NUME: Petrovici Francesco-Gregorio
GRUPA: 152

Algoritmi de sortare

În cadrul acestei teme, am ales să implementez următorii algoritmi de sortare:

- Bubble sort, complexitate $O(n^2)$;
- Count sort, complexitate O(n);
- Quick sort determinist (cu pivot ales la mijloc), având o complexitate medie de O(n*log(n)), dar care poate degenera în $O(n^2)$;
- Quick sort randomizat (cu pivot aleatoriu), complexitate O(n*log(n));
- Merge sort, complexitate O(n*log(n)). Implementarea acestui algoritm nu mi-a reușit din punct de vedere al eficienței. O cauză posibilă a acestei probleme ar fi faptul că am folosit exclusiv vectori din STL (despre care am vorbit la curs că sunt mai lenți decât array-urile);
- Radix sort, complexitate O(k + n);
- Am mai implementat o funcție de sortare ce folosește „Priority Queue” din STL. La curs am înțeles că în spatele acestei „cozi de priorități” se află algoritmul Heap sort;

Conform cerinței, am creat pentru fiecare din acești algoritmi câte o funcție, ce folosește vectori din STL. Aceste funcții au nume sugestiv: bubble_sort, count_sort, Qsort1 (cel determinist), Qsort2 (cel randomizat), merge_sort, radix_sort, priority_sort (funcția ce folosește Priority Queue). Proiectul poartă numele de „stl_vector”, întrucât acesta reprezintă una din structurile de date cele mai folosite în cadrul temei mele.

Fișierul care conține testele se numește „input.txt” și are următoarea formă:

- T – numărul de teste
- N – numărul de elemente al unui test
- Elementele testului

Din păcate, am făcut testele înainte să fac tema și sunt fixe (nu generate la fiecare rulare în parte). Am folosit programul (defapt sunt câteva funcții pe care le-am făcut), ce se găsește în folderul „generator”.

Cele 7 teste concepute de mine:

- Testul 1 este unul de control având doar 1000 de numere, generate random;
- Testul 2 are 100.000 de numere cuprinse între 0 și INT_MAX, generate random, iar pe toate pozițiile unde se regăsește pivotul Quick sort-ului determinist am plasat valoarea 0.
- Testul 3 este la fel ca testul 2, doar că are 10.000.000 de numere.

- Testul 4, a fost generat într-o manieră deterministă și reprezintă o secvență de 10.000.000 de numere (prima jumătate este în ordine strict crescătoare, iar cea de-a doua în ordine strict descrescătoare).
- Testul 5, are 1.000.000 de numere cuprinse între 0 si 10^6, generate random (nu la fiecare rulare).
- Testul 6, are numărul 2001 de 1.000.000 ori.
- Testul 7, este o secvență de 1.000.000 de numere cuprinse între 0 si 10^6, ordonată strict crescător.

Rezultatele testelor

Sort	Testul 1 (secunde)	Testul 2 (secunde)	Testul 3 (secunde)	Testul 4 (secunde)	Testul 5 (secunde)	Testul 6 (secunde)	Testul 7 (secunde)
STL	0	0.032	5.94	23.73	0.56	0.48	0.35
Bubble	0.014	51.0965	>60	>60	>60	0	0.005
Priority	0.01	0.181	22.89	23.70	2.24	1.86	2
Count	0.0009	0	0.51	NU	0.59	0.049	0.05
Quick 1	0	0.06	6.78	5.79	0.69	0.61	0.40
Quick 2	0	0.033	4.52	9.96	0.46	0.37	0.34
Merge	0.0039	0.483	58???	57.31???	4.90	4.68	4.67
Radix	0	0.057	4.4	7.44	0.50	0.11	0.63

Prin analiza primului test, putem observa că pentru un număr redus de elemente (1000), toți algoritmii de sortare sunt eficienți.

Testul al 2-lea scoate în evidență lipsa de eficiență a algoritmului Bubble sort, care pentru 100.000 de numere, a ajuns la 51 de secunde.

Testul al 3-lea, activează protecția implementată în funcția bubble_sort (dacă timpul de execuție depășește 60 de secunde, sortarea se oprește, lucru ce poate fi observat și în testele 4 și 5). Observăm diferențe semnificative între cele două quicksort-uri (cel cu pivot determinist este mai lent cu 2 secunde, deoarece oriunde ar pica pivotul, am plasat valoarea 0, totuși nu degenerează în n^2). Count sort este cel mai eficient, fiind de aproape 12 ori mai rapid decât sort-ul din STL.

Testul al 4-lea reprezintă unul dintre cele mai favorabile cazuri pentru Quick sort-ul nostru determinist (pivotul va fi aproape întotdeauna un element „mediu”), astfel el devine brusc cel mai eficient dintre algoritmi. O supriză negativă este sort-ul din STL, care este lent pe o astfel de secvență de tip „munte”. În cadrul acestui test, protecția count_sort-ului este activată, deoarece a fost întâlnit un număr prea mare.

Testul 6, scoate în evidență faptul că bubble_sort este mai eficient decât ceilalți algoritmi, atunci când vine vorba de a sorta un array format dintr-un singur element.

În urma acestei analize, am rămas suprins să aflu că sort-ul din STL nu este întotdeauna cel mai eficient algoritm de sortare și că în anumite cazuri specifice, fiecare dintre algoritmi are avantajele sale.

Îmi cer scuze, dar nu pot încărca fișierul „input.txt” pe GitHub, deoarece are peste 100mb. Este vina mea, nu văzusem pe exemplul de teste ca ele trebuie să fie generate aleatoriu în program, eu le-am generat separat și le-am pus pe toate în input. Voi lăsa și un link cu un folder de Google Drive cu întreaga temă.