Generarea submulțimilor unei mulțimi

Materialul de față își propune a arăta modul de obținere a tuturor submulțimilor unei mulțimi fără a folosi algoritmi standard backtracking. Nu se utilizează recursivitatea, materialul fiind util elevilor care sunt încă la începutul studierii algoritmilor și vine în sprijinul lor pentru a putea rezolva totuși o anume categorie de probleme care încercă generarea mai multor șiruri pentru a obține soluția.

Fie o mulțime cu n elemente, să o notăm: {1, 2, 3, …, n}. Aceasta admite 2^n submulțimi. De exemplu, pentru $n=3$, acestea sunt: Ø, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}.

Cunoaștem totodată că folosind n poziții binare putem genera de asemenea 2^n configurații de 0 și 1, de lungime n. Aceste configurații corespund scrierilor în baza 2 pentru numerele naturale cuprinse între 0 și 2^n-1.

Pentru $n=3$, aceste configurații sunt: 000, 001, 010, 011, 100, 101, 110, 111.

Sunt cunoscuți mulți algoritmi pentru generarea secvențelor binare, iar noi le vom genera și vom arăta modul în care unei secvențe binare îi asociem exact o submulțime.

Pentru a exemplifica asta, considerăm $n=4$

Numerele naturale dintre 0 și 2^n-1	**Scrierea binară a numărului**	**Pozițiile unde sunt valori 1 în scrierea binară**	**Submulțimea asociată**
0	0000	Niciuna	Ø
1	0001	4	{4}
2	0010	3	{3}
3	0011	3,4	{3,4}
4	0100	2	{2}
5	0101	2,4	{2,4}
6	0110	2,3	{2,3}
7	0111	2,3,4	{2,3,4}
8	1000	1	{1}
9	1001	1,4	{1,4}
10	1010	1,3	{1,3}
11	1011	1,3,4	{1,3,4}
12	1100	1,2	{1,2}
13	1101	1,2,4	{1,2,4}
14	1110	1,2,3	{1,2,3}
15	1111	1,2,3,4	{1,2,3,4}

Așadar, dacă generăm toate secvențele din coloana a 2-a, putem apoi obține ușor submulțimile. În multe situații nu este importantă ordinea în care se generează submulțimile, așa cum vedeți și mai sus, ele nefiind neapărat în ordine lexicografică, sau cumva. Există evident algoritmi care generează lexicografic, dar nu ne interesează în acest material.

1. Simularea adunarii cu 1 în baza 2, având cifrele binare puse într-un vector pe poziții de la 1 la n.

Observația pe care se bazează totul este următoarea: dacă avem generată o secvență binară, pentru următoarea, se procedează astfel:

- Valorile 1 de la final se transformă în 0.
- Primul 0 întâlnit (din dreapta) se face 1.
- Valorile de pe restul pozițiilor rămân neschimbate:

Exemplu:

1010100111	Aceasta o secvență construită deja, adică este configurația curentă a vectorului
1010100111	Sunt trei valori 1 la final, care se vor transforma în 0, și pe a 4-a poziție de la final este prima valoare de 0 (privind din dreapta) care se va transforma în 1.
1010101000	Celelalte valori rămân nemodificate, așadar am obținut următoatea configurație.

Scrierea în baza 10 a configurației inițiale dă valoarea 679, iar scrierea celei obținute dă 680. Deci am obținut scrierea numărului următor.
v fiind tabloul unidimensional ce memorează configurația curentă pe poziții de la 1 la n, codul care face transformarea este următorul:

```
i = n;
while (v[i] == 1) {
    v[i] = 0;
    i--;
}
v[i] = 1;
```

Așadar, poziția pe care este i la finalul repetiției este chiar cea unde se află cel mai din dreapta 0.

Vom porni de la configurația formată doar din 0, și ultima pe care o vom genera este cea formată doar din 1. Puteți exersa și verifica regula de transformare pentru tabelul de mai sus observând cum ea duce configurația de pe o linie în cea de pe linia următoare.

Pentru a genera toate submulțimile, rulăm în mod repetat configurația de mai sus:

```
while (…) {
    i = n;
    while (v[i] == 1) {
        v[i] = 0;
        i--;
    }
    v[i] = 1;

    for (i=1;i<=n;i++)
        if (v[i] == 1)
            fout<<i<<" ";
    fout<<"\n";
}
```

Așadar, după ce generăm noua configurație, parcurgem vectorul și acolo unde avem valoarea 1 afișam indicele.

Observați că nu am scris condiția de oprire. O prima variantă pentru aceasta este de a folosi un contor și să facem exact 2^n treceri dintr-o secvență în alta.

O altă variantă ar fi să testăm dacă s-a ajuns la configurația formată doar din valori `1`. Aceasta ar necesita cod suplimentar pentru acest test.

O metodă de oprire mai elegantă este următoarea: observăm că dacă aplicăm algoritmul de trecere în secvența următoare când suntem deja în cea formată doar din `1`, nu găsim `0` decât eventual pe poziția `0`, deci în afara intervalului de indici care ne interesează. Așadar, dacă punem de la început și `v[0] = 0`, la poziția `0` vom face `1` abia la final, așadar, condiția de oprire va fi, `v[0] != 1`, nefiind astfel nevoie nici să scriem și nici să execute calculatorul repetiții suplimentare.

Mai jos este soluția de `100` de puncte de la problema de generare a submulțimilor de pe infoarena (https://infoarena.ro/problema/submultimi).

Enunțul pe scurt este următorul:

Se citește un număr `n`, și se cer tote submulțimile nevide ale mulțimii {`1`, `2`, `3`, ..., `n`}.

```
#include <fstream>
using namespace std;
ifstream fin("submultimi.in");
ofstream
fout("submultimi.out");
int v[20];
int n, i;
int main() {
    fin>>n;
    for (i=0;i<=n;i++)
        v[0] = 0;
    while (!v[0]) {
        i = n;
        while (v[i] == 1) {
            v[i] = 0;
            i--;
        }
        v[i] = 1;

        if (i == 0)
            break;

        for (i=1;i<=n;i++)
            if (v[i] == 1)
                fout<<i<<" ";
        fout<<"\n";
    }
    return 0;
}
```

Mai sus întâlnim și secvența `if(i==0) break;` care ar face să se oprească oricum bucla principală. Rolul ei aici este de a evita afișarea unui rând suplimentar liber, la final, corespunzător mulțimii vide. Afișarea aceasteia s-ar încerca la final, când secvența `111...11` s-a transformat în `1`(pe poziția `0`)`000...00`. (Chiar

dacă s-a pornit cu vectorul plin de 0, mai întâi se face prima transformare, deci prima secvență pentru care se tipărește submulțimea corespunzătoare este 000...01.

Numărul de pași pe care îi face algoritmul descris este de ordin $n \cdot 2^n$ (avem 2^n transformări și fiecare necesită traversare a șirului de n valori).

Algritmul este cu aplicabilitate practică, asigurând timp instant de rezolvare, pentru valori ale lui n până în jurul valorii 20 (2^{20} este aproximativ un milion).

2. Scrierea efectivă în baza 2 a numerelor de la 0 la 2^n-1.

```
#include <fstream>
using namespace std;
ifstream fin("submultimi.in");
ofstream fout("submultimi.out");
int v[20];
int n, i, p, cod, aux;
int main() {
    fin>>n;

    p = 1; /// calculez 2 la puterea n
    for (i=1;i<=n;i++)
        p = p*2;

    cod = 1; /// Scriu in baza 2 numerele incepand cu 1.
             /// Sarind de 0 evit multimea vida.
    while (cod < p) {
        aux = cod;
        i = n;
        while (aux) {
            if (aux % 2 == 1)
                fout<<i<<" ";
            i--;
            aux /= 2;
        }
        fout<<"\n";

        cod++;
    }
    return 0;
}
```

Probleme rezolvate.

1. Problema care pe infoarena se numeste **numere8** (https://infoarena.ro/problema/numere8). A fost dată la olimpiada județeană de informatică.

Enunțul, pe scurt, este următorul: Se dă un șir format din exact 10 numere, distincte. Se cere să determinăm:

- În câte moduri putem alege două submulțimi disjuncte de sumă egală (pot rămâne și elemente în afara celor două submulțimi).

- Determinați suma maximă comună pentru care există două submulțimi ca mai sus și două astfel de submulțimi.

Exemplu:

numere8.in	numere8.out
60 49 86 78 23 97 69 71 32 10	65 276 78 97 69 32 60 49 86 71 10

Sunt 65 de variante de a împărți cele 10 numere date în două submulțimi de sumă egală. Dacă formăm una dintre submulțimi cu numerele de pa a doua linie și pe cealaltă din numerele de pe a treia linie, obținem una dintre cele 65 de soluții și suma maximă comună pentru aceste submulțimi (276). Observăm că unul dintre cele 10 numere nu apare în niciuna dintre cele două submulțimi. Nu se numără și numerotarea inversă a celor două submulțimi. De asemenea, nu obținem submulțimi noi permutând aceleași elemente.

Soluție.

Dacă am genera toate configurațiile de 0 și 1 de lungime 10 și am considera pentru o submulțime elementele din dreptul valorilor 0 iar pentru cealaltă submulțime elementele din dreptul valorilor 1, am rata cazurile când rămân și elemente în fara celor două submulțimi. Modalitatea prin care putem lăsa și elemente în afară este următoarea: în loc să generăm numere în baza 2, vom genera numere în baza 3 (adică secvențe de lungime 10 formate din 0, 1, 2). În acest mod, elementele din dreptul valorilor 1 considerăm că sunt într-o submulțime, cele din dreptul valorilor 2 sunt în cealaltă submulțime iar cele din dreptul valorilor 0 sunt nealese la selectarea curentă de submulțimi.

Generarea tuturor secvențelor cu 0, 1, 2 se face asemănător cu cele binare. Ca diferență, la trecerea de la o secvență la alta:

- Valorile 2 de la final devin 0.
- Prima valoare diferită de 2 întâlnită crește cu 1.
- Valorile de pe celelalte poziții rămân nemodificate.

În general, dacă am genera următorul număr într-o bază B:

- Valorile B-1 de la final devin 0.
- Prima valoare diferită de B-1, întâlnită, crește cu 1.
- Valorile de pe celelalte poziții rămân nemodificate.

Așadar:

```
#include <fstream>
using namespace std;
ifstream fin("numere8.in");
ofstream fout("numere8.out");
int v[11], w[11], a[11];
int s1, s2, i, total, maxim;
int main() {
    for (i=1;i<=10;i++)
        fin>>a[i];

    while (v[0]==0) {
```

```
        i=10;
        while (v[i] == 2) {
            v[i] = 0;
            i--;
        }
        v[i]++;

        if (i==0)
            break;

        s1 = s2 = 0;
        for (i=1;i<=10;i++) {
            if (v[i] == 1)
                s1 += a[i];
            if (v[i] == 2)
                s2 +=a[i];
        }
        if (s1 == s2) {
            total++;

            if (s1 > maxim) {
                maxim = s1;
                for (i=1;i<=10;i++)
                    w[i] = v[i];
            }

        }
    }
    fout<<total/2<<" "<<maxim<<"\n";
    for (i=1;i<=10;i++)
        if (w[i] == 1)
            fout<<a[i]<<" ";
    fout<<"\n";
    for (i=1;i<=10;i++)
        if (w[i] == 2)
            fout<<a[i]<<" ";
    return 0;
}
```

Câteva observații:

- Am afișat valoarea total înjumătățită întrucât se numără de două ori soluții: de exemplu, dacă se generează `1 0 0 0 0 0 2 1 2 2` se obține aceeași soluție cu `2` și `1` schimbate, adică `2 0 0 0 0 0 1 2 1 1`.
- Întrucât valorile date sunt strict pozitive, nu am ocolit soluțiile în care se generează numai `0` și `1` sau pe cele în care se generează numai `0` și `2` deoarece una dintre submulțimi ar avea suma `0` iar cealaltă suma struct pozitivă, așa că nu s-ar număra.
- Am ocolit cazul cu soluția formată doar din `0` pentru că s-ar număra și soluția cu cele două submulțimi vide.

Timpul de calcul este așadar de ordinul $3^n \cdot n$.

2. Jocul **Flip**. Problema se găsește pe infoarena.ro (https://infoarena.ro/problema/flip). Enunțul pe scurt este următorul:

Se dă o matrice cu `n` linii și cu `m` coloane, cu valori întregi. Avem posibilitatea de a schimba semnul pentru toate valorile de pe unele linii și pentru valorile de pe unele coloane (alegem noi pe care). Ne propunem să aplicăm operațiile încât matricea obținută să aibă suma maximă. Dacă schimbăm semnul și pentru linia `i` și pentru coloana `j`, valoarea din poziția `i,j` își schimbă semnul de două ori, deci rămâne nemodificată. Dimensiunile matricei au valoarea maxim `16`.

Exemplu

flip.in	flip.out
5 3 4 -2 2 3 -1 5 2 0 -3 4 1 -3 5 -3 2	28

Se aplică operații pentru coloana a doua și pentru linia a treia.

Soluție

Observăm că putem încerca toate modurile de a alege coloanele la care operăm. Numărul de coloane fiind maxim `16`, avem până la 2^{16} variante de a alege cum interschimbăm coloanele. De exemplu, dacă avem `m = 4` coloane iar șirul în care generăm submulțimile este `1 0 1 1`, considerăm cazul în care schimbăm semnul pentru coloanele `1`, `3`, `4`. Pentru fiecare astfel de încercare, observăm că putem decide exact ce facem cu fiecare linie: cele cu suma pozitivă rămân neschimbate, cele cu suma negativă se schimbă (luăm în calcul la soluție modulul sumelor pentru fiecare linie).

```
#include <fstream>
#define DIM 17

using namespace std;
int a[DIM][DIM], v[DIM];
int n, m, i, j, s, sum ,maxim;

int main(){
    ifstream fin("flip.in");
    ofstream fout("flip.out");

    fin>>n>>m;

    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
            fin>>a[i][j];

    while (v[1] == 0) {
        j = m;
        while (v[j] == 1) {
```

```
                v[j] = 0;
                j -- ;
            }
            v[j] = 1;

            sum = 0;
            for (i=1;i<=n;i++) {
                s = 0;
                for (j=1;j<=m;j++)
                    if (v[j] == 0)
                        s += a[i][j];
                    else
                        s -= a[i][j];
                if (s > 0)
                    sum += s;
                else
                    sum -= s;
            }
            if (sum > maxim)
                maxim = sum;
        }
        fout<<maxim<<"\n";
        return 0;
    }
```

Obținem așadar timp de calcul de ordin $2^m \cdot n^2$. Întrucât și numărul de linii era mic, am fi putut face și invers, să generăm toate modurile de a aplica operația liniilor și pentru fiecare era unic determinat modul de a aplica operația la coloane.