Algoritmi de sortare

**1. Algoritmi de sortare cu complexitate de O(n2):**

* Bubble sort
* Pancake sort
* Insertion sort

Graficul ilustrează timpii de rulare a celor 3 algoritmi pe vectori de diferite dimensiuni, cu numere generate în mod aleator, din intervalul [0, 109].

**Observații:**

- Atunci când vectorul este sortat crescător, bubble sort și insertion sort au complexitate liniară, fiind foarte rapizi, dar pancake sort are o complexitate de n2 (timpul de rulare fiind de 1048 ms pe un vector cu 104 elemente);

- Dacă vectorul este sortat descrescător, atât bubble sortul cât și pancake sortul intră pe cazul cel mai nefavorabil, un vector cu 104 elemente fiind sortat în aproximativ o secundă în ambele cazuri. Insertion sortul are și el un timp de executare mai mare în acest caz, însă este mult mai eficient decât ceilalți doi algoritmi (aproximativ 0.4 secunde pe un vector cu aceeași dimensiune);

- Dacă majoritatea valorilor din vector se repetă, este de preferat folosirea insertion sortului sau a bubble sortului deoarece aceștia scot timpi semnificativi mai buni la execuție față de pancake sort (bubble sort - 2 ms, pancake sort - 690 ms, pe un vector de 104 elemente) ;

- Spre deosebire de bubble sort și insertion sort, pancake sort nu este o sortare stabilă;

- Niciuna dintre aceste metode de sortare nu folosește spațiu de memorie suplimentar;

- Dacă vectorul este generat aleator, pancake sortul și bubble sortul obțin aproximativ aceeași timpi, însă dacă numărul de inversiuni este mic, bubble sortul devine mult mai eficient ( bubble sort – 2 ms, pancake sort – 670 ms pe un vector sortat crescător de 104 elemente unde au fost interschimbate 2 perechi de numere);

- Insertion sortul se comportă foarte bine atunci când numărul de elemente este o putere a lui 2. Din acest motiv insertion sortul este folosit în Timsort unde se sortează prin inserție subsecvențe de 32 elemente din vectorul inițial, iar apoi subsecvențele vor fi sortate la rândul lor prin merge sort.

**2. Algoritmi de sortare recursivi:**

* Merge sort
* Quick sort (cu pivot ales aleator)

Graficul ilustrează timpii de rulare a celor 2 algoritmi pe vectori de diferite dimensiuni, cu numere generate în mod aleator, din intervalul [0, 109].

**Observații:**

- Cei doi algoritmi au implementări recursive și abordează problema sortării unui vector de elemente într-o manieră de timp Divide et Impera;

- Ambii algoritmi au complexitate de n\*log n pe cazul general (quick sortul poate ajunge la n2 pe cazul cel mai nefavorabil); fiind mai eficienți decât algoritmii prezentați la punctul anterior, merge sortul și quick sortul sunt capabili să sorteze și vectori de mai mult de 104 elemente într-un timp bun;

- Pentru vectorii cu mai puțin de 10^5 elemente nu există diferențe semnificative între cei doi algoritmi, în ceea ce privește timpul;

- În cazul quick sortului, pivotul a fost ales în mod aleator pentru a se evita cât de mult posibil cazurile nefavorabil (atunci când vectorul era deja sortat crescător sau descrescător sau atunci când valoarea maximă din vector este 1 sau 2) ;

- Se observă că merge sortul este mai lent decâ quick sortul deoarece merge sortul nu este un algoritm de sortare „in place”, iar copierea elementelor din vectorul inițial în vectorul auxiliar și invers încetinesc algoritmul. De asemenea, merge sortul este mai puțin eficient și din punctul de vedere al spațiului de memorie utilizat, din același motiv;

- Spre deosebire de quick sort, merge sortul este un algoritm de sortare stabil și de aceea este adesea preferat atunci când trebuie sortați vectori cu elemente cu mai multe proprietăți, dacă se dorește păstrarea ordinii din vectorul inițial în cazurile de egalitate.

**3. Algoritmi de sortare fără comparări:**

* Count sort
* Radix sort cu baza 10
* Radix sort cu baza 256 (și operații pe biți)

Prima diagramă înregistrează timpul obținut în funcție de lungimea totală a vectorului (elementele fiind de tip int în acest caz), iar a doua înregistrează timpii obținuți în funcție de intervalul din care fac parte elementele vectorului (pe un vector de 106 elemente).

**Observații:**

- Spre deosebire de algoritmii preznetați până acum, counting sort și radix sort nu compară elementele între ele ci le pun în niște containere după care extrag elementele în ordinea corectă;

- Radix sortul și counting sortul nu sunt sortări stabile;

- Counting sortul are o complexitate de n + vmax (unde n – numărul de elemente și vmax – valoarea maximă din vector); Algoritmul este foarte rapid din punctul de vedere al timpului de excuție, având complexitate aproape liniară însă necesită foarte multă memorie suplimentară atunci când intervalul de valori al elementelor este foarte mare;

- Complexitatea radix sortului este O((n+b) \* logb(vmax)) (unde n – numarul de elemente, b – baza radix sortului, vmax – valoarea maximă din vector); radix sortul cu bază 256 este mai eficient decât cel de bază 10 deoarece are baza mai mare și a fost implementat folosind operații pe biți care sunt mai rapide decât cele aritmetice;

- Se pot sorta doar șiruri de numere cu radix sort;

- Deoarece nu folosesc comparații, cei doi algoritmi nu au cazuri nefavorabile și obțin aproximativ același timp indiferent de ordinea elementelor din vector.

**3. Algoritmi de sortare vs algoritmul nativ din C++:**

Graficul ilustrează timpii de rulare a celor 4 algoritmi pe vectori de diferite dimensiuni, cu numere generate în mod aleator, din intervalul [0, 10^5).

**Observații:**

- Deși count sortul este mult mai rapid deât ceilalți doi algoritmi, acesta necesită foarte multă memorie suplimentară și nu mai reprezintă un algoritm de sortare practic dacă mărim intervalul din care iau valori elementele și dacă elementele vectorului nu se repetă de prea multe ori;

- Dacă lungimea vectorului este < 106 , quick sortul este mai rapid decât radix sortul, însă după pragul de 106 numere radix sortul devine mai eficient.