

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



**fit@hcmus**

# BÁO CÁO ĐỒ ÁN MÔN HỌC CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

## Đề tài: Sorting Algorithms

**Giáo viên hướng dẫn:** Phan Thị Phương Uyên

**Lớp:** 22CTT3A

**Nhóm:** 10

***Họ và tên***

Trần Xuân Minh Hiền

Võ Phi Hổ

Phan Văn Hoa

Đặng Minh Hoàng

***MSSV***

22120102

22120106

22120107

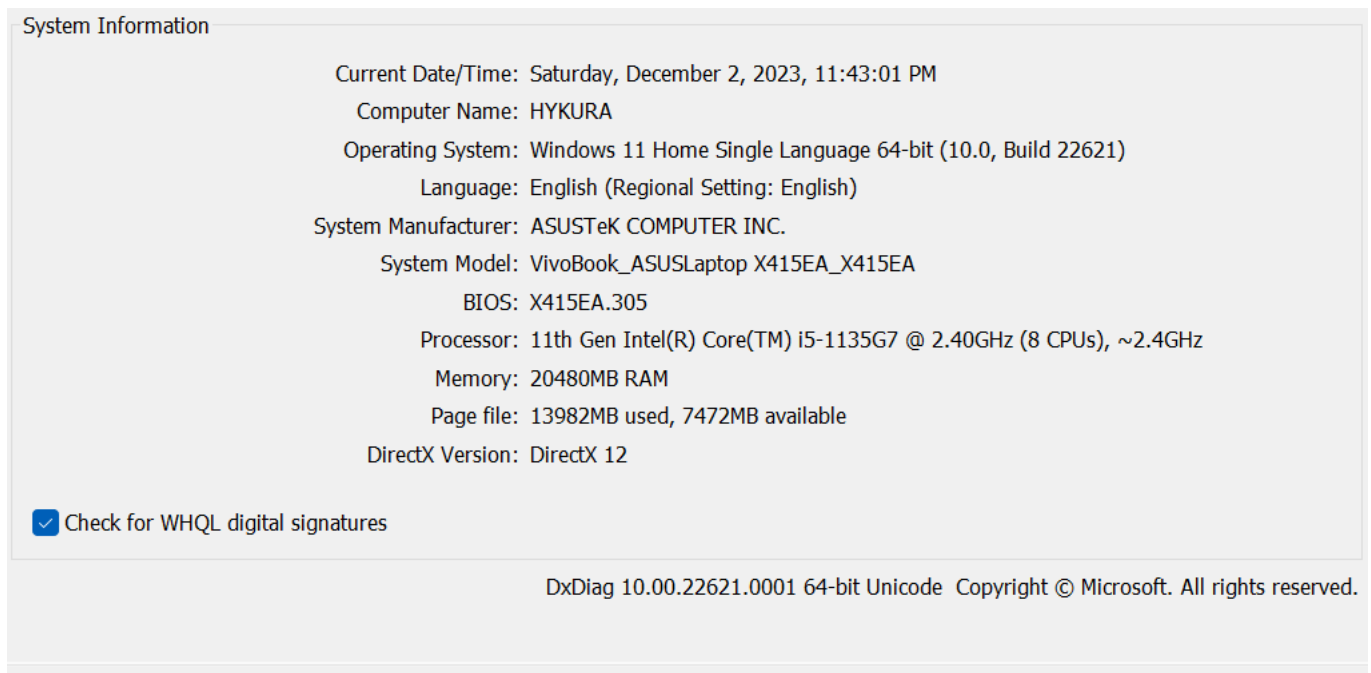
22120110

TP.Hồ Chí Minh, 12/2023

# 1 Trang giới thiệu

## Đâu mới là thuật toán sắp xếp tốt nhất?

Có lẽ đây cũng là câu hỏi của rất nhiều người khi mới học lập trình, trong bài viết này hãy cùng mình thực hiện một thử nghiệm nhỏ với 11 thuật toán sắp xếp để tìm ra câu trả lời nhé! Các thuật toán bao gồm: **Selection Sort**, **Insertion Sort**, **Bubble Sort**, **Shaker Sort**, **Shell Sort**, **Heap Sort**, **Merge Sort**, **Quick Sort**, **Counting Sort**, **Radix Sort**, **Flash Sort**. Và cũng đã hoàn thành 5/5 Commandline: cho 3 chế độ thuật toán và cho 2 chế độ so sánh. Dưới đây là thông số kỹ thuật phần cứng của máy tính chúng tôi đã sử dụng để chạy các thuật toán này:



Hình 1 – Cấu hình máy tính thực hiện

# Mục lục

<b>1</b>	<b>Trang giới thiệu</b>	<b>1</b>
<b>2</b>	<b>Trình bày các thuật toán</b>	<b>5</b>
2.1	Selection Sort . . . . .	5
2.2	Insertion Sort . . . . .	6
2.3	Bubble Sort . . . . .	6
2.4	Shaker Sort (Cocktail Sort) . . . . .	7
2.5	Shell Sort . . . . .	8
2.6	Heap Sort . . . . .	9
2.6.1	Cấu trúc dữ liệu heap . . . . .	9
2.6.2	Xây dựng một min-heap . . . . .	9
2.6.3	Xây dựng một max-heap . . . . .	10
2.7	Merge Sort . . . . .	10
2.8	Quick Sort . . . . .	11
2.9	Counting Sort . . . . .	13
2.10	Radix Sort . . . . .	13
2.11	Flash Sort . . . . .	14
2.11.1	Giai đoạn 1: Phân loại các phần tử của mảng . . . . .	14
2.11.2	Giai đoạn 2: Phân hoạch các phần tử . . . . .	15
2.11.3	Giai đoạn 3: Sắp xếp các phần tử trên mỗi phân đoạn . . . . .	16
<b>3</b>	<b>Kết quả thử nghiệm và nhận xét:</b>	<b>17</b>
3.1	Bảng thời gian chạy(milliseconds) và số lượng phép so sánh: . . . . .	17
3.2	Biểu đồ thời gian chạy: . . . . .	21
3.3	Biểu đồ phép so sánh: . . . . .	23
3.4	Comment: . . . . .	25
3.5	Kết luận: . . . . .	27
<b>4</b>	<b>Tổ chức dự án và ghi chú lập trình:</b>	<b>28</b>
4.1	Tổ chức dự án: . . . . .	28
4.2	Ghi chú lập trình: . . . . .	29

## Danh sách hình vẽ

1	Cấu hình máy tính thực hiện . . . . .	1
2	Biểu đồ thời gian chạy cho Randomized Input . . . . .	21
3	Biểu đồ thời gian chạy cho Sorted Input . . . . .	21
4	Biểu đồ thời gian chạy cho Reversed Input . . . . .	22
5	Biểu đồ thời gian chạy cho Nearly Sorted Input . . . . .	22
6	Biểu đồ số phép so sánh cho Randomized Input . . . . .	23
7	Biểu đồ số phép so sánh cho Sorted Input . . . . .	23
8	Biểu đồ số phép so sánh cho Reversed Input . . . . .	24
9	Biểu đồ số phép so sánh cho Nearly Sorted Input . . . . .	24
10	Files trong dự án . . . . .	28

## Danh sách bảng

1	Data order: Randomized - table 1 . . . . .	17
2	Data order: Randomized - table 2 . . . . .	17
3	Data order: Sorted - table 1 . . . . .	18
4	Data order: Sorted - table 2 . . . . .	18
5	Data order: Reversed - table 1 . . . . .	19
6	Data order: Reversed - table 2 . . . . .	19
7	Data order: Nearly Sorted - table 1 . . . . .	20
8	Data order: Nearly Sorted - table 2 . . . . .	20

## 2 Trình bày các thuật toán

Trong phần này, nhóm 10 sẽ trình bày ý tưởng, mô tả các bước thực hiện và phân tích độ phức tạp thời gian và không gian của từng thuật toán mà đã được đề cập ở phần trên.

Trong bài báo cáo này các thuật toán sẽ được sử dụng để sắp xếp mảng theo thứ tự không giảm. Và tất nhiên việc sắp xếp theo thứ tự không tăng cũng tương tự.

### 2.1 Selection Sort

**Ý tưởng: Selection Sort** là thuật toán sắp xếp bằng cách đi tìm phần tử nhỏ nhất (lớn nhất) trong mảng từ vị trí đang xét đến vị trí cuối cùng và hoán đổi vị trí với phần tử đang xét.

- Giả sử cần sắp xếp tăng dần một mảng  $a$  với  $n$  phần tử cho trước  $a[0]$ ,  $a[1]$ ,  $a[2]$ , ...,  $a[n-1]$ .
- Ta sẽ tìm phần tử nhỏ nhất trong mảng  $a[0]$  đến  $a[n-1]$  và thực hiện hoán đổi vị trí phần tử nhỏ nhất với  $a[0]$
- Ta tiếp tục sẽ tìm phần tử nhỏ nhất trong mảng  $a[1]$  đến  $a[n-1]$  và thực hiện hoán đổi vị trí phần tử nhỏ nhất với  $a[1]$
- Thực hiện tương tự cho đến phần tử thứ  $n-1$ , ta sẽ được một mảng đã được sắp xếp theo thứ tự tăng dần...

**Mã giả:** [1]

---

```
SelectionSort(A)
  n <- length(A)
  for i <- 0 to n - 2 do
    minIndex <- i
    for j <- i + 1 to n - 1 do
      if A[j] < A[minIndex]
        then minIndex <- j
    swap A[i] and A[minIndex]
```

---

**Độ phức tạp thuật toán:**[3]

- Worst case:  $O(n^2)$ .
- Best case:  $O(n^2)$ .
- Average case:  $O(n^2)$ .

**Độ phức tạp không gian:**  $O(1)$ . [3]

## 2.2 Insertion Sort

**Ý tưởng:** Nếu như mảng con chỉ có 1 phần tử  $a[1]$  thì mảng con có thể được xem là đã sắp xếp. Xét  $a[2]$ , ta so sánh với  $a[1]$ , nếu  $a[2] < a[1]$ , ta chèn vào trước  $a[1]$ . Với  $a[3]$ , chúng ta so sánh nó với mảng con được sắp xếp  $a[1..2]$ , tìm vị trí để chèn  $a[3]$  vào mảng con đó để có thứ tự tăng dần. Trong một bài phát biểu chung, chúng ta sẽ sắp xếp mảng  $a[1..k]$  nếu mảng  $a[1..k-1]$  đã được sắp xếp bằng cách chèn  $a[k]$  vào vị trí chính xác của nó.

**Mã giả:** [1]

---

```
InsertionSort(A)
  n <- length(A)
  for i <- 1 to n - 1 do
    key <- A[i]
    j <- i - 1
    while j >= 0 and A[j] > key do
      A[j + 1] <- A[j]
      j <- j - 1
    A[j + 1] <- key
```

---

**Độ phức tạp thời gian:** [4]

- Worst case:  $O(n^2)$ .
- Best case:  $O(n)$ , trong trường hợp mảng đã được sắp xếp.
- Average case:  $O(n^2)$ .

**Độ phức tạp không gian:**  $O(1)$ . [4]

**Cải tiến:**

- Binary insertion Sort — Tìm vị trí để chèn bằng cách sử dụng tìm kiếm nhị phân, làm giảm số lượng so sánh. Chi tiết tại link: [?].
- Một cách cải tiến khác của Insertion Sort là shell Sort, sẽ được trình bày trong phần 2.5

## 2.3 Bubble Sort

**Ý tưởng:** Sắp xếp bong bóng là thuật toán sắp xếp đơn giản, hoán đổi các phần tử liền kề nếu chúng theo thứ tự sai, lặp đi lặp lại  $n$  lần. Sau lượt thứ  $i$ —th, phần tử thứ  $i$ —th nhỏ nhất sẽ được hoán đổi sang vị trí thứ  $i$ .

**Các biến thể khác:** Có một số biến thể khác trong việc thực hiện:

- Thay vì di chuyển từ đầu đến cuối với  $j$ , Chúng ta có thể lặp lại từ cuối lên, từ  $i + 1$  đến  $n$ .
- Một biến thể khác là  $j$  lặp lại từ 1 đến  $n - i$ .

Mã giả: [1]

---

```
BubbleSort(A)
  n <- length(A)
  for i <- 0 to n - 2 do
    swapped <- false
    for j <- 0 to n - i - 2 do
      if A[j] > A[j + 1]
        swap(A[j] and A[j + 1])
        swapped <- true
    if not swapped
      then break
```

---

Độ phức tạp thời gian: [5]

- Worst case:  $O(n^2)$ .
- Best case:  $O(n)$ , trong trường hợp mảng đã được sắp xếp.
- Average case:  $O(n^2)$ .

Độ phức tạp không gian:  $O(1)$ . [5]

**Cải tiến:** Một cách cải tiến khác của bubble Sort là shaker Sort, sẽ được trình bày trong phần 2.4.

## 2.4 Shaker Sort (Cocktail Sort)

**Ý tưởng:** Shaker Sort, còn được gọi là **Cocktail Sort** hoặc **Bi-Directional Bubble Sort**, là một cải tiến của **Bubble Sort**. Trong **Bubble Sort**, các phần tử được duyệt từ trái sang phải, tức là theo một hướng duy nhất. Tuy nhiên, **Shaker Sort** sẽ duyệt cả hai hướng, từ trái sang phải và từ phải sang trái, xen kẽ lẫn nhau. [6]

Mã giả: [1]

---

```
ShakerSort(A)
  n <- length(A)
  left <- 0
  right <- n - 1
  swapped <- true
  while swapped
    swapped <- false
    for i <- left to right - 1 do
      if A[i] > A[i + 1]
        then swap A[i] and A[i + 1]
        swapped <- true
    if not swapped
      then break
    swapped <- false
    for i <- right to left + 1 do
```



```

    if A[i - 1] > A[i]
        then swap A[i - 1] and A[i]
            swapped <- true
left <- left + 1

```

---

**Độ phức tạp thời gian:** [6]

- Worst case:  $O(n^2)$ .
- Best case:  $O(n)$ , trong trường hợp mảng đã được sắp xếp.
- Average case:  $O(n^2)$ .

**Độ phức tạp không gian:**  $O(1)$ . [6]

## 2.5 Shell Sort

Một nhược điểm của **Insertion Sort** là chúng ta luôn phải chèn một phần tử vào một vị trí gần đầu của mảng. Để khắc phục trường hợp đó, chúng ta sử dụng **Shell Sort**.

**Ý tưởng:** Xem xét một mảng  $a[1..n]$ . Đối với một số nguyên  $h : 1 \leq h \leq n$ , chúng ta có thể chia mảng ban đầu thành  $h$  mảng con:

- Mảng con 1:  $a[1], a[1 + h], a[1 + 2h] \dots$
- Mảng con 2:  $a[2], a[2 + h], a[2 + 2h] \dots$
- ...
- Mảng con  $h$ :  $a[h], a[2h], a[3h] \dots$

Các mảng con này được gọi là mảng con có bước  $h$ . Với bước  $h$ , **Shell Sort** sẽ sử dụng **Insertion Sort** cho các mảng con độc lập, sau đó tương tự với  $\frac{h}{2}, \frac{h}{4}, \dots$  cho đến khi  $h = 1$ .

**Mã giả:** [1]

---

```

ShellSort(A)
    n <- length(A)
    gap <- n / 2
    while gap > 0 do
        for i <- gap to n - 1 do
            key <- A[i]
            j <- i
            while j >= gap and A[j - gap] > key do
                A[j] <- A[j - gap]
                j <- j - gap
            A[j] <- key
        gap <- gap / 2

```

---

**Độ phức tạp thời gian:** [7]

- Worst case:  $O(n^2)$ .

- Best case:  $O(n \log n)$ .
- Average case: phụ thuộc vào gap sequence.

**Độ phức tạp không gian:**  $O(1)$ . [7]

## 2.6 Heap Sort

**Heap Sort** được phát minh bởi J. W. J. Williams vào năm 1981, thuật toán này không chỉ giới thiệu một thuật toán sắp xếp hiệu quả mà còn xây dựng một cấu trúc dữ liệu quan trọng để biểu diễn hàng đợi ưu tiên: cấu trúc dữ liệu heap.

### 2.6.1 Cấu trúc dữ liệu heap

Heap là một cây nhị phân đặc biệt. Một cây nhị phân được coi là tuân theo cấu trúc dữ liệu heap nếu:

- Nó là một cây nhị phân hoàn chỉnh,
- Tất cả các nút trong cây thỏa mãn rằng chúng lớn hơn các con của mình, tức là phần tử lớn nhất là gốc. Một heap như vậy được gọi là max-heap. Ngược lại, nếu tất cả các nút đều nhỏ hơn con của chúng, nó được gọi là min-heap. [8]

### 2.6.2 Xây dựng một min-heap

Để xây dựng một min-heap, chúng ta thực hiện những bước sau: [9]

- Tạo một nút con mới ở cuối của heap (tầng cuối cùng).
- Thêm khóa mới vào nút đó (nối thêm vào mảng).
- Di chuyển nút con lên cho đến khi chúng ta đạt đến nút gốc và thuộc tính heap được đảm bảo.

Để xóa một nút gốc trong một min-heap, chúng ta thực hiện những bước sau: [9]

- Xóa nút gốc.
- Di chuyển khóa của nút con cuối cùng lên gốc.
- So sánh nút cha với các nút con của nó..
- Nếu giá trị của nút cha lớn hơn so với các nút con, đổi chỗ chúng và lặp lại cho đến khi thuộc tính heap được đảm bảo.

### 2.6.3 Xây dựng một max-heap

Việc xây dựng một max-heap tương tự như việc xây dựng một min-heap.

Mã giả: [1]

---

```
maxHeap(a, n, pos)
  pos1 <- 2 * pos + 1
  pos2 <- 2 * pos + 2
  maxPos <- pos
  if pos1 < n and a[pos1] > a[maxPos]
    then maxPos = pos1
  if pos2 < n and a[pos2] > a[maxPos]
    then maxPos = pos2
  if maxPos != pos
    then swap a[pos] and a[maxPos]
         maxHeap(a, n, maxPos)

HeapSort(a)
  n <- length(a)
  for i <- n/2 - 1 downto 0 do
    maxHeap(a, n, maxPos)
  for i <- n - 1 downto 0 do
    swap a[0] and a[i]
    maxHeap(a, i, 0)
```

---

Độ phức tạp thời gian: [8]

- Worst case:  $O(n \log n)$ .
- Best case:  $O(n \log n)$ .
- Average case:  $O(n \log n)$ .

Độ phức tạp không gian:  $O(1)$ . [8]

## 2.7 Merge Sort

**Merge Sort** là một thuật toán chia để trị được phát minh bởi John von Neumann vào năm 1945. Đây là một trong những thuật toán sắp xếp phổ biến nhất.

**Ý tưởng:**

- Chia mảng thành hai mảng con tại vị trí giữa.
- Cố gắng sắp xếp cả hai mảng con. Nếu chúng ta chưa đạt được trường hợp cơ bản, tiếp tục chia chúng thành các mảng con.
- Trộn các mảng con đã sắp xếp.

Mã giả: [1]

---

```

MergeSort(A, l, r)
  if l < r
    then m <- (l + r)/2
         MergeSort(A, l, m)
         MergeSort(A, m + 1, r)
         Merge(A, l, m, r)

Merge(A, l, m, r)
  n1 <- m - l + 1
  n2 <- r - m
  for i <- 0 to n1 do
    L[i] <- A[l + i]
  for j <- 0 to n2 do
    R[j] <- A[m + j + 1]
  i <- 0
  j <- 0
  while i < n1 and j < n2 do
    if L[i] <= R[j]
      then A[k] <- L[i]
           i <- i + 1
    else
      then A[k] <- R[j]
           j <- j + 1
    k <- k + 1
  while i < n1 do
    A[k] <- L[i]
    i <- i + 1
    k <- k + 1
  while j < n2 do
    A[k] <- R[j]
    j <- j + 1
    k <- k + 1

```

---

**Độ phức tạp thời gian:**  $[10]$

- Worst case:  $O(n \log n)$ .
- Best case:  $O(n \log n)$ .
- Average case:  $O(n \log n)$ .

**Độ phức tạp không gian:**  $O(n)$ .  $[10]$

## 2.8 Quick Sort

**QuickSort** là một thuật toán chia để trị, được giới thiệu bởi C. A. R. Hoare, một nhà khoa học máy tính người Anh, vào năm 1960. Nó đã trở nên phổ biến vì hiệu suất của nó và hiện đang là một trong những thuật toán sắp xếp phổ biến nhất. So với **Bubble Sort** thì thuật toán sắp xếp nhanh có tốc độ nhanh hơn. Thay vì đi theo sắp xếp từng cặp như bubble Sort, chúng ta có thể chia dữ liệu ra thành 2 danh sách, rồi so sánh từng phần tử của danh sách với một phần tử được chọn (gọi là phần tử chốt) và mục đích của chúng ta là đưa phần tử chốt về đúng vị

trí của nó.

### Ý tưởng:

- Sắp xếp mảng  $a[1..n]$  có thể được hiểu như việc sắp xếp đoạn từ chỉ mục 1 đến chỉ mục  $n$  của mảng đó.
- Để sắp xếp một đoạn, nếu đoạn đó có ít hơn 2 phần tử, thì chúng ta không cần phải làm gì cả. Ngược lại, chúng ta chọn một phần tử ngẫu nhiên để làm "pivot". Tất cả các phần tử nhỏ hơn pivot sẽ được sắp xếp vào một vị trí trước pivot, và tất cả các phần tử lớn hơn pivot sẽ được sắp xếp vào một vị trí sau pivot.
- Sau đó, đoạn được chia thành hai đoạn, tất cả các phần tử trong đoạn đầu tiên đều nhỏ hơn pivot, và tất cả các phần tử trong đoạn thứ hai đều lớn hơn pivot. Và bây giờ chúng ta phải sắp xếp hai đoạn mới, có độ dài nhỏ hơn độ dài của đoạn ban đầu.

Trong dự án này, chúng tôi sẽ chọn các phần tử ở giữa của các đoạn làm điểm pivot.

**Mã giả:** [1].

---

```
Partition(A, l, r)
  x <- A[r]
  i <- l - 1
  for j <- l to r - 1 do
    if A[j] <= x
      then i <- i + 1
           swap A[i] and A[j]
  swap A[i + 1] and A[r]
  return i + 1

QuickSort(A, l, r)
  if l < r
    then m <- Partition(A, l, r)
         QuickSort(A, l, m - 1)
         QuickSort(A, m + 1, r)
```

---

**Độ phức tạp thời gian:** [11]

- Worst case:  $O(n^2)$ .
- Best case:  $O(n \log n)$ .
- Average case:  $O(n \log n)$ .

**Độ phức tạp không gian:**  $O(1)$ . [11]

**Biến thể:** Dưới đây là triển khai của **Quick Sort** sử dụng đệ quy. Cũng có một thuật toán lặp lại, có thể tìm thấy tại: [12]

## 2.9 Counting Sort

**Counting Sort** là một thuật toán sắp xếp hoạt động bằng cách đếm số lượng đối tượng có giá trị khóa khác nhau (một loại băm). [13]

**Ý tưởng:** Lặp qua đầu vào, đếm số lần xuất hiện của mỗi phần tử, sau đó sử dụng kết quả đó để tính toán chỉ số của phần tử trong mảng đã sắp xếp. [14]

Thuật toán này hoạt động khi mảng chứa các số nguyên không âm trong khoảng  $[l, u]$ .

Trong trường hợp mảng có giá trị âm, thuật toán vẫn có thể hoạt động nhưng tôi sẽ không đề cập đến nó ở đây.

**Counting Sort** hoạt động tốt khi  $n \approx u$ , nhưng sẽ trở nên "thảm họa" nếu  $u \gg n$ . [1]

**Mã giả:** [1]]

---

```
CountingSort(A)
  n <- length(A)
  output <- new array of size n
  count <- new array of size max(A) + 1
  for i <- 0 to max(A) do
    count[i] <- 0
  for i <- 0 to n - 1 do
    count[A[i]] <- count[A[i]] + 1
  for i <- 1 to max(A) do
    count[i] <- count[i] + count[i - 1]
  for i <- n - 1 down to 0 do
    output[count[A[i]] - 1] <- A[i]
    count[A[i]] <- count[A[i]] - 1
  for i <- 0 to n - 1 do
    A[i] <- output[i]
```

---

**Độ phức tạp thời gian:**  $O(n + u)$ . [13]

**Độ phức tạp không gian:**  $O(n + u)$ . [13]

## 2.10 Radix Sort

Giống như **Counting Sort** được đề cập trong phần 2.9, radix Sort chỉ hoạt động với số nguyên.

**Ý tưởng:** Sắp xếp mảng bằng cách sử dụng **Counting Sort** (Hoặc bất kỳ stable algorithms nào) theo chữ số thứ  $i$ -th. [15]

Cho  $d$  là chữ số tối đa của các phần tử trong mảng và  $b$  là cơ sở được sử dụng để biểu diễn mảng, ví dụ cho hệ thập phân:  $b = 10$ .

**Mã giả:** [1]

---

```
CountingSort(A, exp)
  n <- length(A)
  output <- new array of size n
  count <- new array of size 10
  for i <- 0 to 9 do
```

```

    count[i] <- 0
  for i <- 0 to n - 1 do
    index <- (A[i] / exp) % 10
    count[index] <- count[index] + 1
  for i <- 1 to 9 do
    count[i] <- count[i] + count[i - 1]
  for i <- n - 1 down to 0 do
    index <- (A[i] / exp) % 10
    output[count[index] - 1] <- A[i]
    count[index] <- count[index] - 1
  for i <- 0 to n - 1 do
    A[i] <- output[i]

```

```

RadixSort(A, d)
  for i <- 1 to d do
    CountingSort(A, i)

```

---

**Độ phức tạp thời gian:**  $O(d(n + b))$ . [15]

**Độ phức tạp không gian:**  $O(n)$ . [15]

## 2.11 Flash Sort

**Flash Sort** phát minh vào năm 1998 do tác giả Karl-Dietrich Neubert, sở dĩ tác giả tự tin đặt tên là "**Flash Sort**" có lẽ là bởi ông rất tự tin về tốc độ chớp nhoáng của thuật toán sắp xếp này. **Flash Sort** là một thuật toán sắp xếp tại trực tiếp trên mảng mang lại hiệu quả rất cao.

**Ý tưởng:** Thuật toán được chia thành 3 giai đoạn. [16] [16]

- Giai đoạn 1: Phân loại các phần tử của mảng.
- Giai đoạn 2: Phân hoạch các phần tử.
- Giai đoạn 3: Sắp xếp các phần tử trong mỗi phân đoạn.

### 2.11.1 Giai đoạn 1: Phân loại các phần tử của mảng

Xác định giá trị phần tử nhỏ nhất (min) và giá trị phần tử lớn nhất (max). Sau đó chia các phần tử vào m phân đoạn khác nhau, phần tử sẽ  $a[i]$  nằm ở phân đoạn thứ

$$k_{a_i} = \left\lfloor \frac{(m-1)(a_i - \min_a)}{\max_a - \min_a} \right\rfloor + 1. (m = 0.43n)$$

Từ công thức trên, sẽ cho biết phân lớp chứa phần tử  $i$  của mảng  $a$ .

**Mã giả:** [1]

---

```

FlashSort(a)
  n <- length(a)
  minVal <- a[0]
  maxIdx <- 0
  m <- int(0.43 * n)

```

```

L <- new array of size m
for i <- 0 to m do
    L[i] <- 0
for i <- 1 to n do
    if a[i] < minVal
        then minVal <- a[i]
    if a[i] > a[maxIdx]
        then maxIdx <- i
if a[maxIdx] != minVal
    then c1 <- 1.00 * (m-1) / (a[maxIdx] - minVal)
        for i <- 0 to n do
            k <- int (c1 * (a[i] - minVal))
            ++L[k]
        for i <- 0 to n do
            L[i] <- L[i] + L[i-1]

```

---

### 2.11.2 Giai đoạn 2: Phân hoạch các phần tử

Sau khi đã sẵn sàng, chúng ta bắt đầu việc sắp xếp các phần tử vào đúng phân lớp của nó. Việc này sẽ hình thành các chu trình hoán vị: mỗi khi ta đem một phần tử ở đâu đó đến một vị trí nào đó thì ta phải nhắc phần tử hiện tại đang chiếm chỗ ra, và tiếp tục với phần tử bị nhắc ra và đưa đến chỗ khác cho đến khi quay lại vị trí ban đầu thì hoàn tất vòng lặp.

**Mã giả:** [1]

```

swap a[maxIdx] and a[0]
nmove <- 0
j <- 0
k <- m - 1
t <- 0
flash <- 0
while nmove < n - 1 do
    while j > L[k] - 1 do
        j++
        k <- int(c1 * (a[j] - minVal))
    flash = a[j]
    if k < 0
        then break
    while j != L[k] do
        k <- int(c1 * (flash - minVal))
        L[k] <- L[k] - 1
        t <- L[k]
        hold <- a[t]
        a[t] <- flash
        flash <- hold
    nmove <- nmove + 1

```

---



### 2.11.3 Giai đoạn 3: Sắp xếp các phần tử trên mỗi phân đoạn

Mảng hiện tại đã được sắp xếp gần đúng vì các phần tử vào đúng lớp của nó. Nên để sắp xếp đúng hoàn toàn ta dùng **Insertion Sort** để tối ưu. Đoạn code này được viết đúng vì lớp cuối cùng chỉ chứa phần tử lớn nhất của mảng, do đó nó đã được sắp xếp.

**Mã giả:** [1]

---

```
InsertionSort(a)
```

---

**Độ phức tạp thời gian:** [16]

- Worst case:  $O(n^2)$ .
- Best case:  $O(n)$ .
- Average case:  $O(n)$ .

Các thử nghiệm đã chỉ ra rằng  $m \approx 0.43n$  sẽ là tối ưu nhất cho thuật toán này. Trong trường hợp đó, độ phức tạp thời gian của thuật toán là tuyến tính. [2]

**Độ phức tạp không gian:**  $O(1)$ . [8]

### 3 Kết quả thử nghiệm và nhận xét:

#### 3.1 Bảng thời gian chạy(milliseconds) và số lượng phép so sánh:

Data order: Randomized						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	182	100009999	1561	900029999	3980	2500049999
Insertion Sort	101	50302608	889	450209381	2234	1253347949
Bubble Sort	335	99983137	3270	899861085	8278	2500067240
Shaker Sort	230	67190336	2157	600960230	5918	1673434611
Shell Sort	0	639564	0	2216170	0	4459407
Heap Sort	0	630056	16	2126839	15	3732395
Merge Sort	17	583510	17	1937602	16	3382719
Quick Sort	1	281972	4	973604	0	1640381
Counting Sort	0	69994	1	210001	0	246875
Radix Sort	1	140056	3	510070	16	850070
Flash Sort	0	90262	1	269572	0	443299

Bảng 1 – Data order: Randomized - table 1

Data order: Randomized						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	16541	10000099999	153821	90000299999	401334	250000499999
Insertion Sort	10515	4986694011	86300	44987802112	240231	125224652729
Bubble Sort	37678	10000006401	351217	90000241200	1194990	249999454952
Shaker Sort	26389	6653363317	248891	60011345039	973042	166906931041
Shell Sort	32	10122943	102	33855780	324	64253642
Heap Sort	33	7964809	88	26247194	182	45568028
Merge Sort	31	7165779	95	23383595	283	40382293
Quick Sort	2	3487253	28	11619092	103	20564632
Counting Sort	0	365137	16	943450	0	1539209
Radix Sort	20	1700070	24	5100070	68	8500070
Flash Sort	4	827185	11	2495585	25	3963745

Bảng 2 – Data order: Randomized - table 2

Data order: Sorted						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	282	100009999	2206	900029999	6571	2500049999
Insertion Sort	0	19999	0	59999	0	99999
Bubble Sort	0	20001	0	60001	0	100001
Shaker Sort	0	20002	0	60002	0	100002
Shell Sort	0	360042	16	1170050	4	2100049
Heap Sort	0	655992	0	2197817	12	3859218
Merge Sort	0	475242	16	1559914	2	2722826
Quick Sort	0	154959	0	501929	0	913850
Counting Sort	0	70001	0	210001	0	350001
Radix Sort	0	140056	0	510070	16	850070
Flash Sort	0	117195	0	351595	11	585995

Bảng 3 – Data order: Sorted - table 1

Data order: Sorted						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	25098	10000099999	224951	90000299999	562745	250000499999
Insertion Sort	1	199999	3	599999	0	999999
Bubble Sort	1	200001	3	600001	8	1000001
Shaker Sort	2	200002	2	600002	5	1000002
Shell Sort	27	4500051	56	15300061	66	25500058
Heap Sort	25	8226253	127	27022363	159	46789063
Merge Sort	41	5745658	93	18645946	141	32017850
Quick Sort	16	1927691	14	6058228	32	10310733
Counting Sort	1	700001	4	2100001	19	3500001
Radix Sort	7	1700070	51	6000084	44	10000084
Flash Sort	3	1171995	28	3515995	15	5859995

Bảng 4 – Data order: Sorted - table 2

Data order: Reversed						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	157	100009999	1446	900029999	5542	2500049999
Insertion Sort	205	100019998	1600	900059998	5161	2500099998
Bubble Sort	282	100019998	2599	900059998	9136	2500099998
Shaker Sort	347	100005001	2758	900015001	8065	2500025001
Shell Sort	0	475175	3	1554051	9	2844628
Heap Sort	0	605098	5	2055681	16	3606091
Merge Sort	0	476441	6	1573465	15	2733945
Quick Sort	0	164975	1	531939	0	963861
Counting Sort	0	70001	1	210001	0	350001
Radix Sort	0	140056	2	510070	0	850070
Flash Sort	0	100706	1	302106	0	503506

Bảng 5 – Data order: Reversed - table 1

Data order: Reversed						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	17448	10000099999	158517	90000299999	392029	250000499999
Insertion Sort	19019	10000199998	214323	90000599998	414603	250000999998
Bubble Sort	30902	10000199998	236720	90000599998	711629	250000999998
Shaker Sort	31102	10000050001	257200	90000150001	653671	250000499999
Shell Sort	17	6089190	44	20001852	63	33857581
Heap Sort	31	7700828	52	25507884	89	44375011
Merge Sort	16	5767897	47	18708313	100	32336409
Quick Sort	15	2027703	16	6358249	28	10810747
Counting Sort	0	700001	0	2100001	0	3500001
Radix Sort	0	1700070	16	6000084	47	10000084
Flash Sort	0	1007006	17	3021006	15	5035006

Bảng 6 – Data order: Reversed - table 2

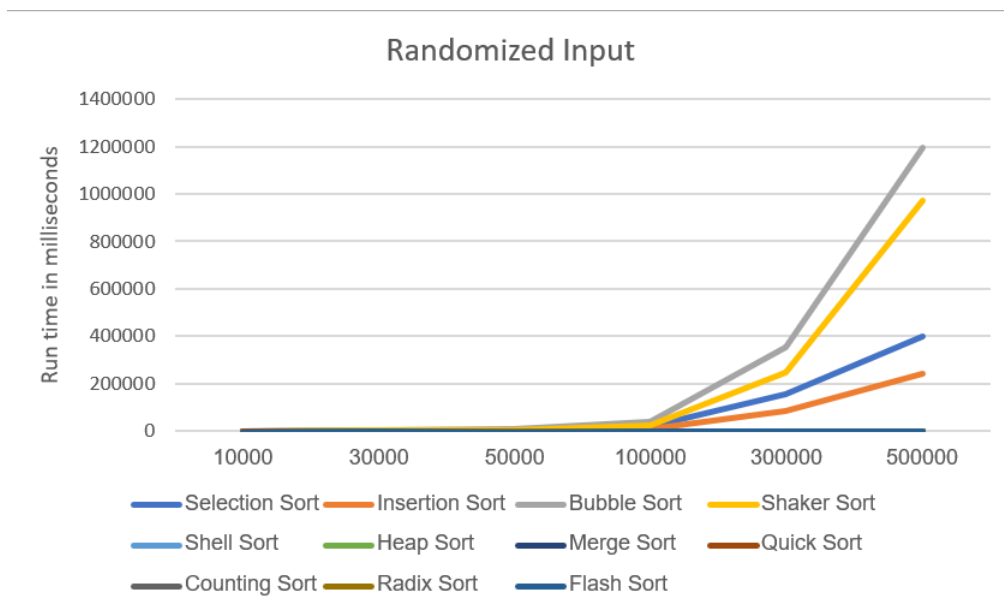
Data order: Nearly Sorted						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	142	100009999	1246	900029999	3451	2500049999
Insertion Sort	0	147499	0	480359	0	676111
Bubble Sort	122	88676577	1112	836060001	2809	2063248200
Shaker Sort	1	167454	0	534935	0	714954
Shell Sort	1	404154	0	1324072	16	2284204
Heap Sort	1	655788	0	2197806	11	3859086
Merge Sort	2	504718	16	1644498	5	2827582
Quick Sort	0	155007	0	501973	0	913890
Counting Sort	0	70001	0	210001	0	350001
Radix Sort	1	140056	0	510070	0	850070
Flash Sort	0	117171	0	351563	0	585969

Bảng 7 – Data order: Nearly Sorted - table 1

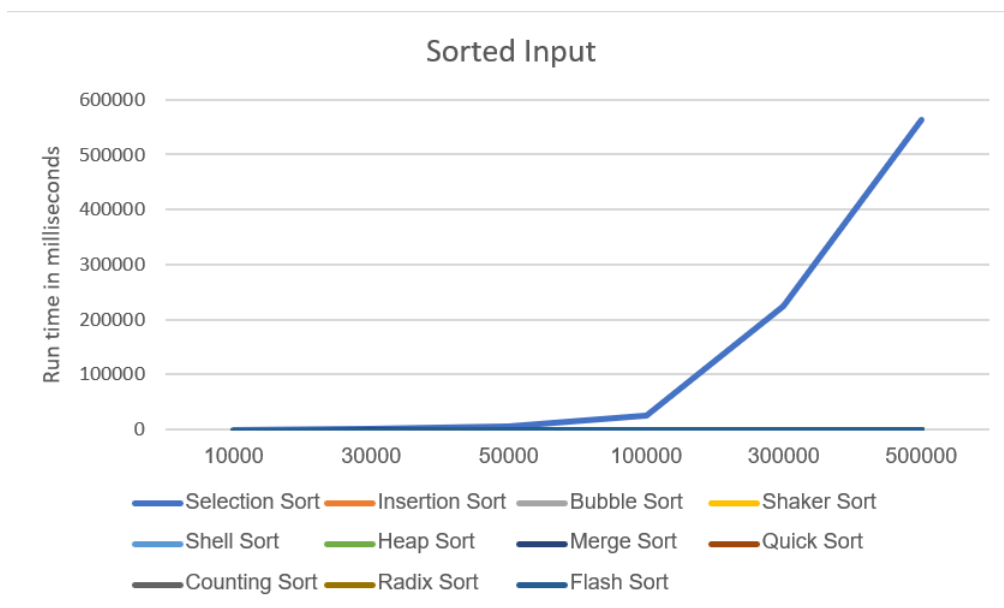
Data order: Nearly Sorted						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	13885	10000099999	122773	90000299999	406783	250000499999
Insertion Sort	0	640251	0	1018199	0	1413819
Bubble Sort	5176	3674224705	14980	10423575165	31734	21646041776
Shaker Sort	0	683670	1	1100801	15	1473289
Shell Sort	15	4682352	16	15441191	48	25635167
Heap Sort	16	8226395	63	27022394	88	46789147
Merge Sort	16	5850110	47	18747209	107	32125353
Quick Sort	0	1927731	25	6058268	15	10310757
Counting Sort	15	700001	4	2100001	0	3500001
Radix Sort	7	1700070	25	6000084	57	10000084
Flash Sort	0	1171967	0	3515973	15	5859963

Bảng 8 – Data order: Nearly Sorted - table 2

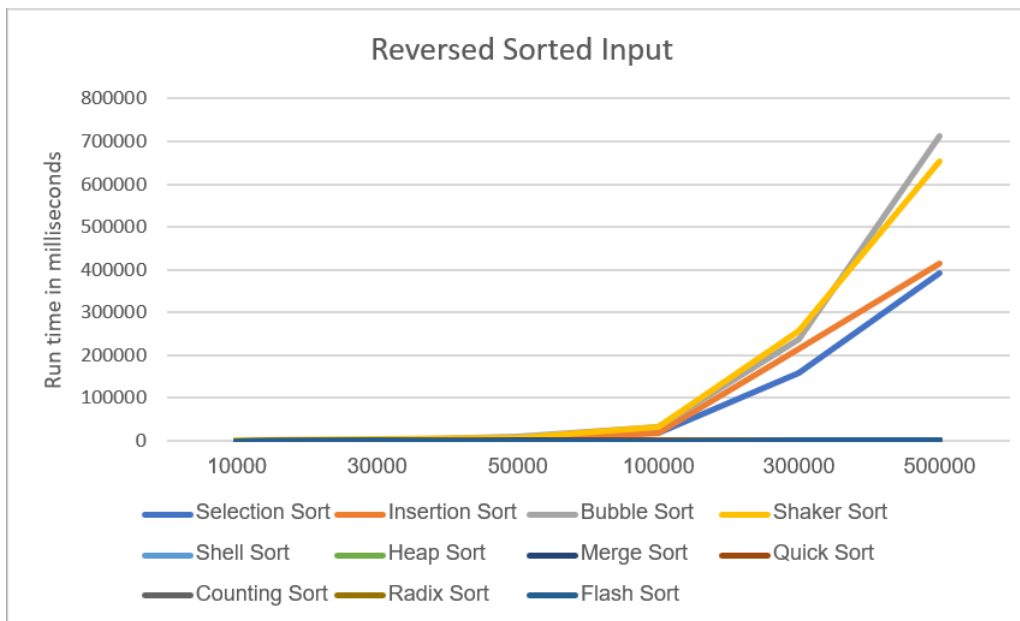
### 3.2 Biểu đồ thời gian chạy:



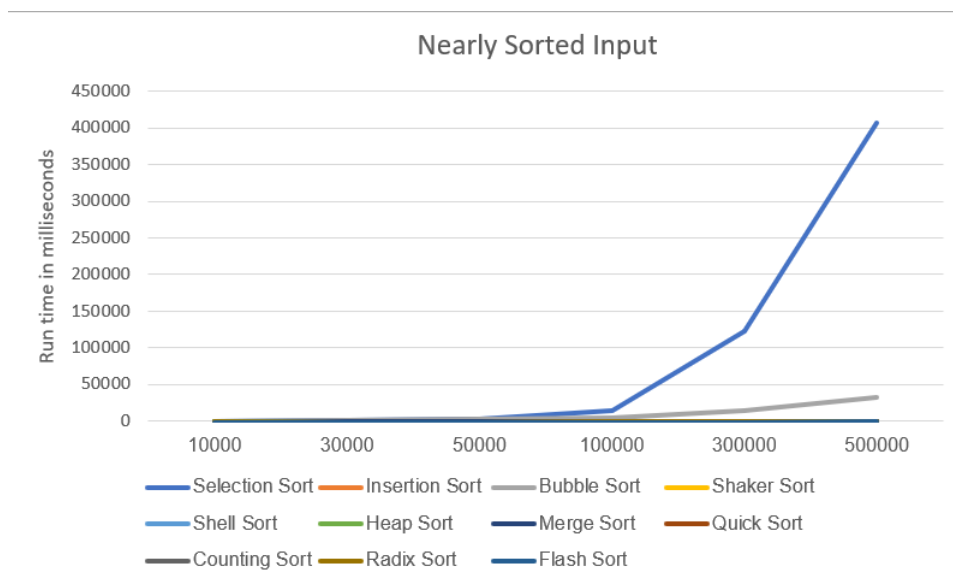
Hình 2 – Biểu đồ thời gian chạy cho Randomized Input



Hình 3 – Biểu đồ thời gian chạy cho Sorted Input

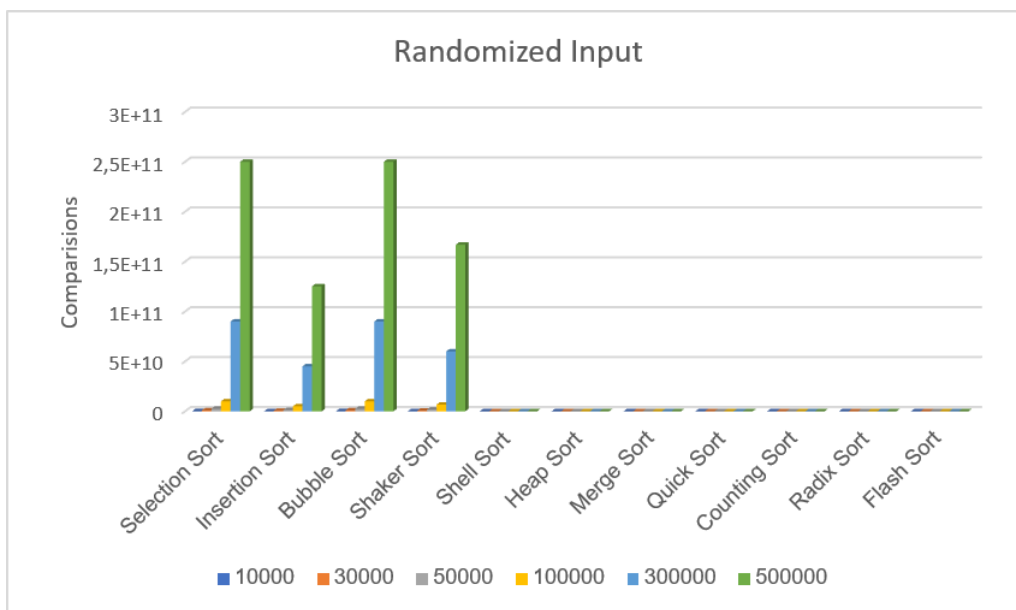


Hình 4 – Biểu đồ thời gian chạy cho Reversed Input

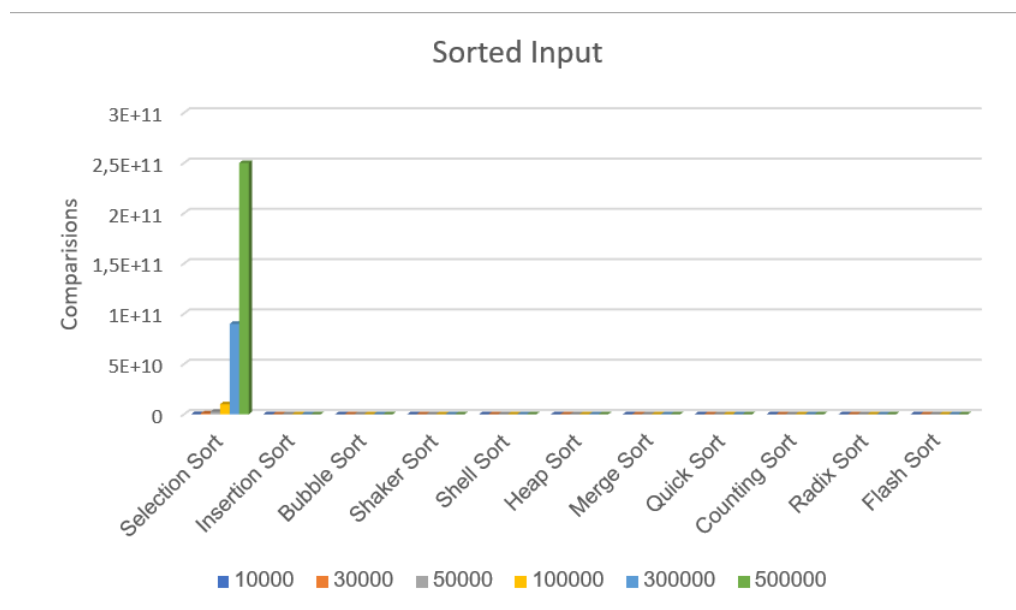


Hình 5 – Biểu đồ thời gian chạy cho Nearly Sorted Input

### 3.3 Biểu đồ phép so sánh:

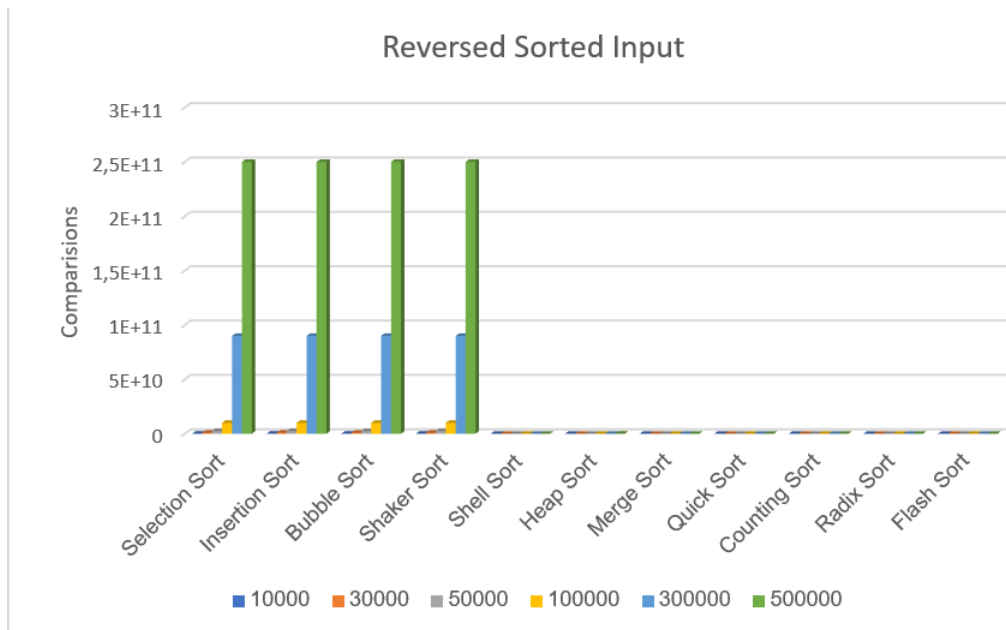


Hình 6 – Biểu đồ số phép so sánh cho Randomized Input

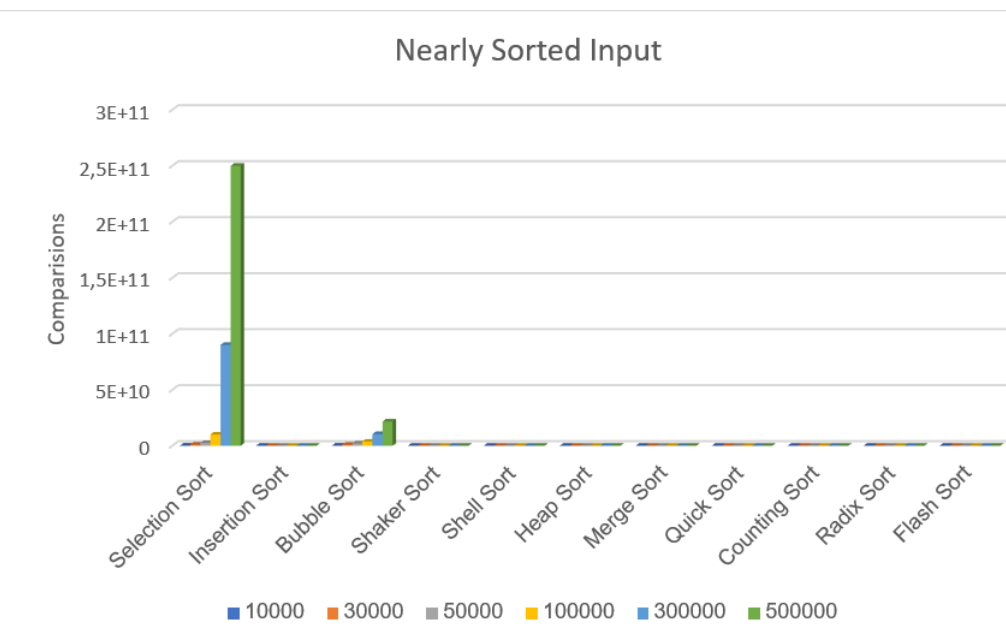


Hình 7 – Biểu đồ số phép so sánh cho Sorted Input





Hình 8 – Biểu đồ số phép so sánh cho Reversed Input



Hình 9 – Biểu đồ số phép so sánh cho Nearly Sorted Input

### 3.4 Comment:

Biểu đồ thời gian chạy cho Randomized Input:

- Đối với số lượng dữ liệu từ 50000 trở xuống thì hầu hết các thuật toán sắp xếp có thời gian chạy thực tế gần như nhau.
- Khi số lượng dữ liệu từ 50000 trở lên thì ta sẽ thấy rõ được các thuật toán sắp xếp cơ bản sẽ chạy chậm hơn như là Bubble Sort sẽ chậm nhất sau đó tới Shaker Sort rồi tới Selection Sort và Insertion Sort, còn các thuật toán sắp xếp nâng cao khác hầu như không thay đổi.

Biểu đồ thời gian chạy cho Sorted Input:

- Ta thấy hầu hết các thuật toán sắp xếp đều có thời gian chạy khi có Sorted Input rất nhanh, trừ Selection Sort có thời gian chạy thực tế cực kì chậm và thời gian chạy càng tăng nhanh đột biến ở lượng dữ liệu từ 50000 trở lên

Biểu đồ thời gian chạy cho Reversed Input:

- Đối với số lượng dữ liệu từ 50000 trở xuống thì hầu hết các thuật toán sắp xếp có thời gian chạy thực tế gần như nhau.
- Đối với số lượng dữ liệu từ 50000 trở lên thì đã có sự chênh lệch khi thời gian chạy của các thuật toán Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort tăng nhanh đột biến so với các thuật toán còn lại.
- Khi số lượng dữ liệu càng tăng thì ta thấy thời gian chạy của thuật toán Bubble Sort là lớn nhất, kế đến lần lượt là Shaker Sort, Insertion Sort, Selection Sort.

Biểu đồ thời gian chạy cho Nearly Sorted Input:

- Ta thấy Selection Sort là không phụ thuộc vào dữ liệu đầu vào nên thuật toán này có thời gian chạy là lâu nhất. Còn Bubble Sort thì nhanh hơn khá nhiều so với kiểu dữ liệu random. Vì có ưu thế khi dữ liệu đã gần như được sắp xếp nên các thuật toán sắp xếp nâng cao sẽ có thời gian chạy nhanh hơn. Các thuật toán sắp xếp không sử dụng phép so sánh cũng có thời gian chạy rất nhanh.

Biểu đồ số phép so sánh cho Randomized Input:

- Các thuật toán sắp xếp cơ bản sử dụng rất nhiều các phép so sánh trong đó 2 thuật toán sử dụng số lượng phép so sánh nhiều nhất là Selection Sort và Bubble Sort, tiếp đến là Shaker Sort và Insertion Sort.
- Các thuật toán sắp xếp nâng cao hay còn gọi là thuật toán sắp xếp ổn định sử dụng rất ít các phép so sánh.

Biểu đồ số phép so sánh cho Sorted Input:

- Cũng như thời gian chạy, số phép so sánh của Selection cũng nhiều hơn so với các thuật toán sắp xếp còn lại, và có sự tăng đột biến thời gian từ lượng dữ liệu 100000 lên 300000 và từ 300000 lên 500000

Biểu đồ số phép so sánh cho Reversed Input:

- Đối với số lượng dữ liệu từ 50000 trở xuống thì hầu hết các thuật toán sắp xếp có số phép so sánh gần như nhau.
- Từ mức dữ liệu 50000 trở lên thì có sự chênh lệch số phép so sánh giữa các thuật toán.

Biểu đồ số phép so sánh cho Nearly Sorted Input:

- Khi thời gian chạy càng lâu thì Selection Sort sẽ có số phép so sánh càng nhiều. Các thuật toán còn lại thì có ưu thế hơn vì dữ liệu đã gần như sắp xếp nên nó sẽ có thời gian chạy nhanh hơn và tất nhiên số phép so sánh cũng sẽ ít hơn.

### 3.5 Kết luận:

Qua những thống kê và nhận xét ở trên, tôi hi vọng các bạn đã có cho bản thân câu trả lời **Đâu mới là thuật toán sắp xếp tốt nhất?**. Còn với nhóm tôi câu trả lời đó là: Không có thuật toán sắp xếp nào là tốt nhất, nó chỉ đơn giản là sự đánh đổi giữa không gian, thời gian và công sức bỏ ra để xây dựng thuật toán, chính vì vậy tùy thuộc vào mỗi loại dữ liệu đầu vào, không gian bộ nhớ cho phép, tốc độ cần thực thi, cấu hình máy thực hiện thuật toán mà lựa chọn thuật toán cho phù hợp.

Mặc dù dữ liệu thống kê cho ta thấy QuickSort có số lượng phép so sánh nhỏ và thời gian chạy rất ngắn nhưng trong trường hợp xấu nhất, thuật toán Quick Sort có thể trở nên rất chậm và tốn nhiều bộ nhớ.

Với kích thước dữ liệu đầu vào nhỏ nhìn chung tốc độ chênh lệch của các thuật toán là không rõ để nhận thấy. Khi kích thước dữ liệu lớn thì ta sẽ thấy được rõ ràng về thời gian chạy của từng thuật toán. Nhanh nhất có lẽ là **Flash Sort** và chậm nhất là **Bubble Sort**.

Với mảng đã được sắp xếp, thì **Bubble Sort** và **Shaker Sort** cho tốc độ nhanh nhất do chi phí để biết được đây là mảng có thứ tự của 2 thuật toán trên là  $O(n)$ . Và các thuật toán khác hầu như cũng cho ra tốc độ nhanh hơn đáng kể.

**Dựa vào sự thay đổi vị trí tương đối, ta có 2 loại:**

- Stable Sorts: **Selection Sort, Shell Sort, Merge Sort, Radix Sort, Flash Sort.**
- Unstable Sorts: **Insertion Sort, Bubble Sort, Shaker Sort, Quick Sort, Counting Sort.**

**Dựa vào không gian bộ nhớ, ta có 2 loại:**

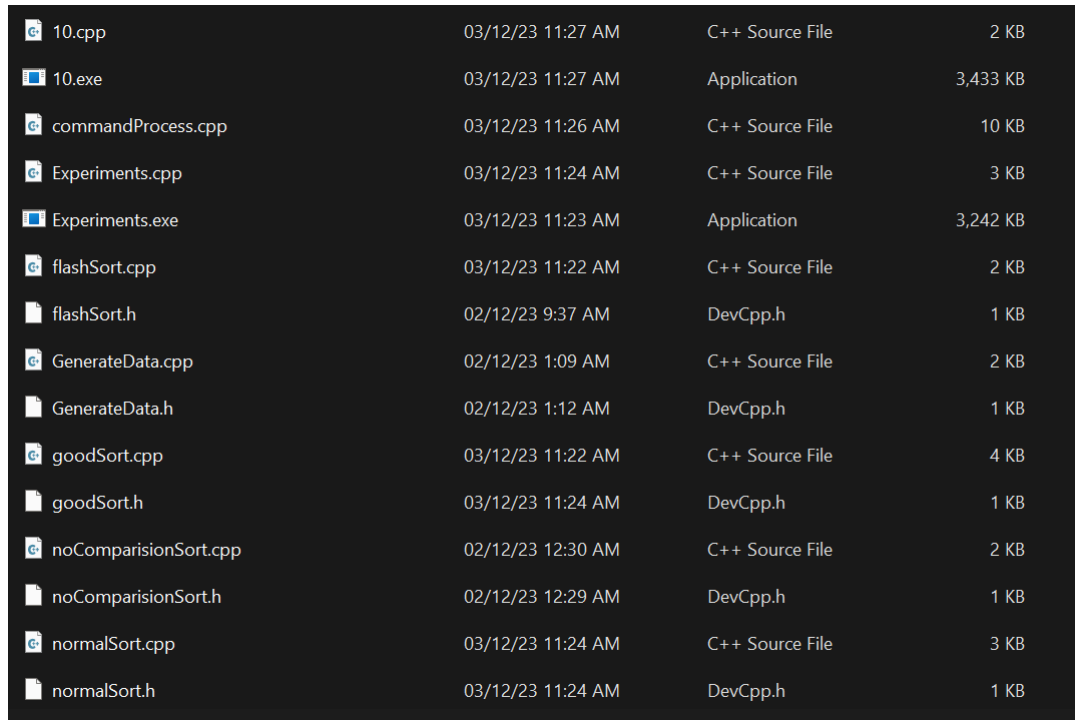
- In-place Sorts: **Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Heap Sort, Flash Sort, Shell Sort, Shaker Sort**
- Not-in-place Sorts: **Counting Sort, Merge Sort, Radix Sort**

Tuy vậy trong một vài trường hợp, cách phân loại có thể không chính xác tùy vào cách người lập trình xây dựng thuật toán hay tùy vào Input bài toán

## 4 Tổ chức dự án và ghi chú lập trình:

### 4.1 Tổ chức dự án:

Cách thức tổ chức dự án theo file **.h** và **.cpp**



10.cpp	03/12/23 11:27 AM	C++ Source File	2 KB
10.exe	03/12/23 11:27 AM	Application	3,433 KB
commandProcess.cpp	03/12/23 11:26 AM	C++ Source File	10 KB
Experiments.cpp	03/12/23 11:24 AM	C++ Source File	3 KB
Experiments.exe	03/12/23 11:23 AM	Application	3,242 KB
flashSort.cpp	03/12/23 11:22 AM	C++ Source File	2 KB
flashSort.h	02/12/23 9:37 AM	DevCpp.h	1 KB
GenerateData.cpp	02/12/23 1:09 AM	C++ Source File	2 KB
GenerateData.h	02/12/23 1:12 AM	DevCpp.h	1 KB
goodSort.cpp	03/12/23 11:22 AM	C++ Source File	4 KB
goodSort.h	03/12/23 11:24 AM	DevCpp.h	1 KB
noComparisionSort.cpp	02/12/23 12:30 AM	C++ Source File	2 KB
noComparisionSort.h	02/12/23 12:29 AM	DevCpp.h	1 KB
normalSort.cpp	03/12/23 11:24 AM	C++ Source File	3 KB
normalSort.h	03/12/23 11:24 AM	DevCpp.h	1 KB

Hình 10 – Files trong dự án

- Trong **normalSort** bao gồm các thuật toán sắp xếp cơ bản có độ phức tạp trung bình là  $O(n^2)$ . Thường rất ít được sử dụng.
- Trong **goodSort** bao gồm các thuật toán sắp xếp tốt gồm: **Merger Sort**, **Heap Sort**, **Quick Sort** có độ phức tạp trung bình là  $O(n \log n)$ . Thường được sử dụng nhiều đặc biệt là **Quick Sort**.
- Trong **noComparisionSort** gồm 2 thuật toán: **Counting Sort** và **Radix Sort** sắp xếp không dựa vào phép so sánh. Nhưng lại có nhiều nhược điểm như: tốn nhiều bộ nhớ và hạn chế về mặt kiểu dữ liệu nên cũng thường rất ít được sử dụng.
- **Flash Sort** là một thuật toán cho tốc độ nhanh(thậm chí nhanh hơn cả **Quick Sort**) và tiêu tốn rất ít bộ nhớ, tuy nhiên cách thức xây dựng thuật toán trên khá phức tạp
- Ngoài ra còn có một vài file khác như: **10.exe** (file chạy các thuật toán bằng CommandLine), file **Experiments** dùng để thực nghiệm lấy ra các số liệu về thời gian chạy và số phép so sánh.

## 4.2 Ghi chú lập trình:

Trong dự án lần này nhóm tôi có sử dụng cấu trúc dữ liệu Map. Dùng để lưu các cặp Key: "Tên của thuật toán sắp xếp" và Value: "Con trỏ hàm của thuật toán sắp xếp tương ứng". Nhằm mục đích để viết CommandLine thuận tiện hơn, tránh việc phải xét nhiều trường hợp. Điều này hoàn toàn không ảnh hưởng gì đến các thuật toán sắp xếp. Trình biên dịch sử dụng là **g++** và phiên bản **C++ 17** trở lên.

## Tài liệu

- [1] Le Minh Hoang (2002) *Giai thuật và lập trình*, Ha Noi University of Education Press
- [2] Lectures from Dr. Văn Chí Nam
- [3] <https://blog.luyencode.net/thuat-toan-sap-xep-selection-sort/>
- [4] <https://www.geeksforgeeks.org/insertion-sort/>
- [5] <https://www.geeksforgeeks.org/bubble-sort/>
- [6] <https://thuvienhuongdan.com/thuat-toan-sap-xep-bubble-sort-shak.html>
- [7] <https://www.tutorialspoint.com/Shell-sort>
- [8] <https://www.programiz.com/dsa/heap-sort>
- [9] <https://www.educative.io/blog/data-structure-heaps-guide>
- [10] <https://www.programiz.com/dsa/merge-sort>
- [11] <https://viblo.asia/p/thuat-toan-sap-xep-nhanh-quick-sort-eW65G>
- [12] <https://www.geeksforgeeks.org/iterative-quick-sort/>
- [13] <https://www.geeksforgeeks.org/counting-sort/>
- [14] <https://www.interviewcake.com/concept/java/counting-sort>
- [15] <https://www.geeksforgeeks.org/radix-sort/>
- [16] <https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-tha>
- [17] <https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot>