Tema 1 de laborator

Aioanei Florin Verzotti Matteo Voaides Robert

Compararea unor algoritmi de sortare prin comportamentul lor asupra mai multor suite de teste

Cuprins

1 Introducere

În această lucrare ne propunem să comparăm performanțele mai multor algoritmi de sortare din punct de vedere al complexității teoretice și al performanței practice. Scopul este de a evidenția avantajele și dezavantajele fiecărui algoritm în funcție de dimensiunea și natura datelor de intrare.

2 Algoritmi analizați

- Quick Sort
  - Random Pivot
  - Median Pivot
  - Half Pivot
  - Ternary Quick Sort
- Radix Sort
  - base 10
  - base 16
  - base 2^{16}
- Merge Sort
- Intro Sort
- Tim Sort
- Shell Sort

3 Complexitate teoretică

Tabela 1: Tabel comparativ al complexităților algoritmilor analizați

Algoritm	**Best case**	**Average**	**Worst Case**
Quick Sort (Random Pivot)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Quick Sort (Median Pivot)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort (Half Pivot)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Ternary Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Radix Sort (base 10)	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
Radix Sort (base 16)	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
Radix Sort (2^{16})	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Intro Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Tim Sort	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Shell Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{5/4})$	$\mathcal{O}(n^2)$

4 Implementare

Toți algoritmii au fost implementați în limbajul C++. Codul sursă este disponibil în secțiunea anexă.

5 Metodologie experimentală

Pentru testarea performanțelor am generat vectori de diferite dimensiuni și structuri:

- vectori complet aleatori
- vectori sortați crescător
- vectori sortați descrescător

Fiecare algoritm a fost rulat de 10 ori pentru fiecare configurație, iar timpul mediu de execuție a fost înregistrat folosind funcționalități de măsurare a timpului din <chrono>.

6 Rezultate experimentale

Figura 1: Performanța în funcție de dimensiunea vectorului (medie pe 10 rulari)

6.1 Observații

Se poate observa că algoritmii cu complexitate $\mathcal{O}(n^2)$ devin rapid ineficienți pe vectori mari. Quick Sort este foarte performant în medie, dar instabil în cazuri nefavorabile. Merge Sort oferă performanță constantă, iar Heap Sort are o implementare mai complicată, dar robustă.

7 Concluzii

- Algoritmii simpli sunt potriviți doar pentru seturi mici de date.
- Quick Sort este ideal pentru cazuri generale, dar trebuie tratat cu atenție pentru cazurile nefavorabile.
- Merge Sort este fiabil și stabil, ideal pentru date mari și sortare externă.

Anexă: Cod sursă (fragment)

```
void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = false;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                std::swap(arr[j], arr[j+1]);
```

```
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```