ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN


## NHÓM 06

# BÁO CÁO ĐỒ ÁN

## KHẢO SÁT CÁC THUẬT TOÁN SẮP XẾP

**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT - 21CTT3B**

Hồ Chí Minh, ngày 31 tháng 10 năm 2022

# THÔNG TIN ĐỒ ÁN

**GIẢNG VIÊN HƯỚNG DẪN:** Phan Thị Phương Uyên

**THÔNG TIN NHÓM 06 - 21CTT3B**

| MSSV | Họ và tên |
|---|---|
| 21120272 | Huỳnh Cao Khôi |
| 21120295 | Nguyễn Hữu Nghĩa |
| 21120297 | Phùng Lê Hoàng Ngọc |
| 21120308 | Phạm Lê Tú Nhi |

**THỜI HẠN DỰ ÁN:** 24/10/2022 - 5/11/2022

**LỜI CẢM ƠN:**

Chúng em xin gửi lời cảm ơn tới thầy Văn Chí Nam vì đã hướng đãn tận tình và đưa lại kiến thức cần thiết cũng như làm nguồn cảm hứng để học hỏi thêm về cấu trúc dữ liệu và giải thuật nói riêng và về ngành công nghệ thông tin nói chung. Chúng em cũng xin gửi lời cảm ơn cô Phan Thị Phương Uyên vì những hướng dẫn kĩ lưỡng và những lời khuyên của cô trong quá trình học.

# Mục lục

# 1 GIỚI THIỆU

Thuật toán sắp xếp là một thuật toán phổ biến, có tính ứng dụng cao trong cuộc sống. Có thể được hiểu đơn giản đây là quá trình đưa các phần tử trong một dãy bất kì về đúng thứ tự của nó. Tuy nhiên, chừng đó khái niệm vẫn còn quá mơ hồ. Vì vậy, đồ án được thực hiện với mục tiêu có được cái nhìn tổng quát về các thuật toán sắp xếp, nắm rõ nguyên lý hoạt động, các đặc trưng, tính chất, điểm ưu việt và hạn chế cùng với tính ứng dụng của một số thuật toán thông dụng.

Đồ án tìm hiểu 11 thuật toán sắp xếp sau: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, và Flash Sort.

Để có được những đánh giá tốt nhất về các thuật toán, nhóm chú trọng vào các yếu tố như độ phức tạp về mặt thời gian và không gian, thời gian thực hiện cũng như số lượng phép so sánh đối với từng kích thước cũng như thứ tự dữ liệu đầu vào khác nhau. Sau đó so sánh sự tương quan giữa các yếu tố để rút ra câu trả lời cho các câu hỏi như: Thuật toán nhanh nhất là gì? Thuật toán nào có tính ứng dụng cao nhất? Mỗi thuật toán sẽ hoạt động tốt nhất với loại dữ liệu nào, v.v

# 2 TRÌNH BÀY THUẬT TOÁN

## 2.1 Selection Sort

### 2.1.1 Ý tưởng

Selection sort là thuật toán sắp xếp bằng cách liên tục tìm phần tử nhỏ nhất (nếu như xắp xếp mảng theo thứ tự tăng dần) trong dãy và đưa nó lên đầu.

Trong mảng cần sắp xếp sẽ dược ra làm 2 phần, phần đầu là phần mảng được sắp xếp rồi, còn phần sau là phần chưa được sắp xếp. Khi tìm phần tử nhỏ nhất thì ta tìm trong phần mảng chưa được sắp xếp và đưa nó lên đầu phần mảng chưa dược sắp xếp. Phần tử đó sau đó sẽ trở thành phần tử trong phần mảng được sắp xếp. Ta cứ lặp lại như vậy đến khi toàn bộ phần tử thuộc phần mảng được sắp xếp.

### 2.1.2 Trình bày thuật toán

**Algorithm** Selection sort(a[], n)

$i \leftarrow 0$
**while** $i < n$ **do**
    Find $a[j]$. $j$ is the position of the smallest value from i to n
    **if** $i \neq j$ **then** ▷ //In case a[i] is the smallest value form i to n
        Swap $a[i]$ with $a[j]$
    **end if**
    $i \leftarrow i + 1$
**end while**

### 2.1.3 Minh họa thuật toán

Ta có mảng ban đầu là a[] = 8, 6, 3, 4, 9, 5 với số phần tử $n = 6$. $i$ được coi là vị trí của phần tử đang xét. Phần tử đầu được coi là phần tử ở vị trí $i$. Ta sẽ sắp xếp theo chiều tăng dần:

8 6 3 4 9 5

Xét $i = 0$, ta thấy từ 0 đến $n - 1 = 5$ phần tử nhỏ nhất là 3, ta hoán vị 3 với phần tử đầu là 8

3 6 8 4 9 5

Xét $i = 1$, ta thấy từ 1 đến 5 phần tử nhỏ nhất là 4, ta hoán vị 4 với phần tử đầu là 6

3 4 8 6 9 5

Xét $i = 2$, ta thấy từ 2 đến 5 phần tử nhỏ nhất là 5, ta hoán vị 5 với phần tử đầu là 8

3 4 5 6 9 8

Xét $i = 3$, ta thấy từ 3 đến 5 phần tử nhỏ nhất là 6, mà 6 ở vị trí đầu rồi nên ta không cần hoán vị

3 4 5 6 9 8

Xét $i = 4$, ta thấy từ 4 đến 5 phần tử nhỏ nhất là 8, ta hoán vị 8 với phần tử đầu là 9

3 4 5 6 8 9

Xét $i = 5$, phần mảng chưa sắp xếp chỉ còn 1 phần tử nên ta kết thúc quá trình sắp xếp. Kết quả cuối cùng là:

3 4 5 6 8 9

### 2.1.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Không |

**Phân tích độ phức tạp thời gian và không gian**

- Do việc xét phần tử và việc tìm phần tử bé nhất luôn phải chạy qua hết tất cả tất cả các phần tử còn cần sắp xếp là Selection Sort không có trường hợp tốt hay xấu nhất.
- Xét mỗi phần tử của mảng có độ phức tạp thời gian: $O(n)$
- Tìm phần tử nhỏ nhất của phần mảng chưa được sắp xếp có độ phức tạp thời gian: $O(n)$
- Độ phức tạp thời gian của Selection Sort là: $O(n) * O(n) = O(n^2)$
- Do thuật toán chỉ hoán đổi các giá trị trong mảng nên độ phức tạp không gian chỉ là $O(1)$.

## 2.2 Insertion Sort

### 2.2.1 Ý tưởng

Insertion Sort là một thuật toán sắp xếp vô cùng tự nhiên, như cách mà chúng ta sắp xếp các quân bài.

Mảng cần sắp xếp được chia thành 2 phần, phần đã sắp xếp và một phần chưa sắp xếp. Ta chọn các giá trị từ phần chưa sắp xếp và đưa nó về đúng vị trí trong phần đã sắp xếp

### 2.2.2 Trình bày thuật toán

**Algorithm** Insertion Sort (a[], n)

```
i ← 1
j
key
while i < n do
    key ← a[i]
    j ← i − 1
    while j ≥ 0  &  key < a[j] do
        a[j + 1] ← a[j]
        j ← j − 1
    end while
    a[j + 1] ← key
    i ← i + 1
end while
```

Để sắp xếp một mảng theo thứ tự tăng dần, ta thực hiện các bước sau:

- Duyệt mảng bắt đầu từ phần tử thứ 1 tới phần tử thứ n.
- So sánh phần tử hiện tại (key) và phần tử đứng trước nó.
- Di chuyển những phần tử lớn hơn key lên vị trí lớn hơn vị trí hiện tại của nó 1 đơn vị. Chèn key vào vị trí thích hợp

### 2.2.3 Minh họa thuật toán

Cho một mảng chưa sắp xếp a[ ] = 7, 8, 5, 2, 4, 6. Dùng thuật toán Insertion Sort để sắp xếp mảng tăng dần.

***Chú thích:*** *Các số màu xanh là các số đang xét (gọi là key), mục tiêu là chèn key vào vị trí thích hợp*

$i = 1$, $j = 0$, $key = a[i] = 8$

7 8 5 2 4 6

$key = 5$, so sánh với 8 và 7 thì nhỏ hơn, dịch chuyển 8, 7 lần lượt lên một ô và chèn 5 vào vị trí thích hợp. Tăng i lên 1 và duyệt tiếp.

5 7 8 2 4 6

$key = 2$, nhỏ hơn 8, 7, 5. Đẩy các phần tử trước lên 1 ô và chèn 2 vào vị trí thích hợp, tăng i lên 1

2 5 7 8 4 6

$key = 4$, nhỏ hơn 8, 7, 5. Đẩy các phần tử trước lên 1 ô và chèn 4 vào vị trí thích hợp, tăng i lên 1

2 4 5 7 8 6

$key = 6$, nhỏ hơn 8, 7. Đẩy các phần tử trước lên 1 ô và chèn 6 vào vị trí thích hợp, tăng i lên 1

2 4 5 6 7 8

$i = 6 > n$. Vòng lặp kết thúc, ta được mảng kết quả:

2 4 5 6 7 8

### 2.2.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Có |

**Phân tích độ phức tạp thời gian**

- Trường hợp tốt nhất: Mảng đã được sắp xếp
- Trường hợp xấu nhất: Mảng sắp xếp ngược

**Phân tích độ phức tạp không gian**

- Insertion Sort không tốn thêm bộ nhớ phụ để lưu trữ, do đó độ phức tạp là O(1)

### 2.2.5 Cải tiến

Một cách cải tiến Insertion Sort đó là Binary Insertion Sort. Thay vì sử dụng phương pháp tìm kiếm tuyến tính (Linear search) để tìm kiếm vị trí cần chèn , ta có thể sử dụng phương pháp tìm kiếm nhị phân, qua đó làm giảm số lượng các phép so sánh đi, làm giảm thời gian chạy.

Tuy độ phức tạp về thời gian trong trường hợp xấu nhất vẫn là $(On^2)$ nhưng việc tìm kiếm nhị phân đã chuyển độ phức tạp thời gian khi chèn phần tử từ $O(n)$ thành $O(logn)$.

## 2.3 Bubble Sort

### 2.3.1 Ý tưởng

Bubble sort là 1 trong những thuật toán đơn giản nhất. Nó sắp xếp bằng cách hoán đổi vị trí 2 phần tử gần nhau nếu như 2 phần tử không theo thứ tự khi xét các phần tử trong mảng và lặp lại việc hoán đổi như vậy đến khi mảng được sắp xếp.

Với 1 mảng có phần tử nhỏ nhất ở cuối mảng thì ta liên tục hoán đổi vị trí của nó n - 1 lần với n là số phần tử của mảng để có thể đưa phần tử đó lên đầu (nếu như sắp xếp tăng dần). Từ đó suy ra ta cần nhiều nhất n - 1 lần lặp để chắc chắn việc mảng sẽ được sắp xếp.

### 2.3.2 Trình bày thuật toán

**Algorithm** Bubble sort(a[], n)

```
i ← 0
while i < n − 1 do
    j ← 1
    while j < n do
        if a[j] < a[j − 1] then
            Swap a[j − 1] with a[j]
        end if
        j ← j + 1
    end while
    i ← i + 1
end while
```

### 2.3.3 Minh họa thuật toán

Ta có mảng ban đầu là a[] = 8, 6, 3, 4, 9, 5, 1 với số phần tử $n = 7$. Sắp xếp theo thứ tự tăng dần:

| 8 | 6 | 3 | 4 | 9 | 5 | 1 |
|---|---|---|---|---|---|---|

Ta xét chi tiết lần chạy việc hoán vị đầu tiên, những lần sau tương tự như vậy

- Xét $i$ là vị trí phần tử đang xét. 2 phần tử gần nhau để so sánh là phần thử thứ $i$ và phần tử trước nó. Với $i = 1$, ta thấy 8 và 6 sai thứ tự, hoán vị phần tử 6 và 8.

$$6 \quad 8 \quad 3 \quad 4 \quad 9 \quad 5 \quad 1$$

- Xét $i = 2$, ta thấy 3 và 8 sai thứ tự, hoán vị phần tử 3 và 8

$$6 \quad 3 \quad 8 \quad 4 \quad 9 \quad 5 \quad 1$$

- Xét $i = 3$, ta thấy 4 và 8 sai thứ tự, hoán vị phần tử 4 và 8

$$6 \quad 3 \quad 4 \quad 8 \quad 9 \quad 5 \quad 1$$

- Xét $i = 4$, ta thấy 9 và 8 đúng thứ tự, không cần hoán vị

$$6 \quad 3 \quad 4 \quad 8 \quad 9 \quad 5 \quad 1$$

- Xét $i = 5$, ta thấy 5 và 9 sai thứ tự, hoán vị 5 và 9

$$6 \quad 3 \quad 4 \quad 8 \quad 5 \quad 9 \quad 1$$

- Xét $i = 6$, ta thấy 1 và 9 sai thứ tự, hoán vị 1 và 9

$$6 \quad 3 \quad 4 \quad 8 \quad 5 \quad 1 \quad 9$$

Sau lần hoán vị đầu tiên, ta có mảng:

$$6 \quad 3 \quad 4 \quad 8 \quad 5 \quad 1 \quad 9$$

Tương tự, sau lần hoán vị thứ 2:

$$3 \quad 4 \quad 6 \quad 5 \quad 1 \quad 8 \quad 9$$

Sau lần hoán vị thứ 3:

$$3 \quad 4 \quad 5 \quad 1 \quad 6 \quad 8 \quad 9$$

Sau lần hoán vị thứ 4:

$$3 \quad 4 \quad 1 \quad 5 \quad 6 \quad 8 \quad 9$$

Sau lần hoán vị thứ 5:

$$3 \quad 1 \quad 4 \quad 5 \quad 6 \quad 8 \quad 9$$

Sau lần hoán vị thứ $n - 1 = 6$:

$$1 \quad 3 \quad 4 \quad 5 \quad 6 \quad 8 \quad 9$$

Chú thích

- Các phần tử màu xanh là phần tử ở vị trí $i$ trước khi hoán đổi.
- Các phần tử màu đỏ là phần tử mới ở vị trí $i$ nếu có hoán đổi.

### 2.3.4 Độ phức tạp

| **Thuật toán** | **Tốt nhất** | **Xấu nhất** | **Trung bình** | **Không gian** | **Là Stable** |
|---|---|---|---|---|---|
| Bubble Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Có |

**Phân tích độ phức tạp thời gian và không gian**

- Vì số lần hoán vị giới hạn bởi 1 hằng số (số phần tử của mảng - 1) nên trường hợp tốt nhất, trung bình, và xấu nhất đều có độ phức tạp thời gian như nhau
- Vì Bubble sort chỉ hoán vị các phẩn tử trong mảng nên độ phức tạp không gian là $O(1)$

### 2.3.5 Cải tiến

Việc lặp n - 1 lần không phải lúc nào cũng cần thiết vì trường hợp đó chỉ phải lặp khi có 1 phần tử nhỏ nhất ở cuối mảng (nếu như sắp xếp tăng dần) nên thay vĩ lặp n

- 1 lần ta có thể kiểm tra xem nếu như mảng được sắp xếp rồi thì dừng vòng lặp. Để kiểm tra thì ta cần xem sau khi xét hết phần tử trong mảng thì có phần tử nào cần sắp xếp không, nếu không thì mảng đã được sắp xếp.

Việc cải tiến này sẽ khiến trường hợp tốt nhất có độ phức tạp thời gian là $O(n)$ thay vì $O(n^2)$ với trường hợp mảng đã được sắp xếp. Với trường hợp mảng đã gần được sắp xếp thì việc cải tiến sẽ giúp thuật toán chạy nhanh hơn.

## 2.4 Shaker Sort

### 2.4.1 Ý tưởng

Shaker sort( hay còn gọi là Cocktail sort, Shuttle sort) có thể được xem là một cải tiến của Bubble sort.

Ý tưởng cơ bản của Shaker sort cũng tương tự như Bubble sort, nhưng nếu Bubble sort chỉ duyệt các phần tử lần lượt từ trái sang phải và đưa phần tử lớn nhất về cuối danh sách thì Shaker sort sẽ duyệt thêm chiều từ phải sang trái, trong quá trình đó đưa phần tử nhỏ nhất về đầu danh sách. Như vậy, trong một lần duyệt, Shaker sort có thể đưa ít nhất hai phần tử về đúng với vị trí của nó.

Thông qua đó, Shaker sort sẽ khắc phục những nhược điểm của Bubble sort bằng việc hạn chế được những phép so sánh không cần thiết, đồng thời giảm độ lớn của danh sách cho những lần lặp tiếp theo.

### 2.4.2 Trình bày thuật toán

**Algorithm** Shaker Sort (a[], n)

$left \leftarrow 0$
$right \leftarrow n - 1$
$k \leftarrow 0$
**while** $left < right$ **do**
  **for** *each element* $a[i]$ *in* $a$ *from* $left$ *to* $right$ **do**
    **if** $a[i] > a[i + 1]$ **then**
      Swap a[i] and a[i + 1]
      Store the position i by k, $k \leftarrow i$
    **end if**
  **end for**
  $right \leftarrow k$
  **for** *each element* $a[i]$ *in* $a$ *from* $right$ *to* $left$ **do**
    **if** $a[i] < a[i - 1]$ **then**
      Swap a[i] and a[i - 1]
      Store the position i by k, $k \leftarrow i$
    **end if**
  **end for**
  $left \leftarrow k$
**end while**

### 2.4.3 Ví dụ thuật toán

Ta có mảng chưa sắp xếp a[] = 4, 11, 23, 5, 16, 10. Dùng thuật toán Shaker Sort để sắp xếp mảng tăng dần.

Left = 0

Right = 6 - 1 = 5

Khởi tạo một biến k = 0, đây là biến dùng để lưu lại giá trị của vị trí hoán đổi cuối cùng, nhằm mục đích thu hẹp phạm vi của mảng và bỏ qua những đoạn đã được sắp xếp thứ tự.

**Lần 1:**

**Lần 1.1:** Duyệt lần lượt từ vị trí left sang vị trí right, nếu phần tử đang xét lớn hơn phần tử đứng liền sau nó thì đổi chỗ hai phần tử, đồng thời lưu vị trí của phần tử đó vào biến k

i = left = 0

**4** **11** 23 5 16 10 → **4** **11** 23 5 16 10

i = 1

4 **11** **23** 5 16 10 → 4 **11** **23** 5 16 10

i = 2

4 11 **23** **5** 16 10 → 4 11 **5** **23** 16 10

Swap 5 and 23

k = i = 2

i = 3

4 11 5 **23** **16** 10 → 4 11 5 **16** **23** 10

Swap 23 and 16

k = i = 3

i = 4

4 11 5 16 **23** **10** → 4 11 5 16 **10** **23**

Swap 23 and 10

k = i = 4

Lúc này, ta đã đưa phần tử lớn nhất mảng là 24 về cuối mảng, đồng thời biến k chứa vị trí của phần tử được hoán vị sau cùng là 4.

Right = k = 4

**Lần 1.2:** Duyệt lần lượt từ vị trí right - 1 về left, nếu phần tử đang xét lớn hơn phần tử đứng liền sau nó thì đổi chỗ hai phần tử, đồng thời lưu vị trí của phần tử đó

vào biến k

i = right = 4

4 11 5 **16** **10** 23 → 4 11 5 **10** **16** 23

Swap 16 and 23

k = i = 4

i = 3

4 11 **5** **10** 16 23 → 4 11 **5** **10** 16 23

i = 2

4 **11** **5** 10 16 23 → 4 **5** **11** 10 16 23

Swap 11 and 5

k = i = 2

i = 1

**4** **5** 11 10 16 23 → **4** **5** 11 10 16 23

Lúc này, phần tử 4 đã được đưa về đầu mảng, đồng thời giá trị k = 2

left = k = 2

Sau đó thực hiện các bước tương tự cho những lần lặp sau

**Lần 2:**

**Lần 2.1:**

i = left = 2

4 5 **11** **10** 16 23 → 4 5 **10** **11** 16 23

Swap 11 and 10

k = i = 2

i = 3

4 5 10 **11** **16** 23 → 4 5 10 **11** **16** 23

right = k = 2

**Lần 2.2:**

i = right = 2

4 **5** **10** 11 16 23 → 4 **5** **10** 11 16 23

left = k = 2

left = 2 = right

Kết thúc thuật toán.

Sau khi thực hiện xong thuật toán Shaker sort, mảng đã được sắp xếp là

4 5 10 11 16 23

Chú thích

- Các phần tử được **in đậm** thể hiện đó là phần tử đang được xét đến trong quá trình lặp.
- Các phần tử có màu xanh là những phần tử bị hoán đổi vị trí với nhau trong quá trình lặp.

### 2.4.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Shaker Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Có |

**Phân tích thời gian**

- Trường hợp tốt nhất: Xảy ra khi danh sách đã được sắp xếp, lúc này trong lần duyệt đầu tiên, sau khi trải qua hai vòng lặp 1.1 và 1.2, do không có phần tử nào được hoán đổi, cả hai biến left vầ right đều = 0, thuật toán kết thúc. Do đó độ phức tạp thời gian lúc này là O(n)

- Trường hợp xấu nhất: Xảy ra khi danh sách được sắp xếp theo chiều ngược lại. Lúc này cần duyệt qua lần lượt tất cả các phần tử trong mảng theo cả hai chiều và tương tự như Bubble sort, độ phức tạp về thời gian lúc này lên đến $O(n^2)$

**Phân tích không gian**

- Shaker Sort là một thuật toán In-place, không đòi hỏi thêm không gian phụ nên độ phức tạp về không gian là O(1)

## 2.5 Shell Sort

### 2.5.1 Ý tưởng

Shell Sort là một thuật toán cái biến thể của Insertion Sort. Với Shell Sort, các phần tử được sắp xếp theo từng lớp danh sách nhỏ hơn, với khoảng cách của các phần tử là $h$ giảm dần, và danh sách sẽ được sắp xếp sau khi hoàn thành sắp xếp lớp khoảng cách $h = 1$.

Vấn đề trong tâm của bài toán Shell Sort là việc sắp xếp mảng theo từng lớp với khoảng cách $h$. Ví dụ một danh sách có $n$ phần tử và khoảng cách $h = \frac{n}{2}$ thì khi danh sách đó được sắp xếp theo khoảng cách $h = \frac{n}{2}$, mỗi phần tử bất kì với phần tử cách nó $h$ khoảng cách sẽ có trật tự (lớn hơn hoặc nhỏ hơn). Việc sắp xếp các phần tử ở lớp này tạo ra trật tự trong mảng nhanh chóng và làm giảm khối lượng công việc cho các vòng lặp sau với lớp danh sách h nhỏ hơn.

### 2.5.2 Trình bày thuật toán

Thuật toán dưới đây được trình bày theo ý tưởng của tác giả Shell Sort - Donald Shell - lấy khoảng cách $h = \frac{n}{2^i}$ với $n$ là số phần tử trong mảng và $h$ là khoảng cách của các phần tử trong mảng mỗi lớp danh sách và $i$ đi từ 1 đến $i'$ sao cho $\frac{n}{2^{i'}} = 1$

**Algorithm** Shell Sort (a[], n)

**for** $gap \leftarrow \frac{n}{2}$ down to 1 with step $gap \leftarrow \frac{gap}{2}$ **do**
  **for** $i \leftarrow gap$ to $n$ with step $i \leftarrow i + 1$ **do**
    **for** each $a[i]$ in list **do**
      Move $a[i]$ to $j$ where $a[j]$ *is the smallest element where* $(i - j) \vdots gap > a[i]$
    **end for**
  **end for**
**end for**

### 2.5.3 Minh họa thuật toán

Chúng ta có một mảng a với 6 phần tử như sau:

| 5 | 6 | 9 | 0 | 3 | 7 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

Thực hiện sắp xếp các phần tử của mảng theo thứ tự tăng dần

**Với** $Gap \leftarrow \frac{6}{2} = 3$

$i = 3$

| 5 | 6 | 9 | 0 | 3 | 7 | → | 0 | 6 | 9 | 5 | 3 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 1 | 2 | 3 | 4 | 5 |

$i = 4$

| 0 | 6 | 9 | 5 | 3 | 7 | → | 0 | 3 | 9 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 1 | 2 | 3 | 4 | 5 |

$i = 5$

| 0 | 3 | 9 | 5 | 6 | 7 | → | 0 | 3 | 7 | 5 | 6 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 1 | 2 | 3 | 4 | 5 |

**Với** $Gap \leftarrow \frac{3}{2} = 1$

$i = 1$

| 0 | 3 | 7 | 5 | 6 | 9 | → | 0 | 3 | 7 | 5 | 6 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 1 | 2 | 3 | 4 | 5 |

$i = 2$

| 0 | 3 | 7 | 5 | 6 | 9 | → | 0 | 3 | 7 | 5 | 6 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 1 | 2 | 3 | 4 | 5 |

$i = 3$

| 0 | 3 | 7 | 5 | 6 | 9 | → | 0 | 3 | 5 | 7 | 6 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 1 | 2 | 3 | 4 | 5 |

$i = 4$

| 0 | 3 | 5 | 7 | 6 | 9 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

$\rightarrow$

| 0 | 3 | 5 | 6 | 7 | 9 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

$i = 5$

| 0 | 3 | 5 | 6 | 7 | 9 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

$\rightarrow$

| 0 | 3 | 5 | 6 | 7 | 9 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

Hoàn thành sắp xếp danh sách theo thứ tự tăng dần.

Chú thích

- Các phần tử được gạch chân là các phần tử có cùng một lớp khoảng cách - có thể bị di chuyển trong 1 lần lặp.
- Các phần tử có màu xanh là các phần tử bị thay đổi vị trí trong quá trình lặp.

### 2.5.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Shell Sort | $O(n * log(n))$ | $O(n^2)$ | $O(n^{\frac{4}{3}})$ | $O(1)$ | Không |

**Phân tích thời gian**

- Trường hợp tốt nhất: Xảy ra khi mảng đã được sắp xếp và thuật toán lặp mỗi lớp phần tử 1 lần
- Trường hợp xấu nhất: Xảy ra khi mảng có số phần tử n với biểu diễn nhị phân của nó có nhiều số 0 liền nhau. Ví dụ khi n là một lũy thừa của 2 và phần tử lớn và nhỏ hơn trung vị lần lượt nằm ở các vị trí chẵn và lẻ.

**Phân tích không gian**

- Shell Sort là một thuật toán In-place, tức là không dùng không gian (danh sách) phụ, nên độ phức tạp không gian là $O(1)$

### 2.5.5 Cải tiến và các biến thể

Một trong những vấn đề lớn trong việc cài đặt thuật toán Shell Sort là việc lựa chọn khoảng cách $h$ cho thuật toán để đạt hiệu quả cao nhất. Đã có nhiều nghiên cứu được đưa ra nhằm tính toán độ phức tạp cho các dãy khoảng cách $h$ khác với dãy $h = \frac{n}{2}, \frac{n}{4}, ..., 1$ được cài đặt lần đầu bởi Donald Shell.

#### Cải tiến của Hibbard

Một trong những cải tiến của Shell Sort được thực hiện bởi Thomas N. Hibbard. Dãy khoảng cách được cài đặt là dãy $h = 1, 3, 7, 15, 31, ...$ với công thức $h_t = 2^t - 1$ với $t = 1, 2, 3....$

Độ phức tạp thời gian tốt nhất của Shell Sort theo dãy Hibbard là $O(n * log(n))$ và độ phức tạp thời gian tệ nhất đã được cải thiện là $O(n^{\frac{3}{2}})$

#### Cải tiến của Sedgewwick

Nhiều dãy khác được đề xuất bởi Robert Sedgewwick, trong đó có dãy $h = 1, 4, 13, 40, ...$ với công thức $h_{k+1} = 3 * h_k + 1$ với $h_1 = 1$.

Độ phức tạp thời gian của dãy trên được chứng minh là $O(n^{\frac{4}{3}})$, là một cải thiện với dãy ban đầu của Shell và Hibbard.

## 2.6 Heap Sort

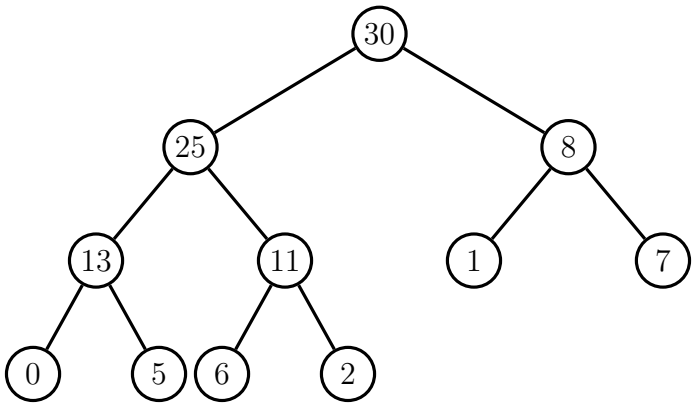
### 2.6.1 Ý tưởng

Heap Sort (Sắp xếp vun đống) là một thuật toán sắp xếp dựa trên một cấu trúc dữ liệu đặc biệt gọi là Heap (Đống).

Một Max-heap là dãy n phần tử sao cho với mỗi phần tử (tại vị trí thứ i) trong nửa đầu của dãy luôn lớn hơn hoặc bằng phần tử tại vị trí thứ $2i + 1$ và $2i + 2$. Phần tử đầu tiên của max-heap là cực đại. Tượng tự, cũng có min-heap.

Heap Sort là một thuật toán cải tiến của Selection Sort vì nó cũng sử dụng ý tưởng là "chọn" phần tử cực trị. Đồng thời, cách chọn của Heap sort tận dụng tính chất của cấu trúc dữ liệu heap nên được cải tiến và thời gian chạy tối ưu hơn Selection Sort rất nhiều.

**Ví dụ một max heap:**




**Biểu diễn Max Heap dưới dạng mảng:**

| 30 | 25 | 8 | 13 | 11 | 1 | 7 | 0 | 5 | 6 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|

Heap sort hoạt động theo cách sau:

- Xây dựng dãy cần sắp xếp thành một max-heap (min-heap).
- Dựa theo tính chất của cấu trúc dữ liệu heap (phần tử đứng đầu là phần tử cực trị), ta bỏ đi phần tử đầu tiên (bằng cách hoán đổi nó với phần tử cuối mảng), giảm số lượng phần tử đi 1 và không quan tâm tới nó nữa.

- Lúc này, do đã xóa đi phần tử đầu mảng, ta lại một lần nữa xây lại max heap (min heap).

- Lặp lại quá trình trên cho đến khi dãy ban đầu chỉ còn một phần tử. Kết thúc thuật toán. Ta nhận được một dãy đã sắp xếp.

### 2.6.2 Trình bày thuật toán

Heap Construction là hàm xây dựng một max heap bằng cách xét từ vị trí giữa mảng (node parent cuối cùng) ngược lên đầu mảng, trong quá trình đó, ta xây dựng lại heap ở mỗi vị trí bằng hàm HeapRebuild.

**Algorithm** Heap Construction (a[], n)

```
//Start from the middle of the array (first half)
Initialize index = (n − 1)/2
while index ≥ 0 do
    //Heap Rebuild at the position index
    HeapRebuild(index, a, n)
    index ← index − 1
end while
```

Heap Rebuild là hàm kiểm tra tính đúng đắn của heap, từ đó xây lại một max heap tại vị trí k.

**Algorithm** Heap Rebuild (pos, a[], n)

Initialize $k = pos$, $isHeap = false$
**while** not isHeap and $2k + 1 < n$ **do**
  $j = 2k + 1$ //j is the pos of the first child
  **if** j < n-1 **then** //k has enough 2 children
    **if** $a[j] < a[j + 1]$ **then** //if first child < second child
      $j = j + 1$ //j is now the pos of second child
    **end if**
  **end if**
  **if** $a[k] \geq a[j]$ **then** //if node at k pos is greater than or equal to its greatest child
    isHeap = true
  **else**
    swap a[k] and a[j] //swap node at k pos with its greatest child
    $k = j$ //recheck at the pos j
  **end if**
**end while**

Heap Sort là hàm sort sau khi có một max heap bằng cách xóa lần lượt từng phần tử đầu (đưa về cuối mảng), giảm số phần tử mảng xuống 1 và dựng lại max heap với các phần tử còn lại. Kết quả cho ra một dãy sắp xếp tăng.

**Algorithm** Heap Sort (a[], n)

HeapConstruct(a, n) //Build a max heap from the given array
Initialize $r = n - 1$ //r is the index of the last element after each deletion
**while** $r > 0$ **do**
  swap a[0] and a[r] //delete the root (first element)
  $HeapRebuild(0, a, r)$ //rebuild max heap from the remaining elements
  $r = r - 1$ //decrease size of array after deletion
**end while**

### 2.6.3 Minh họa thuật toán

Cho một mảng chưa sắp xếp a[ ] = 4, 10, 3, 5, 1. Dùng thuật toán Heap Sort để sắp xếp mảng tăng dần.

| 4 | 10 | 3 | 5 | 1 |
|---|---|---|---|---|

***Chú thích:*** *Các số màu xanh là các số tại vị trí đang xét, các số màu đỏ là các node con của chúng. Phần mảng màu tím trong bước 2 là phần mảng đã sắp xếp xong.*

**Bước 1: Xây dựng 1 max heap với mảng đã cho**

Bắt đầu xét từ a[index] = 3. k = 2, không có tồn tại node con.

| 4 | 10 | **3** | 5 | 1 |
|---|---|---|---|---|

Xét node a[k] = 4.

Node con thứ nhất a[j] = 10

Node con thứ hai a[j+1] = 3

a[j] > a[j+1], so với node cha thì lớn hơn, đổi chỗ 4 và 10

Cập nhật lại k = j = 1.

| **10** | **4** | **3** | 5 | 1 |
|---|---|---|---|---|

Xét lại node a[k] = 4.

Node con thứ nhất a[j] = 5

Node con thứ 2 a[j+1] = 1

a[k] < a[j], đổi chỗ 4 và 5

Cập nhật lại k = j = 3

| 10 | **5** | 3 | **4** | **1** |
|---|---|---|---|---|

Xét lại node a[k] = 4.

Không tồn tại node con.

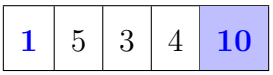
index < 0. Dựng được 1 max heap

| 10 | 5 | 3 | **4** | 1 |
|---|---|---|---|---|

**Bước 2: Xóa lần lượt**

r = 4, đổi chỗ 10 và 1.

Build lại max heap với 1, 5, 3, 4




Xét node a[k] = 1.

Node con thứ nhất a[j] = 5

Node con thứ hai a[j+1] = 3

a[k] < a[j], đổi chỗ 1 và 5.

Cập nhật k = j = 1


Xét node a[k] = 1.

Node con thứ nhất a[j] = 3

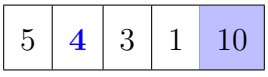
Node con thứ hai a[j+1] = 4

a[k] < node con thứ 2, đổi chỗ 1 và 4.

Cập nhật k = j = 3.

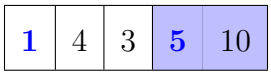

Xét node a[k] = 4.

Không tồn tại node con. Nhận được 1 max heap.




r = r - 1 = 3. Đổi chỗ 5 và 1.

Xây dựng lại max heap gồm 1, 4, 3



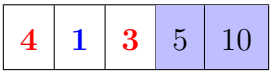

Xét node a[k] = 1.

Node con thứ nhất a[j] = 4

Node con thứ hai a[j+1] = 3

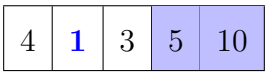
a[k] < a[j], đổi chỗ 1 và 4

Cập nhật k = j = 1.



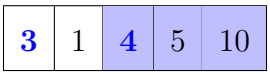

Xét node a[k] = 1.

Không tồn tại node con. Được 1 max heap




r = r - 1 = 2. Đổi chỗ 4 và 3

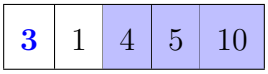
Xây dựng lại max heap gồm 3, 1




Xét node a[k] = 3

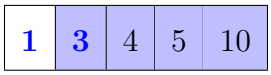
a[j] = 1

a[k] > a[j]. Được một max heap.




r = r - 1 = 1. Đổi chỗ 3 và 1.

Được dãy đã sắp xếp tăng. Kết thúc chương trình




Mảng đã sắp xếp:

### 2.6.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Heap Sort | $O(nlogn)$ | $O(nlogn)$ | $O(nlogn)$ | $O(1)$ | Không |

**Phân tích độ phức tạp thời gian**

- Độ phức tạp về thời gian của Heap Sort trong mọi trường hợp là như nhau.

**Phân tích độ phức tạp không gian**

- Heap Sort không tốn thêm bộ nhớ phụ để lưu trữ, do đó độ phức tạp là O(1)

## 2.7 Merge Sort

### 2.7.1 Ý tưởng

Merge sort là thuật toán dựa theo qui luật chia để trị. Nó chia đôi mảng ra thành 2 mảng nhỏ hơn đến khi không thể chia được. Sau khi chia đôi hết thì lấy 2 mảng con vừa chia xong sẽ gộp lại thành 1 mảng lớn có thứ tự. lặp lại như vậy đến khi số phần tử của mảng gộp lại giống số phần tử mảng ban đầu.

Hàm Merge sort sử dụng đệ qui để thực hiện thao tác chia và gộp lại.

### 2.7.2 Trình bày thuật toán

**Algorithm** Merge sort(a[], left, right)

▷ // left and right are position in the main array to mark the begin and end of a smaller array

**if** $left \geq right$ **then** ▷ // If the smaller array only has 1 or no element then exit

Exit function

**end if**

$mid \leftarrow (left + right)/2$ ▷ // Find the middle position of the smaller array

▷ // Divide the array into 2 smaller arrays, the first one from left to mid, the second one from mid + 1 to right

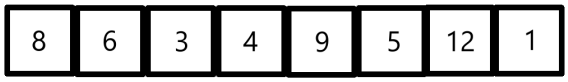
Merge sort(a[], $left$, $mid$) ▷ // Divide the first divided array

Merge sort(a[], $mid + 1$, $right$) ▷ // Divide the second divided array

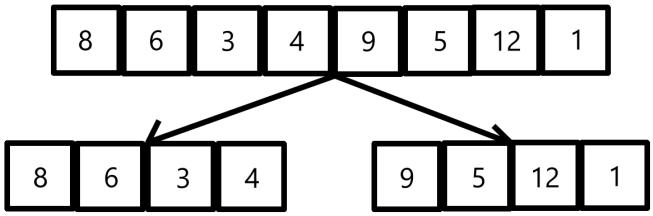
Merge the first and second array together in a sorted order

### 2.7.3 Minh họa thuật toán

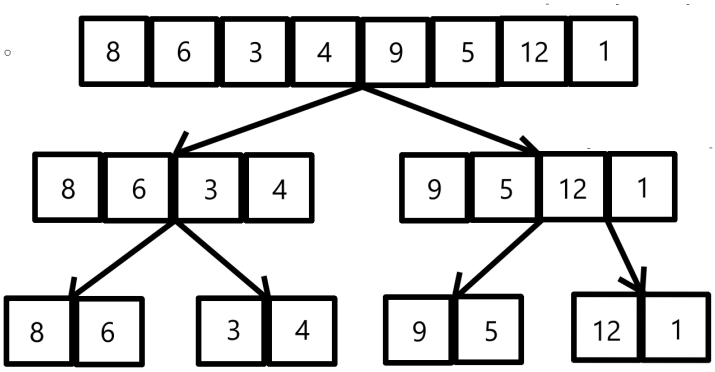
Ta có mảng ban đầu là a[] = 8, 6, 3, 4, 9, 5, 12, 1 với số phần tử $n = 8$. Ta sẽ sắp xếp theo chiều tăng dần:



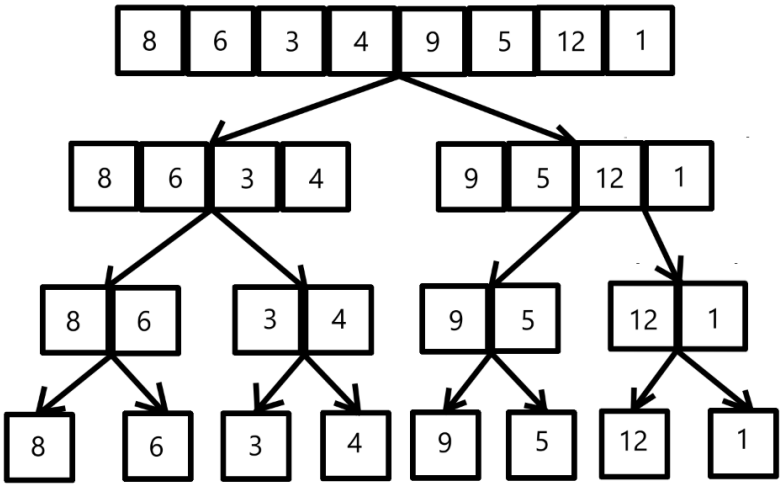

Ta tiến hành chia đôi mảng ra thành 2 phần:



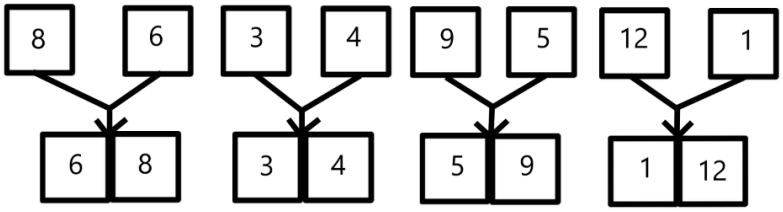

Ta tiếp tục chia đôi 2 mảng con vừa được chia đôi:



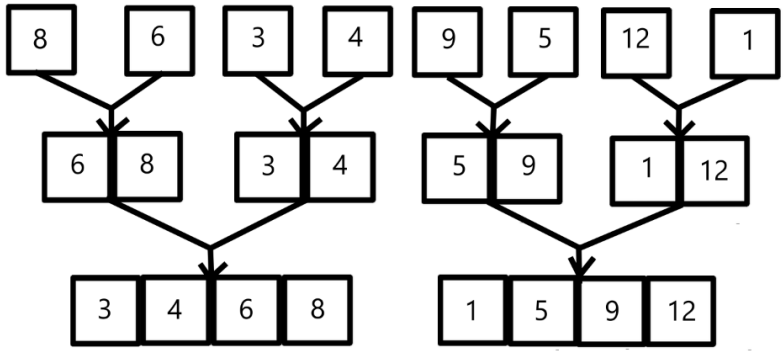

Vì còn mảng con có nhiều hơn 1 phần tử nên ta tiếp tục chia đôi:



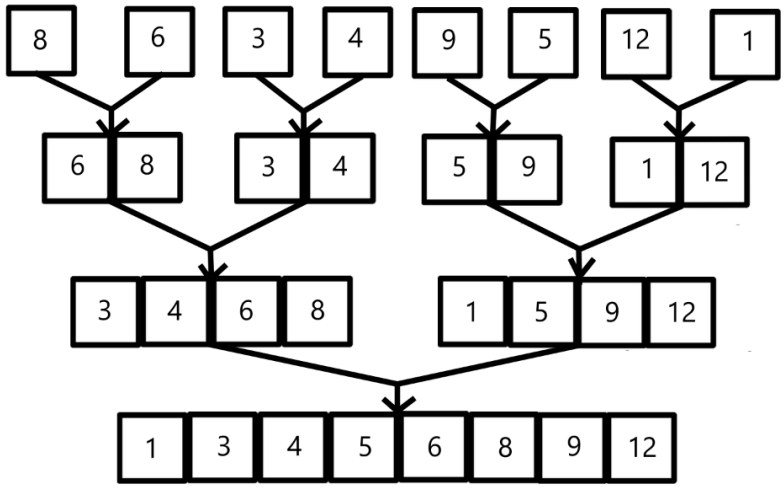

Khi đã chia đôi xong rồi ta bắt đầu gộp lại những mảng con thành mảng mẹ trước đó nhưng sắp xếp lại. Ví dụ ở 2 mảng con {8} và {6} từ mảng mẹ {8, 6} thì ta gộp lại theo thứ tự thì ra một mảng mẹ mới {6, 8}. Tương cho những mảng con kia:



Tiếp tục gộp lại thành những mảng có 4 phần tử:




Gộp lại thành mảng ban đầu nhưng theo thứ tự sắp xếp:




Giải thích chi tiết cách gộp 2 mảng lại thành mảng được sắp xếp:

- Bởi vì hàm Merge sort sử dụng đệ qui nên các mảng con trước ghi gộp lại luôn được sắp xếp. Từ đó có thể dễ dàng tạo ra mảng sắp xếp khi gộp 2 mảng sắp xếp lại
- Ví dụ như ta có 2 mảng {4, 6, 8} và {1, 2, 5}. Ta tạo ra mảng mẹ trống để nhét phần tử vào:

- So sánh phần tử đầu tiên của 2 mảng, ta thấy $1 < 4$ nên nhét 1 vào mảng mẹ:

- Do phần tử đầu tiên ở mảng 2 được chọn nên ta so sánh tiếp phần tử thứ hai của mảng 2 với phần tử đầu tiên của mảng 1, thấy rằng $2 < 4$ nên nhét 2 vào mảng mẹ:

| 1 | 2 | | | | |
|---|---|---|---|---|---|

- Ta so sánh tiếp tục phần tử thứ ba của mảng 2 để với phần tử đầu tiên của mảng 1, thấy rằng $4 < 5$ nên nhét 4 vào mảng mẹ:

| 1 | 2 | 4 | | | |
|---|---|---|---|---|---|

- Vì phần tử đầu tiên của mảng 1 được chọn nên ta so sánh tiếp tục phần tử thứ ba của mảng 2 với phần tử thứ hai của mảng 1, thấy rằng $5 < 6$ nên nhét 5 vào mảng mẹ:

| 1 | 2 | 4 | 5 | | |
|---|---|---|---|---|---|

- Vì mảng 2 không còn phần tử để so sánh nên ta nhét lần lượt các phần tử còn lại của mảng 1 vào mảng mẹ:

| 1 | 2 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|

### 2.7.4 Độ phức tạp

| **Thuật toán** | **Tốt nhất** | **Xấu nhất** | **Trung bình** | **Không gian** | **Là Stable** |
|---|---|---|---|---|---|
| Merge Sort | $O(n * \log n)$ | $O(n * \log n)$ | $O(n * \log n)$ | $O(n)$ | Có |

**Phân tích độ phức tạp thời gian và không gian**

- Vì thuật toán Merge sort chia mảng ra 2 phần nhỏ đến khi mỗi mảng con chỉ còn 1 hoặc 0 phần tử nên ta sẽ có độ phức tạp là $O(\log n)$ (giống như độ phức tạp của thời gian của thuật toán Tìm kiếm nhị phân)

- Khi kết hợp 2 mảng con lại ta phải chạy hết mảng nên ta có độ phức tạp thời gian là: $O(n)$
- Từ đó suy ra độ phức tạp của Merge sort là: $O(n * \log n)$
- Do thuật toán cần tạo mảng phụ để chứa những phần tử từ 2 mảng con rồi mới cho vào mảng mẹ nên độ phức tạp không gian là: $O(n)$

## 2.8 Quick Sort

### 2.8.1 Ý tưởng

Tương tự như Merge Sort, Quick Sort cũng là một thuật toán sắp xếp dựa trên giải thuật "Chia để trị" (Divide and Conquer).

Quick Sort hoạt động dựa trên nguyên tắc chọn một phần tử mốc (pivot) và phân hoạch mảng cần sắp xếp xung quanh phần tử pivot đó.

Có thể chọn pivot là bất kì phần tử nào trong mảng, nhưng ở đây chọn pivot là phần tử ở giữa mảng để trình bày thuật toán.

### 2.8.2 Trình bày thuật toán

Hàm Partition dùng để phân hoạch mảng xung quanh phần tử pivot, chia mảng làm 2 phần, đưa phần tử pivot về đúng vị trí của nó, đưa những phần tử nhỏ hơn pivot về bên trái, lớn hơn pivot về bên phải.

Ở đây chọn pivot là phần tử giữa mảng và sử dụng phân hoạch Hoare (Hoare's Partition) để phân hoạch mảng.

**Algorithm** Partition (a[], first, last)

$first \leftarrow$Starting index
$last \leftarrow$Ending index
Initialize $middle \leftarrow (first + last)/2$
Initialize $pivot \leftarrow a[middle]$
Initialize $i \leftarrow first$
Initialize $j \leftarrow last$
**while** $i \leq j$ **do**
    **if** $a[i] \leq pivot$ **then**
        Increase i by 1
    **end if**
    **if** $a[j] \geq pivot$ **then**
        Decrease j by 1
    **else**
        Swap $a[i]$ and $a[j]$
        $i++$
        $j--$
    **end if**
**end while**
**if** $middle < j$ **then**
    Swap a[middle] and a[j]
    $middle = j$
**end if**
**if** $middle > i$ **then**
    Swap a[middle] and a[i]
    $middle = i$
**end if**
Return middle

**Algorithm** Quick Sort (a[], first, last)

**if** $first \leq last$ **then**

$pi = Partition(a, first, last)$

//Sort the left part:

$Partition(a, first, pi - 1)$

//Sort the right part:

$Partition(a, pi + 1, last)$

**end if**

Để sắp xếp một mảng theo thứ tự tăng dần, cần thực hiện các bước sau:

- Đưa các phần tử nhỏ hơn pivot về bên trái, lớn hơn pivot về bên phải, đưa pivot về đúng vị trí của nó. Lúc này ta nhận được một mảng gồm 2 phần.
- Lặp lại thao tác trên với phần bên trái và phần bên phải.

### 2.8.3 Minh họa thuật toán

Cho một mảng chưa sắp xếp a[ ] = 77, 91, 60, 47, 51. Dùng thuật toán Quick Sort để sắp xếp mảng tăng dần.

| 77 | 91 | 60 | 47 | 51 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

***Chú thích:*** *Các ô màu tím hiển thị phần tử mốc (pivot).*

$i = 0$

$j = 4$

$middle = 2$

$pivot = a[middle] = 60$

| 77 | 91 | 60 | 47 | 51 |
|---|---|---|---|---|
| i | | | | j |

a[i] = 77 > pivot

a[j] = 51 < pivot

Đổi chỗ 77 và 51

Tăng i và j lên 1

| 51 | 91 | 60 | 47 | 77 |
|---|---|---|---|---|
| | i | | j | |

| 51 | 91 | 60 | 47 | 77 |
|---|---|---|---|---|
| | | i, j | | |

| 51 | 47 | 60 | 91 | 77 |
|---|---|---|---|---|
| | | j | i | |

Trả về middle = 2

Sắp xếp phần bên trái từ first = 0 đến last = middle - 1 = 1

| 51 | 47 | 60 | 91 | 77 |
|---|---|---|---|---|
| i | j | | | |

| 51 | 47 | 60 | 91 | 77 |
|---|---|---|---|---|
| | j | i | | |

| 47 | 51 | 60 | 91 | 77 |
|---|---|---|---|---|
| | j | i | | |

Trả về middle = j = 1

Sắp xếp phần bên phải từ first = 3 tới last = 4

| 47 | 51 | 60 | 91 | 77 |
|---|---|---|---|---|
| | | | i | j |

| 47 | 51 | 60 | 91 | 77 |
|---|---|---|---|---|
| | | | | j, i |

| | | | | |
|---|---|---|---|---|
| 47 | 51 | 60 | 77 | 97 |
| | | | | j |

Trả về middle = j = 4

First = 3 = Last. Chương trình kết thúc. Nhận được mảng kết quả là:

47 51 60 91 77

### 2.8.4 Độ phức tạp

| **Thuật toán** | **Tốt nhất** | **Xấu nhất** | **Trung bình** | **Không gian** | **Là Stable** |
|---|---|---|---|---|---|
| Insertion Sort | $O(nlogn)$ | $O(n^2)$ | $O(nlogn)$ | $O(1)$ | Không |

**Phân tích độ phức tạp thời gian**

- Trường hợp tốt nhất: Mỗi lần phân hoạch đều chọn được phần tử pivot là phần tử trung vị (median)
- Trường hợp xấu nhất: Mỗi lần phân hoạch, ta đều chọn phải phần tử cực trị làm pivot

**Phân tích độ phức tạp không gian**

- Insertion Sort không tốn thêm bộ nhớ phụ để lưu trữ, do đó độ phức tạp là O(1)

### 2.8.5 Cải tiến

Một cách để cải thiện Quick Sort đó là làm giảm khả năng rơi vào trường hợp tệ nhất - chọn phải pivot là phần tử cực trị. Bằng cách sử dụng một phương pháp chọn pivot khác gọi là "Median of three" (trung vị của 3 số).

"Median of three" là phương pháp chọn pivot bằng cách lấy ra 3 phần tử ngẫu nhiên trong dãy cần sắp xếp hoặc 3 phần tử: đầu dãy, cuối dãy, giữa dãy. Trong 3 phần tử đó, chọn phần tử trung vị làm pivot. Bằng cách này, ta sẽ giảm được khả năng chọn

phải phần tử cực trị, khiến cho trường hợp xấu nhất khó xảy ra hơn.

Sau đây là hàm chọn phần tử median of three:

**Algorithm** Median of Three (a[], first, last)

Initialize $mid \leftarrow (first + last)/2$
**if** $a[first] \leq a[last]$ **then**
    swap a[first] and a[last]
**end if**
**if** $a[mid] \leq a[first]$ **then**
    swap a[mid] and a[first]
**end if**
**if** $a[last] \leq a[mid]$ **then**
    swap a[last] and a[mid]
**end if**
return mid

## 2.9 Counting Sort

### 2.9.1 Ý tưởng

Counting sort là một thuật toán sắp xếp không dựa trên so sánh. Thuật toán này có thể được thể hiện đơn giản bằng việc đếm số lần xuất hiện của phần tử trong danh sách. Số đếm được lưu trữ vào một danh sách con riêng. Danh sách này sẽ có kích thước phụ thuộc vào giá trị của phần tử lớn nhất trong danh sách. Trong đó, mỗi phần tử sẽ đại diện cho số lần xuất hiện của phần tử có giá trị bằng với chỉ số của phần tử đó trong danh sách chính. Giả sử ta có 3 phần tử 5 trong danh sách chính, thì trong danh sách con, giá trị của phần tử thứ 5 sẽ là 3.

Bằng cách thực hiện các phép tính toán trên danh sách con, ta sẽ tìm ra vị trí của các phần tử trong danh sách đã sắp xếp.

### 2.9.2 Trình bày thuật toán

**Algorithm** Counting Sort (a[], n)

$max \leftarrow largest\ element\ of\ the\ list$

$count \leftarrow l$

**for** *each element* $a[i]$ *in* $a$ **do**

Find the number of times element a[i] appears in array a and assign it to array count

$count[a[i]] \leftarrow count[a[i]] + 1$

**end for**

**for** *each element* $count[i]$ *in* $count$ **do**

Assign count[i] by the sum of count[i] and its previous element

$count[i] \leftarrow count[i] + count[i-1]$

**end for**

**for** *each element* $count[i]$ *in* $count$ **do**

Assign count[i] by the value of its previous element, except the element count[0]

$count[i] \leftarrow count[i-1]$

**end for**

$count[0] \leftarrow 0$

**for** *each element* $a[i]$ *in* $a$ **do**

$result[count[a[i]]] \leftarrow a[i]$

$count[a[i]] \leftarrow count[a[i]] + 1$

**end for**

**for** $i \leftarrow 0$ *to* $n$ **do**

$a[i] \leftarrow result[i]$

**end for**

### 2.9.3 Minh họa thuật toán

Ta có mảng chưa sắp xếp a[] = 4, 11, 9, 5, 13, 9. Dùng thuật toán Counting Sort để sắp xếp mảng tăng dần.

Tìm phần tử lớn nhất(max) trong mảng, ở đây cụ thể là 13

Tạo một mảng count có max + 1 phần tử, cụ thể ở đây là 14 và khởi tạo các phần tử bằng 0

Tạo một mảng result có số phần tử bằng với số phần tử trong mảng gốc.

**Lần 1:** Đếm số lần xuất hiện của các phần tử trong mảng và ghi kết quả vào mảng count

Count:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | **2** | 0 | **1** | 0 | **1** |

**Lần 2:** Thay đổi giá trị của các phần tử trong mảng count bằng tổng của nó với phần tử trước nó.

Count:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **1** | **2** | 2 | 2 | 2 | **4** | 4 | **5** | 5 | **6** |

**Lần 3:** Di chuyển các phần tử của count sang phải 1 ô và chuyển giá trị phần tử đầu tiên của count về 0.

Count:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** | **1** | 2 | 2 | 2 | **2** | 4 | **4** | 5 | **5** |

Lúc này giá trị của các phần tử trong mảng count đại diện cho vị trí của từng phần tử trong danh sách đã được sắp xếp.

**Lần 4:** Sắp xếp các phần tử trong mảng a vào mảng result dựa vào các chỉ số trong mảng count

a:

| 4 | 11 | 9 | 5 | 13 | 9 |
|---|---|---|---|---|---|

i = 0, count[a[0]] = count[4] = 0 => Đưa 4 vào vị trí 0

result:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | - | - | - | - | - |

count[4] = 1

i = 1, count[a[1]] = count[11] = 4 => Đưa 11 vào vị trí 4

result:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | - | - | - | 11 | - |

count[11] = 4 + 1 = 5

i = 2, count[a[2]] = count[9] = 2 => Đưa 9 vào vị trí 2

result:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | - | 9 | - | 11 | - |

count[9] = 2 + 1 = 3

i = 3, count[a[3]] = count[5] = 1 => Đưa 5 vào vị trí 1

result:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | 5 | 9 | - | 11 | - |

count[5] = 1 + 1 = 2

i = 4, count[a[4]] = count[13] = 5 => Đưa 13 vào vị trí 5

result:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | 5 | 9 | - | 11 | 13 |

count[13] = 5 + 1 = 6

i = 5, count[a[5]] = count[9] = 3 => đưa 9 vào vị trí 3

result:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | 5 | 9 | 9 | 11 | 13 |

count[9] = 3 + 1 = 4

i = 6, kết thúc vòng lặp 4.

**Lần 5:** Gán lần lươt các giá trị của mảng result vào mảng a.

a:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | 5 | 9 | 9 | 11 | 13 |

**Chú thích:**

- Các phần tử màu xanh trong mảng count thể hiện rằng các phần tử đó cần được chú ý, bởi chúng đại diện các phần tử cần được sắp xếp trong mảng. Hay nói cách khác, ta chỉ cần quan tâm đến sự thay đổi giá trị của các phần tử ấy mà thôi.
- Trong quá trình sắp xếp các phần tử vào mảng result, nếu phần tử trong mảng vẫn chưa được gán với giá trị nào, chúng sẽ được thể hiện bằng dấu "-".

### 2.9.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Counting Sort | $O(n+m)$ | $O(n+m)$ | $O(n+m)$ | $O(n+m)$ | Có |

**Phân tích thời gian**

Trong quá trình thực hiện thuật toán:

- Tìm phần tử max và di chuyển các phần tử trong danh sách mất O(n) thời gian.
- Tạo danh sách con và thực hiện các phép toán trong danh sách con mất O(m) thời gian, với m là giá trị của phần tử lớn nhất trong danh sách.

Do đó. trong mọi trường hợp, độ phức tạp thời gian của thuật toán là O(n + m)

**Phân tích không gian**

- Trong Counting sort, ngoài cần không gian phụ để lưu trữ danh sách sau khi đã sắp xếp, còn cần thêm không gian để lưu trữ các số lần xuất hiện của các phần tử trong danh sách. Kích thước không gian phụ này phụ thuộc vào giá trị của phần tử lớn nhất trong danh sách.
- Do đó, độ phức tạp về không gian của Counting sort là O(m + n)

**Tính Stable:**

Trong minh họa ở trên, có thể thấy có hai phần tử cùng mang giá trị là 9, và sau quá trình sắp xếp, thứ tự tương đối giữa hai phần tử trên không thay đổi, do đó thuật toán Counting Sort được xem là Stable.

### 2.9.5 Cải thiện

Hiện tại, thuật toán được trình bày ở trên chỉ hoạt động với dữ liệu đầu vào là số nguyên không âm, điều này chưa đáp ứng được nhu cầu trong thực tế. Vì vậy để thuật toán hoạt động với cả số nguyên âm nữa cần một số bổ sung sau:

- Giả định dữ liệu đầu vào là một danh sách gồm các số không âm. Để làm được điều này, trước hết ta cần tìm thêm phần tử min của danh sách. Sau đó trừ giá trị của tất cả các phần tử trong danh sách cho min. lúc này ta sẽ có được một danh sách gồm phần tử bé nhất là 0, lớn nhất là max - min
- Thực hiện việc sắp xếp tương tự cho danh sách mới được điều chỉnh.
- Cộng thêm các phần tử trong danh sách đã được cho min, lúc này ta có được danh sách hoàn chỉnh cần tìm.

Giả sử ta có một mảng b như sau:

b:

-4 2 -3 0 1 6

Trong mảng trên, phần tử max là 6, min là -4, do đó nếu ta trừ toàn bộ các phần tử trong mảng cho -4 ta được mảng b' như sau:

b':

0 6 1 4 5 10

Giờ ta chỉ cần sắp xếp các phần tử trong mảng b' rồi lần lượt cộng kết quả đó với min, là có thể sắp xếp được mảng a theo thứ tự.

Bên cạnh đó, trong trường hợp danh sách không có phần tử âm, ta có thể tiết kiệm được không gian cho mảng count. Với ví dụ trên, thay vì dùng 14 ô nhớ cho mảng count, thì với phần tử min là 4, kích thước mảng count lúc này chỉ cần là 13 - 4 + 1 = 10

## 2.10 Radix Sort

### 2.10.1 Ý tưởng

Radix sort là thuật toán sắp xếp không bằng cách so sánh. Ý tưởng của Radix sort là sắp xếp theo từng chữ số một từ hàng đơn vị tới hàng chục, hàng trăm...

Để sắp xếp những chữ số đó thì Radix có thể sử dụng phương pháp giống như Counting sort hay Bucket sort. Ta sẽ dùng theo phương pháp Counting sort với phần tử trong mảng cần so sánh có cơ số là 10.

Nó sẽ tạo một mảng để lưu vị trí mà phần tử ban đầu cần xếp vào. Mảng đó chỉ cần số phần tử là cơ số của phần tử trong mảng cần sắp xếp (ví dụ số nguyên dương thì có cơ số là 10, số nhị phân thì cơ số là 2). Cách lưu vị trí và sắp xếp sẽ giống như cách sắp xếp các phần tử của mảng của thuật toán Counting sort. Điểm khác biệt ở đây là ta chỉ sắp xếp chữ số ở hàng và từ các chữ số đó ta truy về số ban đầu. Lặp lại đến khi sắp xếp hết tất cả các hàng.

### 2.10.2 Trình bày thuật toán

**Algorithm** Radix sort(a[], n)

Find the max number of digits in a number in the array, Assign it to d

$m \leftarrow 0$

**while** $m < d$ **do** ▷ //Loop through each digit number in a number, from units place digit to tens place digit, to hundreds place digit,...

Create an array where each element is equal to 0, named *count*[]

$exp \leftarrow 10^m$

$i = 0$

**while** $i < n$ **do** ▷ // Store count of occurrences of digit in *count*

$count[(a[i]/exp) \mod 10] \leftarrow count[(a[i]/exp) \mod 10] + 1$

$i \leftarrow i + 1$

**end while**

$i \leftarrow 1$

**while** $i < 10$ **do** ▷ // Update count[] to store position of elements based on current digit number

$count[i] \leftarrow count[i - 1] + count[i]$

**end while**

Create an array that stores the sorted result based on current digit number, named *output*[]

$i = n - 1$

**while** $i \geq 0$ **do** ▷ // Build *output* array

$output[count[(a[i]/exp) \mod 10] - 1] \leftarrow a[i]$

$count[(a[i]/exp) \mod 10] \leftarrow count[(a[i]/exp) \mod 10] - 1$

$i \leftarrow i - 1$

**end while**

Assign *output*[] to $a$[]

$m \leftarrow m + 1$

**end while**

### 2.10.3 Minh họa thuật toán

Ta có mảng ban đầu là a[] = 18, 36, 23, 34, 9, 55 với số phần tử $n = 6$. Ta sẽ sắp xếp theo chiều tăng dần:

| 18 | 36 | 23 | 34 | 9 | 55 |
|---|---|---|---|---|---|

Sắp xếp dựa theo các chữ số ở hàng đơn vị:

- Ta tạo mảng count[10] để đếm số chữ số xuất hiện (ví dụ có 2 chữ số 5 thì count[5] = 2).

- Thực hiện đếm số chữ số hàng đơn vị xuất hiện ở mảng count, ta thấy các chữ số 8, 6, 3, 4, 9, 5 chỉ xuất hiện một lần nên ta có mảng *count*:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

- Tiến hành thay đổi giá trị mảng *count* để mảng lưu trữ vị trí các phần tử bằng cách dùng $count[i] = count[i-1] + count[i]$ với $i$ là vị trí đang xét trong mảng.

- Khi $i = 1, 2, 3$, giá trị $count[i]$ không thay đổi do $count[i-1] = 0$. Xét $i = 4$, do $count[3] = 1$ nên tăng $count[4]$ lên 1:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 2 | 1 | 1 | 0 | 1 | 1 |

- Xét $i = 5$, do $count[4] = 2$ nên tăng $count[5]$ lên 2:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 2 | 3 | 1 | 0 | 1 | 1 |

- Tương tự như vậy, ta có mảng count sau khi thay đổi là:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 5 | 6 |

- Giờ ta sẽ tiến hành sắp xếp dựa theo các chữ số hàng đơn vị. Xét các phần tử mảng $a$ theo thứ tự từ phần tử cuối lên phần tử đầu. Giả sử phần tử $a[i]$ có chữ số hàng đơn vị là $k$ thì vị trí của phần tử $a[i]$ sau khi sắp xếp là $count[k] - 1$, đồng thời giảm $count[k]$ đi 1

- Xét $i = n - 1 = 5$, ta thấy $a[5] = 55$ có chữ số hàng đơn vị là 5, suy ra xếp 55 ở vị trí $count[5] - 1 = 2$, đồng thời giảm $count[5]$ đi 1

  Mảng count:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 2 | 3 - 1 = 2 | 4 | 4 | 5 | 6 |

  Mảng kết quả:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Ouput | | | 55 | | | |

- Xét $i = 4$, ta thấy $a[4] = 9$ có chữ số hàng đơn vị là 9, suy ra xếp 9 ở vị trí $count[9] - 1 = 5$, đồng thời giảm $count[9]$ đi 1

  Mảng count:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 2 | 2 | 4 | 4 | 5 | 6 - 1 = 5 |

  Mảng kết quả:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Ouput | | | 55 | | | 9 |

- Xét $i = 3$, ta thấy $a[3] = 34$ có chữ số hàng đơn vị là 4, suy ra xếp 34 ở vị trí $count[4] - 1 = 1$, đồng thời giảm $count[4]$ đi 1

  Mảng count:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 1 | 2 - 1 = 1 | 2 | 4 | 4 | 5 | 5 |

  Mảng kết quả:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Ouput | | 34 | 55 | | | 9 |

- Tương tự với $i = 2, 1, 0$. Ta có kết quả cuối cùng như sau: Mảng count:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 5 |

Mảng kết quả:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Ouput | 23 | 34 | 55 | 36 | 18 | 9 |

Sau khi có được mảng kết quả, ta dùng mảng đó để sắp xếp mảng dựa theo các chữ số ở hàng chục. Ở mảng trên, phần tử 9 không có hàng chục nên ta coi chữ số hàng chục là 0. Từ đó sắp xếp giống như dựa trên chữ số hàng đơn vị, ta có:

Mảng count khi dùng để đếm số chữ số xuất hiện:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 1 | 1 | 1 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |

Mảng count sau khi biến đổi:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 1 | 2 | 3 | 5 | 5 | 6 | 0 | 0 | 0 | 0 |

Mảng count sau khi sắp xếp:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 1 | 2 | 3 | 5 | 5 | 0 | 0 | 0 | 0 |

Mảng kết quả:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Ouput | 9 | 18 | 23 | 34 | 36 | 55 |

Vì mảng không có hàng trăm để sắp xếp tiếp nên lấy mảng kết quả trên làm kết quả cuối cùng của việc sắp xếp của thuật toán.

### 2.10.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Radix Sort | $O(d * n)$ | $O(n^2)$ | $O(d * (n + b))$ | $O(n + b)$ | Có |

**Phân tích độ phức tạp thời gian và không gian**

- $d$ là số chữ số lớn nhất của phần tử trong mảng
- $b$ là hệ cơ số của phần tử trong mảng
- Trường hợp xấu nhất: khi tất cả phần tử trong mảng có cùng số chữ số trừ 1 phần tử có số chữ số rất lớn
- Trường hợp tốt nhất: khi tất cả phần tử trong mảng có cùng số chữ số
- Do thuật toán có một mảng phụ để lưu biến đếm có kích cỡ là b và một mảng phụ lưu kết quả mảng nên ta có độ phức tạp không gian là: $O(n + b)$

### 2.10.5 Khác nhau giữa Radix sort và Counting sort

Tuy thuật toán Radix sort xài theo phương pháp của Counting sort nhưng đối với phần tử có giá trị như $10^9$ thì Counting sort phải tạo một mảng có $10^9$ phần tử để thực hiện việc so sánh, còn Radix sort chỉ cần quan tâm đến cơ số của phần tử trong mảng, từ đó tạo ra mảng tương ứng.

Tùy phạm vi phần tử mà Radix sort có thể chậm hay nhanh hơn Counting sort. Với mảng cần sắp xếp với số phần tữ là $n$, Counting sort sẽ luôn nhanh hơn Radix sort nếu như phạm vị giá trị trong mảng nhỏ hơn hoặc ngang với $n$. Còn với trường hợp như phạm vi giá trị là $n^2$ thì Counting Sort sẽ có độ phức tạp là $O(n^2)$, Radix sort sẽ vẫn giữ nguyên độ phức tạp là $O(d * (n + b))$, nhanh hơn nhiều so với Counting sort.

Với việc sắp xếp theo từng chữ số một để thì Radix sort có dùng cách đó để sắp xếp theo từng kí tự trong mảng chuỗi. Còn Counting sort chỉ sắp xếp được số nguyên dương hay 1 kí tự mà thôi.

## 2.11 Flash Sort

### 2.11.1 Ý tưởng

Flash Sort là một thuật toán cải tiến của Bucket Sort. Có 3 quá trình cơ bản diễn ra trong Flash Sort:

- Quá trình 1: Xác định số bucket dùng để chia mảng và vị trí các bucket trong danh sách
- Quá trình 2: Phân bố vị trí của mỗi phần tử trong danh sách về đúng bucket tương ứng
- Quá trình 3: Sắp xếp danh sách sử dụng Insertion Sort - hoặc một loại sorting algorithm khác

Vấn đề trọng tâm của bài toán Flash Sort chính là việc sắp xếp các phần tử vào đúng bucket của nó với độ phức tạp không gian và thời gian tốt nhất - quá trình 2. Để đạt được điều này, flash sort đánh dấu vị trí cuối cùng của mỗi bucket và sắp xếp các phần tử không đúng bucket vào vị trí đó, đồng thời giảm độ dài của bucket. Với thao tác trên, Flash Sort có thể phân bố vị trí phần tử của danh sách về đúng bucket tương ứng chỉ sau $O(n)$ độ phức tạp thời gian và dùng chỉ dùng $O(n)$ độ phức tạp không gian.

### 2.11.2 Trình bày thuật toán

**Quá trình 1: Xác định số bucket**

**Algorithm** Flash Sort (a[], n)

$m \leftarrow [0.43 * n]$

$L \leftarrow \ list\ of\ length\ m\ with\ all\ element\ value\ = 0$

$min \leftarrow min\ value\ in\ a[]$

$max \leftarrow max\ value\ in\ a[]$

**for** *each element in a* **do**

$current - bucket \leftarrow (m-1) * (a[i] - min)/(max - min)$

$Increase\ L[current - bucket]\ by\ 1$

**end for**

**for** *each element in L but first* **do**

*Add value of element prior*

**end for**

Lúc này ta đã có mảng L với m phần tử chứa vị trí cuối cùng của các bucket trong danh sách a

**Quá trình 2: Phân bố vị trí của mỗi phần tử vào đúng bucket**

**Algorithm** Flash Sort (a[], n)

$numMove \leftarrow 0$

$j \leftarrow 0$

$bucket \leftarrow 0$

$hold \leftarrow 0$

**while** $numMove < n - 1$ **do**

$bucket \leftarrow bucket\ of\ current\ element$

**while** $element\ j\ in\ a\ is\ at\ the\ right\ bucket\ (is\ already\ moved\ to\ bucket) \iff j > L[bucket] - 1$ **do**

$Move\ to\ next\ element \iff j++$

**end while**

$hold \leftarrow a[j]$

**while** $element\ j\ in\ a\ is\ not\ at\ the\ end\ of\ right\ bucket$ **do**

$bucket \leftarrow bucket\ of\ hold$

$L[bucket] \leftarrow L[bucket] - 1$

$Swap\ value\ of\ hold\ and\ element\ at\ the\ end\ of\ bucket\ a[L[bucket]]$

$numMove \leftarrow numMove + 1$

**end while**

**end while**

Lúc này ta đã đưa được các phần tử về đúng bucket của nó. Bước tiếp theo chúng ta sẽ dùng thuật toán Insertion Sort để sắp xếp lại danh sách a.

**Quá trình 3: Sắp xếp lại danh sách sử dụng Insertion Sort**

**Algorithm** Flash Sort (a[], n)

**for** $Each\ element\ ai\ in\ a$ **do**

$Move\ ai\ to\ j\ where\ j > 0\ and\ a[j]\ is\ smallest\ element > ai$

**end for**

### 2.11.3 Minh họa thuật toán

Chúng ta có một mảng a với 10 phần tử như sau:

| 5 | 16 | 9 | 10 | 32 | 28 | 19 | 12 | 3 | 15 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Giai đoạn 1: Xác định số bucket**

$$\begin{array}{c|c} m \leftarrow [0.43 * 10] = 4 & L \leftarrow [0;0;0;0] \\ min \leftarrow a[8] = 3 & max \leftarrow a[4] = 32 \end{array}$$

Tìm các bucket cho mỗi phần tử ai trong mảng a, với công thức:

$$bucket = (m-1) * \frac{(ai - min)}{max - min} = 3 * \frac{ai - 3}{29}$$

Ta được:

| a | 5 | 16 | 9 | 10 | 32 | 28 | 19 | 12 | 3 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket: | 0 | 1 | 0 | 0 | 3 | 2 | 1 | 0 | 0 | 1 |

Dãy L lúc này là:

5 3 1 1

Tính prefix sum cho từng phần tử trong mảng L:

5 8 9 10

**Giai đoạn 2: Phân bố vị trí của mỗi phần tử vào đúng bucket**

$numMove = 0$

$bucket = 0$

$hold = 0$

**Với** $numMove < 10 - 1 = 9$

Xét từ phần tử đầu tiên.

Phần tử a[0] chưa được di chuyển vào bucket - với j = 0 < L[bucket = 0] = 5. Ta bắt đầu đưa phần tử a[0] về đúng bucket.

Thực hiện đưa phần thử *hold* về đúng vị trí của của bucket.

$hold = a[0] = 5$

$bucket = 0$

Đặt hold về vị trí cuối cùng của bucket $= L[0] - 1 = 4$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 16 | 9 | 10 | 5 | 28 | 19 | 12 | 3 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 1 |

$numMove = 1$

$hold = a[4] = 32$

$bucket = 3$

Đặt hold về vị trí cuối cùng của bucket $= L[4] - 1 = 9$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 16 | 9 | 10 | 5 | 28 | 19 | 12 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 3 |

$numMove = 2$

$hold = a[9] = 15$

$bucket = 1$

Đặt hold về vị trí cuối cùng của bucket $= L[1] - 1 = 7$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 16 | 9 | 10 | 5 | 28 | 19 | 15 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 3 |

$numMove = 3$

$hold = a[7] = 12$

$bucket = 0$

Đặt hold về vị trí cuối cùng của bucket $= L[1] - 1 = 3$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 16 | 9 | 12 | 5 | 28 | 19 | 15 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 3 |

$numMove = 4$

$hold = a[3] = 10$

$bucket = 0$

Đặt hold về vị trí cuối cùng của bucket $= L[1] - 1 = 2$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 16 | 10 | 12 | 5 | 28 | 19 | 15 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 3 |

$numMove = 5$

$hold = a[2] = 9$

$bucket = 0$

Đặt hold về vị trí cuối cùng của bucket $= L[0] - 1 = 1$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 9 | 10 | 12 | 5 | 28 | 19 | 15 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 3 |

$numMove = 6$

$hold = a[2] = 16$

$bucket = 1$

Đặt hold về vị trí cuối cùng của bucket $= L[2] - 1 = 6$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 9 | 10 | 12 | 5 | 28 | 16 | 15 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 3 |

$numMove = 8$

$hold = a[6] = 19$

$bucket = 1$

Đặt hold về vị trí cuối cùng của bucket $= L[2] - 1 = 5$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 3 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 3 |

$numMove = 9$

$hold = a[5] = 28$

$bucket = 2$

Đặt hold về vị trí cuối cùng của bucket $= L[2] - 1 = 8$ và giảm giá trị $L[bucket] - 1$.

| a | 5 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 3 |

$numMove = 10$

$hold = a[8] = 3$

$bucket = 0$

Đặt hold về vị trí cuối cùng của bucket $= L[0] - 1 = 0$ và giảm giá trị $L[bucket] - 1$.

| a | 3 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| bucket | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 3 |

Lúc này, vị trí j của phần tử hold = 0 bằng với vị trí cuối cùng cùng của bucket 0 = L[0] = 0. Ta dừng vòng lặp.

Các phần tử trong mảng đã đuộc sắp xếp vào đúng vị trí của bucket trong mảng. Nhận thấy chúng ta chỉ cần đi qua mỗi phần tử 1 lần để hoàn thành sắp xếp các phần tử trong mảng vào đúng bucket của nó. Độ phức tạp thời gian là O(n).

Chú thích

- Các phần tử có màu xanh là các phần tử được đưa về đúng bucket trong quá trình thực hiện thuật toán.

**Quá trình 3: Sắp xếp lại danh sách sử dụng Insertion Sort**

| a | 3 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Với a[1]:**

| a | 3 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Với a[2]:**

| a | 3 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Với a[3]:**

| a | 3 | 9 | 10 | 12 | 5 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Với a[4]:**

| a | 3 | **5** | 9 | 10 | 12 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Di chuyển a[4] về vị trí i = 1.

**Với a[5]:**

| a | 3 | 5 | 9 | 10 | 12 | 19 | 16 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Với a[6]:**

| a | 3 | 5 | 9 | 10 | 12 | **16** | 19 | 15 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | **5** | 6 | 7 | 8 | 9 |

Di chuyển a[6] về vị trí i = 5.

**Với a[7]:**

| a | 3 | 5 | 9 | 10 | 12 | **15** | 16 | 19 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | **5** | 6 | 7 | 8 | 9 |

Di chuyển a[7] về vị trí i = 5.

**Với a[8]:**

| a | 3 | 5 | 9 | 10 | 12 | 15 | 16 | 19 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Với a[9]:**

| a | 3 | 5 | 9 | 10 | 12 | 15 | 16 | 19 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Danh sách đã được sắp xếp. Hoàn thành thuật toán trên mảng. Nhận thấy ta chỉ cần đi qua mỗi phần tử trong mảng 1 lần để hoàn thành sắp xếp các phần tử trong mảng. Độ phức tạp thời gian là O(n).

Chú thích

- Các phần tử có màu xanh là các phần tử đã được đưa xét đến trong quá trình sắp xếp.
- Các phần tử được in đậm là các phần tử đã được di chuyển về vị trí khác trong quá trình thực hiện thuật toán.

### 2.11.4 Độ phức tạp

| Thuật toán | Tốt nhất | Xấu nhất | Trung bình | Không gian | Là Stable |
|---|---|---|---|---|---|
| Flash Sort | $O(n)$ | $O(n^2)$ | $O(n)$ | $O(n)$ | Không |

**Phân tích thời gian**

- Trường hợp tốt nhất: Khi mảng đã sắp xếp và các bucket tuyến tính và có độ lớn bằng nhau, tức là khoảng cách của các phần tử trong danh sách không xa nhau
- Trường hợp xấu nhất: Khi mảng có các phần tử được hoặc chưa sắp xếp mà các bucket không tuyến tính và có độ lớn không bằng nhau, tức là khoảng cách của các phần tử trong danh sách xa nhau.

**Phân tích không gian**

- Flash Sort sử dụng mảng phụ để chứa vị trí cuối cùng của các bucket trong danh sách nên có độ phức tạp không gian là $O(n)$

### 2.11.5 Cải tiến và các biến thể

**Cải tiến của Flash Sort**

Dựa trên cách phân chia bucket của thuật toán Flash Sort, dễ thấy được rằng bucket cuối cùng của một mảng chỉ có một phần tử với phần tử đó là phần tử lớn nhất trong mảng đó.

$$(m-1) * \frac{a_{max} - a_{min}}{a_{max} - a_{min}} = m - 1$$

Để cải thiện thuật toán, sau khi xác định vị trí và số phần tử của các bucket trong mảng, ta có thể đổi vị trí của phần tử $a_{max}$ với phần tử cuối cùng. Vì bucket cuối cùng chỉ có một phần tử và chính là $a_{max}$ nên chúng ta chỉ cần xử lí m - 1 bucket còn lại của mảng.

# 3 KẾT QUẢ THỬ NGHIỆM VÀ NHẬN XÉT

## 3.1 Randomize Input

### 3.1.1 Bảng thống kê:

| **Data order: Randomized** | | | | | | |
|---|---|---|---|---|---|---|
| **Data size** | 10000 | | 30000 | | 50000 | |
| **Resulting statics** | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 121.611 | 100019998 | 973.929 | 900059998 | 2780.36 | 2500099998 |
| Insertion Sort | 64.3733 | 50076399 | 532.013 | 449713180 | 1532.84 | 1252354082 |
| Bubble Sort | 430.781 | 199980001 | 3937.73 | 1799940001 | 13107.5 | 4999900001 |
| Shaker Sort | 162.919 | 65453544 | 1736.84 | 594225971 | 5299.9 | 1649369296 |
| Shell Sort | 2.5411 | 669758 | 10.0508 | 2296092 | 22.1448 | 4492524 |
| Heap Sort | 2.7805 | 500718 | 8.832 | 1691797 | 16.2892 | 2969790 |
| Merge Sort | 4.619 | 583767 | 20.22 | 1937132 | 25.493 | 3382845 |
| Quick Sort | 1.4616 | 427830 | 4.8928 | 1261782 | 8.3533 | 2278456 |
| Counting Sort | 0.1684 | 70000 | 0.5298 | 210004 | 0.9204 | 315540 |
| Radix Sort | 1.572 | 120057 | 4.026 | 450071 | 9.159 | 750071 |
| Flash Sort | 0.4717 | 100619 | 1.3488 | 291866 | 3.6339 | 478357 |

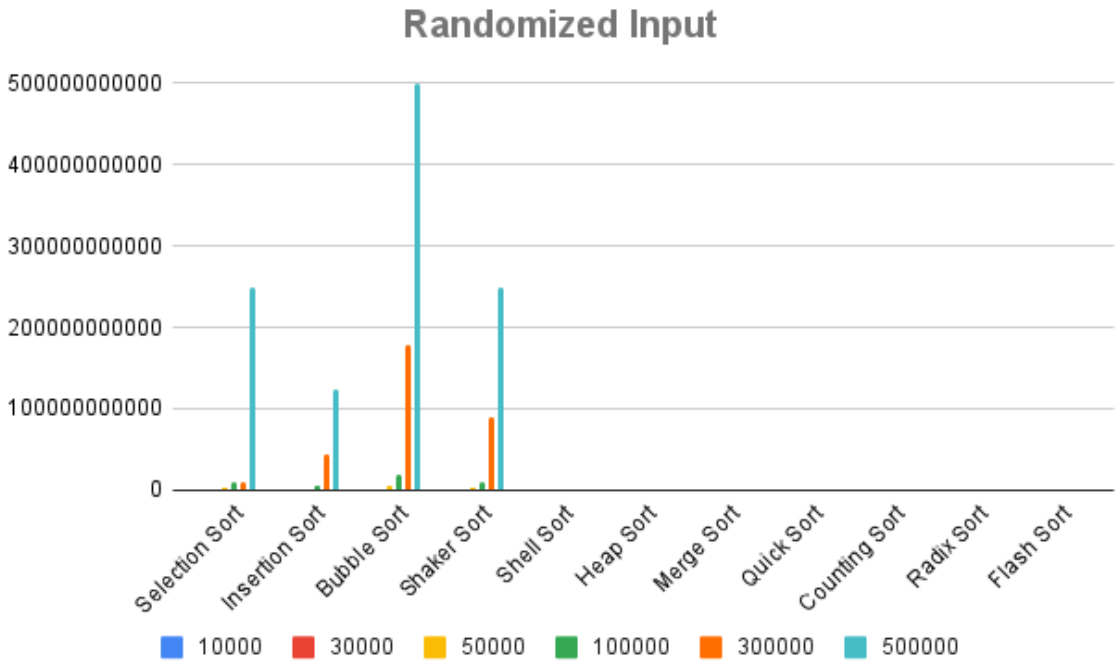
Bảng 1: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu ngẫu nhiên[1/2]

| **Data order: Randomized** | | | | | | |
|---|---|---|---|---|---|---|
| **Data size** | 100000 | | 300000 | | 500000 | |
| **Resulting statics** | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 12623.7 | 10000199998 | 108870 | 10000199998 | 307430 | 250000999998 |
| Insertion Sort | 6550.23 | 4994872630 | 57656 | 44967312080 | 175211 | 124867177202 |
| Bubble Sort | 54165.8 | 19999800001 | 454633 | 179999400001 | 1,342,880 | 499999000001 |
| Shaker Sort | 21223.8 | 6609339748 | 298190 | 59961703588 | 774239 | 166559250892 |
| Shell Sort | 37.9223 | 9911442 | 100.611 | 33841992 | 193.793 | 63743751 |
| Heap Sort | 35.9562 | 6338501 | 99.713 | 20901693 | 179.945 | 36291660 |
| Merge Sort | 50.062 | 7166083 | 113.839 | 23383199 | 176.766 | 40382264 |
| Quick Sort | 17.0939 | 5011796 | 43.2293 | 17453936 | 72.1699 | 32632906 |
| Counting Sort | 1.3557 | 565540 | 5.505 | 1565540 | 7.9246 | 2565540 |
| Radix Sort | 17.109 | 1500071 | 34.571 | 4500071 | 56.94 | 7500071 |
| Flash Sort | 9.5502 | 917355 | 19.3056 | 2804801 | 36.7717 | 4682081 |

Bảng 2: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu ngẫu nhiên[2/2]

### 3.1.2 Biểu đồ và nhận xét

Từ dữ liệu trên, ta có đồ thị biểu diễn số lượt so sánh của mỗi thuật toán như sau:



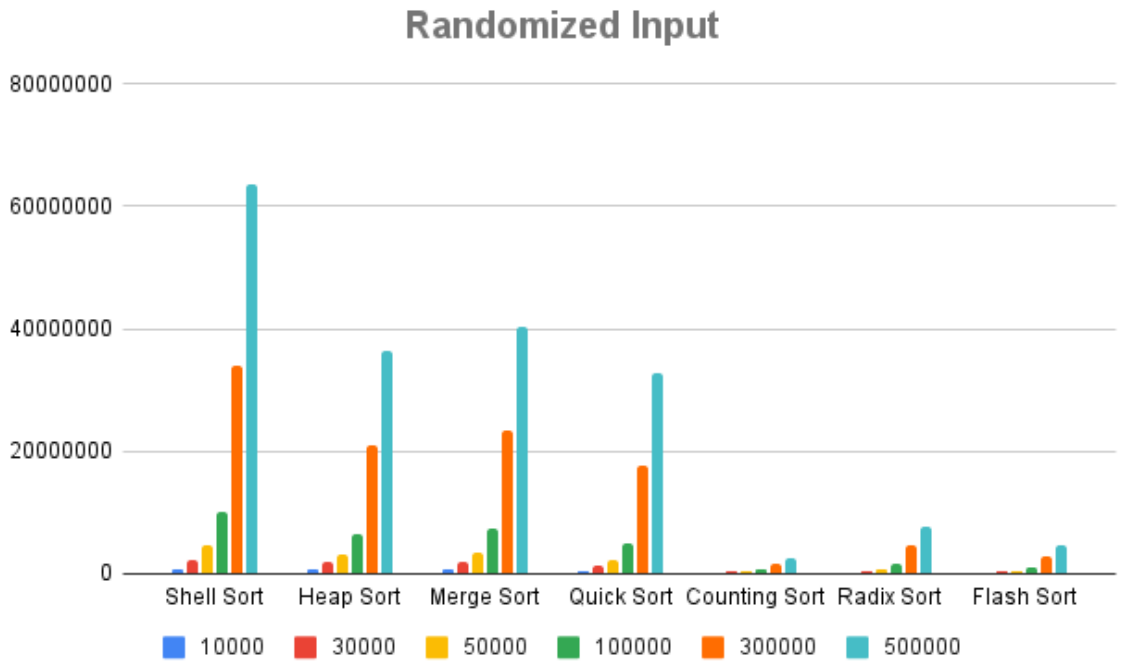

Hình 1: Biểu đồ phép so sánh của 11 thuật toán sắp xếp

**Nhận xét:** Thông qua đồ thị trên, ta có thể rút ra được một số kết luận sau:

- Thuật toán có số lượng phép so sánh lớn nhất là bubble sort, trên mọi kích thước dữ liệu. Điều này có thể lý giải bởi Bubble sort sẽ duyệt lần lượt từng phần tử một của mảng và so sánh phần tử đó với các phần tử đứng sau nó, và không tận dụng được thứ tự của dữ liệu đầu vào, dân đến Bubble sort đòi hỏi một lượng phép so sánh khổng lồ, có thể lên đến 500 tỷ với kích thước dữ liệu là 500000.

- Ngoài Bubble sort, các thuật toán như Selection sort, Insertion sort hay thâm chí là Shaker sort, dù là một phiên bản cải tiến của Bubble sort, nhưng với độ phức tạp là $O(n^2)$, cũng đòi hỏi một lượng phép so sánh lớn, gấp hàng nghìn những thuật toán còn lại.

Vì lượng chênh lệch quá lớn như vậy, nên đồ thị trên không thể thể hiện được hết tất cả số liệu của từng thuật toán, do đó,để hiểu rõ về những thuật toán khác, cần xét riêng đồ thị của 7 thuật toán khác với 4 thuật toán đã nêu trên.



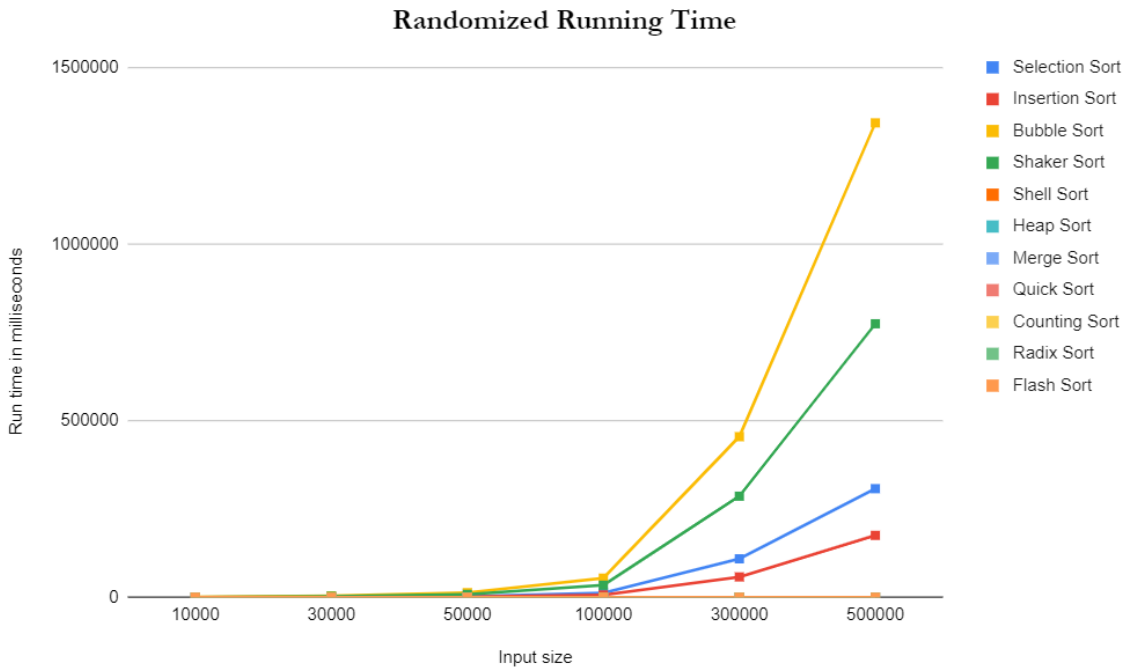

Hình 2: Biểu đồ phép so sánh của 7 thuật toán sắp xếp còn lại

Với đồ thị trên,ta rút ra các kết luận sau:

- Thuật toán có số phép so sánh bé nhất là Counting sort, bởi bản thân Counting sort không phải là một thuật toán dựa trên so sánh, nên nó sẽ có lợi thế hơn về phép so sánh so với những thuật toán khác, bên cạnh Counting sort, những thuật toán có phép so sánh ít có thể kể đến như Radix sort hay Flash sort.

- Các thuật toán còn lại như Shell sort, Heap sort, Quick sort, Merge sort đều là những thuật toán có độ phức tạp trung bình $O(n * logn)$, do đó số lượng phép so sánh của chúng không quá lớn và tương đối đồng đều.

Từ dữ liệu trên, ta có đồ đô thị biểu diễn thời gian chạy của mỗi thuật toán như sau:




Hình 3: Biểu đồ thời gian thực thi của 11 thuật toán sắp xếp

**Nhận xét** Thông qua đồ thị trên, ta có thể rút ra được một số kết luận sau:

- Thuật toán có thời gian chạy lâu nhất trên mọi kích thước dữ liệu là Bubble Sort. Điều này có thể lí giải vì số lượng so sánh của bubble sort là một con số rất lớn so với các thuật toán khác trên tập dữ liệu ngẫu nhiên (từ việc sử dụng vòng lặp 2 lần trên toàn bộ mảng), và độ phức tạp của nó cũng là $O(n^2)$ trên mọi trường hợp.

- Ngoài ra, một số thuật toán chạy khá chậm khác cũng đáng chú ý như Shaker Sort, Selection Sort, Insertion Sort, cũng có độ phức tạp thời gian là $0(n^2)$ trên tập dữ liệu ngẫu nghiên.

- Thuật toán có thời gian chạy nhanh nhất trên mọi kích thước dữ liệu là Counting Sort, vì tập dữ liệu cần sắp xếp là một tập dữ liệu số nguyên, và vì Counting Sort không phải là một thuật toán sắp xếp dựa trên so sánh, độ phức tạp thời gian của nó chỉ là $O(n)$.

## 3.2 Nearly Sorted Input

### 3.2.1 Bảng thống kê

| **Data order: Nearly sorted** | | | | | | |
|---|---|---|---|---|---|---|
| **Data size** | 10000 | | 30000 | | 50000 | |
| **Resulting statics** | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 114.482 | 100019998 | 1011.81 | 900059998 | 3192.98 | 2500099998 |
| Insertion Sort | 0.2183 | 181602 | 1.3998 | 528098 | 1.554 | 664658 |
| Bubble Sort | 215 | 199980001 | 2396 | 1799940001 | 6040 | 4999900001 |
| Shaker Sort | 0.3951 | 277150 | 1.073 | 801359 | 1.5159 | 910663 |
| Shell Sort | 0.9807 | 399761 | 3.4066 | 1315947 | 5.3962 | 2226560 |
| Heap Sort | 2.9878 | 519030 | 6.7547 | 1742704 | 9.5596 | 3060945 |
| Merge Sort | 5.52 | 500749 | 10.41 | 1642916 | 12.81 | 2800094 |
| Quick Sort | 0.4898 | 327577 | 1.2223 | 1086879 | 1.8662 | 1950247 |
| Counting Sort | 0.2147 | 70004 | 0.5476 | 210004 | 0.7625 | 350004 |
| Radix Sort | 1.351 | 120057 | 3.449 | 450071 | 9.759 | 750071 |
| Flash Sort | 0.401 | 122868 | 1.1187 | 368666 | 2.2024 | 614470 |

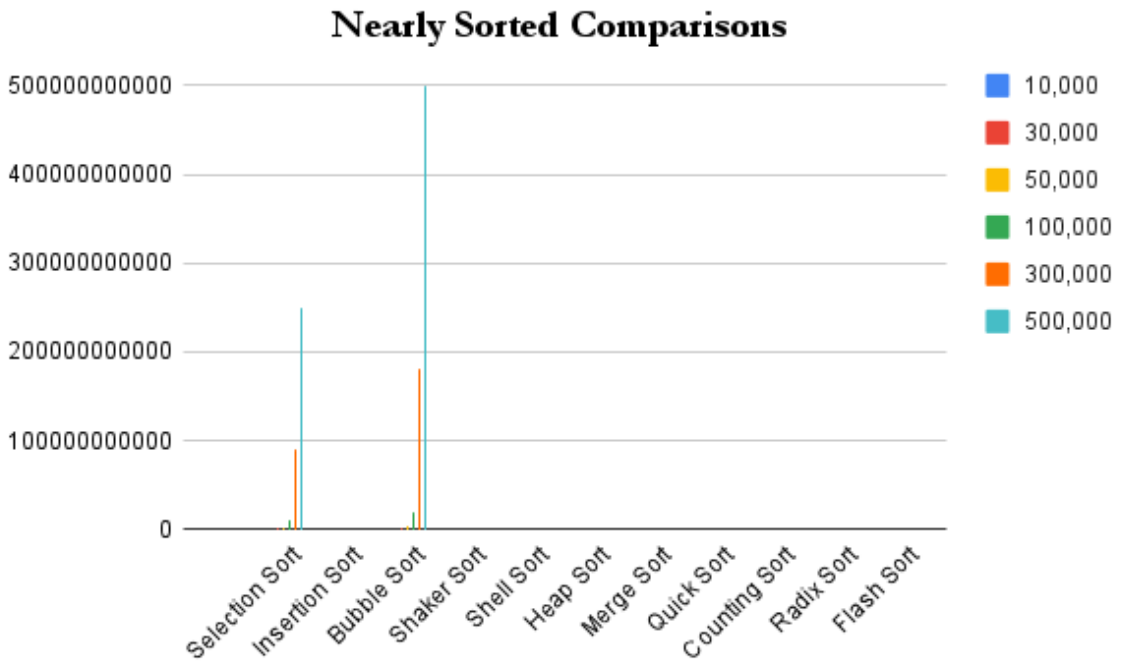
Bảng 3: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu gần sắp xếp[1/2]

| **Data order: Nearly sorted** | | | | | | |
|---|---|---|---|---|---|---|
| **Data size** | 100000 | | 300000 | | 500000 | |
| **Resulting statics** | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 20596.5 | 10000199998 | 108095 | 90000599998 | 316019 | 250000999998 |
| Insertion Sort | 1.2279 | 768378 | 1.6784 | 1380698 | 3.3582 | 1902710 |
| Bubble Sort | 22746 | 19999800001 | 224038 | 179999400001 | 588413 | 499999000001 |
| Shaker Sort | 1.8442 | 1010663 | 1.9447 | 1410663 | 2.3997 | 1810663 |
| Shell Sort | 7.1209 | 4684287 | 25.2998 | 15434663 | 40.1356 | 25647084 |
| Heap Sort | 24.7001 | 6525366 | 65.1569 | 21461297 | 104.85 | 37183475 |
| Merge Sort | 26.285 | 5834480 | 79.357 | 18731331 | 141.239 | 32110914 |
| Quick Sort | 6.6428 | 4150510 | 12.999 | 13431205 | 19.5862 | 23069651 |
| Counting Sort | 1.7188 | 700004 | 7.871 | 2100004 | 10.4032 | 3500004 |
| Radix Sort | 14.725 | 1500071 | 40.139 | 5400085 | 69.021 | 9000085 |
| Flash Sort | 3.5961 | 1228969 | 10.7507 | 3686969 | 16.9524 | 6144967 |

Bảng 4: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu gần sắp xếp[2/2]

### 3.2.2 Đồ thị và nhận xét

Từ dữ liệu trên, ta có đồ thị biểu diễn số lượt so sánh của mỗi thuật toán như sau:



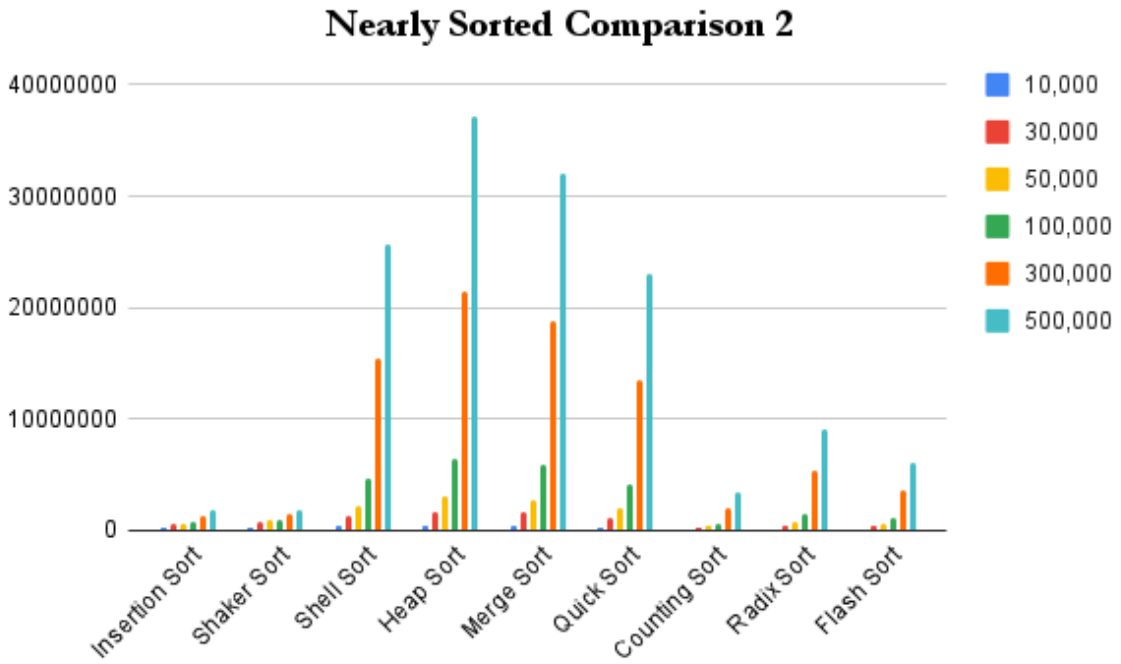

Hình 4: Biểu đồ phép so sánh của 11 thuật toán sắp xếp

**Nhận xét**

- Giới hạn của biểu đồ là 500 tỷ.
- Thuật toán sử dụng nhiều phép so sánh nhất là thuật toán Bubble Sort, với số lượt so sánh tại cỡ dữ liệu 500,000 lên đến 500 tỷ lần. Điều này có thể được lý giải vì thuật toán Bubble Sort sẽ so sánh 2 phần tử với nhau cho đến hết vòng lặp $O(n^2)$ bất kể trật tự phần tử.
- Thuật toán sử dụng nhiều thuật toán thứ 2 là Selection Sort, với số lượt so sánh tại cỡ dữ liệu 500,000 lên đến khoảng 250 tỷ lần.
- Các thuật toán sử dụng nhiều phép so sánh nhất là các thuật toán không tận dụng được trật tự gần sắp xếp của mảng, và vì vậy phải trải qua tất cả thứ tự so sánh phần tử cho đến phần tử thứ tự cuối cùng.
- Các thuật toán khác Selection Sort và Bubble Sort sử dụng số phép so sánh rất nhỏ so với 2 thuật toán trên. Để minh họa, tỉ lệ lượt so sánh của lần lượt bubble

sort và selection sort với heap sort ở kích thước dữ liệu 500,000 là gần 13,500 lần và 6700 lần. Vì vậy, không thể nhìn nhận và so sánh rõ ràng tử biểu đồ.

Từ nhận xét trên, thấy được sự cần thiết mở rộng phân tích với 9 phép so sánh khác Bubble Sort và Selection Sort.



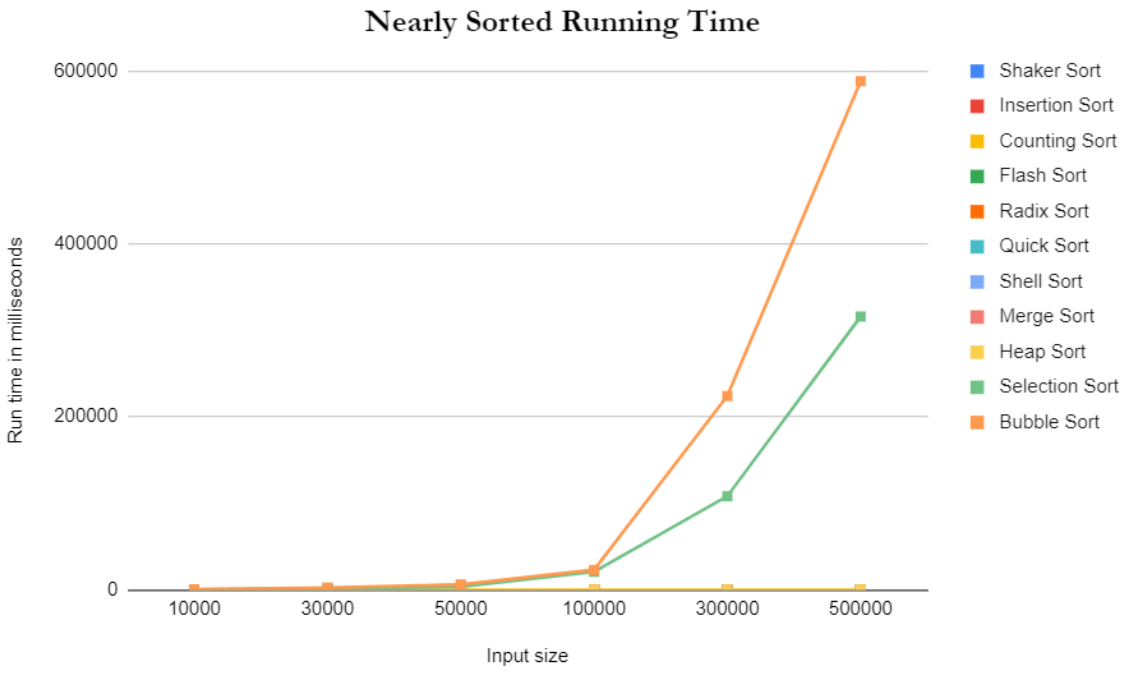

Hình 5: Biểu đồ phép so sánh của 9 thuật toán sắp xếp còn lại

**Nhận xét phụ:**

- Giới hạn của đồ thị trên chỉ đến 40 triệu, nhỏ hơn đồ thị 1 gấp 12,500 lần.
- Dễ thấy các thuật toán trên đều có độ phức tạp từ $O(n * log(n))$ tới $O(n)$
- Heap Sort là thuật toán sử dụng nhiều phép so sánh nhất trong 9 thuật toán trên. Với một thuật toán sử dụng $O(n * log(n))$ để tạo Heap và chỉnh sửa, quá trình chạy của thuật toán phải duyệt qua 1 phần tử nhiều lần.
- Insertion Sort là thuật toán sử dụng ít phép so sánh nhất trong 9 thuật toán trên. Với trật tự dữ liệu gần như sắp xếp, số lần insertion sort phải đi qua các phần tử là số phần tử không nằm đúng vị trí của mình.

- Nhìn chung, đối với trật tự dữ liệu gần so sánh, các thuật toán có thể tận dụng lợi thế gần sắp xếp của dữ liệu sẽ nhanh nhất (Insertion Sort, Shaker Sort, Shell Sort). Trong các thuật toán còn lại, các thuật toán sắp xếp dựa vào so sánh (Heap Sort, Merge Sort, Quick Sort) sẽ chậm hơn các thuật toán sắp xếp không dựa vào so sánh (Counting Sort, Radix Sort và Flash Sort).

Từ dữ liệu trên, ta có biểu đồ biểu diễn thời gian chạy của mỗi thuật toán như sau:




Hình 6: Biểu đồ thời gian thực thi của 11 thuật toán sắp xếp

**Nhận xét**

- Thuật toán có thời gian chạy chậm nhất trên mọi kích thước dữ liệu là Bubble Sort. Điều này có thể lí giải vì số phép so sánh của bubble sort là một con số rất lớn so với các thuật toán khác, và độ phức tạp của nó cũng là $O(n^2)$.

- Thuật toán chạy chậm thứ 2 là Selection Sort cũng có độ phức tạp thời gian là $0(n^2)$ khi chạy trên tệp dữ liệu gần có thứ tự.

- Thuật toán có thời gian chạy nhanh nhất trên mọi kích thước dữ liệu là Insertion Sort, vì tập dữ liệu là gần có thứ tự, chỉ có một vài phần tử nằm sai vị trí, nên

việc tìm ra các phần tử đó và chèn vào vị trí đúng cần rất ít thao tác, do đó mà thời gian chạy của Insertion Sort trong trường hợp này là rất nhanh. Có thể thấy rằng, Insertion Sort tận dụng được tập dữ liệu đầu vào.

- Một thuật toán chạy rất nhanh khác là Shaker Sort, một bản cải tiến toàn diện của Bubble Sort, có thể tận dụng được tập dữ liệu đầu vào, khiến cho việc sắp xếp lại tập dữ liệu gần có thứ tự chỉ tốn độ phức tạp thời gian là $O(n)$.

## 3.3 Reversed input

### 3.3.1 Bảng thống kê

| Data order: Reversed | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 10000 | | 30000 | | 50000 | |
| Resulting statics | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 92.583 | 100019998 | 837.606 | 900059998 | 2344.83 | 2500099998 |
| Insertion Sort | 100.666 | 100009999 | 926.101 | 900029999 | 2517.83 | 2500049999 |
| Bubble Sort | 318.516 | 199980001 | 2905.53 | 1799940001 | 7998.29 | 4999900001 |
| Shaker Sort | 225.694 | 100005001 | 1989.1 | 900015001 | 5592.55 | 2500025001 |
| Shell Sort | 0.5738 | 475175 | 1.7139 | 1554051 | 3.1652 | 2844628 |
| Heap Sort | 1.6406 | 483402 | 3.837 | 1641469 | 6.6053 | 2882199 |
| Merge Sort | 2.653 | 476441 | 6.478 | 1573465 | 10.175 | 2733945 |
| Quick Sort | 0.2599 | 317609 | 0.8254 | 1056915 | 1.5202 | 1900295 |
| Counting Sort | 0.1553 | 70004 | 0.4666 | 210004 | 0.8921 | 350004 |
| Radix Sort | 0.602 | 120057 | 2.001 | 450071 | 3.743 | 750071 |
| Flash Sort | 0.3135 | 108553 | 0.5885 | 325653 | 0.9803 | 542753 |

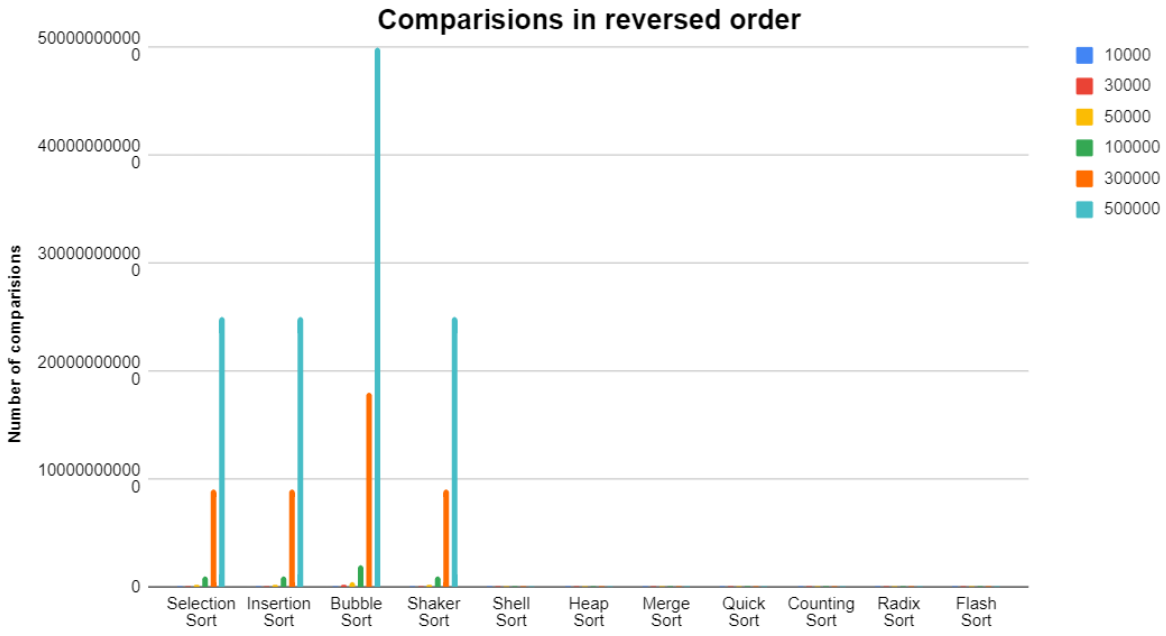
Bảng 5: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp ngược[1/2]

| **Data order: Reversed** | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 100000 | | 300000 | | 500000 | |
| Resulting statics | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 9285.87 | 10000199998 | 127077 | 90000599998 | 257457 | 250000999998 |
| Insertion Sort | 10300.1 | 10000099999 | 98524.4 | 90000299999 | 234836 | 250000499999 |
| Bubble Sort | 33013 | 19999800001 | 358183 | 179999400001 | 1205082 | 499999000001 |
| Shaker Sort | 22553.7 | 10000050001 | 268076 | 90000150001 | 668295 | 250000250001 |
| Shell Sort | 6.7042 | 6089190 | 22.4336 | 20001852 | 43.0913 | 33857581 |
| Heap Sort | 14.148 | 6153394 | 45.4804 | 20381638 | 81.2093 | 35456561 |
| Merge Sort | 19.153 | 5767897 | 61.265 | 18708313 | 97.129 | 32336409 |
| Quick Sort | 3.3064 | 4050554 | 10.1735 | 13131237 | 17.2032 | 22569695 |
| Counting Sort | 1.6781 | 700004 | 4.5447 | 2100004 | 7.5482 | 3500004 |
| Radix Sort | 7.186 | 1500071 | 24.251 | 5400085 | 40.737 | 9000085 |
| Flash Sort | 2.0816 | 1085503 | 6.4419 | 3256503 | 10.1745 | 5427503 |

Bảng 6: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp ngược[2/2]

### 3.3.2 Biểu đồ và đánh giá

Từ dữ liệu trên, ta có đồ thị biểu diễn số lượt so sánh của mỗi thuật toán như sau:

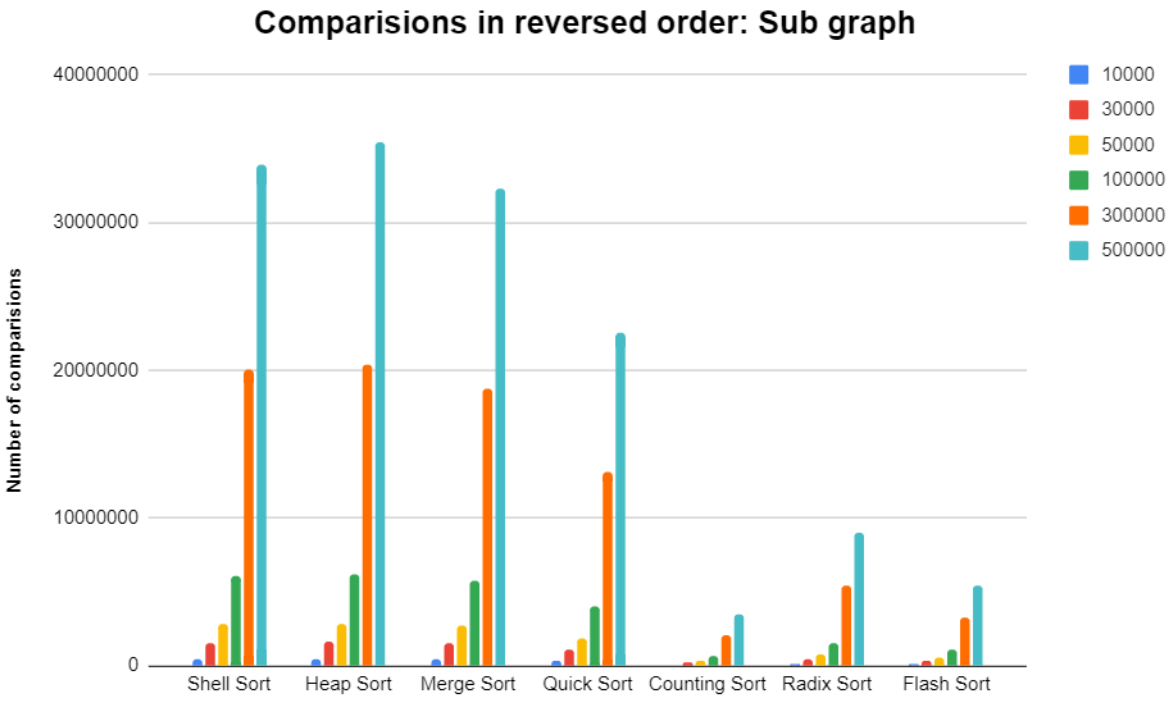



Hình 7: Biểu đồ phép so sánh của 11 thuật toán sắp xếp

**Nhận xét**

- Qua đồ thị ta thấy có 4 thuật toán có số lượng so sánh nhiều hơn đáng kể những thuật toán khác, Selection, Insertion, Bubble và Shaker sort. Trong đó Bubble sort có số lượng so sánh nhiều nhất, với số lượng tầm 500 tỷ khi kích cỡ dữ liệu là 500000. Nguyên nhân 4 thuật toán này có số lượng cao như vậy là vì có độ phức tạp $O(n^2)$. Ví dụ như Selection sort sẽ liên tục tìm phần tử nhỏ nhất bằng cách đi qua mảng số lần tương ứng với số phần tử trong mảng. Còn những thuật toán còn lại chỉ cần xét các phần tử trong mảng 1, 2 lần hoặc chia chỏ mảng ra.

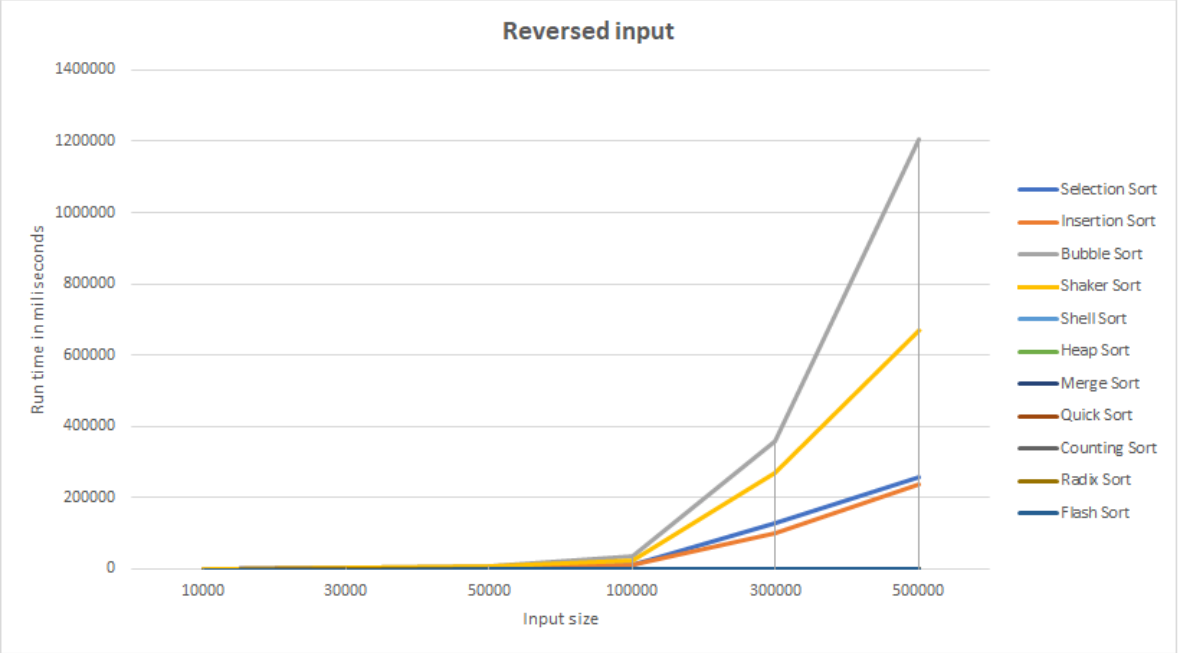
- Để phân tích rõ hơn thì ta có bảng của 7 thuật toán còn lại:




Hình 8: Biểu đồ phụ đo số lượng so sánh của 7 thuật toán sắp xếp còn lại

- Ta thấy rằng thuật toán Counting sort có số lượng so sánh nhỏ nhất trong tất cả các thuật toán. Với đặc điểm của thuật toán là không dùng phép so sánh, số lượng so sánh của nó chỉ phụ thuộc vào kích cỡ mảng mà nó cần đi qua nên giảm đi dược một cách đáng kể so với thuật toán sắp xếp dùng phép so sánh. 2 thuật toán Radix và Flash sort cũng không dùng phép so sánh nên ta có thể thấy sự khác nhau rõ ràng giữa 3 thuật toán này và các thuật toán còn lại.

Từ dữ liệu trên, ta có đồ biểu đồ đường biểu diễn thời gian chạy của mỗi thuật toán như sau:



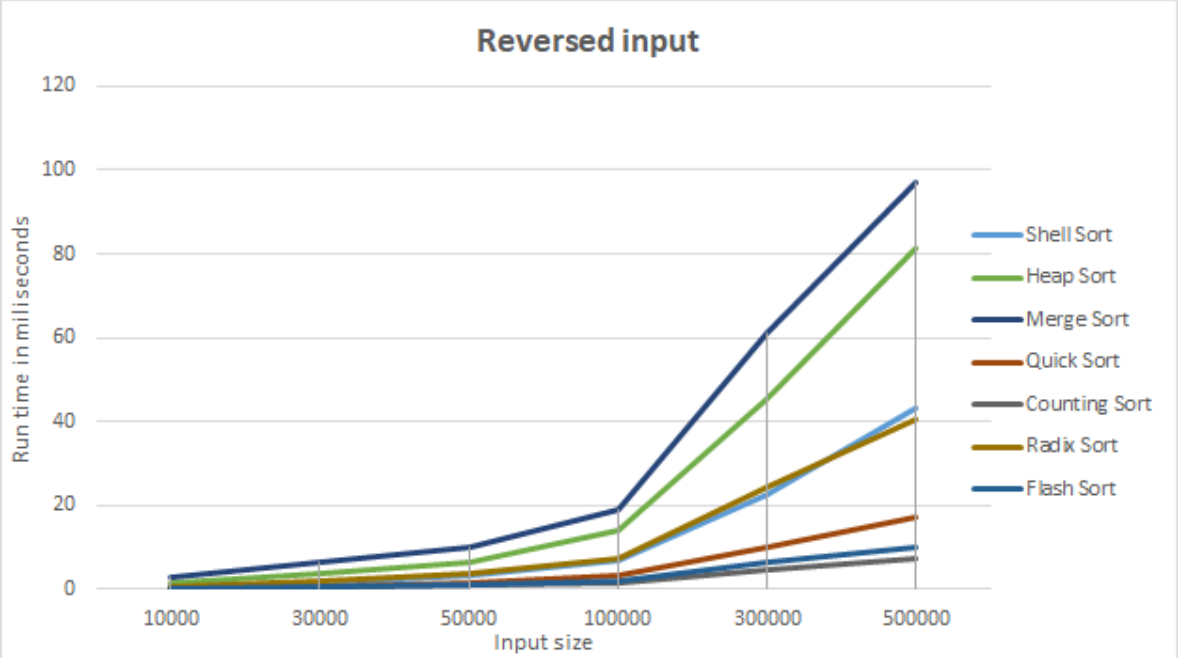

Hình 9: Biểu đồ thời gian thực thi của 11 thuật toán sắp xếp

**Nhận xét**

- Thuật toán có tốc độ thực thi chậm nhất đối với toàn bộ các kích thước dữ liệu là Bubble sort,bởi trong trường hợp này, Bubble sort phải di chuyển lần lượt từng phần tử từ vị trí đầu mảng về cuối mảng, do đó tiêu tốn rất nhiều thời gian.

- Cũng tương tự như Bubble sort, trong trường hợp này một số thuật toán khác như Insertion sort, Selection sort hay cả Shaker Sort cũng có thời gian thực thi rất lớn, chênh lệch rất nhiều so với các thuật toán khác, lên đến hàng chục nghìn lần

Mức chênh lệch như vậy dẫn đến đồ thị trên không thể hiện được hết các thông số của toàn bộ các thuật toán. Do đó, cần xét riêng 7 thuật toán còn lại khác với 4 thuật toán đã nêu trên.




Hình 10: Biểu đồ thời gian thực thi của 7 thuật toán sắp xếp còn lại

Qua đó, ta tiếp tục rút ra được những ý sau:

- Thuật toán có tốc độ thực thi nhanh nhất đối với mỗi kích thước dữ liệu là Counting sort. Ngoài ra các thuật toán như Radix sort hay Flash sort cũng có tốc độ nhanh không kém.

- Các thuật toán còn lại như Shell sort, Heap sort, Quick sort, Merge sort đều là những thuật toán có độ phức tạp trung bình $O(n * logn)$, do đó thời gian thực hiện của chúng cũng tương đối nhanh. Trong đó, Quick sort có tốc độ nhanh nhất, thể hiện tính ưu việt của nó so với những thuật toán có cùng độ phức tạp còn lại.

## 3.4 Sorted input

### 3.4.1 Bảng thống kê

| Data order: Sorted | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 10000 | | 30000 | | 50000 | |
| Resulting statics | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 110.66 | 100019998 | 970.311 | 900059998 | 2753.41 | 2500099998 |
| Insertion Sort | 0.0577 | 29998 | 0.1576 | 89998 | 0.2685 | 149998 |
| Bubble Sort | 215.15 | 199980001 | 1945.85 | 1799940001 | 5305.77 | 4999900001 |
| Shaker Sort | 0.0285 | 20002 | 0.0785 | 60002 | 0.2861 | 100002 |
| Shell Sort | 0.7519 | 360042 | 1.7353 | 1170050 | 4.9742 | 2100049 |
| Heap Sort | 1.9548 | 519036 | 8.7209 | 1742717 | 12.5793 | 3060914 |
| Merge Sort | 2.552 | 475242 | 14.102 | 1559914 | 14.138 | 2722826 |
| Quick Sort | 0.4797 | 327609 | 1.6171 | 1086915 | 2.5557 | 1950295 |
| Counting Sort | 0.1522 | 70004 | 0.4601 | 210004 | 0.7161 | 350004 |
| Radix Sort | 1.37 | 120057 | 3.988 | 450071 | 9.115 | 750071 |
| Flash Sort | 0.5352 | 122894 | 0.9963 | 368694 | 2.7729 | 614494 |

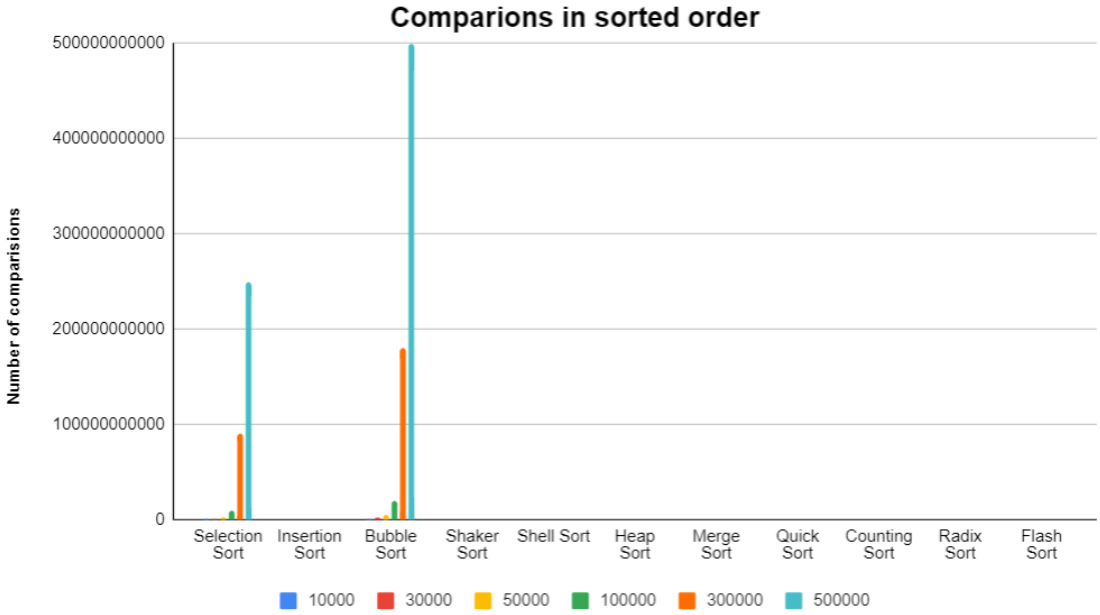
Bảng 7: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu được sắp xếp[1/2]

| **Data order: Sorted** | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 100000 | | 300000 | | 500000 | |
| Resulting statics | Running time | Comparison | Running time | Comparison | Running time | Comparison |
| Selection Sort | 14330 | 10000199998 | 105586 | 90000599998 | 346420 | 250000999998 |
| Insertion Sort | 0.852 | 299998 | 1.1536 | 899998 | 1.824 | 1499998 |
| Bubble Sort | 21514 | 19999800001 | 372149 | 179999400001 | 723582 | 499999000001 |
| Shaker Sort | 0.4442 | 200002 | 0.7126 | 600002 | 1.3108 | 1000002 |
| Shell Sort | 10.864 | 4500051 | 22.4628 | 15300061 | 40.9031 | 25500058 |
| Heap Sort | 21.6496 | 6525399 | 60.3868 | 21461203 | 106.934 | 37183363 |
| Merge Sort | 33.53 | 5745658 | 81.904 | 18645946 | 126.675 | 32017850 |
| Quick Sort | 6.4667 | 4150554 | 13.2634 | 13431237 | 20.352 | 23069695 |
| Counting Sort | 1.4689 | 700004 | 4.5159 | 2100004 | 7.6871 | 3500004 |
| Radix Sort | 13.164 | 1500071 | 41.926 | 5400085 | 70.718 | 9000085 |
| Flash Sort | 5.1484 | 1228994 | 9.5225 | 3686994 | 18.2701 | 6144994 |

Bảng 8: Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu được sắp xếp[2/2]

### 3.4.2 Đồ thị và đánh giá

Từ dữ liệu trên, ta có đồ thị biểu diễn số lượt so sánh của mỗi thuật toán như sau:




Hình 11: Biểu đồ phép so sánh của 11 thuật toán sắp xếp

**Nhận xét**

- Nhìn qua đồ thị ta có thể thấy được rằng 2 thuật toán Selection sort và Bubble sort có số lượng so sánh rất lớn so với các thuật toán khác khi kích thước dữ liệu từ 100000. Nguyên nhân là do 2 thuật toán có độ phức tạp $O(n^2)$ với mọi trường hợp.

- Trong 2 thuật toán Selection sort và Buuble sort thì Số lượng so sánh của Bubble sort lớn hơn với số lượng so sánh lần lượt tầm 200 tỷ, 500 tỷ khi kích cỡ dữ liệu là 300000, 500000 trong khi số lượng so sánh của Selection sort là tầm 100 tỷ, 200 tỷ.

- Để có thể thấy được số lượng so sánh của những thuật toán còn lại thì ta tạo ra một biểu đồ phụ với 9 thuật toán còn lại:


Hình 12: Biểu đồ phụ đo số lượng so sánh của 8 thuật toán sắp xếp còn lại

- Ta thấy rằng thuật toán Shaker sort có số lượng so sánh nhỏ nhất trong tất cả các thuật toán. Điều kiện dừng của Shaker sort là khi nó không cần hoán vị các phần tử nữa, tức là mảng đã được sắp xếp rồi. Cho nên với trường hợp mảng dữ liệu đầu vào là một mảng được sắp xếp rồi thì thuật toán chỉ cần chạy qua mảng một lần và kết thúc.

- Ngoài ra, thuật toán Insertion sort có số lượng so sánh khá gần Shaker sort vì nó cũng dựa trên việc nếu như 2 phần tử không cần phải hoán vị thì thuật toán chỉ cần chạy qua mỗi phần tử 1 lần mà không cần quay lại những phần tử trước đó để xét

Từ dữ liệu trên, ta có đồ đô thị biểu diễn thời gian chạy của mỗi thuật toán như sau:


Hình 13: Biểu đồ thời gian thực thi của 11 thuật toán sắp xếp

**Nhận xét**

- Thuật toán có thời gian chạy chậm nhất trên mọi kích thước dữ liệu là Bubble Sort. Điều này có thể lí giải vì số lượng so sánh của bubble sort là một con số rất lớn so với các thuật toán khác, và độ phức tạp của nó cũng là $O(n^2)$.Tuy nhiên, ta có thể cải tiến Bubble Sort bằng cách kiểm tra trước xem mảng đã được sắp xếp trước hay chưa (bằng cách sử dụng biến flag và ngừng chương trình ngay nếu không có sự đổi chỗ nào xảy ra). Bằng cách cải tiến này, Bubble Sort là thuật toán chạy rất nhanh trên tệp dữ liệu đã sắp xếp sẵn, với độ phức tạp thời gian chỉ là $O(n)$.

- Thuật toán có thời gian chạy nhanh nhất trên mọi kích thước dữ liệu là Shaker Sort. Shaker Sort được coi là một bản cải tiến của Bubble Sort, và tương tự như Bubble Sort cải tiến, Shaker Sort cũng có thể kiểm tra trước mảng đã được sắp

xếp chưa qua một lần lặp. Với tập dữ liệu đã có thứ tự, độ phức tạp thời gian của nó chỉ là $O(n)$.

- Một thuật toán chạy nhanh đáng chú ý nhất là Insertion Sort. Nó tận dụng được thứ tự có sẵn của tệp dữ liệu, chỉ cần duyệt mảng một lần duy nhất sẽ kết luận được mảng đã được sắp xếp, với độ phức tạp thời gian chỉ là $O(n)$, thời gian chạy của Insertion Sort chỉ chênh lệch một khoảng rất nhỏ so với Shaker Sort khi đi trên tập dữ liệu đã có thứ tự.

# 4 TỔNG KẾT

## 4.1 Kết quả thử nghiệm

Sau khi thực thiện thử nghiệm, ta có các thống kê sau:

| | **Thuật toán thực hiện** | |
|---|---|---|
| **Data Order** | **Tệ nhất** | **Tốt nhất** |
| Randomized | Bubble Sort | Counting Sort |
| Reversed | Bubble Sort | Shaker Sort |
| Nearly Sorted | Bubble Sort | Insertion Sort |
| Reversed | Bubble Sort | Counting Sort |

Bảng 9: Bảng thống kê thuật toán hoạt động tốt nhất và tệ nhất với lượt so sánh

| | **Thuật toán thực hiện** | |
|---|---|---|
| **Data Order** | **Tệ nhất** | **Tốt nhất** |
| Randomized | Bubble Sort | Counting Sort |
| Reversed | Bubble Sort | Shaker Sort |
| Nearly Sorted | Bubble Sort | Insertion Sort |
| Reversed | Bubble Sort | Counting Sort |

Bảng 10: Bảng thống kê thuật toán hoạt động tốt nhất và tệ nhất với thời gian chạy

## 4.2 Độ phức tạp

| **Algorithms** | **Best case** | **Average case** | **Worst case** | **Space** | **Stable** | **In - place** |
|---|---|---|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Không | Có |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Có | Có |
| Bubble Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Có | Có |
| Shaker Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Có | Có |
| Shell Sort | $O(n * log(n))$ | $O(n^{\frac{4}{3}})$ | $O(n^2)$ | O(1) | Không | Có |
| Heap Sort | $O(nlogn)$ | $O(nlogn)$ | $O(nlogn)$ | O(1) | Không | Có |
| Merge Sort | $O(nlogn)$ | $O(nlogn)$ | $O(nlogn)$ | $O(n)$ | Có | Không |
| Quick Sort | $O(nlogn)$ | $O(nlogn)$ | $O(n^2)$ | $O(1)$ | Không | Có |
| Counting Sort | $O(n+m)$ | $O(n+m)$ | $O(n+m)$ | $O(n+m)$ | Có | Không |
| Radix Sort | $O(d*n)$ | $O(d*(n+b))$ | $O(n^2)$ | $O(n+b)$ | Có | Không |
| Flash Sort | $O(n)$ | $O(n)$ | $O(n^2)$ | $O(n)$ | Không | Không |

Thông qua các bảng thống kê và đồ thị ở trên, ta có thể đi đến một số kết luận như sau:

- Đối với hầu hết các kiểu dữ liệu và kích thước dữ liệu, Counting sort là thuật toán có tốc độ nhanh nhất cũng như thực hiện ít phép so sánh nhất trong tất cả các thuật toán. Bên cạnh đó là Flash sort hay Radix sort, cả ba đều có độ ổn định rất tốt trên mọi trường hợp. Tuy nhiên, nhược điểm lớn của Counting Sort là đòi hỏi một lượng lớn không gian bộ nhớ để lưu trữ vị trí các phần tử, nếu xét trên khía cạnh tổng quát nhất thì flash sort sẽ là lựa chọn tối ưu hơn.
- Ngược lại, Bubble sort là thuật toán chậm nhất, cũng như thực hiện nhiều phép so sánh nhất trong mọi trường hợp, điều này xuất phát từ việc Bubble sort không tận dụng tốt được thứ tự của dữ liệu đầu vào dẫn đến những phép so sánh không

cần thiết trong quá trình thực thi thuật toán, từ đó kéo dài thời gian thực thi thuật toán. Nhận định trên có thể dễ dàng chứng minh được bởi số phép so sánh của Bubble sort trên mọi kiểu dữ liệu, đối với mỗi kích thước đều như nhau.

- Các thuật toán như Shaker sort hay Insertion sort tận dụng tốt dữ liệu đầu vào, nên với các kiểu dữ liệu như Sorted hay Nearly Sorted, cả hai thuật toán đều có tốc độ ấn tượng, vượt qua cả counting sort. Tuy nhiên, với trường hợp tệ nhất của chúng là khi trật tự dữ liệu ngược thì tốc độ lại giảm đi đáng kể.

- Các thuật toán stable bao gồm : Counting sort, Shaker sort, Bubble sort, Merge sort, Radix sort, Insertion sort.

## 4.3 Tổng quan từng thuật toán

### 4.3.1 Insertion Sort

- Nhìn chung, Insertion Sort là một thuật toán chạy khá chậm khi có độ phức tạp thời gian trung bình là $O(n^2)$.

- Điểm mạnh của nó là thao tác cài đặt vô cùng đơn giản, là một thuật toán stable và in-place.

- Insertion Sort hoạt động tốt trên tập dữ liệu có kích thước nhỏ. Nó cũng hoạt động tốt trên tập dữ liệu gần như có thứ tự hoặc tập dữ liệu đã được sắp xếp, điều này tận dụng được nguyên tắc hoạt động của Insertion Sort là tìm ra phần tử sai vị trí và chèn nó vào mảng đã sắp xếp.

### 4.3.2 Selection Sort

- Selection Sort là một sắp xếp thực hiện với thao tác chính là chèn. Ý tưởng chính của Selection Sort là đưa cực trị của mỗi danh sách về vị trí đầu tiên.

- Selection Sort có nhiều ưu điểm, môt trong đó chính là sự đơn giản trong việc trình bày và hiểu thuật toán. Selection Sort cũng là thuật toán sử dụng hoán vị

ít nhất trong tất cả các thuật toán sắp xếp được xét. Selection Sort cũng không yêu cầu không gian dữ liệu phụ - là một thuật toán In-place.

- Selection Sort cũng có nhiều hạn chế. Selection Sort hoạt động không tốt với danh sách có kích thước lớn, như được minh họa phần phân tích dữ liệu.

### 4.3.3 Bubble Sort

- Nhìn chung, Bubble sort là một thuật toán đơn giản, dễ hiểu và dễ cài đặt, cũng như không cần thêm không gian phụ cho việc sắp xếp.

- Tuy nhiên, Bubble sort vẫn tồn tại những hạn chế, đó là không tận dụng được trật tự của dữ liệu đầu vào. Trong mọi trường hợp, thuật toán đều phải lần lượt duyệt qua từng phần tử và so sánh với mọi phần tử đứng sau nó. Điều này tiêu tốn một lượng lớn thời gian cho những phép so sánh không cần thiết, dẫn đến thời gian thực thi thuật toán là rất lớn cho mọi trường hợp. Về cơ bản, độ phức tạp thời gian của Bubble sort luôn là $O(n^2)$ cho mọi trường hợp.

- Mặc dù vậy, những nhược điểm trên có thể được khắc phục theo rất nhiều cách khác nhau. Trên thực tế, đã có nhiều thuật toán là phiên bản cải tiến của Bubble sort, một trong số đó là Shaker sort.

### 4.3.4 Shaker Sort

- Là thuật toán cải tiến của Bubble sort, Shaker sort đã khắc phục được những nhược điểm vốn có của Bubble sort như luôn thực hiện một lượng phép so sánh như nhau đối với mọi trật tự dữ liệu đầu vào, dẫn đến việc tiêu tốn thời gian không cần thiết. Trong hầu hết các trường hợp, Shaker sort sẽ nhanh hơn Bubble sort xấp xỉ 2 lần.

- Tương tự như Insertion sort, Shaker sort tận dụng tốt trật tự dữ liệu đầu vào, điều này giúp thuật toán Shaker sort có tốc độ rất nhanh đối với những kiểu dữ

liệu đầu vào gần như được sắp xếp. Đặc biệt, với dữ liệu đầu vào đã được sắp xếp sẵn, tốc độ thuật toán rất nhanh với độ phức tạp về thời gian chỉ là $O(n)$

- Còn đối với những các kiểu dữ liệu khác, tốc độ của shaker sort không quá nổi trội, bởi trên thực tế, độ phức tạp thời gian của Shaker sort lên đến $O(n^2)$

### 4.3.5 Shell Sort

- Nhìn chung, Shell Sort giải quyết các bài toán so sánh khá nhanh, đặc biệt đối với dữ liệu có hoặc gần có trật tự, vì nó có thể tận dụng trật tự dữ liệu như Insertion Sort.

- Điểm mạnh của Shell Sort nằm ở độ phức tạp thời gian của nó - là một cải thiện so với Insertion Sort. Với các dãy khoảng cách cải tiến, trường hợp tệ nhất của Shell Sort sẽ được cải thiện tới $\mathrm{O}(n^{(\frac{4}{3})}$

- Điểm hạn chế của Shell Sort là độ phức tạp thời gian tốt nhất của nó. Kể cả với dữ liệu đã sắp xếp, Shell Sort sẽ cần đến độ phức tạp $\mathrm{O}(n * log(n))$.

### 4.3.6 Heap Sort

- Heap Sort là một thuật toán sắp xếp đặc biệt khi nó dựa trên cấu trúc dữ liệu heap (đống) và sử dụng tính chất đặc biệt của heap để sắp xếp.

- Heap Sort là thuật toán sắp xếp có độ phức tạp thời gian như nhau ở mọi trường hợp - đều là $O(nlogn)$.

- Heap Sort được xem như phiên bản cải tiến của Selection Sort vì giống nhau ở ý tưởng chọn phần tử cực trị và đưa về đúng vị trí của nó. Tuy nhiên, việc tận dụng được tính chất đặc biệt của cấu trúc dữ liệu Heap giúp chọn phần tử cực trị nhanh hơn rất nhiều so với Selection Sort.

- Heap Sort tuy trên thực tế không nhanh bằng Quick Sort và Merge Sort, nhưng nó không tốn bộ nhớ phụ như Merge Sort. Heap Sort hữu dụng nếu như ta có

nhu cầu lấy ra phần tử cực trị từ lần thứ 2 trở đi.

- Heap Sort là một thuật toán không stable do nó làm thay đổi vị trí tương đối của các phần tử trong quá trình xây dựng heap.

### 4.3.7 Merge Sort

- Merge Sort là một thuật toán sắp xếp thực hiện theo phương pháp Chia để trị - giống như Quick sort.

- Merge sort có nhiều ưu điểm, trong đó có độ phức tạp thời gian tại trường hợp tệ nhất chỉ O($n * log(n)$). Một ưu điểm khác là Merge Sort là một thuật toán sắp xếp Stable.

- Merge Sort cũng có một số hạn chế. Một trong những hạn chế lớn nhất của Merge Sort là không gian dữ liệu mà nó yêu cầu - là một thuật toán không In-place.

### 4.3.8 Quick Sort

- Nhìn chung, Quick Sort, đúng như tên của nó, là một thuật toán sắp xếp rất nhanh, với độ phức tạp thời gian trung bình là 0($nlogn$). Quick Sort sử dụng phương pháp "Chia để trị" (Divide and Conquer) để tiến hành sắp xếp.

- Quick Sort có ưu điểm khá vượt trội về mặt thời gian chạy so với các thuật toán khác. Tuy có cùng độ phức tạp thời gian là 0($nlogn$) như Merge Sort và Heap Sort, thực tế cho thấy rằng Quick Sort chạy nhanh hơn so với hai thuật toán trên.Vì lí do đó mà Quick Sort được sử dụng rộng rãi và được cài đặt thư viện sẵn trên nhiều ngôn ngữ lập trình.

- Quick Sort có một số hạn chế. Đầu tiên, đối với trường hợp xấu nhất, độ phức tạp thời gian của Quick Sort lên tới $O(n^2)$. Tuy khả năng không cao, nhưng có thể làm giảm khả năng xảy ra trường hợp tệ nhất bằng cách sử dụng phương pháp chọn pivot như "Median of three". Thứ hai, Quick Sort là một thuật toán không ổn đinh (stable) khi nó làm thay đổi vị trí tương đối của các phần tử trong mảng.

### 4.3.9 Counting Sort

- Counting sort là một thuật toán không dựa trên phép so sánh, thường được sử dụng để sắp xếp các dãy số không âm, hay bảng kí tự đã được chuyển về dạng số để sắp xếp.

- Counting sort có tốc độ tương đối nhanh hơn so với một số thuật toán khác. Counting sort sẽ thật sự hiệu quả trong trường hợp đồ dài dãy cần được sắp xếp không quá nhỏ so với phần tử lớn nhất trong dãy.

- Tuy nhiên, counting sort vẫn còn tồn tại những hạn chế, đó là tiêu thụ một lượng lớn không gian bộ nhớ để lưu trữ dữ liệu phục vụ cho việc sắp xếp, kích thước của không gian này phụ thuộc vào giá trị của phần tử lớn nhất trong dãy, điều này khiến cho counting sort không thực sự hiệu quả đối với việc sắp xếp các phần tử có giá trị quá lớn so với kích thước của dãy cần sắp xếp.

### 4.3.10 Radix Sort

- Nhìn chung, Radix sort giúp ta thực hiện việc sắp xếp tương đối nhanh, lợi thế hơn nhiều thuật toán khác.

- Vì Radix sort là một thuật toán sắp xếp dựa trên việc sắp xếp từng chữ số của mỗi phần tử, vì vậy nó sẽ hiệu quả với những phần tử có kích thước lớn, bởi phạm vi sắp xếp thực chất cũng chỉ là từ 0 đến 9.

- Một nhược điểm đáng để tâm đến của Radix sort là bởi vì việc sắp xếp của Radix sort dựa trên các chữ số hay kí tự, dẫn đến nó sẽ không thể áp dụng linh hoạt đối với nhiều kiểu dữ liệu như các thuật toán sắp xếp khác. Bên cạnh đó, cũng như các thuật toán không dựa trên sắp xếp khác, Radix sort đòi hỏi một không gian bộ nhớ phụ nhất định cho việc sắp xếp.

### 4.3.11 Flash Sort

- Nhìn chung, Flash Sort giải quyết các bài toán với tốc độ rất nhanh và vượt trội hơn với các thuật toán khác rất nhiều.
- Điểm mạnh của Flash Sort nằm ở tốc độ của nó - với độ phức tạp thời gian tốt nhất và trung bình đều nằm trong khoảng O($n$).
- Điểm hạn chế của Flash Sort là trong trường hợp tệ nhất, độ phức tạp của thuật toán là O($n^2$). Là một thuật toán nằm nhóm thuật toán không dựa vào so sánh, Flash Sort sắp xếp các phần tử trong danh sách dựa vào giả định về dữ liệu đầu vào - ở đây là yêu cầu dữ liệu có sự phân bố đồng đều.

## 4.4 Tổng quan theo nhóm thuật toán

Ta có thể chia các thuật toán được phân tích thành các nhóm thuật toán sau:

- Thuật toán sắp xếp dựa vào so sánh - không dựa vào so sánh

  **Các thuật toán sắp xếp dựa vào so sánh**

  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Shell Sort
  - Shaker Sort
  - Heap Sort
  - Merge Sort
  - Quick Sort

  **Các thuật toán sắp xếp không dựa vào so sánh**

  - Counting Sort

- Radix Sort
- Flash Sort

- Thuật toán sắp xếp In Place - không In Place

  **Các thuật toán sắp xếp In Place**

  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Shaker Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

  **Các thuật toán sắp xếp không In Place**

  - Merge Sort
  - Counitng Sort
  - Radix Sort
  - Flash Sort

### 4.4.1 Thuật toán sắp xếp dựa vào so sánh - không dựa vào so sánh

Nhìn chung trong tất cả các trật tự dữ liệu, các thuật toán sắp xếp không dựa vào so sánh biểu diễn tốt hơn - có thời gian chạy và số lượt so sánh nhỏ hơn các thuật toán dựa vào so sánh.

Với các thuật toán sắp xếp không dựa vào so sánh, điểm hạn chế của chúng thường là vì chúng có dựa vào một giả định nhất định về dữ liệu. Thuật toán Counting Sort và Radix Sort giả định rằng giữa liệu nằm trong phạm vi dữ liệu nhỏ và thuật toán Flash Sort giả định rằng giữ liệu được phân bố đều. Ngoài ra, chúng còn phụ thuộc vào không

gian lưu trữ lớn. Điểm manh của chúng nằm ở khả năng sắp xếp với độ phức tạp thời gian tốt nhất O($n$).

Với các thuật toán sắp xếp không dựa trên so sánh, điểm hạn chế của chúng là có giới hạn độ phức tạp thời gian tốt nhất là O($n * log(n)$), nên trường hợp tốt nhất của chúng không thể nhanh hơn các thuật toán không dựa vào sắp xếp có độ phức tạp thời gian nhỏ nhất là O($n$). Điểm mạnh của chúng nằm ở khả năng sắp xếp nhiều loại dữ liệu khác nhau và không yêu cầu không gian lưu trữ lớn.

Giới hạn ở đồ án nằm ở biến số dữ liệu được nghiên cứu - ở đây là thời gian chạy của thuật toán và lượt so sánh - đã không đánh giá được một cách khách quan điểm mạnh và yếu của các thuật toán, rõ ràng nhất là với nhóm thuật toán sắp xếp dựa vào so sánh và không dựa vào so sánh. Các thuật toán sắp xếp không phụ thuộc vào so sánh được kết luận tốt hơn vì:

- Chưa đánh giá được không gian lưu trữ yêu cầu bởi mỗi thuật toán so sánh
- Phân phối dữ liệu đưa ra so sánh khá đều - với các giá trị đã sắp xếp, gần sắp xếp và ngược lại thường chỉ lệch nhau 1 đơn vị và giá trị nhỏ hơn 500,000 (giới hạn của kích thước dữ liệu tìm hiểu). Đối với kiểu dữ liệu ngẫu nhiên thì độ lệch dữ liệu sẽ không lớn, và phù hợp với các thuật toán sắp xếp không phụ thuộc vào so sánh.

### 4.4.2 Thuật toán In-place - không In-place

Nhìn chung trong tất cả các trật tự dữ liệu, các thuật toán không In-Place biểu diễn tốt hơn các thuật toán Inplace.

Điều này có thể được giải thích dễ hiểu là một sử đánh đổi hiệu quả thời gian và không gian. Các thuật toán muốn có hiệu quả thời gian tốt hơn thì phải sử dụng nhiều không gian hơn, và ngược lại. Ở trong đồ án điều này được minh họa rõ với kết luận trên - các thuật toán không dựa trên so sánh sử dụng nhiều không gian dữ liệu hơn - và biểu diễn tốt hơn các thuật toán dựa vào sao sánh sử dụng ít không gian dữ liệu hơn.

# 5 TỔ CHỨC ĐỒ ÁN VÀ LƯU Ý LẬP TRÌNH

## 5.1 Tổ chức đồ án

Các file trong đồ án được tổ chức thành 4 thư mục như sau

- Thư mục Algorithm: Thư mục chứa source code của 11 thuật toán. Mỗi thuật toán được tổ chức thành một file .cpp và một file .h riêng biệt. Mỗi hàm thuật toán được chia làm hai phiên bản: Một phiên bản dùng để đo số phép so sánh và một phiên bản dùng để đo thời gian thực thi thuật toán

- Thư mục Command: Thư mục chứa source code của 5 command. Mỗi command được tổ chức trong một file.cpp và một file command.cpp chung thực hiện chức năng phân loại command dựa trên dữ liệu đầu vào.

- Helper: Thư mục chứa các hàm hỗ trợ các yêu cầu cần thiết trong quá trình thực hiện đồ án bao gồm: thao tác đọc và ghi vào file, khởi tạo dữ liệu cho việc sắp xếp, hoán vị hai phần tử, các hàm xác định thuật toán sắp xếp và các hàm xác định loại dữ liệu.

- 3ForLoop: Thư mục chứa source code của hàm chạy 3 vòng lặp for tính lượt so sánh và thời gian chạy của 4 kiểu dữ liệu và 11 thuật toán so sánh trên 6 cỡ dữ liệu: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000, bao gồm 1 file 3forloop.cpp và 1 file 3forloop.h.

- Ngoài ra, ngoài các thư mục có 1 file main.cpp để chạy các chương trình đã viết sử dụng tham số dòng lệnh để chạy các lệnh đã cho. Đồ án cũng bao gồm thêm một file 3-for-loop.exe dùng để chạy 3 vòng lặp for tính lượt so sánh và giời gian chạy của 4 kiểu dữ liệu và 11 thuật toán trên sau cỡ dữ liệu như trên.

## 5.2 Lưu ý lập trình

**Cấu trúc dữ liệu và thư viện:**

Trong đồ án sử dụng các cấu trúc dữ liệu sau: mảng động (dynamic array), hash map. Một số thư viện được sử dụng trong đồ án:

- Thư viện fstream: được dùng cho các thao tác liên quan đến file.
- Thư viện string.h: được dùng để thực hiện các phép toán liên quan đến chuỗi.
- Thư viện chrono: được dùng để đo thời gian thực thi của thuật toán.
- Thư viện unordered_map: được dùng để khởi tạo và thực thi các câu lệnh liên quan đến hash map.

**Hướng dẫn sử dụng:**

Để compile file dùng để chạy command, đồ án sử dụng g++ với cú pháp sau:

$$g++ *.cpp \ Algorithm\backslash *.cpp \ Command\backslash *.cpp \ Helper\backslash *.cpp \ -o \ main.exe$$

Trong cú pháp trên:

- *.cpp: Compile file main.cpp
- Algorithm*.cpp: Compile các file cpp trong folder algorithm - Các file hàm thuật toán sắp xếp
- Command*.cpp: Compile các file cpp trong folder Command - Các file hàm thuật toán lệnh cho tham số dòng lệnh
- Helper*.cpp: Complile các file cpp trong folder Helper
- -o main.exe: Kết quả quá trình complie các file trên sẽ được viết vào file main.exe. Ở đây người dùng có thể sử dụng tên file khác để đặt cho file exe dùng để chạy chương trình

**Thông số máy tính:**

Dữ liệu thu được từ đồ án được thực hiện trên một số thiết bị có các thông số như sau:

Thiết bị 1: Sử dụng để đo thời gian của dữ liệu ngược của tất cả các thuật toán

- Loại thiết bị: DELL Vostro 3400
- Hệ điều hành: Window 10 Home Single Langue
- Loại hệ thống: 64-bit operating system - x64-based processor
- CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
- RAM: 8GB
- Ổ cứng: SSD

Thiết bị 2: Sử dụng để đo thời gian của dữ liệu ngẫu nhiên, dữ liệu gần sắp xếp và đã sắp xết của tất cả thuật toán.

- Loại thiết bị: DELL Precision 3520
- Hệ điều hành: Window 10 Pro
- Loại hệ thống: 64-bit Operating System, x64-based processor
- CPU: Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
- RAM: 8GB
- Ổ cứng: SSD

# 6 DANH SÁCH THAM KHẢO

Nội dung đồ án được viết qua sự tham khảo các nội dung sau. Tài liệu tham khảo của từng thuật toán được đặt trong mỗi mục của thuật toán đó. Ngoài ra các tài liệu liên quan đến quá trình lập trình và dữ liệu cũng được sử dụng.
Nội dung tham khảo được trình bày theo APA phiên bản 6, gồm tổ chức/các nhân tác giả, ngày publish nội dung tham khảo, ngày tham khảo và nguồn dẫn đến nội dung tham khảo.


**Selection Sort**

1. Sort Algorithm. (2022, September 1). Retrieved November 5, 2022, from https://www.geeksforgeeks.org/selection-sort/

**Insertion Sort:**

2. Insertion Sort. (2022, October 18). Retrieved November 1, 2022, from https://www.geeksforgeeks.org/insertion-sort/

3. GeeksforGeeks. (2022a, June 20). Binary Insertion Sort. Retrieved November 2, 2022, from https://www.geeksforgeeks.org/binary-insertion-sort/

4. GeeksforGeeks. (2022b, July 19). Cocktail Sort. Retrieved November 1, 2022, from https://www.geeksforgeeks.org/cocktail-sort/

**Bubble Sort:**

5. GeeksforGeeks. (2022j, November 1). Bubble Sort Algorithm. Retrieved November 5, 2022, from https://www.geeksforgeeks.org/bubble-sort/

**Shaker Sort:**

6. Wikipedia contributors. (2022b, October 31). Cocktail shaker sort. Retrieved November 1, 2022, from https://en.wikipedia.org/wiki/Cocktail_shaker_sort

7. [CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT] – THUẬT TOÁN SẮP XẾP SHAKERSORT. (2019, February). Retrieved November 1, 2022, from https://123docz.net/document/2251215-de-tai-mang-khong-day.html

8. Sơn P. T. (2022, August 2). Bài tập thuật toán sắp xếp Shaker Sort trong C. Retrieved November 1, 2022, from https://laptrinhtudau.com/bai-tap-thuat-toan-sap-xep-shaker-sort-trong-c/

**Shell Sort:**

9. GeeksforGeeks. (2022i, October 24). ShellSort. Retrieved October 28, 2022, from https://www.geeksforgeeks.org/shellsort/

10. RobEdwards. (2016, December 9). Sorts 5 Shell Sort. Retrieved October 28, 2022, from https://www.youtube.com/watch?v=ddeLSDsYVp8

Tham khảo thêm về độ phức tạp của thuật toán và các biến thể:

11. Shell Sort. (2022, May 21). Retrieved October 28, 2022, from https://codeahoy.com/learn/sortingalgorithmsa/shellsort/

**Heap Sort:**

12. GeeksforGeeks. (2022d, September 22). Heap Sort. Retrieved October 23, 2022, from https://www.geeksforgeeks.org/heap-sort/

13. Thuật toán sắp xếp vun đống - Heap Sort Algorithm C/C++. (2021, May 7). Retrieved November 6, 2022, from https://duongdinh24.com/thuat-toan-heap-sort/

**Merge Sort:**

14. GeeksforGeeks. (2022e, September 23). Merge Sort Algorithm. Retrieved November 5, 2022, from https://www.geeksforgeeks.org/merge-sort/

**Quick Sort:**

15. GeeksforGeeks. (2022h, September 27). QuickSort. Retrieved October 23, 2022, from https://www.geeksforgeeks.org/quick-sort/

16. GeeksforGeeks. (2022f, September 26). Hoare's vs Lomuto partition scheme in QuickSort. Retrieved October 23, 2022, from https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/

**Counting Sort:**

17. C++ Program to Implement Counting Sort. (n.d.). Retrieved November 2, 2022, from https://www.tutorialspoint.com/cplusplus-program-to-implement-counting-sort

18. CS Dojo. (2017, March 22). Learn Counting Sort Algorithm in LESS THAN 6 MINUTES! Retrieved November 2, 2022, from https://www.youtube.com/watch?v=OKd534EWcdk

19. GeeksforGeeks. (2022g, September 27). Counting Sort. Retrieved November 2, 2022, from https://www.geeksforgeeks.org/counting-sort/

**Radix Sort:**

20. Dey, M. (2021, July 22). Time and Space complexity of Radix Sort. Retrieved November 5, 2022, from https://iq.opengenus.org/time-and-space-complexity-of-radix-sort/

21. GeeksforGeeks. (2022c, August 9). Radix Sort. Retrieved November 5, 2022, from https://www.geeksforgeeks.org/radix-sort/

**Flash Sort:**

22. Flash Sort - Thuật Toán Sắp Xếp Thần Thánh. (n.d.). Retrieved October 29, 2022, from https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh

23. Hawkular. (2015, March 24). Flash sort- a sorting algorithm. Retrieved October 29, 2022, from https://www.youtube.com/watch?v=kdhPnSIT7dY

24. Lập trình cùng Eric. (2021, March 26). Flash Sort thuật toán sắp xếp với độ phức tạp O(n) ? Retrieved October 29, 2022,
from https://www.youtube.com/watch?v=CAaDJJUszvE

25. Flashsort. (2022, June 29). Retrieved October 29, 2022, from
https://en.wikipedia.org/wiki/Flashsort

**Độ phức tạp thuật toán:**

26. Sorting algorithm. (2022, October 26). Retrieved October 31, 2022, from https://en.wikipedia.org/wiki/Sorting_algorithm

**Tham số dòng lệnh:**

27. GeeksforGeeks. (2022b, July 8). Command line arguments in C/C++. Retrieved October 25, 2022, from https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/