**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**KHOA CÔNG NGHỆ THÔNG TIN**


# BÁO CÁO LAB 3

## SORTING

**NHÓM 13**

**Giảng viên lý thuyết: ThS. Văn Chí Nam**

**Giảng viên hướng dẫn thực hành: Phan Thị Phương Uyên**

**Môn học: THỰC HÀNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**THÀNH PHỐ HỒ CHÍ MINH – 2021**

# THÔNG TIN NHÓM

## Nhóm 13

| STT | MSSV | Họ và tên |
|---|---|---|
| 1 | 20120352 | Vũ Hoàng Phúc |
| 2 | 20120373 | Lê Trương Kinh Thành |
| 3 | 20120376 | Trần Văn Thật |
| 4 | 20120380 | Nguyễn Phúc Thuần |

# MỤC LỤC

# PHẦN 1: MỞ ĐẦU, GIỚI THIỆU

## 1.1. Giới thiệu

- Các thuật toán thực hiện: Set 2(11 thuật toán): Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort.
- Ngôn ngữ thực hiện: C++

## 1.2. Tổng quan:

Thông tin mỗi thuật toán:

- Ý tưởng
- Các bước thực hiện
- Độ phức tạp
- Cải tiến thuật toán
- Kết quả thực nghiệm
- Nhận xét

## 1.3. Người thực hiện thuật toán

| STT | Thuật toán | Người thực hiện |
|---|---|---|
| 1 | Selection Sort | Trần Văn Thật |
| 2 | Insertion Sort | Nguyễn Phúc Thuần |
| 3 | Bubble Sort | Nguyễn Phúc Thuần |
| 4 | Shaker Sort | Lê Trương Kinh Thành |
| 5 | Shell Sort | Vũ Hoàng Phúc |
| 6 | Heap Sort | Trần Văn Thật |
| 7 | Merge Sort | Trần Văn Thật |
| 8 | Quick Sort | Lê Trương Kinh Thành |
| 9 | Counting Sort | Lê Trương Kinh Thành |
| 10 | Radix Sort | Nguyễn Phúc Thuần |
| 11 | Flash Sort | Vũ Hoàng Phúc |

# PHẦN 2: THUẬT TOÁN

## 2.1. Selection Sort

*** Ý tưởng:** Thuật toán này sắp xếp một mảng bằng cách tìm phần tử có giá trị nhỏ nhất trong đoạn chưa được sắp xếp và hoán đổi cho phần tử ở đầu mảng chưa được sắp xếp.

Thuật toán sẽ chia thành 2 mảng con: mảng đã được sắp xếp và chưa được sắp xếp.

Ở mỗi bước lặp của thuật toán, phần tử nhỏ nhất của mảng con chưa được sắp xếp sẽ được di chuyển về mảng con đã được sắp xếp.

*** Mô tả thuật toán.**

Bước 1: i := 0, min_index := i  (min_index: vị trí của phần tử nhỏ nhất của mảng)

Bước 2: j := i + 1.

Bước 3: Nếu a[j] < a[j+1] thì min_index := j và hoán vị a[min_index] và a[i]

Bước 4: j := j +1.

Bước 5: Nếu j < n thì quay lại bước 3, ngược lại thì chuyển sang bước 6.

Bước 6: i := i + 1.

Bước 7: Nếu i = n – 1 thì kết thúc, ngược lại quay về bước 2.

*** Minh hoạ các bước:**

Giả sử ta có mảng sau: 15 38 17 82 9, n = 5.

**Lượt 1:** i = 0, min_index = 0, mảng là: 15 38 17 82 9

j = 1, kiểm tra a[j] < a[min_index] (a[1] < a[0]: sai)

j = 2, kiểm tra a[j] < a[min_index] (a[2] < a[0]: sai)

j = 3, kiểm tra a[j] < a[min_index] (a[3] < a[0]: sai)

j = 4, kiểm tra a[j] < a[min_index] (a[4] < a[0]: đúng), min_index = 4.

j = n = 5, thoát vòng lặp.

Hoán vị a[4] và a[0] → Mảng là: **9** 38 17 82 **15.**

**Lượt 2:** i = 1, min_index = 1, mảng là: 9 38 17 82 15

j = 2, kiểm tra a[j] < a[min_index] (a[2] < a[1]: đúng), min_index = 2.

j = 3, kiểm tra a[j] < a[min_index] (a[3] < a[2]: sai)

j = 4, kiểm tra a[j] < a[min_index] (a[4] < a[2]: đúng), min_index = 4.

j = n = 5, thoát vòng lặp.

Hoán vị a[4] và a[1] → Mảng là: 9 **15** 17 82 **38**.

**Lượt 3:** i = 2, min_index = 2, mảng là: 9 15 17 82 38

j = 3, kiểm tra a[j] < a[min_index] (a[3] < a[2]: sai)

j = 4, kiểm tra a[j] < a[min_index] (a[4] < a[2]: sai)

j = n = 5, thoát vòng lặp.

Hoán vị a[2] và a[2] (giữ nguyên mảng cũ) → Mảng là: 9 15 **17** 82 38.

**Lượt 4:** i = 3, min_index = 3, mảng là: 9 15 17 82 38

j = 4, kiểm tra a[j] < a[min_index] (a[4] < a[3]: đúng), min_index = 4.

j = n = 5, thoát vòng lặp.

Hoán vị a[4] và a[3] → Mảng là: 9 15 17 **38 82**.

**Lượt 5:** i = n – 1 = 4, kết thúc.

Mảng đã sắp xếp là: 9 15 17 38 82

*** Đánh giá độ phức tạp của thuật toán:**

+ Trường hợp tốt: $O(n^2)$

+ Trung bình: $O(n^2)$

+ Trường hợp xấu: $O(n^2)$

## 2.2. Insertion Sort

**Ý tưởng:** Thuật toán sắp xếp chèn thực hiện sắp xếp dãy số theo cách duyệt từng phần tử và chèn từng phần tử đó vào đúng vị trí trong mảng con (dãy số từ đầu đến phần tử phía trước nó) đã sắp xếp sao cho dãy số trong mảng sắp đã xếp đó vẫn đảm bảo tính chất của một dãy số tăng dần.

**Thuật toán:**

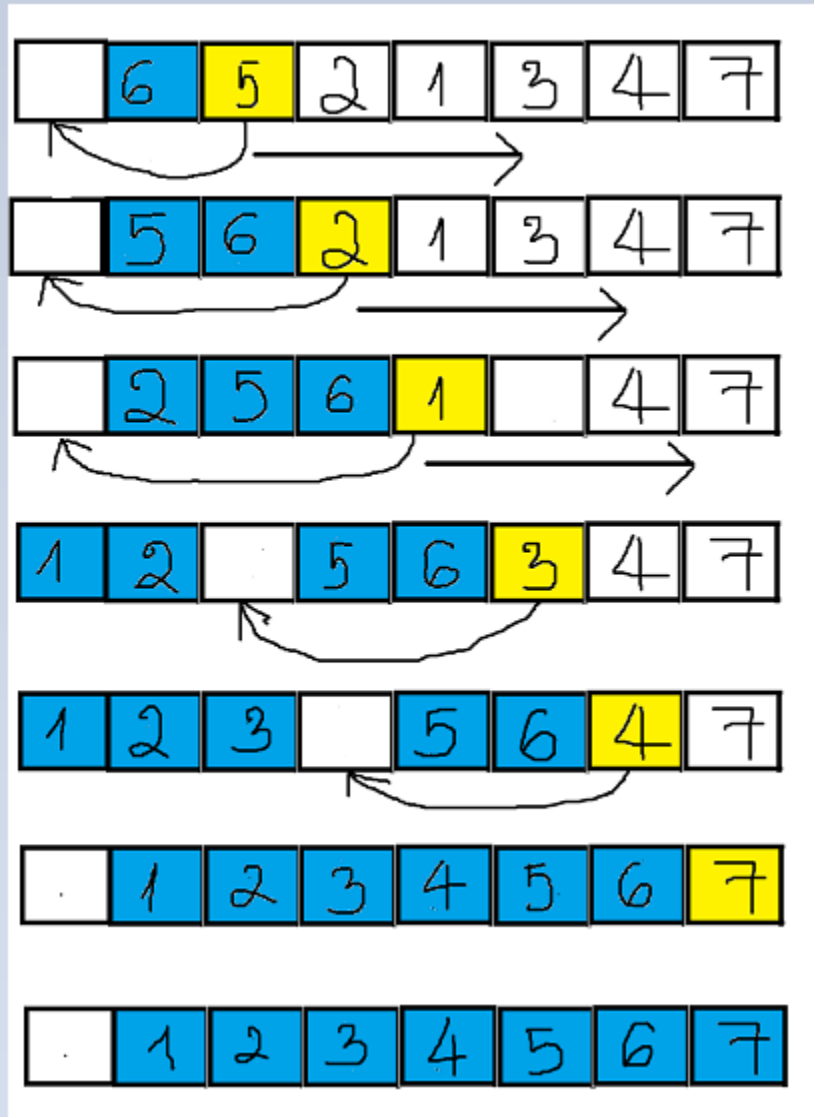
B1: Khởi tạo mảng với dãy con đã sắp xếp có k = 1 phần tử (phần tử đầu tiên, phần tử có chỉ số 0)

B2: Duyệt từng phần tử từ phần tử thứ 2, tại mỗi lần duyệt phần tử ở chỉ số i thì đặt phần tử đó vào một vị trí nào đó trong đoạn từ [0…i] sao cho dãy số từ [0…i] vẫn đảm bảo tính chất dãy số tăng dần. Sau mỗi lần duyệt, số phần tử đã được sắp xếp k trong mảng tăng thêm 1 phần tử.

B3: Lặp B2 với i := i + 1 cho tới khi duyệt hết tất cả các phần tử của mảng.

**Minh họa các bước:**

Giả sử ta có mảng sau: 6 5 2 1 3 4 7



**Đánh giá độ phức tạp về thời gian của thuật toán:**
Best case: O(n), 3n – 2 phép so sánh
Average case: O($n^2$), tối đa $n^2$ – 2 phép so sánh
Worst case: O($n^2$), $n^2$ – 2 phép so sánh

## 2.3 Bubble Sort

**Ý tưởng:** Thuật toán sắp xếp nổi bọt thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự (số sau bé hơn số trước với trường hợp sắp xếp tăng dần) cho đến khi dãy số được sắp xếp. Các số nhỏ sẽ dần nổi lên đầu dãy, các số lớn sẽ dần chìm xuống cuối dãy

**Thuật toán:**
B1: i := 0, haveSwap := false
B2: j := 0

B3: nếu a[j] > a[j+1] thì hoán vị chúng, haveSwap = true
B4: j := j + 1
B5: nếu j = n - i - 1 thì sang bước 6, ngược lại quay về bước 3
B6: nếu haveSwap = false thì kết thúc, ngược lại sang bước 7
B7: i := i + 1
B8: Nếu i = n – 1 thì kết thúc, ngược lại quay về bước 2

**Minh họa các bước**

Giả sử ta có mảng sau: 6 5 2 1 3 4 7
Lượt thứ 1: 6 5 2 1 3 4 7
6 5 2 1 3 4 7
5 6 2 1 3 4 7
5 2 6 1 3 4 7
5 2 1 6 3 4 7
5 2 1 3 6 4 7
5 2 1 3 4 6 7
5 2 1 3 4 6 7
Lượt thứ 2: 5 2 1 3 4 6 7
5 2 1 3 4 6 7
2 5 1 3 4 6 7
2 1 5 3 4 6 7
2 1 3 5 4 6 7
2 1 3 4 5 6 7
2 1 3 4 5 6 7
Lượt thứ 3: 2 1 3 4 5 6 7
2 1 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
Lượt thứ 4: 1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
Do không có sự thay đổi nào nên quá trình sắp xếp kết thúc

**Đánh giá độ phức tạp về thời gian của thuật toán:**

Best case: O(n), không có sự thay đổi nào xảy ra, số phép so sánh là 2n + 2
Average case: $O(n^2)$, số so sánh nhỏ hơn $n.(n+1)-2+n$
Worst case: $O(n^2)$, số so sánh là $n.(n+1)-2+n$

## 2.4. Shaker Sort

**Idea**

Sau khi đưa phần tử nhỏ nhất về đầu mảng sẽ đưa phần tử lớn nhất về cuối dãy. Do đưa các phần tử về đúng vị trí ở cả hai đầu nên Shaker Sort sẽ giúp cải thiện thời gian sắp xếp dãy số do giảm được độ lớn của mảng đang xét ở lần so sánh kế tiếp.

**Giải thuật**

Bước 1: $l = 1$; $r = n$; // từ $l$ đến $r$ là đoạn cần sắp xếp

$k = n$; // ghi nhận vị trí k xảy ra hoán vị sau cùng

// để làm cơ sở thu hẹp đoạn $l$ đến $r$.

Bước 2:

Bước 2a: $j = r$; // đẩy phần tử nhỏ về đầu mảng

Trong khi ($j > l$) thực hiện

Nếu $a[j] < a[j\text{-}1]$: $a[j] \leftrightarrow a[j\text{-}1]$;

$k = j$; // lưu lại nơi xảy ra hoán vị.

$j = j - 1$;

$l = k$; // loại các phần tử đã có thứ tự ở đầu dãy

Bước 2b: $j = l$; // đẩy phần tử lớn về cuối mảng

Trong khi ($j < r$) thực hiện

Nếu $a[j] > a[j\text{-}1]$: $a[j] \leftrightarrow a[j\text{+}1]$;

$k = j$; // lưu lại nơi xảy ra hoán vị.

$j = j + 1$;

$r = k$; // loại các phần tử đã có thứ tự ở cuối dãy

Bước 3: Nếu $l <$ r : Lặp lại bước 2.

Ngược lại: Dừng.

## Ví dụ


- **l = 1, r = 8**

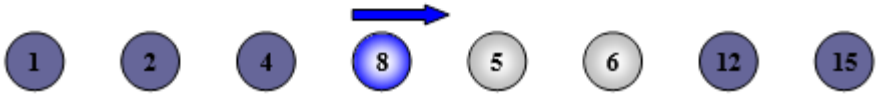
Lượt đi


Lượt về : cập nhật l = 2


- **l = 2, r = 6**

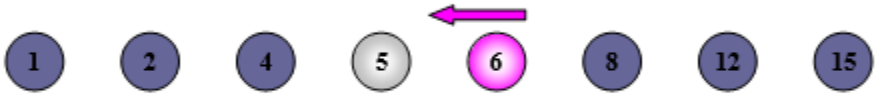
Lượt đi


Lượt về : cập nhật l = 4

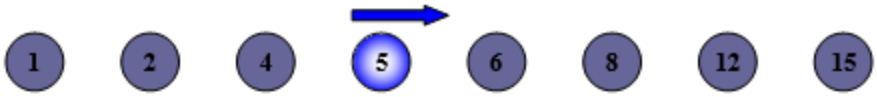


- **l = 4, r = 5**

Lượt đi : **không đổi**




Lượt về : **không đổi**




**Kết quả**


**Độ phức tạp**

Best case: O(n).

Worse case: O(n).

Average case: O(n).

## 2.5 Shell Sort

***Ý tưởng:**

-Shell Sort là một thuật toán sắp xếp mang lại hiệu quả cao dựa trên thuật toán Insertion Sort. Thuật toán này tránh các trường hợp phải tráo đổi vị trí của hai phần tử xa nhau trong giải thuật sắp xếp chọn (nếu như phần tử nhỏ hơn ở vị trí bên phải khá xa so với phần tử lớn hơn bên trái)

-Thuật toán này sử dụng giải thuật sắp xếp chọn trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách gần hơn. Khoảng cách này còn được gọi là khoảng(interval). Interval sẽ nhận các giá trị lần lượt là $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}$ cho đến khi interval=1

*** Mô tả thuật toán:**

Bước 1:Khởi tạo giá trị của interval

Bước 2:Chia list thành các sublist nhỏ hơn tương ứng với interval

Bước 3:Sắp xếp các sublist bằng thuật toán Insertion Sort

Bước 4:Lặp lại cho đến khi list được sắp xếp

*** Minh hoạ các bước:**

Giả sử ta có mảng gồm các giá trị:

| 4 | 2 | 1 | 8 | 6 | 3 | 7 | 5 |
|---|---|---|---|---|---|---|---|

Ban đầu ta có:Interval= $\frac{8}{2} = 4$. Ta có các cặp giá trị: {4,6},{2,3},{1,7},{8,5}

| 4 | 2 | 1 | 8 | 6 | 3 | 7 | 5 |
|---|---|---|---|---|---|---|---|

So sánh các giá trị trong từng danh sách con với nhau và tráo đổi chúng(nếu cần). Sau bước này, ta thu được mảng mới:

| 4 | 2 | 1 | 5 | 6 | 3 | 7 | 8 |
|---|---|---|---|---|---|---|---|

Sau đó, ta lấy Interval= $\frac{8}{4} = 2$. Ta có hai danh sách con:{4,1,6,7},{2,5,3,8}

| 4 | 2 | 1 | 5 | 6 | 3 | 7 | 8 |
|---|---|---|---|---|---|---|---|

Tiếp tục so sánh các giá trị trong từng danh sách con với nhau và tráo đổi chúng(nếu cần). Sau bước này, ta thu được mảng mới:

| 1 | 2 | 4 | 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

Cuối cùng với Interval=1 ta sử dụng Insertion Sort để sắp xếp mảng. Dưới đây là hình minh họa cho từng bước

| 1 | 2 | 4 | 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 4 | 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 4 | 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 4 | 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 4 | 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

Mảng sau khi sắp xếp là:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

*** Đánh giá độ phức tạp của thuật toán:**

+ Trường hợp tốt: O(n)

+ Trung bình: O(n)

+ Trường hợp xấu: O(n)

## 2.6 Heap Sort

*** Ý tưởng:**

Thuật toán Heap sort là một kỹ thuật sắp xếp dựa trên so sánh dựa trên cấu trúc cây nhị phân. Chúng ta tìm phần tử lớn nhất và đặt phần tử lớn nhất ở cuối cây nhị phân sau đó lặp lại quá trình tương tự cho các phần tử còn lại.

*** Mô tả thuật toán.**

Bước 1: i = n / 2 – 1

Bước 2: Heapify cho mảng a: heapify(a, n, i)

Bước 3: i := i – 1

Bước 4: Nếu i >= 0 thì quay lại bước 2, ngược lại chuyển đến bước 5.

Bước 5: j = n – 1

Bước 6: Hoán đổi a[0] và a[j]

Bước 7: Gọi hàm heapify: heapify(a, j, 0)

Bước 8: j := j – 1

Bước 9: Nếu j >= 0 thì quay lại bước 6, ngược lại thì kết thúc.

*** Mô tả hàm heapify(a, n, i)**

Bước 1: largest = i, left = 2 * i + 1, right = 2 * i + 2.

Bước 2: Nếu left < n và a[left] > a[largest] thì largest = left, ngược lại chuyển đến bước 3. (Con bên trái lớn hơn gốc)

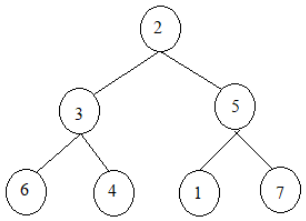
Bước 3: Nếu right < n và a[right] > a[largest] thì largest = right, ngược lại chuyển đến bước 4. (Con bên phải lớn hơn gốc)

Bước 4: Nếu largest ≠ i thì hoán vị a[i] và a[largest], đệ quy lại hàm heapify(a, n, largest).

*** Minh hoạ các bước:**

Giả sử ta có mảng sau: 2 3 5 6 4 1 7, n = 7.

Ta biểu diễn thành cây như sau:




Bước 1: i = n / 2 – 1 = 2.

Bước 2: Heapify cho mảng a: heapify(a, 7, 2)

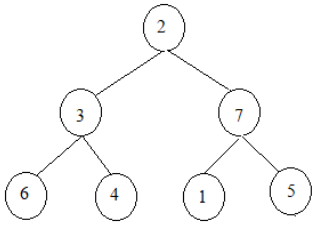
Bước 2.1: largest = 2, left = 5, right = 6

Bước 2.2: Nếu left < n và a[left] > a[largest] (5 < 7 và 1 > 5: sai)

Bước 2.3: Nếu right < n và a[right] > a[largest] (6 < 7 và 7 > 5: đúng) thì largest = 6.

Bước 2.4: Nếu largest ≠ i (6 ≠ 2: đúng) thì hoán vị a[2] và a[6].

Khi đó ta có mảng: 2 3 7 6 4 1 5.




Đệ quy lại hàm heapify(a, 7, 6).

Bước 2.1: largest = 6, left = 13, right = 14

Bước 2.2: Nếu left < n và a[left] > a[largest] (13 < 7: sai)

Bước 2.3: Nếu right < n và a[right] > a[largest] (14 < 7: sai)

Bước 2.4: Nếu largest ≠ i sai, thoát hàm.

Bước 3: i = i – 1 = 1, quay lại bước 2.

Bước 2: Heapify cho mảng a: heapify(a, 7, 1)

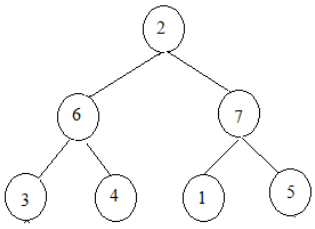
Bước 2.1: largest = 1, left = 3, right = 4

Bước 2.2: Nếu left < n và a[left] > a[largest] (3 < 7 và 6 > 3: đúng) thì largest = 3.

Bước 2.3: Nếu right < n và a[right] > a[largest] (3 < 7 và 4 > 6: sai)

Bước 2.4: Nếu largest ≠ i (3 ≠ 1: đúng) thì hoán vị a[3] và a[1].

Khi đó ta có mảng: 2 6 7 3 4 1 5




Đệ quy lại hàm heapify(a, 7, 3).

Bước 2.1: largest = 3, left = 7, right = 8

Bước 2.2: Nếu left < n và a[left] > a[largest] (7 < 7: sai)

Bước 2.3: Nếu right < n và a[right] > a[largest] (8 < 7: sai)

Bước 2.4: Nếu largest ≠ i sai, thoát hàm.

Bước 3: i = i – 1 = 0, quay lại bước 2.

Bước 2: Heapify cho mảng a: heapify(a, 7, 0)

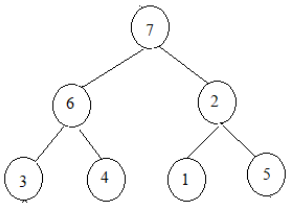
Bước 2.1: largest = 0, left = 1, right = 2

Bước 2.2: Nếu left < n và a[left] > a[largest] (1 < 7 và 6 > 2: đúng) thì largest = 1.

Bước 2.3: Nếu right < n và a[right] > a[largest] (2 < 7 và 7 > 6: đúng) thì largest = 2.

Bước 2.4: Nếu largest ≠ i (2 ≠ 0: đúng) thì hoán vị a[2] và a[0].

Khi đó ta có mảng: 7 6 2 3 4 1 5




Đệ quy lại hàm heapify(a, 7, 2).

Bước 2.1: largest = 2, left = 5, right = 6

Bước 2.2: Nếu left < n và a[left] > a[largest] (5 < 7 và 1 > 2: sai)

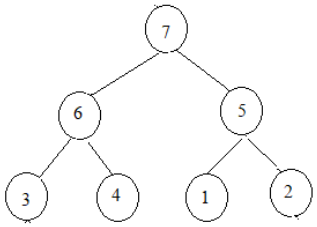
Bước 2.3: Nếu right < n và a[right] > a[largest] (6 < 7 và 5 > 2: đúng) thì largest = 6.

Bước 2.4: Nếu largest ≠ i (6 ≠ 2: đúng) thì hoán vị a[2] và a[6].

Khi đó ta có mảng: 7 6 5 3 4 1 5.

Đệ quy lại hàm heapify(a, 7, 6).

Bước 2.1: largest = 6, left = 13, right = 14




Bước 2.2: Nếu left < n và a[left] > a[largest] (13 < 7: sai)

Bước 2.3: Nếu right < n và a[right] > a[largest] (14 < 7: sai)

Bước 2.4: Nếu largest ≠ i sai, thoát hàm.

Bước 3: i = i – 1 = – 1 sai chuyển bước 5.

Bước 5: j = n – 1 = 6.

Bước 6: Hoán đổi a[0] và a[6] ta có mảng: 2 6 5 3 4 1 7

Bước 7: heapify(a, 6, 0) ta có mảng: 6 4 5 3 2 1 7

Bước 8: j = j – 1 = 5, qua lại bước 6.

Bước 6: Hoán đổi a[0] và a[5] ta có mảng: 1 4 5 3 2 6 7

Bước 7: heapify(a, 5, 0) ta có mảng: 5 4 1 3 2 6 7

Bước 8: j = j – 1 = 4, qua lại bước 6.

Bước 6: Hoán đổi a[0] và a[4] ta có mảng: 2 4 1 3 5 6 7

Bước 7: heapify(a, 4, 0) ta có mảng: 4 3 1 2 5 6 7

Bước 8: j = j – 1 = 3, qua lại bước 6.

Bước 6: Hoán đổi a[0] và a[3] ta có mảng: 2 3 1 4 5 6 7

Bước 7: heapify(a, 3, 0) ta có mảng: 3 2 1 4 5 6 7

Bước 8: j = j – 1 = 2, qua lại bước 6.

Bước 6: Hoán đổi a[0] và a[2] ta có mảng: 1 2 3 4 5 6 7

Bước 7: heapify(a, 2, 0) ta có mảng: 2 1 3 4 5 6 7

Bước 8: j = j – 1 = 1, qua lại bước 6.

Bước 6: Hoán đổi a[0] và a[1] ta có mảng: 1 2 3 4 5 6 7

Bước 7: heapify(a, 1, 0) ta có mảng: 1 2 3 4 5 6 7.

Bước 8: j = j – 1 = 0.

Ta có mảng đã sắp xếp là: **1 2 3 4 5 6 7**.

*** Đánh giá độ phức tạp của thuật toán:**

+ Trường hợp tốt: O(nlog(n))

+ Trung bình: O(nlog(n))

+ Trường hợp xấu: O(nlog(n))

## 2.7 Merge Sort

*** Ý tưởng:** Merge sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia (đệ quy). Sau cùng gộp các nửa đó thành mảng đã sắp xếp, tạo một hàm để gộp hai nửa mảng lại với nhau.

*** Mô tả thuật toán.**

Bước 1: mid_index = left + (right – left) / 2. (vị trí xem như là giữa mảng)

Bước 2: Gọi đệ quy merge sort cho nửa mảng bên trái (từ vị trí left đến mid_index)

Bước 3: Gọi đệ quy merge sort cho nửa mảng bên phải (từ vị trí mid_index + 1 đến right)

Bước 4: Gộp hai nửa mảng đã sắp xếp lại với nhau bằng một hàm mới.

* Hàm gộp hai nửa mảng

Bước 1: n_left := mid_index – left + 1 và n_right = right – mid_index

Bước 2: Khởi tạo hai mảng tạm Left[n_left] và Right[n_right]

Bước 3: Copy dữ liệu sang mảng tạm

Bước 3.1: i = 0.

Bước 3.2: Left[i] := a[left + i]

Bước 3.3: i := i + 1

Bước 3.4: Nếu i < n_left, quay lại bước 3.1, ngược lại chuyển sang bước 3.5.

Bước 3.5: j = 0.

Bước 3.6: Right[j] := a[mid_index + j]

Bước 3.3: j := j+ 1

Bước 3.4: Nếu i < n_right, quay lại bước 3.5, ngược lại chuyển sang bước 4.

Bước 4: i := 0, j := 0, k := left (k: khởi tạo chỉ số bắt đầu mảng lưu kết quả)

Bước 5: Nếu i < n_left và j < n_right thì chuyển đế bước 6, ngược lại, chuyển sang bước 8.

Bước 6: Nếu Left[i] <= Right[j] thì a[k] = Left[i], i++, ngược lại thì a[k] = Right[j], j++.

Bước 7: k := k + 1.

Bước 8: Copy các phần tử còn lại của mảng Left vào mảng kết quả nếu có.

Bước 8.1: Nếu i < n_left thì chuyển tới bước 8.2, ngược lại chuyển sang bước 9.

Bước 8.2: a[k] = Left[i]

Bước 8.3: i := i + 1, k := k + 1

Bước 9: Copy các phần tử còn lại của mảng Right vào mảng kết quả nếu có.

Bước 8.1: Nếu j < n_right thì chuyển tới bước 9.2, ngược lại kết thúc hàm.

Bước 8.2: a[k] = Right[j]

Bước 8.3: j := j + 1, k := k + 1

*** Minh hoạ các bước:**

Giả sử ta có mảng sau: 1 23 3 5 100 54

Lượt 1: 1 23 3 – 5 100 54

Lượt 2: 1 – 23 3 – 5 – 100 54

Lượt 3: 1 – 23 – 3  5 – 100 – 54

Lượt 4: 1 3 23 – 5 54 100

Lượt 5: 1 3 5 23 54 100

Mảng sau khi đã xếp là: 1 3 5 23 54 100

*** Đánh giá độ phức tạp của thuật toán:**

+ Trường hợp tốt: O(nlog(n))

+ Trung bình: O(nlog(n))

+ Trường hợp xấu: O(nlog(n))

## 2.8 Quick Sort

**Ý tưởng**

Thuật toán sắp xếp quick sort là một thuật toán chia để trị( Divide and Conquer algorithm). Nó chọn một phần tử trong mảng làm điểm đánh dấu(pivot). Thuật toán sẽ thực hiện chia mảng thành các mảng con dựa vào pivot đã chọn. Việc lựa chọn pivot ảnh hưởng rất nhiều tới tốc độ sắp xếp. Nhưng máy tính lại không thể biết khi nào thì nên chọn theo cách nào. Dưới đây là một số cách để chọn pivot thường được sử dụng:

Luôn chọn phần tử đầu tiên của mảng.

Luôn chọn phần tử cuối cùng của mảng

Chọn một phần tử random.

Chọn một phần tử có giá trị nằm giữa mảng(median element).

**Giải thuật**

Đặt pivot là phần tử cuối cùng của dãy số arr. Chúng ta bắt đầu từ phần tử trái nhất của dãy số có chỉ số là left, và phần tử phải nhất của dãy số có chỉ số là right -1(bỏ qua phần tử pivot). Chừng nào left < right mà arr[left] > pivot và arr[right] < pivot thì đổi chỗ hai phần tử left và right. Sau cùng, ta đổi chỗ hai phần tử left và pivot cho nhau. Khi đó, phần tử left đã đứng đúng vị trí và chia dãy số làm đôi(bên trái và bên phải)

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

pivot = 6, left = 0, right = 5

arr[left] = 10 < arr[pivot] = 70 và left <= right, left = 1
arr[left] = 80 > arr[pivot] = 70, tạm dừng

arr[right] = 50 < arr[pivot] = 70, tạm dừng

Do left < right, đổi chỗ arr[left], arr[right]
arr[] = {10, 50, 30, 90, 40, 80, 70}
left = 2, right = 4

arr[left] = 30 < arr[pivot] = 70 và left <= right, left = 3
arr[left] = 90 > arr[pivot] = 70, tạm dừng

arr[right] = 40 < arr[pivot] = 70, tạm dừng

Do left < right, đổi chỗ arr[left], arr[right]
arr[] = {10, 50, 30, 40, 90, 80, 70}
left = 4, right = 3

// Do left >= right
arr[] = {10, 50, 30, 40, 70, 80, 90}.  // Đổi chỗ arr[left] và arr[pivot]

Bây giờ, 70 đã nằm đúng vị trí, các phần từ <= 70 nằm phía trước và lớn hơn 70 nằm
phía sau.
```

Khi đó ta sẽ có 2 mảng con: mảng bên trái của pivot và mảng bên phải của pivot. Tiếp tục công việc với mỗi mảng con(chọn pivot, phân đoạn) cho tới khi mảng được sắp xếp.

**Độ phức tạp**

Best case: O(nlog(n)).

Worse case: O(nlog(n)).).

Average case: O(n^2).

Không gian bộ nhớ sử dụng: O(log(n))

## 2.9 Couting Sort

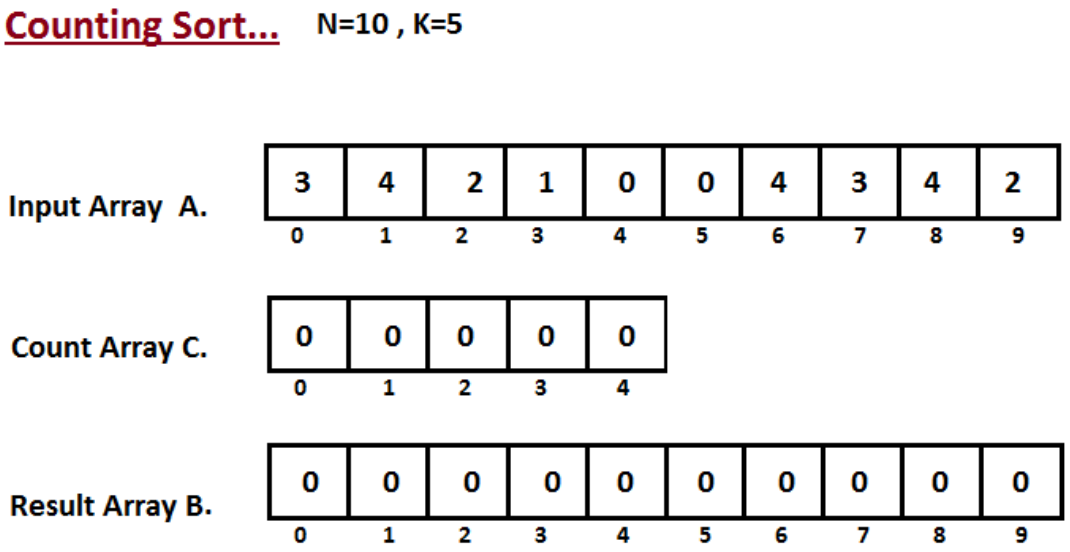
### Ý tưởng

**Counting Sort** là một kỹ thuật sắp xếp dựa trên các khóa giữa một phạm vi cụ thể. Nó hoạt động bằng cách đếm số lượng các đối tượng có các giá trị khóa riêng biệt (loại băm). Sau đó, thực hiện một số phép tính để tính toán vị trí của mỗi đối tượng trong chuỗi đầu ra.

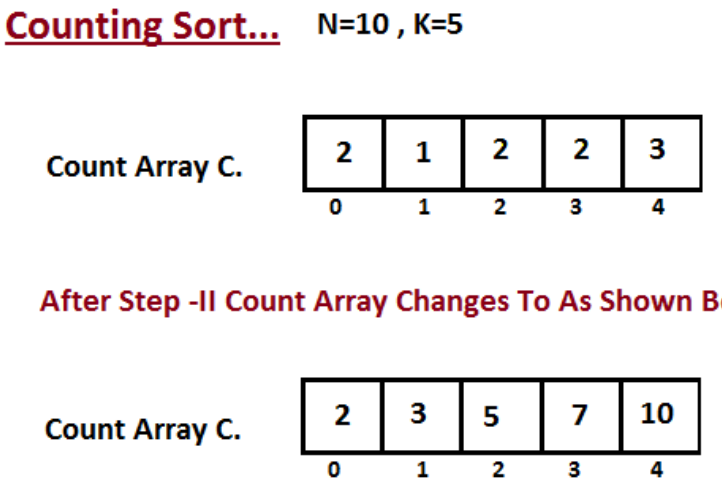
### Giải thuật

**Bước 1:**

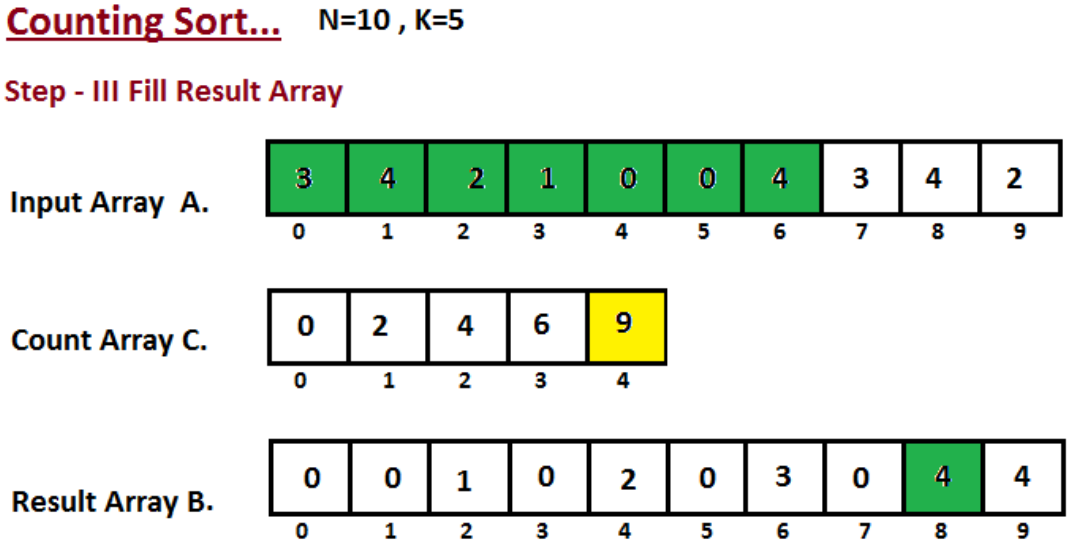
Trong bước đầu tiên, chúng tôi đếm số lần xuất hiện của từng phần tử trong mảng cần sắp xếp A. Kết quả được lưu vào mảng C.




**Bước 2:** Ở bước này, chúng ta cần xem xét sửa đổi giá trị của C. C[i] thể hiện giới hạn trên của chỉ số của phần tử i sau khi sắp xếp.




**Bước 3:** Duyệt qua từng phần tử của A và đặt nó vào đúng chỉ số của mảng chứa các giá trị đã sắp xếp B dựa vào C.




**Độ phức tạp**

**Độ phức tạp thời gian:** O (n + k) với n là số phần tử trong mảng đầu vào và k là phạm vi đầu vào.

**Không gian phụ trợ:** O (n + k)

## 2.10 Radix Sort

**Ý tưởng:** nếu một dãy số đã được sắp xếp hoàn chỉnh thì từng chữ số cũng sẽ được sắp xếp hoàn chỉnh dựa trên giá trị của các chữ số đó

**Thuật toán:** với k cho biết bậc của chữ số dùng để phân loại hiện tại, m là số chữ số của số lớn nhất của mảng.

Bước 1:

k = 1; (k = 1: hàng đơn vị; k = 10: hàng chục,…)

Bước 2: Tạo các bucket (ví dụ như các Queue) chứa các loại phần tử khác nhau

Khởi tạo 10 bucket $B_0, B_1, .., B_9$ rỗng; các bucket $B_i$ chứa các số chứa chữ số thứ $\log_{10} k$ là i

Bước 3:

For i = 0 to n-1 do

Đặt $a_i$ vào bucket $B_t$ với t = chữ số thứ $\log_{10} k$ của $a_i$;

Bước 4: Nối $B_0$, $B_1$, .. , $B_9$ lại (theo đúng trình tự) thành mảng a.

Bước 5:

k = k*10;

Nếu $\log_{10} k < m$ thì trở lại bước 2.

Ngược lại: Kết thúc

**Minh hoạ các bước:**

Giả sử ta có mảng sau: m = 4; 34, 6, 22, 56, 12, 41, 77, 653, 7643, 3424, 180

- Với k = 1:

| Bucket | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Stored Value | 180 | 41 | 22, 12, | 653, 7643 | 34,3424 | x | 6,56 | 77 | x | x |

Mảng a: 180, 41, 22, 12, 653, 7643, 34, 3424, 06, 56, 77

- Với k = 10:

| Bucket | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Stored Value | 6 | 12 | 22,3424 | 34,34 | 41,7643, | 653,56 | X | 77 | 180 | x |

Mảng a: 006, 012, 022, 3424, 034, 041, 7643, 653, 056, 077, 180

- Với k = 100

| Bucket | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Stored Value | 6,12,22,34,41,56,77 | 180 | X | X | 3424 | X | 7643,653 | X | X | X |

Mảng a: 6, 12, 22, 34, 41, 56, 77, 180, 3424, 7643, 653

- Với k = 1000

| Bucket | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Stored Value | 6,12,22,34,41,56,77,180,653 | X | X | 3424 | X | X | X | 7643 | X | X |

Mảng a: 6, 12, 22, 34, 41, 56, 77, 180, 653, 3424, 7643
- Với k = 10000 < $10^m$ ⇒ kết thúc sắp xếp
Đánh giá độ phức tạp về thời gian của thuật toán:
Số phép so sánh của thuật toán phụ thuộc vào số chữ số của phần tử lớn nhất.
Average case: $O(n.log_{10} m)$
Worst case: $O(n.\log_{10} m)$

## 2.11 Flash Sort

***Ý tưởng:**

Trong một tập dữ liệu với một phân bố được biết, ta dễ dàng để ước tính ngay lập tức một phần tử sau khi sắp xếp khi đã biết kích thước của danh sách. Thuật toán nay là một thuật toán in-place, không đệ quy.

*** Mô tả thuật toán:**

Bước 1:Phân lớp dữ liệu: dựa trên giả thiết dữ liệu tuân theo 1 phân bố nào đó, chẳng hạn phân bố đều, để tìm 1 công thức ước tính vị trí (lớp) của phần tử sau khi sắp xếp.

Tìm giá trị nhỏ nhất của các phần tử của mảng(min) và vị trí phần tử lớn nhất của các phần tử trong mảng(max_Index)

Khởi tạo một vecto L có m phần tử(m là số lớp, m=0.45n)

Đếm số lượng phần tử các lớp theo quy luật, phần tử a[i] sẽ thuộc lớp

$k = int((m-1) * (a[i] - \min)/(\text{a[max\_Index]} - \min))$

Tính vị trí kết thúc của phần tử lớp j theo công thức:$L[j] = L[j] + L[j-1]$(j chạy từ 1 đến m-1)

Bước 2: Hoán vị toàn cục: dời chuyển các phần tử trong mảng về lớp của mình.

Đổi chỗ a[max_Index] và a[0].

Với tối đa n-1 lần swap, n phần tử trong mảng sẽ được sắp xếp đúng phân lớp

Khi j>L[k]-1 nghĩa là phần tử a[j] đã ở đúng phân lớp, bỏ qua và đến với các phần tử tiếp theo

Khi một phần tử được đưa về đúng phân lớp thì giảm vị trí cuối cùng của phân lớp đó xuống , tăng biến đếm số lần hoán vị lên 1, quá trình này tới khi L[k]=i, phân lớp k đã đầy.

Bước 3: Sắp xếp cục bộ, tức là để sắp xếp lại các phần tử trong phạm vi của từng lớp để

*** Minh hoạ các bước:**

Ta xét mảng sau:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 14 | 12 | 11 | 19 | 16 | 13 | 17 |

Ta có: m=(int)0.45x7=3

Giá trị nhỏ nhất:min=a[2]=11

Vị trí chứa giá trị lớn nhất : max_Index =3 (a[3]=19 lớn nhất)

c1=(m-1)/(a[max_Index] -min)=(3-1)/(19-11)=1/4

Tiến hành xác định phân lớp theo công thức

$$k = int\left((m-1) * \frac{a[i] - \min}{\text{a[max\_Index]} - \min}\right) = \text{c1} * (\text{a[i]} - \min) = \frac{1}{4}(\text{a[i]} - 11)$$

Khi đó:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 14 | 12 | 11 | 19 | 16 | 13 | 17 |
| Phân lớp | 0 | 0 | 0 | 2 | 1 | 0 | 1 |

Sau đó đến bước hoán vị

Swap a[max_Index] và a[0]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 19 | 12 | 11 | 14 | 16 | 13 | 17 |
| Phân lớp | 2 | 0 | 0 | 0 | 1 | 0 | 1 |

Lúc này move=1, việc hoán vị sẽ dừng lại khi move=n-1=7-1=6

*Note:Các phần tử được tô màu đỏ là các phần tử đã về đúng phân lớp của mình*

Swap a[0] và a[6],

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 17 | 12 | 11 | 14 | 16 | 13 | 19 |
| Phân lớp | 1 | 0 | 0 | 0 | 1 | 0 | 2 |

Xét tiếp phần tử a[0]=17:

Lúc này move=2

Swap a[0] và a[5]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 13 | 12 | 11 | 14 | 16 | 17 | 19 |
| Phân lớp | 0 | 0 | 0 | 0 | 1 | 1 | 2 |

Xét a[0]=13;

Swap a[0] và a[3]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 14 | 12 | 11 | 13 | 16 | 17 | 19 |
| Phân lớp | 0 | 0 | 0 | 0 | 1 | 1 | 2 |

Lúc này move=3;

Xét a[0]=14

Swap a[0] và a[2]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 11 | 12 | 14 | 13 | 16 | 17 | 19 |
| Phân lớp | 0 | 0 | 0 | 0 | 1 | 1 | 2 |

Lúc này move=4

Xét a[0]=11

Swap a[0] và a[2]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 12 | 11 | 14 | 13 | 16 | 17 | 19 |
| Phân lớp | 0 | 0 | 0 | 0 | 1 | 1 | 2 |

Lúc này move=5

Xét a[0]=12

Lúc này a[0] được giữ nguyên

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 12 | 11 | 14 | 13 | 16 | 17 | 19 |
| Phân lớp | 0 | 0 | 0 | 0 | 1 | 1 | 2 |

Lúc này move=6

Ta dừng việc hoán vị, ta thu được mảng mà các phân tử đã được đặt đúng phân lớp

Phân lớp

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 12 | 11 | 14 | 13 | 16 | 17 | 19 |
| 0 | 0 | 0 | 0 | 1 | 1 | 2 |

Cuối cùng ta sử dụng Insertion Sort để sắp xếp lại mảng. Dưới đây là hình minh họa cho từng bước

| | | | | | | |
|---|---|---|---|---|---|---|
| 12 | 11 | 14 | 13 | 16 | 17 | 19 |
| 12 | 11 | 14 | 13 | 16 | 17 | 19 |
| 12 | 11 | 14 | 13 | 16 | 17 | 19 |
| 11 | 12 | 14 | 13 | 16 | 17 | 19 |
| 11 | 12 | 14 | 13 | 16 | 17 | 19 |
| 11 | 12 | 14 | 13 | 16 | 17 | 19 |
| 11 | 12 | 13 | 14 | 16 | 17 | 19 |
| 11 | 12 | 13 | 14 | 16 | 17 | 19 |
| 11 | 12 | 13 | 14 | 16 | 17 | 19 |
| 11 | 12 | 13 | 14 | 16 | 17 | 19 |

Mảng được sắp xếp là

| | | | | | | |
|---|---|---|---|---|---|---|
| 11 | 12 | 13 | 14 | 16 | 17 | 19 |

*** Đánh giá độ phức tạp của thuật toán:**

+ Trường hợp tốt: O(n)

+ Trung bình: O(n)

+ Trường hợp xấu: O(n)

# PHẦN 3: KẾT QUẢ THỰC NGHIỆM VÀ NHẬN XÉT

## 3.1 Kết quả thực nghiệm

| SortedData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 10,000 | | 30,000 | | 50,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 0.440 | 100009999 | 4.408 | 900029999 | 9.847 | 2500049999 |
| Insertion Sort | 0.000 | 29998 | 0.000 | 89998 | 0.000 | 149998 |
| Bubble Sort | 0.000 | 20001 | 0.000 | 60001 | 0.001 | 100001 |
| Shaker Sort | 1.661 | 66887474 | 13.750 | 601151499 | 39.968 | 1668020470 |
| Shell Sort | 0.001 | 360042 | 0.003 | 1170050 | 0.005 | 2100049 |
| Heap Sort | 0.014 | 670329 | 0.037 | 2236648 | 0.039 | 3925351 |
| Merge Sort | 0.014 | 475242 | 0.032 | 1559914 | 0.037 | 2722826 |
| Quick Sort | 0.007 | 326627 | 0.020 | 1193885 | 0.033 | 2088631 |
| Counting Sort | 0.001 | 79992 | 0.003 | 239992 | 0.004 | 365523 |
| Radix Sort | 0.000 | 100092 | 0.015 | 360115 | 0.032 | 600115 |
| Flash Sort | 0.000 | 108998 | 0.001 | 326998 | 0.002 | 544998 |

| SortedData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 100,000 | | 300,000 | | 500,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 34.492 | 10000099999 | 283.955 | 90000299999 | 790.943 | 250000499999 |
| Insertion Sort | 0.000 | 299998 | 0.000 | 899998 | 0.016 | 1499998 |
| Bubble Sort | 0.001 | 200001 | 0.002 | 600001 | 0.003 | 1000001 |
| Shaker Sort | 156.945 | 6682455034 | 2028.040 | 59965718485 | 4381.260 | 166800915206 |
| Shell Sort | 0.012 | 4500051 | 0.036 | 15300061 | 0.055 | 25500058 |
| Heap Sort | 0.087 | 8365080 | 0.275 | 27413230 | 0.467 | 47404886 |
| Merge Sort | 0.101 | 5745658 | 0.248 | 18645946 | 0.416 | 32017850 |
| Quick Sort | 0.073 | 4348705 | 0.241 | 15796387 | 0.389 | 29085757 |
| Counting Sort | 0.008 | 665532 | 0.017 | 1865528 | 0.017 | 3065533 |
| Radix Sort | 0.047 | 1200115 | 0.287 | 4200138 | 0.516 | 7000138 |
| Flash Sort | 0.006 | 1089998 | 0.016 | 3269998 | 0.020 | 5449998 |

| NearlySortedData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 10,000 | | 30,000 | | 50,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 0.294 | 100009999 | 2.730 | 900029999 | 7.704 | 2500049999 |
| Insertion Sort | 0.000 | 176270 | 0.000 | 446762 | 0.000 | 545510 |
| Bubble Sort | 0.267 | 99339376 | 2.487 | 899379376 | 4.390 | 1556934480 |
| Shaker Sort | 0.000 | 20002 | 0.000 | 60002 | 0.000 | 100002 |
| Shell Sort | 0.001 | 400828 | 0.003 | 1339838 | 0.005 | 2328948 |
| Heap Sort | 0.006 | 669727 | 0.023 | 2236614 | 0.054 | 3925306 |
| Merge Sort | 0.009 | 510394 | 0.026 | 1653048 | 0.045 | 2811964 |
| Quick Sort | 2.022 | 100029997 | 17.517 | 900089997 | 48.604 | 2500149997 |
| Counting Sort | 0.000 | 70003 | 0.001 | 210003 | 0.002 | 350003 |
| Radix Sort | 0.000 | 100092 | 0.015 | 360115 | 0.015 | 600115 |
| Flash Sort | 0.001 | 244278 | 0.001 | 701714 | 0.003 | 1002402 |

| NearlySortedData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 100,000 | | 300,000 | | 500,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 30.010 | 10000099999 | 372.098 | 90000299999 | 862.144 | 250000499999 |
| Insertion Sort | 0.000 | 627554 | 0.016 | 1393578 | 0.000 | 1898922 |
| Bubble Sort | 12.209 | 4506277360 | 36.973 | 13159651197 | 56.562 | 19782404280 |
| Shaker Sort | 0.001 | 200002 | 0.002 | 600002 | 0.007 | 1000002 |
| Shell Sort | 0.012 | 4704195 | 0.035 | 15469737 | 0.055 | 2568111429 |
| Heap Sort | 0.081 | 8364913 | 0.430 | 27413146 | 0.476 | 47404795 |
| Merge Sort | 0.069 | 5834796 | 0.331 | 18749942 | 0.378 | 32116407 |
| Quick Sort | 195.526 | 10000299997 | 1738.960 | 90000899997 | 6695.470 | 250001499997 |
| Counting Sort | 0.006 | 700003 | 0.010 | 2100003 | 0.094 | 3500003 |
| Radix Sort | 0.063 | 1200115 | 0.296 | 4200138 | 0.547 | 7000138 |
| Flash Sort | 0.004 | 1466702 | 0.016 | 3532874 | 0.024 | 5827442 |

| ReverseData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 10,000 | | 30,000 | | 50,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 0.290 | 100009999 | 2.677 | 900029999 | 9.211 | 2500049999 |
| Insertion Sort | 0.234 | 100009999 | 2.166 | 900029999 | 6.007 | 2500049999 |
| Bubble Sort | 0.670 | 100019998 | 6.445 | 900059998 | 16.481 | 2500099998 |
| Shaker Sort | 2.576 | 100005001 | 23.182 | 900015001 | 63.596 | 2500025001 |
| Shell Sort | 0.001 | 475175 | 0.004 | 1554051 | 0.005 | 2844628 |
| Heap Sort | 0.006 | 606771 | 0.021 | 2063324 | 0.059 | 3612724 |
| Merge Sort | 0.007 | 476441 | 0.019 | 1573465 | 0.053 | 2733945 |
| Quick Sort | 1.446 | 100029997 | 15.467 | 900089997 | 36.795 | 2500149997 |
| Counting Sort | 0.001 | 80002 | 0.002 | 240002 | 0.007 | 400002 |
| Radix Sort | 0.000 | 100092 | 0.000 | 360115 | 0.032 | 600115 |
| Flash Sort | 0.199 | 100049007 | 1.776 | 900147007 | 4.726 | 2500245007 |

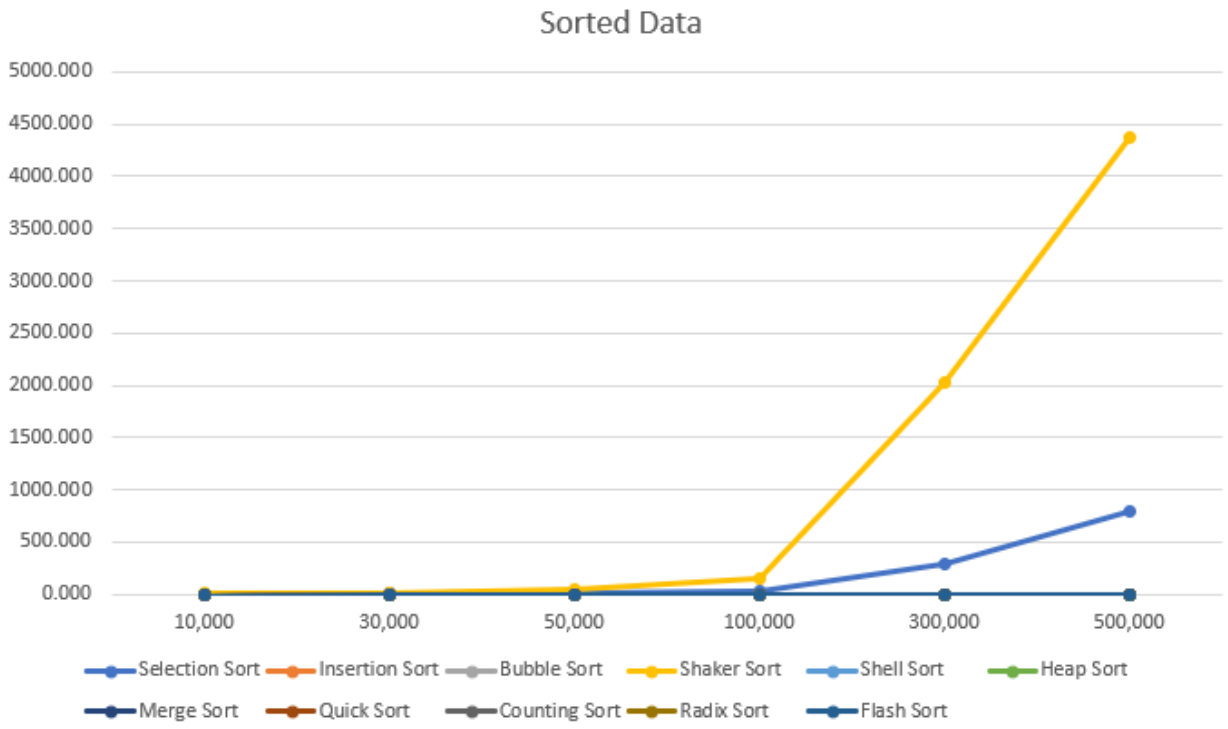
| ReverseData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 100,000 | | 300,000 | | 500,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 34.986 | 10000099999 | 289.513 | 90000299999 | 792.986 | 250000499999 |
| Insertion Sort | 6.007 | 2500049999 | 239.323 | 90000299999 | 610.014 | 250000499999 |
| Bubble Sort | 64.963 | 10000199998 | 606.719 | 90000599998 | 1639.730 | 250000999998 |
| Shaker Sort | 265.094 | 10000050001 | 1825.690 | 90000150001 | 7543.040 | 250000250001 |
| Shell Sort | 0.015 | 6089190 | 0.015 | 6089190 | 0.088 | 33857581 |
| Heap Sort | 0.141 | 7718943 | 0.353 | 25569379 | 0.456 | 44483348 |
| Merge Sort | 0.105 | 5767897 | 0.337 | 18708313 | 0.397 | 32336409 |
| Quick Sort | 166.249 | 10000299997 | 1162.450 | 90000899997 | 3276.090 | 250001499997 |
| Counting Sort | 0.008 | 800002 | 0.013 | 2400002 | 0.024 | 4000002 |
| Radix Sort | 0.063 | 1200115 | 0.297 | 4200138 | 0.550 | 7000138 |
| Flash Sort | 19.465 | 10000490007 | 181.182 | 90001470007 | 644.775 | 250002450007 |

| RandomData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 10,000 | | 30,000 | | 50,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 0.312 | 100009999 | 2.695 | 900029999 | 7.414 | 2500049999 |
| Insertion Sort | 0.192 | 50014519 | 1.566 | 451995218 | 4.317 | 1247933907 |
| Bubble Sort | 0.730 | 100016157 | 5.219 | 899857501 | 18.610 | 2500060797 |
| Shaker Sort | 0.002 | 154504 | 0.006 | 521093 | 0.011 | 767950 |
| Shell Sort | 0.002 | 653638 | 0.007 | 2359125 | 0.013 | 4307321 |
| Heap Sort | 0.008 | 638337 | 0.024 | 2150306 | 0.040 | 3771325 |
| Merge Sort | 0.010 | 583566 | 0.024 | 1937283 | 0.044 | 3382648 |
| Quick Sort | 0.736 | 42515753 | 3.379 | 244514889 | 34.175 | 1758326421 |
| Counting Sort | 0.001 | 78075 | 0.002 | 237399 | 0.002 | 380730 |
| Radix Sort | 0.009 | 100092 | 0.018 | 360115 | 0.061 | 600115 |
| Flash Sort | 0.098 | 50096715 | 0.952 | 451098329 | 2.477 | 1251919932 |

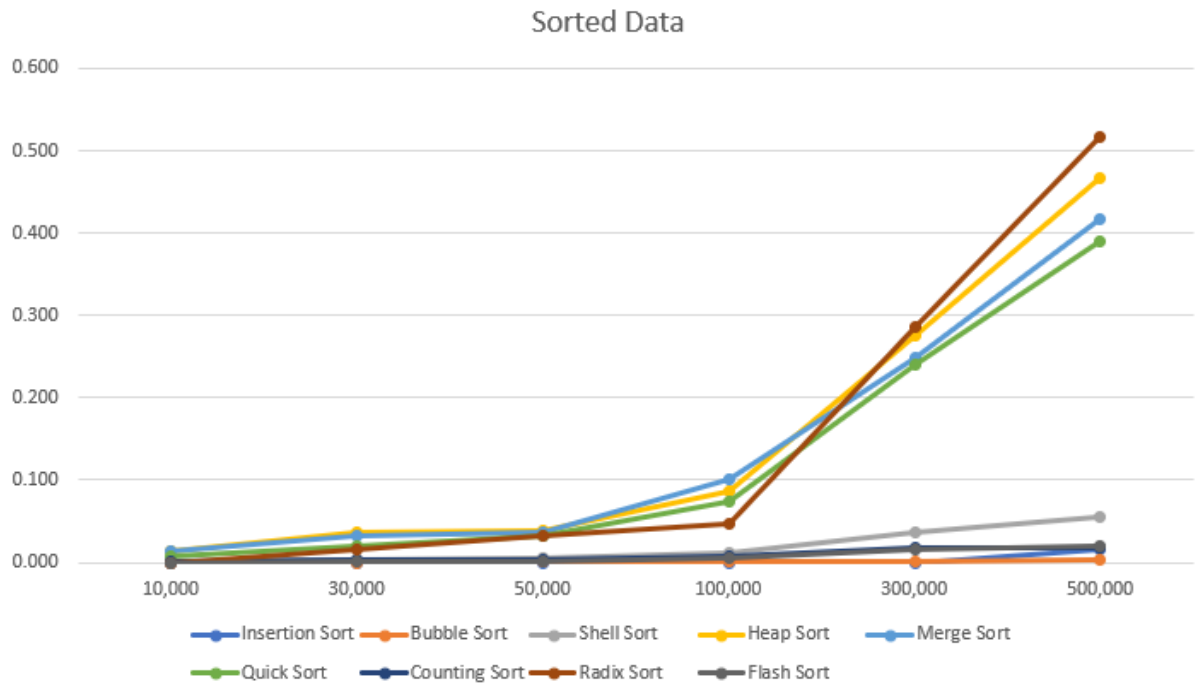
| RandomData | | | | | | |
|---|---|---|---|---|---|---|
| Data size | 100,000 | | 300,000 | | 500,000 | |
| Resulting statics | Running time | Comparision | Running time | Comparision | Running time | Comparision |
| Selection Sort | 29.628 | 10000099999 | 274.452 | 90000299999 | 809.781 | 250000499999 |
| Insertion Sort | 12.715 | 4994830201 | 125.608 | 45042439676 | 352.912 | 125079807649 |
| Bubble Sort | 63.741 | 10000199805 | 554.870 | 89999958400 | 1564.810 | 249998719901 |
| Shaker Sort | 0.011 | 679400 | 0.009 | 979490 | 0.016 | 1617493 |
| Shell Sort | 0.033 | 10090027 | 0.103 | 34505141 | 0.183 | 64098536 |
| Heap Sort | 0.091 | 8045903 | 0.317 | 26489139 | 0.560 | 45968651 |
| Merge Sort | 0.079 | 7165528 | 0.275 | 23382200 | 0.492 | 40381308 |
| Quick Sort | 184.060 | 9092214701 | 1955.600 | 89249334409 | 5217.800 | 249311114749 |
| Counting Sort | 0.007 | 730708 | 0.014 | 2126983 | 0.027 | 3527369 |
| Radix Sort | 0.081 | 1200115 | 0.234 | 3600115 | 0.416 | 6000115 |
| Flash Sort | 0.005 | 938529 | 0.042 | 2852949 | 0.036 | 4708269 |

## 3.2 Biểu đồ

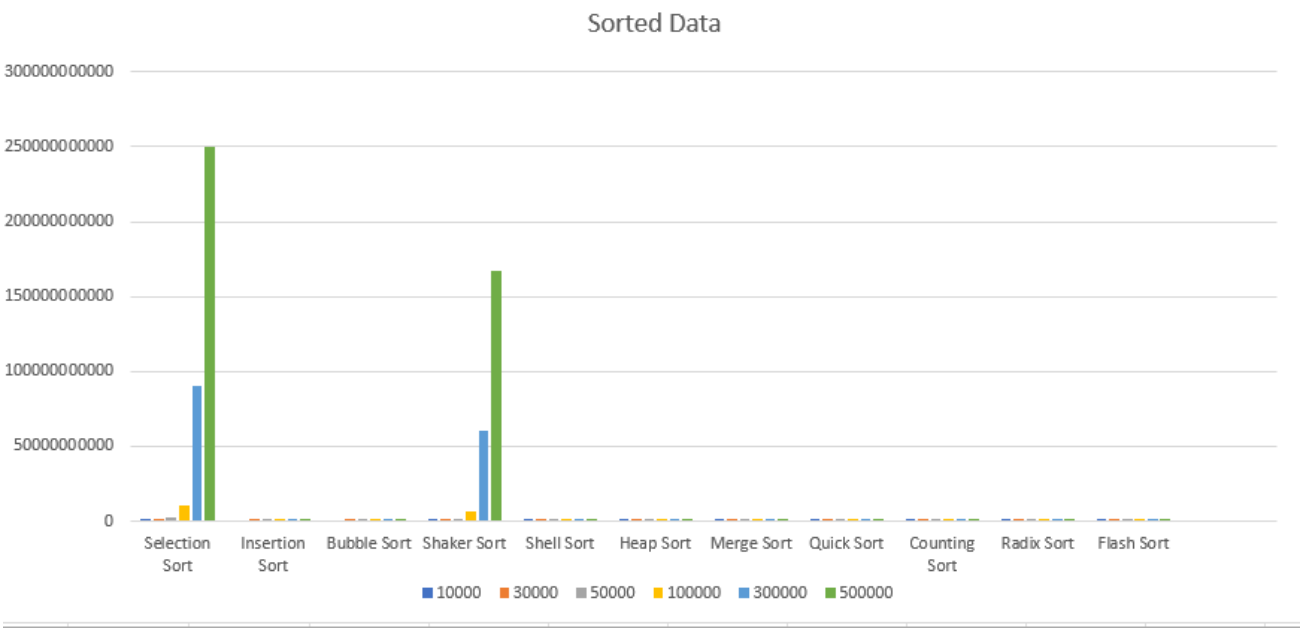
3.2.1. Sorted Data



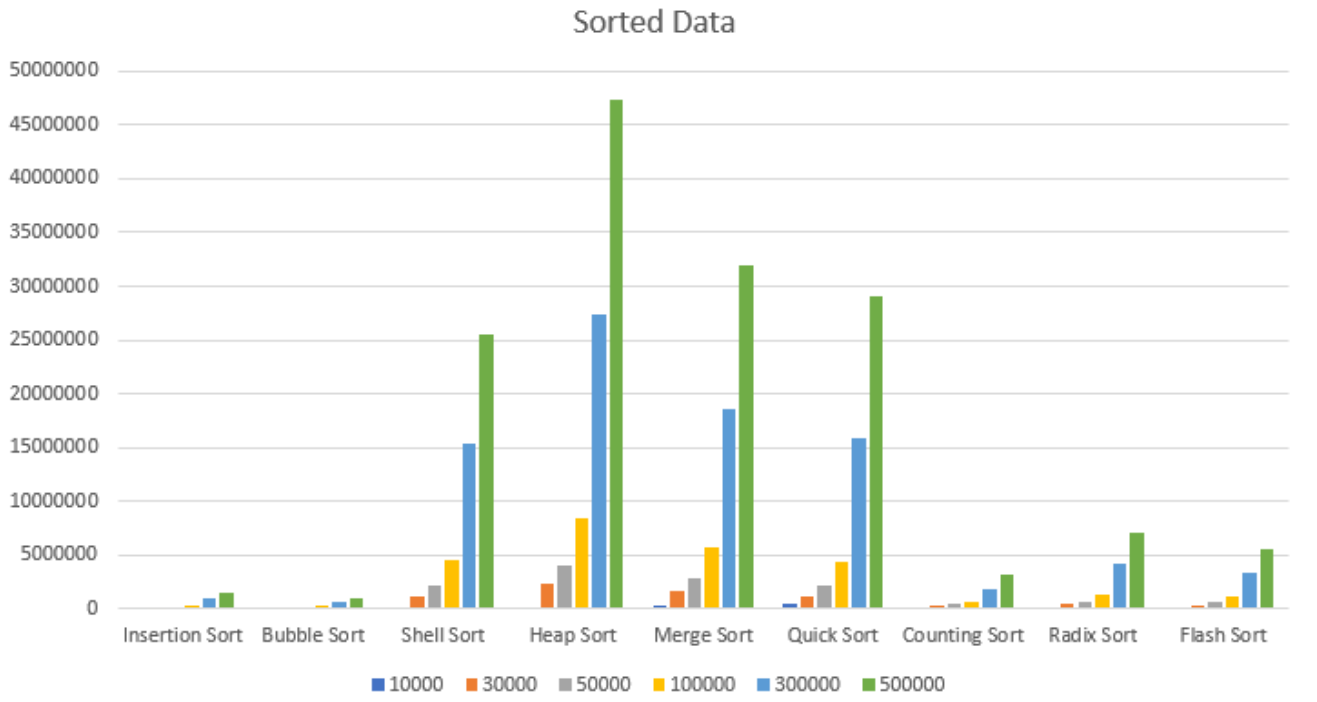

Thời gian thực hiện của các thuật toán với kiểu dữ liệu đã sắp xếp (Đơn vị: giây)




Thời gian thực hiện của các thuật toán với kiểu dữ liệu đã sắp xếp có thời gian nhỏ (Đơn vị: giây)

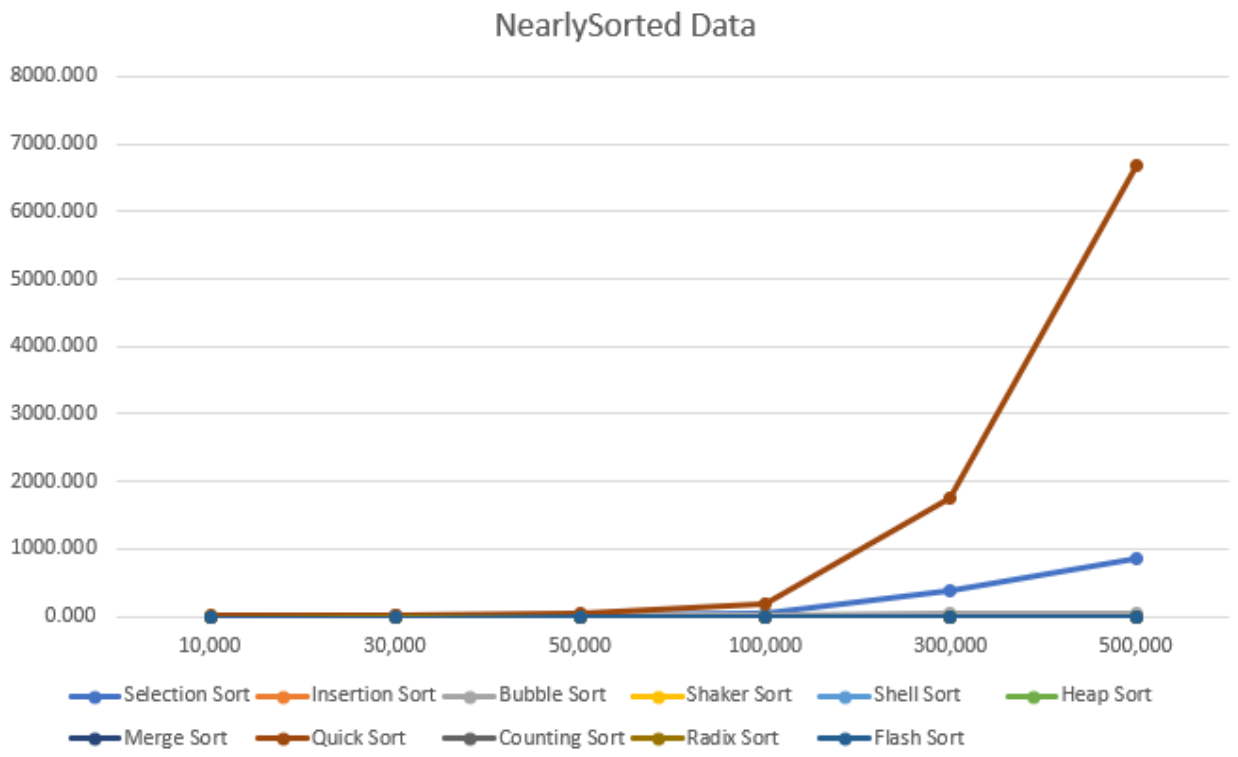



Số phép so sánh của các thuật toán với kiểu dữ liệu đã sắp xếp

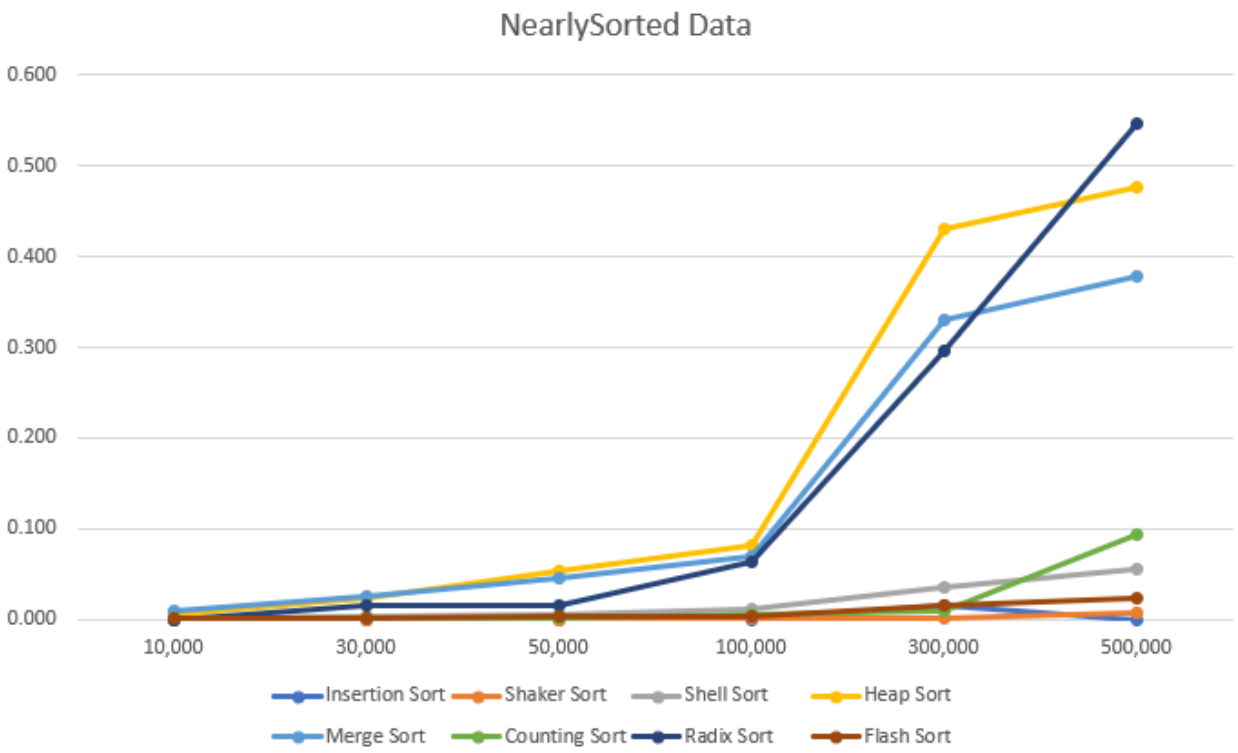



Số phép so sánh của các thuật toán với kiểu dữ liệu đã sắp xếp có số so sánh nhỏ

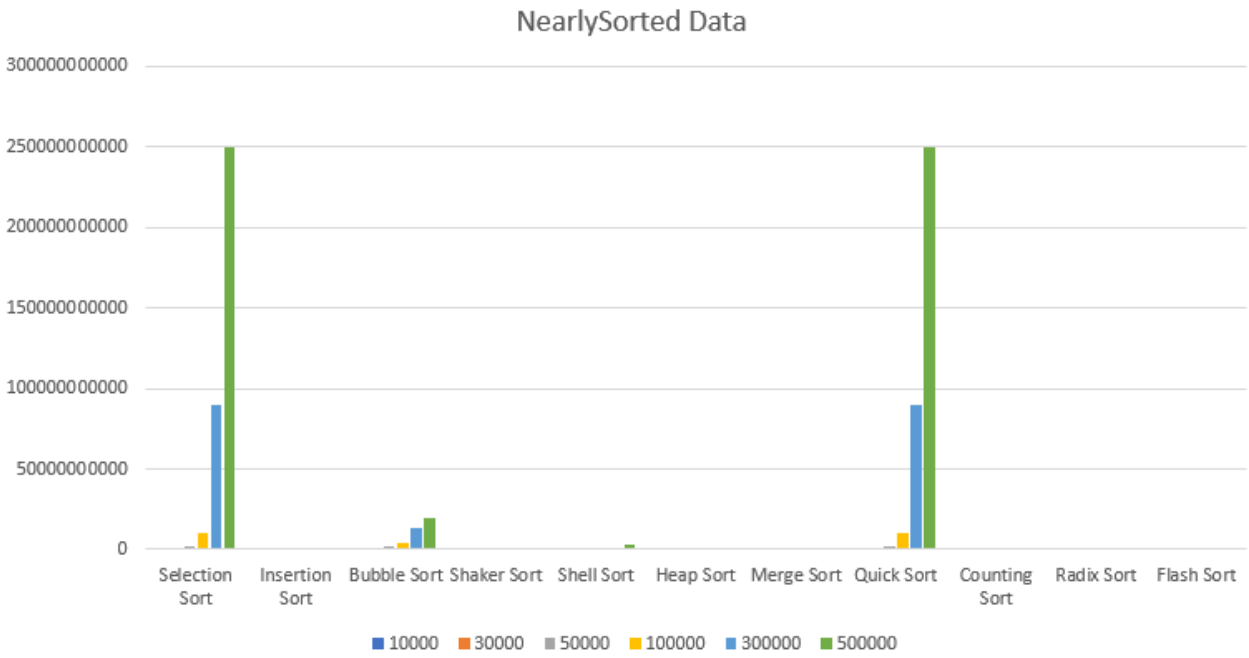
### 3.2.2. NearlySorted Data



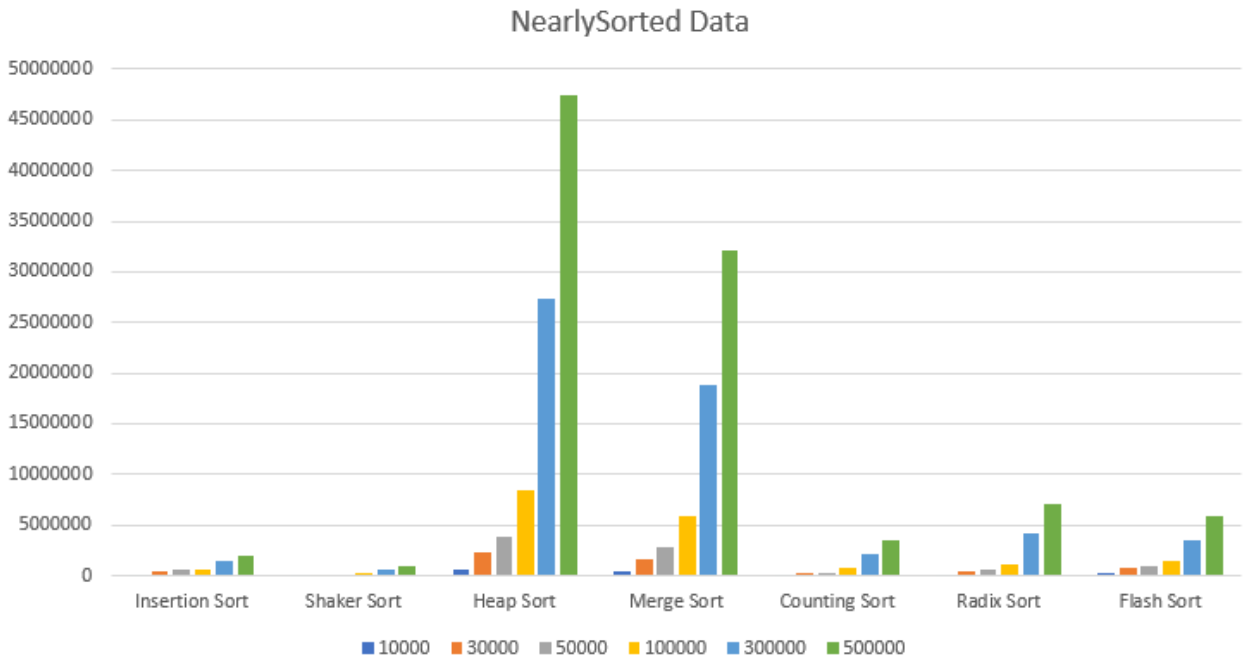

Thời gian thực hiện của các thuật toán với kiểu dữ liệu gần như được sắp xếp (Đơn vị: giây)




Thời gian thực hiện các thuật toán với kiểu dữ liệu gần như được sắp xếp có thời gian nhỏ (Đơn vị: giây)

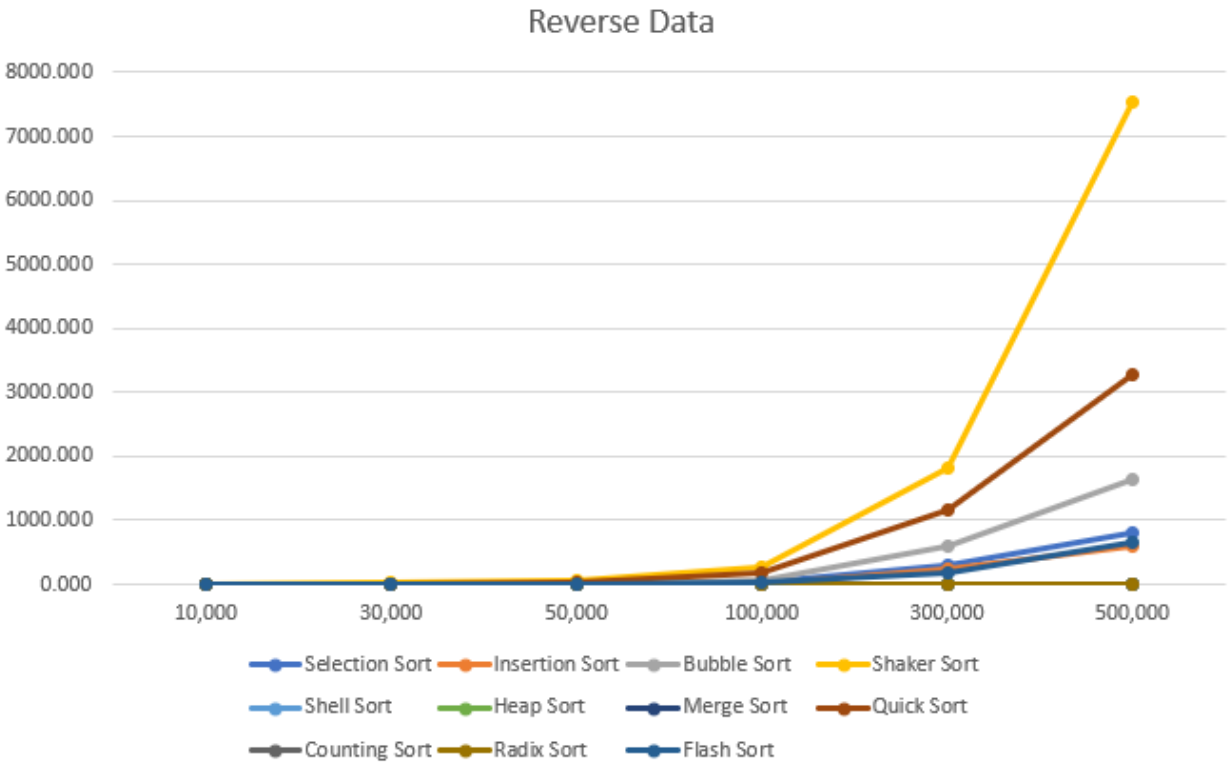



Số phép so sánh của các thuật toán với kiểu dữ liệu gần như được sắp xếp

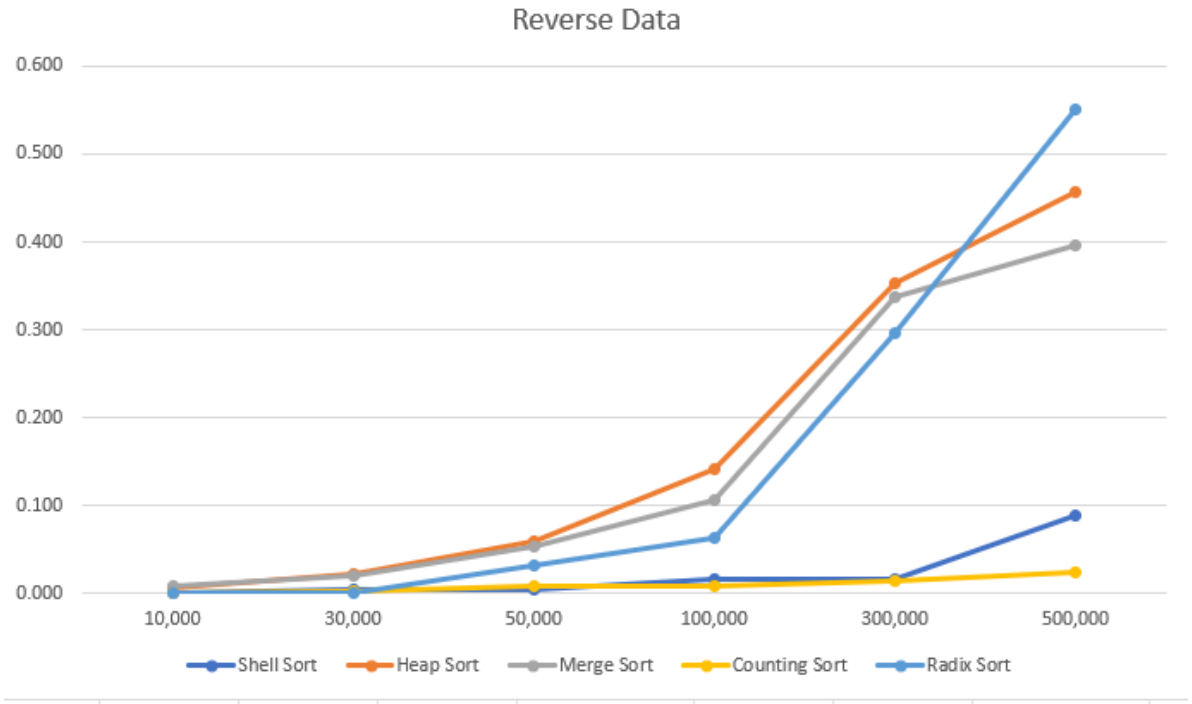



Số phép so sánh của các thuật toán với kiểu dữ liệu gần như được sắp xếp có số so sánh nhỏ

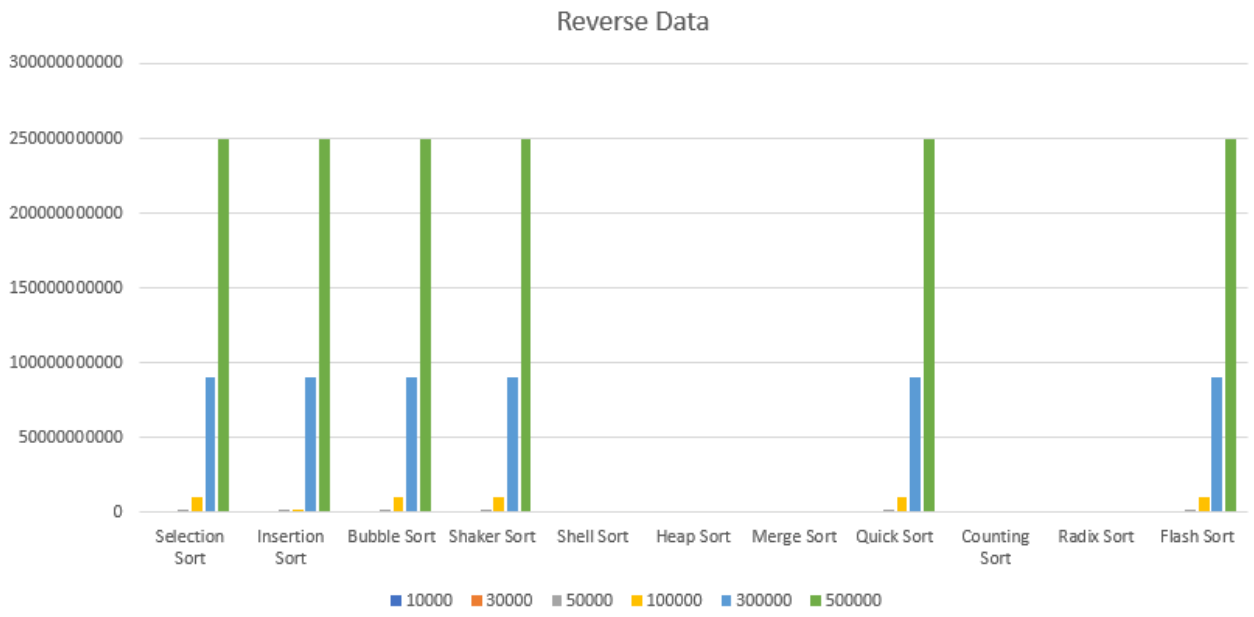
### 3.2.3. Reverse Data



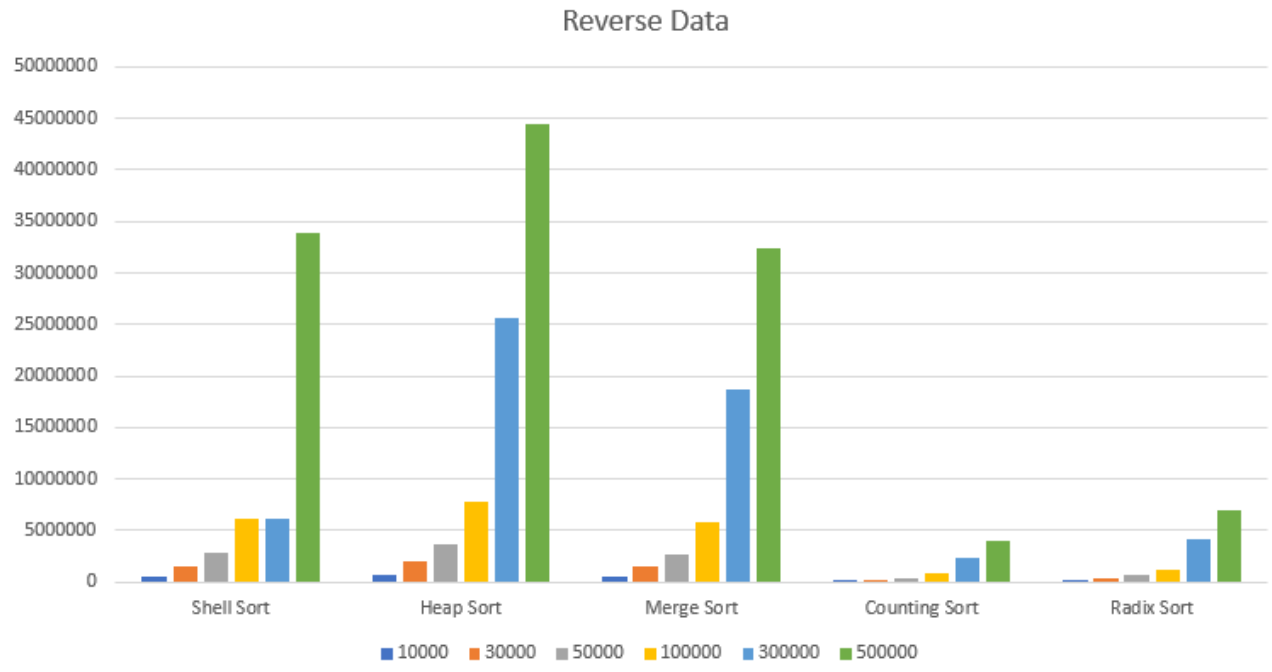

Thời gian thực hiện của các thuật toán với kiểu dữ liệu ngược (Đơn vị: giây)




Thời gian thực hiện của các thuật toán với kiểu dữ liệu ngược có thời gian rất nhỏ (Đơn vị: giây)

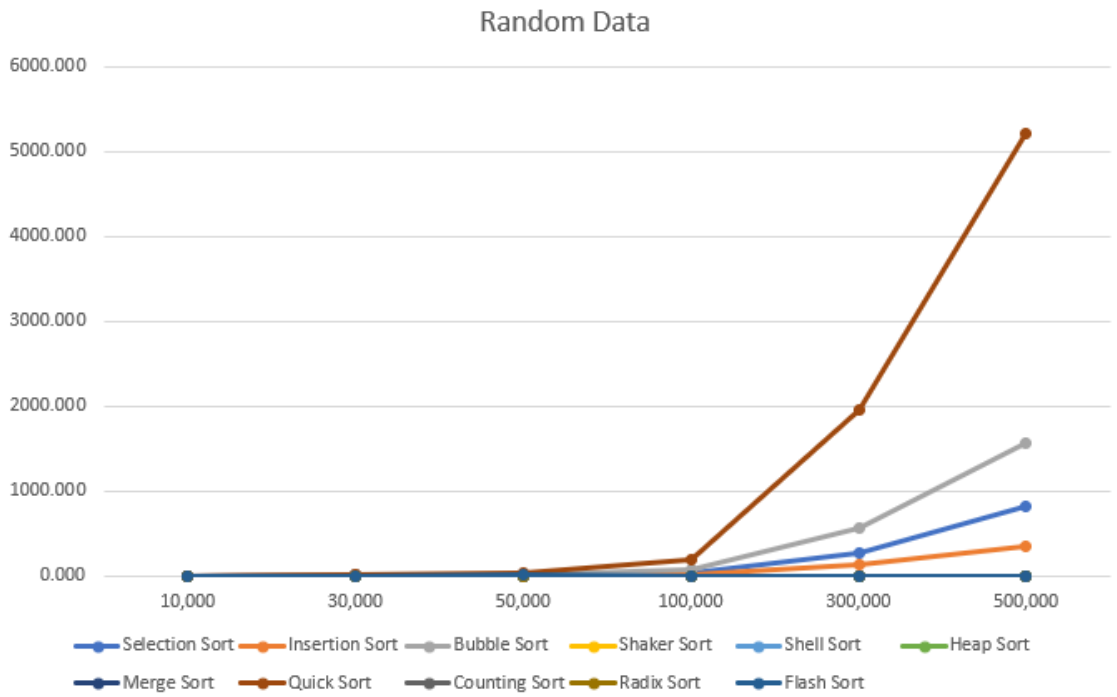



Số phép so sánh của các thuật toán với kiểu dữ liệu ngược

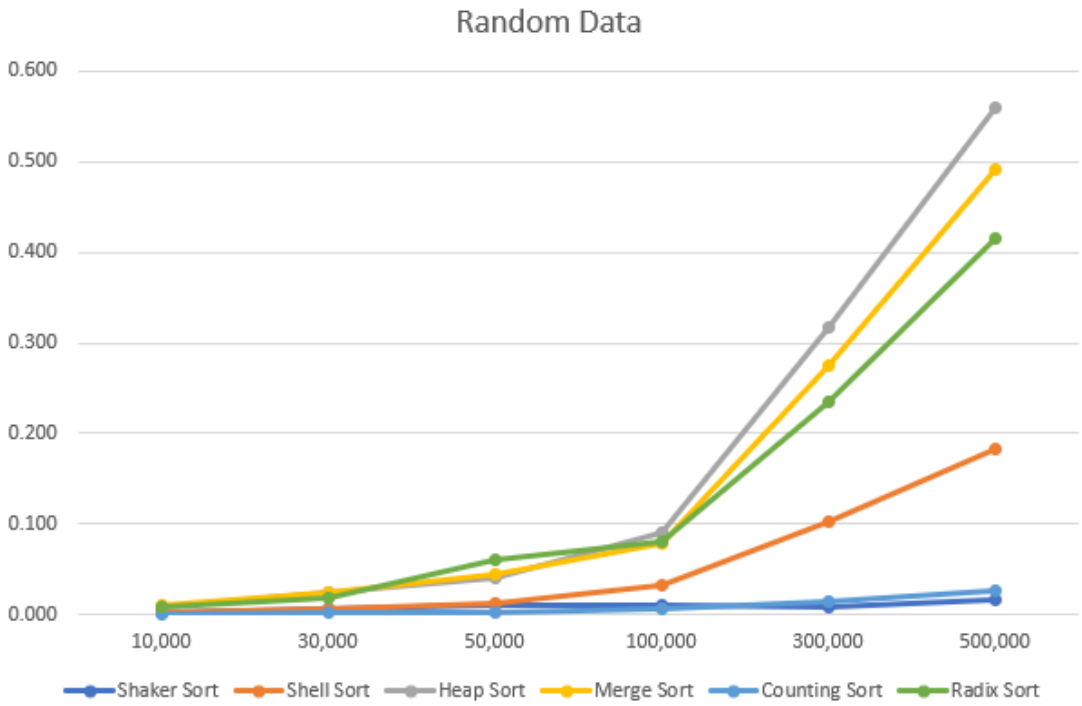



Số phép so sánh của các thuật toán với kiểu dữ liệu ngược có số phép so sánh nhỏ

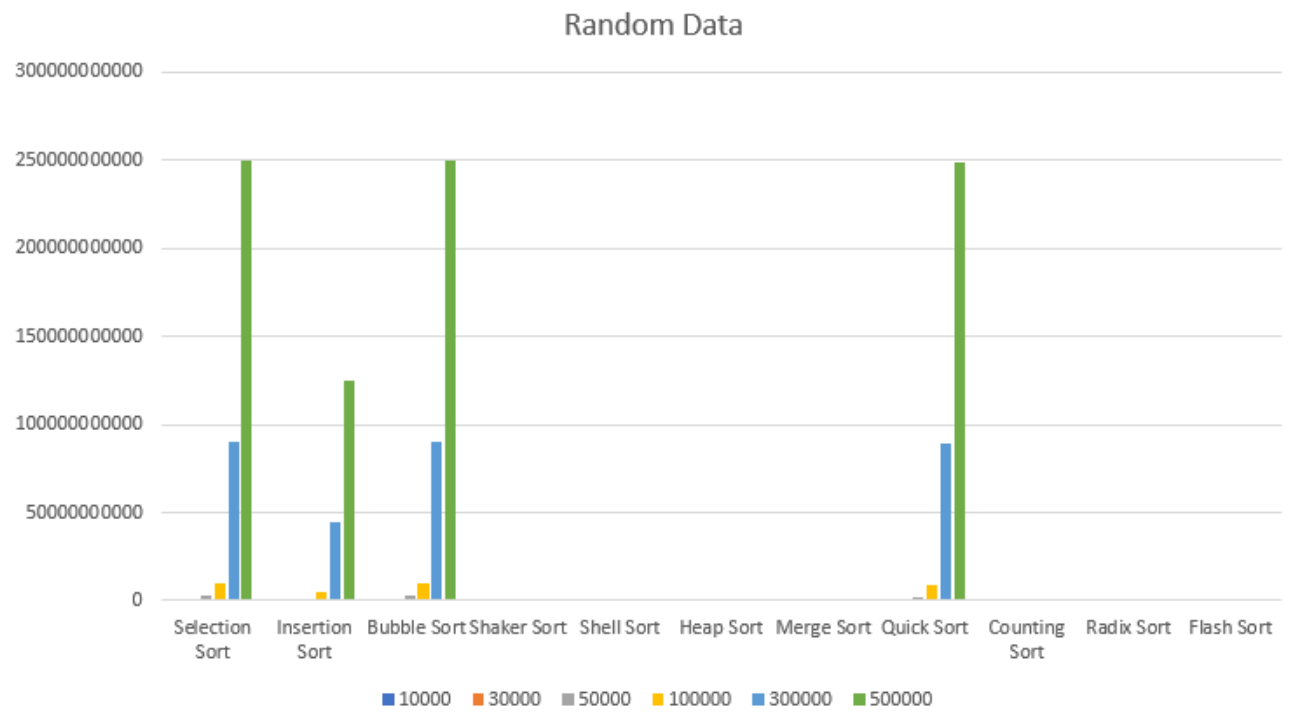
### 3.2.4. Random Data



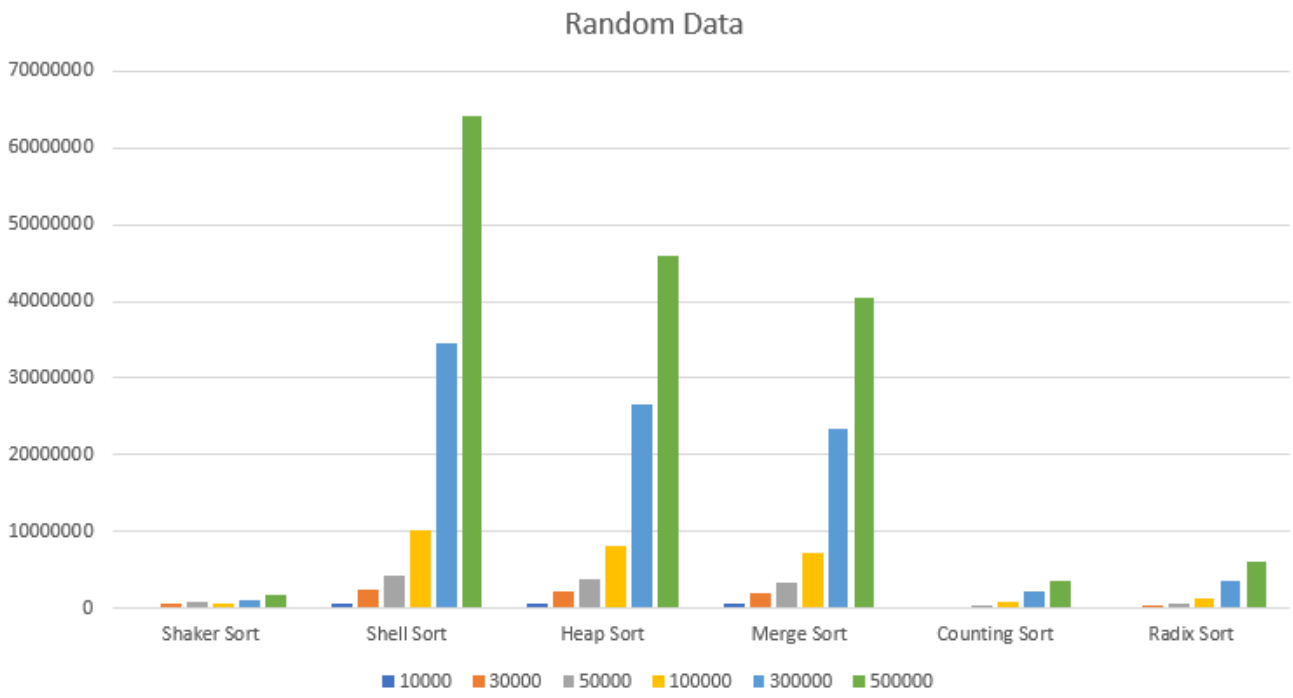

Thời gian thực hiện của các thuật toán với kiểu dữ liệu ngẫu nhiên (Đơn vị: giây)




Thời gian thực hiện của các thuật toán với kiểu dữ liệu ngẫu nhiên có thời gian rất nhỏ (Đơn vị: giây)




Số phép so sánh của các thuật toán với kiểu dữ liệu ngẫu nhiên




Số phép so sánh của các thuật toán với kiểu dữ liệu ngẫu nhiên có số so sánh nhỏ

**3.3 Nhận xét:**

1. Nhận xét cho mỗi biểu đồ (tức là theo data order)

a. Sorted Data

* So sánh

Thực hiện nhiều phép so sánh: Selection Sort, Shaker Sort

Thực hiện nhiều phép so sánh nhất: Selection Sort

Số phép so sánh được thực hiện không đáng kể: Insertion Sort, Bubble Sort

Thực hiện ít phép so sánh nhất: Bubble Sort

* Thời gian

Thời gian thực hiện nhiều nhất: Shaker Sort

Thời gian thực hiện ít nhất: Bubble Sort, Counting Sort, Flash Sort

b. Nearly Sorted Data

* So sánh

Thực hiện nhiều phép so sánh: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Quick Sort

Thực hiện nhiều phép so sánh nhất: Quick Sort

Số phép so sánh được thực hiện không đáng kể: Insertion Sort, Shaker Sort

Thực hiện ít phép so sánh nhất: Shaker Sort

* Thời gian

Thời gian thực hiện nhiều nhất: Quick Sort

Thời gian thực hiện ít nhất: Insertion Sort, Shaker Sort

c. Reverse Data

* So sánh

Thực hiện nhiều phép so sánh: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Quick Sort, Flash Sort

Thực hiện nhiều phép so sánh nhất: Flash Sort

Số phép so sánh được thực hiện không đáng kể: Counting Sort, Radix Sort

Thực hiện ít phép so sánh nhất: Counting sort

* So sánh

Thời gian thực hiện nhiều nhất: Shaker Sort

Thời gian thực hiện ít nhất: Counting sort

d. Random Data

* So sánh

Thực hiện nhiều phép so sánh: Selection Sort, Bubble Sort, Quick Sort, Insertion Sort, Flash Sort

Thực hiện nhiều phép so sánh nhất: Selection Sort
Số phép so sánh được thực hiện không đáng kể: Counting Sort
Thực hiện ít phép so sánh nhất: Counting Sort
* Thời gian
Thời gian thực hiện nhiều nhất: Quick Sort
Thời gian thực hiện ít nhất: Counting Sort, Flash Sort

2. Nhận xét trên toàn bộ 11 thuật toán, thuật toán nào là nhanh nhất, chậm nhất và sự ổn định.

- Với lượng dữ liệu nhỏ, giữa các thuật toán hầu như không có chênh lệch đáng kể
- Với lượng dữ liệu lớn, các thuật toán dần có sự chênh lệch đáng kể:
    - Trường hợp tốt nhất, dữ liệu đầu vào đã được sắp xếp:
        - Các thuật toán chạy nhanh nhất: Bubble Sort, Counting Sort, Flash Sort
        - Các thuật toán chạy chậm nhất: Shaker Sort, Selection Sort
    - Trường hợp trung bình, dữ liệu đầu vào ngẫu nhiên hoặc gần như đã được sắp xếp:
        - Các thuật toán chạy nhanh nhất: Shell Sort, Flash Sort, Counting Sort
        - Các thuật toán chạy chậm nhất: Quick Sort, Selection Sort, Insertion Sort, Bubble Sort
    - Trường hợp tốt nhất, dữ liệu đầu vào có thứ tự đảo ngược với nhu cầu được sắp xếp:
        - Các thuật toán chạy nhanh nhất: Shell Sort, Counting Sort, Merge Sort
        - Các thuật toán chạy chậm nhất: Shaker Sort, Quick Sort, Bubble Sort
- Các thuật toán: Radix Sort, Counting Sort, Shell Sort, Heap Sort, Merge Sort luôn có thời gian chạy rất ngắn trong hầu hết các trường hợp
- Các thuật toán ổn định:

    Counting Sort
    Bubble Sort
    Shaker Sort
    Insertion Sort
    Merge Sort
    Radix Sort

- Các thuật toán không ổn định:

    Quick Sort
    Selection Sort
    Heap Sort
    Shell Sort
    Flash Sort

# PHẦN 4: CÁCH TỔ CHỨC, GHI CHÚ

### 4.1. Cách tổ chức

Project được tổ chức với 5 file:

- SortAlgorithms.h: header file dùng để khai báo thư viện, cấu trúc, hàm sử dụng
- DataGenerator.cpp: random dữ liệu đầu vào
- SortAlgorithm.cpp: chứa mã của các thuật toán sắp xếp và các hàm có liên quan
- Control.cpp: chứa hàm menu để chạy chương trình
- Main.cpp: chạy chương trình

### 4.2 Ghi chú

Phần command line được thiết kế theo phương thức tự nhận biết loại command thứ 1,2,3,4,5 dựa vào tham số đầu vào. Người lập trình chỉ cần nhập đúng tham số đầu vào chương trình sẽ tự nhận diện.

### 4.3 Các cấu trúc dữ liệu đặc biệt đã sử dụng:

- Heap
- Queue

# TÀI LIỆU THAM KHẢO

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

https://www.youtube.com/watch?v=CAaDJJUszvE

https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-12.php

https://codelearn.io/learning/data-structure-and-algorithms/856660

https://vietjack.com/cau-truc-du-lieu-va-giai-thuat/giai-thuat-shell-sort.jsp

https://nguyenvanhieu.vn/thuat-toan-sap-xep-selection-sort/

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

https://nguyenvanhieu.vn/thuat-toan-sap-xep-merge-sort/

https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U

https://nguyenvanhieu.vn/thuat-toan-sap-xep-quick-sort/

https://nguyenvanhieu.vn/counting-sort/

https://stackoverflow.com/questions/56717163/quicksort-algorithm-failed-with-stackoverflow-error

https://nguyenvanhieu.vn/thuat-toan-sap-xep-chen/vn

https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat/thuat-toan-sap-xep-theo-co-so-radix-sort

https://dnmtechs.com/sap-xep-theo-co-so-radix-sort/