ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

# BÁO CÁO THỰC HÀNH
# CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

## Các giải thuật sắp xếp

Họ và tên: Chung Hoàng Tuấn Kiệt
MSSV: 19120553
Lớp: 19CNTN
Giảng viên: Bùi Huy Thông

Tháng 12 năm 2020

# I. Thuật toán

## 1. Selection sort (Sắp xếp chọn)

### 1.1 Ý tưởng

- Ý tưởng của thuật toán là chuyển lần lượt các giá trị nhỏ nhất lên trên đầu mảng và tiếp tục duyệt đến khi hết mảng. Các bước cụ thể như sau:

  + Bước 1: Tại vị trí i của mảng ta gán giá trị của phần tử tại đó vào biến pos (position: vị trí).

  + Bước 2: Duyệt từ vị trí i đến hết mảng nếu phần tử nào nhỏ hơn phần tử thứ pos thì gán vị trí đó vào biến pos.

  + Bước 3: Đổi vị trí phần tử thứ i và phần tử có vị trí pos

  + Bước 4: Quay lại bước 1 và tăng giá trị i. Lặp đến khi duyệt hết mảng.

### 1.2 Mã giả

```
Selection_sort(array,n)
 for i: (0→n - 1)
    pos = i
    for j: (i+1→n - 1)
            if array[j] < array[pos]
                    pos = j
    swap(array[i],array[pos])
```

Ví dụ: Cho mảng 5 phần tử

| 9 | 7 | 3 | 5 | 1 |
|---|---|---|---|---|

Ta thực hiện sắp xếp bằng Selection Sort:

- Lưu vị trí của phần tử đang xét vào pos, lúc này pos = 0. Duyệt lần lượt qua các vị trí. Tại 7 do 7 bé hơn 9 nên pos = 1. Sau đó tiếp tục xét. Do 3 bé hơn 7 nên pos = 2. Đến cuối mảng thì 1 bé hơn 3 nên pos = 4. Sau đó thực hiện hoán đổi array[i] và array[pos]. Mảng lúc này là:

| 1 | 7 | 3 | 5 | 9 |
|---|---|---|---|---|

- Tại vị trí i = 1. Ta lưu pos = 1. Tiếp tục duyệt như trên. Sau lần duyệt này pos = 2. Ta thực hiện hoán đổi và được mảng mới:

| 1 | 3 | 7 | 5 | 9 |
|---|---|---|---|---|

- Ta lập lại thao tác trên đến khi duyệt hết mảng.

### 1.3 Đánh giá thuật toán

- Khi kích thước của mảng đủ lớn thì thuật toán Selection Sort chạy khá lâu do số thao tác cần thực hiện để duyệt mảng là n(n+1)/2 với n là kích thước của mảng.
- Đối với mảng đã được sắp thì số thao tác duyệt mảng vẫn cố định là n(n+1)/2. Vậy nên, dù mảng đã được sắp xếp thì khi dung Selection Sort chi phí duyệt mảng vẫn cố định và sẽ phát sinh thêm n bước để sắp xếp lại toàn bộ mảng (không đáng kể khi n đủ lớn) nên thời gian sắp xếp xấp xỉ nhau đối với các bộ dữ liệu có cùng kích thước.
- Ở trường hợp Generated Reversed Sorted Data (mảng đã được sắp xếp và bị đảo ngược) thì thuật toán Selection Sort có thời gian xử lý nhanh hơn so với các bộ dữ liệu có cùng kích thước khác. Trong trường hợp Generated Reversed Sorted Data, toàn bộ các lệnh if đều đúng (do với mọi vị trí array[min] > array[j]) nên code linear (tuyến tính) hơn nên ở trường hợp này lại nhanh hơn các trường hợp khác (Do số thao tác duyệt mảng là cố định).

## 2. Insertion sort (Sắp xếp chèn)

### 2.1 Ý tưởng

- Ý tưởng của thuật toán là xem như mảng có 1 phần tử đã được sắp xếp và sau đó lần lượt chèn các phần tử khác vào mảng phía trước phần tử đó sau cho không xuất hiện nghịch thế (phần tử đứng trước lớn hơn phần tử đứng sau). Các bước cụ thể như sau:

  + Bước 1: Khởi tạo mảng với dãy con đã sắp xếp có k (k = 1) phần tử(phần tử đầu tiên của mảng).

  + Bước 2: Duyệt lần lượt các phần tử trong mảng. Tại mỗi phần tử thứ i lưu lại giá trị i – 1 vào biến pos (pos (position): vị trí) và giá trị phần tử tại đó vào biến key.

  + Bước 3: Thực hiện vòng lập while, trong khi biến pos lớn hơn hoặc bằng 0 và biến key bé hơn phần tử tại vị trí pos thì gán phần tử tại vị trí pos + 1bằng phần tử tại vị trí pos và giảm giá trị pos đi 1 đơn vị.

  + Bước 4: Gán phần tử tại vị trí pos + 1 bằng key và quay lại bước 2.

## 2.2 Mã giả

```
Insertion_sort(array,n)
 for i: (1→n - 1)
    pos = i - 1
    key = array[pos]
    while (pos >= 0 & key < array[pos])
          array[pos+1] = array[pos]
          pos = pos – 1
    array[pos+1] = key
```

Ví dụ: : Cho mảng 5 phần tử

| 9 | 7 | 3 | 5 | 1 |
|---|---|---|---|---|

Ta thực hiện sắp xếp bằng Insertion Sort:

- Ta xem như mảng gồm 1 phần tử là đã được sắp xếp (ở mảng là là mảng 1 phần tử là số 9).
- Ta bắt đầu duyệt từ vị trí i = 1.
- Ta lưu vị trí liền trước vị trí i vào biến pos và giá trị tại đó vào biến key (lúc này pos = 0 và key = 7). Ta xét thấy thỏa điều kiện trong vòng lặp while (do cả pos >= 0 và key < array[pos]) nên thực hiện chèn gán array[1] = array[0]. Sau đó thoát vòng while và gán lại array[0] = 7. Mảng lúc này là:

| 7 | 9 | 3 | 5 | 1 |
|---|---|---|---|---|

- Ở lần duyệt tiếp theo ta được kết quả:

| 3 | 7 | 9 | 5 | 1 |
|---|---|---|---|---|

- Ta thực hiện duyệt đến cuối mảng thì mảng sẽ được sắp xếp.

### 2.3 Đánh giá thuật toán

- Thuật toán này có độ phức tạp là $O(n^2)$ nhưng nó tối ưu hơn thuật toán Selection Sort là ở các mảng gần như được sắp thì chi phí để sắp xếp sẽ ít hơn. Thuật toán này chỉ dừng lại để duyệt khi phần tử đó nhỏ hơn phần tử liền trước nó. Do đó chi phí dùng để sắp xếp mảng sẽ ít hơn.
- Đối với các mảng đã sắp sẵn hoặc gần như được sắp thì thời gian duyệt mảng là O(n).
- Trong trường hợp Generated Reversed Sorted Data (mảng đã được sắp xếp và bị đảo ngược) thời gian duyệt gần như bằng với thuật toán Selection Sort là $O(n^2)$. Các thuật toán này đều nằm trong nhóm thuật toán sắp xếp cơ bản.

## 3. Binary Insertion Sort (Sắp xếp chèn nhị phân)

### 3.1 Ý tưởng

- Tương tự thuật toán Insertion Sort (chèn các phần tử vào mảng đã được sắp xếp ở phía trước nó) nhưng Binary Insertion Sort sử dụng tìm kiếm nhị phân để tìm ra vị trí cần phải chèn vào thay vì chèn tuần tự. Các bước:
  + Bước 1: Tại vị trí i (duyệt mảng lần đầu thì i = 0) lưu lại phần tử đó vào biến key và vị trí liền trước nó (i – 1) và biến pos.
  + Bước 2: Sử dụng tìm kiếm nhị phân để tìm vị trí cần chèn vào mảng đã được sắp ở phía trước phần tử đó.
  + Bước 3: Chèn phần tử đến vị trí đã tìm ra, tăng i lên một đơn vị và quay lại bước 1.

### 3.2 Mã giả

```
Binary_Insertion_Sort(array,n)
 for i: (1→n-1)
    left = 0
    right = i – 1
    mid = (right + left)/2
    key = array[i]
    pos = i - 1
    while (left <= right)
            mid = (right + left)/2
            if (key > array[mid])
                    left = mid + 1
            else
                    right = mid – 1
    while (pos >= left)
            array[pos+1] = array[pos]
            pos = pos – 1
    array[pos] = key
```

Ví dụ: Cho mảng 5 phần tử:

| 9 | 7 | 3 | 5 | 1 |
|---|---|---|---|---|

- Ta thực hiện sắp xếp mảng bằng Binary Insertion Sort.
- Ta bắt đầu duyệt từ vị trí i = 1.
- Ta gán left = 0, right = i – 1, pos = i – 1 và key = array[i].
- Ở lần duyệt đầu tiên ta được key = 7, left = right = 0, pos = 0. Sau khi duyệt qua vòng lặp while ta có 7 < 9 nên left = 0, right = -1 và thoát vòng lặp.
- Sau đó ta thực hiện sắp xếp chèn như Insertion Sort và ta được mảng mới:

| 7 | 9 | 3 | 5 | 1 |
|---|---|---|---|---|

- Ta thực hiện duyệt tuần tự như thế đến cuối mảng thì mảng sẽ được sắp xếp.

### 3.3 Đánh giá thuật toán

- Thuật toán Binary Insertion Sort có số thao tác chèn phần tử vào mảng là tương đương thuật toán Insertion Sort.
- Thuật toán Binary Insertion Sort sử dụng thao tác tìm kiếm nhị phân nên vòng lặp while khi thực hiện chèn phần tử vào mảng chỉ cần xét một điều kiện (pos >= left) nên số thao tác xét điều kiện để thực hiện vòng lặp sẽ giảm đi một nửa so với Insertion Sort phải xét song song hai điều kiện (pos >= 0 & key < array[pos]) nên thuật toán thực thi nhanh hơn.
- Đối với dữ liệu lớn thì thuật toán này vẫn chạy khá chậm khi gặp trường hợp xấu do đó thuật toán này vẫn được sắp vào nhóm thuật toán sắp xếp cơ bản.

## 4. Bubble Sort (Sắp xếp nổi bọt)

### 4.1 Ý tưởng:

- Bubble Sort (sắp xếp nổi bọt) ta sẽ duyệt tuần tự qua các vị trí. Sau mỗi lần duyệt phần tử nhỏ sẽ được chuyển lên đầu mảng. Các bước thực hiện như sau:
  + Bước 1: Tại vị trí i của mảng ta duyệt từ cuối mảng trở lên vị trí i.
  + Bước 2: Xét xem nếu tại đó phần tử đừng trước lớn hơn phần tử liền sau thì ta hoán đổi vị trí.
  + Bước 3: Quay lại bước 1

### 4.2 Mã giả

```
Bubble_Sort(array,n)
 for i: (0→n-2)
    for j: (n - 1→i+1)
          if (array[j] < array[j – 1])
                swap(array[j],array[j-1])
```

Ví dụ: Cho mảng 5 phần tử:

| 9 | 7 | 3 | 5 | 1 |
|---|---|---|---|---|

- Ta duyệt từ vị trí i = 0.
- Ở mỗi lần duyệt gán j = n – 1. Duyệt đến khi j < i + 1 thì dừng. Sau đó duyệt đến vị trí kế tiếp. Lặp như thế đến cuối mảng.
- Ở lần duyệt đầu tiên ta gán i = 0, j = 4. Ta xét thấy array[4] < array[3] nên ta hoán đổi vị trí. Mảng lúc này sẽ là:

| 9 | 7 | 3 | 1 | 5 |
|---|---|---|---|---|

- Ta tiếp tục xét thấy array[3] < array[2]. Nên ta tiếp tục hoán đổi. Ta thực hiện như thế đến khi j = i + 1 (lúc này sẽ duyệt đến j = 1). Sau khi duyệt lần đầu ta được:

| 1 | 9 | 7 | 3 | 5 |
|---|---|---|---|---|

- Ta thực hiện như thế đến cuối mảng.

### 4.3 Đánh giá thuật toán

- Thuật toán Bubble Sort thuộc vào nhóm thuật toán sắp xếp cơ bản với độ phức tạp $O(n^2)$.
- Trong toàn bộ quá trình biến j thay vì tăng lên thì lại giảm đi (do thuật toán này duyệt ngược từ cuối mảng đến vị trí đang xét) nên tốc độ của thuật toán chậm hơn (do toán tử trừ thực hiện lâu hơn toán tử cộng). Do đó khi thực hiện trên dữ liệu lớn ta sẽ thấy sự khác biệt của Bubble Sort so với các thuật toán cùng nhóm khác.
- Ở các dữ liệu đã được sắp thì chi phí cần duyệt qua vẫn cố định là n(n+1)/2 (với n là kích thước mảng). Do vậy vẫn cần nhiều thời gian để duyệt hết các mảng đã được sắp sẵn.

## 5. Shaker Sort (Sắp xếp rung lắc)

### 5.1 Ý tưởng

- Shaker Sort là một cải tiến của Bubble Sort. Sau khi đưa phần tử nhỏ nhất về đầu dãy 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 giúp cải thiện thời gian sắp xếp dãy số do giảm được độ lớn của mảng đang xét ở lần so sánh kế tiếp. Các bước thực hiện:
  + Bước 1: Gán left = 0 và right = n – 1 (n là kích thước mảng).
  + Bước 2: Trong khi left bé lơn right ta thực hiện 2 vòng lập để đưa phần tử nhỏ nhất lên đầu dãy và phần tử lớn nhất xuống cuối dãy.
  + Bước 3: Cho biến i chạy từ đoạn left lên right – 1. Nếu phần tử nào lớn hơn phần tử liền sau nó thì đổi chổ và lưu lại vị trí phần tử đó vào biến k.
  + Bước 4: Gán right = k.
  + Bước 5: Cho biến i chạy từ right xuống left + 1. Nếu phần tử nào bé hơn phần tử liền trước nó thì đổi chổ và lưu lại vị trí đó vào biến k.
  + Bước 6: Gán left = k và quay lại bước 2.

### 5.2 Mã giả

```
Shaker_Sort(array,n)
 left = 0
 right = n – 1
 while (left <= right){
    for i: (left→right - 1)
            if (array[i]>array[i+1]){
                    swap(array[i],array[i+1])
                    k = i
            }
    right = k
    for i: (right→left+1)
            if (array[i] <array[i – 1]){
                    swap(array[i],array[i-1])
                    k = i
            }
    left = k
```

Ví dụ: Cho mảng 5 phần tử:

| 9 | 7 | 3 | 5 | 1 |
|---|---|---|---|---|

- Ở lần duyệt đầu tiên ta gán left = 0 và right = 4.
- Ở vòng for thứ nhất ta thấy array[0] > array[1] nên ta thực hiện hoán đổi và gán k = 0. Tiếp tục xét thấy array[1] > array[2] nên tiếp tục hoán đổi và gán k = 1. Ta thực hiện hết vòng for thứ nhất thì k = 3 và mảng lúc này là:

| 7 | 3 | 5 | 1 | 9 |
|---|---|---|---|---|

- Sau đó gán right = k. Lúc này right = 3. Ta thực hiện vòng for thứ hai, ta duyệt từ i = 3 đến vị trí i = 1. Ta xét thấy array[3] < array[2] nên ta hoán đổi và gán k = 3. Ta tiếp tục duyệt sau vòng for thứ hai ta được k = 1 và mảng lúc này là:

| 1 | 7 | 3 | 5 | 9 |
|---|---|---|---|---|

- Ta gán left = k. Lúc này left = 1. Sau lần duyệt đầu tiên ta có left = 1 và right = 3. Vùng mảng cần sắp xếp đã nhỏ hơn. Ta tiếp tục duyệt đến khi left > right thì dừng.

### 5.3 Đánh giá thuật toán

- Thuật toán sẽ thu hẹp vùng mảng cần sắp lại sau mỗi lần duyệt nên khi gặp các mảng đã sắp sẵn thì chi phí duyệt mảng chỉ mất O(n).
- Thuật toán là cải tiến của Bubble Sort nên sẽ hiệu quả hơn Bubble Sort. Thời gian cần để xử lý các mảng không tăng là $O(n^2)$ nhưng do có khả năng thu hẹp mảng đã được sắp nên đối với các trường hợp mảng không tăng thì Shaker Sort chạy nhanh hơn Bubble Sort.
- Đối với các mảng gần như được sắp thì thời gian chạy rất ngắn do sau mỗi lần duyệt vùng mảng cần sắp đã được thu hẹp lại.

## 6. Shell Sort

### 6.1 Ý tưởng

- Ta sẽ sử dụng Insertion Sort để sắp mảng với các dãy tăng dần có bước nhảy là h (với h có độ lớn nhỏ hơn hoặc bằng phân nửa độ dài mảng). Sau mỗi lần duyệt ta sẽ giảm bước nhảy xuống phân nửa và tiếp tục duyệt đến khi bước nhảy bằng 0 thì dừng. Các bước thực hiện như sau:
  + Bước 1: Gán h bằng độ dài mảng chia 2
  + Bước 2: Duyệt tuần tự qua các vị trí từ đầu mảng đến cuối mảng với bước nhảy là h sử dụng Insertion Sort để sắp xếp.
  + Bước 3: Giảm h xuống một nửa. Quay trở lại bước 2

## 6.2 Mã giả

```
Shell_Sort(array,n)
 h = length(array)/2
 while (h>0)
    for j: (h→n – 1)
            key = array[j]
            pos = j – h
            while (pos>= 0 & array[pos] > key)
                    array[pos+h] = array[pos]
                    pos-=h
            array[pos+h] = key
    h = h/2
```

Ví dụ: Cho mảng gồm 5 phần tử:

| 9 | 7 | 3 | 5 | 1 |
|---|---|---|---|---|

- Mảng có độ dài 5 nên ban đầu h = 2. Ta sắp xếp các dãy con bằng Insertion Sort với bước nhảy là 2.
- Các dãy cần được sắp lúc này là: 9, 3, 1 và 7, 5.
- Ta sử dụng Insertion Sort như trên và sau lần duyệt thứ nhất ta sẽ được:

| 1 | 5 | 3 | 7 | 9 |
|---|---|---|---|---|

- Ta tiếp tục giảm bước nhảy h đi một nửa. Giá trị h ở lần duyệt kế tiếp là 1. Ta thực hiện duyệt như thế đến khi h = 0 thì dừng.

## 6.3 Đánh giá thuật toán

- Thuật toán sắp xếp với bước nhảy giảm dần kết hợp Insertion Sort làm tăng tính hiệu quả nên được sắp vào nhóm thuật toán sắp xếp cấp cao. Thuật toán giảm được số lần duyệt mảng từ tuyến tính O(n) giảm xuống log(n).
- Thuật toán có độ phức tạp nlog(n) do đó thuật toán này có thể sắp xếp hiệu quả trên các bộ dữ liệu lớn với thời gian thực.
- Việc sử dụng Insertion Sort để sắp xếp các dãy sẽ làm tăng tính hiệu quả của thuật toán thay vì các cách sắp xếp khác như Selection Sort hay Interchange Sort.

## 7. Heap Sort (Sắp xếp vun đống)

### 7.1 Ý tưởng

- Thuật toán Heap Sort gồm 2 giai đoạn:
  + Giai đoạn 1: Hiệu chỉnh dãy ban đầu thành Heap
  + Giai đoạn 2: Sắp xếp dãy số dựa trên Heap
- Các bước được thực hiện như sau:
  +Bước 1: Hiệu chỉnh dãy ban đầu thành Heap
  + Bước 2: Gán giá trị n – 1 (n là độ dài mảng) vào biến right.
  + Bước 3: Hoán đổi vị trí đầu mảng và vị trí right.
  + Bước 4: Giảm giá trị r xuống một đơn vị.
  + Bước 5: Nếu r lớn hơn 0 thì tiếp tục hiệu chỉnh dãy từ đầu mảng đến vị trí right thành heap và quay lại bước 3. Nếu r bằng 0 thì dừng.

### 7.2 Mã giả

```
Shift(array,left,right)
 i = left, j = 2*l + 1, x = array[l]
 while (j<= r)
    if (j < r)
            if (array[j]<array[j+1])
                    j++
    if (array[j] <= x)
            break
    array[i] = array[j]
    i = j
    j = 2*i + 1
 array[i] = x
Make_Heap(array,n)
 left = size/2 – 1
 right = size – 1
 while (l>= 0)
    Shift(array, l, r)
    l = l – 1
Heap_Sort(array,n)
 right = n – 1
 left = 1
 Make_Heap(array,n)
 while (right>=left)
    swap(array[0], array[right])
    r = r – 1
    Shift(array, 0, right)
```

Ví dụ: Cho mảng gồm 6 phần tử:

| 9 | 7 | 2 | 3 | 5 | 1 |
|---|---|---|---|---|---|

- Ta có phần tử từ vị trí: size/2  đến cuối mảng (size là kích thước mảng) là một heap tự nhiên. Ta có 2 giai đoạn cần thực hiện.
- Ta bắt đầu xét từ vị trí i = 2. Lúc này i = 2 và j = 5. Do j bằng với right nên điều kiện j < r được bỏ qua. Ta gán x = array[2]. Tiếp tục xét thấy array[j] <= x nên ta thoát vòng lặp. Sau đó mảng vẫn như cũ.

| 9 | 7 | 2 | 3 | 5 | 1 |
|---|---|---|---|---|---|

- Ta tiếp tục thực hiện như thế đến khi kết thúc giai đoạn 1 kết quả sẽ là:

| 9 | 7 | 2 | 3 | 5 | 1 |
|---|---|---|---|---|---|

- Giai đoạn 2 ta sẽ tiến hành sắp xếp mảng. Ta thực hiện hoán đổi phần tử ở đầu mảng và phần tử ở vị trí right (right ban đầu bằng size – 1, với size là kích thước mảng). Sau đó giảm giá trị của right xuống một đơn vị và tiếp tục tạo heap trên vùng left đến right.
- Mảng sau lần duyệt đầu tiên của gian đoạn 2 là:

| 7 | 5 | 2 | 3 | 1 | 9 |
|---|---|---|---|---|---|

- Sau đó tiếp tục duyệt đến khi right bé hơn left thì dừng.

### 7.3 Đánh giá thuật toán

- Thuật toán có độ phức tạp là O(nlog(n)) nên có thể sắp xếp hiệu quả trên các bộ dữ liệu lớn trong thời gian ngắn.
- Thuật toán được sắp vào nhóm thuật toán sắp xếp cấp cao
- Thuật toán có chi phí duyệt mảng gần như cố định nên việc duyệt trên các mảng đã được sắp xếp hay mảng đảo ngược thì thời gian đều xấp xỉ nhau.
- Thuật toán sắp xếp dựa trên chỉnh mảng đó và không phát sinh them mảng phụ nên tối ưu về mặt bộ nhớ hơn.

## 8. Merge Sort (Sắp xếp trộn)

### 8.1 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. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Các bước được thực hiện như sau:
  + Bước 1: Tính giá trị ở giữa (mid = (left + right)/2)
  + Bước 2: Thực hiện Merge Sort ở nửa trái mảng
  + Bước 3: Thực hiện Merge Sort ở nửa phải mảng
  + Bước 4: Thực hiện Merge hai mảng đã được sắp ở trên

### 8.2 Mã giả

```
Merge(array,left,mid,right)
 array_left ← array[left:mid+1]
 array_right←array[mid+1:right]
 n1 = mid – left + 1, n2 = right – mid, i = 0, j = 0, k = 0
 while (i < n1 & j < n2)
    if (array_left[i] < array_right[j])
            array[left+k] = array_left[i]
            k = k+1
            i = i + 1
    else
            array[left+k] = array_right[j]
            k = k+1
            j = j + 1
 while (i < n1)
    array[k] = array_left[i]
    k = k + 1
    i = i + 1
 while (j < n2)
    array[k] = array_right[j]
    k = k + 1
    j = j + 1

Merge_Sort(array,left,right)
    if (left < right)
            mid = left + (right – left)/2
            Merge_Sort(array,left,mid)
            Merge_Sort(array, mid+1, right)
            Merge(array, left, mid, right)
```

Ví dụ: Cho mảng có 8 phần tử

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

- Ta sẽ phân nhỏ mảng lớn thành các mảng con. Theo như lời gọi đệ quy ta sẽ tách ra thành các mảng con: [9], [7], [3] , [4] , [2] , [6] , [5] , [1].
- Sau đó ta tiến hành Merge và được các mảng con: [7,9], [3,4], [2,6],[1,5].
- Sau đó ta tiếp tục Merge và ta được các mảng con: [3,4,7,9],[1,2,5,6].
- Cuối cùng ta Merge 2 mảng con lại và hoàn thành sắp xếp.

### 8.3 Đánh giá thuật toán

- Thuật toán có độ phức tạp là O(nlog(n))nên có thể sắp xếp trên các bộ dữ liệu có kích thước lớn với thời gian thực.
- Thuật toán có chi phí cố định là nlog(n) nên đối với các trường hợp mảng đã được sắp hay mảng đảo ngược thì hàm Merge Sort chạy với thời gian xấp xỉ nhau.
- Ngày nay đã có nhiều bản cải tiến của Merge Sort giúp đẩy nhanh tốc độ của thuật toán. Chẳng hạn khi độ dài mảng cần sắp xếp nhỏ hơn 5 thì sẽ sử dụng Insertion Sort để hỗ trợ.
- Thuật toán phát sinh mảng phụ trong quá trình sắp xếp nên đối với các bộ dữ liệu có kích thước lớn thì thuật toán này không tối ưu về mặt bộ nhớ so với Heap Sort. Thuật toán được sắp vào nhóm thuật toán sắp xếp cấp cao.

## 9. Quick Sort (Sắp xếp nhanh)

### 9.1 Ý tưởng

- Thuật toán Quick Sort sử dụng phương pháp chia để trị như Merge Sort. 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. Thông thường chúng ta sẽ chọn phần tử đầu tiên của mảng con đang sắp hoặc phần tử giữa mảng của mảng đang sắp. Các bước thực hiện như sau:

  + Bước 1: Gán biến i = left, j = right, key = array[(left+right)/2] (với left là vị trí bắt đầu của mảng, right là vị trí kết thúc mảng)

  + Bước 2: Thực hiện vòng lắp while trong khi biến i bé hơn hoặc bằng biến j

  + Bước 3: Trong vòng lặp while ở bước 2 ta thực hiện vòng lặp while khác. Trong vòng lặp đó ta tăng biến i lên một đơn vị đến khi phần tử tại đó lớn hơn hoặc bằng key.

  + Bước 4: Trong vòng lặp while ở bước 2 ta thực hiện vòng lặp while khác. Trong vòng lặp đó ta giảm biến j xuống một đơn vị đến khi phần tử tại đó bé hơn hoặc bằng key.

  + Bước 5: Nếu giá trị của i bé hơn hoặc bằng giá trị của j thì ta đổi chổ hai phần tử tại vị trí i và vị trí j. Tăng i lên một đơn vị và giảm j đi một đơn vị.

+ Bước 6: Khi đã thoát khỏi vòng lặp ở bước 2 ta kiểm tra nếu j lớn hơn left thì ta thực hiện Quick Sort trên đoạn mảng con từ vị trí từ left đến j.
+ Bước 7: Ta tiếp tục kiểm tra nếu i bé hơn right thì ta thực hiện Quick Sort trên đoạn mảng con từ vị trí i đến right.

## 9.2 Mã giả

```
Quick_Sort(array,left,right)
 i = left, j = right, key = array[(left+right)/2]
 do{
    while (array[i] < x) i = i + 1
    while (array[j] > x) j = j – 1
    if (i<=j)
            swap(array[i],array[j])
            i = i + 1
            j = j – 1
 }while(i <= j)
 if (j > l) Quick_Sort(array, l, j)
 if (i < r) Quick_Sort(array, i, r)
```

Ví dụ: Cho mảng 5 phần từ

| | | | | |
|---|---|---|---|---|
| 9 | 7 | 3 | 5 | 1 |

- Ở lần duyệt đầu tiên ta chọn phần tử ở giữa là phần tử có vị trí i = 2. Thực hiện vòng lặp ta thấy array[0] > array[2] và array[4] < array[2] nên ta hoán đổi. Tiếp tục xét thấy array[2]==array[2] và array[1] > array[2] nên ta tiếp tục hoán đổi. Kết thúc vòng lặp while ta sẽ được.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 7 | 5 | 9 |

- Sau đó ta xét đến 2 lệnh if bên dưới. Khi này i = 2 và j = 1. Ta tiếp tục thực hiện phân hoạch mảng và sắp xếp.

## 9.3 Đánh giá thuật toán

- Thuật toán Quick Sort sử dụng phương pháp chia để trị như thuật toán Merge Sort nhưng ở Quick Sort không phát sinh mảng phụ nên có thể xem là tối ưu về mặt bộ nhớ so với Merge Sort.
- Việc chọn phần tử có thể ảnh hưởng đến tốc độ sắp xếp của giải thuật nhưng chỉ khi nào ta cố tình sắp đặt mảng để xảy ra trường hợp xấu nhất thì giải thuật sẽ gặp vấn đề về thời gian. Trong các test case thông thường thì giải thuật Quick Sort được đánh giá là giải thuật hiệu quả nhất trong nhóm thuật toán sắp xếp cấp cao.
- Thuật toán phân hoạch thành các mảng con bằng cách chọn một phần tử làm mốc (pivot) nên chi phí để sắp xếp mảng xấp xỉ nlog(n) hoặc nhỏ hơn. Thuật

toán thực hiện tốt trên các bộ dữ liệu có kích thước lớn và tối ưu về mặt bộ nhớ cũng như thời gian sắp xếp.

- Thuật toán cài đặt đơn giản và ngắn hơn Merge Sort nên chúng ta có thể cài đặt giải thuật một cách nhanh chóng.

## 10. Flash Sort

### 10.1 Ý tưởng

- Thuật toán là sự kết hợp của Counting Sort và Insertion Sort. Thuật toán gồm 3 giai đoạn:

  + Giai đoạn 1: Tạo ra mảng số lớp gồm m phần tử (m xấp xỉ 0.43*n) và sau đó đếm số phần tử ở mỗi lớp theo công thức đã được chứng minh. Sau đó thực hiện cộng dồn số phần tử ở mỗi lớp đến các lớp liền sau đó và thực hiện đến cuối mảng phân lớp.

  + Giai đoạn 2: Thực hiện vòng lập duyệt toàn bộ mảng để đặt các phần tử về đúng phân lớp của nó theo công thức đã được sắp sẵn.

  + Giai đoạn 3: Dùng thuật toán Insertion Sort để sắp xếp các phần mảng con ở các phân lớp.

### 10.2 Mã giả

```
Flash_Sort(array,n)
    m = 0.43*n, k = 0
    L[1...m] = 0
    Min = min(array,n)
    Max = max(array,n)
    for i: (0→m – 1)
            k = (m – 1)*((array[i] – Min)*1.0/(Max – Min))
            L[k] = L[k] + 1
    for k: (1→m – 1)
            L[k] = L[k] + L[k – 1]
    count = 0, i = 0, j = 0, x = 0, y = 0, t = 0, k = m – 1
    while (count < n)
            while (i >= L[k]) // Bỏ qua các phần tử đã đặt đúng vị trí
                    i = i + 1
                    k = (m – 1)*((array[i] – Min)*1.0/(Max – Min))
            x = array[i]
            while (i < L[k])
                    k = (m – 1)*((array[i] – Min)*1.0/(Max – Min))
                    y = array[L[k] – 1]
                    array[L[k] – 1] = x
                    x = y
                    L[k] = L[k] – 1
                    count = count + 1
```

```
for k: (1→m – 1)
        for i: (L[k] – 2:L[k – 1] – 1)
                if (array[i] > array[i+1])
                        t = array[i]
                        j = i
                        while (t > array[j+1])
                                array[j] = array[j+1]
                                j = j + 1
                        array[j] = t
```

Ví dụ: Mảng gồm 6 phần tử:

| 9 | 7 | 2 | 3 | 5 | 1 |
|---|---|---|---|---|---|

- Ta có m = int(0.43*n) = int(0.43*6) = 2. Vậy mảng bây giờ sẽ gồm 2 phân lớp. Khi nhìn vào công thức đếm chúng ta có thể dễ dàng thấy các phần tử ở phần lớp cuối có giá trị bằng nhau và là các phần tử có giá trị lớn nhất. Vậy ở phân lớp thứ nhất có 5 phần tử và ở phân lớp thứ 2 có 1 phần tử.
- Sau khi thực hiện đặt các phần tử vào đúng phân lớp ta sẽ được:

| 5 | 3 | 2 | 7 | 1 | 9 |
|---|---|---|---|---|---|

- Sau đó tiến hành Insertion Sort trên các phần tử của phân lớp thứ nhất thì ta sẽ kết thúc quá trình sắp xếp.

### 10.3 Đánh giá thuật toán

- Theo tính toán giá trị m xấp xỉ 0.43*n (n là kích thước mảng) thì thuật toán sẽ tiếp cận với chi phí tuyến tính.
- Thuật toán đã chia nhỏ mảng ra làm các đoạn mảng con nên khi sử dụng thuật toán Insertion Sort sẽ trở nên hiệu quả hơn. Insertion Sort sẽ có hiệu quả cao khi sắp xếp trên các mảng kích thước nhỏ. Điều này làm cho thuật toán được sắp vào nhóm thuật toán sắp xếp cấp cao.
- Thuật toán phát sinh mảng phụ nhưng không quá lớn nên có thể được áp dụng trên các bộ dữ liệu có kích thước lớn.
- Thuật toán có sự kết hợp của Counting Sort nên khi ta test trên các bộ dữ liệu mà phần tử trong đó có giá trị xấp xỉ nhau thì thuật toán sẽ trở nên cực kì hiệu quả.

## 11. Counting Sort (Sắp xếp đếm)

### 11.1 Ý tưởng

- Counting sort là một thuật toán sắp xếp các con số nguyên không âm, không dựa vào so sánh. Trong khi các thuật toán sắp xếp tối ưu sử dụng so sánh có độ phức tạp O(nlogn) thì Counting sort chỉ cần O(n) nếu độ dài của danh sách không quá nhỏ so với phần tử có giá trị lớn nhất. Các bước thực hiện:
  + Bước 1: Tìm giá trị lớn nhất của mảng đó và sau đó khởi tạo mảng mới có kích thước bằng với giá trị lớn nhất vừa tìm được và tăng them 1 đơn vị.
  + Bước 2: Đếm số lần xuất hiện của mỗi giá trị phần tử trong mảng
  + Bước 3: Tiến hành cộng dồn để biết được đến phần tử đó đã có bao nhiêu số bé hơn hoặc bằng với chính nó.
  + Bước 4: Tiến hành tạo một mảng mới và đặt các giá trị của mảng ban đầu vào đúng vị trí của nó và gán giá trị mảng đã được sắp vào mảng ban đầu.

### 11.2 Mã giả

```
Counting_Sort(array,n)
    max = Max(array,n)
    b[0….max + 1] = 0
    for i: (0→n – 1)
            b[array[i]] = b[array[i]] + 1
    for i: (1→max)
            b[i] = b[i] + b[i – 1]
    res[0…n] = 0
    for i: (0→n – 1)
            res[b[array[i]] – 1] = array[i]
            b[array[i]] = b[array[i]] – 1\

    for i: (0→n – 1)
            array[i] = res[i]
```

Ví dụ: Cho mảng gồm 6 phần tử

| 9 | 7 | 2 | 3 | 5 | 1 |
|---|---|---|---|---|---|

- Ta thực hiện đếm số lần phần tử xuất hiện. Ta nhận được kết quả:

| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|

- Sau đó ta tiến hành cộng dồn đến cuối mảng thì ta được:

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

- Sau đó ta sẽ tiến hành sắp xếp mảng bằng cách đặt các phần tử vào đúng vị trí.

### 11.3 Đánh giá thuật toán

- Thuật toán counting sort là thuật toán không dựa trên so sánh nên thuật toán sẽ rất hiệu quả khi sắp xếp trên mảng số nguyên không âm.
- Thuật toán sẽ phát sinh thêm hai mảng phụ gồm một mảng đếm số lần xuất hiện của mỗi giá trị và một mảng dùng để đặt các phần tử vào đúng vị trí của mảng. Do đó thuật toán sẽ tốn thêm nhiều bộ nhớ để lưu trữ các giá trị và thuật toán sẽ trở nên không hiệu quả nếu như kích thước của mảng sai khác nhiều so với phần tử lớn nhất hoặc phần tử nhỏ nhất của mảng. Nếu kích thước của mảng quá nhỏ so với giá trị của phần tử lớn nhất thì thuật toán của chúng ta sẽ làm lãng phí bộ nhớ một cách vô ích.

## 12. Radix Sort (Sắp xếp theo cơ số)

### 12.1 Ý tưởng

- Giống như Counting Sort, Radix Sort là thuật toán sắp xếp không dựa trên so sánh. Radix Sort dựa trên sắp xếp cơ số và thuật toán chỉ cần tốn them một mảng 10 phần tử là có thể sắp xếp mảng. Các bước được thực hiện như sau:
  + Bước 1: Tìm giá trị lớn nhất (max) của mảng.
  + Bước 2: Tìm số chữ số của số max đó và gán vào biến m
  + Bước 3: Lặp m lần để sắp xếp mảng
  + Bước 4: Chúng ta có hai cách tiếp cận vấn đề hoặc là sắp xếp với chữ số từ trái sang phải hoặc là tử phải sang trái. Ở đây chúng ta chọn hướng tiếp cận từ phải sang trái (bắt đầu từ hang đơn vị).
  + Bước 5: Sắp xếp theo cơ số với chữ số thứ i (với i là số lần đã lặp ở bước 4) bằng cách đặt các phần tử vào đúng vị trí ở lần duyệt đó trên mảng mới và sau đó sao chép kết quả vào mảng ban đầu. Quay lại bước 4.

### 12.2 Mã giả

```
Len_Max(array,n)
 max = array[0]
 for i: (0→n-1)
    if (max < array[i])
            max = array[i]
 count = 0
 while (max > 0)
    count = count + 1
    max/=10
 return count
Digit(num,k)
    return (num/10^(k – 1))%10
```

```
Sort(array,n,k)
 f[0…9] = 0, j = 0
 for i: (0→n-1)
     f[Digit(array[i],k)]+=1
 for i: (1→9)
     f[i] = f[i] + f[i – 1]
 res[0..n - 1]
 for i: (0→n-1)
     j = Digit(array[i],k)
     res[f[j] – 1] = array[i]
     f[i] = f[i] – 1
 array[0..n – 1] = res[0..n – 1]
Radix_Sort(array,n)
     d = Len_Max(array,n)
     for i: (1→d)
             Sort(array,n,i)
```

Ví dụ: cho mảng có 6 phần tử

| 91 | 7 | 23 | 3 | 5 | 1 |
|---|---|---|---|---|---|

- Ta tiến hành sắp xếp các phần tử. Ở lần duyệt đầu tiên ta được:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 91<br>1 | | 23<br>3 | | 5 | | 7 | | |

- Ở lần duyệt kế tiếp ta sẽ có kết quả:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1<br>3<br>5<br>7 | | 23 | | | | | | | 91 |

- Khi đó, ta đã hoàn thành sắp xếp.

## 12.3 Đánh giá thuật toán

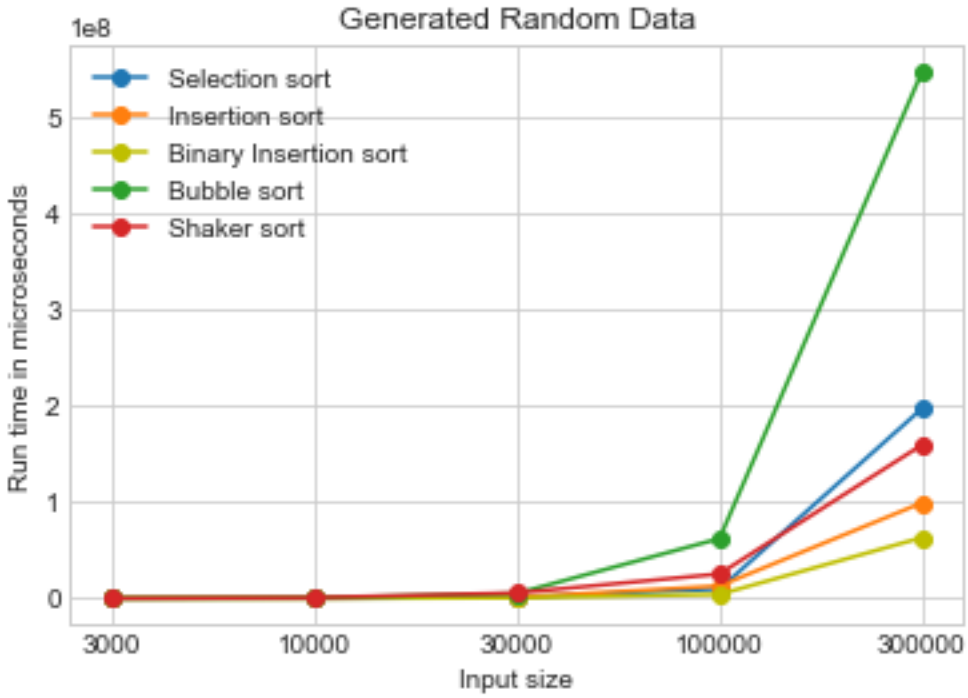
- Thuật toán có số lần duyệt mảng là n*max_len với n là kích thước mảng và max_len là số chữ số của phần tử lớn nhất.
- Mảng phụ chỉ tốn 10 phần tử do đó thuật toán sẽ tối ưu về mặt bộ nhớ và không cần phải lo việc hao phí bộ nhớ như Counting Sort.
- Chi phí sắp xếp của thuật toán có thể xem là chi phí tuyến tính và sẽ sử dụng hiệu quả trên các bộ dữ liệu lớn. Đối với các bộ dữ liệu có sự chênh lệch giữa giá trị phần tử và kích thước mảng thì Radix Sort là lựa chọn tốt hơn so với Counting Sort.

- Thuật toán không dựa trên so sánh nên chi phí sắp xếp có thể xem là như nhau ở tất cả các trường hợp. Do đó chi phí sắp xếp của thuật toán là chi phí tuyến tính và không đổi.

## II. Đánh giá và so sánh các thuật toán trên các bộ dữ liệu thực

### 1. Trên bộ dữ liệu được chọn ngẫu nhiên (Generated Random Data)

#### a. Nhóm thuật toán sắp xếp cơ bản

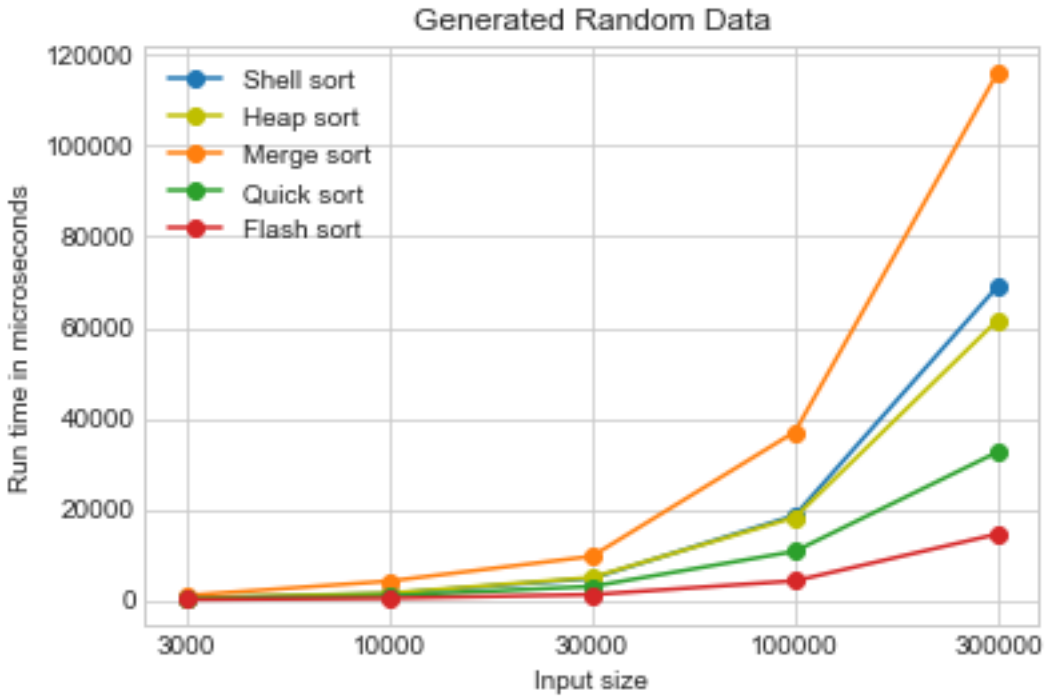



- Bubble Sort là thuật toán có thời gian thực thi lớn nhất trên bộ dữ liệu này. Thuật toán Bubble Sort sử dụng phương pháp duyệt từ cuối mảng lên đầu mảng (tại mỗi vị trí và biến chạy chính thì sử dụng toán tử trừ để thực hiện) do đó cần nhiều thời gian hơn so với các thuật toán duyệt từ các vị trí tại mỗi điểm đến cuối mảng (biến chạy chính sử dụng toán tử cộng để thực hiện).
- Thông qua đồ thị ta có thể thấy thuật toán Binary Insertion Sort là thuật toán chạy hiệu quả nhất. Số thao tác mà thuật toán Binary Insertion Sort thực hiện bằng với số thao tác mà thuật toán Insertion Sort thực hiện nhưng điểm tối ưu của Binary Insertion Sort là có kết hợp tìm kiếm nhị phân dẫn đến điều kiện so sánh trong vòng lặp while khi chèn chỉ còn một điều kiện. Đây được xem là thao tác cơ sở nên việc giảm đi một điều kiện cần xét sẽ làm tăng tốc độ của thuật toán lên rất nhiều.
- Các thuật toán còn lại chạy xấp xỉ nhau trong đó thuật toán Insertion Sort chạy nhanh nhất trong 3 thuật toán còn lại. Insertion Sort có chọn lọc để sắp xếp (chỉ khi nào phần tử đó bé hơn phần tử cuối của mảng đã được sắp phía bên

trái nó thì mới tiến hành chèn). Việc chọn lọc như thế sẽ tiết kiệm được rất nhiều chi phí thay vì duyệt với chi phí cố định là $n^2$ như Selection Sort.

- Shaker Sort là một thuật toán cải tiến của Bubble Sort do đó nó thuật toán này chạy nhanh hơn Bubble Sort do nó có thể thu hẹp vùng mảng cần sắp sau mỗi lần duyệt.

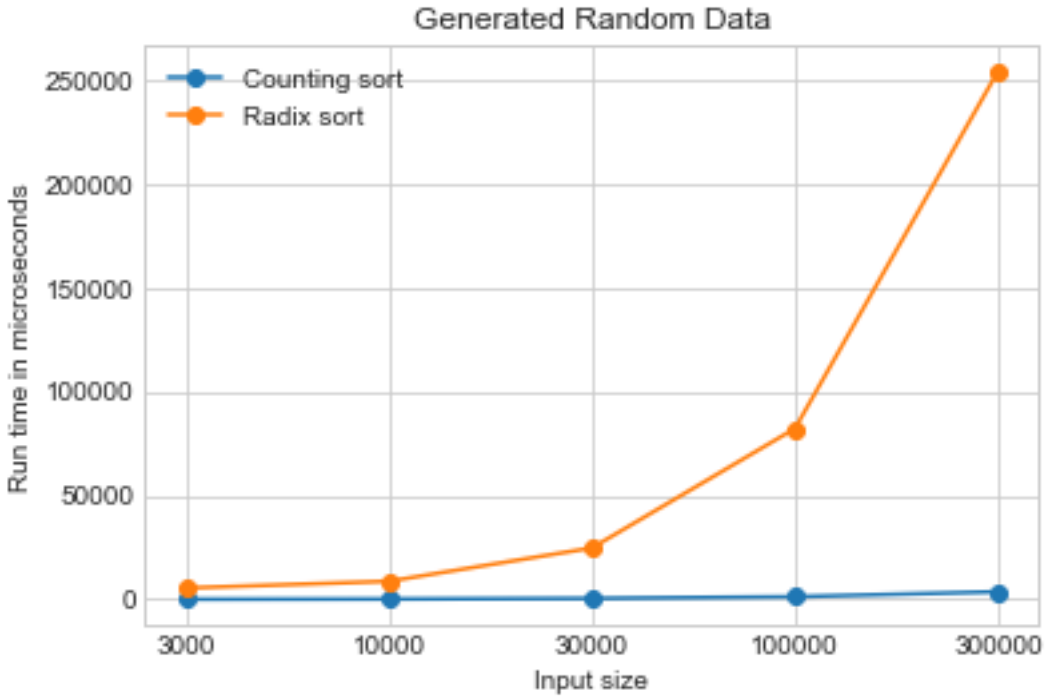
**b. Nhóm thuật toán sắp xếp cấp cao**




- Trong nhóm thuật toán này Merge Sort là thuật toán chạy tốn nhiều thời gian và bộ nhớ để tạo mảng phụ trong quá trình sắp xếp nhiều nhất. Đây là thuật toán có tính ổn định cao với chi phí cố định để sắp xếp mảng là nlog(n). Do đó thuật toán này hầu như chạy cùng một lượng thời gian với các bộ dữ liệu khác nhau.
- Flash Sort là thuật toán chạy nhanh nhất trong nhóm thuật toán này. Do sự kết hợp một phần của Counting Sort và thuật toán Insertion Sort nên Flash Sort sẽ là một lựa chọn thích hợp để sắp xếp trên các bộ số nguyên dương. Mặt khác Flash Sort phát sinh mảng phụ có kích thước không quá lớn nên sẽ là một lựa chọn tốt khi sắp xếp trên các bộ dữ liệu lớn (việc chọn kích thước của mảng phụ tùy thuộc vào người sử dụng thuật toán, kích thước dữ liệu đầu vào hoặc hiệu quả thuật toán mà họ mong muốn).
- Quick Sort là một thuật toán sử dụng phân hoạch mảng nên thuật toán này thông thường sẽ chạy nhanh hơn các thuật toán còn lại trong cùng nhóm thuật toán này.

- Heap Sort và Shell Sort có thời gian chạy xấp xỉ nhau. Hai thuật toán này đều không phát sinh mảng phụ trong quá trình sắp xếp nên sẽ tối ưu về mặt bộ nhớ.

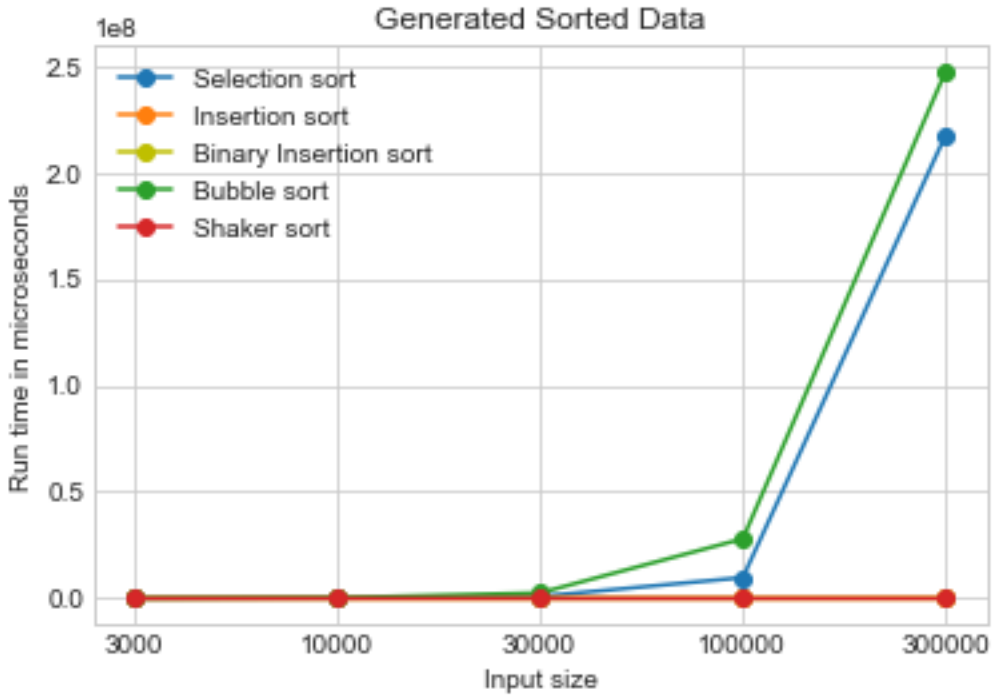
### c. Nhóm thuật toán sắp xếp đếm




- Nhóm thuật toán này chỉ có Radix Sort và Counting Sort. Trong đó Radix Sort dựa trên việc sắp xếp cơ số và mảng phụ phát sinh chỉ có kích thước 10 phần tử.
- Thuật toán Counting Sort phát sinh mảng phụ với kích thước bằng với giá trị lớn nhất của mảng nê sẽ tối ưu về mặt thời gian hơn so với Radix Sort nhưng lại tốn rất nhiều bộ nhớ và sẽ trở nên tiêu tốn bộ nhớ một cách vô ích nếu kích thước của mảng lệch xa so với giá trị nhỏ nhất hoặc lớn nhất của mảng.

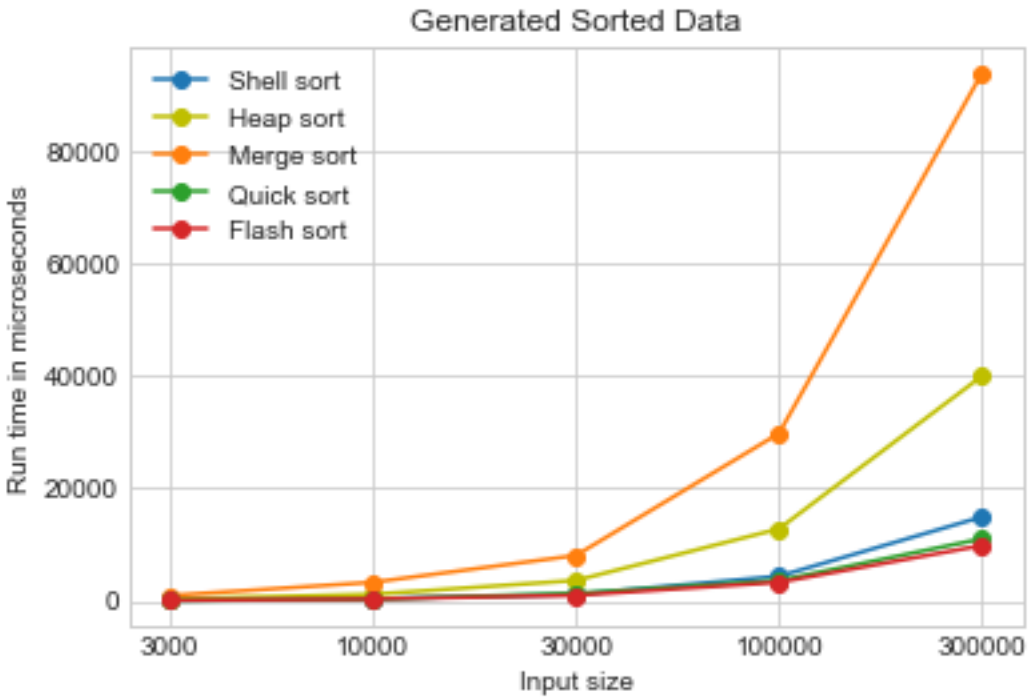
**2. Trên bộ dữ liệu được sắp xếp sẵn (Generated Sorted Data)**

**a. Nhóm thuật toán sắp xếp cơ bản**



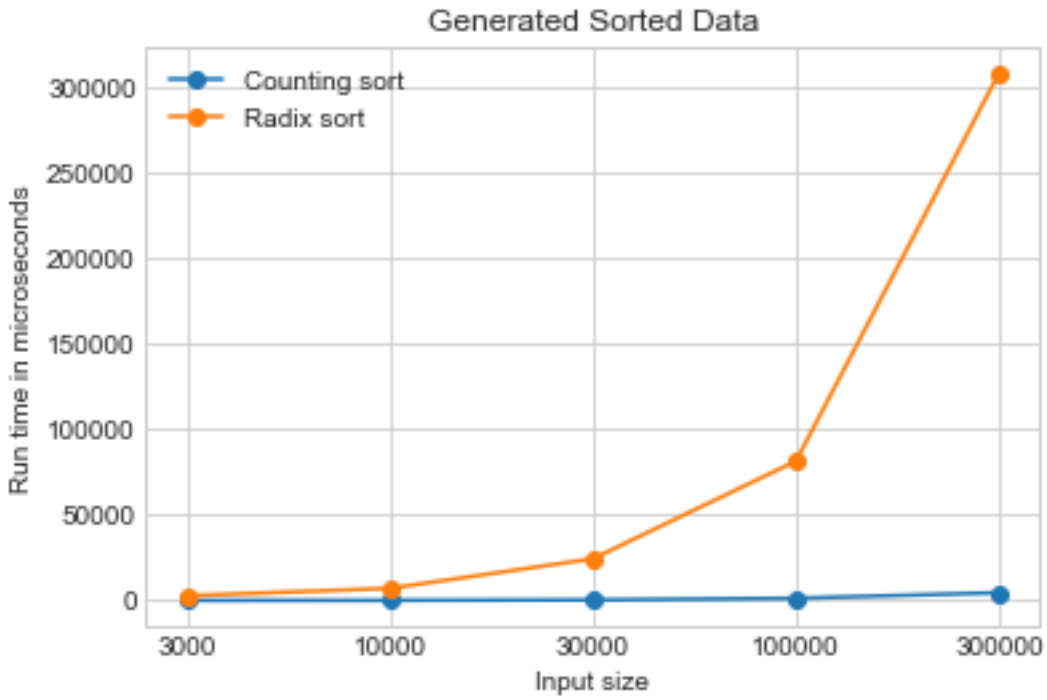

- Đối với bộ dữ liệu đã được sắp xếp sẵn thì ba thuật toán Binary Insertion Sort, Insertion Sort và Shaker Sort có thời gian thực thi gần như bằng nhau nên khi ta vẽ đồ thị thì ba đường trên gần như trùng nhau.
- Thuật toán Selection Sort và Bubble Sort sẽ phải tốn chi phí cố định là n(n+1)/2 để duyệt mảng do đó sẽ cần nhiều thời gian hơn dù bộ dữ liệu đã được sắp sẳn.

**b. Nhóm thuật toán sắp xếp nâng cao**




- Đối với bộ dữ liệu đã được sắp xếp thì Flash Sort và Quick Sort có thời gian duyệt gần như bằng nhau.
- Merge Sort có chi phí sắp xếp và chi phí mảng phụ cố định nên cần nhiều thời gian nhất để duyệt mảng.
- Heap Sort và Shell Sort có thời gian chạy gần như bằng nhau và đều hiệu quả do không phát sinh mảng phụ.

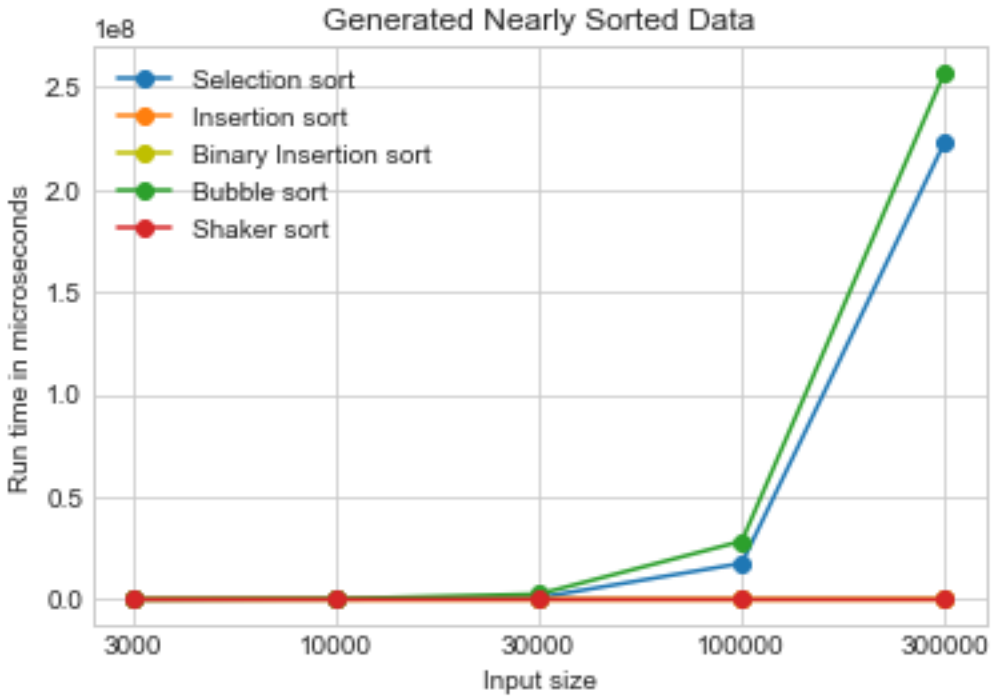
### c. Nhóm thuật toán sắp xếp đếm




- Trong các trường hợp thì Counting Sort luôn chạy nhanh hơn Radix Sort nhưng lại đòi hỏi bộ nhớ nhiều hơn so với Radix Sort.
- Đối với các bộ dữ liệu có kích thước nhỏ và kích thước mảng không sai lệch quá nhiều so với giá trị phần tử trong mảng thì Counting Sort sẽ là một lựa chọn tốt.
- Đối với các bộ dữ liệu có kích thước lớn thì chúng ta nên chọn Radix Sort để tối ưu về mặt bộ nhớ.

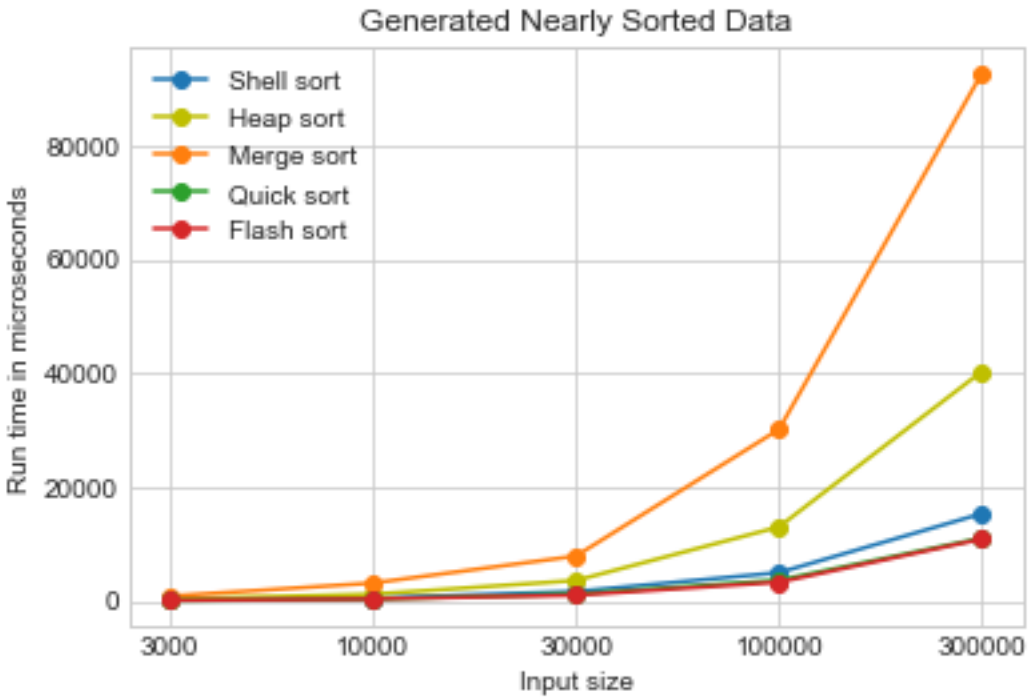
3. **Trên bộ dữ liệu gần như được sắp xếp (Generated Nearly Sorted Data)**

a. **Nhóm thuật toán sắp xếp cơ bản**



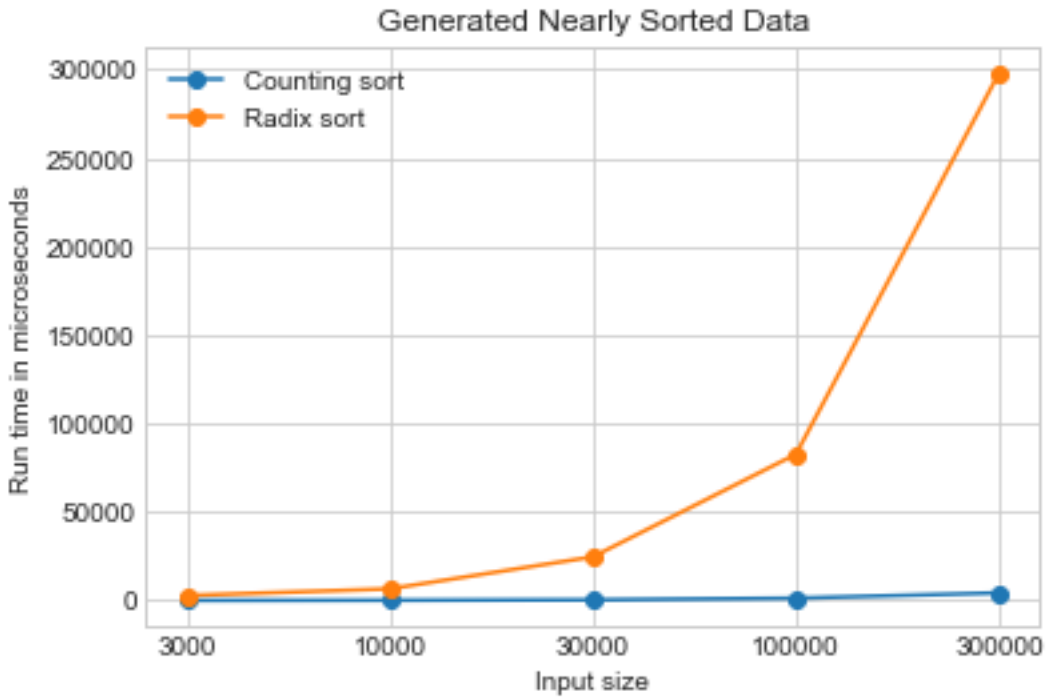

- Trên bộ dữ liệu gần như đã được sắp xếp thì thời gian duyệt mảng của ba thuật toán Insertion Sort, Binary Insertion Sort và Shaker Sort xử lý trong thời gian rất ngắn và gần như bằng nhau.
- Thuật toán Selection Sort và Bubble Sort với lượng chi phí duyệt mảng là cố định nên trên bộ dữ liệu gần như đã được sắp xếp số thao tác xử lý của cả hai thuật toán trên vẫn mang giá trị cố định.

**b. Nhóm thuật toán sắp xếp cấp cao**




- Thuật toán Merge Sort là thuật toán tốn nhiều thời gian nhất bởi lượng chi phí mảng phụ và số thao tác duyệt mảng là cố định.
- Quick Sort và Flash Sort là hai thuật toán chạy nhanh nhất và có thời gian chạy gần như bằng nhau. Trong trường hợp này Quick Sort sẽ được ưu tiên hơn do không phải phát sinh mảng phụ trong quá trình sắp xếp.
- Shell Sort là thuật toán chạy nhanh chỉ sau Quick Sort và Flash Sort.
- Heap Sort là thuật toán sắp xếp trực tiếp trên chính mảng đó và không phát sinh mảng phụ nên sẽ là một lựa chọn để tối ưu về mặt bộ nhớ.

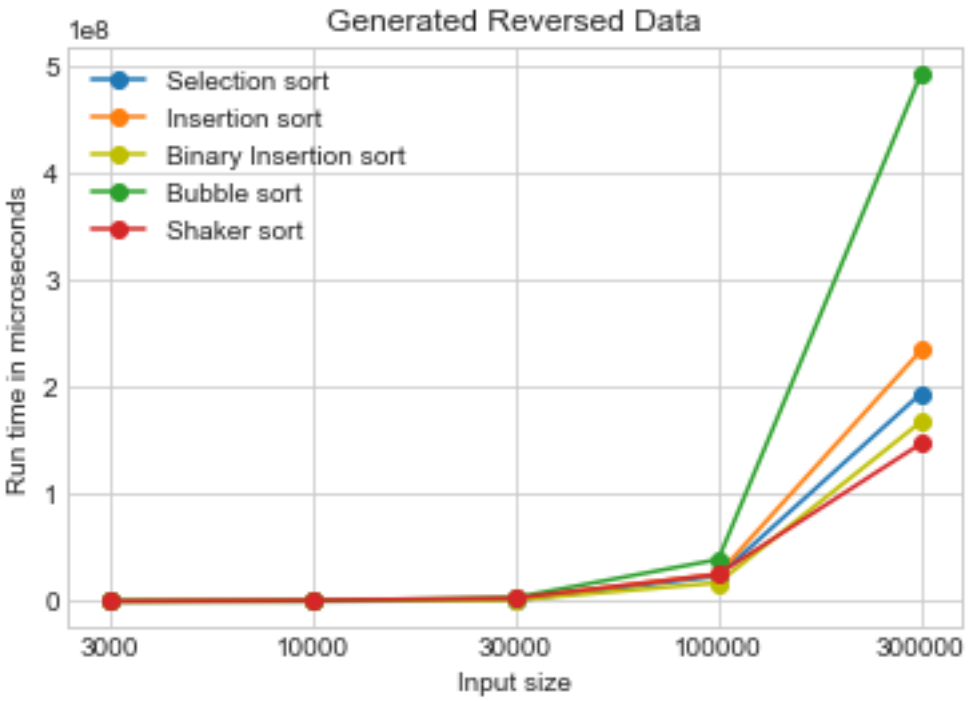
### c. Nhóm thuật toán sắp xếp đếm




- Thuật toán Counting Sort chạy nhanh hơn Radix Sort nhưng dung lượng bộ nhớ cần cung cấp sẽ nhiều hơn Radix Sort. Radix Sort duyệt qua một mảng với số thao tác là cố định nên sẽ tốn chi phí sắp xếp xấp xỉ nhau ở các bộ dữ liệu khác nhau có cùng kích thước.
- Thời gian thực thi của Counting Sort rất nhỏ. Chúng ta nhìn vào đồ thị có thể thấy thời gian thực thi của Counting Sort là gần như bằng 0.

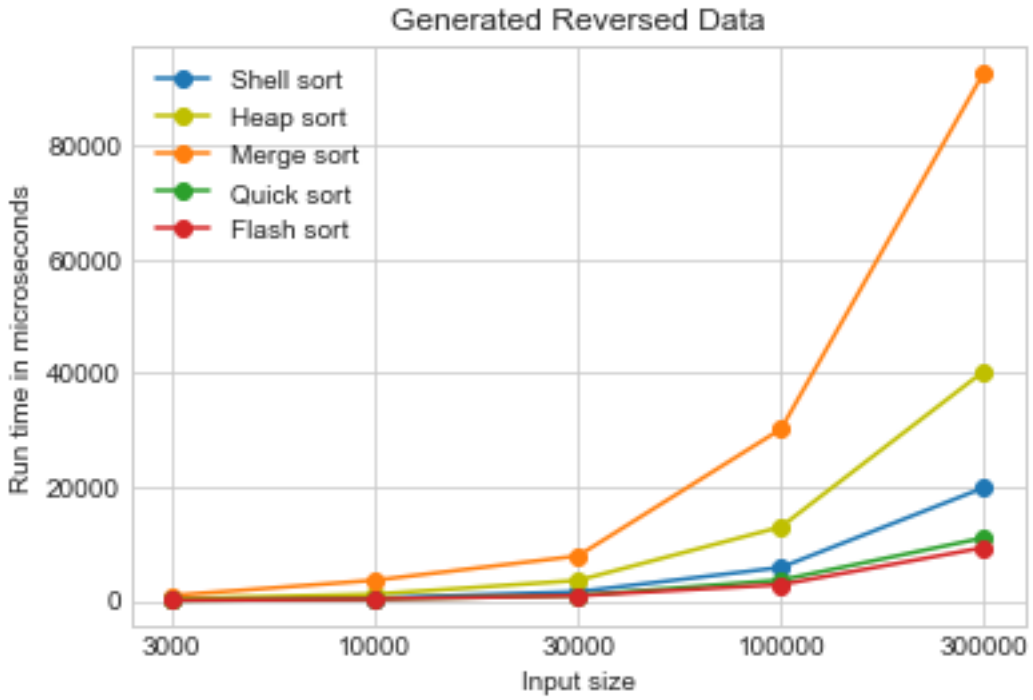
### 4. Trên bộ dữ liệu được đảo ngược giá trị (Generated Reversed Data)

### a. Nhóm thuật toán sắp xếp cơ bản



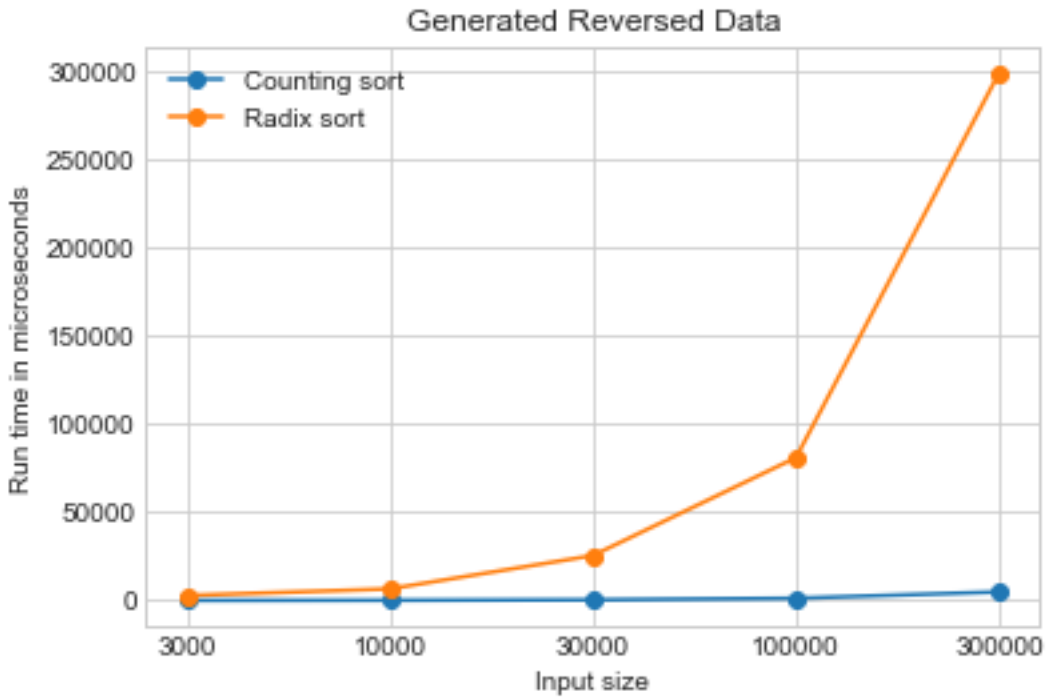

- Thuật toán Shaker Sort là thuật toán chạy hiệu quả nhất trong nhóm thuật toán này (do thuật toán này có thể thu hẹp vùng mảng cần sắp sau mỗi lần duyệt).
- Thuật toán Bubble Sort là thuật toán tốn chi phí thời gian để sắp xếp là lớn nhất do mảng mỗi lần được duyệt từ cuối mảng trở lên.
- Binary Insertion Sort sẽ tiết kiệm được nhiều thời gian hơn so với Insertion Sort thông qua phương pháp tìm kiếm nhị phân để giảm đi một điều kiện cần xét trong vòng lặp while khi chèn. Việc này giúp thuật toán xử lý nhanh hơn.
- Nhóm thuật toán này đều tốn lượng thời gian lớn để sắp xếp do đó không được áp dụng trên các bộ dữ liệu có kích thước lớn.

### b. Nhóm thuật toán sắp xếp cấp cao




- Nhóm thuật toán này thường được sử dụng để sắp xếp trên các bộ dữ liệu lớn/
- Merge Sort có chi phí sắp xếp không đổi với các kiểu có cùng kích thước nên thông thường sẽ là thuật toán chạy chậm hơn so với các thuật toán khác.
- Quick Sort thường sẽ là thuật toán được lựa chọn nhiều nhất bởi tính hiệu quả cao, tối ưu về mặt bộ nhớ và cách code đơn giản. Chúng ta có thể thấy Quick Sort có thời gian thực thi chỉ chậm hơn Flash Sort.
- Flash Sort sẽ phát sinh một mảng phụ trong quá trình sắp xếp. Trên các bộ dữ liệu có kích thước đủ lớn thì thuật toán này không phải là một lựa chọn tối ưu.
- Heap Sort và Shell Sort có thời gian thực thi xấp xỉ nhau và tính hiệu quả khá cao khi cả hai thuật toán này đều tối ưu về mặt bộ nhớ do không phát sinh mảng phụ trong quá trình sắp xếp.

### c. Nhóm thuật toán sắp xếp đếm




- Counting Sort luôn có thời gian thực thi nhỏ hơn Radix Sort nhưng bên cạnh đó chi phí mảng phụ phát sinh có thể là rất lớn và gây lãng phí bộ nhớ.
- Counting Sort thích hợp sử dụng trên các bộ dữ liệu có kích thước và giá trị phần tử không quá sai khác nhau.
- Radix Sort sẽ là một lựa chọn an toàn hơn khi chúng ta sắp xếp một bộ số nguyên dương ngẫu nhiên.