TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA TP.HCM

MÔN HỌC: THỰC HÀNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

BÁO CÁO

LAB 3: SORTING

GIÁO VIÊN: BÙI HUY THÔNG
SINH VIÊN: PHAN VĂN HOÀNG

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 10 NĂM 2022

Mục lục

1 Giới thiệu thuật toán

1.1 Selection Sort

1.1.1 Ý tưởng

Đầu tiên tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp, sau đó đưa nó về đầu mảng. Tiếp tục tìm phần tử nhỏ nhất từ các phần tử chưa được sắp xếp còn lại rồi đưa về đầu mảng chưa được sắp xếp (sau phần tử vừa được đưa trước đó). Tiếp tục như vậy cho đến khi tất cả các phần tử được sắp xếp.

1.1.2 Các bước thuật toán

selectionSort(a[n], n)

```
Xem phần tử đầu tiên là phần tử nhỏ nhất có vị trí là min_index.
Duyệt qua mảng để tìm vị trí phần tử nhỏ nhất trong mảng chưa được sắp xếp.
Sau đó, đưa phần tử có giá trị nhỏ nhất lên đầu mảng chưa được sắp xếp và tăng min_index lên 1.
Lặp lại cho đến khi mảng được sắp xếp (min_index = n - 1) .
```

1.1.3 Ứng dụng

- Sắp xếp mà không tốn thêm không gian lưu trữ.
- Sắp xếp mảng có ít phần tử

1.1.4 Độ phức tạp

Time Complexity	Best Case	$O(n^2)$
	Average Case	$O(n^2)$
	Worst Case	$O(n^2)$
Space Complexity		$O(1)$

Bảng 1: Độ phức tạp Selection Sort

1.2 Insertion Sort

1.2.1 Ý tưởng

Ta xem phần tử đầu tiên của mảng là một mảng đã được sắp xếp. Duyệt các phần tử còn lại trong mảng chưa được sắp xếp. Đưa từng phần tử vào mảng đã được sắp xếp, vị trí đúng của phần tử đang xét là vị trí có giá trị thỏa lớn hơn hoặc bằng phần tử trước và nhỏ hơn hoặc bằng phần tử sau. Lặp lại n - 1 lần để hoàn thành quá trình sắp xếp.

1.2.2 Các bước thuật toán

insertionSort(a[n], n)

```
1 Gán i bằng vị trí phần tử thứ 2 trong mảng, temp = a[i].
2 Duyệt từ i về đầu mảng, trong khi phần tử đang duyệt còn lớn hơn a[i] hoặc chưa duyệt
  đến đầu mảng  thì di chuyển phần tử đang duyệt lên một vị trí, còn nếu không thì gán giá
  trị phần tử tại vị trí đang duyệt bằng temp.
3 Tăng i lên 1. Lặp lại bước 1 cho đến hết mảng.
```

1.2.3 Ứng dụng

- Sắp xếp danh sách với số phần tử nhỏ.
- Sắp xếp dách sách gần như được sắp xếp.
- Dùng trong Flash Sort và nhiều hàm khác.
- Không tốn nhiều bộ nhớ.

1.2.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n)$ $O(n^2)$ $O(n^2)$
Space Complexity		$O(1)$

Bảng 2: Độ phức tạp Insertion Sort

1.3 Bubble Sort

1.3.1 Ý tưởng

Lặp lại việc đưa phần tử nhỏ nhất của phần mảng chưa được sắp xếp lên đầu mảng bằng cách đi từ cuối mảng về đầu mảng và hoán đổi các phần tử liền kề nếu chúng không đúng thứ tự. Sau mỗi lần duyệt thì sẽ có một phần tử đưa về đúng vị trí.

1.3.2 Các bước thuật toán

bubbleSort(a[n], n)

```
1 Bắt đầu từ phần tử cuối cùng của mảng, nếu nó nhỏ hơn phần tử trước thì hoán đổi với
  nhau, nếu không thì chuyển lên xét phần tử trước. Quá trình tiếp tục cho đến phần tử đầu
  tiên của mảng chưa được sắp xếp, phần tử nhỏ nhất được đưa lên đầu mảng.
2 Lặp lại bước 1, cho đến khi phần tử cuối cùng của mảng bằng với phần tử cuối cùng của
  mảng chưa được sắp xếp.
```

1.3.3 Ứng dụng

- Sắp xếp chiếm ít không gian lưu trữ.
- Sắp xếp các phần tử chưa được sắp xếp theo một thứ tự cụ thể.
- Sắp xếp học sinh dựa trên chiều cao thành một hàng.

1.3.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n)$ $O(n^2)$ $O(n^2)$
Space Complexity		$O(1)$

Bảng 3: Độ phức tạp Bubble Sort

1.4 Shaker Sort

1.4.1 Ý tưởng

Dựa trên ý tưởng của Bubble Sort. Mỗi lần lặp thuật toán chia thành 2 giai đoạn:

- Giai đoạn 1: đưa phần tử nhỏ nhất về đầu mảng bằng 1 lần lặp của Bubble Sort bắt đầu từ vị trí cuối cùng được hoán đổi. Đánh dấu vị trí cuối cùng được hoán đổi.
- Giai đoạn 2: đưa phần tử lớn nhất về cuối mảng bằng 1 lần lặp của Bubble Sort bắt đầu từ vị trí cuối cùng được hoán đổi. Đánh dấu vị trí cuối cùng được hoán đổi.

1.4.2 Các bước thuật toán

shakerSort(a[n], n)

```
1 Duyệt mảng từ trái sang phải, giống như Bubble Sort. Trong vòng lặp, các phần tử liền kề
  được so sánh và nếu giá trị ở bên trái lớn hơn giá trị ở bên phải, thì các giá trị sẽ
  được hoán đổi. Vào cuối lần lặp đầu tiên, số lớn nhất sẽ nằm ở cuối mảng.
2 Lần duyệt thứ hai, duyệt mảng theo hướng ngược lại - bắt đầu từ phần tử ngay trước phần
  tử được sắp xếp gần đây nhất và quay trở lại đầu mảng. Ở đây, các phần tử liền được so
  sánh và được hoán đổi nếu giá trị bên phải nhỏ hơn giá trị bên trái. Vòng lặp này sẽ đưa
  giá trị nhỏ nhất về đầu mảng.
3 Lặp lại bước một, duyệt từ trái sang phải từ vị trí vừa được sắp xếp cho đến khi mảng
  được sắp xếp.
```

1.4.3 Ứng dụng

- Cần tính ổn định.

1.4.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n)$ $O(n^2)$ $O(n^2)$
Space Complexity		$O(1)$

Bảng 4: Độ phức tạp Shaker Sort

1.5 Shell Sort

1.5.1 Ý tưởng

Shell Sort là một biến thể, cải tiến của Insertion Sort. Trong Insertion Sort ta chỉ chuyển một phần tử về trước bằng cách dồn các phần tử về phía sau, tốn rất nhiều chi phí nếu phần tử đang xét là phần tử nhỏ nhất thì cần phải chuyển về ngoài cùng bên trái của mảng. Shell Sort cải tiến điều đó bằng cách chia thành nhiều phần để sắp xếp. Đầu tiên, nó sắp xếp các phần tử cách xa nhau và sau đó liên tiếp giảm khoảng cách giữa các phần tử được sắp xếp rồi sắp xếp. Lặp lại việc giảm khoảng cách khi khoảng cách còn lớn hơn 0.

1.5.2 Các bước thuật toán

shellSort(a[n], n)

```
Khởi tạo gap bằng kích thước mảng.
Gán gap  = gap / 2
Chia danh sách thành các danh sách con nhỏ hơn với khoảng cách bằng gap.
Sắp xếp các danh sách con này bằng cách sử dụng Insertion Sort.
Lặp lại bước 2 cho đến khi gap  = 0.
```

1.5.3 Ứng dụng

- Thư viện chuẩn C sử dụng ShellSort thay cho qsort khi xử lý các hệ thống nhúng.
- Các máy nén, chẳng hạn như bzip2, cũng sử dụng nó để tránh các vấn đề có thể xảy ra khi các thuật toán sắp xếp vượt quá độ sâu đệ quy của ngôn ngữ.

1.5.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n*log(n))$ $O(n*log(n))$ $O(n^2)$
Space Complexity		$O(1)$

Bảng 5: Độ phức tạp Shell Sort

1.6 Heap Sort

1.6.1 Ý tưởng

Heap Sort được xây dựng dựa trên cấu trúc dữ liệu Heap. Heap Sort có thể coi là một cải tiến của Selection Sort. Chúng ta tìm phần tử lớn nhất và di chuyển nó đến cuối mảng chưa được sắp xếp. Sự khác biệt chính là thay vì quét qua toàn bộ mảng để tìm phần tử lớn nhất, chúng ta chuyển mảng thành một Max-heap để tìm phần tử lớn nhất.

1.6.2 Các bước thuật toán

heapSort(a[n], n)

```
Tạo Max-heap từ đầu vào.
Hoán đổi phần tử đầu tiên với phần tử cuối cùng của Heap.
Giảm kích thước đi 1. Tạo lại Max-heap.
Lặp lại bước 2 cho đến khi kích thước Heap <= 1.
```

1.6.3 Ứng dụng

- Có thể dùng Heap Sort trong thuật toán Dijkstra.
- Sử dụng khi cần giá trị nhỏ nhất (ngắn nhất) hoặc lớn nhất (dài nhất) ngay lập tức.
- Tìm thứ tự trong thống kê.
- Xử lý các hàng đợi ưu tiên trong thuật toán Prim.
- Mã hóa Huffman.
- Nén dữ liệu.
- Trong hệ điều hành Linux, Heap Sort được sử dụng rộng rãi để lập lịch công việc cho các quy trình do độ phức tạp về thời gian O (nlogn) và độ phức tạp về không gian O (1).

1.6.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n*log(n))$ $O(n*log(n))$ $O(n*log(n))$
Space Complexity		$O(1)$

Bảng 6: Độ phức tạp Heap Sort

1.7 Merge Sort

1.7.1 Ý tưởng

Merge Sort là một thuật toán dựa trên kỹ thuật chia để trị. Dựa trên ý tưởng chia nhỏ danh sách thành nhiều danh sách con cho đến khi mỗi danh sách con bao gồm một phần tử duy nhất và hợp nhất các danh sách con đó theo cách dẫn đến một danh sách được sắp xếp.

1.7.2 Các bước thuật toán

mergeSort(a[n], left, right)

```
Nếu left <= right thì thoát hàm.
Tìm vị trí ở giữa của mảng, middle.
Sắp xếp mảng bên trái (left đến middle) bằng mergeSort.
Sắp xếp mảng bên phải (middle + 1 đến right) bằng mergeSort.
Trộn hai mảng đã được sắp xếp lại, được mảng sắp xếp hoàn chỉnh.
```

1.7.3 Ứng dụng

- Sắp xếp danh sách liên kết.
- Đếm số lượng đảo ngược trong một danh sách. (Cặp số đảo ngược)
- Ứng dụng trong External Sort.

1.7.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n * log(n))$ $O(n * log(n))$ $O(n * log(n))$
Space Complexity		$O(1)$ hoặc $O(n)$

Bảng 7: Độ phức tạp Merge Sort

1.8 Quick Sort

1.8.1 Ý tưởng

Quick Sort sử dụng kỹ thuật chia để trị. Chọn một phần tử làm trục, đặt nó vào đúng vị trí của nó trong mảng đã được sắp xếp và phân vùng mảng đã cho xung quanh trục đã chọn, ở bên trái là các phần tử nhỏ hơn, bên phải là các phần tử lớn hơn. Chúng ta sẽ tiếp tục gọi Quick Sort ở phần bên trái và bên phải của phần tử trục.

1.8.2 Các bước thuật toán

quickSort(a[n], left, right)

```
1 Nếu left < right thì dừng.
2 Chọn phần tử giữa mảng làm trục. Tìm vị trí đúng của nó  (pivotIndex) trong mảng đã được
  sắp xếp. Đưa các phần tử nhỏ hơn về bên trái, lớn hơn về bên phải.
3 Gọi quickSort cho mảng ở bên trái của pivotIndex.
4 Gọi quickSort cho mảng ở bên phải của pivotIndex.
```

1.8.3 Ứng dụng

- Các biến thể của Quick Sort được sử dụng để tách k phần tử nhỏ nhất hoặc lớn nhất.
- Sắp xếp tệp theo tên/ ngày, sắp xếp sinh viên theo mã số của họ, sắp xếp hồ sơ tài khoản theo ID đã cho.
- Sử dụng trong tìm kiếm thông tin.

1.8.4 Độ phức tạp

1.9 Counting Sort

1.9.1 Ý tưởng

Thay vì thao tác so sánh, sắp xếp đếm sử dụng chỉ số mảng để xác định thứ tự tương đối của các phần tử. Đối với mỗi phần tử x, sắp xếp đếm đếm số phần tử nhỏ hơn x và đặt phần tử x trực tiếp vào đúng vị trí của

Time Complexity	Best Case Average Case Worst Case	$O(n * log(n)$ $O(n * log(n)$ $O(n^2)$
Space Complexity		$O(log(n))$

Bảng 8: Độ phức tạp Quick Sort

nó trong mảng đã sắp xếp.

1.9.2 Các bước thuật toán

countingSort(a[n], n)

```
Tìm phần tử lớn nhất (giả sử max) từ mảng đã cho.
Khởi tạo một mảng đếm count có độ dài max + 1 với tất cả các phần tử có giá trị bằng 0.
Tính số lượng phần tử của từng phần tử tại chỉ mục tương ứng của chúng trong mảng count.
Tính tổng tích lũy của các phần tử của mảng count.
Tìm chỉ số của từng phần tử của mảng ban đầu trong mảng count và đặt phần tử vào đúng vị
trí của nó ở mảng sắp xếp (vị trí đúng của phần tử a[i] là count[a[i]] - 1 đối với mảng
bắt đầu từ 0). Giảm giá trị của vị trí đó trong mảng count đi 1.
Lặp lại bước 5 cho đến khi tất cả các phần tử vào đúng vị trí.
```

1.9.3 Ứng dụng

- Nếu phạm vi dữ liệu đầu vào không lớn hơn nhiều so với số lượng đối tượng được sắp xếp, thì sử dụng sắp xếp đếm hiệu quả.
- Nó thường được sử dụng như một chương trình con trong các thuật toán sắp xếp khác, chẳng hạn như Radix Sort, Flash Sort.
- Có thể được sử dụng với dữ liệu đầu vào có âm.

1.9.4 Độ phức tạp

d = maxValue - minValue
n = sizeOfArray

Time Complexity	Best Case Average Case Worst Case	$O(max(d, n))$ $O(max(d, n))$ $O(max(d, n))$
Space Complexity		$O(d + n)$

Bảng 9: Độ phức tạp Counting Sort

1.10 Radix Sort

1.10.1 Ý tưởng

Chia số nguyên thành các số khác nhau theo từng chữ số, sau đó so sánh từng chữ số riêng biệt. Phương pháp cụ thể là: thống nhất tất cả các giá trị được so sánh thành cùng một độ dài chữ số và điền vào các số 0

phía trước các số có chữ số ngắn hơn. Sau đó, bắt đầu từ chữ số thấp nhất, sắp xếp theo thứ tự từng chữ số một (phải sang trái). Theo cách này, từ chữ số thứ tự thấp nhất cho đến khi hoàn thành chữ số có thứ tự cao nhất thì mảng sẽ được sắp xếp.

1.10.2 Các bước thuật toán

radixSort(a[n], n)

```
Tìm phần tử lớn nhất trong mảng. Gọi k là số chữ số của phần tử đó.
Lặp lại vòng lặp k lần.  Sử dụng Counting Sort để sắp xếp. Lần lặp 1 sắp xếp các phần tử
tăng dần theo hàng đơn vị, lần lặp 2 tăng dần theo hàng chục,...
```

1.10.3 Ứng dụng

- Chạy trên các máy song song.
- Cần tính ổn định.
- Sắp xếp chuỗi.

1.10.4 Độ phức tạp

max: giá trị lớn nhất trong mảng.

Time Complexity	Best Case Average Case Worst Case	$O((\log_{10} max + 1) * n)$ $O((\log_{10} max + 1) * n)$ $O((\log_{10} max + 1) * n)$
Space Complexity		$O(n + 10)$

Bảng 10: Độ phức tạp Radix Sort

1.11 Flash Sort

1.11.1 Ý tưởng

Flash Sort cải tiến dựa trên điểm mạnh của Counting Sort và Insertion Sort.

- Xác định số phân lớp để phân bổ dữ liệu của mảng vào. Thường để tốt nhất, số phân lớp được tính $m = 0.43 * sizeOfArray$.
- Tạo các phân lớp và phân bổ dữ liệu vào tương tự Counting Sort. Có m phân lớp [0, 1,..., m - 1]. Phần tử có giá trị x sẽ thuộc về phân lớp $k = \lfloor \frac{(m-1)(x-min)}{max-min} \rfloor$.
- Sau khi phân bổ dữ liệu thì tiến hành sắp xếp mảng bằng Insertion Sort.

1.11.2 Các bước thuật toán

flashSort(a[n], n)

```
1 Tìm các giá trị min, max của mảng.
2 Tạo mảng L có m = 0.43 * n phần tử. Mỗi phần tử thứ i sẽ thuộc về vùng có chỉ số  k = (m
  - 1) * (a[i] - min) / (max - min).
3 Đếm số phần tử của mỗi vùng.
4 Tính tổng tích lũy của mỗi vùng (trừ vùng đầu tiên) trong mảng L, L[k] += L[k - 1]. Lúc
  này giá trị L[k] - 1 là vị trí phần tử cuối cùng của phân vùng k.
5 Đưa các phần tử về đúng phân vùng của nó trong mảng ban đầu. Các phần tử trong phân vùng
  k có chỉ mục từ L[k - 1] đến L[k] - 1.
6 Sau khi đưa các phần tử về đúng phân vùng thì tiến hành Insertion Sort trên mảng.
```

1.11.3 Ứng dụng

- Tối ưu tốc độ giải thuật.
- Trong các chương trình cần thuật toán sắp xếp có độ ổn đỉnh cao.

1.11.4 Độ phức tạp

Time Complexity	Best Case Average Case Worst Case	$O(n)$ $O(n)$ $O(n^2)$
Space Complexity		$O(0.43 * n)$

Bảng 11: Độ phức tạp Flash Sort

2 Kết quả thực nghiệm và nhận xét

2.1 Bảng kết quả thực nghiệm

2.1.1 Bảng dữ liệu ngẫu nhiên

Data Order: Randomize

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.112	100,009,999	1.007	900,029,999	2.798	2,500,049,999	11.211	10,000,099,999	100.909	90,000,299,999	282.493	250,000,499,999
Insertion Sort	0.064	50,063,792	0.579	450,346,542	1.610	1,250,632,218	6.445	5,007,424,934	58.158	45,036,683,879	160.813	124,665,124,456
Bubble Sort	0.258	100,009,999	2.733	900,029,999	7.892	2,500,049,999	32.021	10,000,099,999	289.049	90,000,299,999	818.207	250,000,499,999
Shaker Sort	0.223	66,633,194	2.135	600,321,129	6.083	1,666,209,424	24.330	6,674,836,061	217.730	60,085,109,832	616.634	166,353,804,763
Shell Sort	0.003	658,553	0.006	2,324,890	0.011	4,588,751	0.022	10,065,188	0.075	34,452,442	0.132	63,888,717
Heap Sort	0.002	496,854	0.006	1,680,679	0.010	2,951,934	0.023	6,304,049	0.073	20,798,201	0.132	36,122,491
Merge Sort	0.005	430,016	0.017	1,430,544	0.029	2,498,859	0.059	5,297,289	0.177	17,306,473	0.298	29,906,585
Quick Sort	0.001	407,702	0.005	1,446,819	0.010	2,563,337	0.019	5,364,983	0.061	18,030,454	0.110	32,526,421
Counting Sort	0.001	80,003	0.000	240,003	0.001	365,539	0.003	665,541	0.006	1,865,541	0.011	3,065,541
Radix Sort	0.002	140,056	0.008	510,070	0.014	850,070	0.027	1,700,070	0.081	5,100,070	0.136	8,500,070
Flash Sort	0	96,098	0.002	290,273	0.002	483,664	0.004	890,961	0.014	2,666,990	0.031	4,817,425

2.1.2 Bảng dữ liệu gần như được sắp xếp

Data Order: Nearly Sorted

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.112	100,009,999	1.008	900,029,999	2.824	2,500,049,999	11.198	10,000,099,999	100.947	90,000,299,999	281.907	250,000,499,999
Insertion Sort	0	193,786	0.001	480,378	0.001	760,030	0.001	947,270	0.001	1,543,126	0.002	2,604,858
Bubble Sort	0.122	100,009,999	1.104	900,029,999	3.073	2,500,049,999	12.324	10,000,099,999	111.492	90,000,299,999	309.662	250,000,499,999
Shaker Sort	0	197,189	0.002	441,314	0.002	697,853	0.002	793,833	0.001	990,616	0.004	1,690,611
Shell Sort	0.001	415,940	0.002	1,301,793	0.003	2,310,299	0.006	4,729,226	0.016	15,427,015	0.027	25,694,036
Heap Sort	0.002	518,147	0.005	1,740,625	0.008	3,056,037	0.017	6,513,003	0.052	21,440,245	0.091	37,118,966
Merge Sort	0.005	354,621	0.015	1,141,444	0.034	1,941,093	0.052	3,968,553	0.157	12,678,071	0.265	21,657,033
Quick Sort	0.001	261,903	0.002	889,046	0.004	1,530,609	0.007	3,261,147	0.024	10,837,953	0.04	18,902,787
Counting Sort	0	80,005	0.001	240,005	0.001	400,005	0.002	800,005	0.008	2,400,005	0.013	4,000,005
Radix Sort	0.002	140,056	0.008	510,070	0.014	850,070	0.028	1,700,070	0.12	6,000,084	0.205	10,000,084
Flash Sort	0	122,870	0.001	368,667	0.002	614,467	0.004	1,228,968	0.012	3,686,966	0.019	6,144,965

2.1.3 Bảng dữ liệu đã được sắp xếp

Data Order: Sorted

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.113	100,009,999	1.005	900,029,999	2.813	2,500,049,999	11.194	10,000,099,999	101.158	90,000,299,999	280.77	250,000,499,999
Insertion Sort	0	39,998	0	119,998	0	199,998	0.001	399,998	0.001	1,199,998	0.001	1,999,998
Bubble Sort	0.123	100,009,999	1.101	900,029,999	3.129	2,500,049,999	12.363	10,000,099,999	111.367	90,000,299,999	309.54	250,000,499,999
Shaker Sort	0	20,002	0	60,002	0	100,002	0.001	200,002	0.001	600,002	0.001	1,000,002
Shell Sort	0	360,042	0.001	1,170,050	0.002	2,100,049	0.006	4,500,051	0.017	15,300,061	0.026	25,500,058
Heap Sort	0.001	518,233	0.005	1,740,799	0.007	3,056,038	0.016	6,512,874	0.054	21,440,254	0.092	37,118,865
Merge Sort	0.005	321,628	0.015	1,052,684	0.026	1,838,364	0.052	3,876,732	0.164	12,570,236	0.268	21,542,140
Quick Sort	0.001	261,831	0.003	888,958	0.004	1,530,545	0.007	3,261,059	0.023	10,837,889	0.04	18,902,886
Counting Sort	0.001	80,005	0.001	240,005	0.001	400,005	0.002	800,005	0.008	2,400,005	0.013	4,000,005
Radix Sort	0.003	140,056	0.008	510,070	0.014	850,070	0.028	1,700,070	0.119	6,000,084	0.203	10,000,084
Flash Sort	0.001	122,891	0.001	368,691	0.001	614,491	0.004	1,228,991	0.011	3,686,991	0.02	6,144,991

2.1.4 Bảng dữ liệu được sắp xếp đảo ngược

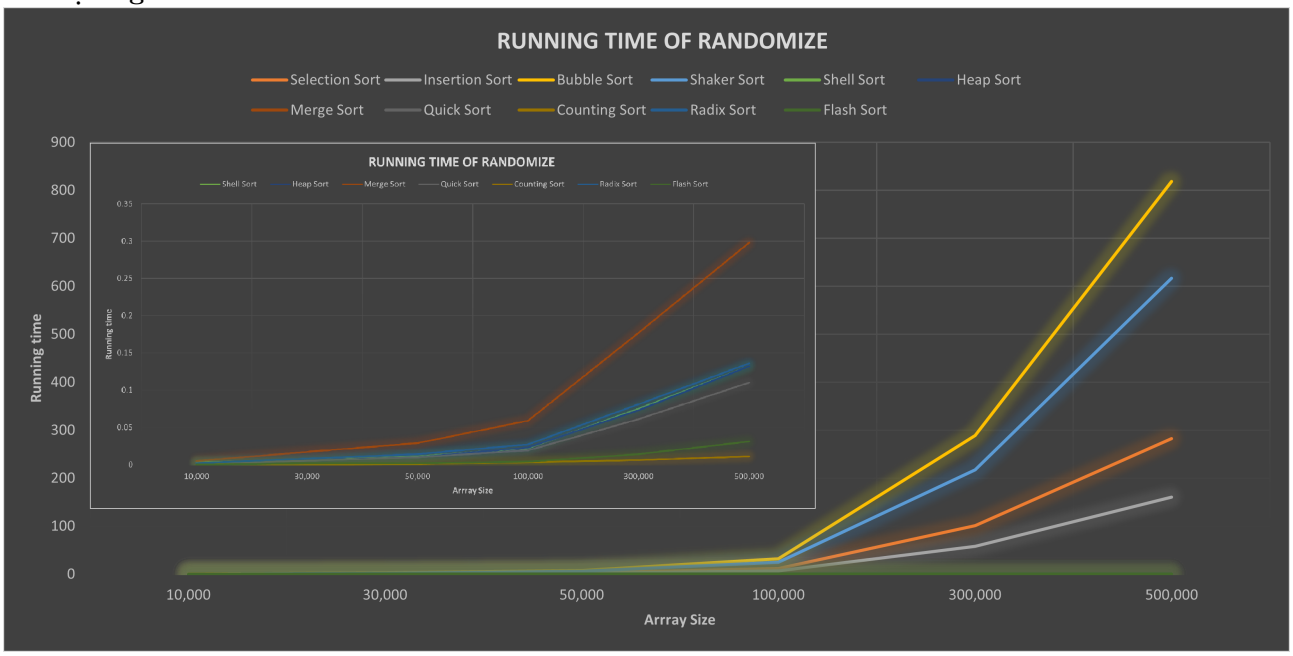
Data Order: Reverse

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	0.119	100,009,999	1.067	900,029,999	3.018	2,500,049,999	11.926	10,000,099,999	107.089	90,000,299,999	296.452	250,000,499,999
Insertion Sort	0.129	100,019,999	1.155	900,059,999	3.26	2,500,099,999	12.851	10,000,199,999	116.186	90,000,599,999	321.722	250,000,999,999
Bubble Sort	0.29	100,009,999	2.606	900,029,999	7.523	2,500,049,999	29.174	10,000,099,999	262.122	90,000,299,999	756.855	250,000,499,999
Shaker Sort	0.309	100,005,001	2.788	900,015,001	7.946	2,500,025,001	31.293	10,000,050,001	279.62	90,000,150,001	775.825	250,000,250,001
Shell Sort	0	475,175	0.002	1,554,051	0.004	2,844,628	0.007	6,089,190	0.022	20,001,852	0.038	33,857,581
Heap Sort	0.001	476,736	0.005	1,622,788	0.008	2,848,013	0.016	6,087,449	0.051	20,187,383	0.088	35,135,727
Merge Sort	0.004	322,827	0.015	1,066,235	0.025	1,849,483	0.05	3,898,971	0.156	12,632,603	0.263	21,860,699
Quick Sort	0	281,817	0.002	948,939	0.003	1,630,526	0.007	3,461,040	0.024	11,437,875	0.04	19,902,867
Counting Sort	0.001	80,005	0.001	240,005	0.001	400,005	0.003	800,005	0.008	2,400,005	0.013	4,000,005
Radix Sort	0.002	140,056	0.008	510,070	0.014	850,070	0.027	1,700,070	0.119	6,000,084	0.203	10,000,084
Flash Sort	0	108,550	0.001	325,650	0.002	542,750	0.004	1,085,500	0.01	3,256,500	0.017	5,427,500

2.2 Biểu đồ kết quả thực nghiệm

2.2.1 Biểu đồ đường thời gian chạy các hàm

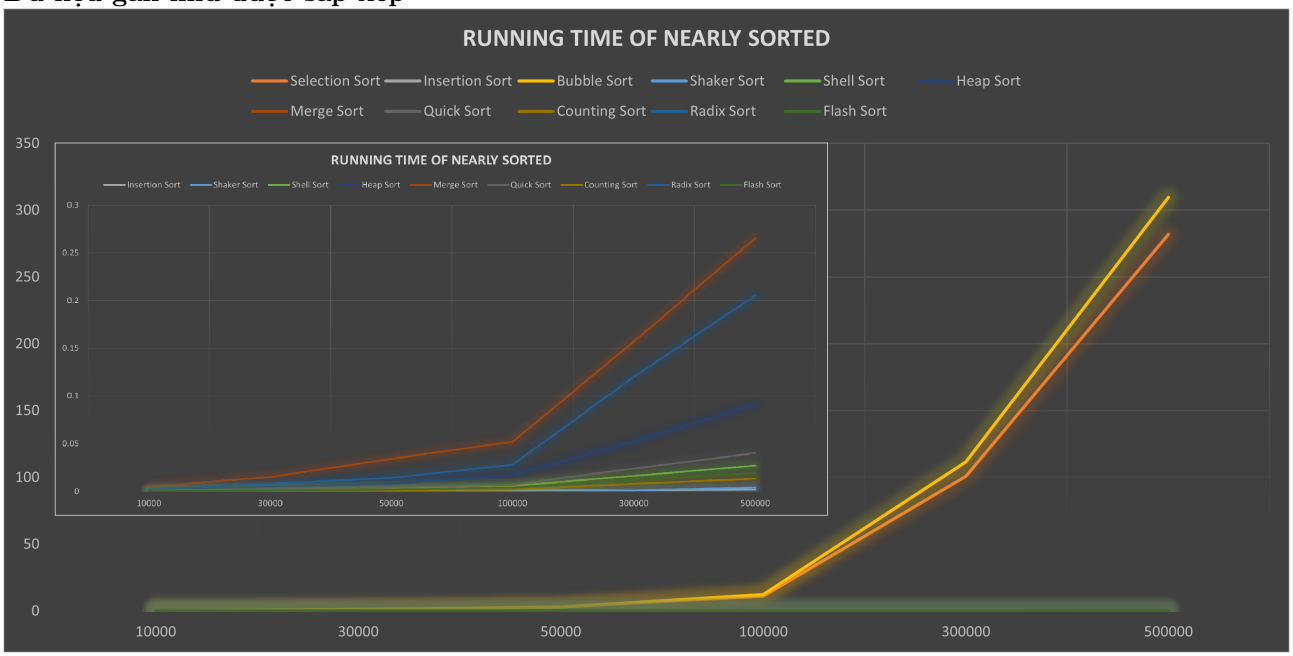
1. **Dữ liệu ngẫu nhiên**



Dựa vào biểu đồ và bảng thực nghiệm ta thấy ở trường hợp này:

- Thuật toán chạy nhanh nhất là: Counting Sort.
- Thuật toán chạy chậm nhất là: Bubble Sort.

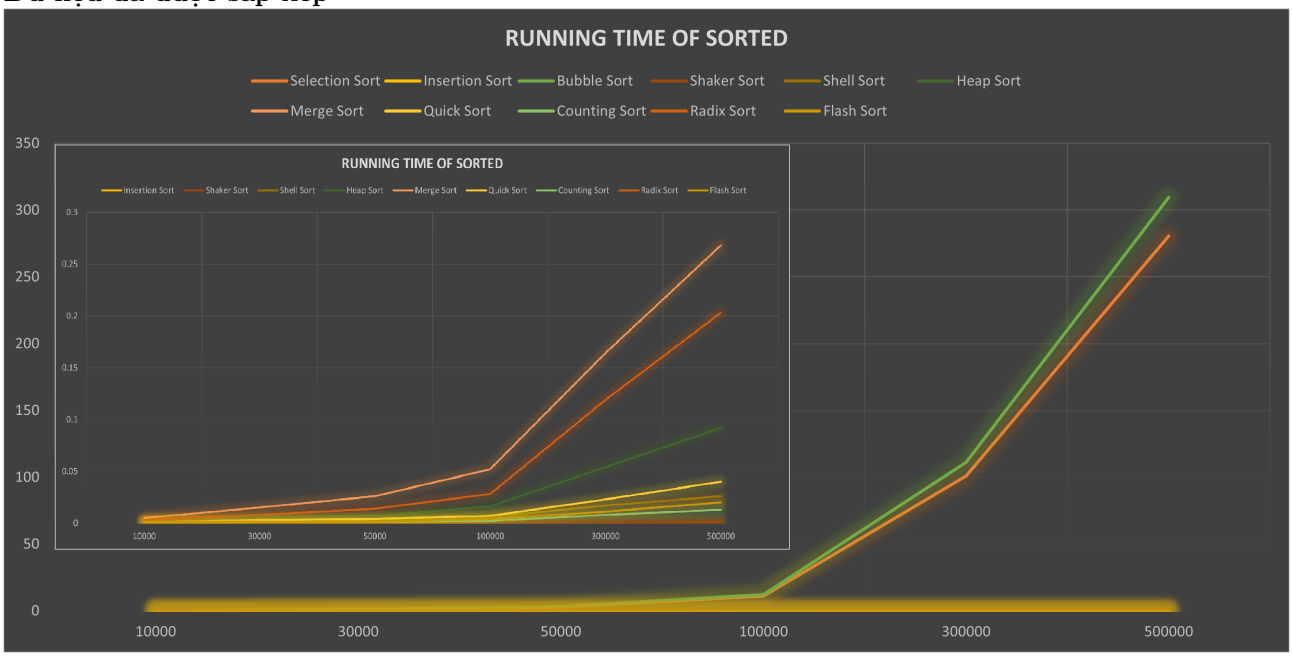
2. **Dữ liệu gần như được sắp xếp**



Dựa vào biểu đồ và bảng thực nghiệm ta thấy ở trường hợp này:

- Thuật toán chạy nhanh nhất là: Insertion Sort.
- Thuật toán chạy chậm nhất là: Bubble Sort, Selection Sort.

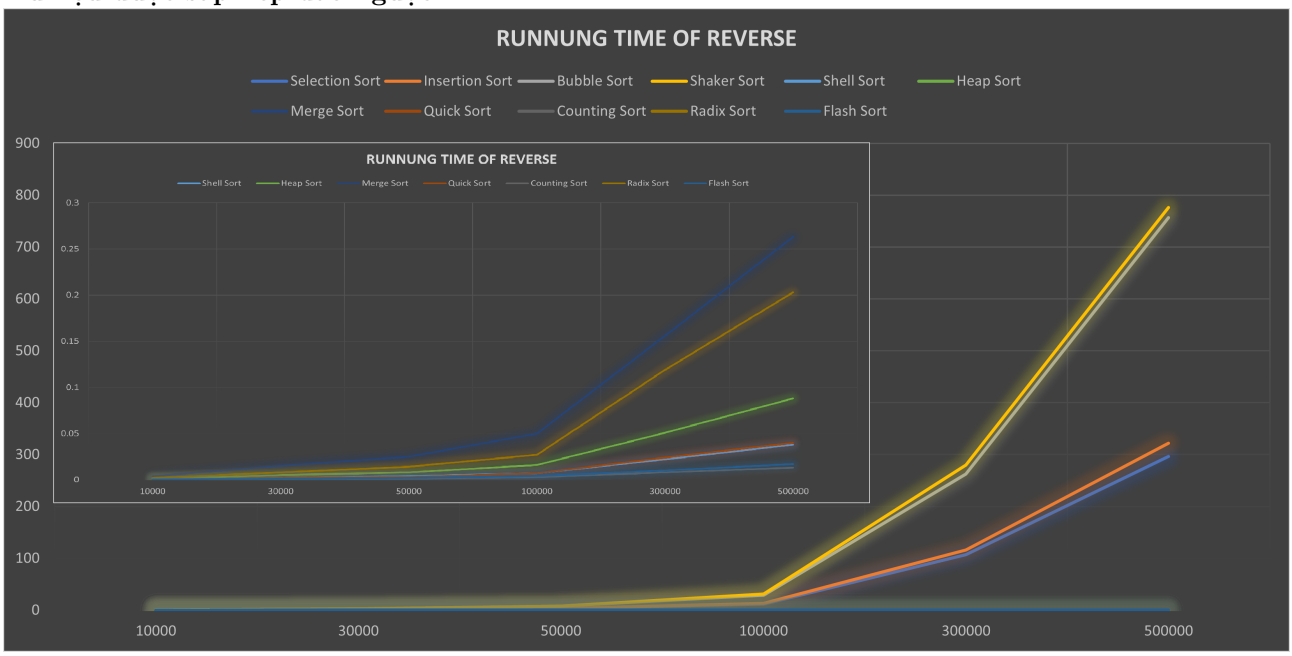
3. **Dữ liệu đã được sắp xếp**



Dựa vào biểu đồ và bảng thực nghiệm ta thấy ở trường hợp này:

- Thuật toán chạy nhanh nhất là: Insertion Sort, Shaker Sort.
- Thuật toán chạy chậm nhất là: Bubble Sort, Selection Sort.

4. **Dữ liệu được sắp xếp đảo ngược**



Dựa vào biểu đồ và bảng thực nghiệm ta thấy ở trường hợp này:

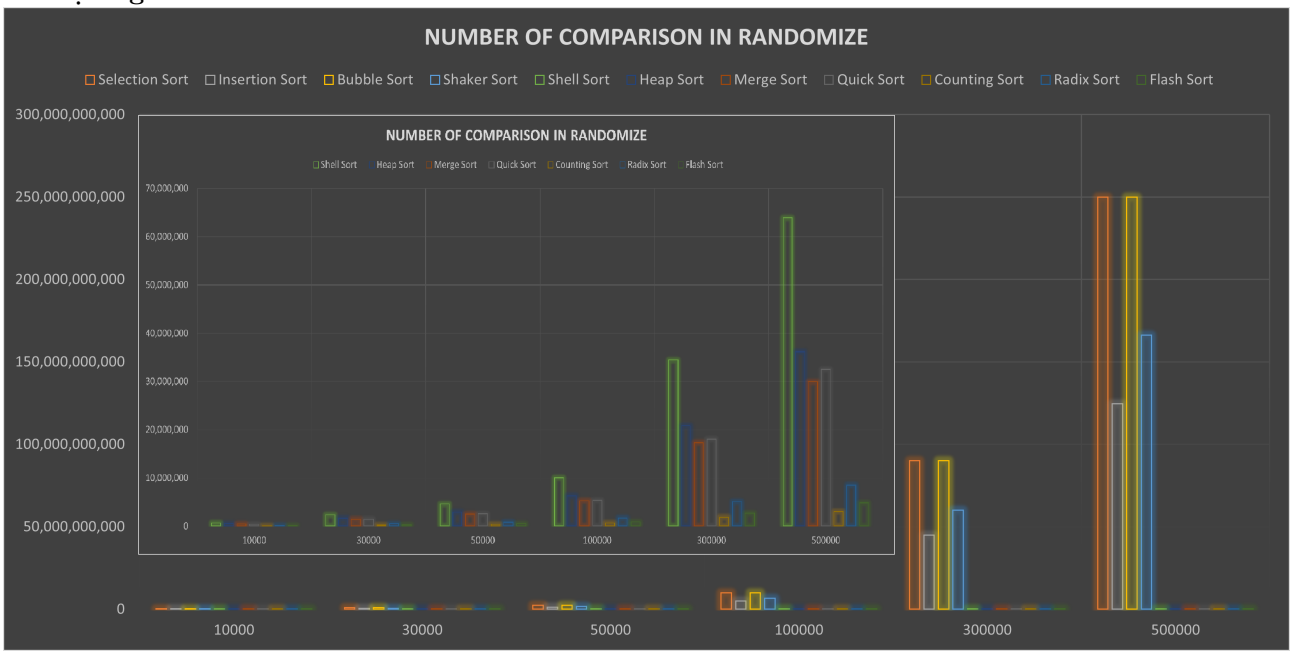
- Thuật toán chạy nhanh nhất là: Counting Sort, Flash Sort.
- Thuật toán chạy chậm nhất là: Shaker Sort, Bubble Sort.

5. **Kết luận**

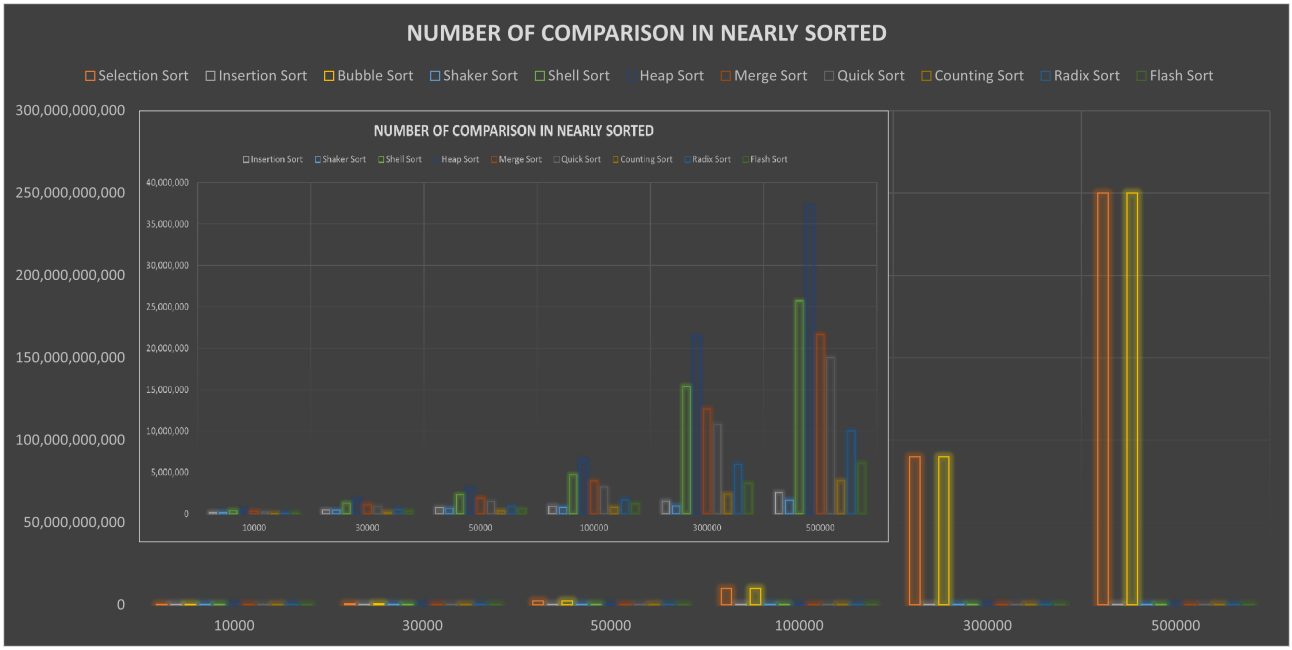
- Trong trường hợp mảng không được sắp xếp Counting Sort, Flash Sort, Quick Sort chạy rất nhanh và ổn định. Ở trường hợp này, Counting Sort chạy nhanh nhất vì đây là trường hợp tốt nhất của Counting Sort, vì phạm vi của dữ liệu là [0; n] (với n là số phần tử của mảng). Khi đó Counting Sort có độ phức tạp $O(n)$. Tuy nhiên điểm yếu của nó là sử dụng nhiều bộ nhớ hơn so với Flash Sort và Quick Sort.
- Ở trường hợp đặc biệt, mảng gần như đã được sắp xếp hoặc đã được sắp xếp thì Insertion Sort và Shaker Sort (đã được cải tiến) chạy tốt nhất với độ phức tạp $O(n)$.
- Trong mọi trường hợp, Selection Sort và Bubble Sort luôn chạy chậm nhất.

2.2.2 Biểu đồ cột số phép so sánh của các hàm

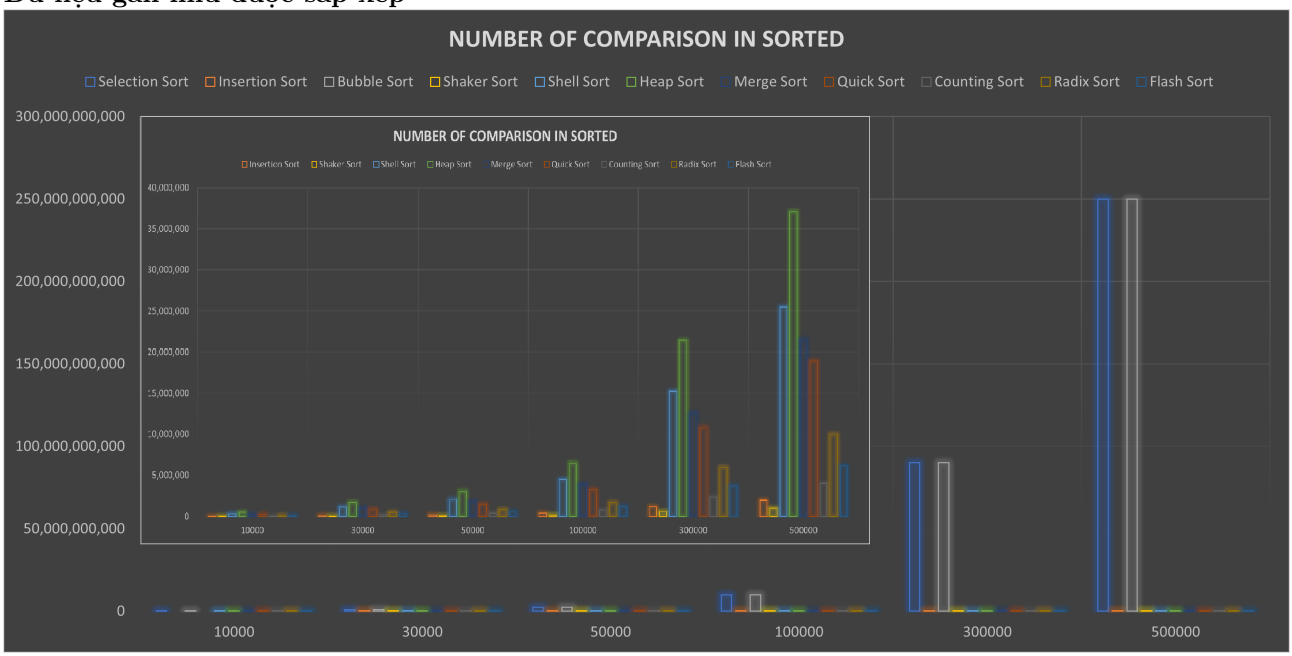
1. **Dữ liệu ngẫu nhiên**



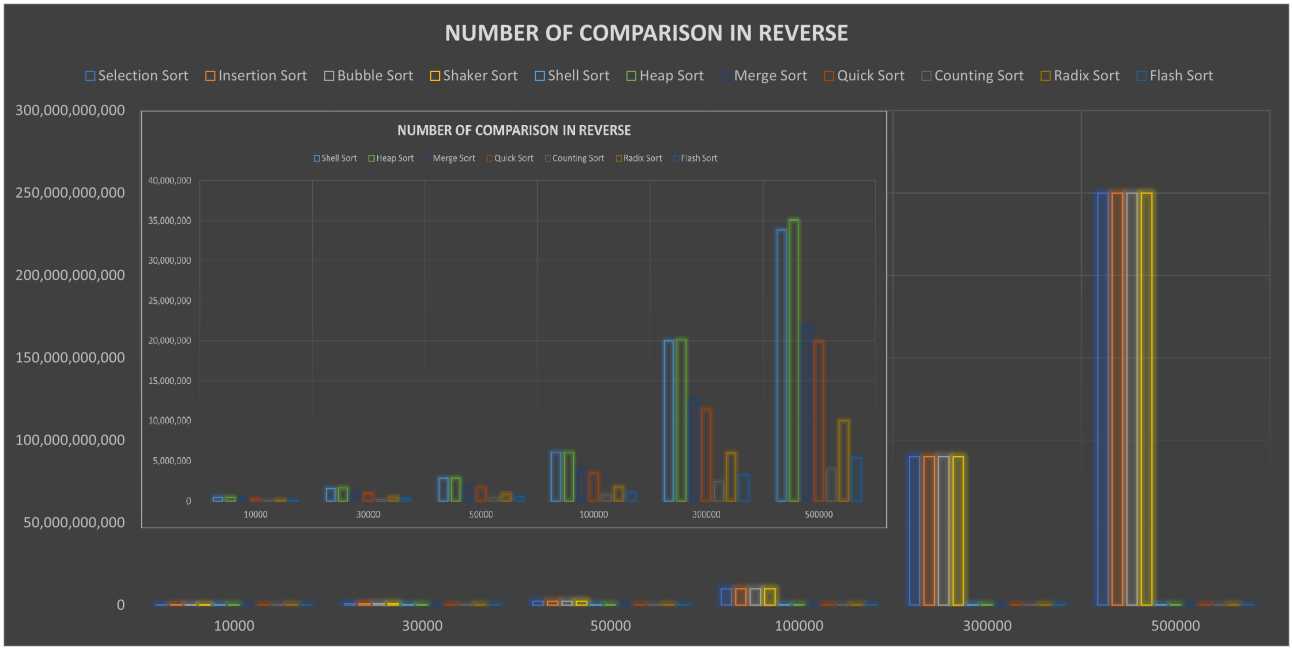
2. **Dữ liệu gần như được sắp xếp**



3. **Dữ liệu gần như được sắp xếp**



4. **Dữ liệu được sắp xếp đảo ngược**



5. **Kết luận:** Tương tự thời gian chạy của các hàm:
   - Ở các trường hợp bình thường, Counting Sort sử dụng ít phép so sánh nhất.
   - Ở trường hợp đặc biệt, mảng gần như được sắp xếp hoặc đã được sắp xếp thì Shaker Sort và Insertion Sort sử dụng ít phép so sánh nhất.
   - Số phép so sánh của Selection Sort và Bubble Sort không phụ thuộc vào kiểu dữ liệu đầu vào, chỉ phụ thuộc vào kích thước đầu vào. Số phép so sánh của 2 hàm trên nhiều nhất trong mọi trường hợp.

2.3 Nhận xét

1. Trong trường hợp mảng gần như được sắp xếp, chúng ta nên sử dụng Insertion Sort.
2. Trong trường hợp, phạm vi dữ liệu đầu vào không lớn hơn nhiều so với kích thước dữ liệu thì ta nên sử dụng Counting Sort nếu không cần tiết kiệm bộ nhớ.
3. Trong trường hợp, kích thước dữ liệu nhỏ chúng ta nên sử dụng các hàm sắp xếp đơn giản như Selection Sort, Bubble Sort.
4. Nếu cần sắp xếp với tốc độ nhanh trong nhiều trường hợp khác nhau, thì Flash Sort và Quick Sort vẫn là lựa chọn hàng đầu.

3 Nhận xét tổng thể

3.1 Các thuật toán ổn định

- Flash Sort.
- Radix Sort.

- Quick Sort.
- Merge Sort.

3.2 Các thuật toán không ổn định

- Selection Sort, Bubble Sort: chỉ phù hợp với dữ liệu nhỏ.
- Insertion Sort, Shaker Sort: sử dụng với dữ liệu nhỏ, hoặc dữ liệu gần như được sắp xếp.
- Shell Sort: chạy tốt so với các thuật toán khác trong trường hợp dữ liệu bị sắp xếp ngược và tiết kiệm bộ nhớ lưu trữ.
- Heap Sort: chạy chậm hơn các thuật toán khác trong trường hợp dữ liệu gần như được sắp xếp.
- Counting Sort: không thể sử dụng với các số khác số nguyên. Phụ thuộc vào số có giá trị lớn nhất trong dữ liệu.

4 Cách tổ chức mã nguồn và các cấu trúc, thư viện đặc biệt được sử dụng

4.1 Cấu trúc máy tính

- CPU: AMD Ryzen 5 4600H with Radeon Graphics (12CPUs), 3.0GHz.
- RAM: 16GB

4.2 Cấu trúc mã nguồn

4.2.1 DataGenerator.h và DataGenerator.cpp

- Chứa các hàm phát sinh dữ liệu: random, nearly sorted, sorted, reserve.

4.2.2 DisplayOnConsole.h và DisplayOnConsole.cpp

- Chứa các hàm đọc file vào mảng, ghi mảng vào file.
- 5 hàm xuất ra màn hình của 5 kiểu command.
- 2 hàm tính thời gian chạy hàm và số phép so sánh.
- 2 hàm xuất output parameter của algorithm mode và comparison mode.
- Chứa hàm sao chép mảng.

4.2.3 Enum.h

- Enum là một kiểu dữ liệu tự định nghĩa trong C++. Mục đích dùng để tập hợp các giá trị có ý nghĩa, liên quan với nhau thành một nhóm, giúp việc đọc code dễ hiểu hơn. Khi khởi tạo các biến trong enum thì chỉ được phép gán giá trị các biến là các số nguyên, nếu khởi tạo mặc định thì các biến có giá trị tăng dần theo thứ tự khai báo bắt đầu từ 0. Các biến này là hằng số.
- Cụ thể, ví dụ CommandType chứa các biến Command1, Command2, Command3, Command4, Command5 và biến ErrorCommandType để báo lỗi.

4.2.4 InputDataProcessing.h và InputDataProcessing.cpp

- Hàm **getModeType**: giá trị trả về cho biết là algorithm mode, comparison mode hay đầu vào sai.
- Hàm **getAlgorithmType**: trả về kiểu sắp xếp nào.
- Hàm **getInputType**: cho biết đầu vào là đọc dữ liệu từ file hay là tự phát sinh dữ liệu.
- Hàm **getInputOrderType**: xác định kiểu phát sinh dữ liệu.
- Hàm **getOutputParameterType**: xác định các giá trị tính toán (thời gian, số phép so sánh, hoặc cả hai) được xuất ra màn hình.
- Hàm **convertToString**: mục đích để chuyển đầu vào sang kiểu dữ liệu string giúp dễ xử lý hơn.

4.2.5 ReadCommandLineArgument.h và ReadCommandLineArgument.cpp

- Hàm **readCommandTypeInput**: dùng để xác định yêu cầu của người dùng.

4.2.6 FuncsSupportSort.h và FuncsSupportsort.cpp

- Chứa các hàm phụ của của các hàm sắp xếp.

4.2.7 SortFunctions.h và SortFunctions.cpp

- Chứa 11 hàm sắp xếp không đếm số phép so sánh.

4.2.8 SortFunctionsWithCmp.h và SortFunctionsWithCmp.cpp

- Chứa 11 hàm sắp xếp có đếm số phép so sánh.

5 Các nguồn tham khảo

5.1 Thuật toán

1. Selection Sort
   - Độ phức tạp: Nhấn vào đây
2. Insertion Sort:
   - Độ phức tạp: Nhấn vào đây
3. Bubble Sort:
   - Độ phức tạp: Nhấn vào đây
4. Shaker Sort:
   - Độ phức tạp: Nhấn vào đây
5. Shell Sort:
   - Code hàm: Nhấn vào đây
   - Ứng dụng: Nhấn vào đây

- Độ phức tạp: Nhấn vào đây

6. Heap Sort:
   - Ứng dụng: Nhấn vào đây
   - Độ phức tạp: Nhấn vào đây
7. Merge Sort:
   - Ứng dụng: Nhấn vào đây
   - Độ phức tạp: Nhấn vào đây
8. Quick Sort:
   - Code hàm: Nhấn vào đây
   - Ứng dụng: Nhấn vào đây
   - Độ phức tạp: Nhấn vào đây
9. Counting Sort:
   - Ứng dụng: Nhấn vào đây
   - Độ phức tạp: Nhấn vào đây
10. Radix Sort:
    - Ứng dụng: Nhấn vào đây
    - Độ phức tạp: Nhấn vào đây
11. Flash Sort:
    - Code hàm: Nhấn vào đây