Analiză comparativă a algoritmilor de sortare

Introducere:

Sortarea unui șir reprezintă o problemă de bază pentru manipularea datelor. De cele mai multe ori, e mult mai convenabilă utilizarea algoritmului de sortare nativ al limbajului de programare utilizat, însă, în momentele în care avem seturi de date cu anumite proprietăți sau doar vrem să implementăm manual o sortare convenabilă, avem de unde alege. Fiecare algoritm are punctele sale tari sau slabe și, în funcție de problemele care trebuie rezolvate, pot fi un real ajutor sau doar o mai mare bătaie de cap. Ne propunem să analizăm diferențele între mai multe tipuri de implementări ale acelorași algoritmi de sortare, cât și diferențele dintre algoritmi diferiți.

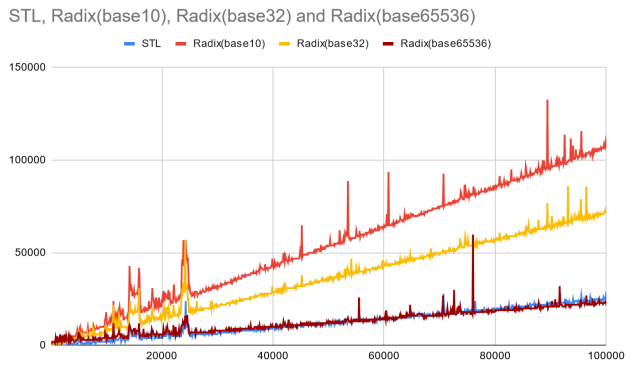
Știm că un algoritm de sortare nu avea o complexitate mai mică de O(n*logn) (demonstratie). Dar aceeași complexitate nu implică neapărat și același timp de rulare. Mulți algoritmi pot fi foarte eficienți pe date de intrare favorabile (Quicksort, Bubblesort) sau altele care să nu difere ca timp de rulare în funcție de setul de date prelucrat (Mergesort).

RadixSort:

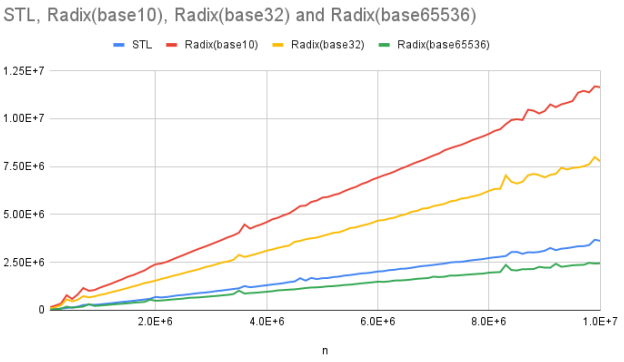
Radix Sort funcționează prin sortarea numerelor în funcție de cifrele acestora, începând de la cea mai puțin semnificativă cifră (cifra unităților). În general, pentru un set de date de intrare cu n numere de maxim d cifre, în care fiecare cifră poate lua maxim k valori, complexitatea algoritmului este de O(d*(n+k)), dacă algoritmul folosit pentru sortarea după fiecare cifră se realizează în O(n+k). Cu toate acestea, pentru un set de n numere și un număr pozitiv r≤b, complexitatea este $O((b/r)*(n+2^r))$, dacă algoritmul folosit pentru sortarea după fiecare cifră se realizează în O(n+k). Alegând r≤lg n , atunci ajungem la o complexitate liniară, O(n).

Ca implementare, am optat pentru o sortare prin numărare pentru bucketurile create de cifre. Putând varia baza de lucru, observăm că bazele care reprezintă puteri ale lui doi funcționează chiar mai bine decât sort-ul implicit din STL.[1]

2



3



[1] Pentru consecvență, am folosit pentru toate testele ca maxim numărul 92233720368547758O7, adică cel mai mare număr care se poate scrie cu ajutorul tipului de date long long din C++.

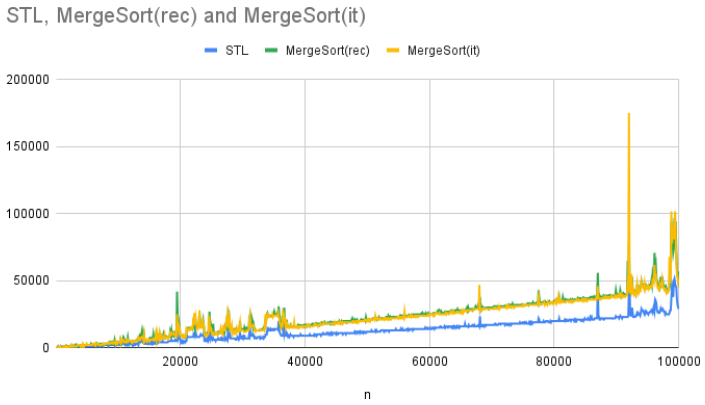
[2] Range-ul pentru numărul de elemente este de la 100 la 100000, cu pas 100.

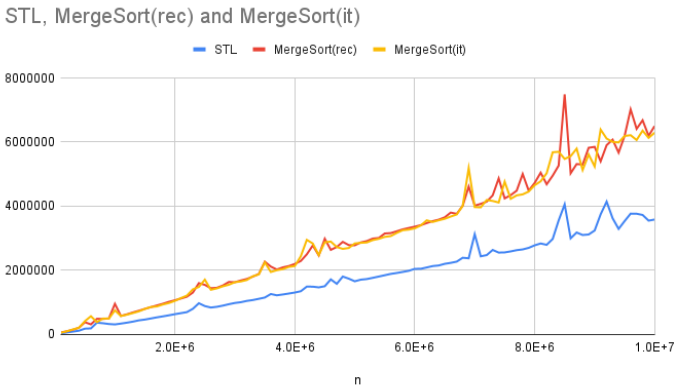
[3] Range-ul pentru numărul de elemente este de la 100000 la 10000000, cu pas 100000.

Putem observa că, pentru testele de început, baza 2^16 funcționează semnificativ mai lent decât celelalte, explicația fiind că, pentru aceste cazuri, cum n nu este suficient de mare, acel număr pozitiv r≤b este relativ mic, deci performanța nu va fi deloc îmbunătățită. Pe măsură ce n crește, observăm, de asemenea, o îmbunătățire semnificativă, fiind aproximativ la fel de eficient ca sortarea nativă din C++.

MergeSort:

MergeSort se bazează pe împărțirea recursivă a unui vector până se ajunge la vectori de un element (deja sortați), folosind interclasarea pentru construirea soluției.Deși are complexitate tot O(n*log n) constantă (nu depinde de cazuri particulare ale datelor de intrare), are dezavantajul că folosește memorie suplimentară.





Observăm că varianta iterativă a algoritmului nu aduce o îmbunătățire semnificativă (relativ contraintuitiv, deoarece eliminarea recursivității reduce din apelurile de funcții, deși numărul de operații ar părea să crească pe anumite cazuri).

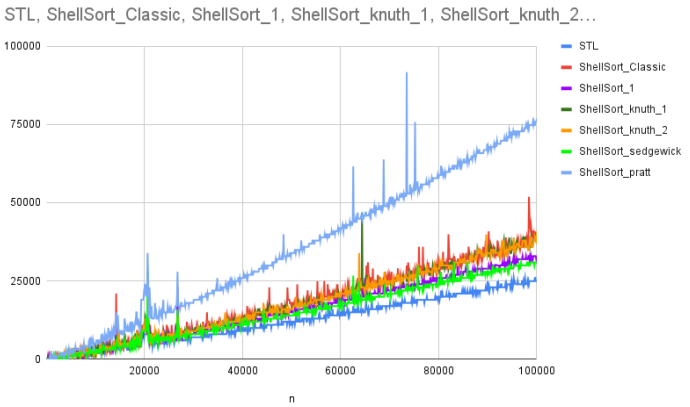
ShellSort:

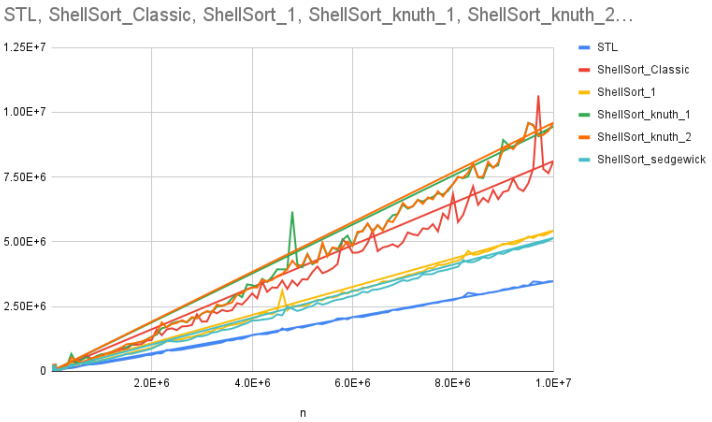
ShellSort este o îmbunătățire a InsertSort-ului, folosind ideea că, dacă InsertSort-ul e eficient pe date aproape sortate, care nu trebuie să se deplaseze prea mult, ShellSort permite elementelor să facă salturi mai mari, aducând astfel un plus de eficiență. Există totuși mai multe secvențe de numere care produc gap-uri eficiente.[4]

Implementările propuse folosesc secvențele 1, 5, 6, 7, 8. Am observat că folosind secvența Pratt avem cel mai mare timp de rulare (caz pentru care am decis și să îl eliminăm din a doua tură de teste). Cel mai eficient s-a dovedit a fi rutina care folosește secvența lui Sedgewick, probabil pentru că gap-urile pe care le permite sunt mai mari decât restul. O explicație mai cuprinzătoare poate fi găsită aici.

[4] https://en.wikipedia.org/wiki/Shellsort

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A000225	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Knuth, 1973,[3] based on Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_0 = 3$ $a_q = \min\left\{n \in \mathbb{N} : n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p : 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\right\}$ $I = \left\{0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k\right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8\ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1982[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{\left\lfloor \frac{5h_{k-1} - 1}{11} \right\rfloor, 1\right\}, h_0 = N$	$\left\lfloor \frac{5N-1}{11} \right\rfloor, \left\lfloor \frac{5}{11}\left\lfloor \frac{5N-1}{11} \right\rfloor - 1 \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right)\right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]



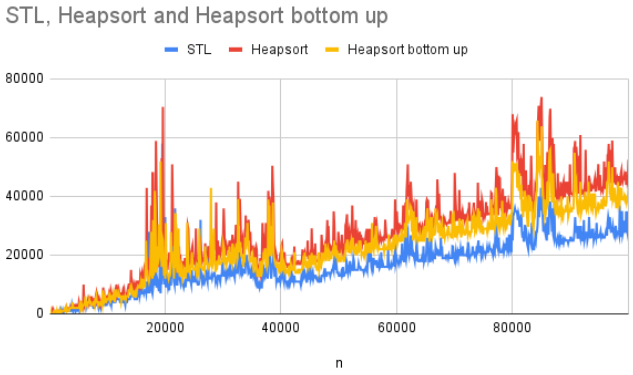


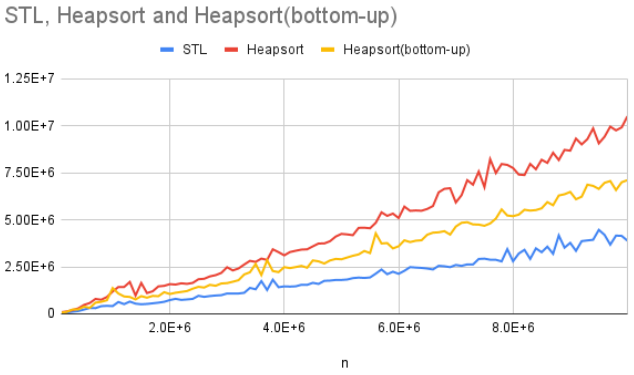
HeapSort:

HeapSort poate fi privit ca o versiune îmbunătățită a SelectSort-ului, căci heapsort împarte vectorul, la fiecare pas, în regiune cu elemente sortate și regiune cu elemente nesortate, iar la fiecare pas este ales cel mai mare element din zona nesortată și pus în poziția corespunzătoare. Spre deosebire de SelectSort, HeapSort reține mereu zona care trebuie sortată într-o structură de arbore binar (de unde și denumirea algoritmului), răspunzând mult mai rapid la partea de selecție a maximului.

De asemenea, există și o versiune îmbunătățită a acestui algoritm care, teoretic, ar trebui să fie mai rapid decât Quicksort pe anumite teste. Mai multe detalii pot fi găsite în acest articol.

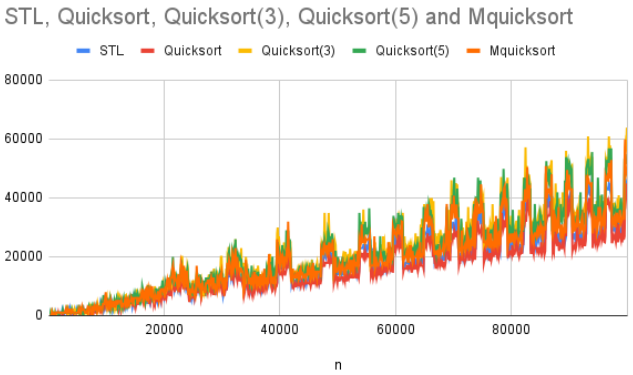
Interesant este faptul că varianta clasică de heapsort funcționează mai bine decât sort-ul din C++(a fost făcută realizată o verificare după sortare cu vectorul sortat de sort-ul implicit, iar rezultatele erau aceleași).

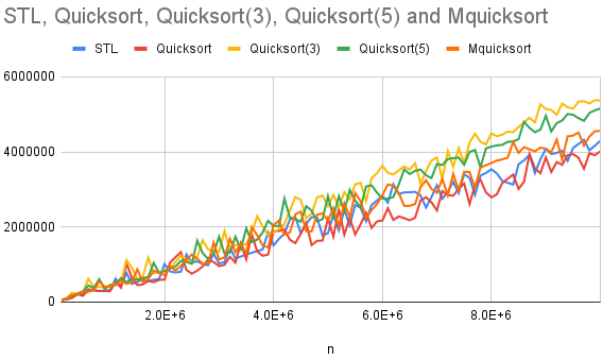




Quicksort:

Quicksort este cel mai utilizat algoritm de sortare, funcționând tot pe principiul divide-et-impera (ca și MergeSort, doar că fără memorie suplimentară). La fiecare pas se alege un pivot care împarte vectorul inițial în două, iar prin apeluri recursive este sortată fiecare parte . Niște articole interesante legate de acest algoritm am găsit aici, aici și aici.
Din motive nu foarte clare, varianta de alegere dinamică a pivotului nu funcționează pe vectori cu mai mult de 10^5 elemente cu elemente maxim 92233720368547758O7 (funcționează pe 10^8 elemente cu 20000000 ca maximul din array - maxim pentru care s-a realizat al doilea grafic, unde observăm că alegerea dinamică a pivotului e mai eficientă decât alegerea medianei din 3 sau din 5 pe majoritatea cazurilor).





Notabil este faptul că alegerea pivotului ca mediană a trei, respectiv cinci elemente nu aduce o îmbunătățire algoritmului clasic, când alegem pivotul elementul din mijloc (cel puțin pe date de intrare aleatoare).

Concluzii:

Observăm că, în principiu, cel mai eficient (comparativ cu sort-ul din C++, pe care îl presupunem relativ constant) este quicksort. Diferența notabilă de optimizare se observă la Heapsort, unde abordarea bottom-up conduce la valori mult mai apropiate de stl sort decât varianta clasică. În rest, cu excepția ShellSort-ului și a RadixSortului (baza 2^16), diferențele de optimizare sunt, în principiu, nesemnificative per total, deși pentru teste individuale se poate vedea o diferență considerabilă.