Căutare binară

În această lecție vom studia căutarea unei valori într-un șir despre care știm că este ordonat. Algoritmul optim care realizează acest lucru are însă mult mai multe utilizări, pentru probleme care aparent nu au nicio legătură cu enunțul deja dat.

Să enunțăm problema: Fie `v` un tablou unidimensional cu `n` valori întregi, **ordonate crescător**. Se mai dă o valoare `x` și se cere să precizăm dacă `x` se află sau nu în `v`. În caz afirmativ trebuie afișată o poziție pe care îl găsim pe `x`. În continuare vom considera elementele plasate pe poziții între `1` și `n`.

Soluția 1 - Căutare secvențială

```
for (i=1;i<=n;i++)
    if (v[i] == x) {
        break;
    }
if (i <= n)
    cout<<"DA "<<i;
else
    cout<<"NU";
```

Aceasta este o abordare care nu ține seama de faptul că șirul dat este ordonat crescător. Timpul de executare al unui astfel de cod este de ordinul `n`.

Soluția 2 - Căutare binară

Analizăm ce se întâmplă dacă vom compara pe `x` cu o valoare aflată pe o poziție oarecare `p` din șir. Dacă `v[p]` este egal `x` am fost norocoși afișăm `p` și taskul este încheiat. În caz contrar, vom profita de faptul că sirul este ordonat crescător și decidem de care parte a poziției `p` continuăm căutarea. Dacă `x > v[p]` are sens să căutăm doar între pozițiile `p+1` și `n` iar dacă `x < v[p]` vom căuta doar între pozițiile `1` și `p-1`. Așadar, dacă inițial aveam `n` poziții candidat pentru locul unde se află `x`, după acest test avem fie `p-1` fie `n-p` poziții. Evident, una dintre cele două valori este mai mică sau egală cu `n/2` iar cealaltă este în general mai mare sau egală cu `n/2`. Dacă am fost norocoși vom continua în secvența mai scurtă, însă putem avea și ghinion și să fim aruncați în cealaltă parte.

Nu am spus însă cum alegem poziția `p` unde încercăm. Dacă o vom face la jumătatea secvenței candidat, vom elimina hazardul și indiferent de partea în care vom continua căutarea, lungimea noii secvențe candidat este cel mult egală cu jumătate din cea a secvenței inițiale.

Pentru noua secvență candidat vom relua procedeul (determinăm poziția de la mijloc, apoi decidem de care parte a sa continuăm, bineînțeles, dacă nu am fost deja norocoși să nimerim valoarea căutată).

Concret, procedăm astfel:

Alegem două variabile care vor reprezenta mereu indicii de început și de final ai secvenței candidat. Să le numim `st` (pentru indicele stâng) și `dr` (pentru indicele drept). Inițial `st = 1` și `dr = n`. Calculăm `mid = (st + dr) / 2`.

Dacă `v[mid]` este egal cu `x`, am găsit și ne oprim. Dacă `x > v[mid]`, va trebui să continuăm în partea dreaptă, adică actualizăm `st = mid+1` și lăsăm `dr` nemodificat. Ultimul caz (`x < v[mid]`) modifică starea astfel: `st` rămâne nemodificat și `dr` devine `mid-1`.

În continuare avem un exemplu pentru a ilustra modul în care se modifică intervalul candidat:

Fie șirul: `n = 11` și valorile: `1, 2, 5, 7, 9, 12, 15, 16, 20, 29, 78` și `x = 16`

st	dr	mid	
1	11	(1+11)/2 = 6	v[mid] = 12 și este mai mic decât x=16, deci se continuă în dreapta
7	11	(7+11)/2 = 9	v[mid] = 20 și este mai mare decât x=16, deci se continuă în stânga
7	8	(7+8)/2 = 7	v[mid] = 15 și este mai mic decât x=16, deci se continuă în dreapta
8	8	(8+8)/2 = 8	v[mid] = 16, am găsit valoarea căutată și ne oprim

Observăm că dacă intervalul de căutat are lungime impară, cele două părți candidat au ambele aceeași lungime (parte întreagă din jumătatea lungimii anterioare). Dacă intervalul de căutat are lungimea pară, partea în care putem continua în dreapta este egală cu jumătate din lungimea curentă, iar partea din stânga este chiar cu `1` mai scurtă. Deci, indiferent de situație suntem siguri că realizăm înjumătățirea lungimii secvenței candidat pentru pasul următor.

Așadar, dacă inițial căutăm într-o secvență de lungime `n`, după prima încercare lungimea secvenței de căutat va fi maxim `n/2`, după încă o încercare se ajunge la o secvență de lungime $n/2^2$, apoi $n/2^3$, … Așadar când puterea lui `2` face ca numitorul să devină mai mare ca numărătorul nu mai are sens continuarea, garantându-se astfel număr de pași de ordinul $\log_2 n$ până se găsește elementul căutat (sau se decide că `x` nu se află în șir).

Înainte de a scrie algoritmul se cuvine să analizăm ce se întâmplă când valoarea de căutat nu se află în șir. Cel mai bine facem asta pe un exemplu. Să presupunem că pentru șirul de mai sus căutăm valoarea `x = 18`.

Se parcurg stările:

st	dr	mid	
1	11	(1+11)/2 = 6	v[mid] = 12 și este mai mic decât x=18, deci se continuă în dreapta
7	11	(7+11)/2 = 9	v[mid] = 20 și este mai mare decât x=18, deci se continuă în stânga
7	8	(7+8)/2 = 7	v[mid] = 15 și este mai mic decât x=18, deci se continuă în dreapta
8	8	(8+8)/2 = 8	v[mid] = 16, și este mai mic decât x=18, deci se continuă în dreapta
9	8	În acest caz (st > dr) nu mai putem vorbi de un interval, acesta degenerându-se	

Se poate verifica pe mai multe exemple dar se poate și arăta ușor că are sens continuarea căutării cât timp secvența de la `st` la `dr` reprezintă cel puțin un element, așadar **`st <= dr`**.

```
st = 1;
dr = n;
while (st <= dr) {
    mid = (st + dr) / 2;
    if (v[mid] == x)
        break;
    else
        if (v[mid] < x)
            st = mid+1;
        else
            dr = mid-1;
}
if (st <= dr)
    cout<<"DA "<<mid;
else
    cout<<"NU";
```

La terminarea repetiției putem testa dacă încă mai este adevărată condiția acesteia, caz în care am ieșit cu break (deci am găsit valoarea).

Când valoarea de căutat nu se află în șir, observăm că `st` ajunge mai mare decât `dr` și, lucru extrem de important, `v[st]` este cea mai din stânga valoare mai mare decât `x` iar `v[dr]` este cea mai din dreapta valoare mai mică decât `x`.

Pentru exemplu de mai sus, la căutarea valorii `18` s-ar fi ieșit astfel:

```
                        dr  st
1  2  5  7  9  12  15  16  20  29  78
```

Pentru o scriere mai compactă observăm că, întrucât pe prima ramură a primului if folosim break, este posibilă evitarea lui else, și al doilea if să apară imediat după primul.

```
while (st <= dr) {
    mid = (st + dr) / 2;
    if (v[mid] == x)
        break;
    if (v[mid] < x)
        st = mid+1;
    else
        dr = mid-1;
}
```

S-ar putea spune că pentru problema anterioară oricum avem timp de executare de ordin `n` de la citirea șirului și se mai poate spune că am fi putut face căutarea chiar de la citire. Astfel nu s-ar justifica optimizarea semnificativă oferită de căutarea binară.

Nu vom insista aici pe contraexemple, valoarea algoritmului observându-se cel puțin din cele ce urmează în acest material.

Aplicații

1. Se consideră un șir în care elementele sunt egale cu `0` până la un moment dat, după care toate sunt egale cu `1`. Să se determine pozițiile unde se schimbă valoarea elementelor (ultima poziție cu valoarea `0` și prima cu valoarea `1`). Considerăm șirul memorat într-un vector `v` indexat de la `0` și cu `n` elemente.

Exemplu: `n = 18` iar șirul: `0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1`. Se vor afișa valorile `6` și `7`.

Rezolvare.

Inițial considerăm `st = 1` și `dr = n`. Pe principiul de mai sus, vom calcula mereu `mid = (st + dr)/2`. De această dată nu se mai pune problema găsirii unei valori anume ci vom trata cazurile ce pot să apară, acestea fiind acum două:

- `v[mid] = 1`. Suntem în partea de șir ce conține valoarea din dreapta (`1`), vom continua căutarea în stânga, așadar `dr = mid-1`
- `v[mid] = 0`. Suntem în partea de șir ce conține valoarea din stânga (`0`), vom continua căutarea în dreapta, așadar `st = mid+1`

Ne asigurăm astfel că, în partea în care mergem, intervalul de la `st` la `dr` conține valori de ambele feluri sau suntem în cazul la limită când dr memorează cea mai din dreapta poziție unde este `0` sau st memorează cea mai din stanga poziție unde este `1`.

```
st = 1;
dr = n;
while (...) {
    mid = (st + dr) / 2;
    if (v[mid] == 1)
        dr = mid-1;
    else
        st = mid+1;
}
cout<<...;
```

Așa cum am mai amintit și cum se poate vedea și din algoritm, sunt doar două cazuri de tratat. Iată o simulare pe exemplul de mai sus

st	dr	mid	
1	18	(1+18)/2 = 9	st mid dr 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 v[9] = 1, continuăm în stânga
1	8	(1+8)/2 = 4	st mid dr 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 v[4] = 0, continuăm în dreapta

5	8	(5+8)/2 = 6	st mid dr 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 v[6] = 0, continuăm în dreapta
7	8	(7+8)/2 = 7	st dr mid 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 v[7] = 1, continuăm în stânga
7	6	Acum st > dr	dr st 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1

Observăm că și în acest caz oprirea se face când se degenerează intervalul de căutare, adică `st` devine mai mare decât `dr`. Observăm că de câte ori am fost cu mid pe un element `0`, `st` a mers pe poziția cu `1` mai mare, adică st tinde spre un element cu valoarea `1`. Analog, dr tinde către un element cu valoarea `0`. Cum ambele rămân la final vecine, concluzionăm deci că `st` va memora prima poziție (cea mai din stânga) pe care se află `1` iar `dr` memorează ultima poziție (cea mai din dreapta) unde se află `0`.

Forma finală a algoritmului este:

```
st = 1;
dr = n;
while (st <= dr) {
    mid = (st + dr) / 2;
    if (v[mid] == 1)
        dr = mid-1;
    else
        st = mid+1;
}
cout<<dr<<" "<<st;
```

2. Se consideră un șir cu `n` elemente, nu neapărat distincte, ordonate crescător. Verificați dacă o anume valoare `x` apare în șir și în caz afirmativ determinați **cea mai mică** poziție pe care aceasta se află.

Exemplu: pentru `n = 16` și șirul: `2 7 7 9 9 9 9 9 10 10 10 20 22 22 30 100` și `x = 9`, trebuie obținută valoarea `4`.

O primă abordare ar fi să folosim căutarea binară pentru a obține o apariție oarecare a lui `x` (în timp logaritmic, după cum știm) și apoi să folosim o repetiție care parcurge la stânga element cu element cât timp avem valoarea `x`. Din păcate, această a doua etapă nu se comportă bine pe un șir în care majoritatea elementelor sunt `x`, putând să se ajungă până la `n/2` pași, ajungând astfel la timp de executare de ordin `n` pe această secvență. Putem însă, printr-o observație, să aducem problema la cea anterioară.

Ne imaginăm că vom construi un alt șir `w` în care `w[i]` va fi `0` daca `v[i]` este strict mai mic decât `x` și `w[i]` va fi `1` dacă `v[i]` este mai mare sau egal decât `x`.

```
2 7 7 9 9 9 9 9 10 10 10 20 22 22 30 100
0 0 0 1 1 1 1 1  1  1  1  1  1  1  1  1
```

Aplicând pe noul șir algoritmul prezentat anterior, rămâne doar să afișăm valoarea variabilei st pentru a obține rezultatul corect.

Noi nu construim efectiv vectorul `w`, insă dacă `v[mid]` este strict mai mic decât `x` alegem să continuăm în dreapta, iar dacă `v[mid]` este mai mare sau egal decât `x` vom continua în stânga.

```
st = 1;
dr = n;
while (st <= dr) {
    mid = (st + dr) / 2;
    if (v[mid] < x)
        st = mid+1;
    else
        dr = mid-1;
}
cout<<st;
```

3. **Căutare binară a rezultatului.**

Pentru început considerăm următoarea problemă (*fabrica*, *pbinfo.ro*):

La secția de împachetare a produselor dintr-o fabrică lucrează `n` muncitori. Fiecare muncitor împachetează același tip de produs, și pentru fiecare se cunoaște timpul necesar pentru împachetarea unui obiect. Să se determine câte obiecte vor fi împachetate de cei `n` muncitori într-un interval de timp dat, `T`.

Restricții: $1 \leq n \leq 1000$, cei `n` timpi de împachetare vor fi mai mici decât 1000, $1 \leq T \leq 1.000.000$.

Soluția acestei probleme este una imediată. Observăm că un muncitor `i` împachetează T/t_i obiecte, unde t_i este timpul în care el împachetează un obiect. Obținem rezultatul însumând aceste valori.

```
#include <iostream>
#define DIM 1010
using namespace std;
int t[DIM], n, i, sol, T;
int main(){
    cin>>n>>T; ///  numarul de muncitori si timpul total avut la dispozitie
    for (i=1;i<=n;i++)
        cin>>t[i]; ///  timpul in care impacheteaza un obiect muncitorul i
    sol = 0;
    for (i=1; i<=n; i++)
        sol += T/t[i];
    cout<<sol;
    return 0;
}
```

Se observă că problema putea fi rezolvată de la citire dar am utilizat tabloul `t` din rațiuni didactice.

Trecem acum la următorul enunț (*fabrica1*, *pbinfo.ro*)

La secția de împachetare a produselor dintr-o fabrică lucrează `n` muncitori. Fiecare muncitor împachetează același tip de produs, și pentru fiecare se cunoaște timpul necesar pentru împachetarea unui obiect. Să se determine durata minimă de timp în care vor împacheta cei `n` muncitori cel puțin `M` obiecte.

Restricții: `1 ≤ n ≤ 1000`, cele `n` numere citite vor fi mai mici decât `1000`, `1 ≤ M ≤ 1.000.000`.

Prima idee de rezolvare este aceea de a calcula, pentru diferite intervale de timp puse la dispoziție, numărul de pachete care se pot împacheta. Rulăm astfel problema anterioară pentru `T=1`, apoi `T=2`, `T=3`, ... și la prima valoare a lui `T` pentru care numărul de pachete calculat este mai mare sau egal decât `M` ne oprim, tipărind acel `T` ca soluție.

```
#include <iostream>
#define DIM 1010
using namespace std;
int t[DIM], n, i, sol, T, M;
int main(){
    cin>>n>>M;//numarul de muncitori si numarul de pachete de realizat
    for (i=1;i<=n;i++)
        cin>>t[i]; //timpul in care impacheteaza un obiect muncitorul i
    for (T=1;;T++) {
        sol = 0;
        for (i=1; i<=n; i++)
            sol += T/t[i];
        if (sol >= M) {
            cout<<T;
            break;
        }
    }
    return 0;
}
```

Pentru datele de test timpul de calcul este de ordinul `n * M` (pentru cazul în care timpul asociat fiecărui muncitor este foarte mare).

Facem însă următoarea observație: Pe măsură ce timpul crește, va crește și numărul total de pachete care se pot realiza de cei `n` muncitori. Iată cum variază această valoare, odată cu creșterea lui `T` pe următorul exemplu:

`n = 6`, `M = 10`, iar timpii de împachetare: `4 7 3 6 7 1`.

```
T:   1 2 3 4 5  6  7  8  9 10 11 12 13 14 15
sol: 1 2 4 6 7 10 13 15 17 18 19 23 24 27 29
```

Așadar, în loc să calculăm pentru valori `T` crescător începând cu `1`, dacă vom calcula valoarea `sol` pentru un `T` oarecare, putem decide dacă apoi vom face calcule pentru valori `T` mai mari sau mai mici decât cea curentă.

Așadar, stabilind o valoare minimă pentru `T`, de exemplu `1` și o valoare maximă (`M * t1` - de exemplu împachetează primul muncitor toate pachetele - am putea optimiza alegând în loc de primul muncitor pe cel cu timpul de împachetare cel mai mare, dar asta nu ne-ar ajuta neapărat pe cazul cel mai defavorabil), vom

calcula pentru început numărul de pachete pentru T = “mijlocul intervalului stabilit” (folosind un for). Dacă acest număr este mai mare decât M, vom încerca un T mai mic, adică mergem în stânga, altfel, vom încerca un T mai mare, mergând în dreapta.

```
#include <iostream>
#define DIM 1010
using namespace std;
int t[DIM], n, i, sol, T, M, st, dr;
int main(){
    cin>>n>>M;
    for (i=1;i<=n;i++)
        cin>>t[i];

    st = 1; dr = M * t[1];
    while (st <= dr) {
        T = (st + dr)/2;

        sol = 0;
        for (i=1; i<=n; i++)
            sol += T/t[i];

        if (sol < M)
            st = T+1;
        else
            dr = T-1;
    }
    cout<<st;
    return 0;
}
```

Timpul de executare ajunge așadat la unul de ordinul $n * \log_2 tmaxim$. tmaxim ajunge la M * valoarea_maximă așadar nu mai mare decât 10^9, iar logaritmul acestei valori este în jur de 30.

Probleme rezolvate

1. Se dau n numere naturale distincte. Să se determine în câte moduri putem alege trei dintre ele așa încât suma valorilor a două alese să fie egală cu al treilea (https://www.infoarena.ro/problema/bete2).

Exemplu: pentru datele de intrare: n = 10 și valorile: 17 4 13 12 5 29 6 11 18 7 se va afișa 12. De exemplu, o soluție este formată din tripletul: 12 5 7. Nu se numără și permutări ale acestei soluții. Șirul dat este format din maxim 3000 de numere.

Rezolvare

O primă metodă este aceea de a genera toate tripletele și a face testul pentru fiecare în parte

```
#include <fstream>
#include <algorithm>
using namespace std;
int v[3010], i, j, k, sol, n;
```

```
int main() {
    ifstream fin("bete2.in");
    ofstream fout("bete2.out");
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i];
    for (i=1;i<n-1;i++)
        for (j=i+1;j<n;j++)
            for (k=j+1; k<=n; k++)
                if(v[k]==v[i]+v[j] || v[i]==v[k]+v[j] ||
v[j]==v[k]+v[i])
                    sol++;
    fout<<sol;
    return 0;
}
```

Această soluție are timp de executare de ordinul n^3. Pentru $n = 3000$ această soluție nu se încadrează în timp.

O soluție mai bună obținem cu următoarea observație. Dacă sortăm inițial numerele din șirul dat și fixăm (cu 2 de for) doi membri ai unui triplet, rămâne doar să verificăm dacă și suma lor se află în șir. Dar cum șirul este acum sortat, acest lucru se poate testa prin căutare binară. Astfel, această etapă a problemei are timpul de executare de ordinul $n^2 \log_2 n$. Sortarea necesară în prealabil are timpul de executare n^2 sau chiar mai mic (dacă folosim funcția de bibliotecă), așadar nu sunt probleme de încadrare în timp.

```
#include <fstream>
#include <algorithm>
using namespace std;
int v[3010], i, j, k, sol, n, st, dr;
int main() {
    ifstream fin("bete2.in");
    ofstream fout("bete2.out");
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i];
    sort(v+1, v+n+1);
    for (i=1;i<n-1;i++)
        for (j=i+1;j<n;j++) {
            int x = v[i] + v[j];
            // caut pe x de la pozitia j+1 pana la pozitia n
            st = j+1; dr = n;
            while (st <= dr) {
                int mid = (st + dr)/2;
                if (v[mid] == x) {
                    sol ++;
                    break;
                }

                if (x < v[mid])
                    dr = mid-1;
                else
                    st = mid + 1;
```

```
            }
        }
    fout<<sol;
    return 0;
}
```

Temă

Exerciții

Considerăm un tablou unidimensional `v` cu `n` elemente, indexat de la `1`, sortat crescător (nu neapărat strict).

1. Cum rămân variabilele `st` și `dr` dacă valoare `x` care se caută este strict mai mare decât toate elementele tabloului? Algoritmul de căutare binară folosit este cel prezentat în paragraful *Soluția 2*.
2. Cum rămân variabilele `st` și `dr` dacă valoare `x` care se caută este strict mai mică decât toate elementele tabloului? Algoritmul de căutare binară folosit este cel prezentat în paragraful *Soluția 2*.
3. Cum rămân variabilele st și dr la *Aplicația 1* dacă șirul dat conține numai valoarea `0` ?
4. Cum rămân variabilele st și dr la *Aplicația 1* dacă șirul dat conține numai valoarea `1` ?

Probleme

1. Determinați o poziție pe care o valoare `x` apare într-un șir sortat descrescător (sau spuneți dacă aceasta nu apare).
2. Determinați cea mai mare poziție pe care o valoare dată `x` apare într-un șir sortat crescător (nu neapărat strict), sau spuneți dacă `x` nu apare în șir.
3. Determinați prima poziție pe care apare o valoare strict mai mare decât un `x` dat, într-un șir crescător (sau spuneți că nu sunt valori mai mari decât `x` în șir).
4. https://www.pbinfo.ro/?pagina=probleme&id=508 (pbinfo, problema Cautare Binara).
5. https://infoarena.ro/problema/bete2
6. https://infoarena.ro/problema/cautbin
7. https://infoarena.ro/problema/nrtri
8. https://infoarena.ro/problema/transport
9. https://infoarena.ro/problema/fact
10. https://infoarena.ro/problema/pachete
11. https://infoarena.ro/problema/grupuri
12. https://infoarena.ro/problema/proc