

BÁO CÁO KẾT QUẢ THỬ NGHIỆM

Sinh viên thực hiện: Võ Nguyễn Minh Long – 25521057

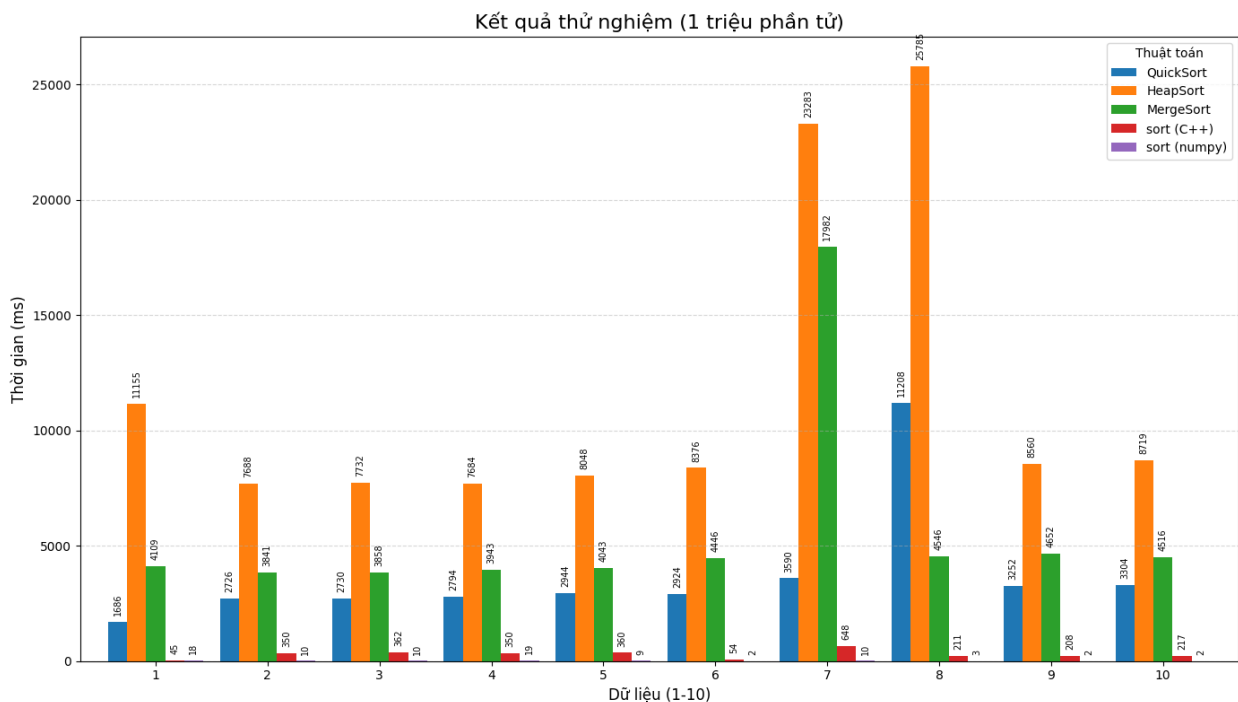
Nội dung báo cáo:

I. Kết quả thử nghiệm

1. Bảng thời gian thực hiện

Dữ liệu	Thời gian thực hiện (ms)				
	Quicksort	Heapsort	Mergesort	Sort (C++)	Sort (NumPy)
1	1686.380	11155.010	4109.010	45.200	18.040
2	2726.250	7688.380	3840.710	350.330	9.510
3	2730.500	7731.600	3858.270	361.510	10.010
4	2794.470	7684.480	3942.940	350.040	18.730
5	2943.710	8047.790	4042.670	360.020	9.050
6	2923.660	8375.630	4446.240	53.690	2.170
7	3589.940	23283.420	17982.360	648.420	9.790
8	11208.180	25785.210	4545.660	210.610	2.590
9	3252.090	8559.880	4652.420	208.250	2.410
10	3304.390	8718.530	4515.860	216.700	2.350
Trung bình	3715.957	11702.993	5593.614	280.477	8.465

2. Biểu đồ (cột) thời gian thực hiện



II. Kết luận

1. So sánh giữa thư viện có sẵn và thuật toán tự cài đặt

- **Hiệu năng vượt trội của thư viện:** Hai hàm `sort(NumPy)` và `sort(C++)` (Python built-in) cho tốc độ xử lý **nhánh hơn gấp hàng trăm đến hàng nghìn lần** so với các thuật toán tự cài đặt bằng Python thuần.
 - Cụ thể, `sort(NumPy)` nhanh nhất với thời gian trung bình chỉ khoảng **8.4 ms**.
 - `sort(C++)` đứng thứ hai với trung bình khoảng **280 ms**.
- **Lý do:** Các hàm thư viện này thực chất chạy trên nền ngôn ngữ C/C++ đã được biên dịch và tối ưu hóa cực tốt về quản lý bộ nhớ (đặc biệt là NumPy với mảng liên kề), trong khi các thuật toán tự viết (Quicksort, Heapsort, Mergesort) chịu hạn chế về tốc độ của trình thông dịch Python.

2. So sánh giữa các thuật toán tự cài đặt (Quick, Heap, Merge)

- **QuickSort (Tốt nhất trong nhóm tự viết):** Trong 3 thuật toán tự cài đặt, QuickSort cho hiệu năng tốt nhất với thời gian trung bình khoảng **3.7 giây** (3715 ms). Điều này phản ánh đúng lý thuyết: QuickSort thường là thuật toán nhanh nhất trong thực tế nhờ tính cục bộ của bộ nhớ (locality of reference) tốt.
- **MergeSort (Ổn định):** Đứng thứ hai với thời gian trung bình khoảng **5.6 giây** (5593 ms). Tuy nhiên, tại bộ dữ liệu số 7, thuật toán này gặp tình trạng tăng vọt thời gian xử lý (gần 18 giây).
- **HeapSort (Chậm nhất):** HeapSort cho kết quả chậm nhất trong các thử nghiệm với thời gian trung bình lên tới **11.7 giây**, chậm gấp 3 lần so với QuickSort. Lý do là HeapSort có khả năng cache kém (do truy cập các phần tử mảng nhảy quãng xa theo chỉ số cha-con) và chi phí hoán đổi phần tử lớn.

3. Phân tích các trường hợp đặc biệt (Dữ liệu 1, 6, 7, 8)

- **Dữ liệu 1 và 6 (Dãy đã sắp xếp/ngược):**
 - Hàm `sort(C++)` (sử dụng Timsort) thể hiện sức mạnh tuyệt đối trên dữ liệu đã có thứ tự (chỉ mất ~45-53 ms so với trung bình 350 ms).
 - QuickSort cũng chạy nhanh nhất trên dữ liệu này (khoảng 1.6s) so với các dữ liệu ngẫu nhiên, chứng tỏ việc chọn Pivot hợp lý đã tránh được trường hợp xấu nhất $O(n^2)$.
 - **Sự bất thường ở Dữ liệu 7 và 8:**
 - Biểu đồ cho thấy sự tăng vọt bất thường ở Dữ liệu 7 (MergeSort, HeapSort) và Dữ liệu 8 (QuickSort, HeapSort). HeapSort đặc biệt mất tới **23-25 giây** ở các trường hợp này. Đây có thể là do đặc điểm phân bố của dữ liệu ngẫu nhiên rơi vào các trường hợp gây phân hoạch không đều (với QuickSort) hoặc tốn nhiều chi phí vun đống (với HeapSort).
-

Kết luận chung

Với tập dữ liệu lớn (1 triệu phần tử), việc sử dụng các thư viện tối ưu như **NumPy** hoặc **Built-in Sort** là lựa chọn bắt buộc trong môi trường Python để đảm bảo hiệu năng. Đối với mục đích học thuật tìm hiểu cấu trúc dữ liệu, **QuickSort** là thuật toán cài đặt thủ công mang lại hiệu quả cao nhất, trong khi **HeapSort** cho thấy hạn chế rõ rệt về tốc độ thực thi trong môi trường ngôn ngữ thông dịch.

III. Thông tin chi tiết – Github

<https://github.com/jlonggg/Sorting-Algorithms-Benchmark>