Combinatorică

Capitolul

7

7.1. Introducere

Precizăm de la început că nu va urma prezentarea unor cunoştinţe teoretice de combinatorică acoperitoare pentru acest domeniu. Vom trata pe rând, probleme de programare în care intervin „elemente de combinatorică”, ceea ce presupune că vom prezenta subprobleme de combinatorică cu explicațiile necesare, fără să intrăm în detalii teoretice. Aceste probleme, de regulă, pot fi rezolvate cu mai multe metode, implementate iterativ sau recursiv. Acolo, unde vom considera necesar, vom prezenta mai multe metode de rezolvare.

7.2. Determinarea submulțimilor unei mulțimi

Fie o mulțime M având n elemente numere naturale consecutive: $\{1, 2, \ldots, n\}$. Se pune problema determinării tuturor submulțimilor mulțimii date, printre care şi mulțimea vidă şi mulțimea dată însăşi. Această problemă se poate rezolva apelând la diferite metode.

7.2.1. Generarea submulțimilor cu prelucrare pe biți

În subalgoritmul următor nu vom evidenția detaliile afişării, cum ar fi acoladele sau virgulele, iar operațiile le vom menționa aşa cum sunt ele cunoscute în limbajul Pascal. Reamintim că mulțimea $M = \{1, 2, \ldots, n\}$ are 2^n submulțimi, deoarece cele n elemente, pe rând pot să aparțină sau nu unei submulțimi.

Exemplu

Fie $n = 3$. Numărul submulţimilor este $2^3 = 8$. Mulţimea vidă o afişăm prima, apoi afişăm celelalte 7 submulţimi. Valoarea 7, ($2^3 - 1$), se poate calcula cu operaţia de translatare la stânga a reprezentării binare a valorii 1 care este echivalentă cu înmulţirea cu 2. Pentru a genera submulţimile folosim reprezentarea numerelor aparţinând mulţimii {1, ..., 7}, generate în variabila *nr*. Fiecare reprezentare este translatată la dreapta cu o poziţie (adică este împărţită cu 2) pentru a facilita testarea valorii ultimei cifre a reprezentării. Dacă această cifră este 1, avem o valoare în submulţime egală cu numărul translatărilor + 1.

Numărul de ordine a submulţimii	***nr***	***i* (numărul poziţiilor cu care efectuăm translatare)**	**Ultima cifră din *nr* după translatare**	**Valoarea din submulţime (*i* + 1)**
1				mulţimea vidă
2	00000001	0	1	{1}
	00000001	1	0	-
	00000001	2	0	-
3	00000010	0	0	-
	00000010	1	1	{2}
	00000010	2	0	-
4	00000011	0	1	{1,
	00000011	1	1	2}
	00000011	2	0	-
5	00000100	0	0	-
	00000100	1	0	-
	00000100	2	1	{3}
6	00000101	0	1	{1,
	00000101	1	0	-
	00000101	2	1	3}
7	00000110	0	0	-
	00000110	1	1	{2,
	00000110	2	1	3}
8	00000111	0	1	{1,
	00000111	1	1	2,
	00000111	2	1	3}

```
Algoritm Generare_submulţimi):
  citeşte n
  scrie mulţimea vidă
```

```
  pentru nr=1,1 shl n-1 execută:               { celelalte 2^n – 1 submulţimi }
    pentru i=0,n-1 execută:                { valorile posibile ale submulţimii }
      dacă (nr shr i) and 1 = 1 atunci{ selectăm elementul submulţimii }
        scrie i+1
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit algoritm
```

7.2.2. Algoritm recursiv pentru generarea submulţimilor în ordine lexicografică

Subalgoritmul următor se apelează pentru valoarea $k = 1$. Şirul s se consideră iniţializat cu valori egale cu 0. Vom avea nevoie şi de elementul s_0, deoarece, conform algoritmului, fiecare valoare nouă a şirului, deci şi s_1 se calculează folosind elementul precedent. În variabila *val* generăm valorile elementelor din submulţimi. Aceste valori, vor fi cu cel puţin 1 mai mari decât ultima valoare scrisă într-o submulţime. Datorită acestei strategii vom obţine submulţimi în ordine lexicografică. Imediat după generarea unei valori noi, până la valoarea n, practic, avem o submulţime nouă.

Exemplu

Fie $n = 3$. În variabila *val* generăm valori posibile de introdus în submulţimi. Limita maximă a acestuia este, evident, n. Valoarea k este parametrul subprogramului şi reprezintă numărul elementelor din submulţimea curentă.

Şirul *s*	***k***	***val***	**Submulţimea afişată**	
(0, 0, 0, 0)			mulţimea vidă	
(0, 1, 0, 0)	1	1	{1}	Pentru $k = 1$, $s_1 = val = 1$.
(0, 1, 2, 0)	2	2	{1, 2}	Pentru $k = 2$, $s_2 = val = 2$.
(0, 1, 2, 3)	3	3	{1, 2, 3}	Pentru $k = 3$, $s_3 = val = 3$.
(0, 1, 2, 3)	4			Revenire (submulţimea are cel mult 3 elemente).
(0, 1, 2, 3)	3			Revenim la $k = 2$ (*val* poate fi cel mult 3).
(0, 1, 2, 3)	2	3	{1, 3}	Pentru $k = 2$, următoarea valoare *val* este 3.
(0, 1, 3, 3)	3			Revenim la $k = 2$.
(0, 1, 3, 3)	2			Revenim la $k = 1$.
(0, 2, 3, 3)	1	2	{2}	Pentru $k = 1$, $s_1 = val = 2$.
(0, 2, 3, 3)	2	3	{2, 3}	Pentru $k = 2$, $s_2 = val = 3$.
(0, 2, 3, 3)	3			Revenim la $k = 2$.
(0, 2, 3, 3)	2			Revenim la $k = 1$.
(0, 3, 3, 3)	1	3	{3}	Pentru $k = 1$, $s_1 = val = 3$.
(0, 3, 3, 3)	2			Revenire.

```
Subalgoritm Submulţimi(k):
  pentru val=s[k-1]+1,n execută:
      s[k] ← val
      Afişează(k)
      Submulţimi(k+1)
  sfârşit pentru
sfârşit subalgoritm
```

7.2.3. Algoritm iterativ pentru generarea tuturor submulțimilor

Dacă dorim un algoritm iterativ, avem posibilitatea implementării unui program care se bazează pe metoda backtracking. Inițializăm cu valoarea 0 primul element al şirului *s* în care vom genera, pe rând, valori posibile ale elementelor submulțimilor. După fiecare incrementare a acestuia afişăm submulțimea care se formează prin adăugarea la submulțimea curentă a acestui element. După fiecare afişare trecem la elementul următor din şirul *s* pe care îl inițializăm cu valoarea ultimului element generat. Dacă valoarea acestuia nu se mai poate mări, revenim în *s* la elementul anterior şi încercăm să îl mărim pe acesta, generând astfel toate submulțimile posibile.

Exemplu

Fie $n = 3$.

k	s_k	**Şirul *s***	**Submulțimea afişată**
		(0, 0, 0)	∅
1	$0 < 3, \Rightarrow s_1 = 1$	(1, 0, 0)	{1}
2	$s_2 = 1, 1 < 3, \Rightarrow s_2 = 2$	(1, 2, 0)	{1, 2}
3	$s_3 = 2, 2 < 3, \Rightarrow s_3 = 3$	(1, 2, 3)	{1, 2, 3}
4	$4 > 3$	(1, 2, 3)	ieşim din `cât timp`
3	$s_3 = 3, 3 = 3$	(1, 2, 3)	ieşim din `cât timp`
2	$s_2 = 2, 2 < 3, \Rightarrow s_2 = 3$	(1, 3, 3)	{1, 3}
3	$s_3 = 3, 3 = 3$	(1, 3, 3)	ieşim din `cât timp`
2	$s_2 = 3, 3 = 3$	(1, 3, 3)	ieşim din `cât timp`
1	$s_1 = 1, 1 < 3, \Rightarrow s_1 = 2$	(2, 3, 3)	{2} şi $s_2 = 2$.
2	$s_2 = 2, 2 < 3, \Rightarrow s_2 = 3$	(2, 3, 3)	{2, 3}
3	$s_3 = 3, 3 = 3$	(2, 3, 3)	ieşim din `cât timp`
2	$s_2 = 3, 3 = 3$	(2, 3, 3)	ieşim din `cât timp`
1	$s_1 = 2, 2 < 3, \Rightarrow s_1 = 3$	(3, 3, 3)	{3}
2	$s_2 = 3, 3 = 3$	(3, 3, 3)	ieşim din `cât timp`
1	$s_1 = 3, 3 = 3$	(3, 3, 3)	ieşim din `cât timp`
0		(3, 3, 3)	ieşim din `repetă`

```
Algoritm Submulţimi_iterativ:
  citeşte n
  scrie mulţimea vidă
  k ← 1
  s[k] ← 0
  repetă
    cât timp s[k] < n do begin
      s[k] ← s[k] + 1
      Afişează(k)
      k ← k + 1                    { avansăm la elementul următor }
      s[k] ← s[k-1]                { iniţializăm elementul următor }
    sfârşit cât timp
    k ← k - 1                      { revenim la elementul precedent }
  până când k = 0
sfârşit algoritm
```

7.2.4. Generarea submulțimilor având *k* elemente

În cazul în care ne interesează doar acele submulțimi care au exact *k* elemente, algoritmul prezentat în secțiunea 7.2.3. trebuie modificat pentru a opri adăugarea de noi elemente în submulțime în momentul în care s-au generat deja *k* elemente. Deoarece ştim că în fiecare submulțime avem *k* elemente, subprogramul `Afişează` nu necesită parametru. În plus, variabila *val* nu primeşte toate valorile cuprinse între 1 şi *n*, ci doar valori posibile ținând cont de valoarea lui *k* şi *i*.

Exemplu

Fie *n* = 3 şi *k* = 2. Dintre cele 8 submulțimi exact trei vor avea câte două elemente. În concluzie, în momentul în care am obținut al doilea element pentru o submulțime generată, o afişăm şi trecem la generarea altei submulțimi.

Şirul *s*	***i***	***val***	**Submulțimea afişată**	
(0, 0, 0, 0)			mulțimea vidă	
(0, 0, 0, 0)	1	1		
(0, 1, 0, 0)	2	2	{1, 2}	
(0, 1, 2, 0)				Revenire.
(0, 1, 2, 0)	1	3	{1, 3}	Următoarea valoare posibilă este 3.
(0, 1, 3, 0)	2			Revenire.
(0, 1, 2, 3)	1	2		
(0, 2, 3, 0)	2	3	{2, 3}	
(0, 2, 3, 0)	1	2		Revenire.

```
Subalgoritm Submulţimi(i):
  pentru val=x[i-1]+1,n-k+i execută:
    x[i] ← val
    dacă i < k atunci
      Submulţimi(i+1)
    altfel
      Afişează
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

7.3. Determinarea partițiilor unei mulțimi date

Dacă ni se cere determinarea tuturor partițiilor unei mulțimi M, având n elemente, trebuie să obținem submulțimi disjuncte şi nevide, reuniunea cărora este mulțimea M. De exemplu, submulțimile {1, 4, 16}, {2, 13} şi {25} formează o partiție a mulțimii {1, 2, 4, 13, 16, 25}. Să observăm că în rezolvare putem exploata corespondența biunivocă dintre elementele mulțimii M şi mulțimea $\{1, 2, ..., n\}$, astfel reducând problema la determinarea partițiilor acesteia din urmă.

Vom utiliza un tablou ajutător în care, pentru fiecare element din mulțimea dată M, vom reține numărul de ordine al submulțimii din care face parte. Pentru exemplul de mai sus, acest tablou va avea conținutul: (1, 2, 1, 2, 1, 3).

În algoritm pornim cu partiția în care fiecare submulțime este formată dintr-un singur element: $\{1\}, \{2\}, ..., \{n\}$. Dacă renunțăm la cea de a n-a submulțime şi adăugăm fiecărei submulțimi pe cel de al n-lea element din mulțime, obținem $n - 1$ partiții noi: $\{1, n\}, \{2\}, ..., \{n - 1\}, ..., \{1\}, \{2\}, ..., \{n - 1, n\}$. Procedând similar cu a $(n - 1)$-a submulțime, cu a $(n - 2)$-a, etc. obținem noi partiții. Acum putem renunța pe rând la submulțimile astfel create, adăugându-le la cele existente.

Exemplu

Fie $n = 3$, deci mulțimea ajutătoare pe care o partiționăm este {1, 2, 3}.

Pas	Partiții	Tablou ajutător
1	{1}, {2}, {3}	(1, 2, 3)
2	{1}, {2, 3}	(1, 2, 2)
3	{1, 3}, {2}	(1, 2, 1)
4	{1, 2}, {3}	(1, 1, 2)
5	{1, 2, 3}	(1, 1, 1)

În subalgoritmul următor utilizăm mulțimea *folosit* în care reținem elementele introduse deja într-o submulțime a partiției. Subalgoritmul `Afişează(t)` realizează afişarea submulţimilor pe baza valorilor păstrate în tabloul *t*. Subalgoritmul `Generează(k,folosit)` este apelat din programul principal într-o structură repetitivă de tip **pentru** de $n - 1$ ori (**pentru** `i=n,2` **execută:** `Generează(i, ∅)`). Partiţia {1}, {2}, {3} se afişează la începutul programului după iniţializarea tabloului *t* cu valoarea (1, 2, 3).

```
Subalgoritm Generează(k,folosit):
  pentru j=k-1,1 execută:
    dacă (t[k] ≠ t[j]) şi (t[j] ∉ folosit) atunci
      t[k] ← t[j]
      folosit ← folosit ∪ t[j]
      Afişează(t)
      pentru i=k+1,n execută:
        Generează(i,∅)
      sfârşit pentru
    sfârşit dacă
  sfârşit pentru
  t[k] ← k
sfârşit subalgoritm
```

7.4. Produs cartezian

Vom întâlni multe probleme în care determinarea produsului cartezian a *n* mulţimi M_i ($i = 1, 2, ..., n$) va apărea ca subproblemă. Menţionăm că numărul elementelor din cele *M* mulţimi poate să difere.

7.4.1. Algoritm iterativ

Prezentăm un algoritm de tip backtracking iterativ. Fiecare element al produsului cartezian este un şir *c* având *n* elemente (numărul mulţimilor) din mulţimea {1, 2, ..., *m*} (fiecare mulţime are *m* elemente). Elementele şirului *c* se iniţializează cu 1 şi se afişează, apoi urmează generarea celorlalte şiruri. Cât timp se poate mări valoarea ultimului element, vor urma produsele carteziene (1, 1, ..., 2), (1, 1, ..., *m*). Apoi vom mări valoarea penultimului element, generând corespunzător şirurile care pe ultima poziţie din nou au toate valorile posibile etc.

Exemplu

Fie $n = 3$, $m = 2$.

i	**Produs cartezian**	**Explicaţii**
1, 2, 3	(1, 1, 1)	
3	(1, 1, 2)	Ieşim din cât timp (c_3 a atins valoarea maximă).

2	(1, 2, 1)	
3	(1, 2, 2)	Ieşim din cât timp (c_3 a atins valoarea maximă).
2		Ieşim din cât timp (c_2 a atins valoarea maximă).
1	(2, 1, 1)	
3	(2, 1, 2)	Ieşim din cât timp (c_3 a atins valoarea maximă).
2	(2, 2, 1)	Ieşim din cât timp (c_2 a atins valoarea maximă).
3	(2, 2, 2)	Ieşim din cât timp (c_3 a atins valoarea maximă).
2		Ieşim din cât timp (c_2 a atins valoarea maximă).
1		Ieşim din cât timp (c_1 a atins valoarea maximă).

```
Algoritm Produs_cartezian:
  citeşte n,m
  pentru i=1,n execută:
    c[i] ← 1
  sfârşit pentru
  Afişare
  i ← n
  repetă
    cât timp c[i] < m execută:
      c[i] ← c[i] + 1
      Afişare
      i ← n
    sfârşit cât timp
    c[i] ← 1
    i ← i - 1
  până când i = 0
sfârşit algoritm
```

7.4.2. Algoritm implementat recursiv

În algoritmul următor presupunem că avem n şiruri având lungimi diferite, notate cu $nrelem_i$ (i = 1, 2, ..., n). Soluţia o aşezăm în ordine lexicografică în şirul c, având n elemente. Următorul element al produsului cartezian se obţine adăugând 1 la componenta având indicele cel mai mare şi care este mai mic decât $nrelem_i$.

Exemplu

Fie n = 3. Cele trei mulţimi au cardinaltăţile: $nrelem_1 = 1$, $nrelem_2 = 2$, $nrelem_3 = 2$. $M_1 = \{1\}$, $M_2 = \{1, 2\}$, $M_3 = \{1, 2\}$.

k **(indice în soluţie)**	**Produs cartezian**	***i*** **(valori posibile)**
1	(1, 0, 0)	1
2	(1, 1, 0)	1

3	(1, 1, 1)	1
4	**(1, 1, 1)**	Se afişează rezultatul şi ieşim din apel.
3	(1, 1, 2)	2
4	**(1, 1, 2)**	Se afişează rezultatul şi ieşim din apel.
3	(1, 1, 2)	*i* nu mai creşte, ieşim din apel.
2	(1, 2, 2)	2
3	(1, 2, 1)	1
4	**(1, 2, 1)**	Se afişează rezultatul şi ieşim din apel.
3	(1, 2, 2)	2
4	**(1, 2, 2)**	Se afişează rezultatul şi ieşim din apel.
3	(1, 2, 2)	*i* nu mai creşte, ieşim din fiecare apel.
2	(1, 2, 2)	*i* nu mai creşte, ieşim din fiecare apel.
1	(1, 2, 2)	*i* nu mai creşte, ieşim din fiecare apel.

```
Subalgoritm Descart(k):
  dacă k = n + 1 atunci
    Afişează
  altfel
    pentru i=1,nrelem[k] execută:
      c[k] ← i
      Descart(k+1)
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

7.5. Partițiile unui număr

Prin partiția unui număr natural se înțelege scrierea lui sub formă de sumă de numere naturale în toate modurile posibile. Totuşi, deosebim două tipuri de partiții:

- două partiții se consideră identice dacă au aceeaşi termeni în aceeaşi poziții;
- două partiții se consideră identice dacă au aceeaşi termeni, chiar dacă aceştia se află pe poziții diferite.

7.5.1. Partiții cu elemente diferite

Termenii partiției le vom păstra într-un şir p. Din numărul n vom scădea pe rând valorile 1, 2, ..., n, dar de fiecare dată aplicăm acelaşi procedeu pentru ceea ce a rămas din număr după scădere, până când ajungem la valoarea 0.

Exemplu

Dacă $n = 4$, avem următoarele partiții:

4 = 1 + 1 + 1 + 1

4 = 1 + 1 + 2
4 = 1 + 2 + 1
4 = 1 + 3
4 = 2 + 1 + 1
4 = 2 + 2
4 = 3 + 1
4 = 4

```
Subalgoritm Partiţie(i,n):
  pentru j=1,n execută:
    p[i] ← j
    dacă j < n atunci
      Partiţie(i+1,n-j)
    altfel
      Afişează(i)
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

7.5.2. Partiţii cu elemente şi ordinea elementelor diferite

În cazul în care nu dorim să generăm partiţiile care diferă doar în ordinea termenilor, în algoritmul de mai sus mai adăugăm un test cu ajutorul căruia admitem doar termeni care se succed în ordine crescătoare.

Exemplu

Dacă $n = 4$, avem următoarele partiţii:
4 = 1 + 1 + 1 + 1
4 = 1 + 1 + 2
4 = 1 + 3
4 = 2 + 2
4 = 4

```
Subalgoritm Part(i,n):
  pentru j=1,n execută:
    p[i] ← j
    dacă p[i] ≥ p[i-1] atunci
      dacă j < n atunci
        Part(i+1,n-j)
      altfel
        Afişează(i)
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

7.6. Permutări

Generarea permutărilor în multe surse bibliografice este asociată cu metoda backtracking. Vom vedea că există şi posibilitatea de a genera permutări circular fără ca algoritmul să devină mare consumator de timp. Vom genera elementele unei mulţimi de numere naturale distincte astfel încât între două modalităţi de aşezare una după alta a elementelor să difere cel puţin ordinea a două elemente. În probleme se cer, de regulă, toate aceste modalităţi de aşezare, dar vom întâlni şi probleme în care trebuie determinate permutări, având o anumită proprietate.

7.6.1. Permutări generate circular

Să presupunem că trebuie să determinăm permutările mulţimii $M = \{1, 2, \ldots, n\}$. Permutarea $perm_i = i$, ($i = 1, ..., n$) se numeşte permutare *identică.* Permutările mulţimii M le vom genera prin permutări circulare spre stânga, pornind de la permutarea identică.

Exemplu

Fie $n = 3$.

Permutare inițială	Explicație	Permutare obținută
1 2 3	Rotim permutarea pe lungime 3, începând cu poziția 1.	2 3 **1**
2 3 1	Rotim permutarea pe lungime 3, începând cu poziția 1.	3 1 **2**
3 1 2	Rotim permutarea pe lungime 3, începând cu poziția 1.	1 2 **3**
1 **2** 3	Această permutare a mai fost; rotim permutarea pe lungime 2, începând cu poziția 2.	1 3 **2**
1 3 2	Rotim permutarea pe lungime 3, începând cu poziția 1, deoarece avem o permutare nouă.	3 2 **1**
3 2 1	Rotim permutarea pe lungime 3, începând cu poziția 1.	2 1 **3**
2 1 3	Rotim permutarea pe lungime 3, începând cu poziția 1.	1 3 **2**
1 **3** 2	Această permutare a mai fost; rotim permutarea pe lungime 2, începând cu poziția 2.	1 2 **3**
1 2 **3**	Această permutare a mai fost; rotim permutarea pe lungime 1, începând cu poziția 3, dar nu mai există elemente, deci generarea s-a terminat.	

```
Algoritm Permutări_circulare:
  citeşte n
  pentru i=1,n execută:              { generăm şi afişăm permutarea identică }
    perm[i] ← i
    scrie i
  sfârşit pentru
```

```
  poz ← 1
  repeat
    aux ← perm[poz]      { permutăm circular elementele începând cu poziția k }
    pentru i=poz,n-1 execută:
      perm[i] ← perm[i+1]
    sfârşit pentru
    perm[n] ← aux                                { se încheie permutarea }

    dacă perm[k] = poz atunci  { am revenit la o permutare care a mai fost }
      poz ← poz + 1      { vom permuta circular începând cu o poziție nouă k }
    altfel
      poz ← 1
      pentru i=1,n execută:                    { afişăm permutarea curentă }
        scrie perm[i]
      sfârşit pentru
    sfârşit dacă
              { am ajuns în ultima poziție, nu mai există elemente de permutat }
  până când poz = n
sfârşit algoritm
```

7.6.2. Algoritm recursiv bazat pe metoda backtracking

Cel mai simplu algoritm recursiv se realizează cu metoda backtracking, care din păcate este mare consumatoare de timp. Generăm pe rând toate valorile posibile care pot intra ca elemente în şirul care reprezintă permutarea curentă, dar imediat după selectarea valorii verificăm dacă aceasta nu apare deja în permutarea respectivă. În cazul în care nu îl găsim, îl considerăm aşezat şi generăm următorul element. Funcția logică `Nuafost(i)` verifică dacă valoarea *i* apare sau nu deja în permutarea curentă.

Amintim în încheiere că numărul permutărilor a *n* elemente este egal cu *n*!.

```
Subalgoritm Permutare(i):
  pentru j=1,n execută:                        { generăm valorile mulțimii }
    perm[i] ← j              { alegem o valoare pentru permutarea curentă }
    dacă Nuafost(i) atunci             { verificăm dacă valoarea este bună }
      dacă i < n atunci               { dacă mai trebuie generate elemente }
        Permutare(i+1)
      altfel
        Afişează
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

7.6.3. Algoritm recursiv

Permutarea identică o generăm în programul principal. Considerăm că mulțimea {1} are o singură permutare: {1}. În continuare, dacă unim permutarea mulțimii {1, 2, ..., $n - 1$} cu permutarea mulțimii {n} obținem permutările:

{n, 2, 3, ..., $n - 1$, 1}
{1, n, 3, ..., $n - 1$, 2}
...
{1, 2, 3, ..., $n - 1$, n}

```
Subalgoritm Permutare_2(i):
  dacă i ≤ n atunci
    pentru j=i,1 execută:
                   { interschimbăm pe rând fiecare element cu al j-lea element }
      aux=perm[j]
      perm[j] ← perm[i]
      perm[i] ← aux
      Permutare_2(i+1)            { reluăm procesul pentru restul permutării }
      aux ← perm[j]                          { punem la loc al j-lea element }
      perm[j] ← perm[i]
      perm[i] ← aux
    sfârşit pentru
  altfel
    Afişează
  sfârşit dacă
sfârşit subalgoritm
```

7.6.4. Permutări cu repetiții

Permutările cu repetiții conțin o aceeaşi valoare de mai multe ori. Acest număr de multiplicitate se poate citi sau se poate calcula sau genera pe diverse criterii impuse de problemă. Permutările se generează în şirul *p* în mod asemănător ca în cazul celor fără repetiții, dar în plus, trebuie ținută evidența numărului de apariții în permutare a fiecărei valori. În subalgoritmul următor am notat cu *buc* şirul care reține numărul de multiplicitate a fiecărui număr din şirul *nr*. În aceste condiții, numărul elementelor din permutare este egal cu suma elementelor din şirul *buc*. Subalgoritmul `Permuta(k,j)` se apelează din programul principal cu parametri 1 şi n, unde n este dimensiunea şirului de numere *nr*. Subalgoritmul se ramifică în funcție de valoarea de multiplicitate a valorii curente din permutare.

Exemplu

Fie $n = 2$, şi $buc_1 = 1$, $buc_2 = 2$.

Permutările cu repetiții vor fi: (1, 2, 2), (2, 1, 2), (2, 2, 1).

```
Subalgoritm Permuta(k,j):
  dacă k > n atunci
    Afişează
  altfel
    pentru i=1,j execută:
      p[k] ← nr[i]                   { aşezăm numărul curent în permutare }
      dacă buc[i] = 1 atunci  { dacă ordinul de multiplicitate curent este 1 }
        nr[i] ← nr[j]
        buc[i] ← buc[j]
        Permuta(k+1,j-1)                { urmează alt număr în permutare }
        nr[i] ← a[k]
        buc[i] ← 1
      altfel
        buc[i] ← buc[i] - 1        { scade ordinul de multiplicitate curent }
        Permuta(k+1,j)
                      { după revenire, refacem ordinul de multiplicitate curent }
        buc[i] ← buc[i] + 1
      sfârşit dacă
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

7.7. Aranjamente

În câte moduri se pot împărţi *n* obiecte în grupuri de câte *k* obiecte? Este o întrebare frecventă în problemele de programare. Grupurile de obiecte care se obţin se numesc aranjamente de *n* luate câte *k*. În timpul generării grupurilor trebuie să fim atenţi să nu generăm de două ori acelaşi grup, să nu pierdem nici unul şi un element aşezat într-un aranjament să apară o singură dată.

7.7.1. Algoritm recursiv de generare a aranjamentelor

Algoritmul de generare a acestor aranjamente seamănă cu cel prezentat pentru generarea recursivă a permutărilor. Vom aborda o implementare care foloseşte un şir ajutător *folosit* în care păstrăm valorile introduse deja în aranjamentul curent. Elementul $folosit_i$ primeşte valoarea *adevărat* în momentul în care valoarea *i* se plasează în aranjament şi primeşte valoarea *fals* la revenirea din apelul recursiv, deoarece va fi înlocuit cu următoarea valoare posibilă. Această abordare este mai avantajoasă decât cea prezentată la generarea recursivă a permutărilor, unde pentru fiecare „propunere” nouă trebuiau verificate toate elementele existente deja în permutarea curentă.

Numărul aranjamentelor a n obiecte luate câte k este $\frac{n!}{(n-k)!}$ [*].

Exemplu

Fie $n = 3$ şi $k = 2$. Avem aranjamentele (1, 2); (1, 3); (2, 1); (2, 3); (3, 1); (3, 2).

```
Subalgoritm Aranjament(i):
  pentru j=1,n execută:          { putem alege valori din mulţimea {1, 2, ..., n} }
    dacă nu folosit[j] atunci        { dacă valoarea j nu este deja folosită }
      a[i] ← j                                                      { o folosim }
      folosit[j] ← adevărat         { notăm faptul că valoarea j este folosită }
      dacă i < k atunci                           { trebuie să plasăm k elemente }
        Aranjament(i+1)
      altfel
        Afişează
      sfârşit dacă
      folosit[j] ← fals      { „scoatem” valoarea j din aranjamentul curent }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

7.7.2. Aranjamente cu repetiţii generate iterativ

Aranjamentele cu repetiţii a n elemente luate câte k se generează astfel încât în afară de aranjamentele în care fiecare element apare o singură dată să apară şi aranjamente în care valorile permise apar de 2, 3, ..., k ori.

Exemplu

Fie $n = 3$ şi k = 2. Aranjamentele cu repetiţii în acest caz vor fi:
(1, 1); (1, 2); (1, 3); (2, 1); (2, 2); (2, 3); (3, 1); (3, 2); (3, 3).

```
Algoritm Aranjamente_cu_repetiţii:
  citeşte n, k
  pentru i=1,k+1 execută:                            { iniţializarea şirului a }
    a[i] ← 1
  Afişează
  cât timp adevărat execută:
    i ← 1
    cât timp a[i] = n execută:          { cât timp pe poziţia curentă avem }
                         { valoarea maximă admisă, trecem la următoarea poziţie }
      i ← i + 1
    sfârşit cât timp
```

[*] Alte detalii se vor învăţa la matematică.

```
    dacă i > k atunci                         { am generat şi ultimul element }
      ieşire forţată din cât timp
    sfârşit dacă
    a[i] ← a[i] + 1
                    { reiniţializarea elementelor din faţa elementului curent cu 1 }
    pentru i=i-1,1 execută:
      a[i] ← 1
    sfârşit pentru
    Afişează
  sfârşit cât timp
sfârşit algoritm
```

7.8. Combinări

Combinările a n obiecte luate câte k le obţinem dacă eliminăm din aranjamente acele grupurile care diferă între ele doar ca ordine. Numărul lor este egal cu $\frac{n!}{m!(n-m)!}$.

Modalitatea cea mai simplă de a genera combinări este prin algoritmul de generare a submulţimilor având k elemente. Prezentăm pseudocodul unui astfel de algoritm:

```
Subalgoritm Combinări(j):
  dacă j=k atunci
    Afişează
  altfel
    j ← j + 1
    pentru i=a[j-1]+1,n-k+j execută:
      c[j] ← i
      Combinări(j)
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

Exemplu

Fie $n = 3$ şi $k = 2$. Combinările de 3 luate câte 2 vor fi: (1, 2); (1, 3); (2, 3).

7.8.1. Algoritm recursiv

Pentru generarea combinărilor putem folosi algoritmul prezentat pentru generarea aranjamentelor în care anterior acceptării unei valori verificăm dacă acesta este mai mare decât elementul precedent. Astfel vom evita grupurile de valori care diferă doar prin ordine. Şirul folosit ţine evidenţa valorilor aşezate deja în combinarea curentă. La revenirea din apel elementul din şirul folosit, corespunzător valorii care urmează să fie suprascrisă, se reface astfel încât valoarea respectivă să poată fi refolosită.

```
Subalgoritm Comb(i):
  pentru j=1,n execută:          { putem alege valori din mulţimea {1, 2, ..., n} }
    dacă nu folosit[j] atunci          { dacă valoarea j nu este deja folosită }
      c[i] ← j                                                   { o propunem }
      dacă c[i] > c[i-1] atunci
        folosit[j] ← adevărat      { notăm faptul că valoarea j este folosită }
        dacă i < k atunci                        { trebuie să plasăm k elemente }
          Comb(i+1)
        altfel Afişează
        sfârşit dacă
        folosit[j] ← fals    { „scoatem” valoarea j din aranjamentul curent }
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

7.8.2. Algoritm iterativ pentru generarea combinărilor

```
Algoritm Combinări_Generate_Iterativ:
  citeşte n, k
  pentru i=1,k execută:
    c[i] ← i
  sfârşit pentru                                  { iniţializarea combinării }
  c[0] ← -1              { avem nevoie pentru a simplifica oprirea algoritmului }
  Afişează
  cât timp adevărat execută:
    i ← k
    cât timp c[i] = n-k+i execută:{ căutăm elementul care poate fi mărit }
      i ← i - 1
    sfârşit cât timp
    dacă i = 0 atunci                          { s-au generat toate elementele }
      ieşire forţată din cât timp
    sfârşit dacă
    c[i] ← c[i] + 1
                       { reiniţializarea elementelor aflate după elementul curent }
    pentru i=i+1,k execută:
      c[i] ← c[i-1] + 1
    sfârşit pentru
    Afişează
  sfârşit cât timp
sfârşit algoritm
```

7.8.3. Combinări cu repetiții generate recursiv

Algoritmul diferă de cel prezentat în introducerea din 7.8. doar în limitele structurii repetitive de tip **pentru** cu care generăm valorile posibile ale combinărilor. Subalgoritmul se apelează pentru valoarea parametrului $j = 0$. Deoarece nu trebuie să ne ferim de dubluri, în **pentru** generăm valori cuprinse între c_{i-1} şi n.

```
Subalgoritm Combinări(j):
  dacă j=k atunci
    Afişează
  altfel
    j ← j + 1
    pentru i=c[j-1],n execută:
      c[j] ← i
      Combinări(j)
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

7.9. Implementări sugerate

Pentru a vă familiariza cu modul în care se abordează problemele în care trebuie abordate probleme de combinatorică, vă sugerăm să implementați algoritmi pentru:

1. Calculul numărului de permutări/aranjamente/combinări folosind numere mari;
2. Generarea permutărilor/aranjamentelor/combinărilor folosind metoda backtracking;
3. Generarea permutărilor/aranjamentelor/combinărilor fără folosirea metodei backtracking;
4. Calcularea valorii numărului lui *Catalan*;
5. Determinarea numărului de ordine a unei permutări cu N elemente;
6. Determinarea permutării de N elemente care are numărul de ordine dat;
7. Determinarea celui de-al k-lea cuvânt (din punct de vedere lexicografic) format din anumite litere;
8. Determinarea numărului de ordine al unui cuvânt (din punct de vedere lexicografic) format din anumite litere.

7.10. Probleme propuse

7.10.1. Partiții perfecte

Se consideră o mulțime de n numere naturale. Numim *partiție perfectă* acea partiție a mulțimii în care suma elementelor din fiecare submulțime este număr prim.

Determinați partițiile perfecte ale mulțimii date.

Date de intrare
Pe prima linie a fişierului **`PARTPERF.IN`** se află un număr natural *n*, reprezentând numărul elementelor din mulţimea care trebuie partiţionată. Pe următoarea linie se află *n* numere naturale distincte, separate prin câte un spaţiu, reprezentând elementele mulţimii.

Date de ieşire
Fişierul de ieşire **`PARTPERF.OUT`** va conţine atâtea linii câte partiţii perfecte are mulţimea dată. Corespunzător unei partiţii, fiecare submulţime se va încadra între acolade, două submulţimi vor fi separate printr-un spaţiu, iar fiecare element în cadrul unei submulţimi va fi precedat şi urmat de un spaţiu.

Restricţii şi precizări
- $2 \le n \le 10$;
- elementele mulţimii sunt numere naturale mai mici decât 50;
- dacă mulţimea nu are partiţii perfecte, în fişier se va scrie `'NU'`;
- ordinea permutărilor va fi lexicografică după numerele de ordine ale elementelor în şirul dat.

Exemple

PARTPERF.IN	**PARTPERF.OUT**
`3` `30 29 15`	`NU`

PARTPERF.IN	**PARTPERF.OUT**
`5` `22 1 34 50 43`	`{ 22 1 34 50 } { 43 }` `{ 22 1 } { 34 50 43 }`

7.10.2. Fete şi băieţi
Se consideră un grup de copii format din *f* fete şi *b* băieţi. Generaţi toate subgrupurile diferite ale grupului, din fiecare fac parte exact *k* băieţi.

Date de intrare
Pe prima linie a fişierului de intrare **`GRUP.IN`** se află trei numere naturale, separate prin câte un spaţiu, reprezentând numărul fetelor, numărul băieţilor şi valoarea *k*.

Date de ieşire
Fişierul de ieşire **`GRUP.OUT`** va conţine atâtea linii câte subgrupuri se pot forma din cei *f* fete şi *b* băieţi. Fetele au numere de ordine de la 1 la *f*, băieţii de la $f + 1$ la $f + b$. Numerele de ordine ale copiilor din fiecare subgrup se vor despărţi prin câte un spaţiu.

Restricții şi precizări
- $1 \leq f, b \leq 10$;
- $1 \leq k \leq b$;
- numerele de ordine în cadrul unei submulțimi se vor afişa în ordine crescătoare;
- ordinea în care se afişează submulțimile poate fi oarecare.

Exemplu

GRUP.IN
```
2 2 1
```

GRUP.IN
```
3
4
1 3
1 2 3
2 3
1 4
1 2 4
2 4
```

7.10.3. Noroc1

Anca şi Vasile se joacă un joc de noroc. Anca a scris pe n bilețele numere distincte şi îl roagă pe Vasile să ghicească suma numerelor de pe cele m bilețele pe care acesta le va extrage la întâmplare dintre cele n. Vasile şi-a dat seama repede că îi trebuie foarte mare noroc să reuşească, aşa că a rugat-o pe Anca să îl lase să-şi noteze numerele scrise pe toate bilețele. Dar nici aşa nu îndrăzneşte să rişte. Scrieți un program care îl ajută pe Vasile să afle toate sumele posibile, urmând ca după aceea să îşi încerce norocul.

Date de intrare

Pe prima linie a fişierului de intrare **NOROC1.IN** se află două numere naturale, n şi m, reprezentând numărul total de bilețele şi numărul celor extrase. Pe următoarea linie se află n numere naturale, separate prin câte un spațiu, reprezentând numerele scrise de Anca pe bilețele.

Date de ieşire

Fişierul de ieşire **NOROC1.OUT** va avea atâtea linii câte sume distincte posibile există. Pe fiecare linie se va scrie un număr natural care reprezintă o astfel de sumă.

Restricții şi precizări
- $1 \leq n \leq 20$;
- $1 \leq m \leq 19$;
- $1 \leq bilete_i \leq 50$;
- sumele se vor afişa în ordine crescătoare în fişier.

Exemplu

NOROC1.IN	**NOROC1.OUT**

`4 2`
`10 5 20 25`
	`15`
	`30`
	`35`
	`25`
	`45`

7.10.4. Noroc2

Anca şi Vasile se joacă un joc de noroc. După ce Anca a inventat un joc la care Vasile a câştigat extrem de rar, a venit rândul lui Vasile să îşi ia revanşa. El a scris pe *n* bileţele numere distincte şi a rugat-o pe Anca să extragă un număr oarecare de bileţele astfel încât suma acestora să fie egală cu numărul pe care îl comunică el Ancăi. Vasile a numerotat bileţelele de la 1 la *n*. Scrieţi un program care o ajută pe Anca să afle care sunt bileţelele pe care trebuie să le aleagă astfel încât suma numerelor scrise pe acestea să fie egală cu numărul pe care trebuie să îl obţină.

Date de intrare

Pe prima linie a fişierului de intrare **`NOROC2.IN`** se află două numere naturale, *n* şi *suma*, reprezentând numărul bileţelelor şi numărul comunicat de Vasile. Pe următoarea linie se află *n* numere naturale, separate prin câte un spaţiu, reprezentând numerele scrise de Vasile pe bileţele.

Date de ieşire

Pe prima linie a fişierului de ieşire **`NOROC2.OUT`** se vor scrie numerele de ordine ale bileţelelor pe care Anca va trebui să le aleagă pentru a obţine ca sumă numărul comunicat de Vasile. Numerele se vor despărţi prin câte un spaţiu.

Restricţii şi precizări

- $1 \leq n \leq 100$;
- $1 \leq suma \leq 65000$;
- $1 \leq bilete_i \leq 200$;
- dacă există mai multe soluţii, se cere una singură;
- dacă nu există nici o soluţie, în fişier se va scrie `'NU'`;
- numerele de ordine se vor afişa în ordine crescătoare.

Exemplu

NOROC2.IN	**NOROC2.OUT**
`4 25` `10 5 20 15`	`1 4`

7.10.5. Litere

Gigel a învățat câteva litere din alfabet și își scrie conștiincios tema. El observă că în cuvintele pe care trebuie să le scrie apar doar literele pe care le-a învățat la școală și că fiecare cuvânt are cel mult k litere. Gigel a devenit curios și ar vrea să știe câte cuvinte de lungime maximă k se pot scrie cu cele n litere pe care le cunoaște.

Date de intrare
Pe prima linie a fișierului de intrare **`LITERE.IN`** se află două numere naturale, n și k, reprezentând numărul literelor pe care le cunoaște Gigel și lungimea celui mai lung cuvânt.

Date de ieșire
În fișierul de ieșire **`LITERE.OUT`** se va scrie numărul cuvintelor distincte care se pot scrie pe lungime de 1, 2, ..., k litere.

Restricții și precizări
- $1 \leq n \leq 26$;
- $1 \leq k \leq 10$.

Exemplu

`LITERE.IN`	**`LITERE.OUT`**	*Explicație*
`4 2`	`20`	Vor fi 4 (4^1) cuvinte scrise cu o singură literă și 16 (4^2) cuvinte scrise cu două. Dintre cele 16 cuvinte, având câte două litere, 4 vor fi scrise cu litere identice și 12 cu litere diferite.

7.10.6. Mulțimi de cifre

Se consideră cel mult trei mulțimi disjuncte, ale căror elemente sunt cifre. Generați toate numerele distincte din cifrele date, folosind la construirea unui număr o singură cifră din fiecare mulțime.

Date de intrare
Pe prima linie a fișierului de intrare **`CIFRE.IN`** se află un număr natural n, reprezentând numărul mulțimilor de cifre. Pe următoarele n linii sunt scrise elementele mulțimilor (cifre separate prin câte un spațiu).

Date de ieșire
În fișierul de ieșire **`CIFRE.OUT`** se vor scrie numerele care se pot forma pe baza cerințelor. Pe o linie se va scrie un singur număr.

Restricții şi precizări

- $1 \leq n \leq 3$;
- Ordinea în care se scriu numerele în fişier poate fi oarecare.

Exemplu

CIFRE.IN	**CIFRE.OUT**	*Explicație*
`2`	`21`	Primul număr este 21; permutând cifrele sale
`2 7`	`12`	obținem 12, îl generăm pe 71, ale cărui cifre
`1`	`71`	permutate conduc la 17.
	`17`	

7.10.7. Serată

La o serată participă b băieți şi f fete. La un moment dat dansează k perechi. Determinați toate posibilitățile în care ar putea dansa k perechi la un moment dat.

Date de intrare

Pe prima linie a fişierului de intrare **`SERATA.IN`** se află trei numere naturale b, f şi k, separate prin câte un spațiu.

Date de ieşire

Fişierul de ieşire **`SERATA.OUT`** va conține mai multe seturi de rezultate. Un set este format din k perechi de numere naturale, precedate de litera `'F'`, respectiv `'B'`, în funcție de sexul dansatorului. Băieții sunt numerotați de la 1 la b, iar fetele de la 1 la f.

Restricții şi precizări

- $2 \leq b, f \leq 7$;
- $1 \leq k \leq 4$;
- Seturile de rezultate în fişier se vor scrie în ordine lexicografică;
- Perechile de dansatori în seturile de rezultate se vor scrie în ordine lexicografică.

Exemplu

SEARATA.IN	**SERATA.OUT**
`2 2 1`	`B1 F1`
	`B1 F2`
	`B2 F1`
	`B2 F2`

7.11. Soluțiile problemelor propuse

7.11.1. Partiții perfecte

Vom rezolva problema generând toate partițiile şi afişând doar acelea care au proprietatea cerută. Din nefericire, nu avem nici o posibilitate de a întrerupe generările înainte de obținerea lor. Singura optimizare constă în generarea numerelor prime cu ciurul lui *Erastostene* şi păstrarea unui şir *nu_e_tăiat* de tip `Boolean` în care valoarea $nu\_e\_tăiat_i$ = *adevărat* semnifică faptul că *i* este număr prim. Evident, s-ar fi putut verifica de fiecare dată fiecare sumă în parte dacă este prim sau nu, dar dacă numărul elementelor este relativ mare, vom avea şi partiții mai multe, deci şi numere (sume) mai multe de verificat.

```
Subalgoritm Ciur(nu_e_tăiat):
  pentru i=2,Max execută:   { în Max avem o constantă reprezentând cel mai }
         { mare număr prim care poate să apară ca sumă de partiții: 50*10 = 500 }
    nu_e_tăiat[i] ← adevărat             { scriem toate numerele pe papirus }
  sfârşit pentru
  pentru i=2,Max-1 execută:
    dacă nu_e_tăiat[i] atunci
      j ← i + i                           { primul multiplu al numărului i }
      cât timp j ≤ Max execută:            { tăiem multiplii numărului i }
        nu_e_tăiat[j] ← fals
        j ← j + i                         { următorul multiplu al numărului i }
      sfârşit cât timp
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Pentru generarea partițiilor de mulțimi putem folosi următorul subalgoritm, diferit de cel prezentat în 7.3.:

```
Subalgoritm Part(i,k):                             { avem două posibilități }
  pentru j=1,k execută:                             { avem deja k submulțimi }
    m[i] ← j        { îl punem pe i într-o submulțime existentă de indice j }
    dacă i < n atunci
      Part(i+1,k)
    altfel
      Verif(k)
    sfârşit dacă
  sfârşit pentru
  m[i] ← k+1          { îl punem pe i în submulțimea nouă de indice k+1 }
  dacă i < n atunci
    Part(i+1,k+1)
```

```
  altfel
    Verif(k+1)
  sfârşit dacă
sfârşit subalgoritm
```

Acest subalgoritm apelează subalgoritmul de verificare `Verif(k)` a partiției generate în care avem *k* submulțimi. Vom calcula sumele elementelor lor şi vom verifica dacă sunt prime sau nu:

```
Subalgoritm Verif(k):
  pentru i=1,k execută:                    { în partiție avem k submulțimi }
    suma ← 0
    pentru j=1,n execută:
      dacă m[j] = i atunci          { elementele din fiecare a i-a submulțime }
        suma ← suma + a[j]
      sfârşit dacă
    sfârşit pentru
    dacă nu nu_e_tăiat[suma] atunci        { dacă suma nu este număr prim }
      Ieşire forțată din subalgoritm { nu are rost să verificăm celelalte partiții }
    sfârşit dacă
  sfârşit pentru
  Afişează(k)     { dacă am ajuns aici, partiția este perfectă şi se poate afişa }
sfârşit subalgoritm
```

7.11.2. Fete şi băieți

Trebuie să generăm toate subgrupurile diferite ale grupului de *f* fete şi *b* băieți, astfel încât din fiecare să facă parte exact *k* băieți. Observăm că nu trebuie să obținem o partiționare a grupului, ci submulțimi diferite din care să facă parte exact *k* băieți. Rezultă că vom genera mai întâi submulțimi de *k* băieți, apoi fiecărei astfel de submulțimi îi vom adăuga pe rând toate submulțimile fetelor.

În implementare vom lucra cu tipul **`set`** (în Pascal), deoarece astfel uşurăm operația de unificare a grupului de băieți selectat cu grupul de fete. În plus, la afişare vom putea să scriem numerele de ordine în ordine crescătoare fără să mai fie nevoie de ordonarea acestora. După generarea fiecărei submulțimi de băieți, respectiv de fete, creăm mulțimea propriu-zisă a lor. În final vom avea *i* submulțimi de fete şi *j* submulțimi de băieți, pe care le vom combina (pe fiecare cu fiecare) cu următorul subalgoritm:

```
Subalgoritm Uniune(băieți,fete,i,j,g):
  pentru jj=1,j execută:               { j = numărul submulțimilor de băieți }
                 { grupul curent g se inițializează cu a jj-a submulțime de băieți }
    g ← băieți[jj]
```

```
    pentru ii=1,i execută:                { i = numărul submulţimilor de fete }
      g ← g ∪ fete[ii]                  { se adaugă a ii-a submulţime de fete }
      Afişează(g)
              { grupul curent g se reiniţializează cu a jj-a submulţime de băieţi }
      g ← băieţi[jj]
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

7.11.3. Noroc1

Va trebui să calculăm suma elementelor submulţimilor având *m* elemente, generate pentru mulţimea celor *n* numere date. În plus, fiecare sumă se va afişa o singură dată. Cel mai uşor am putea scăpa de dubluri dacă am lucra cu tipul **set** (în Pascal), dar din nefericire avem cel mult 19 bileţele extrase, pe care pot fi numere cel mult egale cu 50, dar care sunt distincte, ale căror sumă este 779, deci nu ne este util tipul respectiv. În schimb, vom putea declara un şir de valori logice. În momentul în care am calculat o sumă, elementul corespunzător în şirul de valori booleene va deveni *adevărat*. În final, pe baza acestui şir vom afişa sumele în ordine crescătoare.

7.11.4. Noroc2

Vom genera submulţimi în ordine lexicografică şi de fiecare dată, vom scădea din valoarea sumei date valoarea numărului adăugat în submulţime. Dacă, pe parcursul acestui proces, valoarea rămasă din sumă devine 0, înseamnă că avem o soluţie pe care o scriem în fişierul de ieşire şi oprim programul. Dacă niciodată nu ajungem în această situaţie, înseamnă că suma nu poate fi acoperită prin adunarea unor numere date şi vom scrie în fişier mesajul cerut. În subalgoritmul de generare a submulţimilor de sumă dată am notat cu *s* şirul indicilor bileţelelor din submulţimea curentă. Avem nevoie şi de un element s_0, deoarece s_1 se calculează cu ajutorul acestuia.

```
Subalgoritm Submulţimi(k,suma):
  dacă suma = 0 atunci
    Afişează(k-1)                     { nu mai este nevoie de al k-lea termen }
  altfel
    pentru val=s[k-1]+1,n execută:
                  { dacă în sumă „încape” valoarea bileţelului de indice val }
      dacă suma ≥ bilete[val] atunci
          s[k] ← val              { adăugăm submulţimii indicele bileţelului }
          Submulţimi(k+1,suma-bilete[val])
      sfârşit dacă
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

7.11.5. Litere

Trebuie să calculăm numărul aranjamentelor cu repetiții a *n* litere luate câte $i = 1, ..., k$. Numărul cerut va fi o sumă, unde fiecare termen este egal cu n^i, unde $i = 1, ..., k$.

7.11.6. Mulțimi de cifre

Mulțimile de cifre le vom păstra într-un tablou bidimensional în care a *i*-a linie va conține cifrele celei de-a *i*-a mulțimi. Vom forma şiruri de cifre luând prima cifră din prima mulțime, lângă care punem prima cifră din celelalte două. Apoi, vom determina permutările şirului de cifre obținut. La pasul următor vom înlocui cifra din cea de a treia mulțime cu următoarea din această mulțime, dacă o astfel de cifră există. Din nou vom permuta şi acest şir. Vom continua procesul până când am epuizat toate cifrele din prima mulțime.

În algoritmul care urmează în pseudocod am notat cu *m* tabloul mulțimilor de cifre şi cu *nr* tabloul lungimilor liniilor (numărul elementelor din fiecare mulțime). Citirea datelor se realizează cu subalgoritmul următor:

```
Subalgoritm Citire(m,nr,n):
  citeşte n                                    { numărul mulțimilor }
  pentru i=1,n execută:
    j ← 0
    cât timp nu urmează marca de sfârşit de linie execută:
      j ← j + 1
      citeşte m[i,j]
    sfârşit cât timp
    citeşte marca de sfârşit de linie
    nr[i] ← j
  sfârşit pentru
sfârşit subalgoritm
```

Permutarea o realizăm cu permutări circulare. În algoritm şirul *p* este cel care trebuie permutat, iar *pp* este şirul indicilor.

```
Subalgoritm Perm(p,n):
  poz ← 1
  pentru i=1,3 execută:
    pp[i] ← i
  sfârşit pentru
  repetă
    aux ← pp[poz]
    pentru i=poz,n-1 execută:
      pp[i] ← pp[i+1]
    sfârşit pentru
```

```
    pp[n] ← aux
    dacă pp[poz] = poz atunci
      poz ← poz + 1
    altfel
      poz ← 1
      pentru i=1,n execută:
        scrie p[pp[i]]
      sfârşit pentru
    sfârşit dacă
  până când poz = n
sfârşit subalgoritm
```

Având în vedere că avem cel mult trei mulțimi, în algoritmul principal, putem să luăm pe rând liniile tabloului m şi să formăm şirurile de câte trei cifre astfel:

```
Algoritm Mulțimi_de_cifre:
  Citire(m,nr,n)
  pentru i1=1,nr[1] execută:          { în prima mulțime avem nr1 elemente }
    i ← 1
    p[i] ← m[i,i1]
    dacă nr[2] ≠ 0 atunci            { dacă avem o singură mulțime, nr2 = 0 }
      pentru i2=1,nr[2] execută:    { în a doua mulțime avem nr2 elemente }
        i ← 2
        p[i] ← m[i,i2]
        dacă nr[3] ≠ 0 atunci          { dacă avem două mulțimi, nr3 = 0 }
          pentru i3=1,nr[3] execută:
            i ← 3
            p[i] ← m[i,i3]
            pentru i4=1,i execută:
              scrie p[i4]
            sfârşit pentru
            Perm(p,i)
          sfârşit pentru
        altfel
          pentru i4=1,i execută:
            scrie p[i4]
          sfârşit pentru
          Perm(p,i)
        sfârşit dacă
      sfârşit pentru
    altfel
      pentru i4=1,i execută:
        scrie p[i]
      sfârşit pentru
```

```
      Perm(p,i)
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

7.11.7. Serata

Deoarece la un moment dat dansează doar k perechi şi k este cel mult egal cu n umărul băieţilor (şi/sau fetelor), pentru început vom genera toate combinările de *b* băieţi luate câte *k*. Am putea în mod similar genera şi combinările de câte *k* fete şi ulterior am putea forma perechile de dansatori folosind grupurile de băieţi şi fete. Observăm însă, că din oricare asemenea două grupuri de băieţi respectiv fete se pot forma un număr mare de perechi posibile: fiecare băiat dintr-o grupă poate să formeze o pereche cu fiecare fată din grupul considerat de fete. Astfel, pentru a simplifica problema, putem să generăm de la început aranjamente de *f* fete luate câte *k* fete, şi astfel printr-o simplă combinare a fiecărui grup de băieţi cu fiecare grup de fete obţinem toate combinaţiile pentru soluţia finală.

În algoritm notăm cu *nrbăieţi* numărul combinărilor de *b* băieţi luate câte *k* şi cu *nrfete* numărul aranjamentelor de *f* fete luate câte *k*. În tabloul *băieţi* reţinem toate combinările de *b* luate câte *k* băieţi, iar în *fete* toate aranjamentele de *f* luate câte *k* fete.

Combinările de băieţi le generăm cu următorul subalgoritm:

```
Subalgoritm Combinări_Băieţi(i):
  pentru val=v[i-1]+1,n execută:
    v[i] ← val                          { în şirul v generăm combinarea }
    dacă i < k atunci
      Combinări_Băieţi(i+1)
    altfel
      nrbăieţi ← nrbăieţi + 1           { avem o combinare nouă de k băieţi }
      băieţi[nrbăieţi] ← v              { reţinem combinarea }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Aranjamentele de fete le generăm cu următorul subalgoritm:

```
Subalgoritm Aranjamente_Fete(i):
  pentru val=1,m execută:
    dacă nu folosit[val] atunci
      v[i] ← val                        { în şirul v generăm aranjamentul }
      folosit[val] ← adevărat
      dacă i < k atunci
        Aranjamente_Fete(i+1)
```

```
      altfel
        nrfete ← nrfete + 1          { avem un aranjament nou de k fete }
        fete[nrfete] ← v             { reținem aranjamentul }
      sfârşit dacă
      folosit[val] ← fals
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Combinările de băieți le punem în pereche cu aranjamentele de fete cu următorul subalgoritm:

```
Subalgoritm Afişează:
  pentru i=1,nrbăieți execută:       { fiecare combinare de băieți }
    pentru j=1,nrfete execută:       { cu fiecare aranjament de fete }
      pentru l=1,k execută:          { avem câte k elemente }
        scrie 'B',băieți[i][l],' F',fete[j][l]
      sfârşit pentru
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```