

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO THỰC HÀNH
MÔN HỌC: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT
CHỦ ĐỀ: TÌM HIỂU VÀ THỰC HÀNH CÁC THUẬT TOÁN SẮP XẾP

♡ Giảng viên hướng dẫn: Thầy Lê Đình Ngọc ♡

👤 Sinh viên thực hiện:

21120499 - Nguyễn Duy Long

21120504 - Nguyễn Phương Nam

21120505 - Bùi Thị Thanh Ngân


21120521 - Nguyễn Phúc Phát

Thành phố Hồ Chí Minh - Tháng 11/2022



MỤC LỤC

I. Trang thông tin.....	
II. Giới thiệu.....	
III. Trình bày thuật toán.....	
IV. Kết quả thực nghiệm.....	
1. Bảng số liệu thống kê.....	
2. Đồ thị minh họa.....	
3. Nhận xét chung.....	
V. Tổ chức đề án và các ghi chú.....	
VI. Tài liệu tham khảo.....	

Mẹo: click vào nút  góc phải trang để đến trang mục lục nhanh nhất





II. Giới thiệu

1. Các chủ đề bài báo cáo

Trong bài báo cáo này, chúng em sẽ trình bày về 11 thuật toán sắp xếp được nêu ra trong lab03. Cụ thể, chúng em sẽ trình bày về các thuật toán sắp xếp sau:

- Insertion Sort (Sắp xếp chèn)
- Selection Sort (Sắp xếp chọn)
- Bubble Sort (Sắp xếp nổi bọt)
- Quick Sort (Sắp xếp nhanh)
- Heap Sort (Sắp xếp vun đống)
- Merge Sort (Sắp xếp trộn)
- Shell Sort
- Shaker Sort
- Counting Sort
- Radix Sort
- Flash Sort

2. Mục tiêu bài báo cáo

Bài báo cáo này có mục tiêu nhằm giúp người đọc hiểu rõ hơn về các thuật toán sắp xếp, cách thức hoạt động của chúng, các ưu, nhược điểm của chúng. Đồng thời, bài báo cáo cũng giúp người đọc hiểu rõ hơn về cách thức thực hiện các thuật toán sắp xếp, cách thức thực hiện các thí nghiệm để đánh giá hiệu suất của chúng và cái nhìn tổng quan hơn về độ phức tạp và hiệu suất của các thuật toán qua việc so sánh chúng với nhau thông qua các biểu đồ cụ thể.

Phần 1 – Trình bày các thuật toán sắp xếp: ý tưởng, mã giả, độ phức tạp và những biến thể - cải tiến (nếu có)

Phần 2 – Kết quả thực nghiệm: thống kê các số liệu thu được thông qua việc thực thi chương trình với các dữ liệu đầu vào khác nhau, so sánh với nhau và đưa ra nhận xét.

Phần 3 – Tổ chức mã nguồn: trình bày tổng quan cách thức tổ chức các file chương trình và những thư viện, cấu trúc dữ liệu được sử dụng.

Phần 4 – Tài liệu tham khảo: Các tài liệu tham khảo được sử dụng trong quá trình tìm hiểu.

3. Sơ lược về chương trình:



1.1 Algorithms mode:

command 1: running a sorting algorithm on the given input data
prototype: [Execution file] -a [Algorithm] [Given input] [Output params]
Example: a.exe -a radix-sort input.txt -both

command 2: Run a sorting algorithm on the data generated automatically with specified size and order
prototype: [Execution file] -a [Algorithm] [Input size] [Input order] [output params]
Example: a.exe -a selection-sort 50 -rand -time

command 3: Run a sorting algorithm on ALL data arrangements of a specified size.
prototype: [Execution file] -a [Algorithm] [Input size] [Output params]
Example: a.exe -a binary-insertion-sort 7000 -comp

1.2 Comparison mode:

command 4: Run two sorting algorithms on the given input.
prototype: [Execution file] -c [Algorithm 1] [Algorithm 2] [Given input]
Example: a.exe -c heap-sort merge-sort input.txt

command 5: Run two sorting algorithms on the data generated automatically.
prototype: [Execution file] -c [Algorithm 1] [Algorithm 2] [Input size] [Input order]
Example: a.exe -c quick-sort merge-sort 100000 -sorted

1.3 Given input:

1st line : an integer n, the number of elements in the array
2nd line: n integers, the elements of the array, separated by a single space

1.4 Input order:

-rand: randomized data
-nsorted: nearly sorted data
-sorted: sorted data
-rev: reverse sorted data

1.5 Output params:

-time: running time
-comp: number of comparisons
-both:

1.6 Output order:

-time : algorithm running time
-comp : number of comparisons
-both : both time and comparisons

1.7 Data size:

10 000, 30 000,
50 000, 100 000,
300 000, 500 000

III. Trình bày thuật toán

1. Selection Sort

2. Insertion Sort

3. Bubble Sort

4. Shaker Sort

5. Shell Sort

6. Quick Sort

7. Heap Sort

8. Merge Sort



9. Radix Sort

10. Counting Sort

11. Flash Sort

1 Selection Sort

1. Ý tưởng thuật toán

Mỗi bước sẽ di chuyển một phần tử nhỏ nhất sang bên trái, từ đó mảng sẽ dần được chia làm 2 phần:

Bên trái là mảng đã được sắp xếp tăng dần.

Bên phải là mảng chưa được sắp xếp.

2. Pseudocode:



```
For i = 0 to n - 2
```

```
    min_index = chỉ số phần tử nhỏ nhất trong khoảng i + 1 đến n - 1
```

```
    swap (a(i), a(min_index))
```

3. Độ phức tạp

Không gian: $O(1)$

Thời gian: $O(n^2)$ cho mọi trường hợp.

4. Biến thể và cải tiến

Heap sort sử dụng cùng ý tưởng tìm các giá trị lớn nhất nhỏ nhất, nhỏ nhất nhưng dùng đến cấu trúc heap nên độ phức tạp giảm còn $O(n \log n)$.

Double selection sort, tìm cùng lúc giá trị lớn nhất và nhỏ nhất sau đó di chuyển các giá trị này đến đầu và cuối mảng.

2 Insertion Sort

1. Ý tưởng thuật toán

1. Chèn phần tử thứ hai vào vị trí thích hợp trong mảng con đã được sắp xếp.
2. Chèn phần tử thứ ba vào vị trí thích hợp trong mảng con đã được sắp xếp.
3. Lặp lại cho đến khi chèn phần tử cuối cùng vào vị trí thích hợp trong mảng con đã được sắp xếp.



2. Pseudocode:

```
for i = 1 to n - 1
```

```
  x = a[i]
```

```
  j = i - 1
```

Duyệt j và tìm vị trí thích hợp cho x, đồng thời dịch các phần tử sang phải để tạo chỗ cho x

Chèn x vào vị trí thích hợp

3. Độ phức tạp

Không gian: $O(1)$

Thời gian:

- Trung bình, thuật toán có độ phức tạp là $O(n^2)$
- Trường hợp tốt nhất là với đầu vào đã được sắp xếp đúng thứ tự : $O(n)$
- Trường hợp xấu là dãy bị đảo ngược thứ tự hoàn toàn : $O(n^2)$

4. Biến thể và cải tiến

Shell sort sử dụng cùng ý tưởng tìm các giá trị lớn nhất nhỏ nhất.

Áp dụng trong flash sort.

Có thể dùng binary search để giảm số lần so sánh.

3 Bubble Sort

1. Ý tưởng thuật toán



1. So sánh 2 phần tử liền kề, nếu phần tử đứng trước lớn hơn phần tử đứng sau thì hoán đổi chỗ 2 phần tử này.
2. Lặp lại cho đến khi không còn phần tử nào cần hoán đổi chỗ.

2. Pseudocode:

```
for i = 0 to n - 2
  for j = n - 1 downTo i + 1
    if a[j] < a[j - 1]
      swap (a[j], a[j - 1])
```

3. Độ phức tạp

Không gian: $O(1)$

Thời gian: Với thuật toán trên thì độ phức tạp luôn là $O(n^2)$

4. Biến thể và cải tiến

- Cải tiến:

Trong mỗi vòng lặp của biến j ở trên, kiểm tra xem nếu không có phép hoán vị nào được thực hiện tức mảng đã đúng vị trí ta sẽ dừng thuật toán ngay lập tức. Khi đó, trong trường hợp tốt nhất mảng đã được sắp xếp độ phức tạp về thời gian là $O(n)$

- Biến thể là Shaker Sort.



4 Shaker Sort

1. Ý tưởng thuật toán

1. Chọn khoảng xét từ đầu đến cuối mảng
2. Thực hiện duyệt khoảng xét bằng 2 lượt, so sánh 2 phần tử liền kề và hoán vị: (Lượt đi: đẩy phần tử lớn nhất về cuối; Lượt về: đẩy phần tử nhỏ nhất về đầu)
3. Đồng thời, mỗi lượt đi ghi nhận lại vị trí hoán vị cuối cùng, để thu hẹp 2 phía khoảng xét ở mỗi lượt.
4. Lặp lại bước 2 và 3 đến khi 2 đầu khoảng xét giao nhau.

2. Pseudocode:

```
left = đầu mảng
right= cuối mảng
k = 0 // vị trí hoán vị cuối
while left < right // l->r: khoảng sắp xếp
    for i = left to right
        if a[i] > a[i+1]
            swap 2 phần tử để đẩy phần tử lớn nhất về cuối khoảng
            cập nhật vị trí k = i
    right = k // thu hẹp khoảng xét bên phải
    for i = right to left
        if a[i] < a[i-1]
            swap 2 phần tử để đẩy phần tử nhỏ nhất về đầu khoảng
            cập nhật vị trí k = i
    left = k // thu hẹp khoảng xét bên trái
```

3. Độ phức tạp

Không gian: $O(1)$

Thời gian:

- + Trung bình, thuật toán có độ phức tạp là $O(n^2)$
- + Trường hợp tốt nhất là với đầu vào đã được sắp xếp đúng thứ tự :



$O(n)$

+ Trường hợp xấu là dãy bị đảo ngược thứ tự hoàn toàn : $O(n^2)$

Có độ phức tạp tương tự như Bubble Sort nhưng có thể tối ưu hơn về thời gian trong trường hợp tốt nhất.

5 Shell Sort

1. Ý tưởng thuật toán

Shell sort là một biến thể cải tiến hơn của insertion sort.

Thuật toán sử dụng insertion sort lên các phần tử cách xa nhau sau đó thu hẹp dần khoảng cách này.

Như vậy mảng sẽ được chia thành các mảng con với các phần tử có khoảng cách là h

sắp xếp các mảng con này bằng insertion sort và lặp lại các bước trên với khoảng cách thu hẹp dần thì ta được mảng có thứ tự.

2. Pseudocode:

```
h = n / 2 // khoảng cách giữa các phần tử
While h > 0:

    For i = h đến n - 1:

        (Selection sort cho mảng từ 0 - i với bước chạy là h)

        temp = a(i)

        Đẩy tất cả các phần tử lớn hơn temp lên h đơn vị

        Chèn temp vào vị trí thích hợp

    h / 2
```

3. Độ phức tạp



Không gian: sắp xếp tại chỗ nên là $O(1)$

Thời gian:

+Trung bình thời gian chạy của shell sort sẽ tùy thuộc vào h ta chọn, với những h thích hợp

ta có thể tối ưu shell sort hơn nữa. Với $h = h/2$ ta chọn ở trên:

+Trường hợp tệ nhất là khi shell sort trở thành insertion sort, lúc này độ phức tạp thời gian là $O(n^2)$

+Trường hợp tốt nhất là khi mảng đã được sắp xếp sẵn thì độ phức tạp sẽ là $O(n \log n)$

4. Biến thể và cải tiến

Dobosiewicz sort

Shaker sort

Insertion sort

6 Quick Sort

1. Ý tưởng thuật toán

1. Chọn một phần tử làm pivot.
2. Đưa các phần tử nhỏ hơn pivot về bên trái pivot, các phần tử lớn hơn pivot về bên phải pivot.
3. đệ quy sắp xếp các mảng con bên trái và bên phải pivot.

2. Pseudocode:

```
quickSort(a[], left, right) {  
    i = left, j = right;    // left, right là chỉ số đầu và cuối của mảng  
    pivot = a[(left + right) / 2]; // Chọn pivot là phần tử ở giữa mảng  
  
    while (i <= j) {  
        khi a[i] < pivot thì i++  
        khi a[j] > pivot thì j--  
    }
```



```
    khi a[i] >= pivot và a[j] <= pivot thì đổi chỗ a[i] và a[j]
    if (i <= j) {
        swap(a[i], a[j]);
        i++;
        j--;
    }
    // thực hiện đến khi i > j
}
// Đây là lúc mảng đã được chia thành 2 mảng con: a[left] -> a[j] và a[i] ->
a[right]
// Tiếp tục sắp xếp 2 mảng con này bằng quick sort
if (left < j) {
    quickSort(a, left, j);
}
if (i < right) {
    quickSort(a, i, right);
}
}
```

3. Độ phức tạp

Không gian: $O(\log n)$ vì đệ quy

Thời gian:

- + Trung bình: $O(n \log n)$
- + Trường hợp tốt nhất: $O(n \log n)$
- + Trường hợp tệ nhất: $O(n^2)$

4. Biến thể và cải tiến

Quick sort 3-way: sử dụng 3 pivot để chia mảng thành 3 phần.

Quick sort random: chọn pivot ngẫu nhiên.

Quick sort median: chọn pivot là phần tử ở giữa mảng.

7 Heap Sort

1. Ý tưởng thuật toán



1. Tạo max - heap từ mảng.
2. Lấy phần tử lớn nhất tại vị trí 0 và đưa về cuối mảng, giảm kích thước của mảng đi 1.
3. heapify lại mảng.
4. Lặp lại bước 2 và 3 cho đến khi kích thước của mảng bằng 1.

2. Pseudocode:

```
// tạo max-heap từ heapify 1 nửa phần tử mảng
for i = n/2 - 1 to 1
    heapify(a, i, 0)
for i = n - 1 to 1
    swap a[0] and a[i]
    heapify(a, i, 0)
// viết hàm để heapify
heapify(a, n, i)
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and a[l] > a[largest] : largest = l
    if r < n and a[r] > a[largest] : largest = r
    if largest != i // nếu i không phải là largest
        swap a[i] and a[largest]
        heapify(a, n, largest)
```

3. Độ phức tạp

Không gian: $O(1)$
Thời gian: $O(n \log n)$ cho mọi trường hợp.

8

Merge Sort

1. Ý tưởng thuật toán

Sử dụng thuật toán chia để trị xử lý 2 nửa mảng rồi trộn chúng lại với nhau.
Cụ thể:

Mảng được chia đôi thành 2 phần bằng đệ quy cho đến khi mảng chỉ còn 1 phần



tử.

Trộn 2 mảng đã có thứ tự này lại bằng cách lấy lần lượt số nhỏ nhất ở đầu 2 mảng bỏ vào mảng chính.

2. Pseudocode:

```
// Hàm trộn 2 mảng có thứ tự left và right vào mảng chính:
While 2 cả mảng left và right còn phần tử:
    Lấy lần lượt phần ở đầu 2 mảng
    So sánh 2 phần tử này
    Phần tử nào nhỏ hơn thì lấy bỏ vào mảng chính

Nếu mảng left còn phần tử:
    Bỏ phần còn lại đó vào phía sau mảng chính

Nếu mảng right còn phần tử:
    Bỏ phần còn lại đó vào phía sau mảng chính

// Hàm mergeSort:
Nếu mảng có n <= 1 thì dừng
Chia mảng a thành 2 phần bằng nhau là
    Mảng left
    Mảng right
Thực hiện gọi đệ qui sắp xếp 2 mảng này
Gọi hàm trộn 2 mảng này về mảng chính
```

3. Độ phức tạp

Không gian: $O(n)$ sử dụng để lưu trữ 2 mảng con.
Thời gian: Thời gian chạy của merge sort khá ổn định, trong tất cả các trường hợp đều là $O(n \log n)$.

4. Biến thể và cải tiến

Các biến thể của merge sort chủ yếu tập trung vào việc giảm độ phức tạp về không gian và giảm số lần copy phần tử



Block sort: là một in-place sorting với độ phức tạp ổ định là $O(n \log n)$
Katajainen et al: cũng là một in-place sorting với độ phức tạp $O(n \log n)$ chưa được ổn định

9 Radix Sort

1. Ý tưởng thuật toán

Khác với các thuật toán trước, Radix sort là một thuật toán tiếp cận theo một hướng hoàn toàn khác.

Nếu như trong các thuật toán khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử

thì Radix sort lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó nó còn có tên là Postman's sort.

Nó không hề quan tâm đến việc so sánh giá trị của phần tử và bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

Coi các phần tử trong mảng sắp xếp được cấu thành từng các lớp có độ ưu tiên khác nhau.

Ví dụ, các số tự nhiên chia thành các lớp như: hàng đơn vị, hàng chục, hàng trăm, hàng nghìn, ...

Bước đầu tiên ta sắp xếp dãy các phần tử bằng cách so sánh các phần tử ở lớp có độ ưu tiên thấp nhất (ví dụ các chữ số hàng đơn vị).

Số nào có hàng đơn vị thấp hơn thì ta đưa lên trên. Như vậy các số có hàng đơn vị là 0 ở trên cùng, sau đó đến các số có hàng đơn vị là 1, ...

Sau bước 1, ta thu được 1 thứ tự sắp xếp mới.

Ta lại làm tương tự với các lớp kế tiếp (chữ số thuộc hàng chục, hàng trăm, ...) cuối cùng ta sẽ có dãy đã sắp xếp.

2. Pseudocode:

```
max = số chữ số của phần tử lớn nhất
table = mảng các queue gồm 10 phần tử( các lớp từ 0->9)

for k = 0 to max-1 do:
    for i = 0 to n-1 do:
        unit = chữ số của hàng thứ k
        thêm a[i] vào queue table[unit]
    end for
    i = 0
    j = 0
```




```
// Gán các phần tử trong queue vào mảng theo thứ tự các lớp từ 0 ->9
while j < 10 do:
    while table[j] có phần tử do:
        a[i] = lấy ra phần tử đầu trong queue
        i = i + 1
    j = j + 1
end for
```

3. Độ phức tạp

Với k là số chữ số của phần tử lớn nhất trong mảng, n là số phần tử trong mảng:

Không gian: $O(n + k)$ vì ta cần mảng các queue gồm k phần tử.

Thời gian: $O(nk)$ vì ta cần duyệt qua tất cả các phần tử trong mảng và các chữ số của các phần tử.

10 Counting Sort

1. Ý tưởng thuật toán

1. Tìm phần tử lớn nhất trong mảng
2. Tạo mảng mới có kích thước bằng phần tử lớn nhất + 1, khởi tạo các phần tử bằng 0
3. Lưu số lần xuất hiện của các phần tử trong mảng tại vị trí tương ứng trong mảng mới
4. Duyệt mảng mới, nếu phần tử tại vị trí $i > 0$ thì gán chỉ số i vào mảng ban đầu tại vị trí j
5. Tăng j lên 1 để xét phần tử tiếp theo, giảm giá trị đếm của phần tử tại chỉ số i đi 1
6. Lặp lại bước 4, 5 cho đến khi $j = n$ hay $i = \max$

2. Pseudocode:



```
max = phần tử lớn nhất trong mảng

count = mảng mới có kích thước bằng max + 1, khởi tạo các phần tử bằng 0

for (int i = 0; i < n; i++) {
    count[a[i]]++;    // Lưu số lần xuất hiện của các phần tử trong mảng tại
    vị trí tương ứng trong mảng mới
}

i = j = 0;    // i : chỉ số trong mảng count, j : chỉ số của phần tử đang xét
trong mảng ban đầu

while (i < max + 1) {
    if (count[i] > 0) { // Nếu count[i] > 0 tức là có phần tử có giá trị bằng
i trong mảng ban đầu
        a[j++] = i;    // Đưa phần tử đó vào mảng mới ở vị trí j, tăng j để
xét phần tử tiếp theo
        count[i]--;    // Giảm số lần xuất hiện của phần tử đó để duyệt tiếp
phần tử tiếp theo
    }
    else {                // Nếu không có phần tử nào có giá trị bằng i thì xét
giá trị tiếp theo
        i++;
    }
}
```

3. Độ phức tạp

Là một thuật toán tuyến tính, với k là phần tử lớn nhất trong mảng thì:

Không gian: $O(n + k)$

Thời gian: $O(n + k)$

4. Ưu, nhược điểm:

Ưu điểm:

- + Hiệu quả nếu phạm vi dữ liệu đầu vào không lớn hơn đáng kể so với kích thước của mảng đầu vào.

- + Có thể mở rộng để sắp xếp các phần tử có giá trị âm.

Nhược điểm:



- + Xét trường hợp $k = n \times n \Rightarrow$ worst case
- + Không phải là thuật toán tại chỗ (in-place algorithm) vì cần mảng mới để lưu số lần xuất hiện của các phần tử.

1 1 Flash Sort

1. Ý tưởng thuật toán

Tư tưởng chính của thuật toán là dựa trên sự phân lớp phần tử (Subclasses Arrangement). FlashSort bao gồm ba khối logic:

Phân loại các phần tử (Elements Classification);

Phân bố các phần tử vào đúng các phân lớp (Elements Permutation);

Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự (Elements Ordering).

1. Tìm phần tử lớn nhất và nhỏ nhất trong mảng
2. Tính giá trị m theo công thức : $m = (n * \alpha)$, α thường là 0.45
3. Tạo mảng mới có kích thước bằng m , khởi tạo các phần tử bằng 0
4. Lặp qua mảng gốc, tính chỉ số của phần tử tại vị trí i trong mảng mới theo công thức: $\text{index} = (m - 1) * (a[i] - \min) / (\max - \min)$
5. Tăng giá trị của phần tử tại vị trí index trong mảng mới lên 1
6. Lặp qua mảng mới, tính vị trí bắt đầu của các phân lớp bằng cách cộng dồn các phần tử trong mảng mới
7. Hoán đổi $a[\max]$ với $a[0]$
8. Lặp và hoán đổi để đưa các phần tử về đúng phân lớp
9. Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự bằng thuật toán Insertion Sort

2. Pseudocode:

```
min = phần tử nhỏ nhất trong mảng
max = chỉ số phần tử lớn nhất trong mảng
m = (n * alpha)
Class = mảng mới có kích thước bằng m, khởi tạo các phần tử bằng 0
c1 = (m - 1) / (a[max] - min)

i = 0, khi i < n thì {
    k = c1 * (a[i] - min)
    ++Class[k]; // đếm số phần tử trong mảng gốc thuộc phân lớp thứ k
}
```



```

for (int i = 1; i < m; i++) { // tính vị trí bắt đầu của các phân lớp
    Class[i] += Class[i - 1];
}

swap(a[max], a[0]);
nmove = j = 0; k = m - 1; // nmove là số phần tử đã được sắp xếp, j là chỉ số
phần tử đang xét, k là chỉ số phân lớp đang xét

// hoán đổi các phần tử về đúng phân lớp: hoán vị tối đa n - 1 lần
while ( nmove < n - 1)
{
    khi ( j > Class[k] - 1) {
        tăng j lên 1 để xét phần tử tiếp theo
        tính lại chỉ số phân lớp của phần tử a[j]
    }
    if ( k < 0) break;
    khi (j != Class[k])
    {
        tính lại chỉ số phân lớp
        tìm vị trí đích của phần tử a[j] : pos
        swap(a[j], a[pos]);
        ++nmove; // tăng số phần tử đã được sắp xếp
    }
}
sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự bằng thuật toán
Insertion Sort

```

3. Độ phức tạp

Không gian: $O(m)$ với m là số phân lớp

Thời gian:

- + Tốt nhất: $O(n)$
- + Trung bình là $O(n + m)$
- + Trường hợp xấu nhất là $O(n^2)$

Nhìn lại toàn bộ các giai đoạn của thuật toán, ta thấy như sau:

- Giai đoạn phân lớp đòi hỏi độ phức tạp $O(n)$ và $O(m)$
- Giai đoạn Hoán vị đòi hỏi độ phức tạp $O(n)$ (vì mỗi phần tử chỉ phải đổi chỗ đúng một lần, và n lần cho n phần tử)
- Giai đoạn Insertion_Sort đòi hỏi độ phức tạp $O(n^2/m)$ (mỗi 1 phân lớp đòi hỏi độ phức tạp $O((n/m)^2)$ và m phân lớp đòi hỏi $O(m*(n/m)^2)$)

IV. Kết quả thực nghiệm



1. Bảng số liệu thống kê

- Cấu hình máy thử nghiệm:
- CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz
- RAM: 8 GB
- Compiler: Visual Studio Code
- Dưới đây là 4 bảng thống kê thời gian chạy và số lần so sánh của thuật toán với các trường hợp khác nhau về kiểu dữ liệu (rand, sorted nsorted, rev) và kích thước dữ liệu (10 000, 30 000, 50 000, 100 000, 300 000, 500 000).

Data order: Randomized												
Data size	10000		30000		50000		100000		300000		500000	
Resulting static	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	114	100009999	1206	900029999	4250	2500049999	15461	10000099999	169582	9E+10	321604	2.5E+11
Insertion sort	140	49888053	1161	450399755	3099	1243244904	12235	4994997990	143806	4.505E+10	248589	1.2507E+11
Bubble sort	337	100009999	3615	900029999	12399	2500049999	55462	10000099999	605323	9E+10	1.03E+06	2.5E+11
Shaker sort	326	66706197	4865	600133206	9502	1659825501	51376	6651609372	431089	6.004E+10	766550	1.6677E+11
Shell sort	3	643241	13	2255089	31	4539415	57	10086664	129	33677641	231	65384404
Heap sort	3	637669	16	2150126	33	3772835	38	8045288	152	26490095	174	45967068
Merge sort	10	40004	19	120003	33	200001	50	400002	139	1199996	264	1999972
Quick sort	1	283760	8	902518	11	1591004	15	3348761	46	10921154	73	18152470
Counting sort	0	80002	2	240002	2	348303	3	598306	11	1598306	9	2598306
Radix sort	7	180107	39	660131	62	1100131	86	2200131	284	6600131	353	11000131
Flash sort	1	96925	3	292123	4	481324	9	921905	23	2825275	32	4475339

Data order: Nearly Sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resulting static	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	119	100009999	1111	900029999	3264	2500049999	15992	10000099999	170810	9E+10	436095	2.5E+11
Insertion sort	1	187998	1	345710	1	514830	2	553922	4	1365838	10	1949034
Bubble sort	131	100009999	1005	900029999	4439	2500049999	14466	10000099999	153717	9E+10	423699	2.5E+11
Shaker sort	1	198963	1	363350	3	537937	2	504621	5	1104900	8	1521403
Shell sort	2	415064	4	1273329	11	2279674	17	4623218	40	15446521	66	25642430
Heap sort	3	669767	11	2236548	18	3924706	34	8364543	158	27413321	280	47405156
Merge sort	7	39980	31	102896	25	165036	42	300006	115	900006	245	1500006
Quick sort	0	155017	1	501965	3	913890	7	1927711	19	6058284	28	10310773
Counting sort	0	80002	0	240002	1	400002	4	800002	11	2400002	6	4000002
Radix sort	7	180107	24	660131	61	1100131	95	2200131	417	7800155	537	13000155
Flash sort	1	127964	2	383960	4	639962	8	1279966	18	3839962	38	6399964



Data order: Sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resulting static	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	117	100009999	1132	900029999	3266	2500049999	16046	10000099999	168219	9E+10	434372	2.5E+11
Insertion sort	0	29998	0	89998	0	149998	1	299998	2	899998	6	1499998
Bubble sort	138	100009999	1024	900029999	4471	2500049999	14578	10000099999	153356	9E+10	422836	2.5E+11
Shaker sort	1	20002	0	60002	0	100002	0	200002	0	600002	2	1000002
Shell sort	1	360042	2	1170050	5	2100049	15	4500051	48	15300061	84	25500058
Heap sort	2	670333	9	2236652	15	3925355	35	8365084	163	27413234	299	47404890
Merge sort	3	30006	12	90006	14	150006	28	300006	120	900006	284	1500006
Quick sort	0	154959	1	501929	3	913850	7	1927691	24	6058228	33	10310733
Counting sort	0	80002	0	240002	1	400002	3	800002	13	2400002	14	4000002
Radix sort	7	180107	28	660131	60	1100131	67	2200131	388	7800155	624	13000155
Flash sort	1	127992	2	383992	5	639992	6	1279992	26	3839992	43	6399992

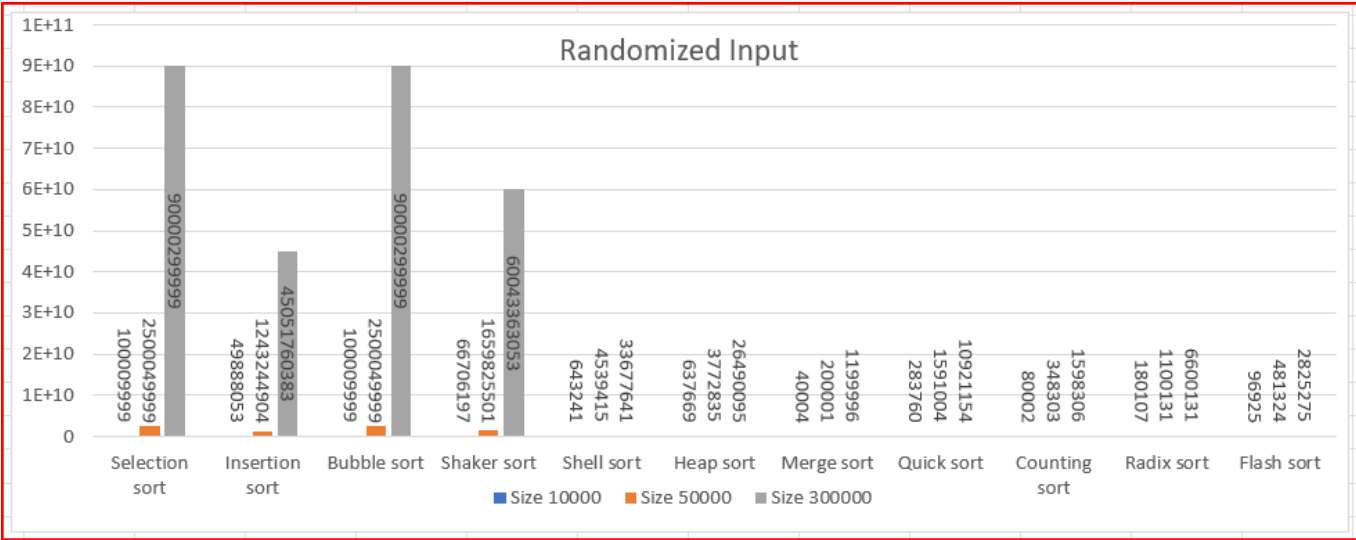
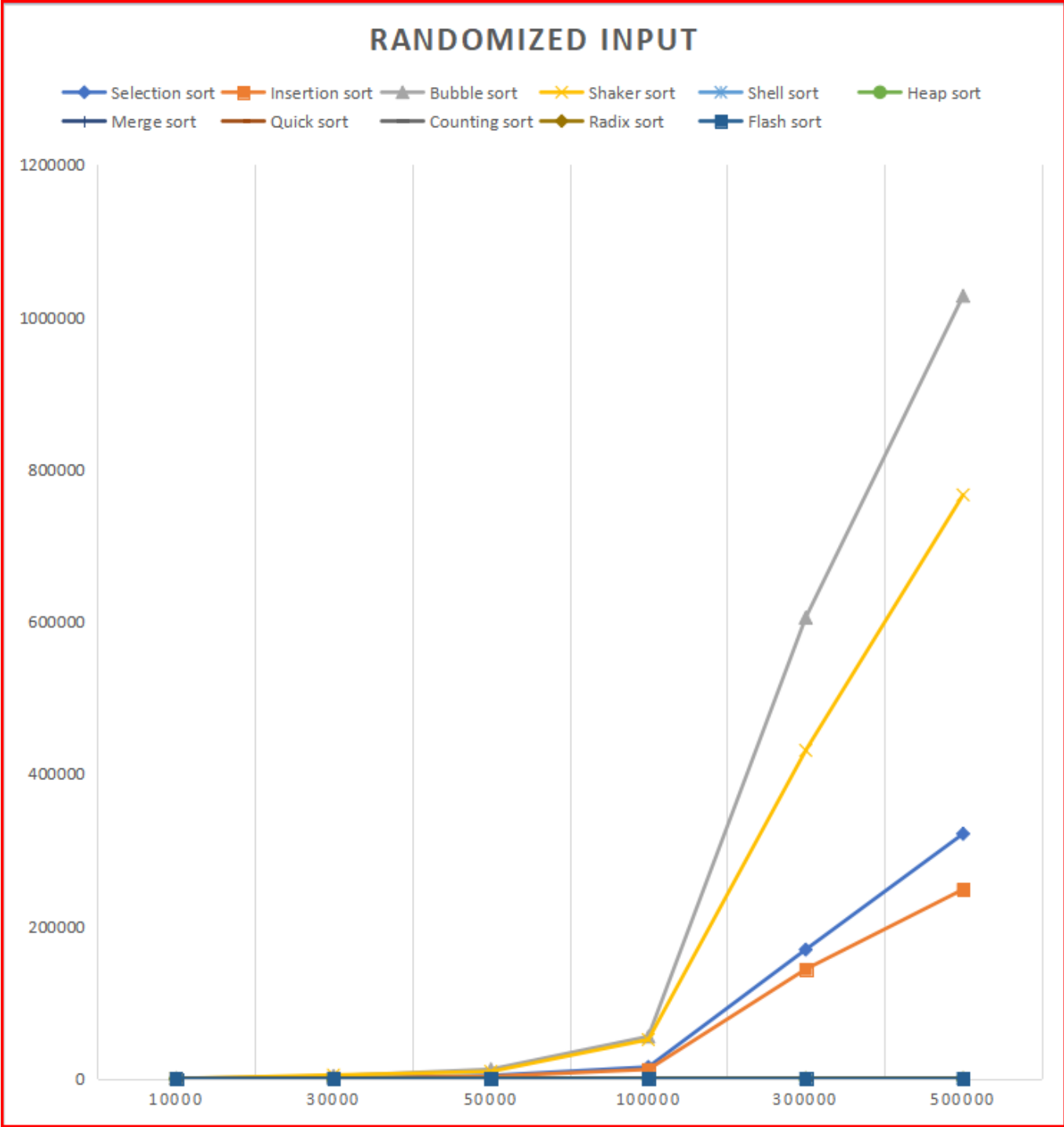
Data order: Reversed												
Data size	10000		30000		50000		100000		300000		500000	
Resulting static	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	122	100009999	1115	900029999	3328	2500049999	13955	10000099999	163605	9E+10	265040	2.5E+11
Insertion sort	269	100009999	1742	900029999	4842	2500049999	24459	10000099999	290655	9E+10	520298	2.5E+11
Bubble sort	347	100009999	4233	900029999	10953	2500049999	74091	10000099999	682241	9E+10	865325	2.5E+11
Shaker sort	436	100005001	5082	900015001	12552	2500025001	74680	10000050001	506411	9E+10	784986	2.5E+11
Shell sort	1	475175	6	1554051	11	2844628	24	6089190	43	20001852	61	33857581
Heap sort	2	606775	9	2063328	16	3612728	59	7718947	69	25569383	99	44483352
Merge sort	6	30007	10	90007	23	150007	49	300007	80	900007	216	1500007
Quick sort	0	164975	2	531939	3	963861	10	2027703	13	6358249	21	10810747
Counting sort	0	80002	1	240002	2	400002	4	800002	7	2400002	11	4000002
Radix sort	5	180107	28	660131	41	1100131	160	2200131	205	7800155	323	13000155
Flash sort	0	110501	2	331501	4	552501	8	1105001	17	3315001	23	5525001

2. Đồ thị minh họa

Các biểu đồ sau đây sẽ giúp ta dễ dàng thống kê và hình dung để có cái nhìn chung về sự thay đổi của thời gian chạy và số lần so sánh của thuật toán với các trường hợp khác nhau về kiểu dữ liệu (rand, sorted nsorted, rev) và kích thước dữ liệu (10 000, 30 000, 50 000, 100 000, 300 000, 500 000).

Trường hợp dữ liệu ngẫu nhiên





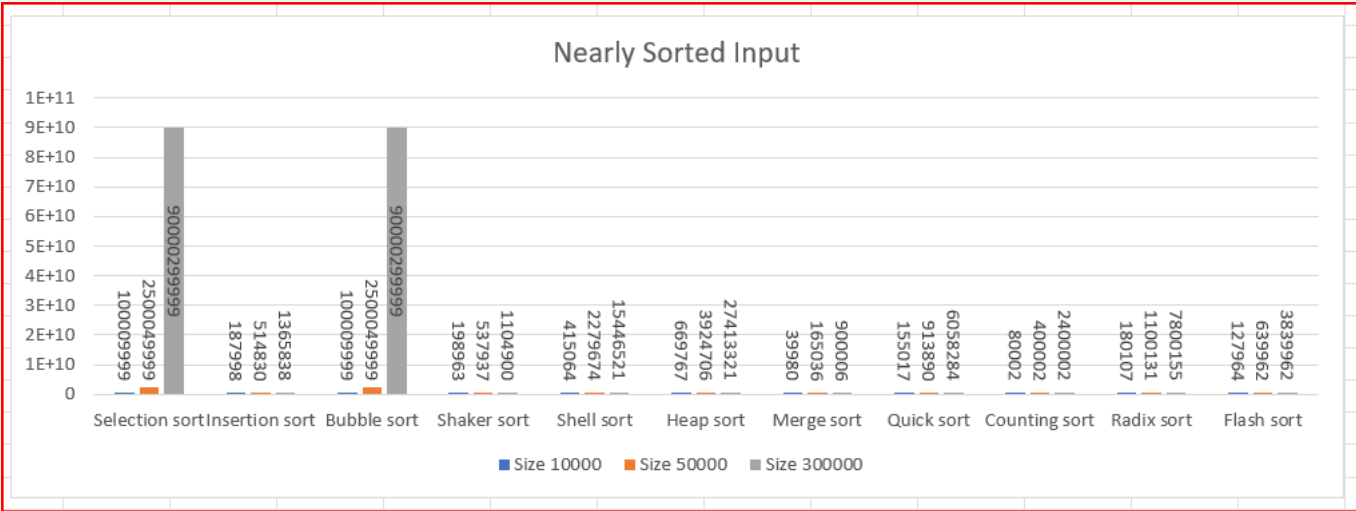
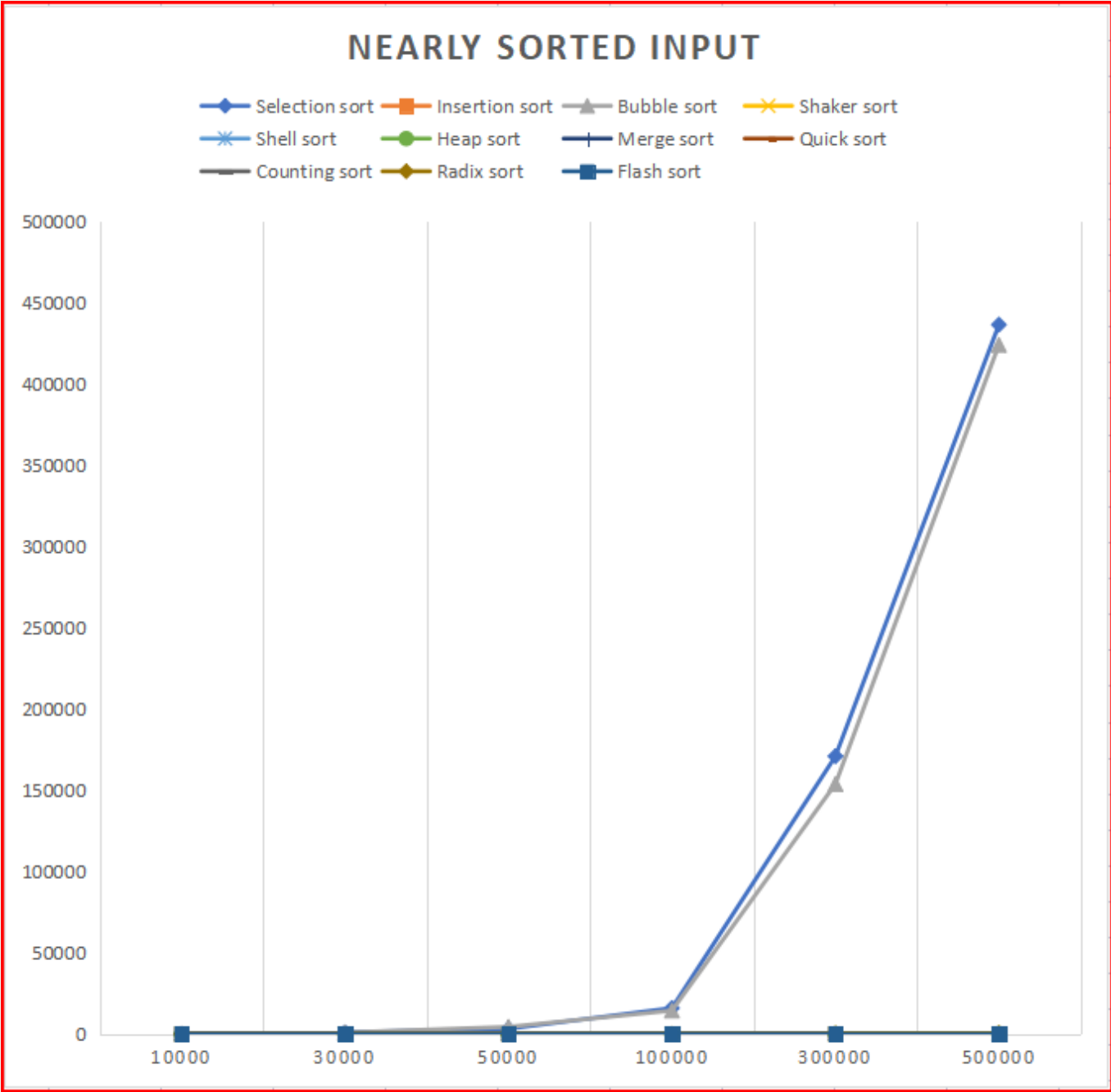
```
// Hàm phát sinh mảng dữ liệu ngẫu nhiên
void GenerateRandomData(int a[], int n)
{
    srand((unsigned int)time(NULL));

    for (int i = 0; i < n; i++)
    {
        a[i] = rand()%n;
    }
}
```

- Về thời gian chạy: (sắp xếp theo hướng thời gian chạy tăng dần)
- + Counting < Flash < Quick < Heap ~ Shell ~ Merge ~ Radix < Insertion < Selection < Shaker < Bubble
- + Theo kết quả thực nghiệm, với lượng dữ liệu lên đến 500 000:
 - + Counting Sort có thời gian chạy trong vòng 9ms
 - + Trong khi đó, Bubble Sort có thời gian chạy 1.03E+6 tức ~ 1.03 triệu ms ~ 1030s ~ 17 phút
- + Độ phức tạp của thuật toán Counting Sort là $O(n+k)$ với k là khoảng giá trị của dữ liệu đầu vào.
- + Đối với trường hợp dữ liệu ngẫu nhiên này, hàm được thiết kế để sinh số ngẫu nhiên phạm vi từ 0 đến n -1, do đó $k = n - 1$. Vậy nên độ phức tạp của thuật toán Counting Sort là $O(n)$ tốt nhất trong các thuật toán sắp xếp.
- + Các thuật toán Heap Sort, Shell Sort, Merge Sort, Radix Sort có sự thời gian chạy thay đổi không nhiều khi số lượng dữ liệu tăng lên. Điều này là do các thuật toán này có độ phức tạp $O(n \log n)$ nên khi số lượng dữ liệu tăng lên, thời gian chạy tăng lên nhưng không nhiều.
- + Các thuật toán Selection Sort, Insertion Sort, Shaker Sort, Bubble Sort có sự thời gian chạy thay đổi rõ rệt khi số lượng dữ liệu tăng lên. Điều này là do các thuật toán này có độ phức tạp $O(n^2)$ nên khi số lượng dữ liệu tăng lên, thời gian chạy cũng tăng lên theo cấp số nhân.
- Về số lần so sánh: (sắp xếp theo hướng số lần so sánh tăng dần)
- + Merge < Counting < Flash < Radix < Quick < Heap < Shell < Insertion < Shaker < Selection ~ Bubble
- + Thuật toán có phép so sánh nhiều nhất là Selection và Bubble, tiếp đến là Shaker và Insertion.
- các thuật toán này có số lần so sánh tăng lên rất nhiều số lượng dữ liệu tăng lên 500 000.
- + Thuật toán có phép so sánh ít nhất là Merge và Counting, tiếp đến là Flash, Radix, Quick, Heap, Shell. Chúng có lượng phép so sánh tăng lên không nhiều khi số lượng dữ liệu tăng lên 500 000.



Trường hợp dữ liệu gần như đã sắp xếp



- Về thời gian chạy: (sắp xếp theo hướng thời gian chạy tăng dần)
- + Counting ~ Shaker ~ Insertion < Quick ~ Flash < Shell < Merge ~ Heap < Radix < Bubble ~ Selection

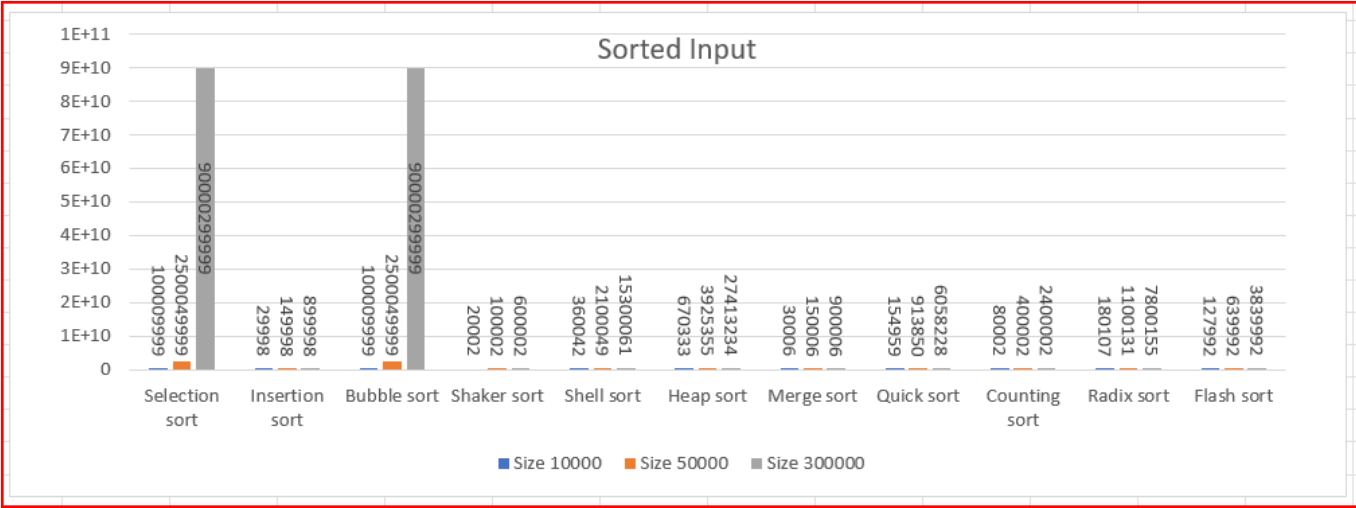
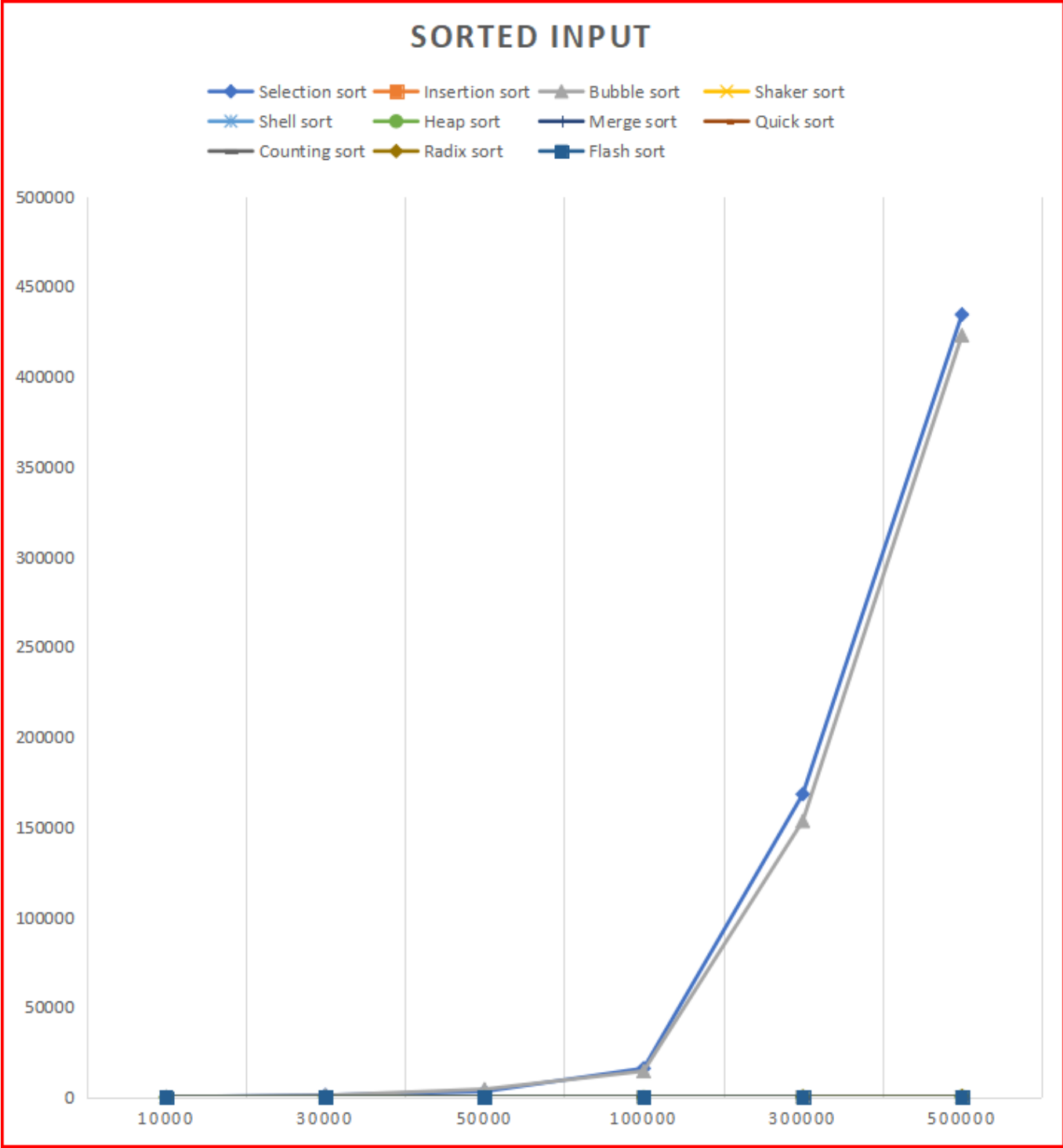
- + Theo kết quả thực nghiệm, với lượng dữ liệu lên đến 500 000:
 - + Counting Sort có thời gian chạy trong vòng 6ms
 - + Trong khi đó, Selection Sort có thời gian chạy 436095ms tức ~ 436s ~ 7 phút
- + Hai thuật toán Bubble và Selection có tốc độ tăng thời gian chạy rõ rệt khi lượng dữ liệu tăng từ 100 000 lên 500 000.
- + Shaker và Insertion trong trường hợp này đạt độ phức tạp thời gian tốt nhất là $O(n)$ vì dữ liệu đã gần như sắp xếp.
- + Các thuật toán còn lại có thời gian chạy tăng không đáng kể khi lượng dữ liệu thay đổi từ 10 000 -> 500 000

- Về số phép so sánh : (sắp xếp theo hướng số lần so sánh tăng dần)
- + Merge < Shaker < Insertion < Counting < Flash < Quick < Radix < Shell < Heap < Bubble ~ Selection

- + Bubble và Selection có số lần so sánh tăng rõ rệt khi lượng dữ liệu tăng từ 50 000 lên 300 000
- + Các thuật toán còn lại có số lần so sánh tăng không đáng kể khi lượng dữ liệu thay đổi từ 10 000 -> 500 000

Trường hợp dữ liệu đã sắp xếp





- Về thời gian chạy: (sắp xếp theo hướng thời gian chạy tăng dần)
- + Shaker ~ Insertion < Counting ~ Quick < Flash < Shell < Merge ~ Heap < Radix < Bubble ~ Selection

- + Theo kết quả thực nghiệm, với lượng dữ liệu lên đến 500 000:
 - + Shaker Sort có thời gian chạy trong vòng 2ms
 - + Trong khi đó, Selection Sort có thời gian chạy 434372ms tức ~ 434s ~ 7 phút

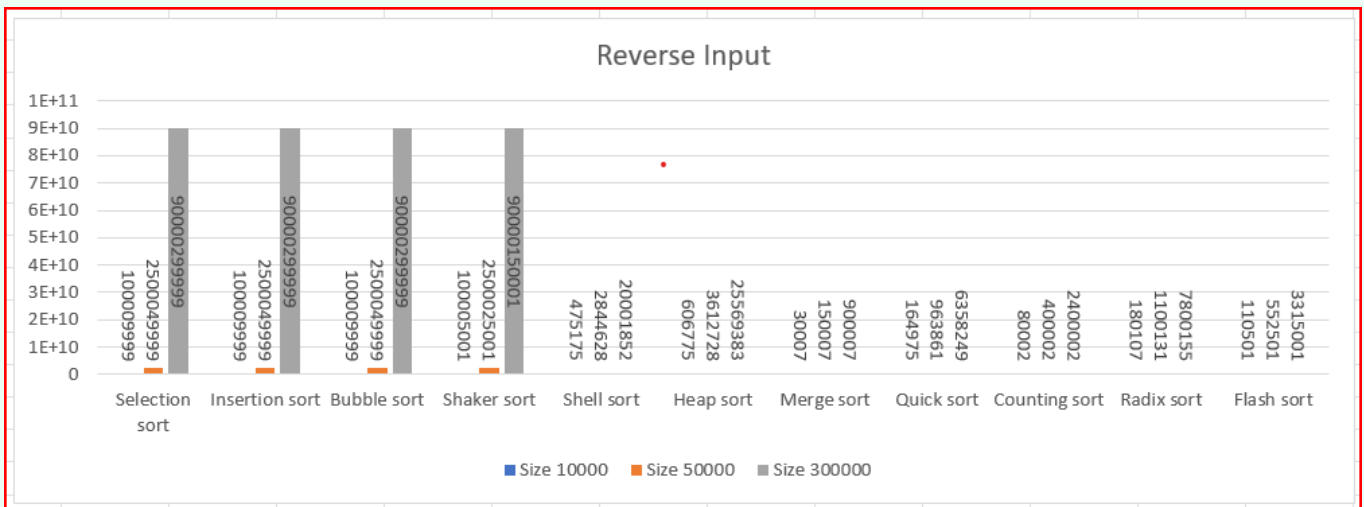
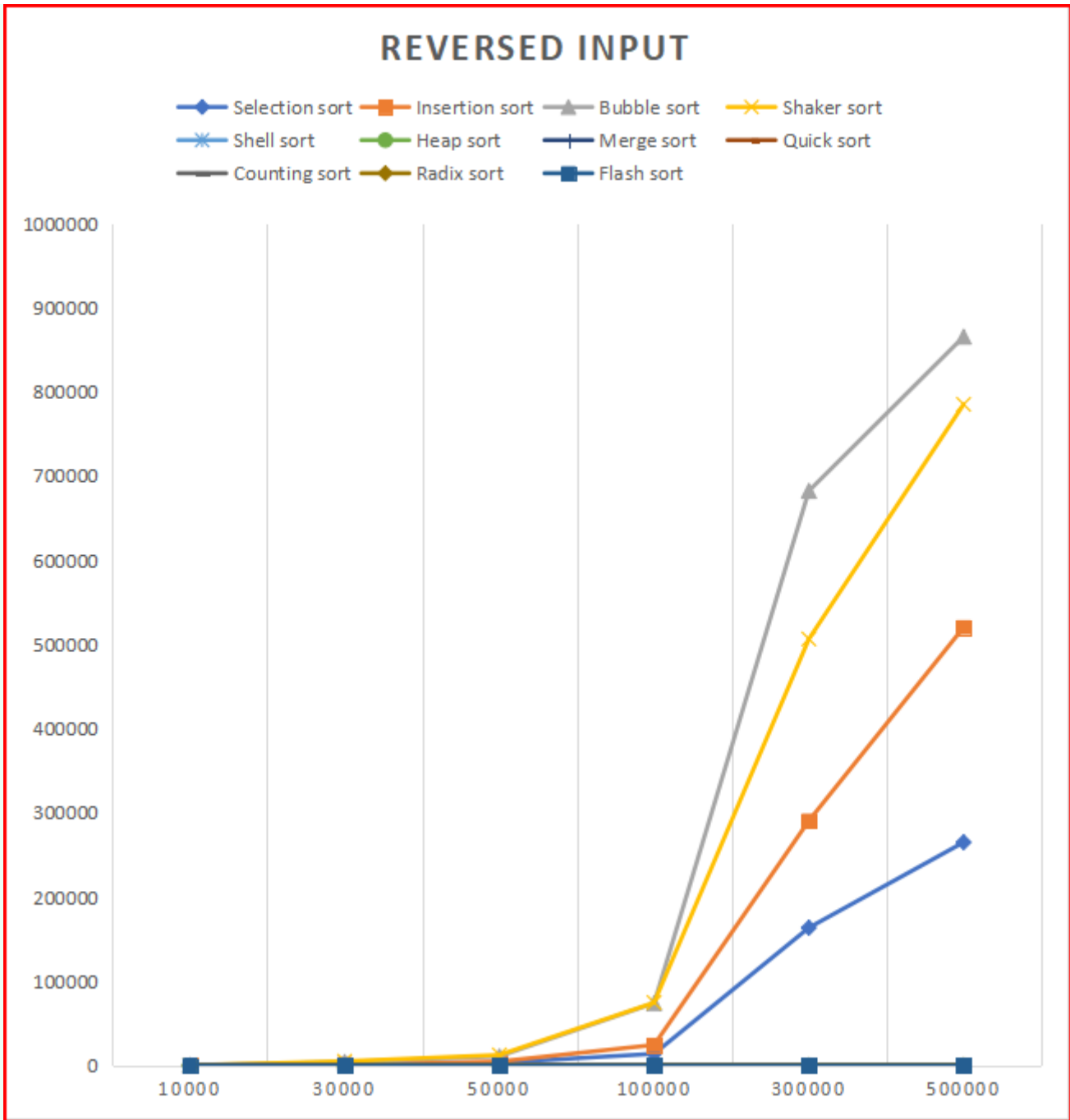
- + Hai thuật toán Bubble và Selection có tốc độ tăng thời gian chạy lớn khi lượng dữ liệu tăng từ 100 000 lên 500 000.
- + Shaker và Insertion trong trường hợp này đạt độ phức tạp thời gian tốt nhất là $O(n)$ vì dữ liệu đã được sắp xếp.
- + Các thuật toán còn lại có thời gian chạy tăng không đáng kể khi lượng dữ liệu thay đổi.

- Về số phép so sánh : (sắp xếp theo hướng số lần so sánh tăng dần)
- + Shaker < Insertion ~ Merge < Counting < Flash < Quick < Radix < Shell < Heap < Bubble ~ Selection

- + Bubble và Selection có số lần so sánh nhiều nhất và tăng rõ rệt khi lượng dữ liệu tăng từ 50 000 lên 300 000
- + Các thuật toán còn lại có số lần so sánh tăng không đáng kể khi lượng dữ liệu thay đổi .

Trường hợp dữ liệu đã đảo ngược





- Về thời gian chạy: (sắp xếp theo hướng thời gian chạy tăng dần)

+ Counting < Quick ~ Flash < Shell < Heap < Merge < Radix < Selection < Insertion



< Shaker ~ Bubble

- + Theo kết quả thực nghiệm, với lượng dữ liệu lên đến 500 000:
 - + Counting Sort có thời gian chạy trong vòng 11ms
 - + Trong khi đó, Bubble Sort có thời gian chạy 865325ms tức ~ 865s ~ 14 phút
- + Thuật toán Bubble có thời gian chạy chậm nhất trong các thuật toán, tiếp đến là Shaker, Insertion, Selection.
- + Các thuật toán còn lại có thời gian chạy tăng không đáng kể khi lượng dữ liệu thay đổi.
- Về số phép so sánh : (sắp xếp theo hướng số lần so sánh tăng dần)
- + Merge < Counting < Flash < Quick < Radix < Shell < Heap < Insertion ~ Shaker ~ Selection ~ Bubble
- + Insertion, Shaker, Selection và Bubble trong trường hợp này có số lần so sánh ngang nhau, nhiều nhất và tăng mạnh khi dữ liệu thay đổi từ 50 000 lên 300 000.

3. Nhận xét chung

*** Về thời gian chạy : - Với kích thước dữ liệu nhỏ (10 000 -> 50 000) nhìn chung tốc độ chênh lệch của các thuật toán không đáng kể, không rõ để phân biệt qua đồ thị. - Trong 4 trường hợp, nhìn chung các thuật toán Counting, Flash và Quick Sort có thời gian chạy tốt nhất. - Bubble và Selection có tốc độ khá chậm trong đa số trường hợp do độ phức tạp luôn là $O(n^2)$. - Các thuật toán Shell, Heap, Merge, Quick có tốc độ ổn định xuyên suốt các trường hợp. - Trong trường hợp dữ liệu gần như được sắp xếp và trường hợp dữ liệu đã được sắp xếp thì Shaker và Insertion Sort có thời gian chạy tốt hơn rất nhiều so với các thuật toán khác. (trường hợp này đạt độ phức tạp $O(n)$)

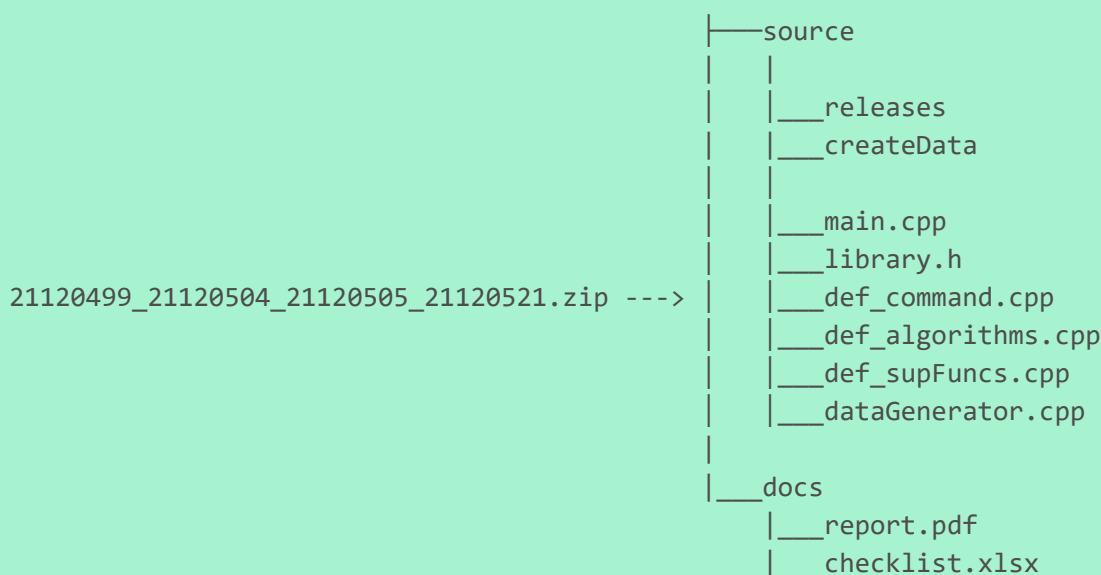
- Phân loại theo độ phức tạp thời gian trung bình:
 - + $O(n^2)$: Bubble, Selection, Insertion, Shaker
 - + $O(n \log n)$: Shell, Heap, Merge, Quick
 - + $O(n)$: Counting, Flash, Radix

*** Về số lần so sánh : - Nhìn chung, trong 4 trường hợp thì Merge, Counting và Flash có số lần so sánh ít nhất. - Bubble và Selection có số lần so sánh nhiều nhất trong cả 4 trường hợp. - Trong trường hợp dữ liệu gần như được sắp xếp và trường hợp dữ liệu đã được sắp xếp thì Shaker và Insertion Sort có số lần so sánh đạt thấp nhất so với các thuật toán khác. - Merge sort có số lần so sánh ổn định và thấp nhất trong đa số trường hợp, bởi vì nó dựa trên mô hình phân chia đệ quy, mỗi lần phân chia đều có số lần so sánh là 1. Trong quá trình merge thì số lần so sánh tăng dần theo số lượng phần tử của mảng con.

V. Tổ chức đồ án và ghi chú



1. Tổ chức đồ án



1. **source** là thư mục chứa mã nguồn của đồ án.

- **releases** là thư mục chứa file thực thi của chương trình **a.exe**.
- **createData** là thư mục chứa file hỗ trợ tạo dữ liệu trong quá trình thống kê dữ liệu **createData.cpp**, và các file dữ liệu được tạo ra.
- **main.cpp** là file chứa hàm **main()**, validate các tham số đầu vào và gọi các hàm thực thi tương ứng với các command.
- **library.h** là file header chứa các khai báo hàm và các thư viện cần thiết, được chia làm 4 phần:
+ I. khai báo các hàm thuật toán sắp xếp tính time và comparisons + II. khai báo các hàm xử lý các lệnh command + III. khai báo các hàm tạo dữ liệu + IV. khai báo các hàm hỗ trợ khác (như đọc file, ghi file, đếm số lượng chữ số của số nguyên, ...)
- **def_command.cpp** là file định nghĩa các hàm command, thực hiện các yêu cầu và xuất kết quả ra file, in ra màn hình console.
- **def_algorithms.cpp** là file định nghĩa các hàm đo thời gian chạy và số lần so sánh của các thuật toán
- **def_supFuncs.cpp** là file định nghĩa các hàm hỗ trợ khác
- **dataGenerator.cpp** là file định nghĩa các hàm tạo dữ liệu

2. **docs** là thư mục chứa các tài liệu liên quan đến đồ án.

- **report.pdf** là file báo cáo đồ án.
- **checklist.xlsx** là file checklist phân chia nhiệm vụ của các thành viên nhóm.

2. Ghi chú



- Thư viện sử dụng : `iostream`, `fstream`, `ctime`, `cstdlib`, `string.h`, `cmath`
- `iostream` : thư viện chuẩn c++ dùng để nhập xuất dữ liệu
- `fstream` : thư viện chuẩn c++ dùng để đọc ghi file
- `ctime` : thư viện chuẩn c++ dùng để đo thời gian chạy thông qua hàm `clock()`
- `cstdlib` : thư viện chuẩn c++ dùng để sử dụng hàm `rand()` và `srand()` để tạo số ngẫu nhiên
- `string.h` : thư viện chuẩn c dùng để xử lý chuỗi ký tự
- `cmath` : thư viện chuẩn c++ dùng để sử dụng hàm `log()`, `pow()`
- Cấu trúc dữ liệu sử dụng : hàng đợi `Queue` hỗ trợ xây dựng thuật toán Radix sort, và mảng kiểu `int` để lưu trữ dữ liệu đầu vào thông qua cấp phát động.

VI. Tài liệu tham khảo

1. selection sort

<https://blog.luyencode.net/thuat-toan-sap-xep-selection-sort/>

2. insertion sort

<https://cafedev.vn/-thuat-toan-insertion-sort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh/>

3. shaker sort

<https://www.geeksforgeeks.org/cocktail-sort/>

4. shell sort

<https://www.geeksforgeeks.org/shellsort/>

5. heap sort

<https://www.geeksforgeeks.org/heap-sort/>

6. merge sort

<https://blog.luyencode.net/thuat-toan-sap-xep-merge-sort/>

7. quick sort

<https://www.geeksforgeeks.org/quick-sort/>

8. counting sort

<https://blog.luyencode.net/counting-sort/>

9. radix sort

<https://www.geeksforgeeks.org/radix-sort/>

10.1 flash sort

<https://www.ddth.com/showthread.php/64851-Flash-sort/>

10.2 flash sort

<https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh>

11. bubble sort

<https://cafedev.vn/thuat-toan-bubble-sort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh/>



12. radix sort

<https://www.iostream.vn/giai-thuat-lap-trinh/distribution-sort-radix-sort-vqu1H1>

13. Complexity

<https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot-nhat>

