri, în care contează ordinea elementelor (de exemplu, tuplurile $(1,2,3)$ și $(1,3,2)$ sunt considerate diferite), în cazul submulțimilor aceasta nu

contează (de exemplu, submulțimile $\{1,2,3\}$ și $\{3,1,2\}$ sunt considerate egale). Din acest motiv, trebuie să găsim o posibilitate de a evita prelucrarea unei soluții care are aceleași elemente ca o altă soluție generată anterior, dar în altă ordine. În acest sens, o variantă simplă, dar ineficientă, o reprezintă prelucrarea doar a soluțiilor care au elementele în ordine strict crescătoare, dar astfel vom încălca chiar principiul de bază al metodei:

$n = 5$

$m = 3$

1. 1, 2, 3

2. 1, 2, 4

3. 1, 2, 5

4. 1, 3, 2

5. 1, 3, 4

6. 1, 3, 5

7. 1, 4, 2

8. 1, 4, 3

9. 1, 4, 5

10. 1, 5, 2

11. 1, 5, 3

12. 1, 5, 4

13. 2, 1, 3

14. 2, 1, 4

15. 2, 1, 5

16. 2, 3, 1

17. 2, 3, 4

18. 2, 3, 5

19. 2, 4, 1

20. 2, 4, 3

21. 2, 4, 5

22. 2, 5, 1

23. 2, 5, 3

24. 2, 5, 4

25. 3, 1, 2

26. 3, 1, 4

27. 3, 1, 5

28. 3, 2, 1

29. 3, 2, 4

30. 3, 2, 5

31. 3, 4, 1

32. 3, 4, 2

33. 3, 4, 5

34. 3, 5, 1

35. 3, 5, 2

36. 3, 5, 4

37. 4, 1, 2

38. 4, 1, 3

39. 4, 1, 5

40. 4, 2, 1

41. 4, 2, 3

42. 4, 2, 5

43. 4, 3, 1

44. 4, 3, 2

45. 4, 3, 5

46. 4, 5, 1

47. 4, 5, 2

48. 4, 5, 3

49. 5, 1, 2

50. 5, 1, 3

51. 5, 1, 4

52. 5, 2, 1

53. 5, 2, 3

54. 5, 2, 4

55. 5, 3, 1

56. 5, 3, 2

57. 5, 3, 4

58. 5, 4, 1

59. 5, 4, 2

60. 5, 4, 3

Backtracking, acela de a evita generarea și testarea unor tupluri inutile cât mai devreme posibil. O altă variantă o reprezintă generarea doar a soluțiilor cu elemente în ordine strict crescătoare, astfel: *inițializăm componenta curentă cu prima valoare strict mai mare decât componenta anterioară* ($\min_k = s[k-1] + 1$). Această variantă este cea mai eficientă variantă posibilă, deoarece nu se generează și testează tupluri inutile și, mai mult, orice tuplu este soluție parțială (elementele sale sunt generate direct în ordine strict crescătoare, deci sunt distincte), ceea ce înseamnă că putem renunța la testarea condițiilor de continuare!

Astfel, vom obține următorul algoritm eficient de tip Backtracking pentru generarea combinărilor:

```
#include <iostream>
#include <iomanip>

using namespace std;

//nrsol = numarul solutiilor gasite (initializat automat cu 0)
int s[101], n, nrsol, m;

//k reprezintă indicele componentei curente s[k] dintr-un
//tablou unidimensional s indexat de la 1
void bkt(int k)
{
    int v;

    //parcurgem toate valorile posibile v pentru s[k]
    //mink = s[k-1] + 1 pentru a obtine
    //s[1] < s[2] < s[3] <...< s[k]
    //maxk = n
    //succ(v) = v + 1
    for(v = s[k-1] + 1; v <= n; v++)
    {
        //atribuim componentei curente s[k] valoarea v
        s[k] = v;

        //dacă s[1],...,s[k] este soluție parțială <=>
        //s[1],...,s[k] sunt diferite intre ele <=>
        //implicit adevarat, deoarece s[1] < s[2] < s[3] <...

        //mai ramane sa verificam doar conditia de solutie
```

```

    if(k == m)
    {
        //avem o solutie s[1],...,s[k] pe care o prelucram,
        //adica o afisam (in acest caz)
        cout << ++nrsol << ". ";
        for(int i = 1; i <= k; i++)
            cout << s[i] << ", ";
        cout << "\b\b " << endl;
    }
    else
        //s[1],...,s[k] este soluție parțială, dar nu este
        //soluție finală, deci adăugăm o nouă componentă s[k+1]
        bkt(k+1);
}

int main()
{
    cout << "n = ";
    cin >> n;

    cout << "m = ";
    cin >> m;

    //"pornesc" algoritmul de backtracking
    //incepand cu prima componenta

    //pentru a calcula corect mink pentru k = 1
    //mink = s[0] + 1 = 1
    s[0] = 0;

    bkt(1);

    return 0;
}

```

La fel ca și în cazul generării permutărilor, putem aproxima complexitatea minimă a acestui algoritm prin numărul soluțiilor pe care le va afișa, deci prin $\mathcal{O}(C_n^m)$.