TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

2. Căutarea binară

Fie t un tablou unidimensional format din n numere întregi sortate crescător şi x un număr întreg. Să se verifice dacă valoarea x apare în tabloul t.

Evident, problema ar putea fi rezolvată printr-o simplă parcurgere a tabloului t (*căutare liniară*), obţinând un algoritm având complexitatea $\mathcal{O}(n)$, dar nu am utiliza deloc faptul că elementele tabloului sunt în ordine crescătoare.

Pentru a efectua o căutare binară într-o secvenţa $t[p], t[p+1], \dots, t[u]$ a tabloului t în care $p \leq u$ vom folosi această ipoteză, comparând valoarea căutată x cu valoarea $t[mij]$ aflată în mijlocul secvenţei. Astfel, vom obţine următoarele 3 cazuri:

a) $x < t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[p], \dots, t[mij-1]$;
b) $x > t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[mij+1], \dots, t[u]$;
c) $x = t[mij] \Longrightarrow$ am găsit valoarea x, deci operaţia de căutare se încheie cu succes.

Dacă la un moment dat $p > u$, înseamnă că nu mai există nicio secvenţă $t[p], \dots, t[u]$ în care să aibă sens să căutăm valoarea x, deci operaţia de căutare eşuează.

Exemplu:

Fie $t = (\underbrace{2}_{p=0,\ m=0},\ \underbrace{2}_{u=1},\ \underbrace{3}_{m=2},\ \cancel{7},\ \underbrace{\cancel{11}}_{u=4},\ \underbrace{\cancel{14}}_{m=5},\ \cancel{21, 21, 21, 30,\ \underbrace{40}_{u=10}})$ cu $n = 11$ elemente şi $x = 2$.

Implementare:

```
#include <iostream>

using namespace std;

int cbin(int t[], int p, int u, int x)
{
    if(p > u)
        return -1;

    int mij = (p + u) / 2;

    if(t[mij] == x)
        return mij;
    else
        if(x < t[mij])
            return cbin(t, p, mij-1, x);
        else
            return cbin(t, mij+1, u, x);
}
```

```
int main()
{
    int n, a[20], x, r;

    cout << "n = ";
    cin >> n;

    cout << "Introduceti " << n << " numere sortate crescator:" << endl;
    for(int i = 0; i < n; i++)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }

    cout << "x = ";
    cin >> x;

    r = cbin(a, 0, n-1, x);

    if(r != -1)
        cout << "Valoarea " << x << " apare pe pozitia " << r << endl;
    else
        cout << "Valoarea " << x << " nu apare in tablou" << endl;


    return 0;
}
```

Notăm cu $T(n)$ numărul de apeluri (recursive) ale funcției `int cbin(...)` necesare pentru a rezolva problema. Valoarea lui $T(n)$ se calculează folosind următoarea relație de recurență:

$$T(n) = \begin{cases} 1 + T\left(\left[\frac{n}{2}\right]\right), & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n \leq 1 \end{cases}$$

Presupunem faptul că $n = 2^k \Longrightarrow T(n) = T(2^k) = T(2^{k-1}) + 1 = T(2^{k-2}) + 1 + 1 = T(2^{k-3}) + 1 + 1 + 1 = \cdots = \underbrace{T(2^0)}_{1} + \underbrace{1 + 1 + \cdots + 1}_{k \text{ ori}} = 1 + k = 1 + \log_2 n$ => complexitate algoritmului este $\mathcal{O}(1 + \log_2 n) \approx \mathcal{O}(\log_2 n)$.

Atenție, doar operația de căutare binară are complexitatea $\mathcal{O}(\log_2 n)$!!! Întregul program/algoritm are complexitatea $\mathcal{O}(n + \log_2 n) \approx \mathcal{O}(n)$ din cauza citirii datelor de intrare cu complexitatea $\mathcal{O}(n)$!!!

Sortarea unui tablou pentru a aplica o căutare binară:

$$\underbrace{\mathcal{O}(n)}_{\substack{citirea\ datelor\\ de\ intrare}} + \underbrace{\mathcal{O}(n\log_2 n)}_{\substack{sortarea\\ tabloului}} + \underbrace{\mathcal{O}(\log_2 n)}_{\substack{căutarea\\ binară}} \approx \mathcal{O}(n\log_2 n)$$

Costul mediu al operației de căutare:

$$\frac{\mathcal{O}(n\log_2 n)}{1} = \mathcal{O}(n\log_2 n) > \underbrace{\mathcal{O}(n)}_{căutare\ liniară} \Rightarrow algoritm\ ineficient!!!$$

Sortarea unui tablou pentru a aplica cel puțin n căutări binare:

$$\underbrace{\mathcal{O}(n)}_{\substack{citirea\ datelor\\ de\ intrare}} + \underbrace{\mathcal{O}(n\log_2 n)}_{\substack{sortarea\\ tabloului}} + \underbrace{\mathcal{O}(n\log_2 n)}_{\substack{n\ căutări\\ binare}} \approx \mathcal{O}(n + 2n\log_2 n) \approx \mathcal{O}(n\log_2 n)$$

Costul mediu al unei operații de căutare:

$$\frac{\mathcal{O}(n\log_2 n)}{n} = \mathcal{O}(\log_2 n) \ll \underbrace{\mathcal{O}(n^2)}_{\substack{n\ căutări\\ liniare}} \Rightarrow algoritm\ foarte\ eficient!!!$$

Presupunem că avem $n = 10^6$ numere sortate, atunci:

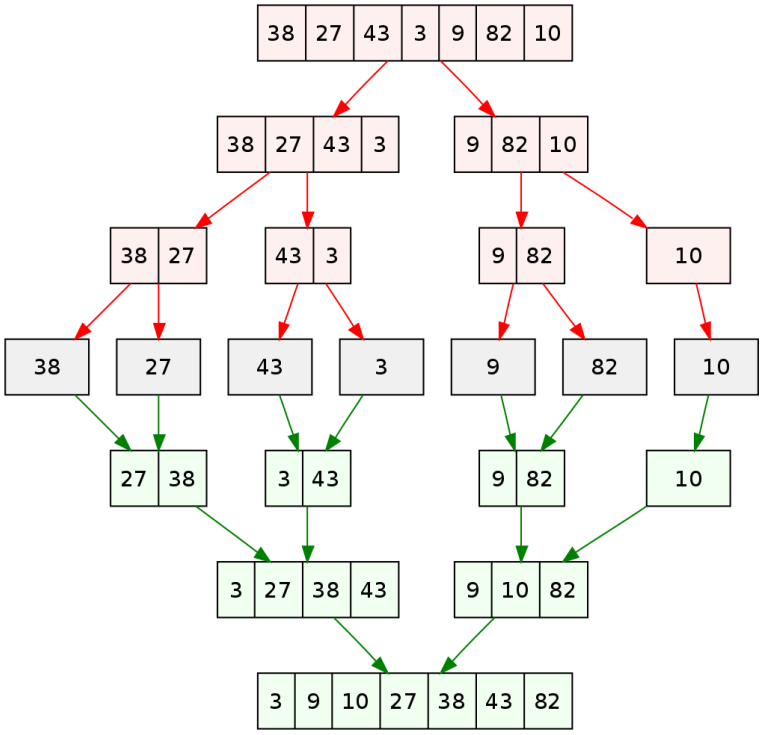
- o căutare liniară va efectua cel mult $c = 10^6$ comparări
- o căutare binară va efectua cel mult $c = \log_2 10^6 = 6\log_2 10 \approx 20$ comparări

3. Sortarea prin interclasare (Mergesort)

Sortarea prin interclasare utilizează tehnica de programare Divide et Impera pentru a sorta crescător un tablou unidimensional de numere, astfel:

- se împarte secvența curentă $t[p], \dots, t[u]$, în mod repetat, în două secvențe $t[p], \dots, t[m]$ și $t[m+1], \dots, t[u]$ până când se ajunge la secvențe implicit sortate, adică secvențe de lungime 1;
- în sens invers, se sortează secvența $t[p], \dots, t[u]$ interclasând cele două secvențe în care a fost descompusă, respectiv $t[p], \dots, t[m]$ și $t[m+1], \dots, t[u]$, și care au fost deja sortate la un pas anterior.

O reprezentară grafică a modului în care rulează această metodă de sortare se poate observa în următoarea imagine (sursa: https://en.wikipedia.org/wiki/Merge_algorithm):



Începem prezentarea detaliată a acestei metodei de sortare reamintind faptul că interclasarea este un algoritm care permite obținerea unui tablou sortat crescător din două tablouri care sunt, de asemenea, sortate crescător. Considerând dimensiunile tablourilor care vor fi interclasate ca fiind egale cu m și n, complexitatea algoritmului de interclasare este $\mathcal{O}(m+n)$.

În cazul sortării prin interclasare, se vor interclasa secvențele $t[p], \dots, t[m]$ și $t[m+1], \dots, t[u]$ într-un tablou aux alocat dinamic de lungime $u-p+1$, iar la sfârșit elementele acestuia se vor copia în secvența $t[p], \dots, t[u]$:

```
void interclasare(int t[], int p, int m, int u)
{
    //parcurgem secventa t[p],...,t[m] cu variabila i,
    //secventa t[m+1],...,t[u] cu variabila j,
    //iar tabloul aux cu variabila k
    int i = p, j = m + 1, k = 0;

    int *aux = new int[u - p + 1];

    //cat timp nu am terminat de parcurs niciuna dintre cele
    //doua secvente, copiem in aux[k] minimul dintre
    //elementele curente t[i] si t[j]
    while(i <= m && j <= u)
        if(t[i] <= t[j])
            aux[k++] = t[i++];
        else
            aux[k++] = t[j++];
```

```
    //daca au mai ramas in prima secventa elemente neparcurse,
    //le copiem in tabloul aux
    while(i <= m)
        aux[k++] = t[i++];

    //daca au mai ramas in a doua secventa elemente neparcurse,
    //le copiem in tabloul aux
    while(j <= u)
        aux[k++] = t[j++];

    //copiem elementele tabloului aux in secventa t[p],...,t[u]
    k = 0;
    for(i = p; i <= u; i++)
        t[i] = aux[k++];

    delete []aux;
}
```

Se observă uşor faptul că funcţia `interclasare` are o complexitate egală cu $\mathcal{O}(u-p+1) \leq \mathcal{O}(n)$.

Sortarea tabloului t utilizând se va efectua, folosind tehnica Divide et Impera, în cadrul următoarei funcţii:

```
void mergesort(int t[], int p, int u)
{
    if(p < u)
    {
        int m = (p + u)/2;

        mergesort(t, p, m);
        mergesort(t, m+1, u);
        interclasare(t, p, m, u);
    }
}
```

Se observă faptul că, în acest caz, funcţia `interclasare` are rolul funcţiei `Combin` din algoritmul generic Divide et Impera.

Complexitatea funcţiei `mergesort` şi, implicit, complexitatea sortării prin interclasare, se obţine rezolvând următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} 1 + T\left(\left[\frac{n}{2}\right]\right) + T\left(\left[\frac{n}{2}\right]\right) + n, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

$$= \begin{cases} 2T\left(\left[\frac{n}{2}\right]\right) + n + 1, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

Presupunem faptul că $n = 2^k \Longrightarrow T(n) = T(2^k) = 2T(2^{k-1}) + 2^k + 1 = 2[2T(2^{k-2}) + 2^{k-1} + 1] + 2^k + 1 = 2^2T(2^{k-2}) + 2^k + 2 + 2^k + 1 = 2^2T(2^{k-2}) + 2 * 2^k + 2 + 1 = 2^2[2T(2^{k-3}) + 2^{k-2} + 1] + 2 * 2^k + 2 + 1 = 2^3T(2^{k-3}) + 2^k + 2^2 + 2 * 2^k + 2 + 1 = 2^3T(2^{k-3}) + 3 * 2^k + 2^2 + 2 + 1 = \cdots = 2^k \underbrace{T(2^0)}_{1} + k * 2^k + 2^{k-1} + \cdots + 2 + 1 = k * 2^k + 2^k + 2^{k-1} + \cdots + 1 = k * 2^k + 2^{k+1} - 1 = 2^k(k+2) - 1 = n * (2 + \log_2 n) - 1 = n\log_2 n + 2n - 1$ => complexitate algoritmului este $\mathcal{O}(n\log_2 n + 2n - 1) \approx \mathcal{O}(n\log_2 n)$.