# Vectori de frecvență

Introducem acest capitol pornind de la următoarea problemă: se dă un șir cu până la 1000000 de numere, care pot fi 1, 2 sau 3 și se cere să spunem pentru fiecare dintre ele de câte ori apare. Soluția presupune folosirea a trei variabile cu semnificație de contor, câte una pentru fiecare valoare posibilă.

```
cin>>n;
for (i=1;i<=n;i++) {
    cin>>x;
    if (x == 1)
        f1++;
    else
        if (x == 2)
            f2++;
        else
            f3++;
}
cout<<"1 "<<f1<<"\n";
cout<<"2 "<<f2<<"\n";
cout<<"3 "<<f3<<"\n";
```

Acum să generalizăm. Presupunem că suntem în aceleași restricții, singura modificare fiind că valorile pot fi între 1 și 100. Evident că vom avea nevoie de 100 de variabile contor, câte una pentru fiecare valoare posibilă. Dacă ne gândim să adaptăm codul de mai sus, întâmpinăm următoarele probleme: cum numim cele 100 de variabile contor ? Cum scriem afișările de la final? Aceste probleme s-ar putea amplifica dacă ne gândim că programul poate avea ca parametru valoarea maximă care poate să apară, deci nu neapărat fix 100, sau dacă această valoare ar fi și mai mare.

Ce ar fi să numim variabilele f[1], f[2], ... f[100] ? Cu alte cuvinte vom folosi un vector cu atâtea componente câte valori posibile pot fi în șirul de la intrare. Acum, la citirea unei valori x este suficient să incrementăm contorul cu o instrucțiune de forma f[x]++. Scăpăm astfel și de if-uri.

În etapa de afișare trebuie să vizităm fiecare contor și să tipărim atât poziția lui (care reprezintă o valoare posibilă la intrare) cât și valoarea de la acea poziție (care reprezintă numărul de apariții).

```
for (i=1;i<=100;i++)
    cout<<i<<” ”<<f[i]<<”\n”;
```

Iată programul complet:

```
#include <iostream>
using namespace std;
int n, i, x, f[101];
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>x;
        f[x]++;
    }
    for (i=1;i<=100;i++)
        cout<<i<<" "<<f[i]<<"\n";
    return 0;
```

```
}
```

**Observații**

- Mulți elevi au tendința de a scrie al doilea for până la `n`. Evident că nu. Semnificația lui este de a traversa cu un indice toate pozițiile posibile din `f`, care corespund deci tuturor valorilor posibile din fișierul de intrare, între `1` și `100` în cazul problemei noastre.
- Timpul de calcul al codului de mai sus este de ordinul `n` (numărul de elemente de la intrare) + `val_max` (cea mai mare valoare posibilă la intrare, în cazul nostru `100`).
- Memoria necesară nu este de ordin `n`, observăm că pe noi nu ne interesează neapărat să memorăm numerele de la intrare, memoria necesară este de ordin `val_max`.
- Se observă un timp de calcul liniar în parametrii care apar. Metoda este una care oferă o optimizare foarte bună în multe situații.
- Pentru a aplica această tehnică trebuie însă ca valorile posibile să nu fie într-un interval foarte mare (să poată fi alocate `val_max` componente). Nu doar memoria ar fi afectată de un `val_max` mare ci și timpul de executare întrucât "concluziile" se trag prin repetiția de la final care face număr de pași de ordinul `val_max`. De asemenea, valorile care apar în fișier vor fi folosite ca indici în vector, deci ele ar trebui să fie naturale. În unele cazuri se poate folosi tehnica și pentru valori negative sau chiar reale, însă cu artificii speciale care vor fi utilizate în problemele rezolvate din acest capitol.
- De multe ori elevii întreabă: cât ar trebui să fie maxim `val_max` pentru a putea folosi tehnica? Nu se poate da un răspuns exact, asta depinde de și performanțele calculatorului. Dar putem face următoarea estimare: calculatoarele actuale fac câteva milioane de instrucțiuni pe secundă. De asemenea, câteva milioane de octeți înseamnă câțiva mega de memorie. Deci recomandarea este să nu ne îndepărtăm mult în sus cu limita de `1000000` pentru `val_max`.

Folosirea unui vector ca în programul de mai sus, unde valoarea unei componente reprezintă o informație despre indice, face să numim structura *vector de frecvență*. Din punct de vedere al limbajului, vectorul de frecvența este un tablou unidimensional ca oricare altul, doar semnificația pe care o dăm componentelor și modul de a interpreta valoarea lor îl face cumva special în cod.

În programul de mai sus am sărit o etapă: nu am inițializat cu `0` componentele tabloului. Acest lucru este esențial, dar noi, pentru a păstra codul compact, ne-am bazat pe faptul că datele globale sunt inițializate implicit cu `0`.

## Probleme rezolvate

*Aplicațiile următoare nu sunt neapărat probleme oarecare, ele strâng împreună cerințele clasice care se rezolvă folosind vectorii de frecvență.*

1. Din fișierul `unice.in` se citesc numărul `n` (maxim `100000`) și apoi n numere naturale de maxim două cifre. Să se scrie în fișierul `unice.out`, în ordine crescătoare, numerele care apar o singură dată (*pbinfo.ro, #267*).

**Rezolvare**

```
#include <fstream>
using namespace std;
int f[100];
int n, i, x;
int main () {
    ifstream fin ("unice.in");
    ofstream fout("unice.out");
    fin>>n;
    for (i=1;i<=n;i++) {
        fin>>x;
```

```
        f[x] ++;
    }
    for (i=0;i<=99;i++)
        if (f[i] == 1)
            fout<<i<<" ";
    return 0;
}
```

Timpul de calcul este de ordin `n`. Am neglijat `val_max` pentru că aici este `100`, deci o valoare foarte mică.

Merită să prezentăm și soluția alternativă, fără vectori de frecvență, pentru a scoate în evidență mai bine atât eleganța cât și eficiența celei de mai sus.

```
#include <fstream>
using namespace std;
int n, i, j, x, aux, v[100010];
ifstream fin ("unice.in");
ofstream fout("unice.out");
int main () {
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i];
    for (i=1;i<=n;i++)
        for (j=i+1;j<=n;j++)
            if (v[i] > v[j]) {
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
    /// facem ca in fata primului element si dupa ultimul
    /// sa fie valori diferite de vecini pentru a nu avea de tratat
    /// cazuri particulare la identificarea elementelor distincte
    /// cand este vorba de analizarea primului si a ultimului element
    v[0] = v[1]-1;
    v[n+1] = v[n] + 1;

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

Mai sus am folosit memorie de ordin `n` iar timpul de executare este dat de sortare. Aici am folosit un algoritm cu timp de calcul de ordin $n^2$.

2. Se dau cel mult `100000` de numere naturale, cu cel mult `2` cifre fiecare. Afişaţi în ordine strict crescătoare valorile impare care se regăsesc printre valorile date, şi în ordine strict descrescătoare valorile pare care se regăsesc printre valorile date (pbinfo.ro, #276).

**Rezolvare**

Valorile din șirul dat fiind de maxim două cifre și despre fiecare având nevoie să știm dacă apare sau nu, vom folosi un vector de frecvență de `100` de componente în care vom seta la `1` în dreptul pozițiilor ce corespund numerelor care sunt în fișier. Parcurgem acum indicii de la `1` la `100` de două ori, întâi pe cei impari crescător, apoi pe cei pari descrescător, pentru a afișa conform cerinței.

```
#include <fstream>
```

```
using namespace std;
ifstream fin ("pareimpare.in");
ofstream fout("pareimpare.out");
int f[100];
int n, i, x;
int main () {
    while (fin>>x) {
        f[x] = 1;
    }
    for (i=1;i<100;i+=2)
        if(f[i]==1)
            fout<<i<<" ";
    fout<<"\n";
    for (i=98;i>=0;i-=2)
        if (f[i]== 1)
            fout<<i<<" ";
    return 0;
}
```

Timpul de calcul este de ordin `n`.

3. Se citește un număr natural. Afișați fiecare cifră care apare, alături de numărul său de apariții.

**Rezolvare**

O soluție este să punem cifrele într-un vector, să îl sortăm și să identificăm secvențele de elemente egale. Valoarea care se repetă în secvență este cifra iar lungimea secvenței este numărul de apariții.

Soluția mai elegantă folosește un vector cu `10` componente, fiecare fiind contor pentru câte o cifră, incrementând componenta corespunzătoare la întâlnirea fiecărei cifre în timpul parcurgerii cifrelor unui număr cu algoritmul standard.

```
#include <iostream>
using namespace std;
int n, i, f[10];
int main () {
    cin>>n;
    while (n) {
        f[n%10]++;
        n /= 10;
    }
    for (i=0;i<=9;i++)
        if (f[i] != 0)
            cout<<i<<" "<<f[i]<<"\n";
    return 0;
}
```

4. Fișierul de intrare conține cel mult `1000000` de numere întregi. Se cere să se afișeze în fișierul de ieșire cel mai mic număr din intervalul [`-100`,`100`] care nu apare în fișierul de intrare. Dacă nu există un astfel de număr se va afișa mesajul `nu exista`. Fișierul de intrare `nrlipsa2.in` conține cel mult `1000000` de numere întregi, separate prin câte un spațiu, dispuse pe mai multe linii. Numerele din fișier sunt cuprinse între `-1000000` și `1000000` (*pbinfo.ro, #1744*).

**Rezolvare**

Presupunând că valorile ar fi toate pozitive am putea marca într-un vector de frecvență pe acelea care apar, apoi parcurgem vectorul de frecvență de la cel mai mic indice posibil și ne oprim când găsim prima poziție nemarcată. Faptul că sunt și valori negative pare să ne încurce pentru că nu putem avea indici negativi într-un

vector (valorile din fișier sunt indici în vectorul de frecvență). Putem rezolva în mai multe moduri această problemă iar noi vom prezenta aici două dintre ele:

Varianta 1.

Folosim doi vectori de frecvență de câte `100` de componente fiecare, unul în care marcăm pentru valorile pozitive și celălalt pentru modulele valorilor negative care apar.

```
#include <fstream>
using namespace std;
ifstream fin ("nrlipsa2.in");
ofstream fout("nrlipsa2.out");
int Plus[101];
int Minus[101];
int x, i;
int main () {
    while (fin>>x) {
        if (x >= 0 && x <= 100)
            Plus[x] = 1;
        if (x >= -100 && x < 0)
            Minus[-x] = 1;
    }
    for (i=100;i>=1;i--)
        if (Minus[i] == 0) {
            fout<<-i;
            return 0;
        }
    for (i=0;i<=100;i++)
        if (Plus[i] == 0) {
            fout<<i;
            return 0;
        }
    fout<<"nu exista";
    return 0;
}
```

Varianta 2. Un singur vector de frecvență, de dimensiune dublă față de `val_max` (la noi `val_max = 100`), dar în care la citirea valorii `x` memorăm despre ea informații la poziția `x+100` și apoi, la interogare, la fel.

```
#include<fstream>
using namespace std;
int f[210];
int x, n, i;
int main () {
    ifstream fin("nrlipsa2.in");
    ofstream fout("nrlipsa2.out");
    while (fin>>x) {
        if (x >= -100 && x <= 100)
            f[x+100] = 1;
    }
    for (i=-100;i<=100;i++)
        if (f[i+100] == 0){
            fout<<i<<" ";
            return 0;
        }
    fout<<"nu exista";
    return 0;
}
```

5. ***Sortarea prin numărare.*** Se dă un șir cu `n` numere. Numărul de elemente poate fi maxim un milion iar valorile sunt naturale cu maxim `4` cifre. Să se afișeze elementele șirului în ordine crescătoare.

**Rezolvare**

O sortare cu un algoritm standard este nepractică, $n^2$ pentru $n$ până la $10^6$ însemnând timp de executare foarte mare. Observăm că în loc să memorăm valorile citite într-un vector, putem să numărăm pentru fiecare de câte ori apare. Pentru aceasta ne trebuie un vector `f` cu indici până la `9999` în care `f[i]` reprezintă numărul de apariții ale valorii `i` în șirul dat. Pentru construirea lui `f` este necesară o incrementarea lui `f[x]` la o anume valoare `x` citită, așadar timp de executare de ordin `n`.

Odată construit `f`, vom parcurge crescător indicii săi, de la `0` la `9999` și la o anume poziție `i` vom afișa de `f[i]` ori valoarea `i`. Numărul de pași făcuți la această etapă este de ordin `n+val_max`. Să justificăm acest lucru: La fiecare poziție `i` se fac `f[i]` repetări. Dar suma valorilor din `f` este chiar `n`, numărul de elemente din tot șirul. Apare și termenul `val_max` întrucât la o valoare foarte mică a lui `n` se fac totuși `val_max` pași, parcurgându-se toată plaja de valori posibile.

```
#include <iostream>
using namespace std;
int n, i, j, x, f[10000];
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>x;
        f[x]++;
    }
    for (i=0;i<=9999;i++)
        for (j=1;j<=f[i];j++)
            cout<<i<<" ";
    return 0;
}
```

Dacă valorile din șirul de sortat permit folosirea lor ca indici într-un vector (în fapt nu trebuie să fie reale și trebuie să acopere o plajă relativ mică de valori), această metodă este cel mai bun algoritm de sortare.

6. ***Ciurul lui Eratostene.*** Pentru `n` dat (maxim un milion), se cere determinarea tuturor numerelor prime cel mult egale cu `n`.

**Rezolvare**

Soluția brut presupune iterarea prin toate valorile de la `2` la `n` și realizarea verificării pentru valoarea curentă cu un algoritm ce caută divizorii până la radicalul valorii de testat. Evident că se pot face o serie de optimizări: se testează doar valorile impare de după `2`, se face `break` după fiecare divizor găsit etc. Complexitatea teoretică este de ordinul $n \cdot \sqrt{n}$. Memoria necesară este constantă.

O soluție care necesită spațiu de memorie de ordin `n` dar timp de calcul de ordinul `n` * (o valoare logaritmică cu bază supraunitară) se obține folosind următoarea abordare:

Pornim cu un vector cu `n` componente care are inițial toate valorile nule. Parcurgem începând cu `2` și, dacă valoarea de la poziția curentă `i` este egală cu `0`, vom marca `1` toți multiplii lui `i`.

Exemplu:

Inițial:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

La primul pas al iterației `i` este `2` și întrucât `f[i]` este `0`, vom marca la `1` toți multiplii lui `2`

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** |

Trecem la următorul pas al iterației, `i` este `3`, acum `f[3]` este tot `0` și vom face `1` în `f` la toate pozițiile multipli de `3`. Observăm că unele dintre ele sunt deja marcate cu `1` (cele multipli de `6`, deci marcate anterior și din `2`), dar acest lucru nu afectează algoritmul.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | **1** | 0 | 1 | **1** | 1 | 0 | **1** | 0 | 1 | **1** | 1 | 0 | **1** | 0 | 1 | **1** | 1 | 0 | **1** | 0 | 1 |

La pasul următor `i` este `4`, dar `f[4]` este deja `1` deci nu vom mai marca multiplii lui. Aceștia sunt deja `1`, marcați de acel divizor care l-a marcat și pe `4`, adică de `2`.

La pasul următor `i` este `5`, `f[5]` este `0` și atunci facem marcarea din `5` în `5`.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | **1** | 0 | 1 | 0 | 1 | **1** | 1 | 0 | 1 | 0 | **1** | 1 | 1 | 0 | 1 | **1** | 1 |

Observăm că doar din numerele prime se realizează marcarea multiplilor lor, așadar acestea fiind singurele care rămân marcate cu `0`.

Algoritmul care realizează acest lucru arată așa:

```
for (i=2;i<=n;i++)
    if (f[i] == 0)
        for (j=i+i;j<=n;j+=i)
            f[j] = 1;
for (i=2;i<=n;i++)
    if (f[i] == 1)
        cout<<i<<" ";
```

Această metodă de precalculare a numerelor prime poartă numele de *Ciurul lui Eratostene*.

**Analiză:**

- De reținut codul îngroșat, `j+=i` precum și faptul că se pornește prima iterație de la `2`. Aceste elemente sunt decisive în obținerea timpului bun (prima) și a corectitudinii, a doua. Dacă am porni iterația de la `1`, deja de la primul pas am marca toate elementele și nu ar mai avea sens restul algoritmului, fiind deja într-o stare greșită.
- Saltul `j+=i` este cel care ne duce la număr de pași: $\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \dots$ . Cu noțiuni mai avansate de matematică se poate justifica, folosind această formulă, complexitatea.
- Cu cât valoarea lui `n` este mai mare, cu atât varianta "`n log`" devine mai bună comparativ cu aceea „`n radical`". Gândiți-vă că la `1000000`, de exemplu, radicalul este `1000` iar log este în jur de `20`. Evident că sunt aproximări, dar oferă o idee bună asupra a ceea ce se întâmplă în practică. Continuând analiza noastră, facem mai bine distincție dacă scriem formula completă a complexității: n·radical înseamnă `1000000 · 1000` iar `n·logaritm` înseamnă `1000000 · 20`.
- Atenție la situațiile când se impune una sau alta dintre metode. Dacă dorim să testăm dacă este prim un singur număr, evident că nu are sens să facem ciur. Ciurul ar implica deja `n` · ceva (plus `n` locații de memorie necesare) pe când testul simplu înseamnă doar `radical(n)`. Dacă însă se cere să testăm dacă sunt prime mai multe numere dintr-un set, în funcție de numărul lor, se cuvine să analizăm dacă este mai bine să precalculăm totul cu un ciur și apoi să verificăm dacă este `0` sau `1` în dreptul fiecărui număr de testat sau dacă facem brut la fiecare. Evident ca mai trebuie ținut cont cât de mari pot fi valorile de testat, pentru a stabili dacă avem memorie să ținem structura de la ciur.
- `f[0]` și `f[1]` pot fi făcute `1` la final, aceste numere nefiind considerate prime.

7. Precalcularea numărului de divizori pentru fiecare valoare mai mică sau egală cu un `n` dat.

**Rezolvare**

Este o aplicație foarte utilă a algoritmului de la ciur. La analizarea multiplilor unui număr, în loc să facem `1` în dreptul fiecărui multiplu, mărim cu `1` acolo (semnificație: valoarea `i` curentă este încă un divizor al acelui multiplu). Modificările sunt: se iterează acum de la `1` (si el este divizor pentru celelalte numere) iar analiza

multiplilor se face și pentru valorile care nu sunt prime (sunt deja nenule în `f` când se ajunge cu `i` la ele). În fine, ultima diferență, când se analizează multiplii lui `i` se incrementează inclusiv `f[i]` (el însuși este divizor al său).

```
for (i=1;i<=n;i++)
    for (j=i;j<=n;j+=i)
        f[j]++;
for (i=1;i<=n;i++)
    cout<<i<<" are "<<f[i]<<" divizori\n";
```

Chiar dacă se iterează pe multiplii tuturor numerelor, nu doar pentru cele prime, compelxitatea teoretică este cea de la ciur.