O comparație teoretică și experimentală a unor metode de sortare

Eduard-Florin Hogea
Departamentul de Informatică
Facultatea de Matematică și Informatică
Universitatea de Vest Timișoara
eduard.hogea00@e-uvt.ro

Rezumat

O comparație teoretică, dar care cuprinde și o parte experimentală a metodelor de sortare studiate de mine până în prezent.

Scopul acestei lucrări este de a ajuta la alegerea algoritmului de sortare potrivit pentru fiecare situație, precizând clar diferențele, avantajele și dezavantajele. Alegerea metodei de sortare potrivite ducând la un timp de execuție mai mic și la o mai puțină memorie folosită.

În comparațiile experimentale vom folosi atât exemple simple, care să ilustreze problema și soluția, dar și exemple complexe prin care se pot obvserva în detaliu diferențele.

Cuprins

1 Introducere

Sortarea este cel mai probabil cea mai studiată problemă în domeniul informaticii. În primul rând, din cauza simplității și soluțiilor ei interesante și diverse, ea stă la baza multor aplicații ale acestui domeniu. Operația de sortare și ordonare a unor articole în funcție de diverse criterii se întâlnește foarte des în practică. Un exemplu de folosire a sortării îl reprezintă motoarele de căutare web, care folosesc astfel de algoritmi(Google, Yahoo, MSN).

Algoritmii de sortare(alături de o scurtă descriere a lor) pe care urmează să îi comparăm:

- Counting sort - Sortare prin numărare
- Radix sort - Sortare pe baza cifrelor
- Shell sorting - Sortare prin inserție cu pas variabil
- Shaker sort- Sortarea prin interschimbarea elementelor vecine
- Selection Sort- Sortarea prin selecție
- Insertion Sort- Sortarea prin inserție
- Bubble Sort- Sortarea prin comparare
- Quick Sort - Sortare prin tehnica divizării

1.1 Motivaţie

Există diferiți algoritmi de sortare, precum cei menționați anterior pe care îi vom și compara. Având o multitudine din aceștia, intervine problema alegerii.

O soluție generală pentru această problemă nu există. Acest lucru se datorează din cauza că diferite metode de sortare sunt în anumite cazuri mai eficiente decât altele din punct de vedere al timpului de execuție și al memoriei utilizate.

Aceste două lucruri reprezentând o problemă uriașă, am decis să fac această lucrare pentru a ușura în alegerea metodei potrivite.

1.2 Descriere informală a soluţiei

Pentru a alege metoda de sortare potrivită fiecărei situații, am considerat ca următorii pași sunt necesari:

1. Alegerea unui exemplu(simplu/complex)
2. Rularea algoritmului de sortare ales pe acest exemplu
3. Interpretarea rezultatelor

Pentru a avea rezultate relevante și pentru a diminua sursele de erori vom folosi același sistem și același limbaj de programare pentru toate testele.

1.3 Exemple simple ce ilustrează problema şi soluţia

Pentru a exemplifica problema eficienței ne vom raporta la urmatoarele exemple simple:

- Am generat aleator 100 de numere cuprinse intre 10 si 100 pentru a demonstra diferenta dintre timpul de executie al algoritmilor de sortare.

Pentru a evidenția diferențele, vom folosi două metode de sortare diferite. Acestea sunt Bubble Sort [1] (care este considerat in majoritatea cazurilor a nu fi atat de eficient) și Quicksort [1] (considerat a fi unul dintre cei mai rapizi algoritmi de sortare).

Folosind un exemplu de doar 100 de numere, diferența dintre timpii de execuție(în secunde) nu este una foarte mare. Astfel am obținut:

- pentru Bubble Sort - 0.003002166748046875
- pentru Quicksort - 0.001998424530029297

Aceste rezultate ilustrează problema eficienței algoritmilor de sortare. Cu toate acestea, pentru a avea o mai buna ilustrare vom folosi un exemplu complex.

1.4 Exemplu complex

În următoarele exemple vom putea observa diferențele dintre algoritmii de sortare. Vom folosi un exemplu mai mare de numere generate aleator pentru a vedea diferențele între timpii de execuție.

Pentru 10000 de numere, generate aleator între 1 si 100000:

- pentru Bubble Sort - 14.44399
- pentru Quicksort - 0.049
- pentru Selection Sort - 11.19399
- pentru Insertion Sort - 7.289
- pentru Shaker Sort - 12.725
- pnetru Radix Sort - 0.066

1.5 Declaraţie de originalitate

Contribuția mea consta în alegerea unui set de date generat aleator și rularea algoritmilor de sortare specificați pe acest set de date. Pentru a avea rezultate relevante am folosit același limbaj de programare pentru toți algoritmii și am folosit un exemplu foarte mare de date de intrare.

1.6 Instrucţiuni de citire

Lucrarea va fi împărtita în două părți, care parcurg partea teoretică, respectiv partea experimentală a obiectivului nostru. Fiecărei metode de sortare i se vor atribui o descriere care să cuprindă o explicație a procesului de funcționare și complexitatea acestuia, respectiv pentru partrea experimentală, timpul de execuție rezultat fiecărei rulări a programului pentru ca în final să cuprindă o comparație și o interpretare a rezultatelor obținute.

2 Descrierea formală a problemei şi soluţiei

Considerăm problema sortării unui tablou x[1...n] având elemente din 1,2,...m astfel algoritmii de sortare după cum sunt prezentați în [1] pe care urmează sa îi comparăm sunt:

- Counting sort - sortare prin numărare

  Algoritmul funcționează prin folosirea unui tablou de frecvență pentru a număra de câte ori apare fiecare element. Astfel, algoritmul are o complexitate liniară dacă situația este una favorabilă având elemente multiple comune. Cel mai defavorabil caz este de complexitate $O(n^2)$.

  Algoritmul pentru Counting sort in Python după cum putem observa in [2] este:

  Listing 1: Counting sort

```
from collections import defaultdict
def counting_sort(A, key=lambda x: x):
    B, C = [], defaultdict(list) # Output and "counts"
    for x in A:
        C[key(x)].append(x) # "Count" key(x)
    for k in range(min(C), max(C)+1): # For every key in the range
        B.extend(C[k]) # Add values in sorted order
     return B
```

- Radix sort - sortare pe baza cifrelor

  Algoritmul se bazează pe următoarea idee: se ordonează tabloul în raport cu cifra cea mai puțin semnificativă a fiecărui număr, după care se sortează în raport cu cifra de rang imediat superior până se ajunge la cifra cea mai semnificativă. Algoritmul are o complexitate medie de $O(kn)$ dacă avem elemente constituite din cel mult k cifre și 10^k nu este semnificativ mai mare decât n.

  Algoritmul pentru Radix sort în Python după cum putem observa în [2] este:

Listing 2: Radix Sort

```
def countingSort(array, place):
    size = len(array)
    output = [0] * size
    count = [0] * 10

    for i in range(0, size):
        index = array[i] // place
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = size - 1
    while i >= 0:
        index = array[i] // place
        output[count[index % 10] - 1] = array[i]
        count[index % 10] -= 1
        i -= 1

    for i in range(0, size):
        array[i] = output[i]

def radixSort(array):
    # Get maximum element
    max_element = max(array)

    # Apply counting sort to sort elements based on place value.
    place = 1
    while max_element // place > 0:
        countingSort(array, place)
        place *= 10
```

- Shell sorting - sortare prin inserție cu pas variabil

  Unul dintre dezavantajele sortării prin inserție este faptul că la fiecare etapă un element al șirului se deplasează cu o singură poziție. O varianta de reducere a numărului de operații efectuate este de a compara elementele aflate la o distanță mai mare și de a realiza deplasarea acestor elemente peste mai multe poziții. Această tehnică se aplică în mod repetat pentru valori din ce în ce mai mici ale pasului, asigurând sortarea șirului. Elementul cheie al algoritmului îl reprezintă alegerea valorii pasului h. Pentru alegeri adecvate se poate obține un algoritm de complexitate $O(n^(3/2))$ în loc de $O(n^2)$ cum este cazul algoritmului clasic de sortare prin inserție.

  Aceasta este ideea algoritmului propus de Donald Shell în 1959 cunoscut sub numele "shell sort".

  Algoritmul pentru Shell sorting în Python după cum putem observa în [2] este:

Listing 3: Shell Sort

```
def shellSort(arr):

    # Start with a big gap, then reduce the gap
    n = len(arr)
    gap = n/2

    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped
    # order keep adding one more element until the entire array
    # is gap sorted
    while gap > 0:

        for i in range(gap,n):

            # add a[i] to the elements that have been gap sorted
            # save a[i] in temp and make a hole at position i
            temp = arr[i]

            # shift earlier gap-sorted elements up until the correct
            # location for a[i] is found
            j = i
            while  j >= gap and arr[j-gap] >temp:
                arr[j] = arr[j-gap]
                j -= gap

            # put temp (the original a[i]) in its correct location
            arr[j] = temp
        gap /= 2
```

- Shaker sort- Sortarea prin interschimbarea elementelor vecine

  Shaker sort presupune sortarea prin interschimbarea elementelor vecine. Ea asigura la fiecare pas al ciclului exterior, plasarea a câte unui element pe poziția finală. Prin reținerea indicelui ultimei interschimbari efectuate, atât la parcurgerea de la stânga la dreapta cât și de la dreapta la stânga se poate limita regiunea analizată cu mai mult de o poziție atât la stânga cât și la dreapta(când tabloul conține porțiuni deja sortate).

  Algoritmul are o complexitate medie de $O(n^2)$ și este o extensie a algoritmului Bubble Sort, operand pe două direcții.

  Algoritmul pentru Shaker sort în Python după cum putem observa în [2] este:

Listing 4: Shaker Sort

```
def cocktailSort(a):
    n = len(a)
    swapped = True
    start = 0
    end = n-1
    while (swapped == True):
```

```
        # reset the swapped flag on entering the loop,
        # because it might be true from a previous
        # iteration.
        swapped = False

        # loop from left to right same as the bubble
        # sort
        for i in range (start, end):
            if (a[i] > a[i + 1]) :
                a[i], a[i + 1]= a[i + 1], a[i]
                swapped = True

        # if nothing moved, then array is sorted.
        if (swapped == False):
            break

        # otherwise, reset the swapped flag so that it
        # can be used in the next stage
        swapped = False

        # move the end point back by one, because
        # item at the end is in its rightful spot
        end = end-1

        # from right to left, doing the same
        # comparison as in the previous stage
        for i in range(end-1, start-1, -1):
            if (a[i] > a[i + 1]):
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True

        # increase the starting point, because
        # the last stage would have moved the next
        # smallest number to its rightful spot.
        start = start + 1
```

- Selection Sort- sortarea prin selecție

  Ideea de bază a acestui algoritm este următoarea: pentru fiecare poziție, începând cu prima, se selectează din subtabloul ce începe cu acea poziție cel mai mic element și se amplasează pe locul respectiv(prin interschimbare cu elementul curent de pe poziția i).

  Algoritmul are o complexitate de $O(n^2)$

  Algoritmul pentru Selection Sort în Python după cum putem observa în [2] este:

Listing 5: Selection Sort

```
def sel_sort(seq):
    for i in range(len(seq)-1,0,-1): # n..i+1 sorted so far
        max_j = i # Idx. of largest value so far
        for j in range(i): # Look for a larger value
            if seq[j] > seq[max_j]: max_j = j # Found one? Update max_j
```

```
        seq[i], seq[max_j] = seq[max_j], seq[i] # Switch largest into place
```

- Insertion Sort- Sortarea prin inserție

  Algoritmul se bazează pe ideea începerii cu al doilea element al tabloului x[1...n], fiecare element este inserat pe poziția adecvata în subtabloul care îl precede.

  Algoritmul are o complexitate medie de $O(n^2)$

  Algoritmul pentru Insertion Sort în Python după cum putem observa în [2] este:

Listing 6: Insertion Sort

```
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
```

- Bubble Sort- Sortarea prin comparare

  Prin această metodă se parcurge vectorul și se compară fiecare element cu succesorul său. Dacă nu sunt în ordine cele două elemente se inteschimba între ele. Vectorul se parcurge de mai multe ori, până când la o parcurgere completă nu se mai executa nicio interschimbare între elemente.

  Algoritmul are o complexitate medie de $O(n^2)$

  Algoritmul pentru Bubble Sort în Python după cum putem observa în [2] este:

Listing 7: Bubble Sort

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

- Quicksort - Sortare prin tehnica divizării

  Quicksort a fost dezvoltat de C. A. R. Hoare în 1960. Acesta efectuează sortarea bazandu-se pe o strategie divide et impera. Astfel, el împarte lista de sortat în două subliste mai ușor de sortat. Pașii algoritmului sunt:

  1. Se alege un element al listei, denumit pivot.
  2. Se reordonează lista astfel încât toate elementele mai mici decât pivotul să fie plasate înaintea pivotului și toate elementele mai mari să fie după pivot. După această partiționare, pivotul se afla în poziția sa finală.
  3. Se sortează recursiv recursiv sublista de elemente mai mici decât pivotul și sublista de elemente mai mari decât pivotul.

  Algoritmul pentru Quicksort în Python după cum putem observa în [2] este

Listing 8: Quicksort

```
    def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if   arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

3 Modelarea şi implementarea problemei şi soluţiei

Limbajul folosit in comparația experimentală este:

- Python 3.8.2

Testele au fost rulate pe platforma Jupyter Notebook iar pentru a măsura timpul am folosit metoda time după cum se poate observa:

Listing 9: Quick Sort timed function

```
import numpy as np
import time

inceput=time.time()

def bubbleSort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]

arr = y

bubbleSort(arr)
print("Bubble sort : ")
sfarsit=time.time()
print(sfarsit - inceput)
```

Listing 10: Quick Sort timed function

```
inceput=time.time()
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        if   arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)
```

```
        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

arr = y
n = len(arr)
quickSort(arr,0,n-1)
print("Quicksort: ")
sfarsit=time.time()
print(sfarsit - inceput)
```

Numerele au fost generate aleator importând modulul random existent în Python 3.8.2 și au fost salvate într-un fișier text de forma(numere.txt). Elementele au fost puse pe o singură coloană, folosind ca și separator "\n" și au fost salvate în variabila *y* folosită în exemplele de mai sus de unde am luat toate datele de intrare pentru testele de eficiență.

Codul pentru preluarea datelor de intrare și afișarea timpului de execuție este următorul:

Listing 11: In and Out

```
f = open("C:/Users/Administrator/Desktop/numere.txt", "r")
x=f.read()
y=[int(n) for n in x.split("\n")]

#algoritmul de sortare

print("Numele algoritmului de sortare folosit : ")
sfarsit=time.time()
print(sfarsit - inceput)
```

4 Studiu de caz / Experiment

În testele efectuate am încercat să accentuez importanța alegerii unei metode de sortare eficiente. Astfel, folosind un set din ce în ce mai mare de date de intrare generate după cum am menționat anterior, am obținut următoarele rezultate:

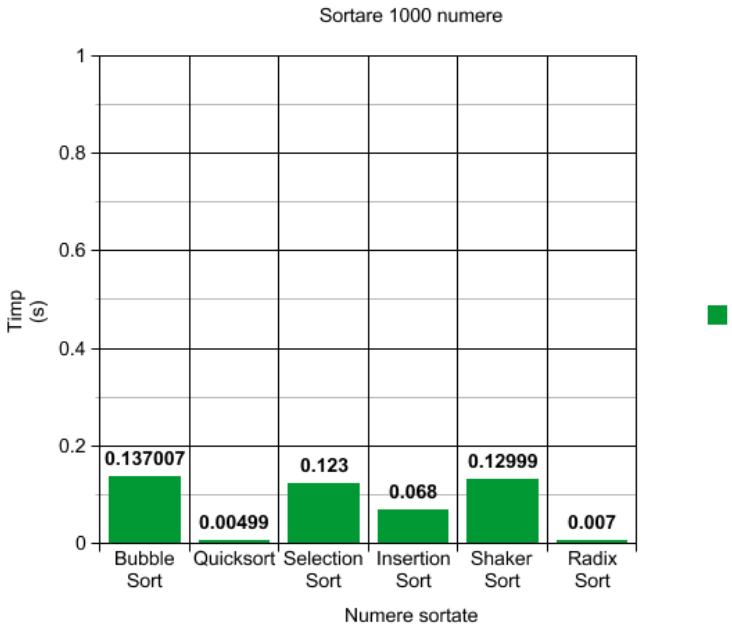


Figura 1: Diferențele dintre timpii de executie pentru 1000 de numere.

Din acestea putem să vedem că atunci când avem de sortat multe numere apare o diferență foarte mare între timpul de execuție al unei metodei ineficiente dar foarte cunoscute de sortare Bubble Sort și al unei metode considerată a fi în cea mai mare parte a timpului mai eficientă Quicksort. Diferențele sunt atât de mari din cauza faptului că, complexitatea medie a algoritmilor diferă foarte mult.

Am repetat experimentul pentru diferite seturi de date însă rezultatele au fost asemănătoare. În cele mai multe cazuri metoda Quicksort a fost cea mai eficientă. Cu toate acestea, metoda Radix Sort s-a dovedit a fi la același nivel, dacă nu chiar mai eficientă când avem numere asemănătoare(diferența dintre cel mai mic și cel mai mare număr este minimă).

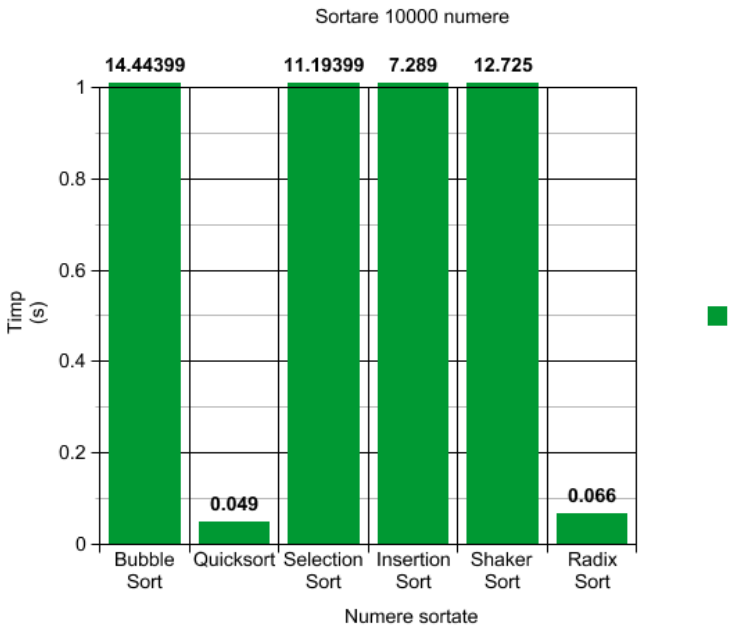


Figura 2: Diferențele dintre timpii de executie pentru 10000 de numere.

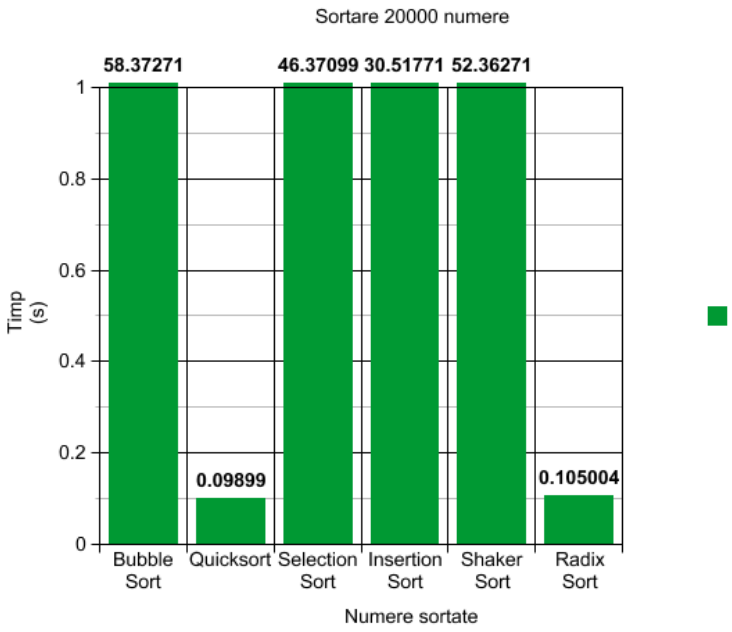
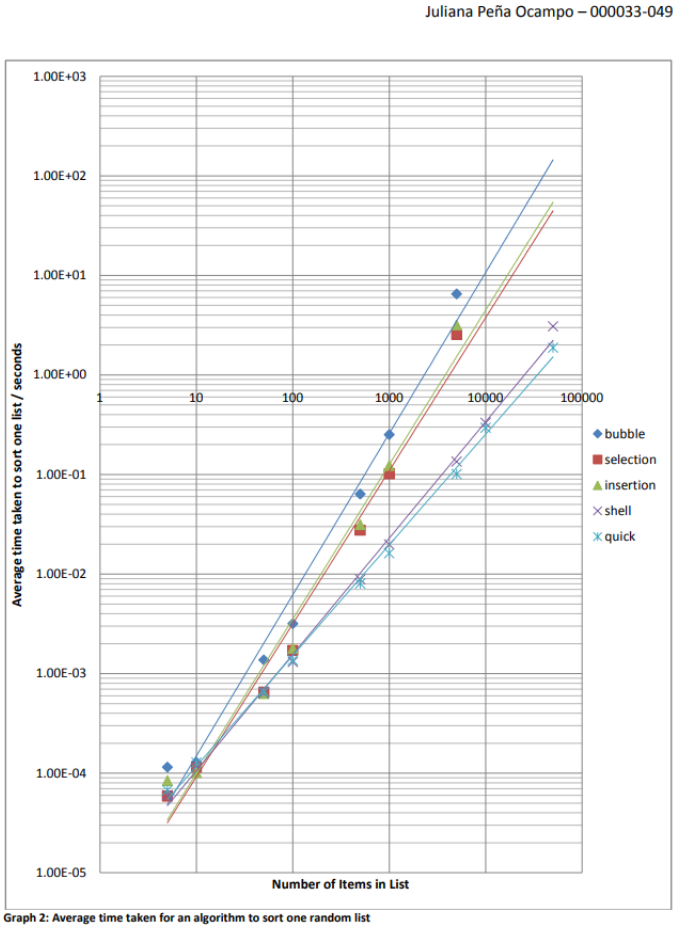


Figura 3: Diferențele dintre timpii de executie pentru 20000 de numere.

5 Comparaţie cu literatura

După cum am putut observa, diferența dintre timpii de execuție este foarte mare. La acest lucru a ajuns și Juliana Pena în [3]. Comparând rezultatele noastre cu cele prezentate acolo, remarcăm că în cea mai mare parte clasamentul metodelor de sortare în ordinea eficienței lor este asemănător.

Comparând graficul nostru prezentat anterior cu graficul de mai jos obținut de către [3] observăm că diferență devine tot mai dramatică dacă avem un număr mai mare de date de intrare.



Graph 2: Average time taken for an algorithm to sort one random list

Problema propusă în această lucrare rămâne în continuare fără soluție din cauza faptului că nu există o metodă de sortare perfectă pentru orice set de date. Cu toate acestea, după cum am demonstrat, în cele mai multe cazuri metode precum Quicksort sau Radix Sort sunt superioare.

6 Concluzii şi direcţii viitoare

În concluzie, în lucrare am prezentat problema alegerii unei metode de sortare și pentru a face acest lucru a trebuit să facem o comparație atât teoretica cât și experimentală. După executarea și studiul de caz efectuat am ajuns la concluzia că unele metode de sortare au un timp de execuție mult mai mic comparativ cu altele.

Pentru a face acest lucru cât mai clar, am încercat să folosim un set de date de intrare cât mai mare. Cu toate acestea, algoritmii de sortare precum Bubble Sort sau Insertion Sort aveau un timp de execuție absurd de mare și am fost nevoiți să ne oprim la 20 000 de elemente.

O alta problemă întâlnită a fost utilizarea funcției timeit() existente în Python. Aceasta funcție nu era destul de precisa și am rezolvat problema prin folosirea funcției time().

Bibliografie

[1] Cursul si seminarul ASD 1, Introducere in proiectarea si analiza algoritmilor (Daniela Zaharie)

[2] Hetland, Magnus Lie. Python Algorithms: mastering basic algorithms in the Python Language. Apress, 2014.

[3] Ocampo, Juliana Pena. "An empirical comparison of the runtime of five sorting algorithms." International Baccalaureate Extended Essay (2008).