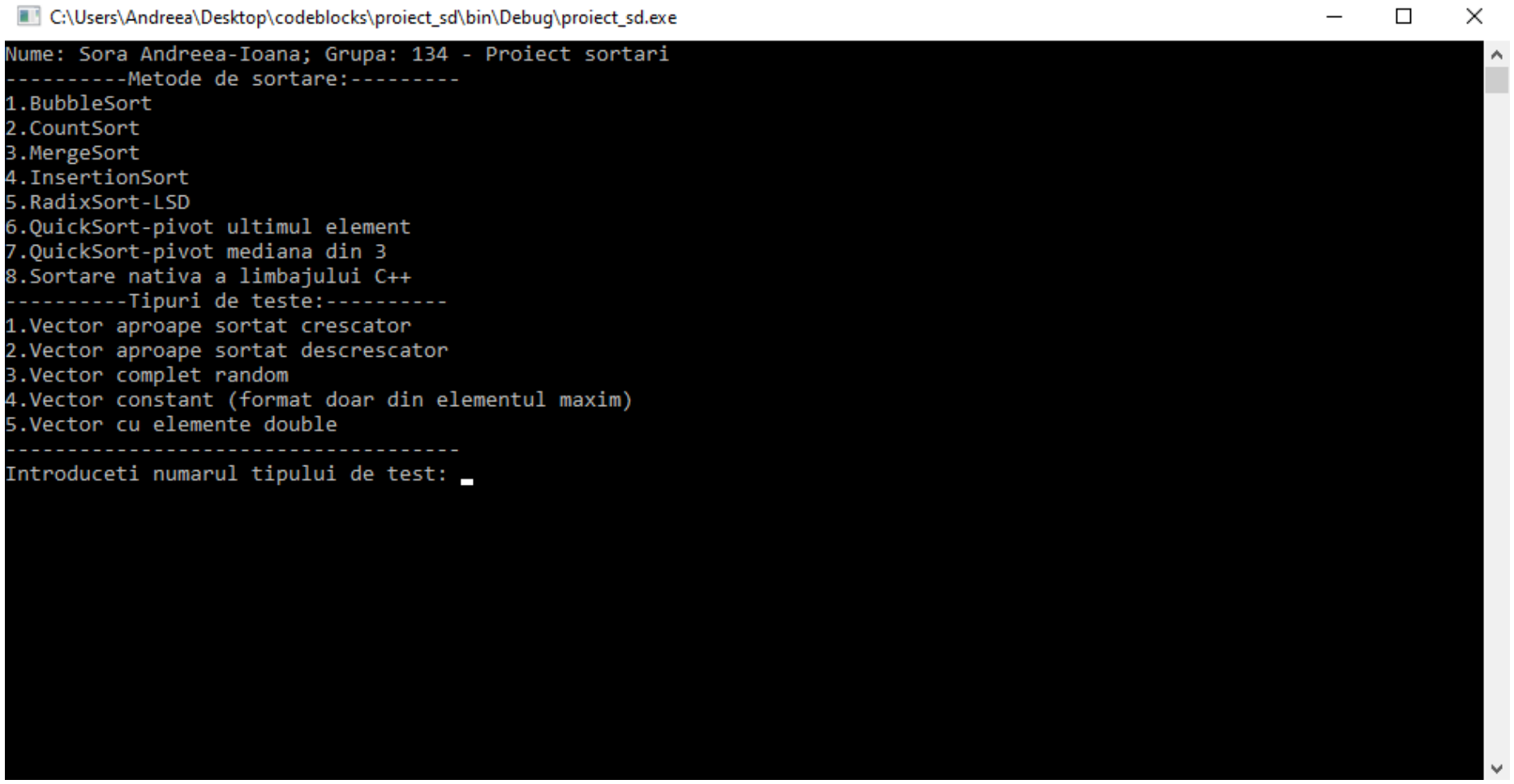
PROIECT SORTĂRI

Sora Andreea-Ioana

Seria 13

Grupa 134

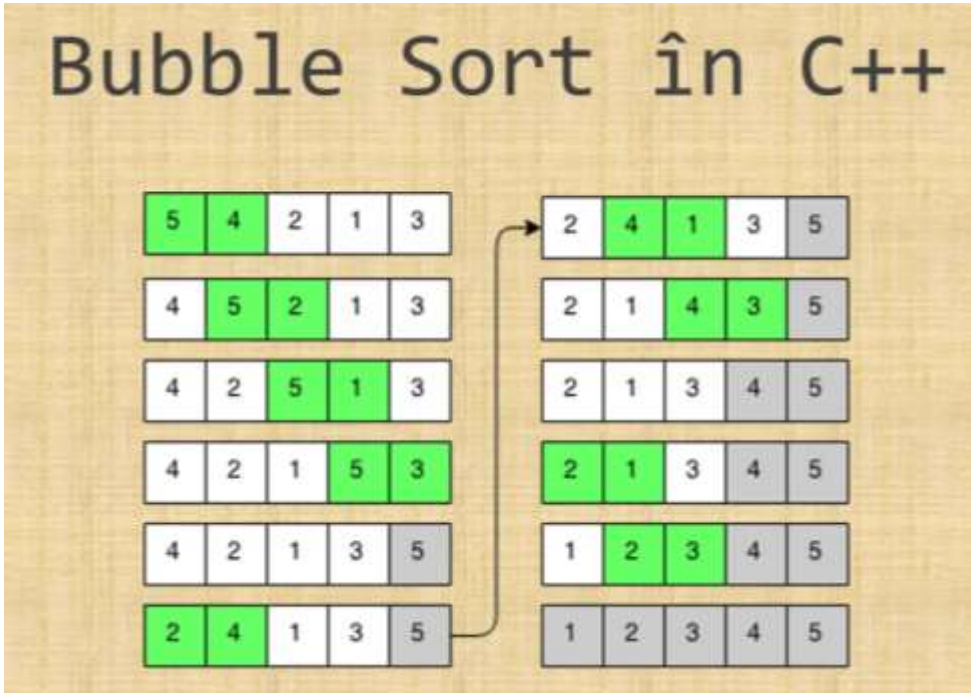
Structuri de Date – Lect. Dr. Marius Dumitran
Facultatea de Matematică și Informatică
Universitatea din București


C:\Users\Andreea\Desktop\codeblocks\proiect_sd\bin\Debug\proiect_sd.exe
Nume: Sora Andreea-Ioana; Grupa: 134 - Proiect sortari
----------Metode de sortare:---------
1.BubbleSort
2.CountSort
3.MergeSort
4.InsertionSort
5.RadixSort-LSD
6.QuickSort-pivot ultimul element
7.QuickSort-pivot mediana din 3
8.Sortare nativa a limbajului C++
----------Tipuri de teste:----------
1.Vector aproape sortat crescator
2.Vector aproape sortat descrescator
3.Vector complet random
4.Vector constant (format doar din elementul maxim)
5.Vector cu elemente double

Introduceti numarul tipului de test:

BubbleSort

Complexitate: O(n^2)
Idee:
Parcurgem vectorul, iar pentru oricare două elemente vecine care nu sunt în ordinea dorită, le interschimbăm valorile.



CountSort

Complexitate:
-timp: O(n+max)
-spațiu:O(max)
Idee:
Parcurgem vectorul și reținem frecvențele valorilor într-un vector frecv. După acest pas, parcurgem frecv de la 0 la max,iar pentru fiecare element frecv[i] întâlnit, adăugăm in vectorul rezultat i de frecv[i] ori.

Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

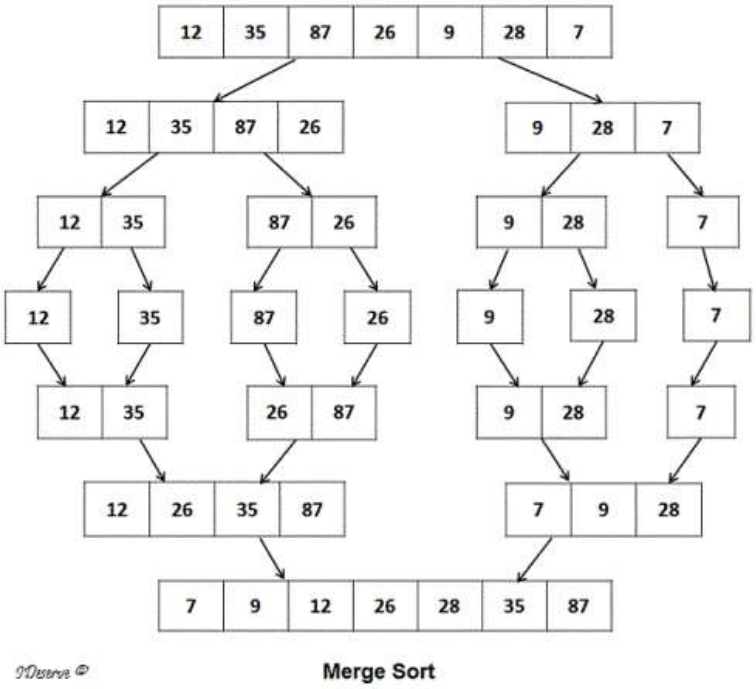
Count Array

0	1	2	3	4
5	3	4	0	2

Sorted Data

0	0	0	0	0	1	1	1	2	2	2	2	4	4

MergeSort

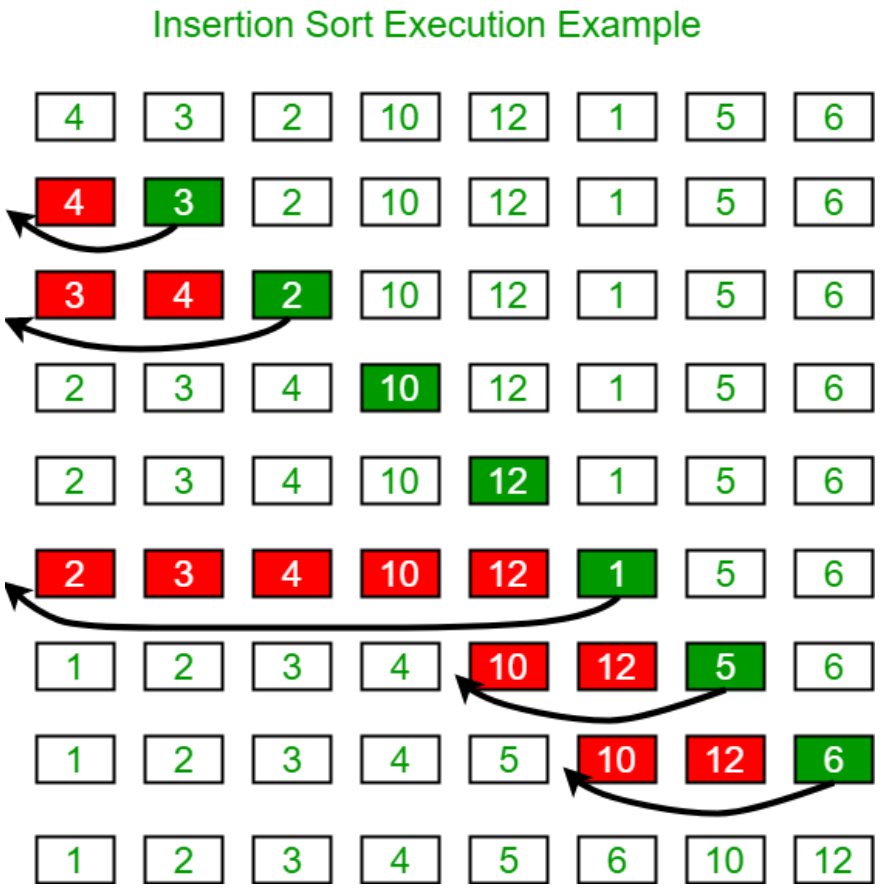


Merge Sort

Complexitate: O(nlogn)
Idee:
Se împarte vectorul în jumătate și se interclasează independent fiecare parte formându-se un subșir ordonat din vector, mai mare, care, la rândul lui se va interclasa cu subșirul corespunzator obținut în acelaşi mod etc.

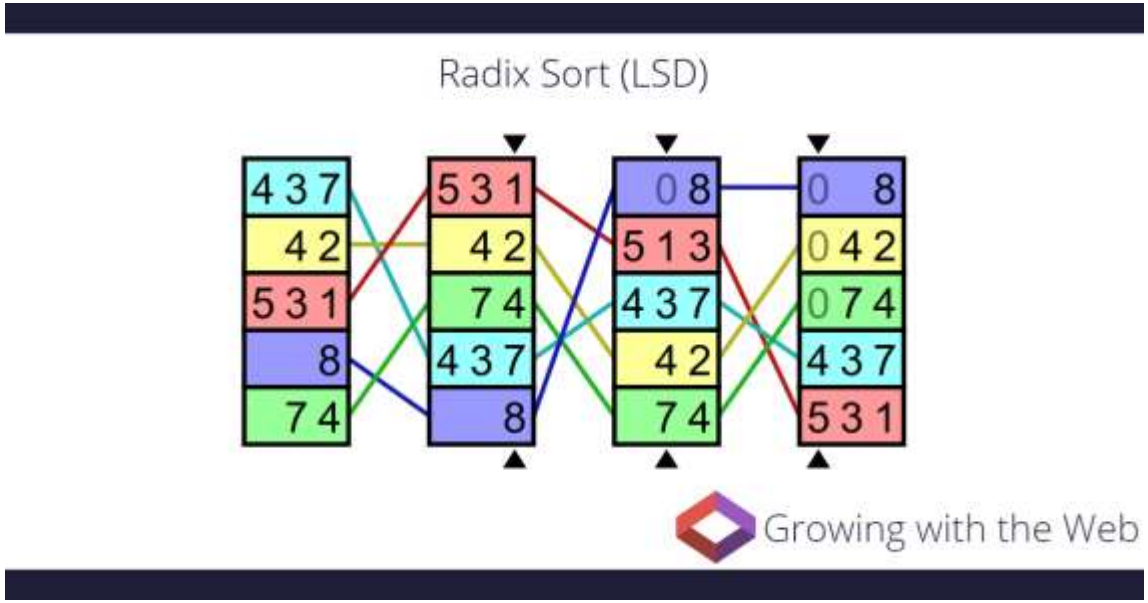
InsertionSort

Complexitate: O(n^2)
Idee:
Parcurgem vectorul și inserăm elementul v[i] în secvența din stânga sa, elementele mutându-se spre dreapta.



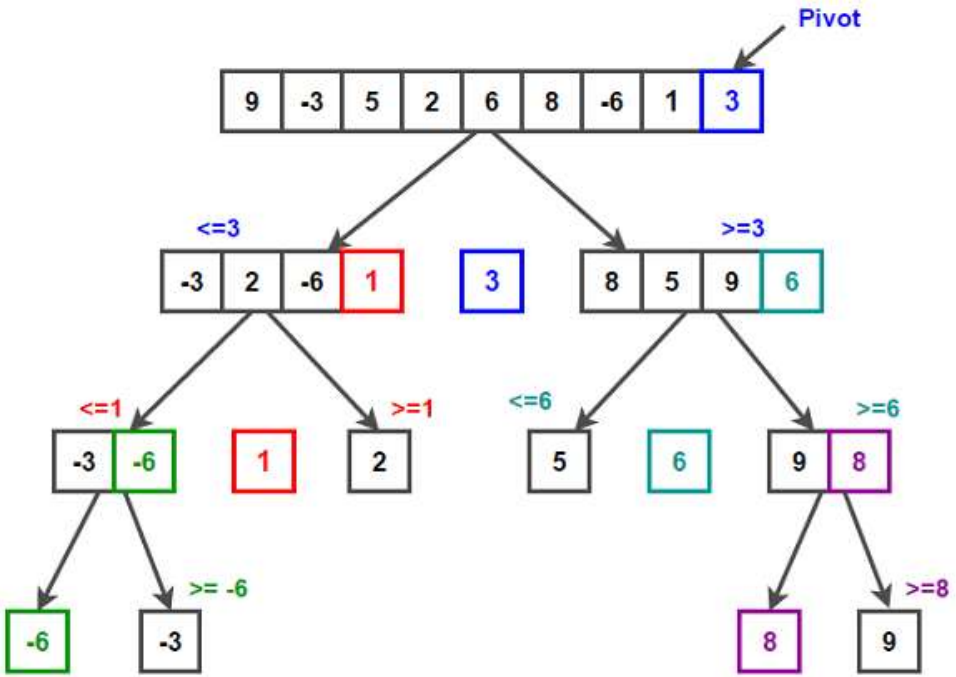
RadixSort – Varianta LSD (Least significant digit)

Complexitate:
-timp: O(nk);k-lungimea medie a fiecărei chei;
-spațiu:O(n+b)
Idee:
Se grupează elementele pe rând după cea mai nesemnificativă cifra (de la coadă la cap), urmând ca la final vectorul să fie sortat.



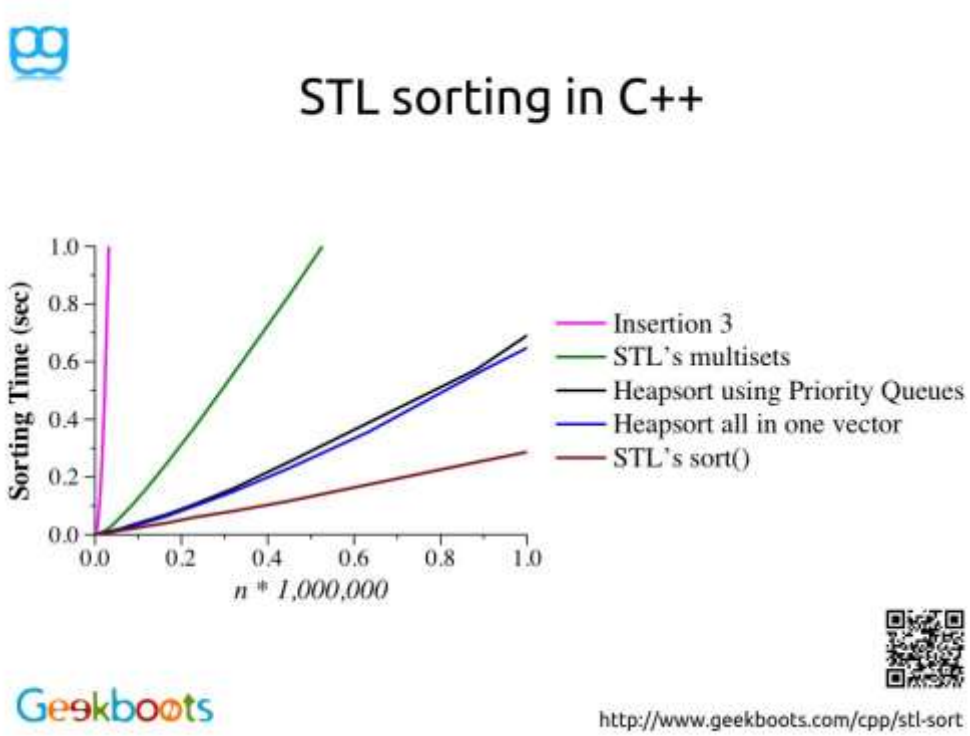
QuickSort

Complexitate:O(n^2) (worst case)/O(nlogn)(best case)
Idee:
Se alege un element drept pivot, iar subvectorul din stânga va fi format din elementele mai mici decât pivotul, iar în dreapta mai mari decât pivotul.
Pivotul poate fi ales ca fiind:
-ultimul/primul sau elementul din mijloc (cazuri nu tocmai bune);
-un element random;
-mediana din 3 (caz favorabil).



Sortarea nativă a limbajului C++ (sort)

Funcția sort este declarată în header-ul algorithm, ea folosind un hibrid între QuickSort, HeapSort și InsertionSort.



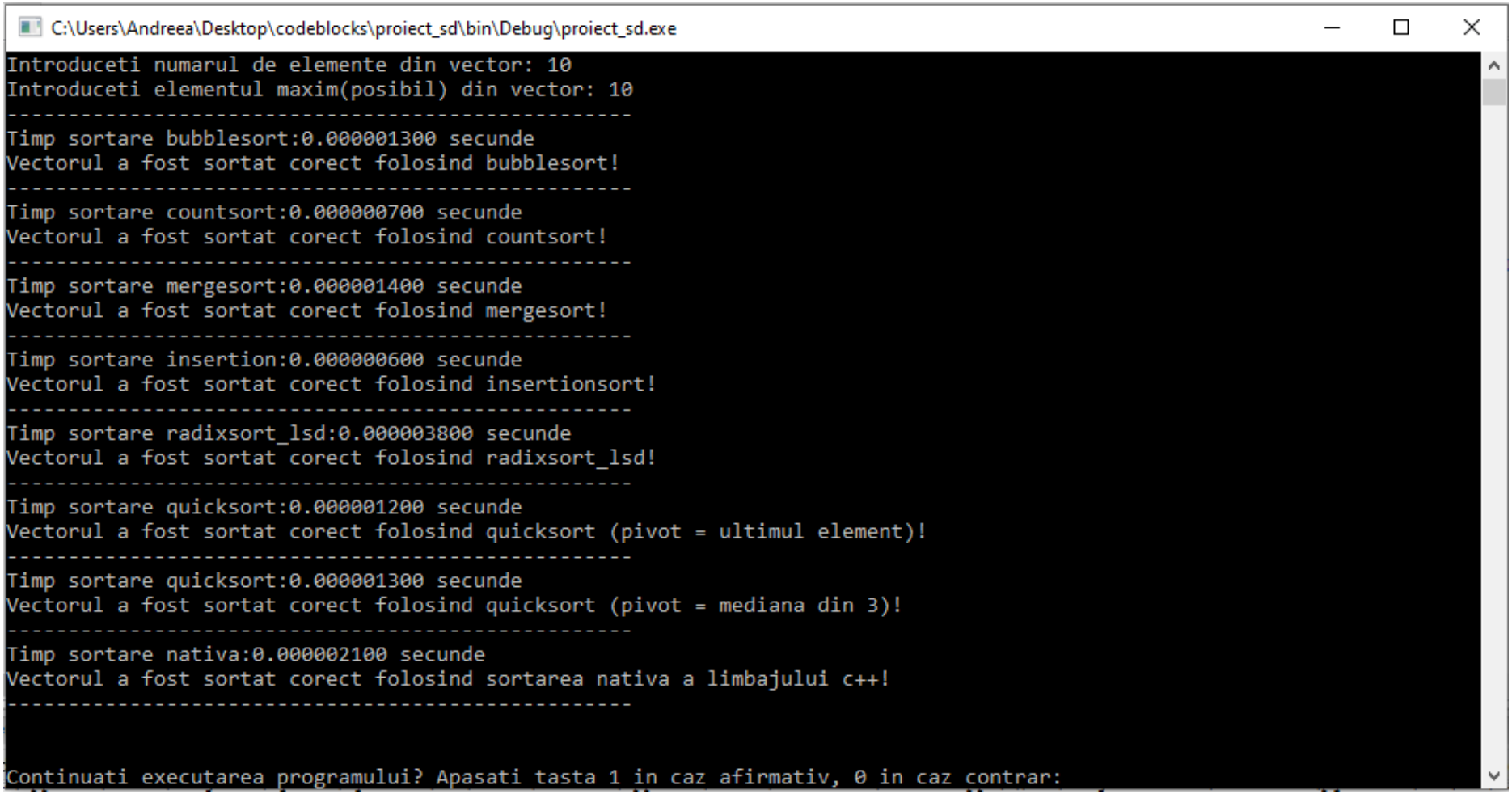
Tipuri de teste:

- ✓Vector aproape sortat crescător;
- ✓Vector aproape sortat descrescător;
- ✓Vector random;
- ✓Vector constant (din elementul maxim);
- ✓Vector de double.

În continuare vom analiza timpul de rulare pentru fiecare algoritm, mai în detaliu pentru un vector random.

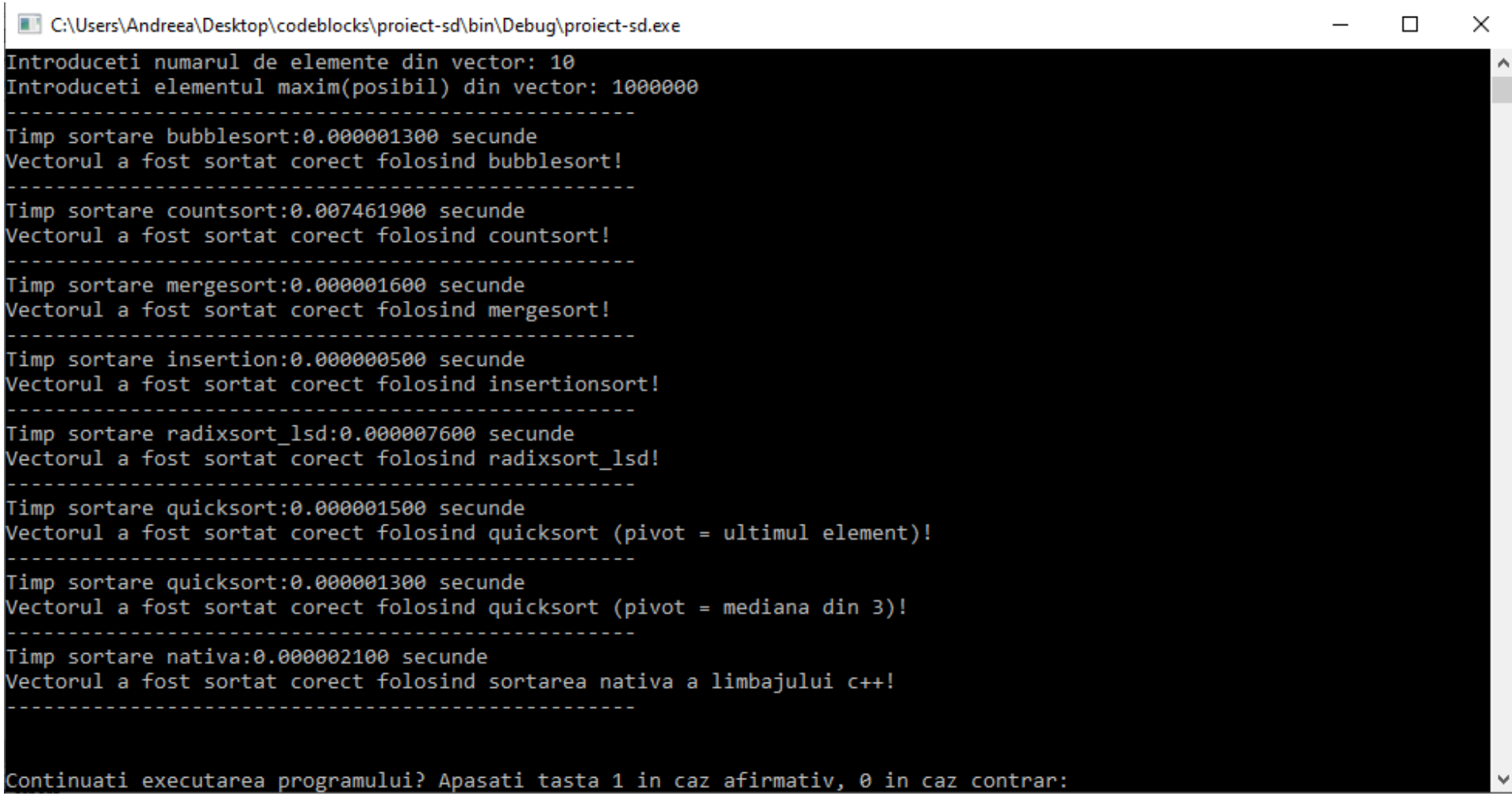
Observație: Timpii nu vor fi întotdeauna la fel, acest lucru va fi influențat si de forma vectorului generat!!

Lungime=10/Maxim=10



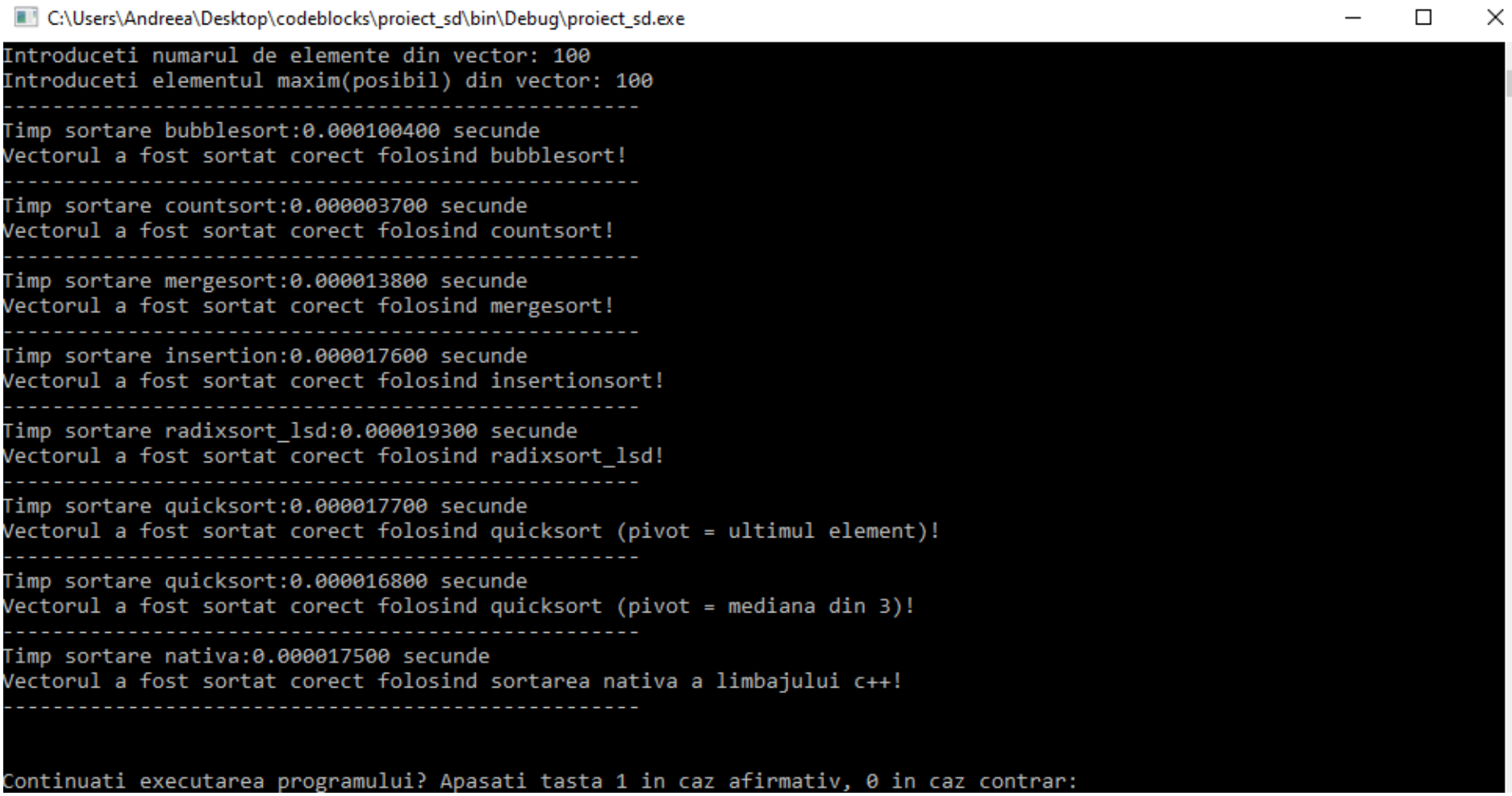
Pentru început, toți algoritmii se mișcă foarte rapid, însă evident putem observa diferențe între CountSort/InsertionSort și celelalte metode de sortare. Pentru un agoritm de complexitate O(n^2), InsertionSort prezintă un timp foarte bun pentru un număr așa mic de elemente.

Lungime=10/Maxim=1000000



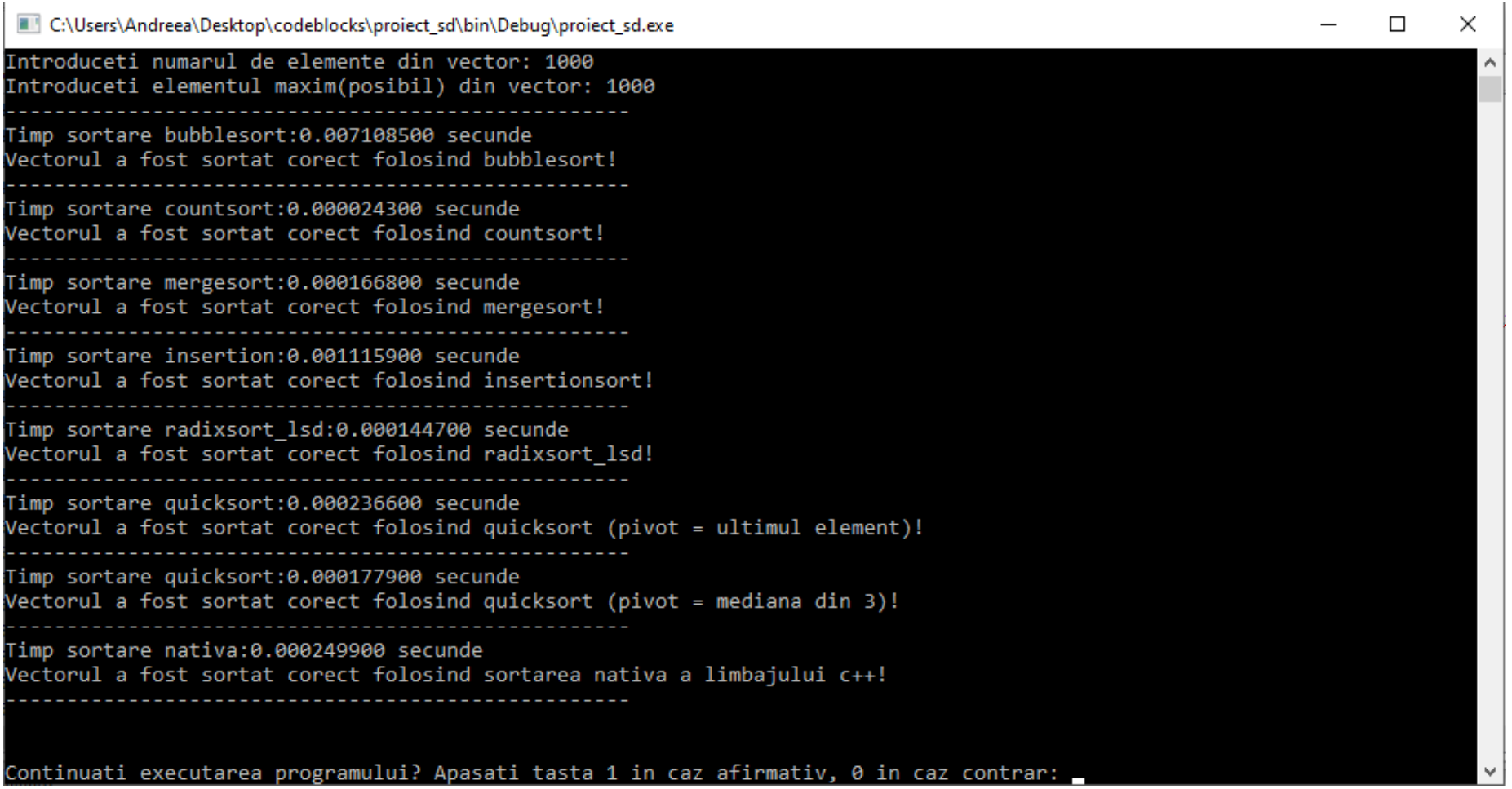
Acest test evidențiază faptul că, în ciuda numărului mic de elemente, dacă maximul este un număr mare, CountSort își crește foarte mult timpul de executare în comparație cu vectorul de 10 elemente (10 maximul). Ceilalți algoritmi au un timp similar.

Lungime=100/Maxim=100



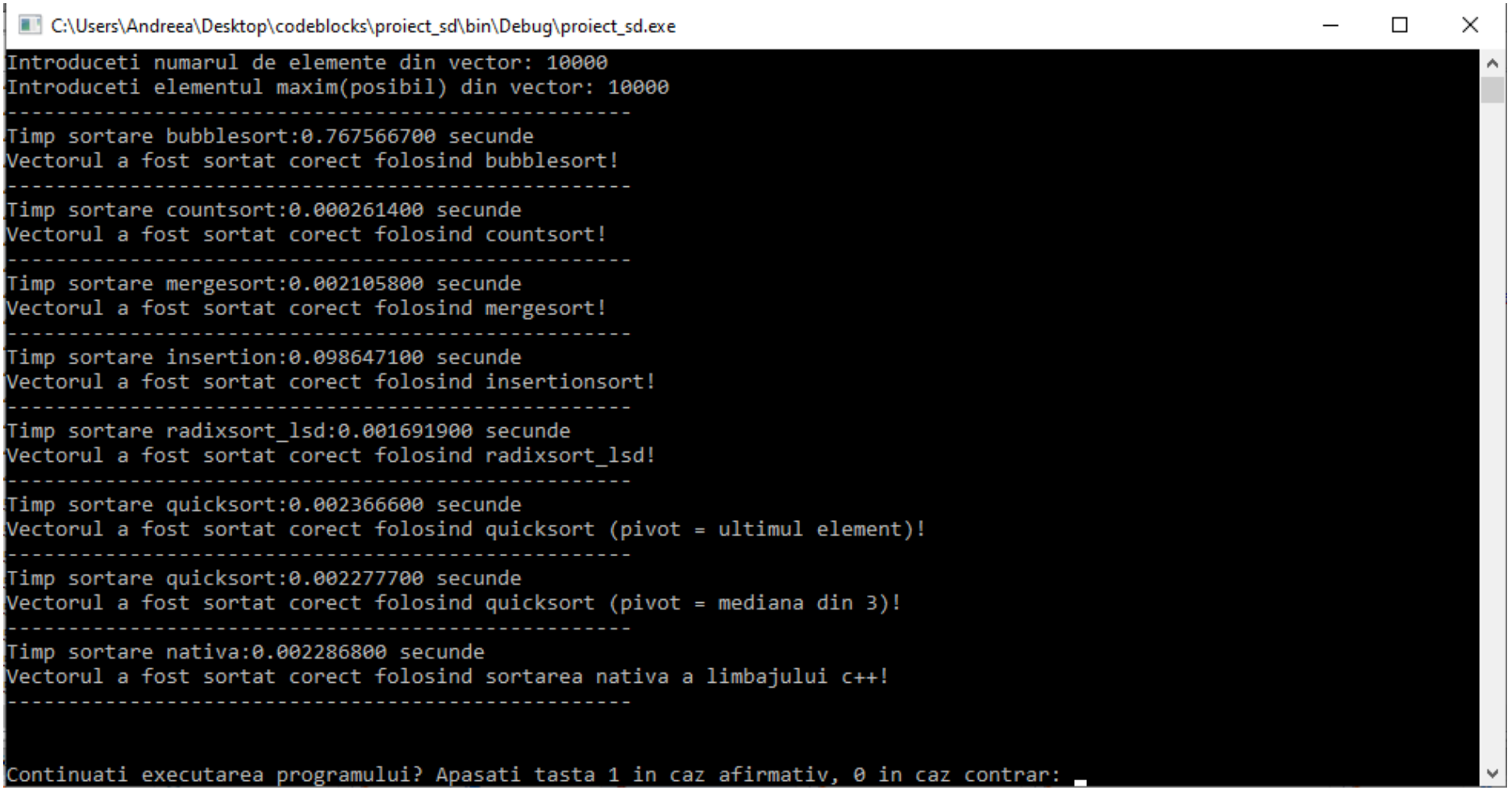
Se observă o creștere destul de bruscă între BubbleSort pe 10 elemente și pe 100 de elemente. Începe să se stabilizeze și diferența între RadixSort și QuickSort. CountSort pare să fie cel mai rapid, prezentând un timp foarte bun.

Lungime=1000/Maxim=1000



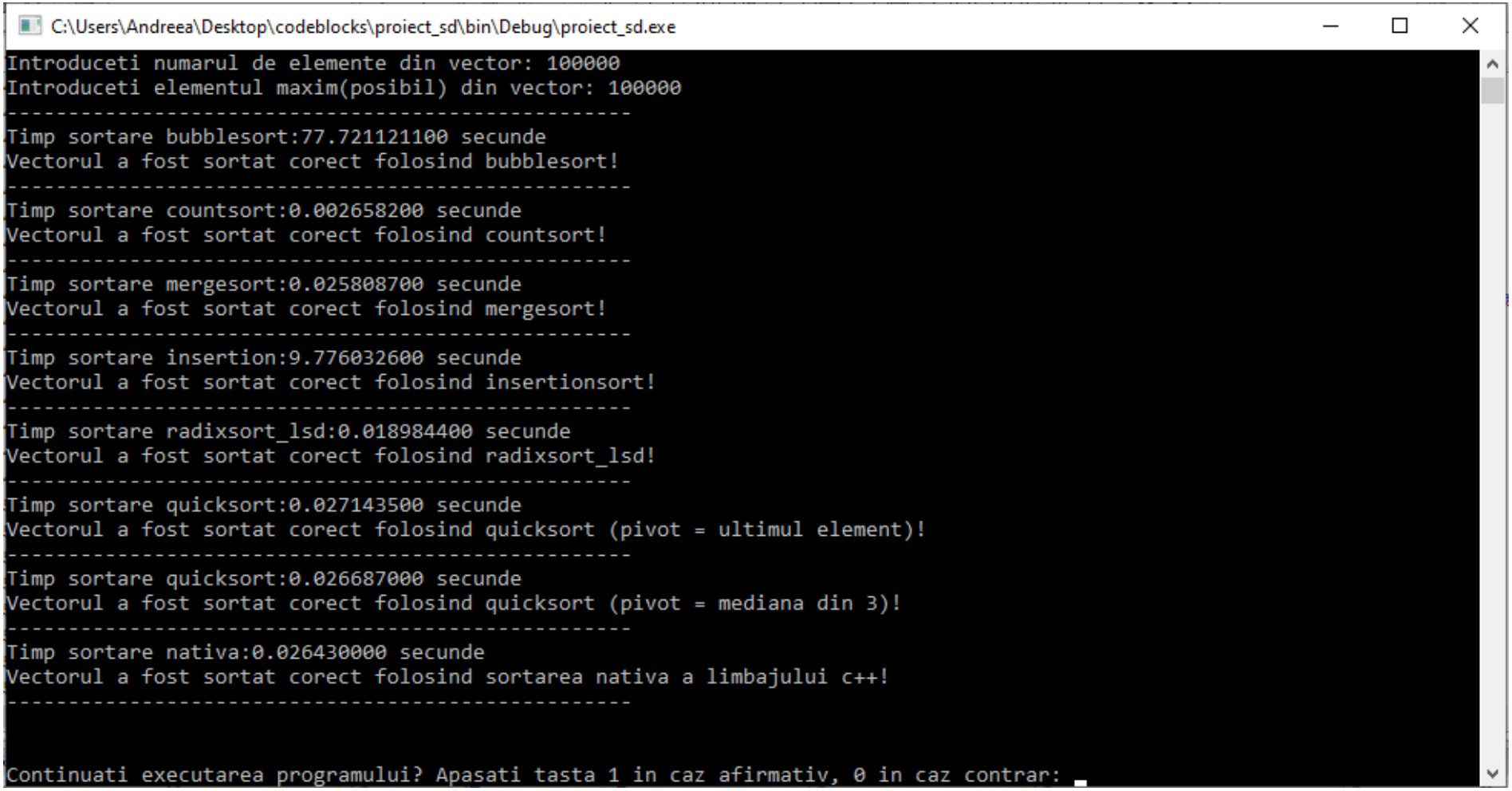
În continuare are loc o creștere pentru algoritmii cu complexitatea O(n^2) .

Lungime=10000/Maxim=10000



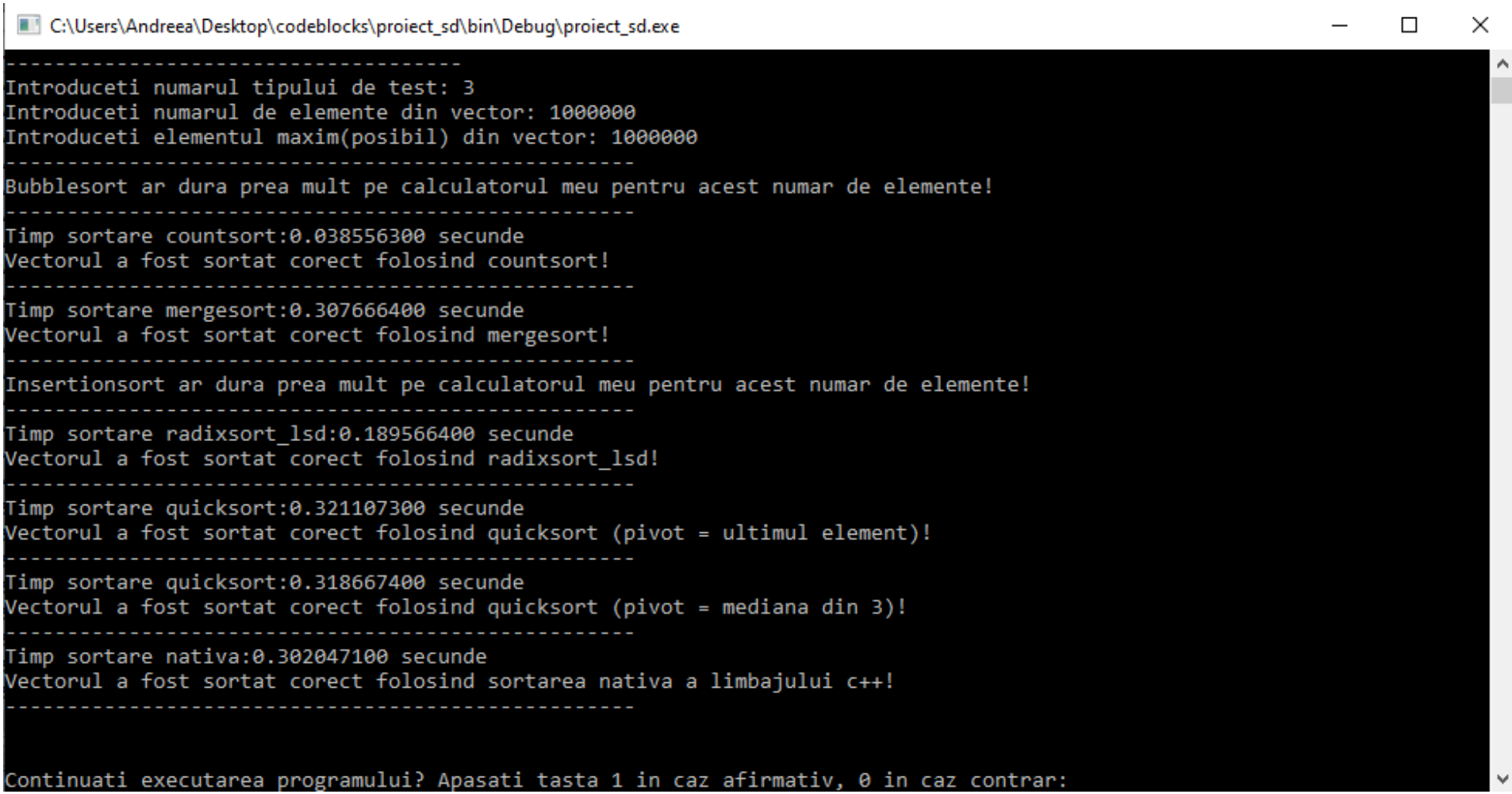
Analog cu slide-ul precedent.

Lungime=100000/Maxim=100000



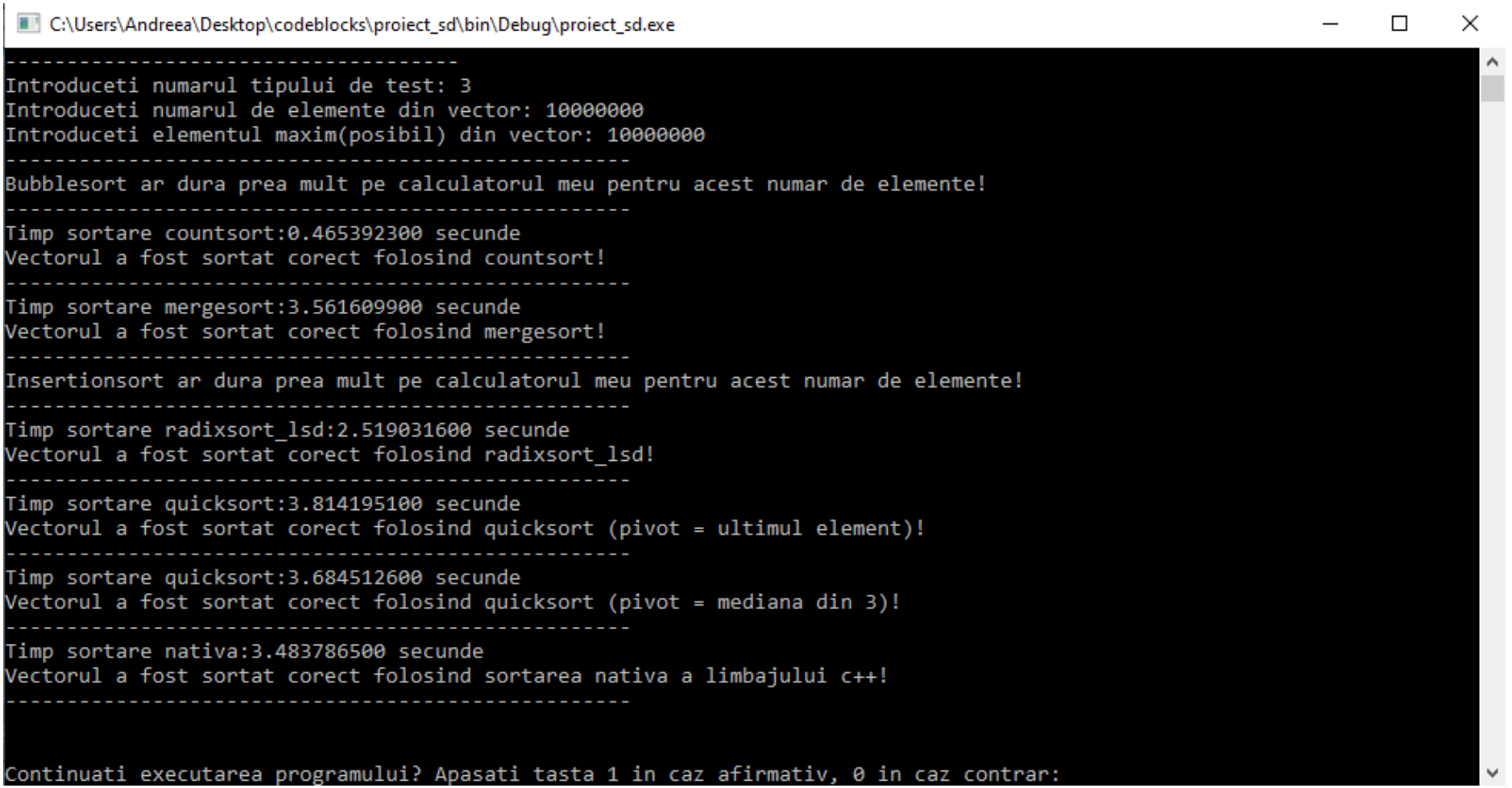
La acest număr de elemente, consider că devin total ineficiente BubbleSort și InsertionSort, sortarea luându-le câteva secunde. CountSort rămâne fruntaș în clasament.

Lungime=1000000/Maxim=1000000



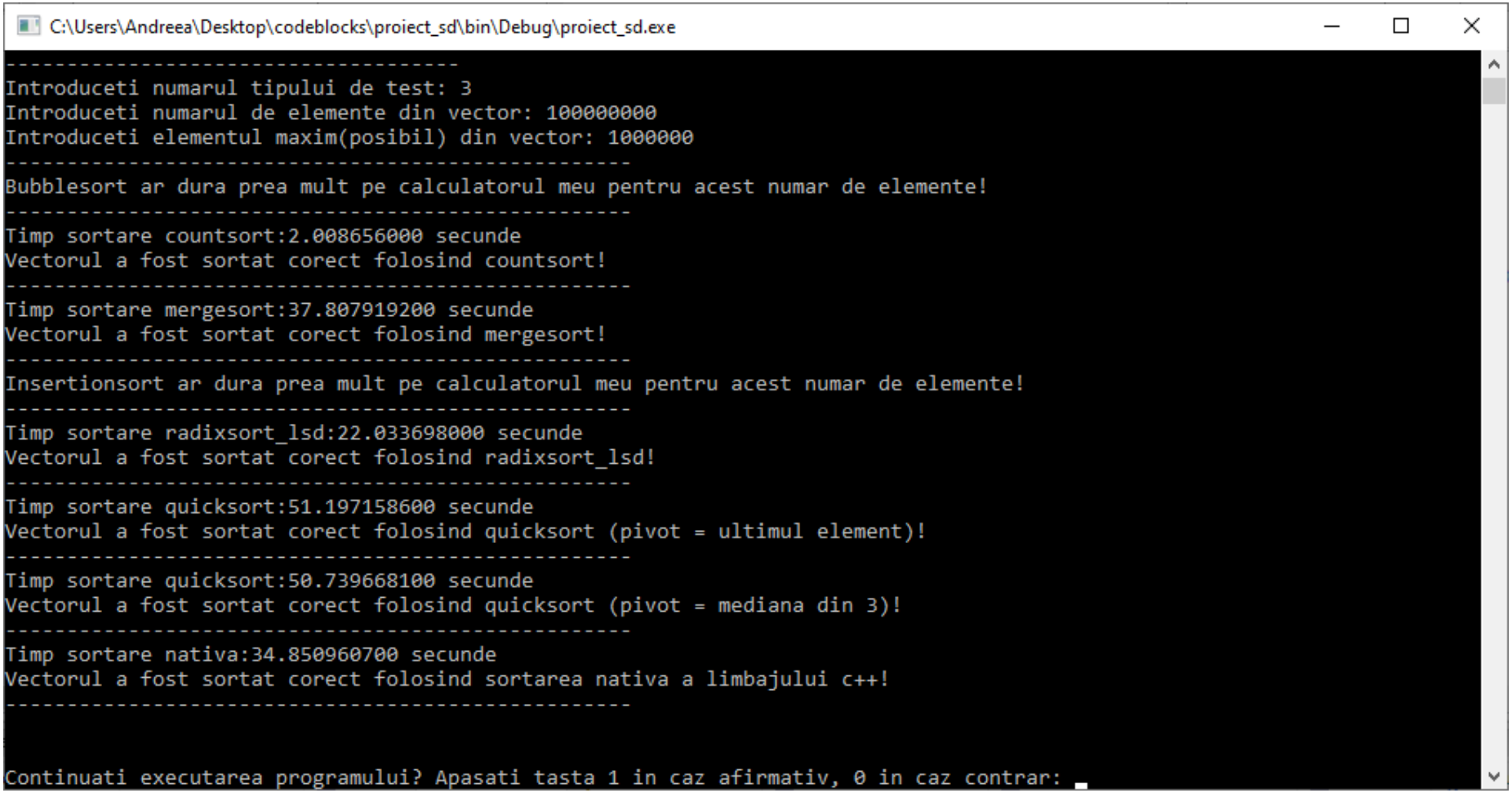
Pentru un milion de elemente, soluțiile optime rămân RadixSort și CountSort. Toate celelalte sortări sunt oarecum similare în timpul de executare.

Lungime=10000000/Maxim=10000000

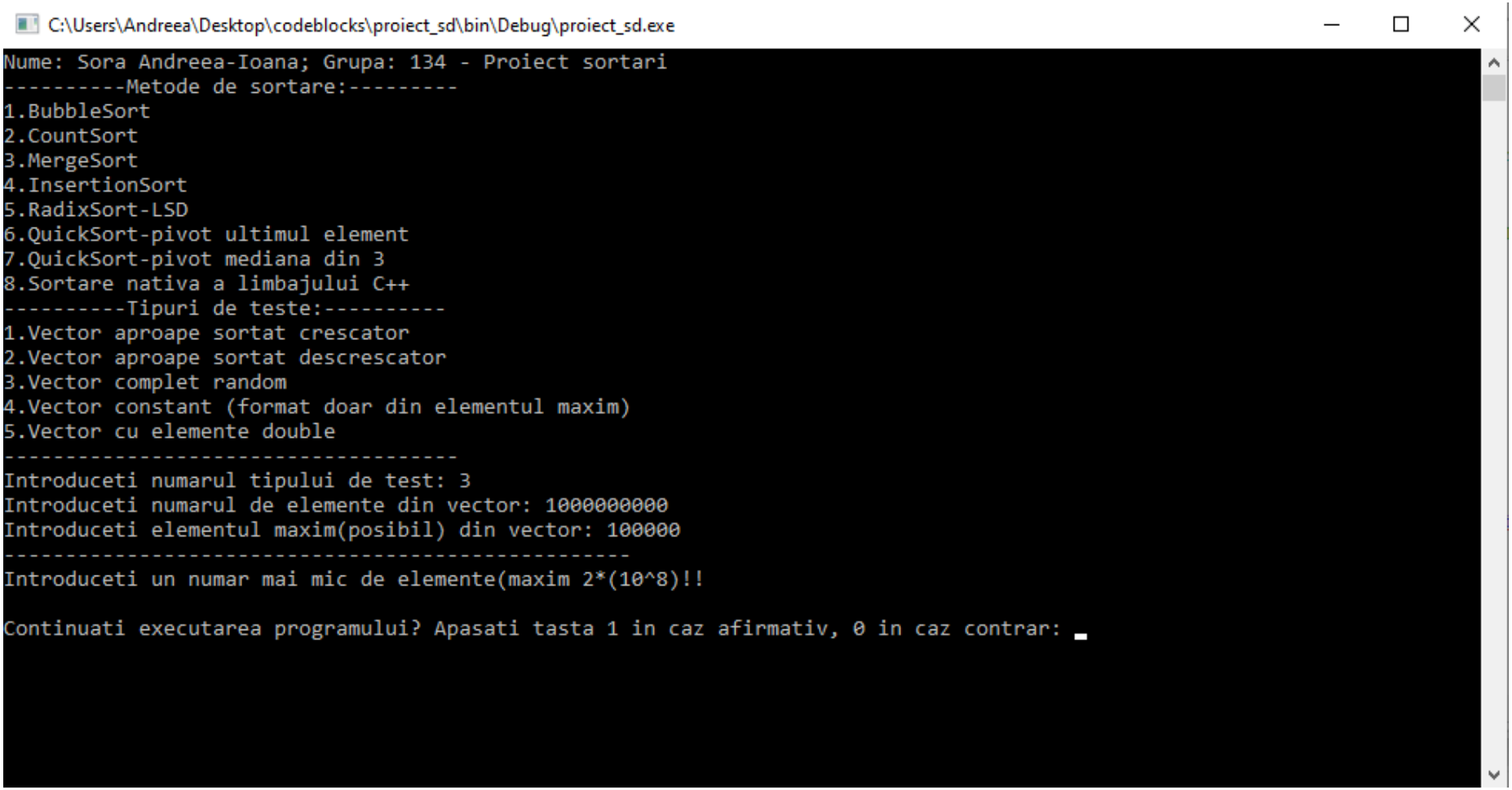


La acest pas, CountSort devine cel mai eficient, având un timp de mai puțin de jumătate de secundă, în comparație cu toți ceilalți care trec de două secunde.

Lungime=10^8/Maxim=1000000



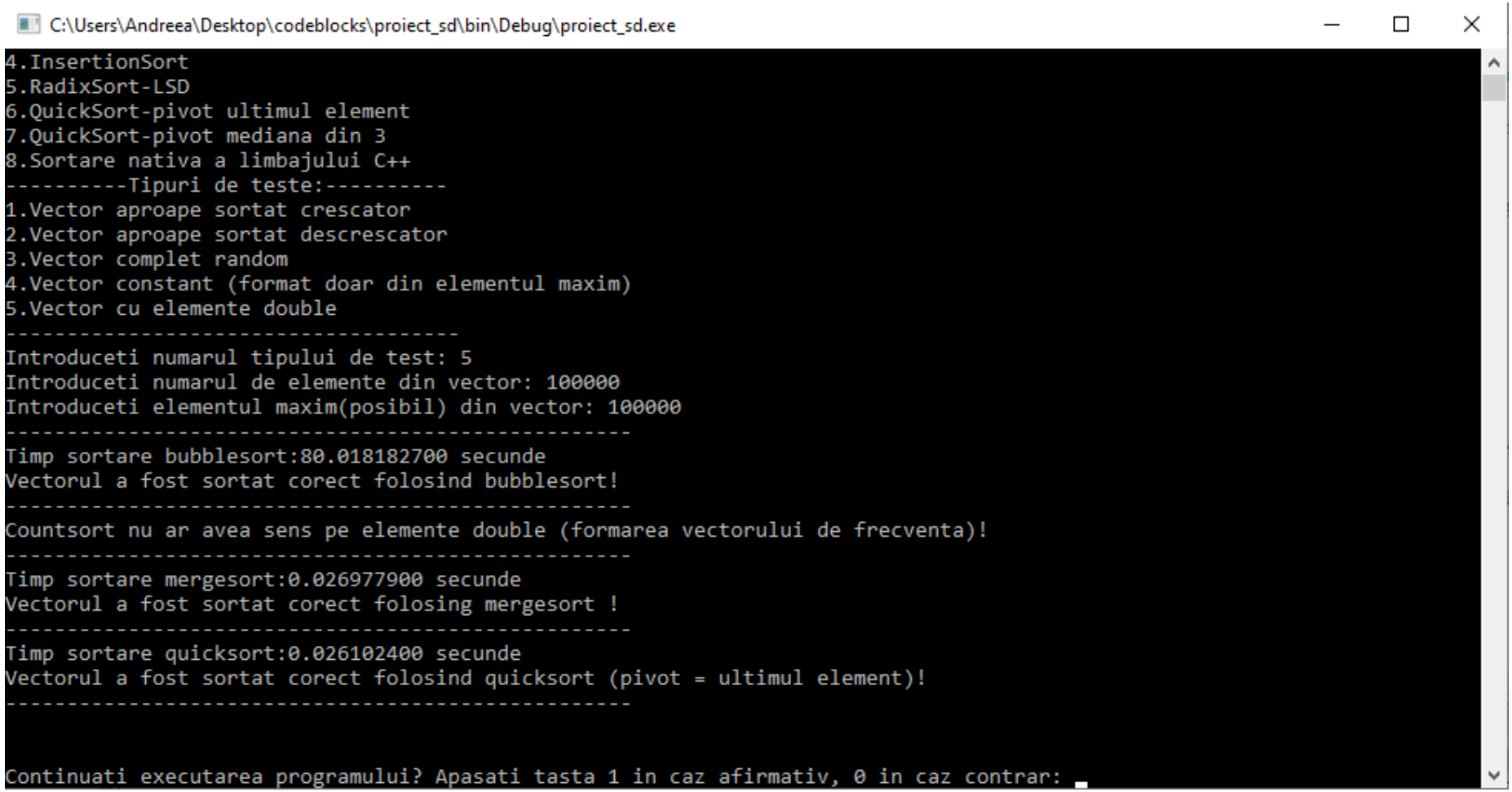
În concluzie, pentru 10^8 elemente toți algoritmii ating un interval de timp destul de ridicat, CountSort fiind de departe cel mai rapid.


C:\Users\Andreea\Desktop\codeblocks\proiect_sd\bin\Debug\proiect_sd.exe
Nume: Sora Andreea-Ioana; Grupa: 134 - Proiect sortari
----------Metode de sortare:---------
1.BubbleSort
2.CountSort
3.MergeSort
4.InsertionSort
5.RadixSort-LSD
6.QuickSort-pivot ultimul element
7.QuickSort-pivot mediana din 3
8.Sortare nativa a limbajului C++
----------Tipuri de teste:----------
1.Vector aproape sortat crescator
2.Vector aproape sortat descrescator
3.Vector complet random
4.Vector constant (format doar din elementul maxim)
5.Vector cu elemente double

Introduceti numarul tipului de test: 3
Introduceti numarul de elemente din vector: 1000000000
Introduceti elementul maxim(posibil) din vector: 100000

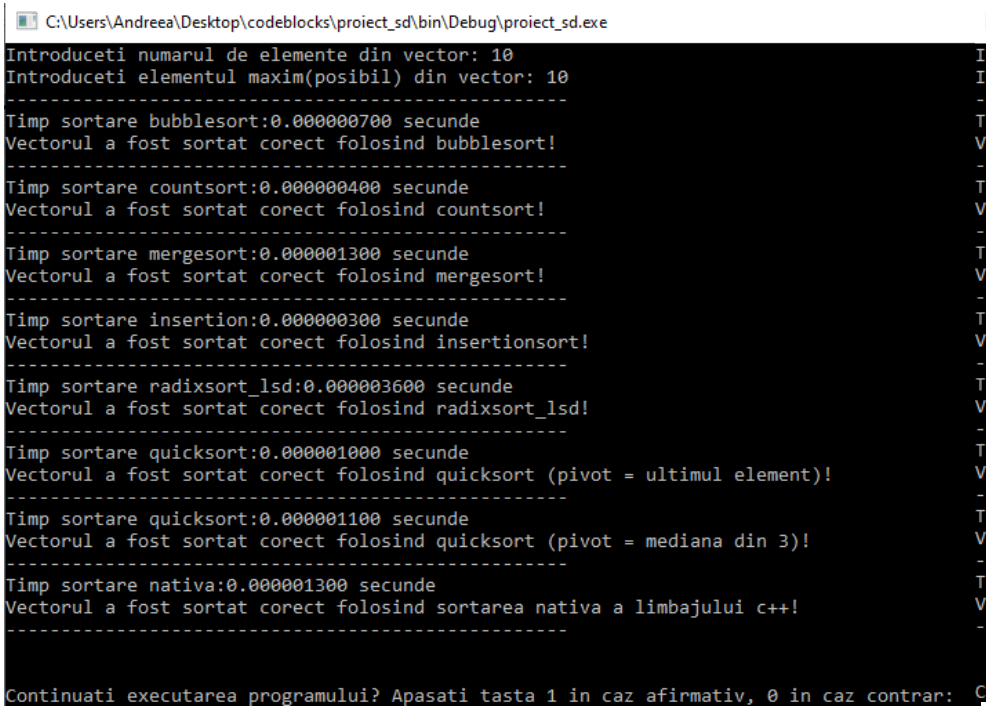
Introduceti un numar mai mic de elemente(maxim 2*(10^8)!!
Continuati executarea programului? Apasati tasta 1 in caz afirmativ, 0 in caz contrar:

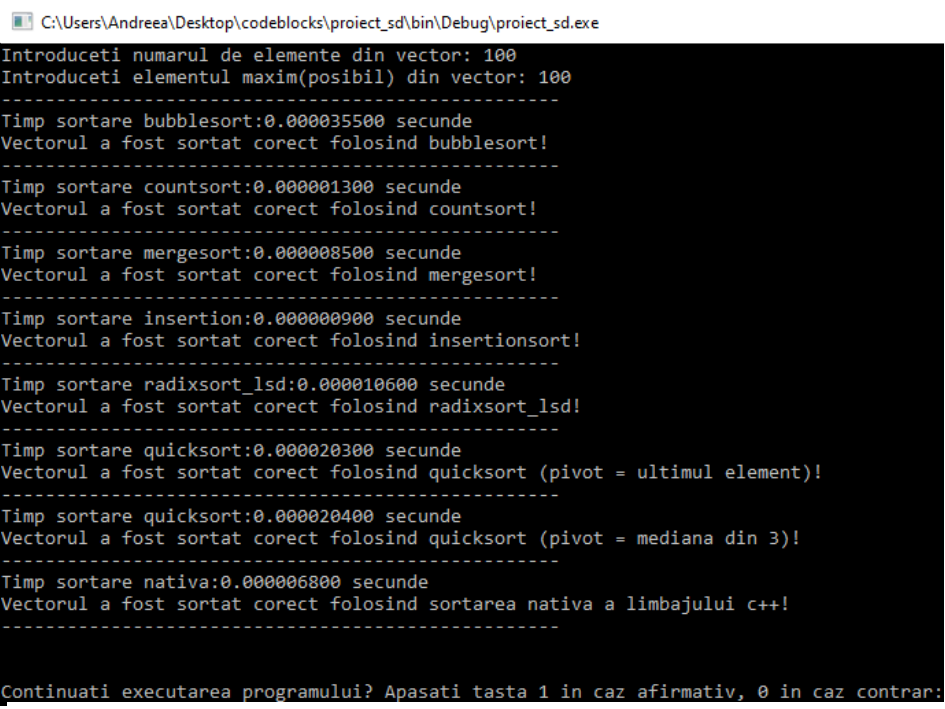
Alte tipuri de teste: double



Așa cum și pe elemente numere naturale BubbleSort devenea ineficientă de la pragul de 10^5 în sus inclusiv, acest lucru este valabil și pentru vectorul de double. Celelalte sunt similare cu vectorul de numere naturale.

Alte tipuri de teste: vector constant



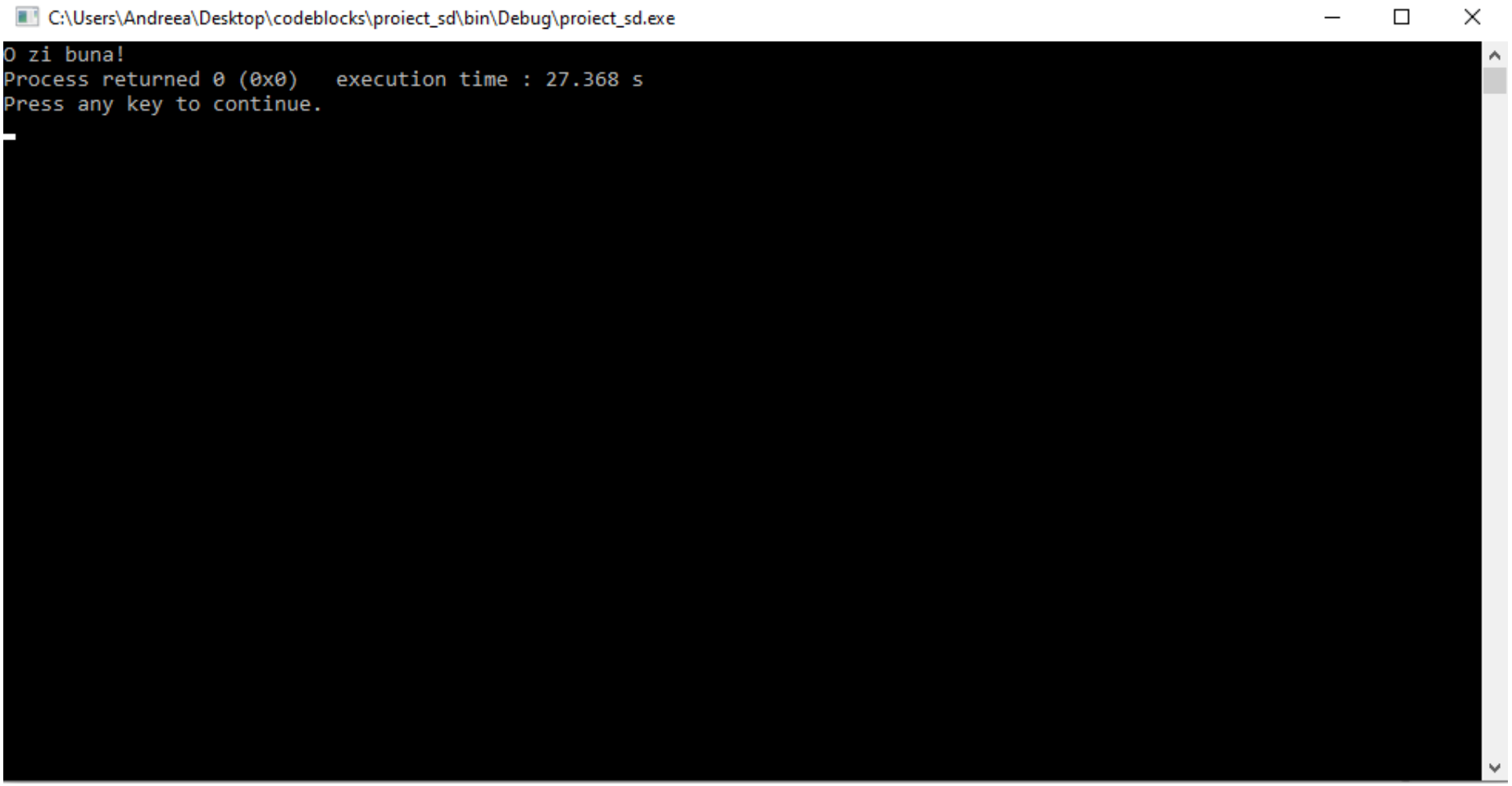


Între vectorii cu 10 și 100 de elemente se poate observa cum Radix, unde în primul test cu 10 elemente este mai slab decât Quick, devine mai rapid în cel cu 100 de elemente, trecere care în cazul numerelor random se întâmplă la un număr mai mare de elemente. Pentru un vector constant, QuickSortul nu pare să fie alegerea cea mai bună.

Alte tipuri de teste: vector aproape sortat crescător/descrescător

Pentru aceste tipuri de teste, timpii de rulare sunt similari cu cei analizați mai sus.

Sfârșit!


C:\Users\Andreea\Desktop\codeblocks\proiect_sd\bin\Debug\proiect_sd.exe
O zi buna!
Process returned 0 (0x0)   execution time : 27.368 s
Press any key to continue.