Interclasarea optimă a două șiruri sortate

În acest material vom analiza care este modalitatea optimă din punct de vedere al timpului de executare pentru a obține un șir sortat care să conțină toate elementele din alte două șiruri sortate care se dau.

Se dau, un șir A cu n elemente numere naturale și un șir B cu m elemente, numere naturale. Se cere construirea unui șir C cu toate cele n+m elemente ordonate crescător. Se garantează că numerele n și m sunt naturale cuprinse între 1 și 10^5 iar elementele celor două șiruri sunt cuprinse între 1 și 10^9. Pentru 40% din punctaj problema va fi testată cu valori n și m cel mult egale cu 1000. Tot pentru 40% din punctaj problema va fi testată cu valori ale elementelor șirurilor date cuprinse între 1 și 10^5.

Ne vom concentra pe secvența care rezolvă problema, citirea și afișarea urmând să le facem doar în exemplul final. Ca rezultat va fi un șir C cu n+m elemente. Considerăm că tablourile cu care vom lucra vor fi declarate astfel:

```
int A[100001], B[100001], C[200002];
```

Vom considera că datele citite la intrare sunt memorate în șiruri începând cu poziția 1.

Exemplu:

Date de intrare	Date de ieșire
3 1 3 5 4 1 2 3 4	1 1 2 3 3 4 5

Soluția 1.

Vom copia în șirul C elementele șirului A, apoi, în continuare vom copia elementele șirului B și vom sorta șirul C.

```
for (i=1;i<=n;i++)
    C[i] = A[i];
for (i=1;i<=m;i++)
    C[n+i] = B[i];
k = n+m;
for (i=1;i<k;i++)
    for (j=i+1;j<=k;j++)
        if (C[i] > C[j]) {
            aux = C[i];
            C[i] = C[j];
            C[j] = aux;
        }
```

Am folosit un algoritm de sortare obișnuit, sortarea prin comparare. Secvența de cod care realizează sortarea este cea critică, timpul de executare fiind de ordinul $(n+m)^2$. Evident, că am putea îmbunătăți algoritmul, folosind o metpdă de sortare mai rapidă. Metoda aplicată mai sus nu ține cont de faptul că șirurile date sunt sortate, ea funcționând pe orice configurație a șirurilor de la intrare.

Soluția 2.

Vom copia inițial în C doar șirul A. Deci după această etapă șirul C este ordonat. Astfel, elementele șirului B le vom insera unul câte unul în C, la locul potrivit, așa încât după fiecare inserare elementele lui C să rămână ordonate crescător. Cumva procedăm ca la sortarea prin inserare.

```
for (i=1;i<=n;i++)
    C[i] = A[i];
k = n;
for (i=1; i<=m; i++) {
    j = k;
    while (j >= 1 && C[j] > B[i]) {
        C[j+1] = C[j];
        j--;
    }
    j++;
    C[j] = B[i];
    k++;
}
```

Și această metodă, chiar dacă ține cont că șirul A este ordonat (prin faptul că imediat după copierea lui A în C ne bazăm pe șirul C ordonat) ajunge tot la timp de executare pătratic, de data aceasta de ordin m*n,

Oricare dintre soluțiile de mai sus vor promova testele pentru care atât n cât și m sunt maxim 1000.

Soluția 3.

Putem folosi un vector de frecvență pentru a sorta. Astfel, vom contoriza numărul de apariții ale fiecărei valori întâlnită la intrare. Din restricții deducem că vom avea o rezolvare corectă pentru 40% din punctaj. Nu pentru 100% pentru că nu am putea aloca memorie pentru un tablou cu 10^9 indici, dar si pentru că în etapa a doua a algoritmului (pe care îl vom prezenta) ar fi necesar și timp de calcul de ordinul 10^9. Așa că această metodă o vom folosi pentru a obține puntajul pe testele în care valoarea maximă a elementelor este de cel mult 10^5.

```
int f[100001];
for (i=0;i<=100000;i++)
    F[i] = 0;
for (i=1;i<=n;i++)
    F[ A[i] ] ++;
for (i=1;i<=m;i++)
    F[ B[i] ] ++;
for (i=0; i<=100000; i++)
    for (j=1;j<=F[i];j++)
        cout<<i<<" ";
```

Am subliniat anterior limitările metodei în cazul în care elementele șirurilor au valori mari. Metoda este însă foarte rapidă în cazul în care n și m sunt mari dar valorile din șiruri sunt mai mici, timpul de executare fiind de ordinul n+m+valoarea_maxima. Faptul că la afișare sunt două foruri unul în altul, nu este periculos întrucât numărul total de afișări care se fac este chiar n+m.

Soluția 4.

În acest caz vom folosi pentru ambele șiruri faptul că ele sunt ordonate (crescător). Se observă că primul element din C va fi minimul dintre A[1] și B[1]. După ce "ducem" în C acest element avansăm la următorul

în șirul din care l-am extras, observăm că al doilea element care va ajunge în `C` este fie primul element din șirul din care n-am extras minimul fie al doilea element din șirul în care am avansat (nu uităm că sunt ordonate ambele șiruri). Astfel, vom folosi două variabile care să indice spre următorul element de extras din fiecare șir (inițial `i=1` pentru șirul `A` și `j=1` pentru șirul `B`, elementul de pe poziția `1` din fiecare șir fiind la început neextras). Vom aplica acest pas până când unul dintre șiruri își va epuiza elementele. Evident, elementele rămase în celălalt vor fi adăugate la soluție.

```
i = 1;
j = 1;
k = 0;
while (i <= n && j <= m) {
    if (A[i] < B[j]) {
        k++;
        C[k] = A[i];
        i++;
    } else {
        k++;
        C[k] = B[j];
        j++;
    }
}
for (;i<=n;i++) {
    k++;
    C[k] = A[i];
}
for (;j<=m;j++) {
    k++;
    C[k] = B[j];
}
```

Instrucțiunea while conține codul care face să se acumuleze elemente în soluție până se epuizează unul dintre șiruri. Atenție că dacă ne grăbim am putea spune că se "termină" șirul cel mai scurt, dar nu este așa, cel care se termină este șirul cu elementul de la fina mai mic.

Odată ajunsă falsă condiția de la while, fie `i>n` fie `j>m`. Astfel, încă de la început condiția unuia dintre cele două foruri va fi falsă, deci blocul subordonat se va executa cel puțin o dată doar la un for (observați că ne-a fost comod să nu mai scriem expresia de inițializare la foruri lăsând pe `i` și pe `j` așa cum ele ies din while).

Observăm că la fiecare pas se duce un element în șirul `C` (nu se fac pași inutili). Așadar, timpul de executare este de ordinul `n+m`, (mai mult nu am putea obține, aceasta fiind și complexitatea necesară pentru citire).

În continuare este prezentat programul complet pentru soluția optimă. Observați și unele prescurtări la cod, spre exemplu, secvența: `k++; C[k] = A[i]; i++;` poate fi scrisă mai elegant: `C[++k] = A[i++];`

```
#include <iostream>
using namespace std;
int A[100001], B[100001], C[200001];
int n, m, i, j, k, aux;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>A[i];
```

```
    cin>>m;
    for (i=1;i<=m;i++)
        cin>>B[i];
    i = 1;
    j = 1;
    k = 0;
    while (i <= n && j <= m)
        if (A[i] < B[j])
            C[++k] = A[i++];
        else
            C[++k] = B[j++];
    for (;i<=n;i++)
        C[++k] = A[i];
    for (;j<=m;j++)
        C[++k] = B[j];
    for (i=1;i<=k;i++)
        cout<<C[i]<<" ";
    return 0;
}
```

Să analizăm și următoarea soluție:

```
i = 1;
j = 1;
k = 0;
while (i <= n || j <= m) {
    while (i<=n && A[i] <= B[j])
        C[++k] = A[i++];
    while (j<=m && B[j] <= A[i])
        C[++k] = B[j++];
}
```

Poate că veți considera că ea este mai aproape de modul natural de gândire. Adică: cât timp elementul curent dintr-un șir este mai mic, punem în soluție din acel șir și avansăm. Am adăugat elemente fictive foarte mari la finalul fiecărui șir pentru a evita cazuri particulare.

Și această soluție duce mereu câte ceva în șirul `C`, timpul de executare fiind tot de ordinul `n+m`. Rămâne ca fiecare să aleagă un tipar sau altul în funcție de situație.

Probleme rezolvate

1. Se dă un șir `A` cu `n` elemente, ordonat crescător și un șir `B` cu `m` elemente, ordonat **descrescător**. Se cere interclasarea lor, în ordine crescătoare în șirul `C`, ordonat crescător.

Exemplu:

Date de intrare	Date de ieșire
3 1 3 5 4 4 3 2 1	1 1 2 3 3 4 5

Se obține tot timp de executare de ordin n+m parcurgând șirul B în ordine inversă.

```
i = 1;
j = m;
k = 0;
while (i <= n && j >= 1)
    if (A[i] < B[j])
        C[++k] = A[i++];
    else
        C[++k] = B[j--];
for (;i<=n;i++)
    C[++k] = A[i];
for (;j>=1;j--)
    C[++k] = B[j];
```

2. Se dă un șir A cu n elemente, ordonat strict crescător și un șir B cu m elemente, ordonat strict crescător. Se cere interclasarea lor, în ordine crescătoare în șirul C. Se cere ca în șirul C elementele să fie disticte (oricare element care apare în vreun șir să apară în C exact o dată).

Exemplu:

Date de intrare	Date de ieșire
3 1 3 5 6 1 2 3 4 5 6	1 2 3 4 5 6

O primă idee este să aplicăm ad literam algoritmul de interclasare optimă pentru cele două șiruri date. În final, C va conține fiecare element comun de exact două ori (șirurile date au elementele distincte). Parcurgem la final șirul C și afișam doar acele valori diferite de valoarea precedentă. Aceasta este o soluție optimă ca timp.

Soluția următoare va pune în C direct doar elementele care sunt necesare.

```
#include <iostream>
using namespace std;
int A[100001], B[100001], C[200001];
int n, m, i, j, k, aux;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>A[i];
    cin>>m;
    for (i=1;i<=m;i++)
        cin>>B[i];
    i = 1;
    j = 1;
    k = 0;
    C[0] = -1;
    while (i <= n && j <= m)
        if (A[i] < B[j]) {
            if (A[i] != C[k]) {
```

```
                k++;
                C[k] = A[i];
            }
            i++;
        }
        else {
            if (B[j] != C[k]) {
                k++;
                C[k] = B[j];
            }
            j++;
        }
    for (;i<=n;i++) {
        if (A[i] != C[k])
            C[++k] = A[i];
    }
    for (;j<=m;j++) {
        if (B[j] != C[k])
            C[++k] = B[j];
    }
    for (i=1;i<=k;i++)
        cout<<C[i]<<" ";
    return 0;
}
```

Observăm că, atunci când dorim să plasăm în soluție un element din unul dintre șiruri, verificăm ca acela să fie diferit de ultimul element plasat deja în soluție (elementele se pun în soluție în ordine crescătoare). Atenție, chiar dacă elementul candidat nu este dus în soluție, vom trece peste el în șirul din care face parte (`i++` sau `j++`). Pentru evitarea cazurilor particulare, am inițializat `C[0]` cu `-1` (o valoare strict mai mică decât elementele șirurilor).

3. Se dă un șir `A` cu `n` elemente, ordonat strict crescător și un șir `B` cu `m` elemente, ordonat strict crescător. Se cere interclasarea în șirul `C` doar a valorilor care apar în ambele șiruri. Elementele din șirul `C` vor fi distincte.

Exemplu:

Date de intrare	Date de ieșire
3 1 3 5 6 2 3 4 5 6 8	3 5

Observăm că, la fel ca la problema anterioară, elementele șirurilor date fiind distincte, dacă am interclasa cu algoritmul clasic totul, elementele care ne interesează apar de două ori.

Nu este necesar să interclasăm totul, ci doar să ducem în soluție un element atunci când avem `A[i]=B[j]`.

```
#include <iostream>
using namespace std;
int A[100001], B[100001], C[200001];
int n, m, i, j, k, aux;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
```

```
        cin>>A[i];
    cin>>m;
    for (i=1;i<=m;i++)
        cin>>B[i];
    i = 1;
    j = 1;
    k = 0;
    C[0] = -1;
    while (i <= n && j <= m)
        if (A[i] < B[j])
            i++;
        else
            if (A[i] > B[j])
                j++;
            else {
                C[++k] = A[i];
                i++;
                j++;
            }

    if (i <= n && A[i] == B[m])
        C[++k] = A[i];

    if (j <= m && B[j] == A[n])
        C[++k] = B[j];

    for (i=1;i<=k;i++)
        cout<<C[i]<<" ";
    return 0;
}
```

Nu mai sunt necesare instrucțiunile for de la final, dar trebuie totuși să testăm dacă elementul următor din șirul nefinalizat este egal cu ultimul element din celălalt șir. De asemenea, am făcut o mică optimizare, avansând în ambele șiruri atunci când elementele ce se compară sunt egale.

4. Se dau `n` numere naturale nenule. Să afișeze câte dintre ele sunt termeni ai șirului lui Fibonacci. Cele `n` numere sunt date în ordine crescătoare.

Exemplu:

Date de intrare	Date de ieșire	Explicație
`10` `1 4 5 5 5 10 11 13 20 21`	`6`	Valorile 1, 5, 5, 5, 13 și 21 sunt termeni Fibonacci

Cunoaștem că șirul lui Fibonacci este definit recurent, primii doi termeni fiind `1` iar fiecare dintre următorii se obține însumând pe precedenții doi. Astfel, la începutul șirului avem: `1, 1, 2, 3, 4, 8, 13, 21, 34, 55, 89, 144 ...`

O primă soluție este de a testa pe rand fiecare termen al șirului dat dacă face parte din șirul lui Fibonacci. Pentru a face acest test, profităm de faptul că șirul Fibonacci este strict crescător începând cu al treilea termen și atunci putem genera termenii săi de la început până întâlnim sau depășim termenul de testat.

```
#include <iostream>
using namespace std;
int n, x, a, b, c, sol, i;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x == 1) {
            sol++;
            continue;
        }
        a = 1;
        b = 1;
        while (a+b <= x) {
            c = a + b;
            a = b;
            b = c;
        }
        if (c == x)
            sol++;
    }
    cout<<sol;
    return 0;
}
```

Timpul de executare al acestui algoritm este de ordinul `n*L`, unde `L` este numărul de pași necesari calculului acelui termen Fibonacci corespunzător valorii maxime din șir. Acest număr `L` cunoaștem că este de ordin logaritmic într-o bază supraunitară: `(1+sqrt(5))/2`.

Soluția a doua va avea timpul de executare de ordinul `n + L`, lucru pe care îl obținem cu următoarea observație: atât șirul dat cât și Fibonacci sunt crescătoare și atunci, la trecerea la un element nou în șirul dat nu este necesar să regenerăm tot șirul Fibonacci ci să avansăm, eventual, de la termenul curent.

```
#include <iostream>
using namespace std;
int n, x, a, b, c, sol, i;
int main () {
    cin>>n;
    a = 1;
    b = 1;
    c = 1;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x == c) {
            sol++;
            continue;
        }
        while (c < x) {
            c = a+b;
            a = b;
            b = c;
        }
    }
```

```
    cout<<sol;
    return 0;
}
```

O aplicație foarte importantă a interclasării optime a două șiruri sortate este metoda sortării prin interclasare (merge sort) care oferă timp de calcul de ordin $n * \log_2 n$ și pe care o vom studia la capitolul Divide et Impera.

Probleme propuse

1. Se dau două șiruri de numere ordonate crescător (nu neapărat strict). Să se obțină în ordine strict crescătoare elementele care apar în cel puțin un șir.

Exemplu

Date de intrare	Date de ieșire
4 1 3 3 3 3 2 3 5	1 2 3 5

2. Se dau două șiruri de numere ordonate crescător (nu neapărat strict). Să se obțină în ordine strict crescătoare elementele care în ambele șiruri.

Exemplu

Date de intrare	Date de ieșire
7 1 3 3 3 4 5 5 3 2 3 5	3 5