Lớp: TTNT2025

BÁO CÁO KẾT QUẢ THỬ NGHIỆM

Sinh viên thực hiện: Nguyễn Viết Thành

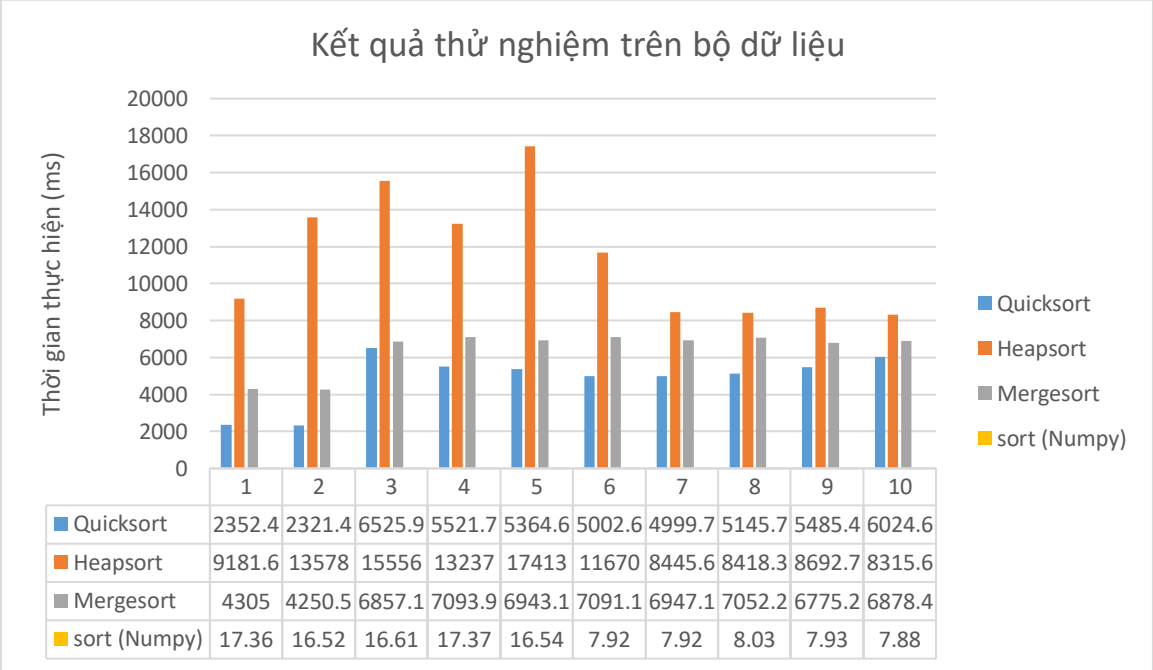
Nội dung báo cáo:

I. Kết quả thử nghiệm

1. Bảng thời gian thực hiện

Dữ liệu	Thời gian thực hiện (ms)			
	Quicksort	**Heapsort**	**Mergesort**	**sort (Numpy)**
1	2352.41	9181.57	4305.03	17.36
2	2321.36	13578.19	4250.53	16.52
3	6525.86	15556.29	6857.1	16.61
4	5521.7	13236.56	7093.85	17.37
5	5364.61	17413.1	6943.13	16.54
6	5002.6	11669.9	7091.05	7.92
7	4999.65	8445.58	6947.09	7.92
8	5145.71	8418.3	7052.16	8.03
9	5485.43	8692.65	6775.24	7.93
10	6024.55	8315.62	6878.4	7.88
Trung bình	4874.39	11450.78	6419.36	12.41

2. Biểu đồ (cột) thời gian thực hiện



	1	2	3	4	5	6	7	8	9	10
Quicksort	2352.4	2321.4	6525.9	5521.7	5364.6	5002.6	4999.7	5145.7	5485.4	6024.6
Heapsort	9181.6	13578	15556	13237	17413	11670	8445.6	8418.3	8692.7	8315.6
Mergesort	4305	4250.5	6857.1	7093.9	6943.1	7091.1	6947.1	7052.2	6775.2	6878.4
sort (Numpy)	17.36	16.52	16.61	17.37	16.54	7.92	7.92	8.03	7.93	7.88

II. Kết luận:

1. Nhận xét tổng quan

Dựa trên kết quả đo lường thời gian thực thi của bốn phương pháp sắp xếp trên 10 bộ dữ liệu (mỗi bộ gồm 1 triệu phần tử), thứ tự hiệu năng từ nhanh nhất đến chậm nhất như sau: sort (Numpy), QuickSort, MergeSort, HeapSort. Điều đó cho thấy NumPy Sort có tốc độ vượt trội với thời gian trung bình thấp nhất. Có thể

thấy sự chênh lệch giữa NumPy Sort và các thuật toán tự cài đặt bằng Python là rất lớn, lên tới hàng trăm lần. Điều này cho thấy ảnh hưởng đáng kể của môi trường thực thi và mức độ tối ưu hóa cài đặt.

2. Giải thích sự chênh lệch thời gian thực thi

2.1. Nguyên nhân NumPy Sort vượt trội

Hàm np.sort() được xây dựng bằng ngôn ngữ C và được tối ưu hóa ở mức thấp, giúp giảm thiểu chi phí do trình thông dịch Python gây ra, tận dụng hiệu quả bộ nhớ liên tục, thực hiện thao tác so sánh và hoán đổi ở tốc độ cao. Trong khi đó, các thuật toán QuickSort, MergeSort và HeapSort trong bài được cài đặt bằng Python thuần, nên phải chịu overhead từ vòng lặp và thao tác cấp cao của Python.

2.2. So sánh QuickSort, MergeSort và HeapSort

QuickSort có thời gian trung bình thấp hơn MergeSort và HeapSort. Nguyên nhân chủ yếu là: cơ chế chia để trị giúp giảm chi phí tổ chức lại dữ liệu, số thao tác quản lý cấu trúc dữ liệu ít phức tạp hơn so với MergeSort, khả năng tận dụng bộ nhớ đệm tốt hơn HeapSort.

MergeSort có thời gian thực thi tương đối ổn định giữa các bộ dữ liệu. Điều này phù hợp với đặc điểm lý thuyết của thuật toán khi độ phức tạp luôn là O(nlogn) trong mọi trường hợp (tốt nhất, trung bình và xấu nhất). Tuy nhiên, do phải tạo mảng phụ trong quá trình trộn, thuật toán phát sinh thêm chi phí bộ nhớ và thời gian.

HeapSort là thuật toán chậm nhất trong thử nghiệm. Mặc dù cũng có độ phức tạp O(nlogn), nhưng việc tổ chức dữ liệu dưới dạng cây nhị phân khiến các thao tác truy cập phần tử diễn ra ở các vị trí xa nhau trong mảng. Điều này làm giảm hiệu quả sử dụng bộ nhớ đệm, dẫn đến thời gian thực thi cao hơn.

2.3. Ảnh hưởng của cấu trúc dữ liệu đầu vào

Hai bộ dữ liệu đã được sắp xếp tăng dần và giảm dần có thời gian thực thi thấp hơn ở một số thuật toán so với dữ liệu ngẫu nhiên. Điều này cho thấy cấu trúc ban đầu của dữ liệu có thể ảnh hưởng đến số phép so sánh và phân hoạch trong quá trình sắp xếp. Tuy nhiên, nhìn chung các thuật toán có độ phức tạp O(nlogn) vẫn duy trì thời gian xử lý tương đối ổn định giữa các loại dữ liệu.

3. Đánh giá ứng dụng thực tế

Từ kết quả thực nghiệm, có thể rút ra một số kết luận ứng dụng như sau:

1. Trong thực tế lập trình Python, nên ưu tiên sử dụng np.sort() hoặc các hàm sắp xếp có sẵn do được tối ưu hóa cao.
2. MergeSort phù hợp khi cần tính ổn định và đảm bảo hiệu năng không bị suy giảm trong trường hợp xấu.
3. QuickSort cho tốc độ xử lý tốt trong đa số trường hợp nhưng cần chú ý cách chọn pivot để tránh trường hợp xấu nhất.

III. Thông tin chi tiết – link github, trong repo gibub cần có

1. Báo cáo
2. Mã nguồn
3. Dữ liệu thử nghiệm

Đường link Github: https://github.com/25521715-NguyenVietThanh/Sorting_Algorithm_Experiment