Lucrare Științifică

Metode și Practici în Informatică

Subiect: Compararea teoretică și experimentală a unor metode de sortare

Student: Costică – Vlad Luchian

Profesor: Adrian Crăciun

Timișoara 2018

"We dont have better algorithms, we just have more data"

Scopul lucrarii:

Lucrarea "Compararea teoretică și experimentală a unor metode de sortare" are drept scop studiul și analiza printr-o comparație teoretică și experimentală a metodelor de sortare, în special cele studiate precum: Bubble sort, Insertion sort, Selection sort, QuickSort, MergeSort și CountingSort. Întru studierea aspectelor teoretice, am prezentat o scurtă descriere a fiecărei sortări, principiile, ordinul de complexitate. Pentru partea experimentală, am executat funcții sortare în Python, de 10^n elemente, pentru n=1 până la 4(pentru cele de complexitate n^2) și pentru n până la 6 sau 7 pentru cele care am putut scoate un timp favorabil(<10 minute). Pentru ca rezultatele să fie cât mai acurate, am repetat experimentul de 10 ori pentru fiecare metodă și număr de elemente. Elementele au fost generate aleator în același program, cu valori cuprinse de la 1 la numărul de valori al tabloului. Rezultatele experimentelor aferente sunt reprezentate în grafice și tablouri. Alte scopuri ale acestei lucrări sunt: să exploreze, să verifice, să valideze, să descrie aspectul de timp de compilare sau/și numărul de operații(comparări și interschimbări) efectuate per algoritm, printr-un studiu empiric.

Obiectiv:

Demonstrarea cu timpi și date concrete a diferențelor între algoritmii usuali de sortare.

Compararea teoretică și experimentală a metodelor de sortare menționate mai sus.

Problematica:

Problema sortării este una trivială, însă aceasta înconjoară societatea și este o parte a vieții cotidiene. Astfel, din punct de vedere a programării, în special sortării unor elemente în dependență de valoarea acestora, am efectuat o lucrare în care am evidențiat diferențele dar și similitudinile algoritmilor. Sortarea este una dintre cele mai studiate probleme în informatică. Literalmente, zeci de algoritmi sunt cunoscuți, fiecare având un avantaj în comparație cu altul. Pe piață sunt multe metode de aranjare a unor elemente, însă la baza comparării acestora stau principiile timpului de execuție(complexitatea timpului) și cantității de memorie necesară sortării. Dat fiind faptul că este nevoie de o efiecentizare a metodelor de sortare, în această lucrare s-au analizat doar cele care au avut gradul de complexitate de maxim $O(n^2)$.

*Precizare: Toate sortările și descrierea implementării lor a fost făcută din punct de vedere al aranjării crescătoare.

Algoritmi de sortare:

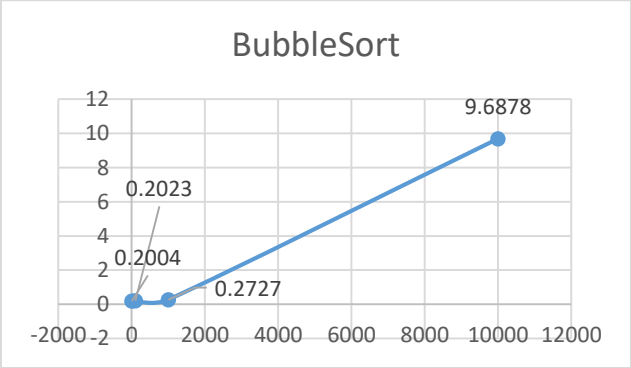
- **Bubble Sort**

BubbleSort este un algoritm popular, însă ineficient. Acesta are complexitatea în cazul nefavorabil și mediu de $O(n^2)$, iar de aceea nu este folosit la sortarea unui număr mare de elemente, decât doar dacă sunt puține elemente care nu sunt în ordinea dorită.

Principiul: Începe de la primul element al tabloului și interschimbă repetat elementele alăturate care nu sunt in ordine.

Algoritm Python:

```
def bubble(x):
    n=len(x)-1
    for i in range(n):
        for j in range(1,n+1):
            if x[i]>x[j]:
                x[i],x[j]=x[j],x[i]
    return x
```



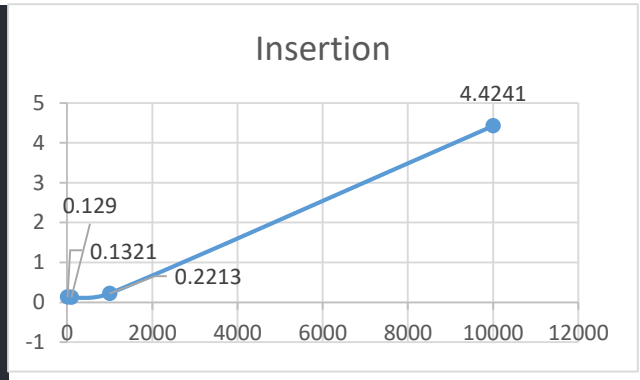
- **Insertion Sort**

Algoritmul de sortare prin inserție este unul eficient când vine vorba de un număr mic de valori. O altă metodă care folosește acest algoritm este Shell sort.

Principiul: Începem cu o "parte stângă" liberă(când de exemplu avem în mâna dreapta câteva cărți iar în stângă le aranjăm). Valorile preluate din input sunt așezate într-un tablou(array). Astfel preluăm câte un element(key/valoarea auxiliara)din tablou și o aranjăm în poziția corectă în partea I din tablou. Pentru a găsi poziția corectă a elementului, comparăm valoarea cu cele deja existente în tabel, de la stânga la dreapta. Mereu,elementele din partea stângă a tabloului sunt sortate, iar cele din partea dreaptă nu.

Algoritm Python:

```
def insertion(x):
    for i in range(1,len(x)):
        aux=x[i]
        while j>=0 and aux<x[j]:
            x[j+1]=x[j]
            j-=1
        x[j+1]=aux
    return x
```



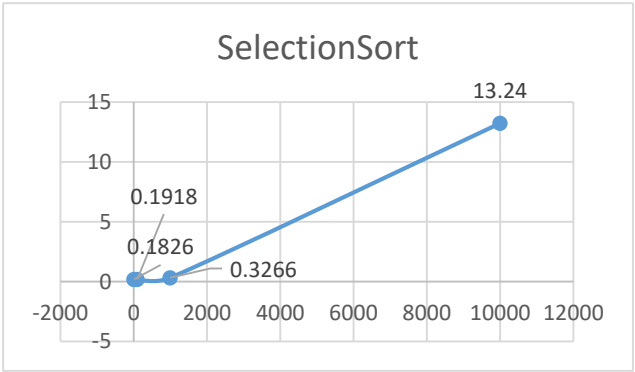
- **Selection sort**

Sortarea prin selecție are o complexitate $O(n^2)$, ceea ce îl face ineficient pentru un număr mare de elemente. El se remarcă însă prin simplicitatea sa.

Principiul: Tabloul este la fel "împărțit" în două subtablouri; algoritmul găsește valoarea minimă din tabel, o interschimbă cu prima poziție, și repetă acest pas până la finalizarea listei. O altă metodă mai eficientă care folosește același principiu este Shaker Sort, care în aceeași iterație plasează minimul la începutul tabloului și maximul la sfârșitul ei.

Algoritm Python:

```
def selection(x):
    for i in range(len(x)):
        k=i
        for j in range(i,len(x)):
            if x[j]<x[k]:
                k=j
            if i!=k:
                x[i],x[k]=x[k],x[i]
    return x
```

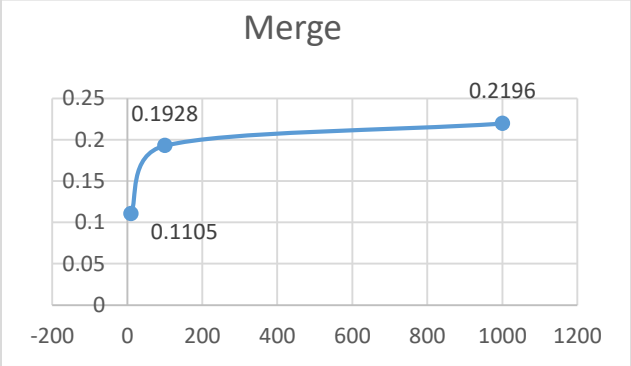


- **MergeSort** – sortarea utilizând principiul "Divide et Impera"
  - Divizarea problemei într-un număr de subprobleme.
  - Stăpânirea subproblemelor rezolvându-le recursiv. Dacă dimensiunea subproblemei este rezonabil de mica, rezolvarea acesteia se face într-o maniera simplă.
  - Combinarea soluțiilor subproblemelor în soluția problemei originale.

Algoritmul, în cazul favorabil are o complexitate de O(n) iar în cel nefavorabil, complexitatea sa este de O(n*log n)

Algoritm Python:

```
def merge(x,s,d):
    if s<d:
        m=(s+d)//2
        x=merge(x,s,m)
        x=merge(x,m+1,d)
        x=interclasare(x,s,m,d)
    return x
def interclasare(x,s,m,d):
    c=[]
    i=s
    j=m+1
    while i<=m and j<=d:
        if x[i]<=x[j]:
            c.append(x[i])
            i+=1
        else:
            c.append(x[j])
            j+=1
        while i<=m:
            c.append(x[i])
            i+=1
        while j<=d:
            c.append(x[j])
            j+=1
        for i in range(len(c)):
            x[i+s]=c[i]
        return x
```



- **QuickSort**

Sortarea rapidă(QuickSort), ca și Sortarea prin Interclasare(MergeSort) aplică "Divide et Impera". Pentru cazul nefavorabil are o complexitate de $O(n^2)$ – în cazul în care pivotul ales nu este o alegere potrivită(nu împarte tabloul în 2 subtablouri mai mult sau mai puțin egale); pentru cazul favorabil și mediu complexitatea este O(n log n).

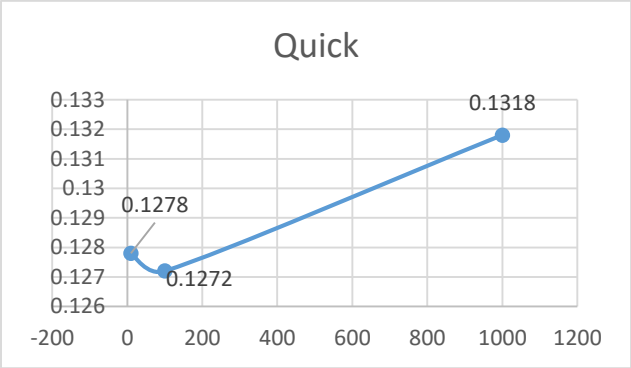
Principiul "Divide et Impera", paradigmă explicată anterior:

- Divizarea: partiționarea tabloului în 2 subtablouri, astfel încât toate elementele din A[s..p-1] sunt mai mici sau egale cu elemental A[p], iar cele din A[q+1..d] sunt mai mari sau egale. Elementul A[p] este numit pivot.
- Conquer: Sortarea celor două subtablouri recursive, folosind aceeași metodă.
- Combinarea: Având în vedere că subtablourile sunt deja sortate, la combinare nu mai este nevoie de o rearanjare. Astfel, tabloul A[s..d] este sortat.

*Observație: Acest algoritm are nevoie și de un spațiu de memorie O(log n)

Algoritm Python:

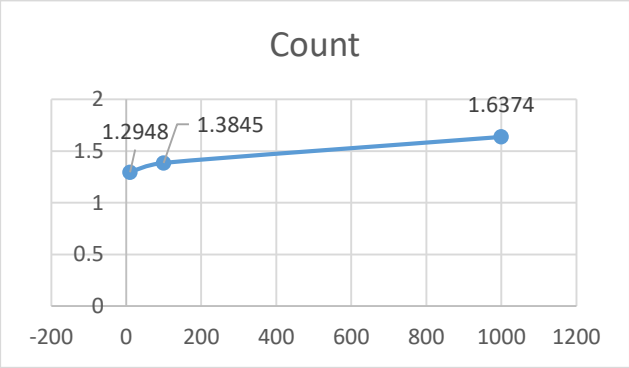
```
def quick(x,s,d):
    if s<d:
        q=partitie(x,s,d)
        x=quick(x,s,q)
        x=quick(x,q+1,d)
    return x
def partitie(x,s,d):
    v=x[s]
    i=s-1
    j=d+1
    while i<j:
        comp+=1
        i+=1
        while x[i]<v:
            i+=1
        j-=1
        while x[j]>v:
            j-=1
        if i<j:
            x[i],x[j]=x[j],x[i]
    return j
```



- **Counting sort**(Instabil)

Sortarea prin numărare este o metodă instabilă de sortare(elementele egale din tabloul original ar putea să nu fie în aceeași ordine în cel final). Acesta are complexitatea O(n+k) unde n este numărul de elemente din tablou, iar k este valorea maximă din tablou.

Principiul: Se parcurge tabloul pentru găsirea valorii maxime. Apoi se crează un tablou de k valori, iar la parcurgerea tabloului original, la fiecare valoare din acesta, se incrementează valorea indexului potrivit din cel de-al doilea tabel. După finalizare se întoarce al doilea tablou, care conține elemente cu valori diferite de 0.

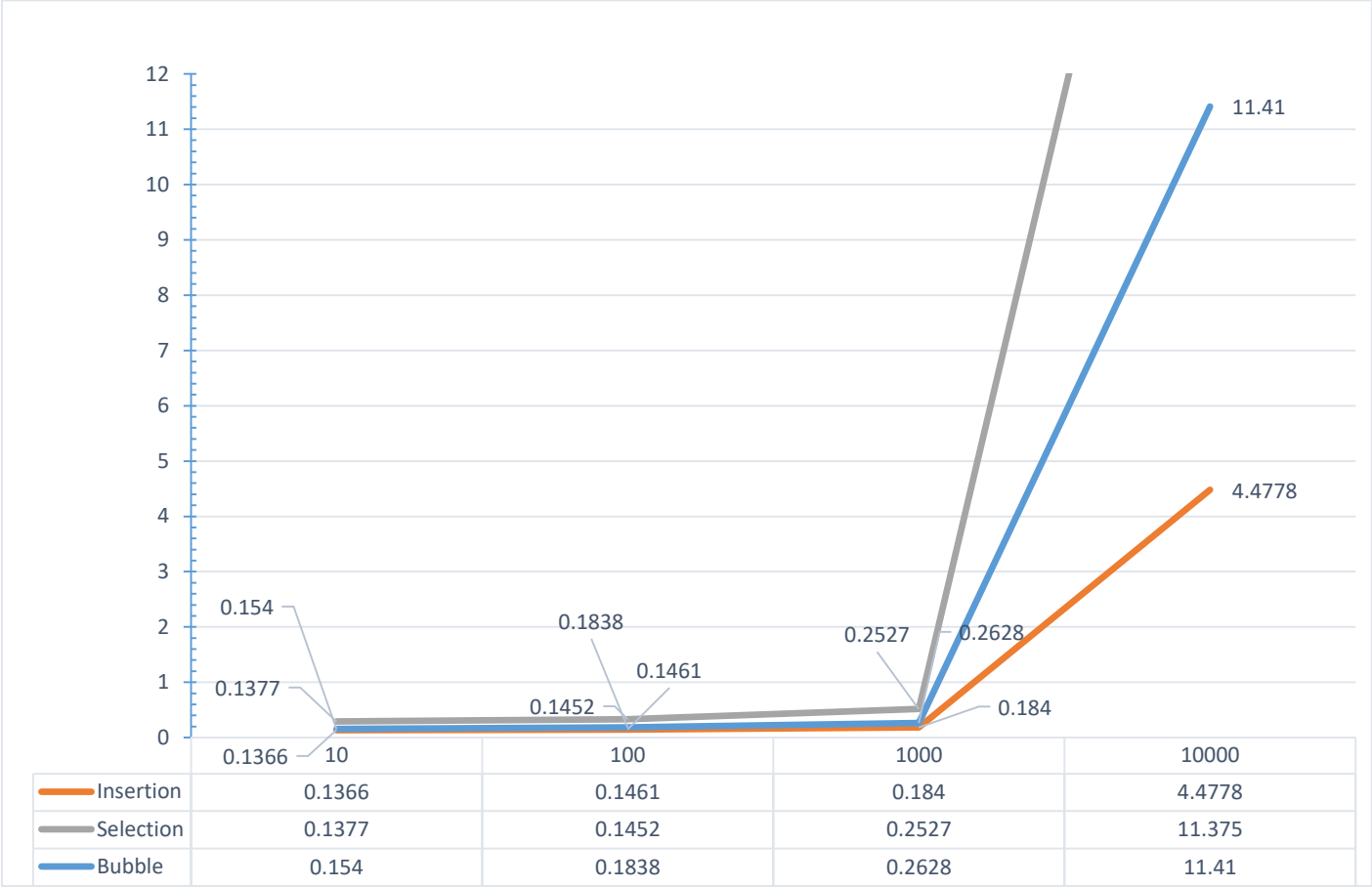


Algoritm Python:

```
def countingsort(x,m):
    f=[0]*(m+2)
    n=len(x)
    y=[0]*(n+1)
    for i in range(0,n):
        f[x[i]]=f[x[i]]+1

    for i in range(2,m+1):
        f[i]=f[i-1]+f[i]

    for i in range(n-1,-1,-1):
        y[f[x[i]]]=x[i]
        f[x[i]]=f[x[i]]-1

    for i in range(0,n):
        x[i]=y[i+1]
    return x
```

ANALIZA PERFORMANȚEI DE SORTARE:

<Grafice si tabelele>



Timișoara 2018

COMPARĂRI

Sortare / Nr. elemente	**Bubble** =(nr elemente-1)^2	**Insertion**	**Selection** =$\sum_{k=1}^{n} k$
10	81	48	55
100	9 801	4 896	5 050
1000	998 001	504 636	500 500
10000	99 980 001	94 845 100	50 005 000

INTERSCHIMBĂRI

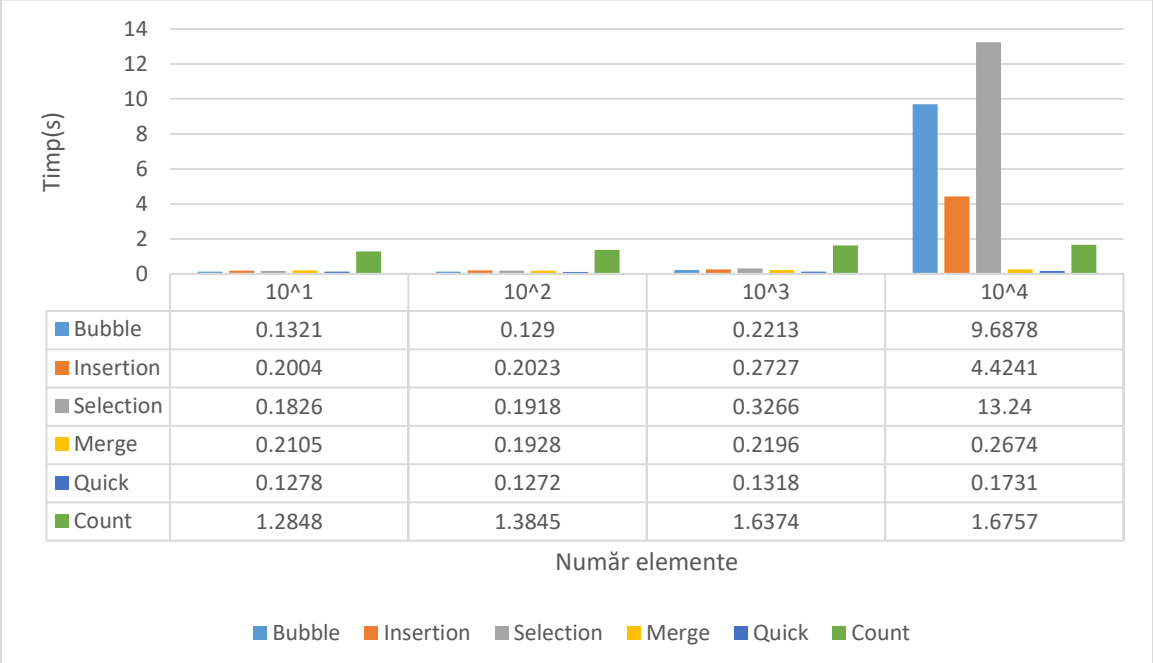
Sortare / Nr. elemente	**Bubble**	**Insertion** =(nr_elem-1)^k *	**Selection**
10	20	9	41
100	2 316	99	4 936
1000	249 440	999	499 488
10000	22 753 411	9 999	4 999 482

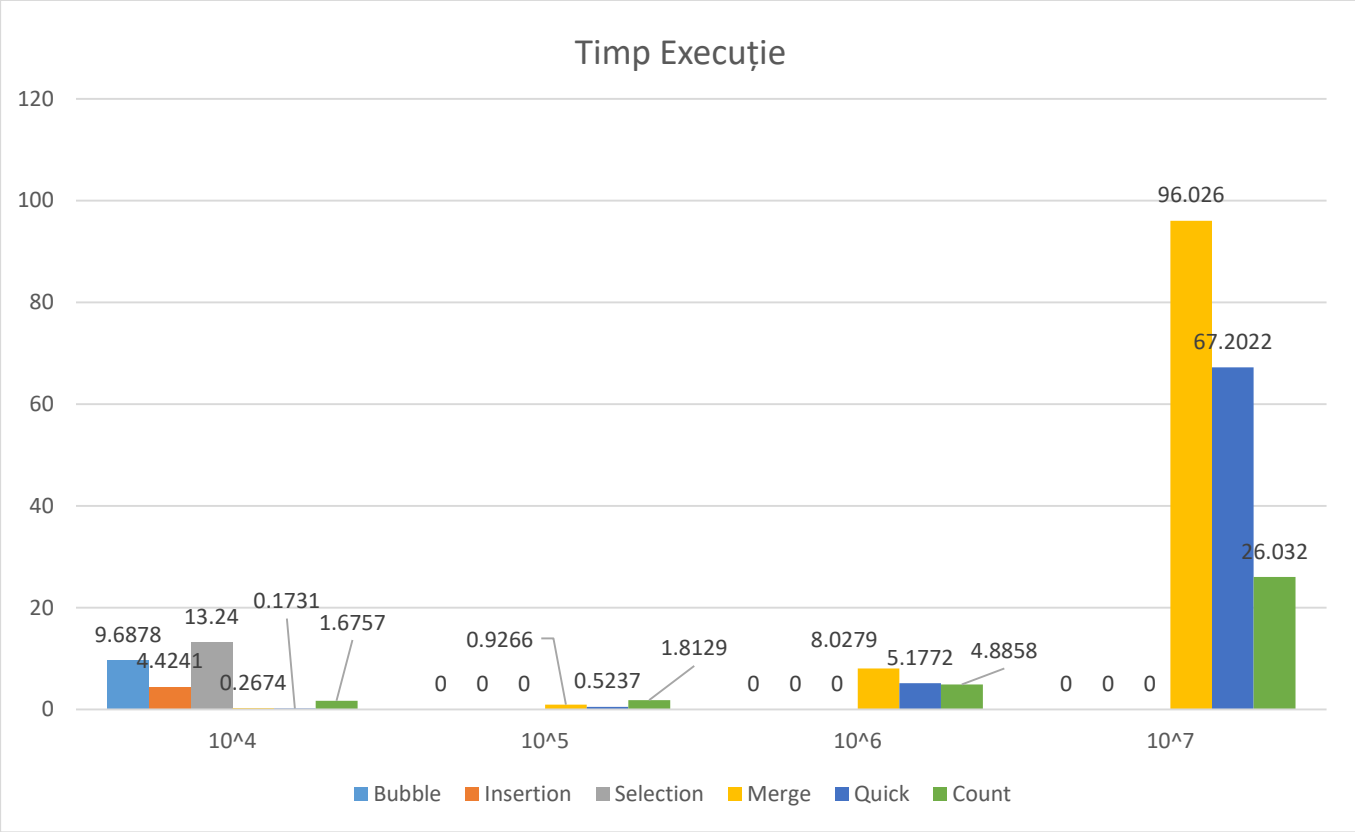
* =(nr_elem-1)^k, k>=1, unde k indică lungimea șirului și este egal cu numărul de zerouri din numărul de elemente

În urma executării de sortări de până la 10 mii elemente, considerând că fiecare operație are același cost, se observă din tabele de mai sus că:

Pentru metoda Bubble Sort, sunt executate cele mai multe operații, urmată de Selection Sort (doar până la 1000 elemente), iar apoi Insertion.

În dependență de metodă și conceptul ei, numărul de comparări sau schimbări este același pentru fiecare sortare de n elemente.





Concluzia:

În urma efectuării sortării de tablouri pentru fiecare algoritm menționat mai sus, rezultatele timpurilor demonstrează încă o dată: complexitatea își spune cuvântul când vine vorba de un număr mare de elemente. Pentru sortarea unui număr mic de elemente, toate metodele s-au descurcat în timp minim(<1s). Abia când are loc trecerea la tablouri cu 10^4 până la 10^7elemente, atunci fiecare metodă își spune cuvântul, prin rapiditate și timpi de execuție minimi.

REFERINȚE:

- International Journal for Research in Applied Science & EngineeringTechnology (IJRASET)©IJRASET 2015: All Rights are Reserved201Performance Analysis of Sorting Algorithms with C#
- The International Arab Journal of Information Technology, Vol. 7, No. 1, January 2010
- The International Journal Of Engineering And Science (IJES)
- International Journal of Scientific Engineering and Research (IJSER)
- Introduction to algorithms (3rd edition) – Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein
- The Algorithm Design Manual (2nd edition) – Steven. S. Skiena

*Pentru verificarea materialelor folosite - https://github.com/CveCt0r/MPI