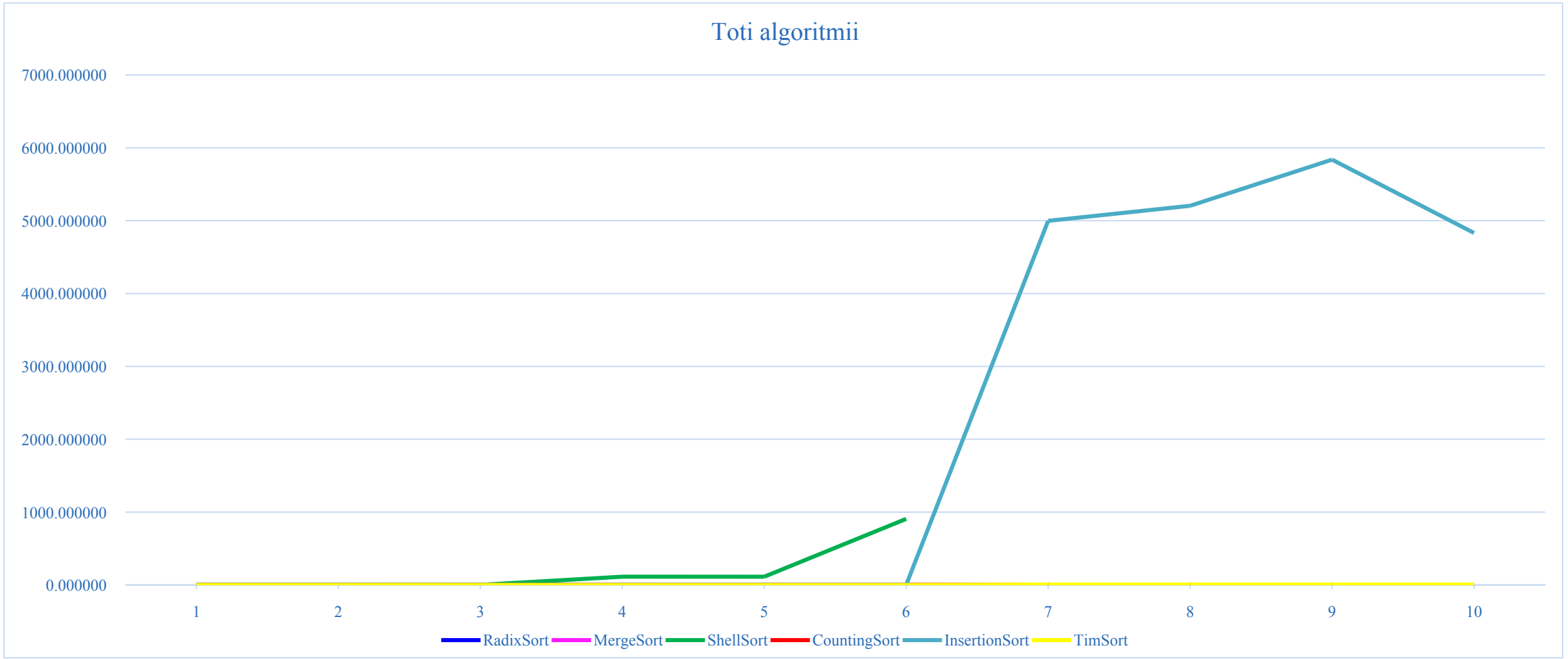
Proiect 1 – Algoritmi de sortare

Dinu Florin – Silviu
Grupa 131

Prezentare Generala

- Algortimii implementati sunt:
  - RadixSort
  - ShellSort
  - MergeSort
  - CountingSort
  - InsertionSort
  - TimSort
  - Algoritmul nativ pentru C++

Prezentare Generala



Teste folosite

- Pentru a avea date comparabile, am generat un fisier denumit **teste_generale.in** pe care l-am pus in folderul **cmake-build-debug** al fiecarui proiect
- Pentru scriptul de generare: GeneratorTeste/main.py

Max	N
10000	100
20000	500
100000	1000
200000	5000
1000000	5000
2000000	10000
10000000	1000000
20000000	1000000
40000000	1000000
50000000	1000000

RadixSort

- Cea mai buna complexitate O(n*log n)
- Calculul complexitatii este: $O(\log_b (max) * (n + b))$, unde **n** reprezinta numarul de numere, **b** baza si **max** numarul maxim
- Pentru a putea folosi numere in orice baza, am facut algoritmul cu impartiri
- Pentru numere in baze puteri ale lui 2 se poate face o implementare alternativa cu sortare pe biti (grupuri de 1, 4, 8...) – deoarece sunt numere naturale, putem ignora MSB daca folosim int (nu si daca folosim unsigned_int)

Radix Sort

RadixSort
0.000017
0.000095
0.000159
0.001068
0.001130
0.002764
0.247757
0.284604
0.292008
0.422082

CountingSort

- Cea mai buna complexitate: O(n*log n)
- Algoritmul foloseste un vector separat pentru numarare cu lungimea **max – min + 1** din vectorul initial

CountingSort

CountingSort

0.000106

0.000167

0.000866

0.001730

0.007571

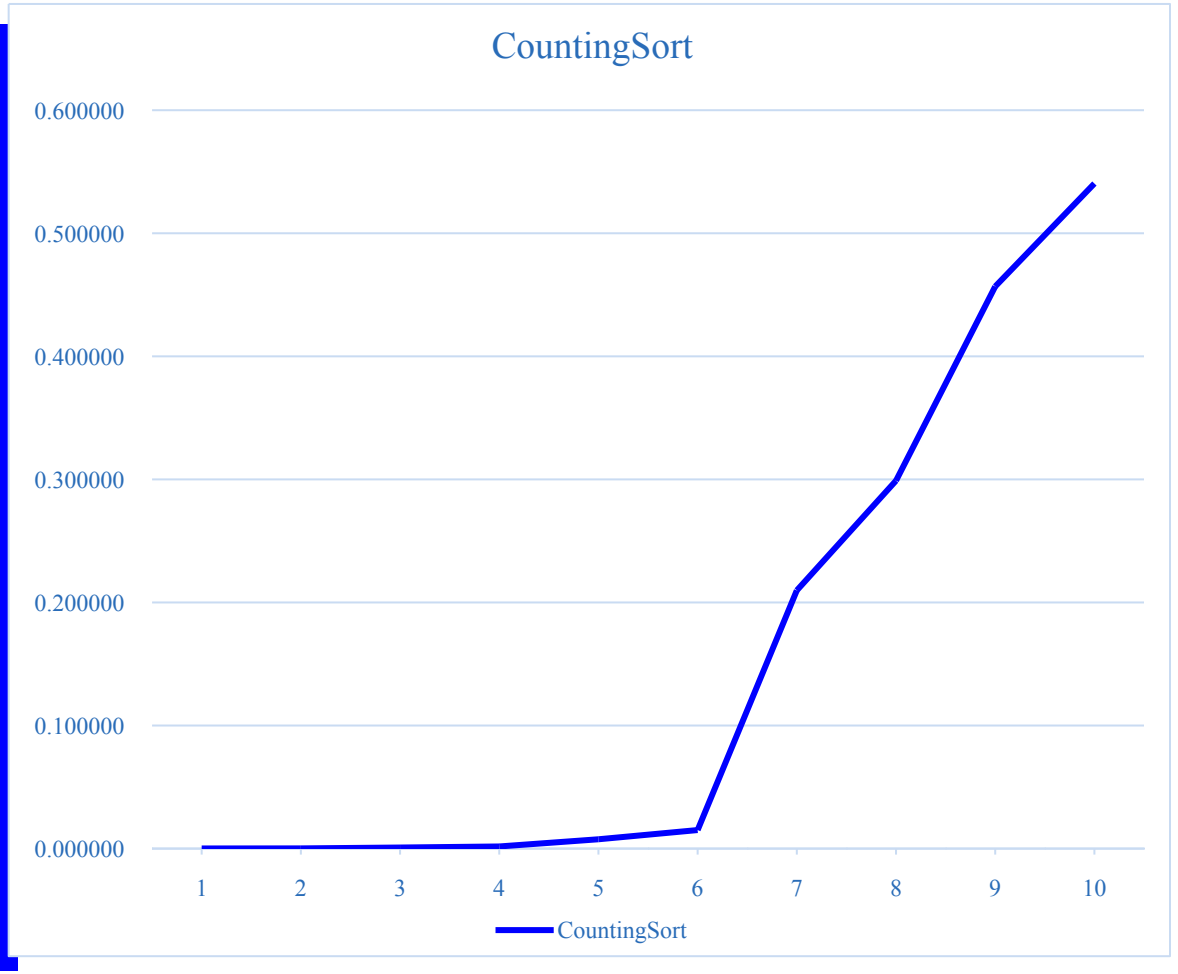
0.015060

0.209803

0.298961

0.456766

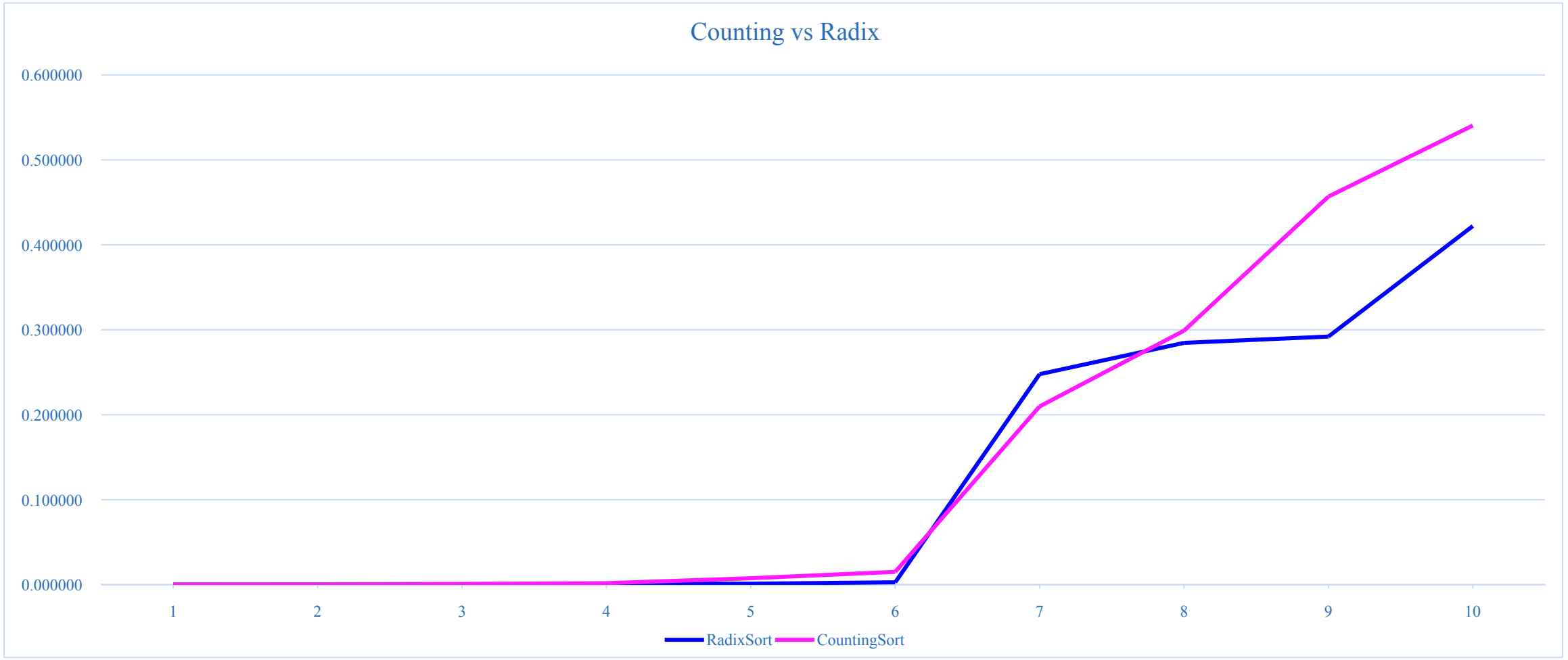
0.540427



CountingSort si RadixSort

- Timpul *mediu* de rulare pentru RadixSort este **0.1251684**
- Timpul *mediu* de rulare pentru CountingSort este **0.1531457**
- Observam faptul ca RadixSort este in medie mai rapid decat CountingSort
- Timpul de rulare pe testul 7 (max = 10000000, n = 1000000) a fost mai slab pentru RadixSort

CountingSort si RadixSort

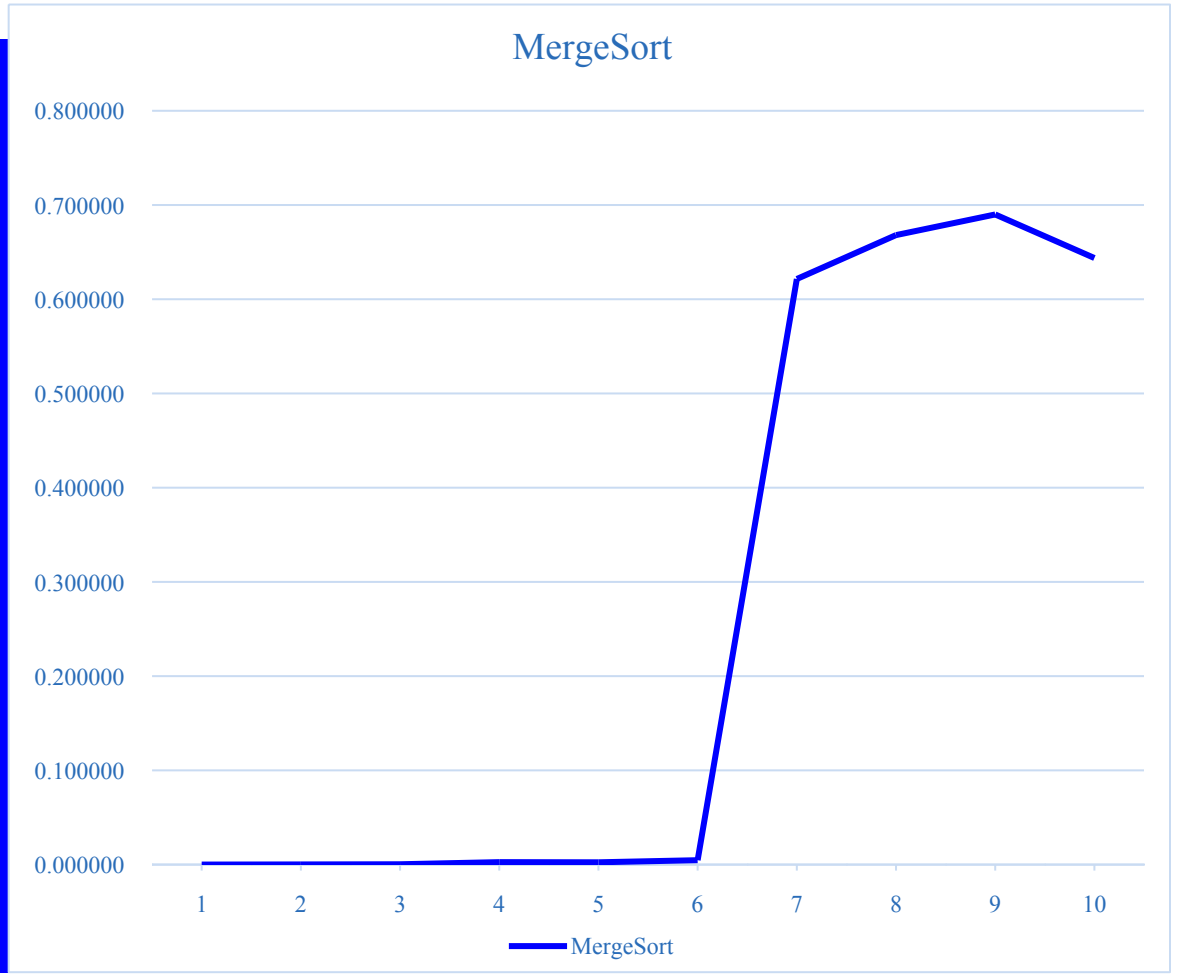


MergeSort

- Complexitatea algorimtului in scenariul cel mai bun, mediu si cel mai prost este de O(n*log n)
- Algoritmul are nevoie de un vector intermediar de lungime n

MergeSort

MergeSort
0.000033
0.000172
0.000358
0.002613
0.002435
0.004671
0.621487
0.668062
0.689968
0.643752



ShellSort

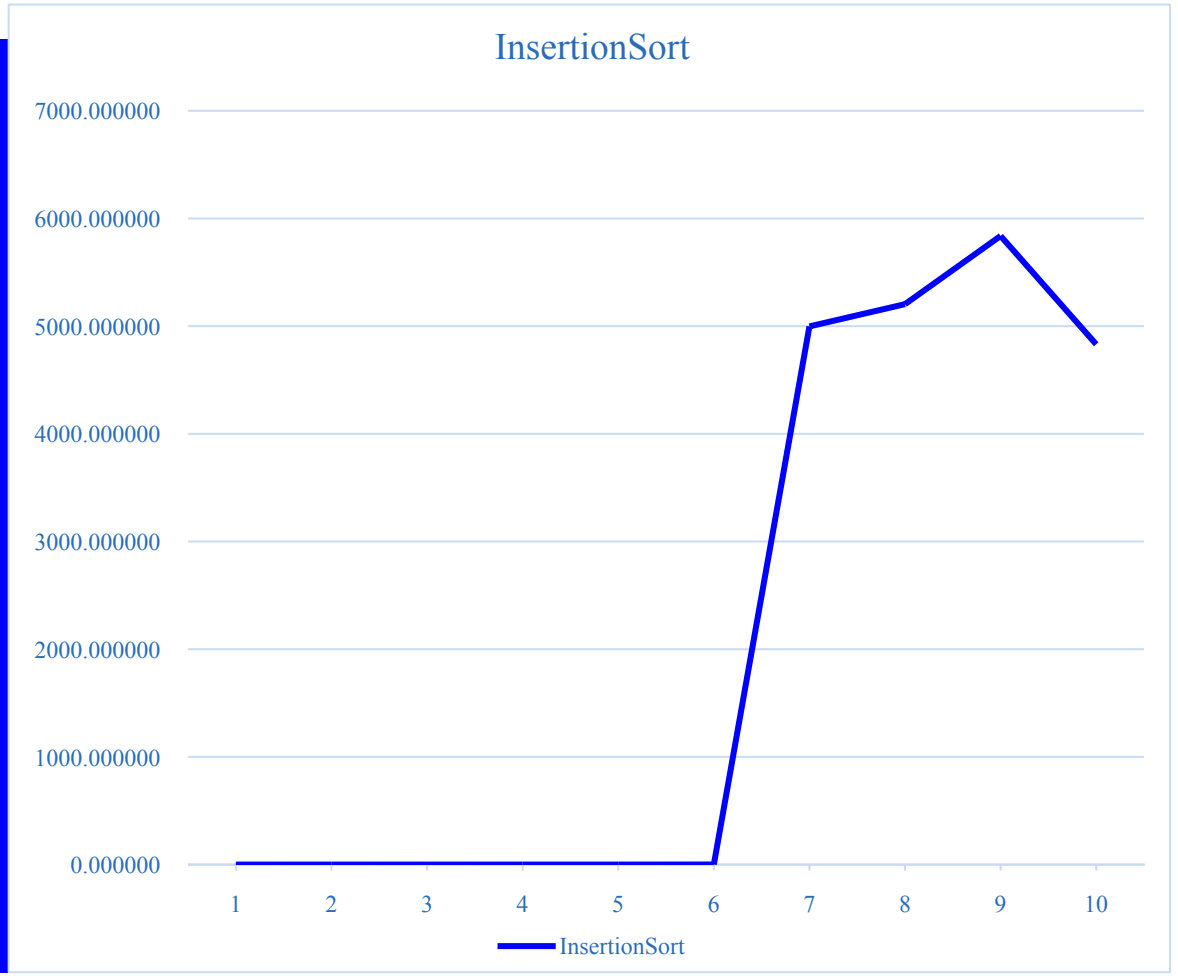
- Complexitatea medie este de O(n*log n)
- Complexitatea in scenariul cel mai prost este de $O(n^2)$
- Pentru **gap** am folosit secventa Knuth

InsertionSort

- Complexitatea este $O(n^2)$
- Implementarea este cea mai simpla dintre toti algoritmii

InsertionSort

InsertionSort
0.000086
0.002176
0.010096
0.229453
0.220580
0.792158
4997.670000
5204.320000
5837.770000
4831.500000

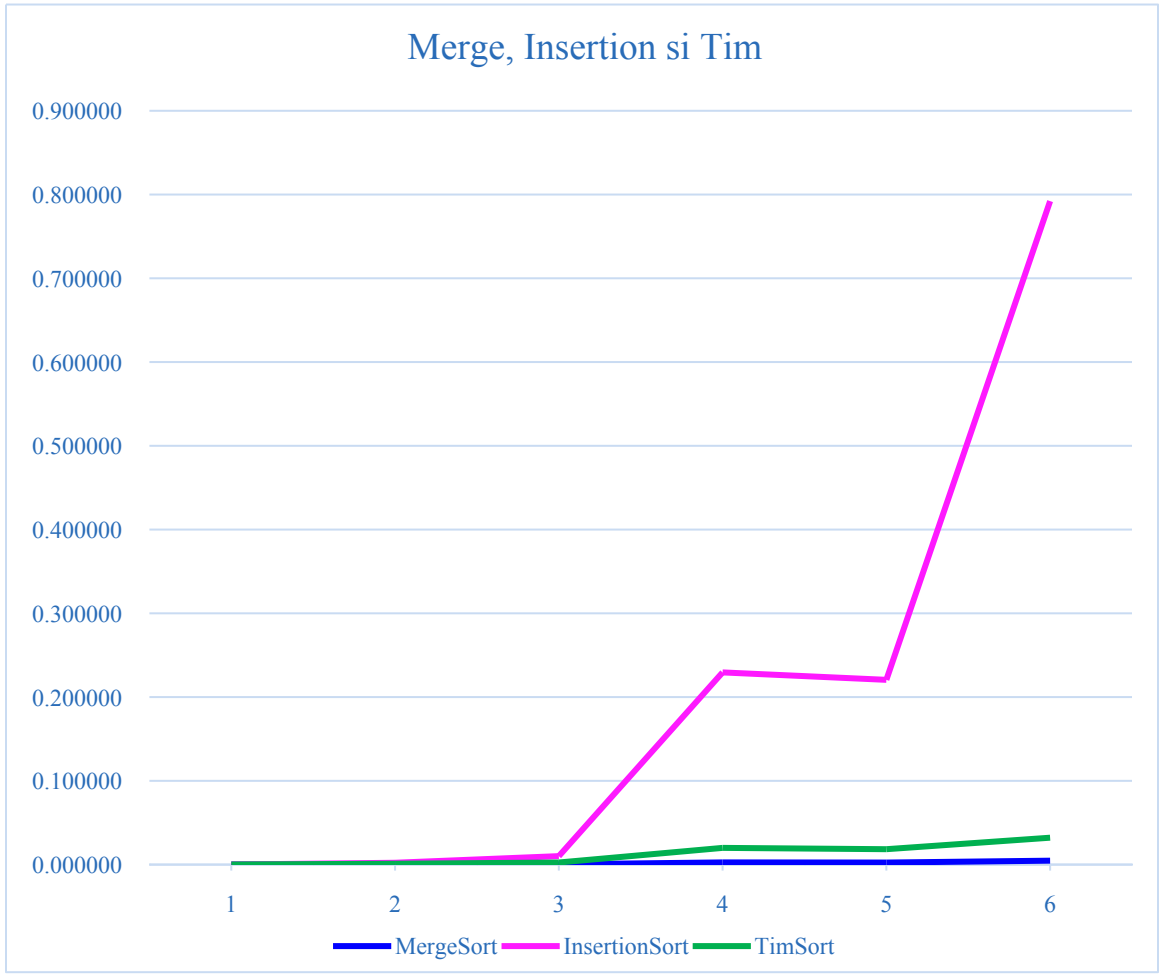


TimSort

- Complexitatea este O(n*log n)
- Este o combinatie intre InsertionSort si MergeSort

TimSort

TimSort
0.000159
0.001015
0.002491
0.019949
0.018379
0.032108
5.811590
5.698630
5.637730
5.662480



Algoritmul nativ

- Complexitatea medie O(n*log n)
- Foloseste o combinatie de IntroSort (combinatie la randul sau dintre QuickSort si HeapSort) si InsertionSort

Nativ

0.000057

0.000327

0.000626

0.004067

0.003569

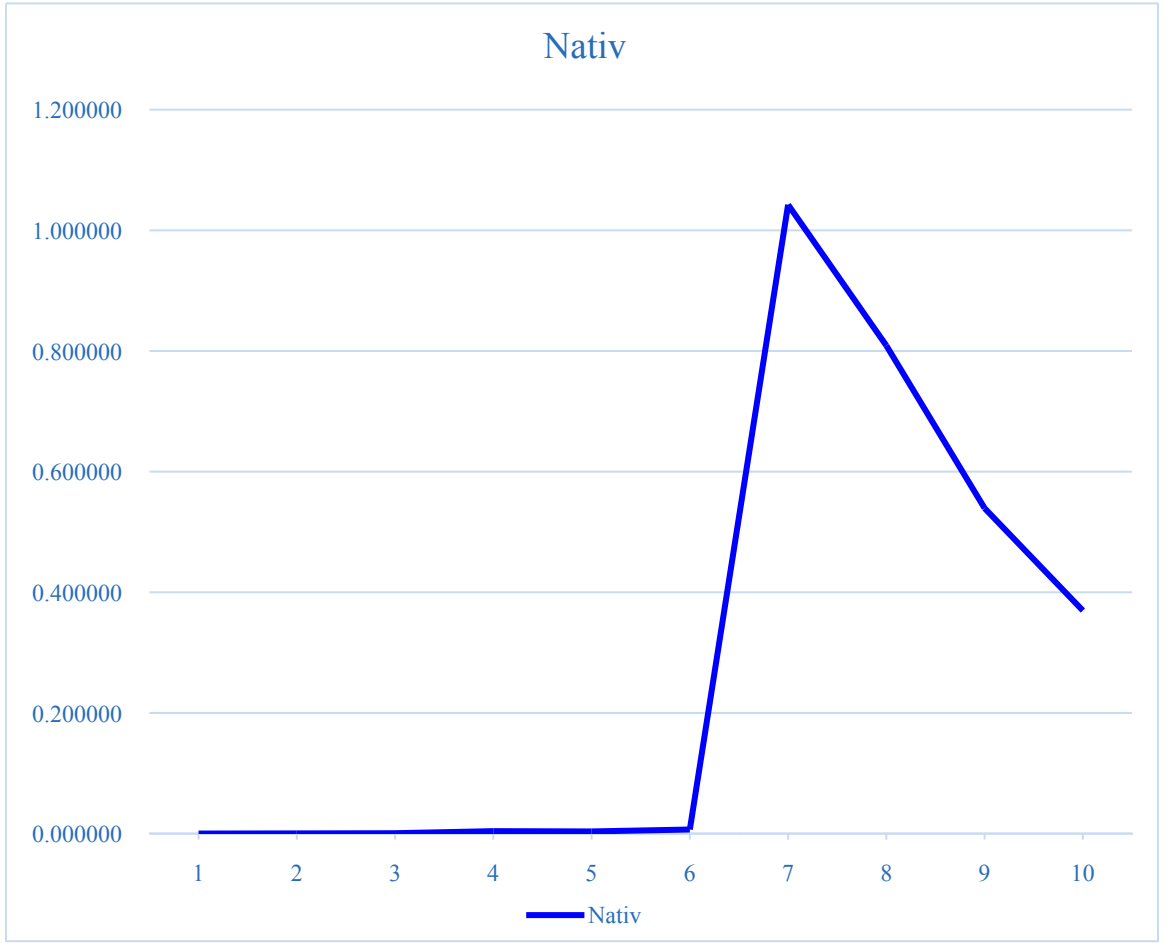
0.006755

1.042490

0.808941

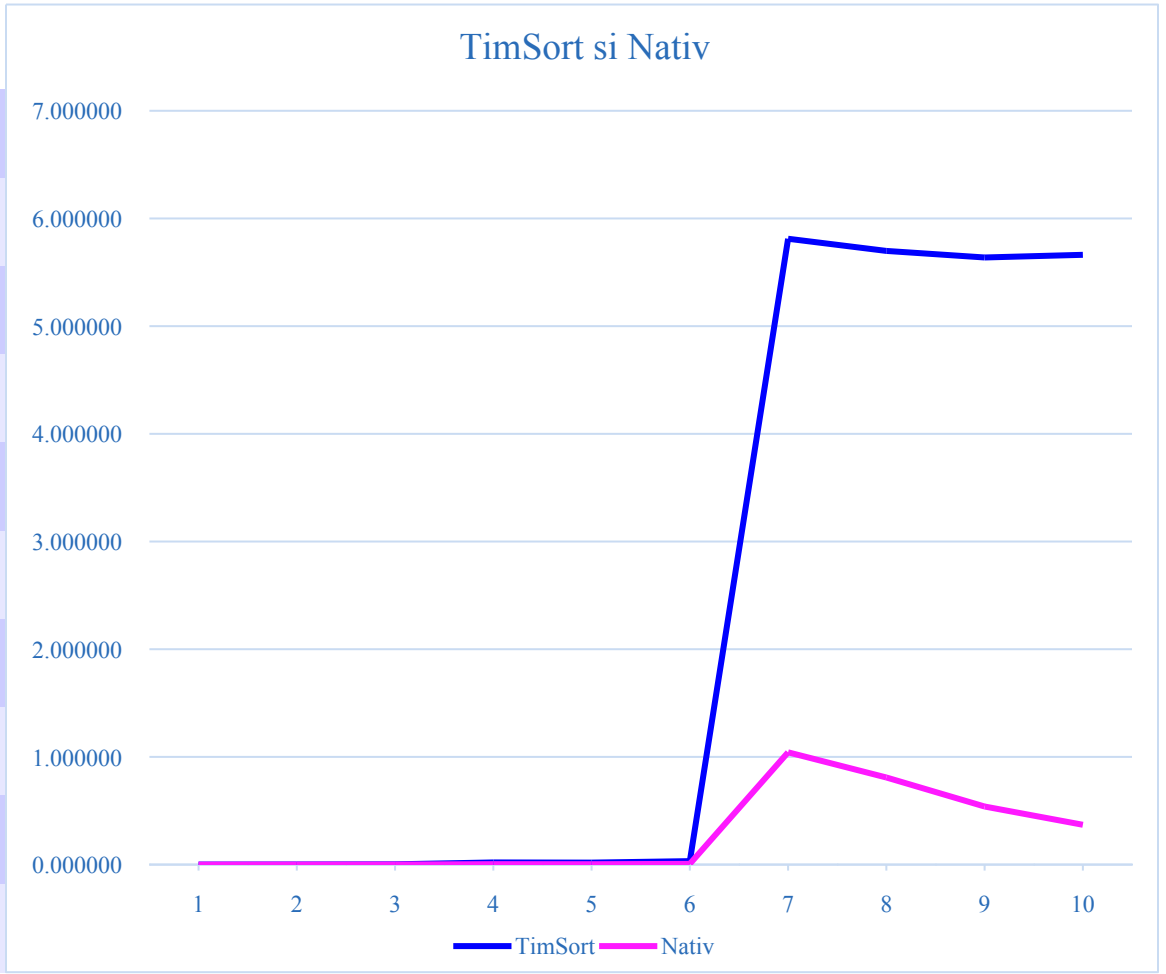
0.539317

0.369710



TimSort vs Nativ

TimSort	Nativ
0.000159	0.000057
0.001015	0.000327
0.002491	0.000626
0.019949	0.004067
0.018379	0.003569
0.032108	0.006755
5.811590	1.042490
5.698630	0.808941
5.637730	0.539317
5.662480	0.369710



TimSort vs Nativ

- Comparatia este relevanta deoarece TimSort este folosit nativ de python
- In toate cazurile algoritmul nativ de C++ a avut o viteza mai mare
- Pentru n mic diferenta nu este semnificativa

Sfarsit

- Fiecare algoritm este implementat in proiectul sau
- In fiecare proiect exista folderul **cmake-build-debug** care are **teste_generale.in** si **raport.out**
- In folderul **Analytics** exista un fisier xlsx cu 2 spreadsheeturi ce contin datele folosite in prezentare
- Pentru implementare am folosit **JetBrains CLion** si sistemul de operare **Ubuntu 21.10**
- Standardul folosit pentru cmake este **C++ 14**

Va multumesc

Dinu Florin - Silviu
Grupa 131