#3 Metoda Divide et Impera:

Sortarea prin interclasare. Sortarea rapidă

Obiective:

1. Exersarea metodei Divide et Impera prin implementarea şi analiza a doi algoritmi de sortare:
   - Sortarea prin interclasare.
   - Sortarea rapidă.
2. Analiza comparativă a diverselor metode de sortare.
3. Folosirea principiului interclasării pentru determinarea eficientă a numărului de inversiuni a unei permutări.
4. Verificarea înţelegerii conceptului de rezolvare Divide et Impera prin abordarea unor probleme de dificultate asemănătoare cu cele prezentate.

3.1 Sortarea prin interclasare

Problemă exemplu

Sortarea prin interclasare

Fie o mulțime de numere întregi de dimensiune `n`. Să se sorteze elementele mulțimii în ordine crescătoare folosind algoritmul sortării prin interclasare.

Datele vor fi citite dintr-un fişier de intrare care conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulțimii separate prin spaţiu. Fişierul de ieşire va conţine o singură linie cu elementele în ordine crescătoare.

Intrare	Ieşire
7 3 1 8 2 7 3 5	1 2 3 3 5 7 8

Înainte de a discuta algoritmul de sortare, să considerăm problema interclasării a doi vectori `a` şi `b` de dimensiuni `n` şi `m`, vectorii fiind în prealabil sortaţi crescător. În urma operaţiei de interclasare vom obţine un vector `c` de dimensiune `n+m` ale cărui elemente vor fi obținute din elementele vectorilor `a` şi `b`.

Pentru a interclasa doi vectori deja sortaţi îi vom parcurge în acelaşi timp folosind doi indici: `i` pentru primul vector `a` şi `j` pentru cel de-al doilea vector `b`. Vectorul `c` nu conţine iniţial nici un element. Algoritmul de completare a vectorului `c` va adăuga de fiecare dată cel mai mic element din vectorii `a` respectiv `b`, astfel:

- Dacă elementul de pe poziţia `i` din vectorul `a` este mai mic decât cel de pe poziţia `j` din vectorul `b` atunci vom adăuga `a[i]` în vectorul `c` şi vom incrementa `i` (trecem la următorul element din `a`).
- Dacă `a[i]` este mai mare decât `b[j]` atunci vom adăuga în vectorul `c` elementul `b[j]`, incrementând de această dată indicele `j`.
- Dacă cele două elemente `a[i]` şi `b[j]` sunt egale le vom adăuga pe ambele, incrementând cei doi indici `i` şi `j`.

Repetăm paşii de completare a vectorului `c` până când cei doi vectori `a` şi `b` sunt parcurşi complet. De asemenea, trebuie verificate separat situaţiile în care mai rămân elemente într-unul din vectori în cazul în care al doilea a fost parcurs în totalitate (acest lucru trebuie verificat întrucât ambii vectori sunt parcurşi în paralel). În acest caz, elementele rămase vor fi copiate în vectorul `c`.

Exemplu

Fie vectorii a={1,2,5,7,9} şi b={2,6,10,12}.

Vectorul c nu conţine iniţial nici un element, c={ }.

Vom adăuga elemente pe rând în c, conform principiului „adăugă cel mai mic element din a sau b”: c={1,2,2,5,6,7,9} până terminăm de parcurs unul dintre vectori (în cazul nostru este vorba de vectorul a). Apoi, adăugăm elementele rămase din b, rezultând în final: c={1,2,2,5,6,7,9,10,12}.

Pseudocod

```
   int[] Interclasează(int[] a,int n,int[] b,int m)
1      k ← 0 *) numărul de elemente din c
2      i ← 0 *) indicele elementului curent din a
3      j ← 0 *) indicele elementului curent din b
4      *) parcurgem simultan vectorii a,b
5      cât timp i < n şi j < m execută
6      |     dacă a[i] == b[j] atunci
7      |     |     c[k++] ← a[i++]
8      |     |     c[k++] ← b[j++]
9      |     altfel
10     |     dacă a[i] < b[j] atunci
11     |     |     c[k++] ← a[i++]
12     |     |altfel c[k++] ← b[j++]
13     |     |■
14     |     ■
15     ■
16     *) vectorul a nu a fost parcurs complet?
17     cât timp i < n execută
18     |     c[k++] ← a[i++]
19     ■
20     *) vectorul b nu a fost parcurs complet?
21     cât timp j < m execută
22     |     c[k++] ← b[j++]
23     ■
24     întoarce c
25 sf.procedură
```

Complexitatea procedurii Interclasează este Θ(n+m) întrucât ambii vectori a şi b sunt parcurşi complet în ciclul din liniile 5-15 respectiv în ciclurile 17-19 şi 21-23.

Revenind la problema sortării, apelăm la principiile Divide et Impera. Considerăm iniţial problema sortării şirului de numere aflat între indicii `0` şi `n-1` (i.e., întreg şirul) şi încercăm să reducem această problemă de dimensiune `n` la două subprobleme de dimensiune `n/2`: două subşiruri delimitate de perechile de indici `0,(n-1)/2` respectiv `(n-1)/2+1,n-1`. Dacă reuşim să rezolvăm cele două subprobleme atunci vom putea apela procedura de interclasare care ne va construi din cele două subşiruri de dimensiuni `n/2` şirul de dimensiune `n` sortat. Pentru a sorta subşirurile vom aplica succesiv etapa divide împărţindu-le până când ajungem la subşiruri de dimensiuni mici (`n=1` sau `n=2`) care pot fi sortate imediat prin cel mult o comparație și o interschimbare.

Algoritmul de sortare prin interclasare este prezentat în continuare. Parametrii procedurii `SortareInterclasare` sunt şirul a respectiv indicii `p`, `q` între care dorim să realizăm sortarea. Vom apela procedura `SortareInterclasare(a,0,n-1)`.

Pseudocod

```
procedura SortareInterclasare(int[] a, int p, int q)
    dacă p < q atunci
        *) etapa divide
        m ← (p + q) / 2

        *) etapa impera: rezolvă subprobleme
        SorteazaInterclasare(a, p, m)
        SorteazaInterclasare(a, m + 1, q)

        *) etapa impera: interclasează
        *) subşirurile [p..m] şi [m+1..q]
        int[] a'← new int[m-p+1]
        pentru i ← p,m execută a'[i-p] ← a[i]
        int[] b'← new int[q-m]
        pentru i ← m+1,q execută b'[i-m-1]←a[i]
        int[] c ←Interclasează(a',m-p+1,b',q-m)

        *) pune rezultatul în a[p..q]
        pentru i ← p,q execută a[p] ← c[i-p]
    ■
sf.procedură
```

Timpul de execuţie al algoritmului de sortare prin interclasare poate fi scris folosind recurenţa specifică algoritmilor Divide et Impera:

$$T(n)=\begin{cases}\Theta(1) & n=1\\ 2\cdot T(n/2)+D(n)+I(n) & n>1\end{cases}$$

unde `D(n)=Θ(1)`[1] şi `I(n)=Θ(n)`[2]. Avem prin urmare:

$$T(n)=\begin{cases}\Theta(1) & n=1\\ 2\cdot T(n/2)+\Theta(n) & n>1\end{cases}$$

Important

Aplicând cazul 3 al teoremei Master pentru:
a=2, b=2, $n^{\log_b a}=n^{\log_2 2}=n$ și f(n)=Θ(n)
vom obţine timpul necesar sortării prin metoda interclasării ca fiind `T(n)=Θ(nlog(n))`.

Trebuie specificat faptul că procedura `SortareInterclasare(...)` realizează acelaşi număr de operaţii indiferent de modul de organizare a datelor de intrare. Nu putem vorbi astfel în cazul sortării prin interclasare de cazul cel mai favorabil, cel mai defavorabil sau mediu. Prin urmare, complexitatea sortării va fi `Θ(nlog(n))` chiar dacă elementele şirului sunt furnizate deja în ordine crescătoare. **De asemenea, trebuie precizat faptul că sortarea prin interclasare are cea mai bună complexitate care poate fi atinsă de un algoritm de sortare bazat pe comparaţii** (pentru o demonstrație vezi (Cormen et al., 2000) (p. 148-149)).

```
class SortarePrinInterclasare
{
    /// <summary>
    /// Sorteaza vectorul a folosind metoda sortarii prin interclasare.
    /// Complexitate: O(nlog(n)).
    /// </summary>
    public static void Sorteaza(int[] a, int left, int right)
    {
        if (left < right)
        {
            int mid = (left + right) / 2;
            Sorteaza(a, left, mid);
            Sorteaza(a, mid + 1, right);
            Interclaseaza(a, left, mid, right);
        }
    }
```

[1] Etapa divide constă în calculul mijlocului intervalului p,q: m←(p+q)/2.
[2] Etapa impera constă în interclasarea a doi vectori de dimensiune maximă n/2.

```
    /// <summary>
    /// Interclaseaza vectorii a[p..m] si a[m+1..q]
    /// si pune rezultatul in a[p..q]
    /// Complexitate: O(q-p)
    /// </summary>
    private static void Interclaseaza(
           int[] a, int left, int mid, int right)
    {
        /// va contine rezultatul interclasarii
        int[] rezultat = new int[right - left + 1];
        int k = 0;

        /// parcurge simultan vectorii a[p..m] si a[m+1..q]
        int i = left;
        int j = mid + 1;
        while (i <= mid && j <= right)
        {
            if (a[i] == a[j])
            {
                rezultat[k++] = a[i++];
                rezultat[k++] = a[j++];
            }
            else
                if (a[i] < a[j])
                    rezultat[k++] = a[i++];
                else
                    rezultat[k++] = a[j++];
        }

        /// au mai ramas elemente in a[p..m]?
        for (int t = i; t <= mid; t++)
            rezultat[k++] = a[t];

        /// au mai ramas elemente in a[m+1..q]?
        for (int t = j; t <= right; t++)
            rezultat[k++] = a[t];

        /// copie rezultatul interclasarii in a[p..q]
        for (int t = left; t <= right; t++)
            a[t] = rezultat[t - left];
    }
}
```

3.2 Sortarea rapidă

Problemă exemplu

Sortarea rapidă

Fie o mulţime de numere întregi de dimensiune `n`. Să se sorteze elementele şirului în ordine crescătoare folosind algoritmul quicksort.

Datele vor fi citite dintr-un fişier de intrare care conţine pe prima linie numărul de elemente `n`, iar pe a doua linie elementele mulţimii separate prin spaţiu. Fişierul de ieşire va conţine o singură linie cu elementele ordonate crescător.

Intrare	Ieşire
7 3 1 8 2 7 3 5	1 2 3 3 5 7 8

Principiul sortării rapide (Hoare, 1961) constă în alegerea unui element din secvenţa care trebuie sortată (fie acesta `x`), alegere ce permite împărţirea secvenţei în două:

- Subsecvenţa alcătuită din elementele mai mici decât `x`.
- Subsecvenţa alcătuită din elementele mai mari decât `x`.

Fiecare subsecvenţă este împărţită la rândul ei după acelaşi principiu până când se ajunge la subsecvenţe de un element pentru care sortarea este imediată.

Pseudocod

```
procedura QSort(int[] a, int p, int q)
     i ← p
     j ← q
     x ← a[(i+j)/2]
     repetă
          cât timp a[i] < x execută i ← i + 1
          cât timp x < a[j] execută j ← j - 1
          dacă i <= j atunci
               *) interschimbă a[i] cu a[j]
               a[i] ←→ a[j]
               i ← i + 1
               j ← j - 1

     cât timp i <= j
     dacă p < j atunci QSort(a, p, j)
     dacă i < q atunci QSort(a, i, q)
sf.procedură
```

QuickSort este un algoritm care rulează în medie foarte repede pentru date de intrare de dimensiuni mari. Complexitatea sa este liniar logaritmică[3] însă algoritmul rulează mai rapid în practică decât alţi algoritmi de sortare având aceeași complexitate (spre exemplu faţă de sortarea prin interclasare). Explicaţia constă în constanta care se ascunde în aproximarea `O(..)` care este mai mică în cazul algoritmului QuickSort. Complexitatea procedurii QSort depinde de modul de alegere a valorii `x` la fiecare pas pentru care, în implementarea prezentată, am optat pentru alegerea la mijlocul intervalului `[p,q]`. Acest lucru nu este însă obligatoriu.

Analizând complexitatea algoritmului în **cazul cel mai favorabil** (când valoarea aleasă `x` împarte subsecvenţa de sortat în două subsecvenţe de lungimi egale) obținem:

$$T(n)=\begin{cases}\Theta(1) & n=1\\ 2\cdot T(n/2)+D(n)+I(n) & n>1\end{cases}$$

unde `D(n)+I(n)=Θ(n)` reprezintă timpul necesar operaţiilor divide şi impera.

$$T(n)=\begin{cases}\Theta(1) & n=1\\ 2\cdot T(n/2)+\Theta(n) & n>1\end{cases}$$

Important

Aplicând teorema Master pentru:

a=2, b=2, $n^{\log_b a}=n^{\log_2 2}=n$ și f(n)=Θ(n), obţinem timpul de execuţie al algoritmului QSort în cazul cel mai favorabil:

`T-CF(n)=Θ(nlog(n))`

În **cazul cel mai defavorabil** valoarea `x` va împărţi subsecvenţa `[p,q]` în două subsecvenţe de lungimi extrem diferite: `1` şi `q-p`. Vom avea astfel:

$$T(n)=\begin{cases}\Theta(1) & n=1\\ T(n-1)+D(n)+I(n) & n>1\end{cases}$$

unde `D(n)+I(n)=Θ(n)`. Desfăcând recurenţa obţinem:

$$T(n)=T(n-1)+n$$
$$T(n-1)=T(n-2)+n-1$$
$$\vdots$$
$$T(n-i)=T(n-(i+1))+n-i$$
$$\vdots$$
$$T(1)=1$$

[3] Nu şi în cazul cel mai defavorabil, a se vedea analiza complexităţii.

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} = \Theta(n^2)$$

Important

Algoritmul QSort are complexitate pătratică în cazul cel mai defavorabil:

$T\text{-}CDF(n) = \Theta(n^2)$

```
class SortareaRapida
{
    /// <summary>
    /// Sorteaza vectorul a folosind algoritmul
    /// sortarii rapide.
    /// Complexitate: O(nlog(n)) in cazul mediu,
    /// O(n^2) in cazul cel mai defavorabil.
    /// </summary>
    public static void Sorteaza(int[] a, int left, int right)
    {
        int i = left;
        int j = right;
        int x = a[(i + j) / 2];
        do
        {
            /// Imparte elementele dintre indicii left si right
            /// in doua subsecvente in functie de x
            while (a[i] < x) i++;
            while (x < a[j]) j--;
            if (i <= j)
            {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                i++;
                j--;
            }
        } while (i <= j);
        /// Rezolva subproblemele
        if (left < j) Sorteaza(a, left, j);
        if (i < right) Sorteaza(a, i, right);
    }
}
```

3.3 Interclasare vs. sortare rapidă

Am arătat anterior că ambii algoritmi de sortare au în cazul mediu o complexitate liniar logaritmică, $\Theta(n\log(n))$. Pentru sortarea prin interclasare complexitatea este valabilă indiferent de modul de organizare a datelor în timp ce pentru sortarea rapidă cazul cel mai defavorabil presupune $O(n^2)$ operaţii.

Figura 3.1 prezintă timpul de execuţie al celor două metode de sortare pentru date generate aleator de dimensiune $n \leq 100,000$. Se observă un timp mai redus pentru algoritmul quick sort cu toate că ambele grafice urmează dependenţa liniar logaritmică faţă de n. Pentru a completa comparaţia, figura 3.2 prezintă şi performanţa algoritmului de sortare prin inserţie pe care am evaluat-o într-un capitol anterior ca fiind $\Theta(n^2)$. De asemenea, figura 3.3 ilustrează şi timpul de execuţie al sortării prin metoda bulelor, tot de complexitate $\Theta(n^2)$, metodă însă mult mai costisitoare în practică.

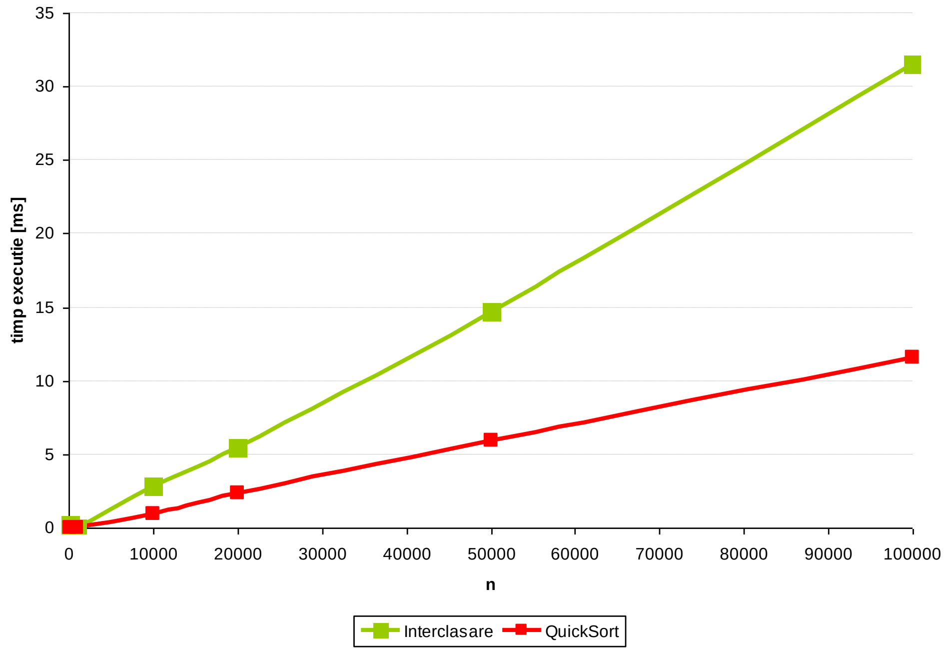

Figura 3.1 Timpul de execuţie exprimat în milisecunde pentru implementările SortareInterclasare şi QSort rulate pentru date generate aleator. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

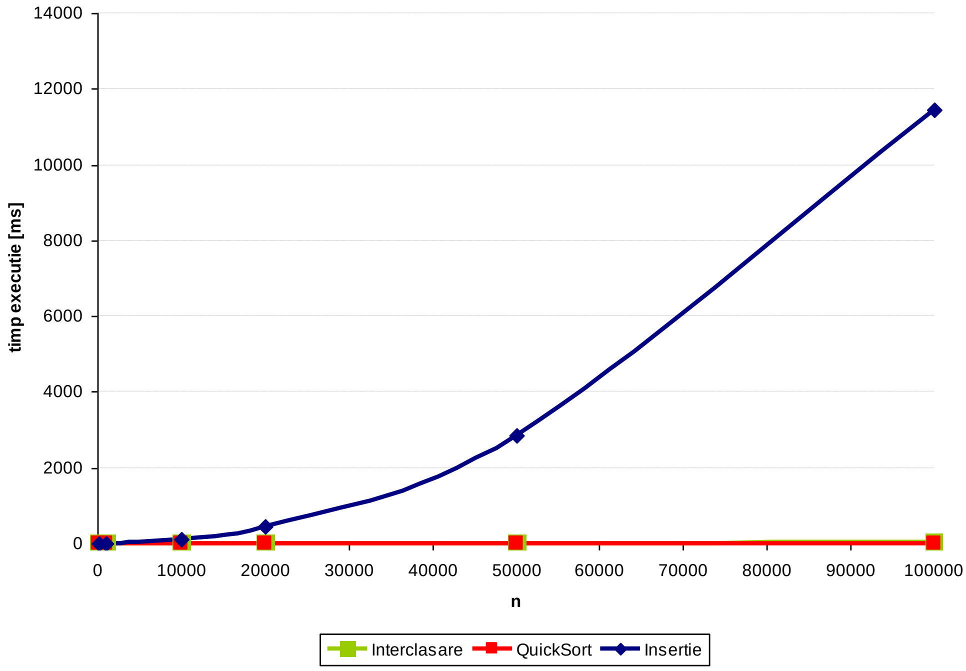

Figura 3.2 Timpul de execuţie exprimat în milisecunde pentru implementările SortareInterclasare, QSort și SortareInserție rulate pentru date generate aleator. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

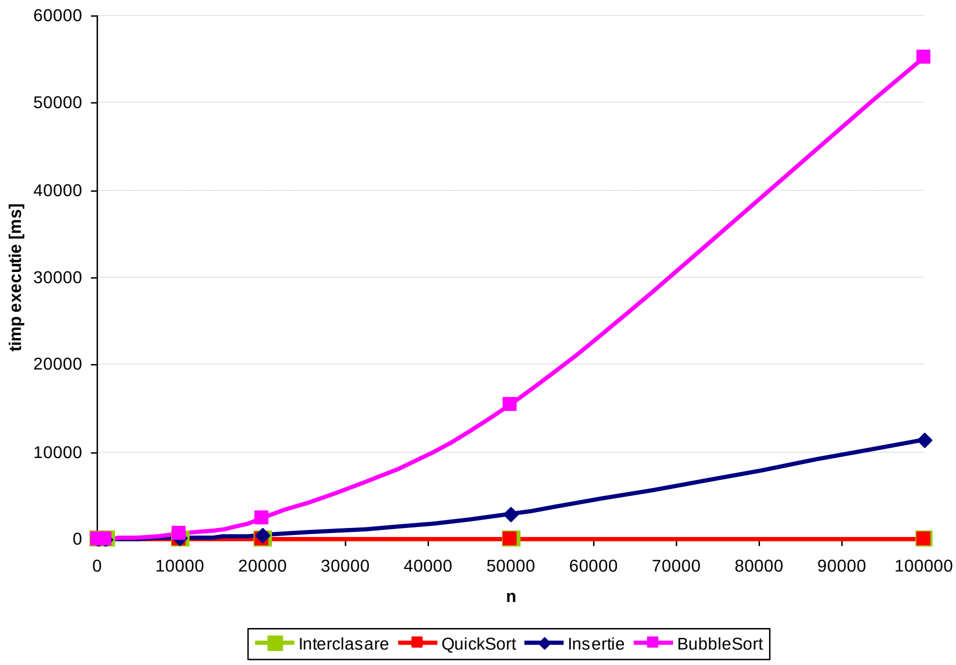

Figura 3.3 Timpul de execuţie exprimat în milisecunde pentru implementările SortareInterclasare, Qsort, SortareInserție și SortareaBulelor rulate pentru date generate aleator. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

3.4 Inversiunile unei permutări

Problemă exemplu

Inversiuni

Pentru o permutare a mulţimii `{1,2,...n}` reprezentată sub forma unui vector de numere întregi `a`, se numeşte inversiune perechea `(i,j)` cu proprietatea că `i<j` şi `a[i]>a[j]`. Dată fiind o permutare a mulţimii `{1,2,...n}`, determinaţi numărul de inversiuni.

Datele vor fi citite dintr-un fişier cu următorul format: prima linie va conţine numărul `n` de elemente, iar a doua linie elementele permutării separate printr-un spaţiu. Fişierul de ieşire va conţine pe prima linie numărul `r` de inversiuni, iar următoarele `r` linii vor conţine fiecare inversiune specificată prin perechea de indici `i` şi `j`.

Intrare	Ieşire
4 3 2 4 1	4 0 1 0 3 1 3 2 3

Putem imagina un algoritm simplu care va testa toate perechile `i<j` pentru verificarea condiţiei `a[i]>a[j]`. Întrucât există $n\times(n-1)/2$ perechi distincte care trebuie verificate rezultă o complexitate pătratică $\Theta(n^2)$. Algoritmul este prezentat în continuare.

Pseudocod

```
int NrInversiuni(int[] a, int n)
    nr ← 0
    pentru i ← 0, n-2 execută
        pentru j ← i+1, n-1 execută
            *) avem inversiune?
            dacă a[i] > a[j] atunci
                nr ← nr + 1
            ■
        ■
    ■
    întoarce nr
sf.procedură
```

Putem obţine un algoritm de o complexitate mai bună dacă adoptăm aceeaşi idee folosită în cazul sortării prin interclasare: împărţim şirul de dimensiune n în două subşiruri de dimensiune $n/2$, calculăm separat numărul de inversiuni din primul şir (nr_1), numărul de inversiuni din al doilea şir (nr_2), respectiv numărăm câte elemente din primul şir sunt mai mari decât elemente prezente în cel de-al doilea ($nr_{1,2}$). Numărul de inversiuni al problemei de dimensiune n va fi $nr_1+nr_2+nr_{1,2}$.

Pseudocod

```
int NrInversiuni-2(int[] a, int p, int q)
    dacă p == q atunci
        întoarce 0
    altfel
        *) etapa divide
        m ← (p + q) / 2
        *) numără inversiunile din cele două
        *) subşiruri
        nr1 ← NrInversiuni-2(a, p, m)
        nr2 ← NrInversiuni-2(a, m+1, q)
        *) etapa impera: interclasează
        nr12 ← Interclasează(a, p, m, q)
        întoarce nr1 + nr2 + nr12
    ■
sf.procedură
```

Procedura `Interclasează(..)` va efectua în plus o numărare a inversiunilor în care primul element aparţine primului subşir iar al doilea celui de-al doilea subşir.

```
class Inversiuni
{
    /// <summary>
    /// Calculeaza numarul de inversiuni pentru o permutare.
    /// Complexitate O(n^2).
    /// </summary>
    public static int NrInversiuni_1(int[] a)
    {
        int nr = 0;
        for (int i = 0; i < a.Length; i++)
            for (int j = i + 1; j < a.Length; j++)
                if (a[i] > a[j])
                    nr++;
        return nr;
    }
```

```
    /// <summary>
    /// Calculeaza numarul de inversiuni pentru o permutare
    /// folosind interclasarea.
    /// Complexitate O(nlog(n)).
    /// </summary>
    public static int NrInversiuni_2(int[] a, int left, int right)
    {
        if (left < right)
        {
            int mid = (left + right) / 2;
            int nr1 = NrInversiuni_2(a, left, mid);
            int nr2 = NrInversiuni_2(a, mid + 1, right);
            int nr12 = Interclaseaza(a, left, mid, right);
            return nr1 + nr2 + nr12;
        }
        return 0;
    }

    /// <summary>
    /// Interclaseaza doua subsiruri ale sirului a
    /// si calculeaza numarul de inversiuni dintre
    /// perechi de elemente din primul si al doilea sir.
    /// </summary>
    private static int Interclaseaza(
           int[] a, int left, int mid, int right)
    {
        // contine numarul de inversiuni
        int nr = 0;
        // contine rezultatul interclasarii
        int[] rezultat = new int[right - left + 1];
        int k = 0;
        int i = left;
        int j = mid + 1;
        while (i <= mid && j <= right)
        {
            if (a[i] < a[j])
            {
                rezultat[k++] = a[i++];
                // a[i] este mai mare decat toate
                // elementele pana la a[j] exclusiv
                nr += j - (mid + 1);
            }
            else rezultat[k++] = a[j++];
        }
        for (int t = i; t <= mid; t++)
        {
            rezultat[k++] = a[t];
```

```
                // a[t] este mai mare decat toate
                // elementele din a[mid+1]..a[right]
                nr += right - (mid + 1) + 1;
            }
            for (int t = j; t <= right; t++)
                rezultat[k++] = a[t];
            for (int t = left; t <= right; t++)
                a[t] = rezultat[t - left];
            return nr;
        }
```

Pentru determinarea expresiei complexității procedurii `NrInversiuni-2(...)` vom folosi relaţia de recurenţă tipică Divide et Impera:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2 \cdot T(n/2) + D(n) + I(n) & n > 1 \end{cases}$$

unde $D(n)=\Theta(1)$ şi $I(n)=\Theta(n)$. Avem deci:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2 \cdot T(n/2) + \Theta(n) & n > 1 \end{cases}$$

Aplicând cazul 3 al teoremei Master pentru a=2, b=2, $n^{\log_b a} = n^{\log_2 2} = n$ și f(n)=Θ(n), obţinem $T(n)=\Theta(n\log(n))$, o complexitate mai bună decât cea pătratică $\Theta(n^2)$ a algoritmului iniţial.

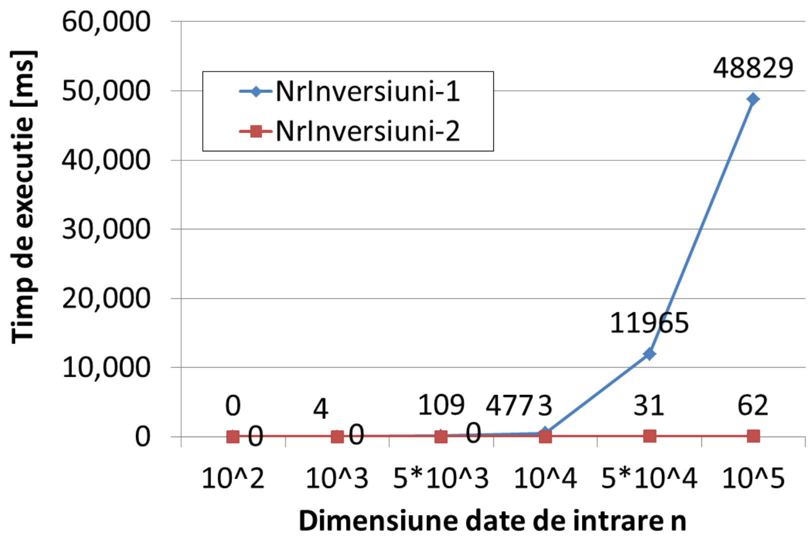

Figura 3.4 Timpul de execuţie exprimat în milisecunde pentru cele două variante de rezolvare pentru problema determinării numărului de inversiuni. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

3.5 Probleme propuse

Problema #1

Operaţii cu mulţimi

Fie două mulţimi `a` și `b` de numere întregi de dimensiune `n` respectiv `m`. Să se determine reuniunea, intersecţia respectiv diferenţa dintre cele două mulţimi.

Fişierul de intrare va conţine pe prima linie numărul `n` de elemente din mulţime urmat pe linia următoare de elementele mulţimii separate printr-un spaţiu. Urmează mulţimea `b` reprezentată similar. Elementele mulţimilor sunt sortate crescător. Fişierul de ieşire conţine câte o linie cu rezultatul celor trei operaţii.

Intrare	Ieşire
5 1 2 3 7 8 4 1 8 9 12	A U B: 1 2 3 7 8 9 12 A ∩ B: 1 8 A – B: 2 3 7

Care este complexitatea fiecărei operaţii?

Problema #2

QuickSort aleator

Fie o mulțime de numere întregi de dimensiune `n`. Să se sorteze elementele mulțimii în ordine crescătoare folosind metoda quick sort însă alegerea pivotului `x` din intervalul `p..q` se va face aleator.

Datele vor fi citite dintr-un fişier de intrare care conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulțimii separate prin spaţiu. Fişierul de ieşire va conţine o singură linie cu elementele şirului ordonate crescător separate printr-un spaţiu.

Intrare	Ieşire
7 3 1 8 2 7 3 5	1 2 3 3 5 7 8

Care este complexitatea algoritmului QuickSort aleator?

Depinde timpul de execuţie de modul de organizare a datelor de intrare?

Problema #3

Elementul sumă (2)

Fie două mulțimi a și b de numere întregi de dimensiune n şi fie un număr x. Să se determine dacă există două elemente din cele două mulțimi a căror sumă să fie x.

Fişierul de intrare va conţine pe prima linie numărul de elemente din ale celor două mulțimi iar următoarele două linii elementele separate printr-un spaţiu. A patra linie conţine valoarea x. Fişierul de ieşire va avea o singură linie conţinând cele două elemente care adunate dau suma x, respectiv -1 dacă nu există soluţie.

Intrare	**Ieşire**
6 7 1 5 3 2 8 2 8 3 1 9 8 12	3 9

Care este complexitatea algoritmului propus?