ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

----------o0o----------

LAB03-SORTING ALGORITHMS

GVHD: Lê Đình Ngọc

Nhóm thực hiện:

21120336_Nguyễn Phương Thảo

21120339_Nguyễn Đình Nam Thuận

21120347_Nguyễn Khắc Triệu

21120370_Phạm Nguyễn Quốc Vũ

HỌC PHẦN: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT
HỌC KỲ I – NĂM HỌC 2022-2023

MỤC LỤC

I. Information

Lab 3: Sorting Algorithm

- Chọn nhóm 11 thuật toán (Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort và Flash Sort
- Hoàn thành các phần nội dung được yêu cầu:
    1. Lập trình (Programming)
    2. Đệ trình (Submission)

II. Introduction

Nhóm thực hiện gồm các thành viên:

21120336_Nguyễn Phương Thảo

21120339_Nguyễn Đình Nam Thuận

21120347_Nguyễn Khắc Triệu

21120370_Phạm Nguyễn Quốc Vũ

III. Algorithm presentation

1. Selection Sort

- Selection Sort (sắp xếp chọn) là một thuật toán sắp xếp đơn giản dựa trên so sánh tại chỗ.

1.1. Ý tưởng

Danh sách được chia thành hai phần (Trái - Phải) trong cùng mảng gốc.

Phần được sắp xếp nằm ở đầu bên trái và phần chưa được sắp xếp chiếm phần còn lại bên phải của danh sách. (ban đầu thì phần bên phải là toàn bộ danh sách)

Mỗi lần lặp thuật toán sẽ liên tục tìm giá trị nhỏ nhất ở phần bên phải, hoán đổi vị trí của nó cho phần tử bên trái ngoài cùng nhất của phần Phải.

Quá trình này tiếp tục lặp lại và phần Trái được sắp xếp lấn dần về phần chưa được sắp xếp từng phần tử một cho tới khi hết mảng.

1.2. Mã giả

```
1. for i = 1 to A.length - 1
2.         min_index = i
//Tìm số nhỏ nhất trong dãy con A[i + 1 ... n].
3.       for j = i + 1 to A.length
4.             if A[j] < A[min_index]
5.                   min_index = j
6.       endfor
//Hoán đổi vị trí của giá trị nhỏ nhất và giá trị hiện tại của mảng chưa sắp xếp.
7.         key = A[i]
8.         A[i] = A[min_index]
9.        A[min_index] = key
10. End for
```

1.3. Mô tả cách thức hoạt động

Ta có một mảng ban đầu:

64	25	12	22	11

- Tìm phần tử nhỏ nhất trong arr [0 ... 4] và đặt nó ở đầu (Hoán đổi giá trị với phần tử đầu arr[0 ... 4]

- Tìm phần tử nhỏ nhất trong arr[1 ... 4] và đặt nó ở đầu arr[1 ... 4]

- Tìm phần tử nhỏ nhất trong arr[2 ... 4] và đặt nó ở đầu arr[2 ... 4]

- Tìm phần tử nhỏ nhất trong arr[3 ... 4] và đặt nó ở đầu arr[3 ... 4]

- Kết thúc

1.4 Đánh giá

- Thuật toán Selection Sort là một thuật toán khá đơn giản khi cài đặt, dễ học và cũng dễ vận dụng vào các dự án thực tế.

- Thuật toán này có độ phức tạp về thời gian là O(n^2) vì có 2 vòng lặp.

- Thuật toán này có độ phức tạp về không gian là O(1). Bộ nhớ chủ yếu được sử dụng cho các biến tạm thời khi hoán đổi hai giá trị trong Mảng. Sắp xếp chọn không bao giờ tạo ra nhiều hơn O (N) lượt hoán đổi và có thể hữu ích khi việc gán biến là một hoạt động tốn kém.

- Thuật toán này không phù hợp với các tập dữ liệu lớn vì độ phức tạp trung bình.

- Sắp xếp chọn không ổn định, tuy nhiên ta có thể cải thiện nó bằng việc áp dụng biến thể của nó là stable selection sort giúp vị trí tương đối của các phần tử được ổn định.

2. Insertion Sort

- Insertion Sort (sắp xếp chèn) là một trong những cách sắp xếp tự nhiên nhất. Nó đơn giản chỉ là thực hiện việc chèn một phần tử từ phần chưa được sắp xếp vào đoạn đã sắp xếp.

2.1. Ý tưởng

- Thuật toán bắt đầu từ phần tử thứ 2 (index = 1) của mảng cần sắp xếp.

- Coi phần tử đầu tiên (tức index = 0) là đoạn đã được sắp xếp. Mỗi phần tử bắt đầu từ vị trí key có index = 1 sẽ là phần chưa được xếp.
- Giải thuật sắp xếp chèn sẽ lần lượt thực hiện việc tìm kiếm liên tiếp qua mảng từ key = 1 cho đến hết mảng. Các phần tử không có thứ tự sẽ được di chuyển và được chèn vào vị trí thích hợp trong danh sách con trước phần tử key (của cùng mảng đó).

2.2. Mã giả

```
1. for i = 1 to A.length - 1
2.          key = A[i]
j = i - 1
//Chèn phần tử key vào mảng đã xếp
4.        while j >=0 and a[j] > key
5.          A[j+1] = A[j]
6.          j = j - 1
7.          endwhile
8.         A[j+1] = key
9. endfor
```

2.3. Mô tả cách thức hoạt động

- Chúng ta có mảng ban đầu như sau:

12	11	13	5	6

- i = 1 (Phần tử thứ 2 của mảng). Vì 11 nhỏ hơn 12 nên di chuyển 12 lên vị trí thứ i và chèn 11 vào trước 12 (vị trí thứ i – 1)

11	12	13	5	6

- Với i = 2. 13 sẽ vẫn ở vị trí của nó vì tất cả các phần tử trong A[0 ... i - 1] đều nhỏ hơn 13

11	12	13	5	6

- Với i = 3: 5 sẽ di chuyển về đầu và tất cả các phần tử khác từ 11 đến 13 sẽ di chuyển trước một vị trí so với vị trí hiện tại của chúng.

- Với i = 4: 6 sẽ chuyển đến vị trí sau 5, và các phần tử từ 11 đến 13 sẽ di chuyển trước một vị trí so với vị trí hiện tại của chúng.

==> Kết thúc

2.4 Đánh giá

- Về cơ bản, sắp xếp chèn có hiệu quả tốt đối với các giá trị dữ liệu nhỏ.

- Sắp xếp chèn có bản chất thích ứng, tức là nó thích hợp cho các tập dữ liệu đã được sắp xếp một phần. Sắp xếp chèn mất thời gian tối đa để sắp xếp nếu các phần tử được sắp xếp theo thứ tự ngược. Và cần thời gian tối thiểu đạt được (n) khi các phần tử đã được sắp xếp.

- Độ phức tạp về thời gian: O (N ^ 2)

- Độ phức tạp về không gian: O (1)

- Sắp xếp chèn là một thuật toán sắp xếp ổn định.

- Insertion có biến thể là Binary insertion sort sử dụng binary search như một công cụ để rút ngắn thời gian tìm kiếm vị trí chèn phần tử key vào đoạn đã sắp xếp.

3. Bubble Sort

- Bubble Sort (Sắp xếp nổi bọt) là một thuật toán sắp xếp đơn giản, với thao tác cơ bản là so sánh hai phần tử kề nhau, nếu chúng chưa đứng đúng thứ tự thì đổi chỗ (swap).

3.1. Ý tưởng

- Giả sử dãy cần sắp xếp có n phần tử.
- Khi tiến hành, ta so sánh hai phần tử đầu ở bên trái mảng. Nếu phần tử đứng trước lớn hơn phần tử đứng sau thì đổi chỗ chúng cho nhau.
- Tiếp tục làm như vậy với cặp phần tử thứ hai và thứ ba và tiếp tục cho đến cuối mảng. (Nghĩa là so sánh (và đổi chỗ nếu cần) phần tử thứ n - 1 với phần tử thứ n.) Sau bước này phần tử cuối cùng chính là phần tử lớn nhất của dãy.
- Sau đó, quay lại so sánh (và đổi chố nếu cần) hai phần tử đầu cho đến khi gặp phần tử thứ n - 2....
- Nếu trong một lần duyệt, không phải đổi chỗ bất cứ cặp phần tử nào thì danh sách đã được sắp xếp xong.

3.2. Mã giả

```
//n=listsize
 For number i from n downto 2
  for number j from 1 to (i - 1)
   if L[j] > L[j + 1] //nếu chúng không đúng thứ tự
    swap(L[j], L[j + 1]) //đổi chỗ chúng cho nhau
   endif
  endfor
 endfor
```

3.3. Mô tả cách thức hoạt động

Ta có mảng ban đầu:

Tại lần lặp thứ 1:

- Thuật toán so sánh hai phần tử đầu tiên và hoán đổi vị trí cho nhau vì 5 > 1

- Hoán đổi vị trí 5 và 4 cho nhau vì 5 > 4

- Hoán đổi vị trí 5 và 2 cho nhau vì 5 > 2

- So sánh 2 phần tử cuối thấy 8 > 5 nên không thay đổi vị trí

Tại lần lặp thứ 2:

- Lần lặp thứ 2, so sánh 2 phần tử đầu tiên thấy 1 < 4 nên không hoán đổi vị trí của chúng.

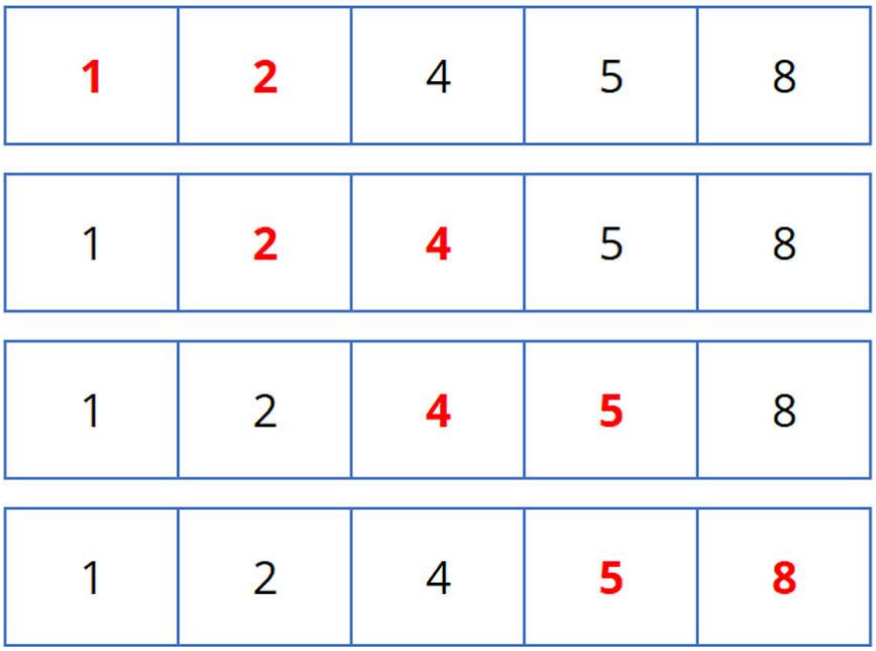
- Hoán đổi vị trí của 4 và 2 cho nhau vì 4 > 2.

- Tiếp theo, so sánh thấy 4 < 5 nên không thay đổi vị trí
- So sánh thấy 5 < 8 nên không thay đổi vị trí

Bây giờ, mảng đã được sắp xếp, nhưng thuật toán của chúng ta không biết liệu nó đã hoàn thành hay chưa.

Thuật toán cần một lần lặp qua toàn bộ mà không có bất kỳ sự hoán đổi nào để biết nó được là đã sắp xếp thành công.

Tại lần lặp thứ 3:



- Như vậy mảng đã được sắp xếp. ==> Kết thúc

3.4. Đánh giá

- Bubble sort là thuật toán sắp xếp đơn giản nhất hoạt động bằng cách hoán đổi nhiều lần các phần tử liền kề nếu chúng không đúng thứ tự.

- Thuật toán này không phù hợp với các tập dữ liệu lớn vì độ phức tạp thời gian trung bình và trường hợp xấu nhất của nó là khá cao.

- Độ phức tạp về thời gian: O (N^2)

- Độ phức tạp về không gian : O (1)

- Bubble sort là thuật toán ổn định.

- Có thể cải tiến Bubble sort dựa trên việc kiểm tra xem mảng đã được xếp sẵn chưa, hoặc sử dụng biến thể của nó là thuật toán Recursive Bubble sort.

4. Heap Sort

4.1. Ý tưởng

Tạo một heap từ các phần tử hiện chưa được sắp xếp của mảng.

Hoán vị phần tử gốc với phần tử cuối cùng của của heap (nút lá ngoài cùng bên phải).

Giảm kích thước heap lùi 1 đơn vị.

Đệ quy các bước trên ta sắp xếp được mảng ban đầu theo thứ tự tăng dần.

Lưu ý: Heap được đề cập trong này là max-heap

4.2. Mả giả

Mảng A, Length(A) = n:

Hàm heapSort(A, n)

Start

heapConstruct (A, n)

for i ← length[A] downto 2

swap A[1] <->A[i]

heap-size[A]← heap-size[A]−1

heapRebuild (A, 0, heap-size)

End

Hàm heapConstruct(A, n)

Start

for i ← (length[A]-1) / 2 downto 0

heapRebuild(A, i, n)

End

Hàm heapRebuild(A, pos, n)

Start

```
while 2 * pos + 1 < n
        j ← 2 * pos + 1;
        if (j < n - 1)
                if (a[j] < a[j + 1])
                        j ← j + 1;
                end if
        end if
        if (a[pos] >= a[j])
                end;
        swap: a[pos] and a[j];
        pos = j;
end while
```

End

4.3. Mô tả cách thức hoạt động

Bước 1: Xây dựng Heap. Xây dựng một heap từ dữ liệu đầu vào. Tạo một max-heap để sắp xếp theo thứ tự tăng dần.

Bước 2: Hoán đổi gốc. Trao đổi phần tử gốc với mục cuối cùng của heap.

Bước 3: Giảm kích thước heap. Giảm kích thước heap đi 1.

Bước 4: Kiểm tra lại. Xây dựng các phần tử còn lại thành một heap có kích thước heap mới bằng cách gọi heap-rebuild trên nút gốc.

Bước 5: Gọi đệ quy. Lặp lại các bước 2,3,4 miễn là kích thước heap lớn hơn 2.

4.4. Đánh giá

Best case = Average case = Worst case: O(nlogn)

Điểm mạnh:

- Nhanh và hiệu quả.

- Không yêu cầu thêm bộ nhớ.

- Dễ hiểu dễ sửa dụng.

Điểm yếu:

- Là một thuật toán không ổn định.

5. Merge Sort

5.1. Ý tưởng

Merge Sort là một thuật toán sắp xếp áp dụng kĩ thuật Chia để Trị (Divide and Conquer).

Bước chia (Divide) rất đơn giản: Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia cho đến khi không chia được nữa (còn lại 1 phần tử).

Bước kết hợp (Merge) là bước hoạt động hiệu quả nhất: Sắp xếp và hợp nhất các phần tử trên từng nửa mảng con cho đến khi toàn bộ mảng được hợp nhất.

Lưu ý: Mảng có 0 hoặc 1 phần tử là mảng được coi là đã sắp xếp.

5.2. Mả giả

Mảng A, length(A) = n:

Hàm mergeSort(A, First, Last)

Start

First ← 0, Last ← n - 1

Mid ← (First + Last) / 2

mergeSort(A, First, Mid)

mergeSort(A, Mid + 1, Last)

Lặp lại các bước trên nếu (First < Last)

Sau đó, gọi merge(A, First, Mid, Last)

End

Hàm merge(A, First, Mid, Last)

Start

n1 ← mid - first + 1;

n2 ← last - mid;

int* L = new int[n1];

Array L, Length(L) = n1

Array R, Length(R) = n2

// chia mang ben trai

for i ← 0 to n1 - 1

L[i] ← a[first + i];

End for

// chia mang ben phai

for i ← 0 to n2 - 1

R[j] ← a[mid + j + 1];

End for

i ← 0;

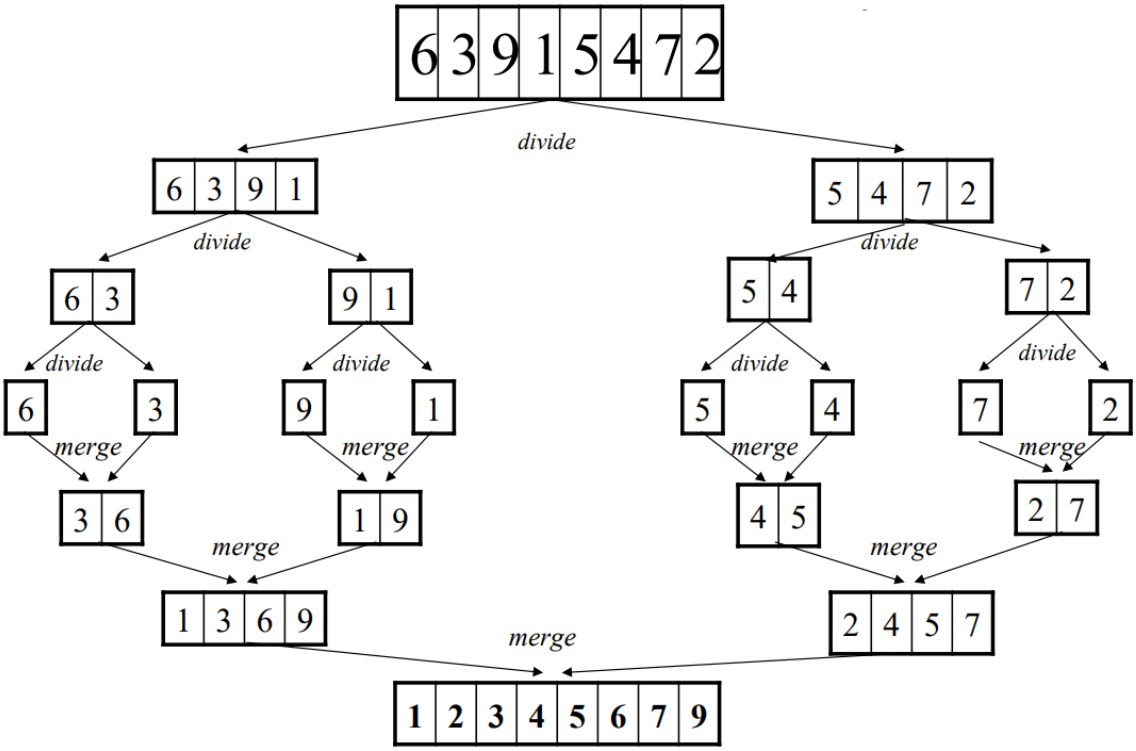
```
j ← 0;
k ← first;
// so sanh va sap xep theo dung thu tu
while (i < n1 and j < n2) {
a[k++] ← (++comp && L[i] < R[j]) ? L[i++] : R[j++];
End while
// tron 2 mang trai va phai vao mang ban dau
while i < n1
        a[k++] ← L[i++];
End while
while j < n2
        a[k++] = R[j++];
End while
delete[] L;
delete[] R;
```

End

5.3. Mô tả cách thức hoạt động

Ví dụ minh họa:

Dãy (mảng) ban đầu: 6, 3, 9, 1, 5, 4, 7, 2



Dãy (mảng) sau khi sắp xếp: 1, 2, 3, 4, 5, 6, 7, 9

5.4. Đánh giá

Merge Sort là thuật toán cực kỳ hiệu quả về mặt thời gian.

Cả trường hợp xấu nhất và trường hợp trung bình đều là O (n * log2n)

Merge Sort yêu cầu một mảng bổ sung có kích thước bằng với kích thước của mảng ban đầu.

Nếu chúng ta sử dụng danh sách liên kết, chúng ta không cần thêm một mảng

- Nhưng chúng ta cần không gian cho các liên kết.

- Và, sẽ rất khó để chia danh sách thành một nửa (O (n))

- Space Complexity: Độ phức tạp không gian của sắp xếp trộn là O (n).

6. Quick Sort

6.1. Ý tưởng

Thuật toán bao gồm ba bước sau:

Phân hoạch (partition): Phân hoạt danh sách.

Để phân hoạch danh sách, trước tiên chúng ta chọn một số phần tử từ danh sách (thường là phần tử đầu hoặc cuối) mà chúng ta hy vọng sẽ phân hoạch được khoảng một nửa các phần tử sẽ đến trước và một nửa sau. Gọi phần tử này là pivot.

Sau đó, chúng ta phân hoạch các phần tử để tất cả những phần tử có giá trị nhỏ hơn pivot nằm trong một danh sách phụ và tất cả những danh sách có giá trị lớn hơn nằm trong danh sách khác.

Đệ quy (recursion): Sắp xếp đệ quy các danh sách con riêng biệt.

Trị (conquer): Đặt các danh sách con đã được sắp xếp lại với nhau.

6.2. Mả giả

Hàm partition:

Start

```
pIndex ← firstt;
pivot  ← arr[last];

for i ← first to  last - 1

    if (arr[i] < pivot)
      swap arr[i] and arr[pIndex]
      increment pIndex by 1
        End if
 End for
```

swap (arr[last], arr[pIndex]).

return pIndex.

End

Hàm quick sort:

Start

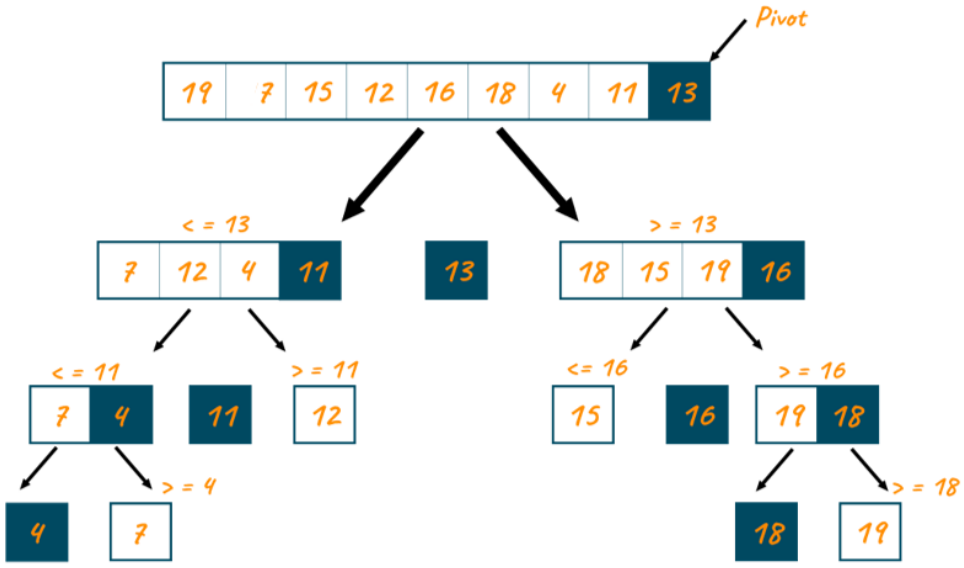
```
if (low < high) {
    pi ← partition(arr, low, high);
    quickSort(arr, low, pi – 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
 end if
```

End

6.3. Mô tả cách thức hoạt động

Ví dụ minh họa:

Dãy ban đầu: 19, 7, 15, 12, 16, 18, 4, 11, 1



Dãy sau khi sắp xếp: 4, 7, 11, 12, 13, 15, 16, 18, 19

6.4. Đánh giá

Worst case: O(n^2)

Best case và Average case: O(nlog2n)

Điểm mạnh:

- Độ phức tạp thời gian trong trường hợp trung bình để sắp xếp một mảng n phần tử là O (n lg n).

- Nhìn chung, nó chạy rất nhanh. Nó thậm chí còn nhanh hơn sắp xếp trộn (merge sort).

- Không yêu cầu thêm bộ nhớ

Điểm yếu:

- Thời gian chạy sẽ khác nhau với các kiểu đầu vào khác nhau

- Trường hợp xấu nhất diễn ra khi mảng đã sắp xếp (Không tận dụng được dữ liệu đầu vào).

- Là một thuận toán sắp xếp không ổn định.

7. Shell Sort

7.1. Ý tưởng

Cho 1 dãy n phần tử. Chọn 1 số nguyên dương h.

Chúng ta sẽ phân hoạch dãy này thành m dãy con thỏa tính chất: 2 phần tử bất kỳ trong 1 dãy con có khoảng cách trên dãy gốc là 1 số nguyên lần h. Sau đó sắp xếp các dãy con này bằng thuật toán sắp xếp Insertion Sort (đã được đề cập ở trên).

Giảm h rồi thực hiện lại bước trên cho tới khi h=1.

Trong phần tiếp theo sẽ được triển khai với việc sắp xếp dãy tăng dần.

7.2. Mã giả

Chọn dãy interval là: n/2, n/4, …, 1.

```
ShellSort (1 dãy n phần tử (A))
start
    assign interval=n/2
    while interval>0 do
    //chia thành interval sublist mỗi sublist gồm các phần tử thứ i, i+interval, i+2.interval, ... (i=1, interval)
    //các phần tử từ 0 tới interval-1 là phần tử đầu tiên của mỗi sublist
    //Sắp xếp các sublist bằng InsertionSort
        for i from interval to n-1 do
            assgin pos=i, temp=A[i]
            while j>=interval and A[j]>temp do
                A[j]=A[j-interval]
                j=j-interval
            end while
            //Đặt temp về đúng vị trí trong sublist
            A[j]=temp
        end do
    //Giảm interval -> giảm số lượng sublist
    interval=interval/2
    end while
end
```

Ghi chú: Với mỗi dãy số khác nhau có thể sẽ tồn tại những dãy interval đem lại hiệu quả tốt hơn những dãy khác tuy nhiên rất khó để có thể xác định đâu là dãy interval tốt nhất nên người ta thường chọn mặc định 1 dãy, như trong phần pseudocode dãy được chọn là n/2, n/4, …, 1.

7.3. Mô tả cách thức hoạt động

Sắp xếp dãy 8 phần tử 52, 3, 7, 18, 11, 34, 4, 6 theo thứ tự tăng dần.

Các ô đồng màu là thuộc cùng 1 dãy con.

Interval =8/2 =4

52	3	7	18	11	34	4	6

Sau khi sắp xếp dãy con:

11	3	4	6	52	34	7	18

Interval=4/2=2

11	3	4	6	52	34	7	18

Sắp xếp dãy con

4	3	7	6	11	18	52	34

Interval=2/2=1

11	3	4	6	52	34	7	18

Sau khi sắp xếp:

3	4	6	7	11	18	34	52

7.4. Đánh giá

- Độ phức tạp:
  - Best case: O(n.log(n))
  - Worst case $O(n^2)$
  - Average: O(n.log(n))
- Yêu cầu thêm bộ nhớ: không.
- Điểm mạnh: Shell Sort là phiên bản cải tiến của Insertion Sort (Shell Sort vẫn sắp xếp dựa trên phương pháp của Insertion Sort tuy nhiên vì các quá trình phân hoạch, các phần tử có khả năng về gần với vị trí của mình hơn nên tốn ít chi phí để chèn vào hơn), thời gian sắp xếp nhanh khi so sánh với các thuật toán khác).

- Điểm yếu: việc chọn interval sẽ ảnh hưởng đến thời gian thực thi, tuy nhiên đây là vấn đề quá phức tạp để kiểm soát, không tận dụng được ưu thế mảng sắp xếp sẵn.

8. Shaker Sort

8.1. Ý tưởng

Cho 1 dãy các phần tử. Xét lần lượt 2 phần tử liền kề nhau bắt đầu từ đầu hoặc cuối dãy, nếu 2 phần tử có thứ tự mâu thuẫn với thứ tự mong muốn thực hiện thao tác đổi chỗ 2 phần tử. Cứ lặp lại cho tới cặp phần tử cuối cùng thì đã đưa được phần tử cực trị về vị trí mong muốn. Thực hiện lại chuỗi thao tác trên nhưng bắt đầu theo chiều ngược lại (không xét phần tử đã đem về đúng vị trí) để đưa cực trị đối lập về đúng vị trí.

Trong phần tiếp theo chỉ đề cập tới việc sắp xếp dãy số nguyên tăng dần.

8.2. Mã giả

```
ShakerSort (dãy n phần tử(A))
start
    assign left=0, right=n-1, right sorted_pos=0, left_sorted_pos=0
    while  (left<right)
        //Đưa phần tử lớn nhất về cuối
        for i from left to right-1 do
            if A[i]>A[i+1] then
                swap(A[i],A[i+1])
                right_sorted_pos=i;
            end if
        end for
        right=right_sorted_pos
        //Đưa phần tử nhỏ nhất về đầu dãy
        for i from right to left+1 do
            if A[i]<A[i-1] then
```

```
                    swap(A[i],A[i-1])
                    left_sorted_pos=i
                end if
            end for
        left=left_sorted_pos
        end while
end
```

Ghi chú: (right/left)_sorted_pos: vị trí mà các phần tử bên phải/trái của nó đã về đúng vị trí.

8.3. Mô tả cách thức hoạt động

Cho dãy 1, 2, 1, 35, 3 (5 phần tử)

	Không xảy ra mâu thuẫn
	Mâu thuẫn và đã đổi chỗ
	Vị trí không đang xét tới.
	Đã về đúng vị trí

Left=0, right=4

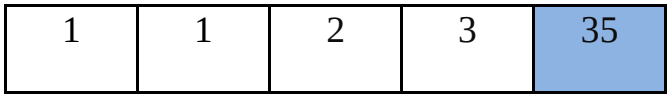
1	2	1	35	3

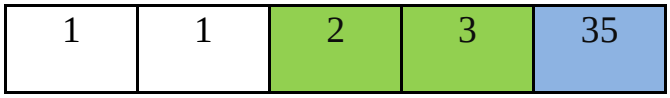
1	1	2	35	3

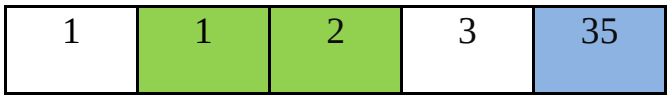
1	1	2	35	3

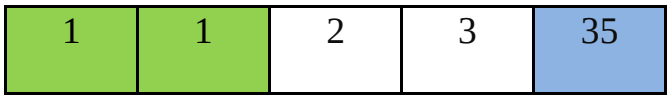
1	1	2	3	35

Left=0, right=3

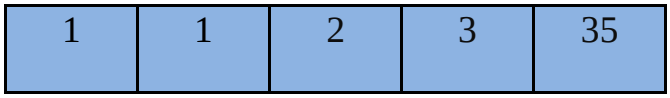








Left=3, right=3



8.4. Đánh giá

1. Độ phức tạp:Tốt nhất: O(n), Xấu nhất: O(n^2), Trung bình: O(n^2)
2. Yêu cầu thêm về bộ nhớ: không có.
3. Điểm mạnh: tận dụng được ưu thế của dãy – dãy con đã được sắp xếp sẵn, cũng vì điểm này mà Shaker Sort được xem là 1 phiên bản tốt hơn của Bubble Sort (vì tận dụng được ưu thế của các dãy con và dãy các phần tử đã về đúng vị trí nên giảm được 1 phần phép so sánh).
4. Điểm yếu: ưu thế về tốc độ trung bình vẫn không so sánh được với các thuật toán khác, tuy giảm được số phép so sánh so với Bubble Sort nhưng số lần hoán vị các phần tử gần như không đổi.

9. Radix Sort

9.1. Ý tưởng

Đây là 1 thuật toán sắp xếp dựa trên nguyên tắc phân thư của bưu điện (phân theo mã vùng bưu điện-Zip code).

Cho 1 dãy các phần tử. Nhóm các phần tử có cùng ký số/ký tự tại 1 vị trí nào đó (có thể cùng vị trí nếu tính từ bên trái hoặc phải) xét trên 1 hệ cơ số xác

định. Thực hiện việc nhóm này từ vị trí trái cùng hoặc phải cùng cho đến khi hết các ký số/ký tự của các phần tử trong dãy và dãy cũng đã được sắp xếp.

Có 2 phiên bản đặc trưng của Radix Sort là:

- Least Significant Digit Radix Sort: sắp xếp theo các ký số/ký tự từ phải qua trái.
- Most Significant Digit Radix Sort: sắp xếp theo các ký số/ký tự từ trái qua phải.

Phần tiếp theo sẽ triển khai Radix sort với hệ cơ số 10 và sắp xếp tăng dần.

9.2. Mã giả

```
RadixSort (Dãy A gồm n phần tử)
start
    //tìm phần tử lớn nhất dãy
    assign max=A[0], t=0, digit=1
    while t<n do
        if max<A[t]
        max=A[t]
        end if
        t++
    end while
    //bắt đầu nhóm các phần tử theo chữ số từ phải qua trái
    while max>0 do
        //đếm số phần tử của mỗi nhóm
```

```
        //tạo 1 mảng phụ để lưu số lượng phần tử của mỗi nhóm
        count_member [9] = {0}
        for j from 0 to n-1 do
            count_member [digit at pos i of A[j]] ++
        end for
        //biến mảng count_member thành mảng thể hiện khoảng vị trí của
  các nhóm
        count_member [0]-=1
        for k from 1 to 9 do
            count_member[k]+=count_member[k-1]
        end for
        //như vậy khoảng vị trí của nhóm k là 0 || count_member[k-1] +1
  tới count_member[k]
        //tạo 1 mảng phụ để sắp xếp các nhóm vào
        temp_arr[n-1]
        //lần lượt đưa các phần tử từ cuối dãy A vào temp_arr
        for m from n-1 to 0 do
            temp_arr [count_member [digit at pos i of A[m]] =A[m]
            count_member [digit at pos i of A[m]] --
        end for
  //chép lại sub_arr vào mảng chính
        for ii from 0 to n-1 do
```

```
                A[i]=temp_arr[i]
            end for
        digit*=10
        end while
end
```

Ghi chú: có thể triển khai việc nhóm các phần tử bằng cách sử dụng danh sách liên kết.

9.3. Mô tả cách thức hoạt động

Sắp xếp dãy 10 số nguyên dương 45, 67,135, 100, 38, 42, 9, 19, 35, 76.

Số lớn nhất dãy là 135 nên d=3 (số chữ số của dãy là 3)

d=1:

45	67	135	100	38	42	9	19	35	76

Sau khi nhóm:

100	42	45	135	35	76	67	38	9	19

d=2

100	42	45	135	35	76	67	38	09	19

Sau khi nhóm

100	09	19	135	35	38	42	45	67	76

d=3

100	009	019	135	035	038	042	045	067	076

Sau khi nhóm:

009	019	035	038	042	045	067	076	100	135

Dãy sau khi xếp xong:

9	19	35	38	42	45	67	76	100	135

9.4. Đánh giá

- Độ phức tạp: O(n) cho mọi trường hợp
- Yêu cầu thêm bộ nhớ: O(n+d)~O(n) với d là số ký số tối đa của dãy, và n là kích thước mảng phụ ghi kết quả.
- Điểm mạnh: tốc độ sắp xếp nhanh, thuật toán gần với cách sắp xếp tự nhiên của con người.
- Điểm yếu: không tận dụng được ưu thế dãy sắp xếp sẵn, việc thay đổi hệ cơ số có thể buộc người lập trình phải triển khai lại thuật toán.

10. Counting Sort

10.1. Ý tưởng

Đúng với tên gọi của mình Counting sort sẽ đếm số phần tử của mỗi giá trị trong mảng để thực hiện việc sắp xếp.

Cho dãy n phần tử có khoảng giá trị [a,b], chúng ta sẽ tạo mảng phụ để đếm số lượng mỗi phần tử của dãy cho sẵn. Tong đó vị trí thứ i của mảng phụ đại diện cho số lượng phần tử có giá trị i trong dãy n.

10.2. Mã giả

Counting sort(dãy A gồm n phần tử)

Start

initialize count_arr with size as told and assign to 0

for i from 0 to n-1 do

count_arr[A[i]]++

```
    end do
    //chép lại vào mảng chính
    for  i from start to end of count_arr
        write count_arr[i] times i into A
    end do
End
```

10.3. Mô tả cách thức hoạt động

Cho dãy 1,3,2,6,3,7,9,0.

B1: Tạo dãy count[9]={0}

B2: Đếm.

index	0	1	2	3	4	5	6	7	8	9
value	1	1	1	2	0	0	1	1	0	1

B3: ghi lại:

Dãy sau khi sắp xếp: 0, 1, 2, 3, 3, 6, 7, 9

10.4. Đánh giá

- Độ phức tạp O(n)
- Bộ nhớ: tuỳ thuộc vào khỏng giá trị của dãy.
- Thuật toán này không thể áp dụng cho số thực vì không thể tạo dãy có index thoả các giá trị của số thực được.

11. Flash Sort

11.1. Ý tưởng

Flash sort là một thuật toán sắp xếp tại chỗ (in-situ, không dùng mảng phụ) có độ phức tạp O(n), không đệ qui, gồm có 3 bước: (1) **Phân lớp dữ liệu**, tức là 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. (2) **Hoán vị toàn cục**, tức là dời

chuyển các phần tử trong mảng về lớp của mình. (3) **Sắp xếp cục bộ**, tức là để sắp xếp lại các phần tử trong phạm vi của từng lớp.

Quá trình này cũng tương tự như việc xếp lớp và xếp chỗ cho học sinh lớp 1 trong một trường học, theo tên học sinh. (Giả sử tên học sinh được phân bố đều; thực tế điều này không đúng, chẳng hạn, số tên vần H, T thường nhiều hơn vần A, B, C.)

Lúc đầu các học sinh đang ngồi trong các phòng học của trường, nhưng không theo thứ tự nào cả.

(1) Phân lớp: 26 phòng học mỗi phòng sẽ là 1 lớp: A, B, C,...

(2) Hoán vị toàn cục: dời từng học sinh về đúng lớp của mình; khi một học sinh vào lớp, một học sinh (ngồi chưa đúng lớp) sẽ phải nhường chỗ.

(3) Sắp xếp cục bộ: cô giáo chủ nhiệm từng lớp sẽ sắp xếp chỗ ngồi cho các em trong lớp của mình.

11.2. Mã giả

```
void flashSort(int a[], int n)

{

// Tìm giá trị nhỏ nhất của các phần tử trong mảng(minVal) và vị trí phần tử  lớn nhất của các phần tử trong mảng(max).

    int minVal = a[0];

    int max = 0;

// Khởi tạo 1 vector L có m phần tử (ứng với m lớp, trong source code lần này chọn số lớp bằng 0.45n)

    int m = int(0.45 * n);

    int* l = new int[m];

    for (int i = 0; i < m; i++)

        l[i] = 0;

    for (int i = 1; i < n; i++)
```

```
{
        if (a[i] < minVal)
                minVal = a[i];
        if (a[i] > a[max])
                max = i;
}
if (a[max] == minVal)
        return;
```

//Đếm số lượng phần tử các lớp theo quy luật, phần tử a[i] sẽ thuộc lớp k = int((m - 1) * (a[i]-minVal) / (a[max] - minVal))

```
double c1 = (double)(m - 1) / (a[max] - minVal);
for (int i = 0; i < n; i++)
{
        int k = int(c1 * (a[i] - minVal));
        ++l[k];
}
```

//Tính vị trí kết thúc của phân lớp thứ j theo công thức L[j] = L[j] + L[j - 1] (j tăng từ 1 đến m - 1)

```
for (int i = 1; i < m; i++)
        l[i] += l[i - 1];
```

//Trước hết chúng ta đổi chỗ a[max] và a[0] bởi chúng ta khởi tạo giá trị k = m - 1 tương ứng với phân lớp mà a[max] thuộc về, do bắt đầu tại phần tử này nên biến j ban

đầu cũng được khởi tạo bằng 0, Với tối đa n - 1 lần swap, n phần tử trong mảng sẽ về đúng phân lớp của mình

```
HoanVi(a[max], a[0]);
int nmove = 0;
int j = 0;
int k = m - 1;
int t = 0;
int flash;
```

//Khi j > L[k] - 1 nghĩa là khi đó phần tử a[j] đã nằm đúng vị trí phân lớp của nó do đó ta bỏ qua và tiếp tục tăng j lên để xét các phần tử tiếp theo.

```
while (nmove < n - 1)
{
    while (j > l[k] - 1)
    {
        j++;
        k = int(c1*(a[j] - minVal));
    }
    flash = a[j];
    if (k < 0) break;
```

//Như đã nói ở trên cứ mỗi khi đưa 1 phần tử về đúng phân lớp của nó ta lại giảm vị trí cuối cùng của phân lớp đó xuống (đồng thời tăng biến đếm số lần swap lên 1 đơn vị), quá trình này được thực hiện cho tới khi L[k] = j, điều này cũng tương đương với việc phân lớp k đã đầy

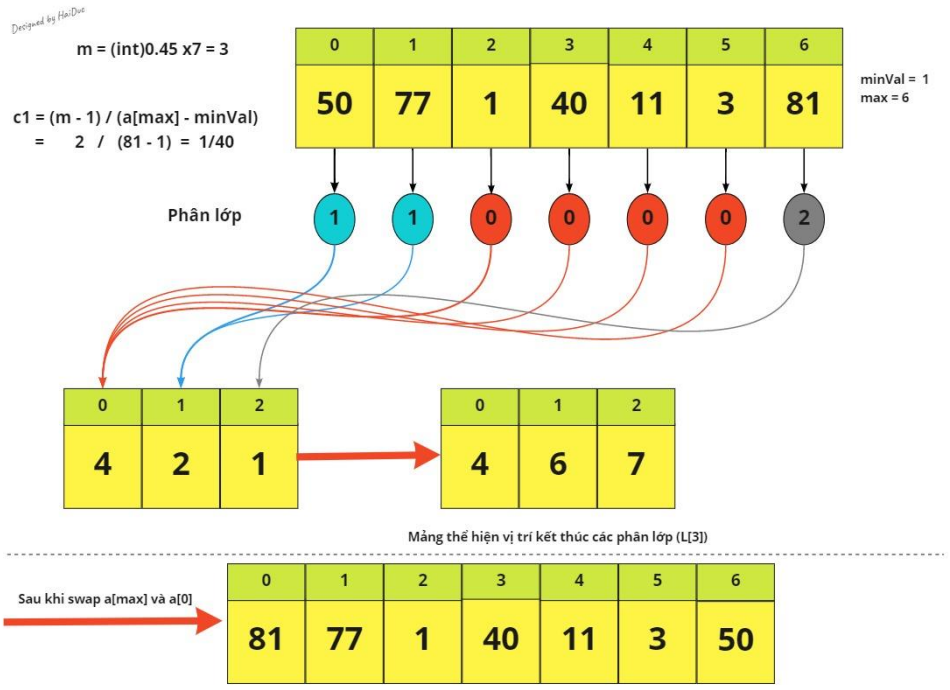
```
    while (j != l[k])
```

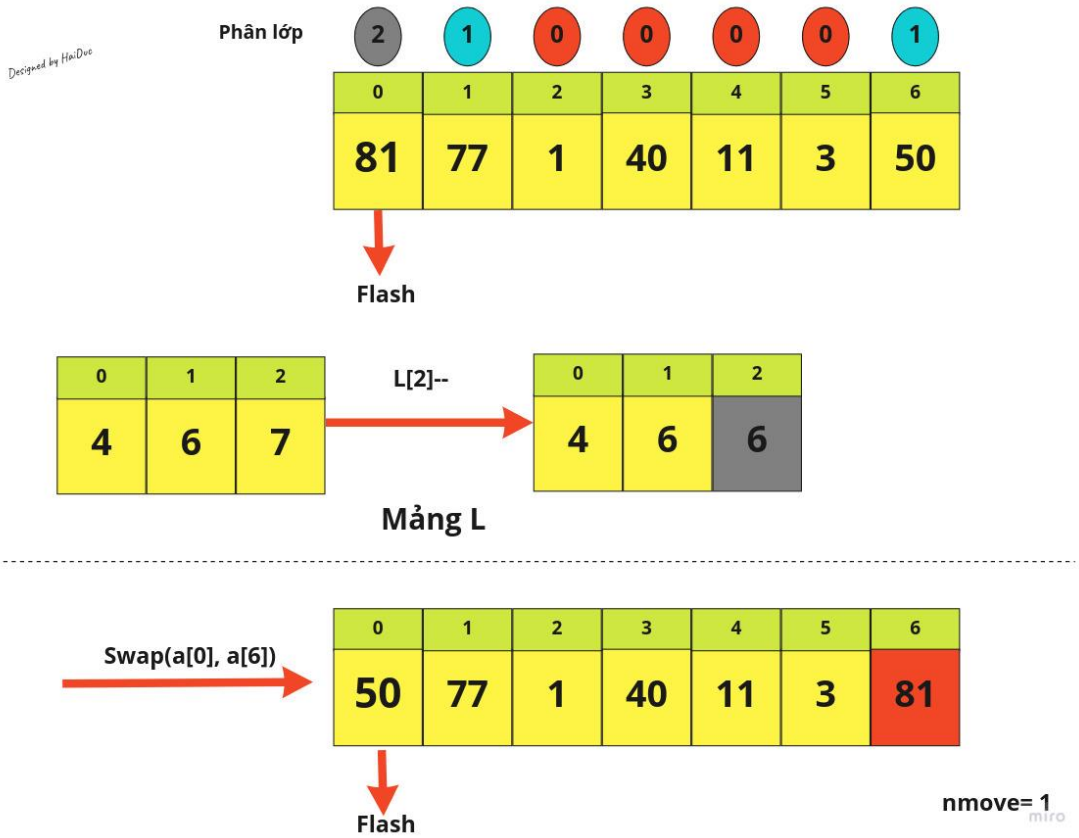
```
            {
                k = int(c1*(flash - minVal));
                int hold = a[t = --l[k]];
                a[t] = flash;
                flash = hold;
                ++nmove;
            }
        }
        delete[] l;
```

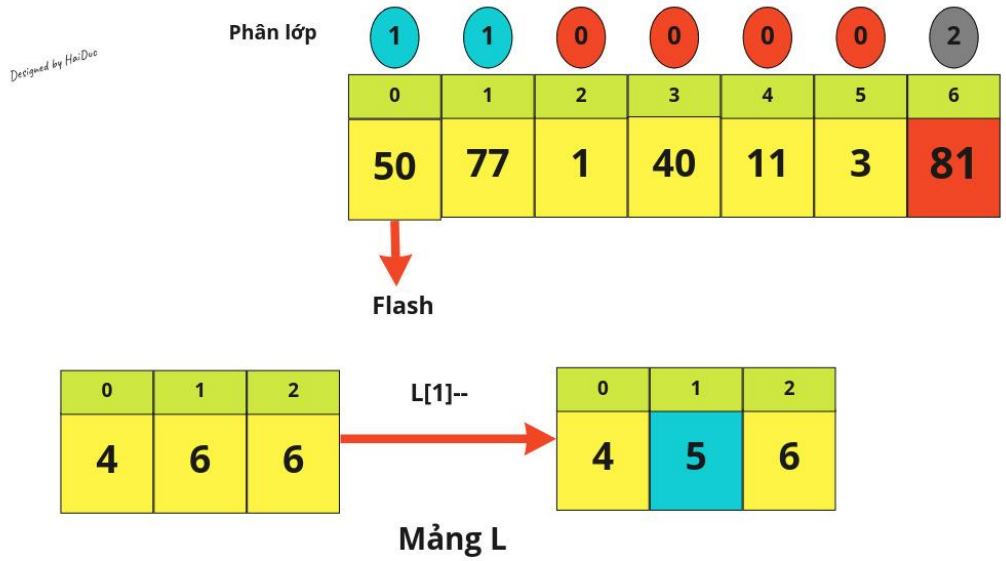
//Cuối cùng, sau bước **hoán vị toàn cục**, mảng của chúng ta hiện tại sẽ được chia thành các lớp(thứ tự các phần tử trong lớp vẫn chưa đúng) do đó để đạt được trạng thái đúng thứ tự thì khoảng cách phải di chuyển của các phần tử là không lớn vì vậy **Insertion Sort** sẽ là thuật toán thích hợp nhất để sắp xếp lại mảng có trạng thái như vậy

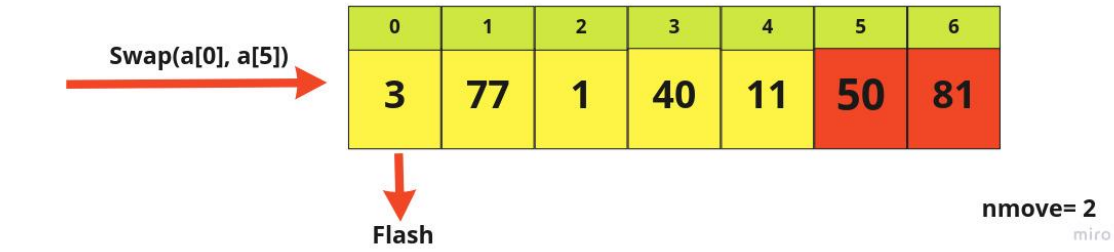
```
        insertionSort(a, n);
}
```

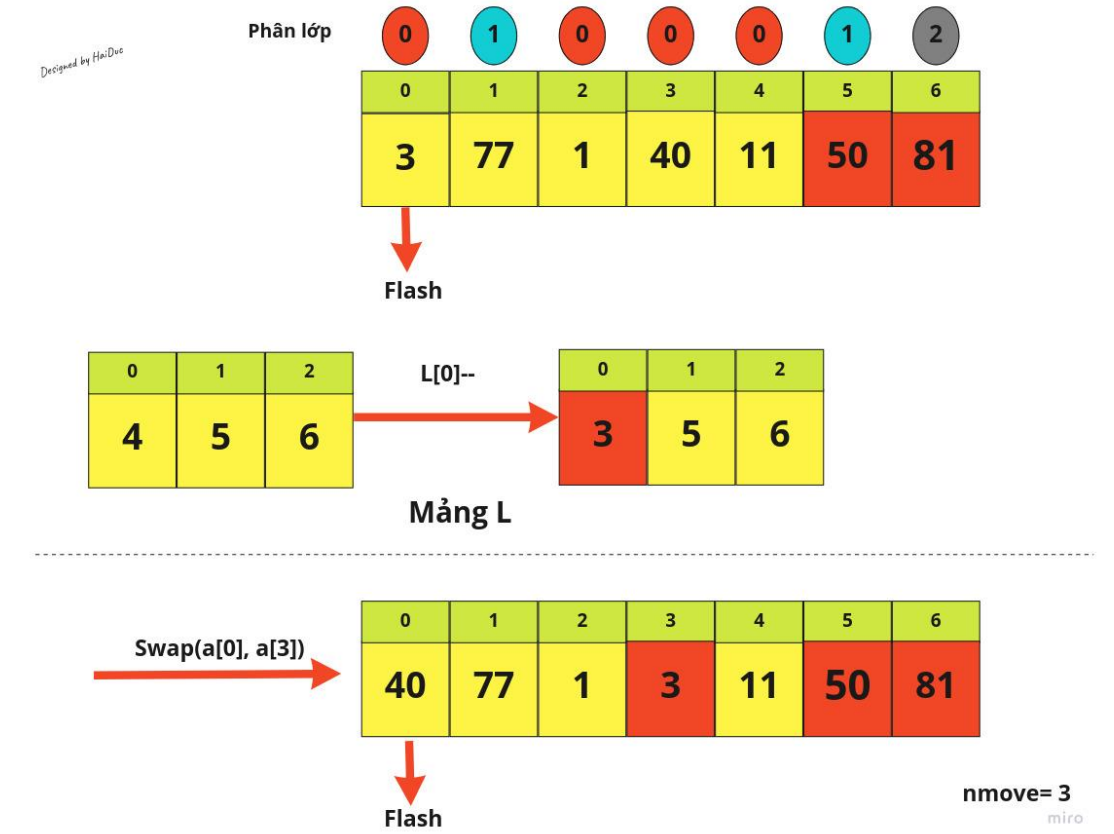
11.3. Mô tả cách thức hoạt động



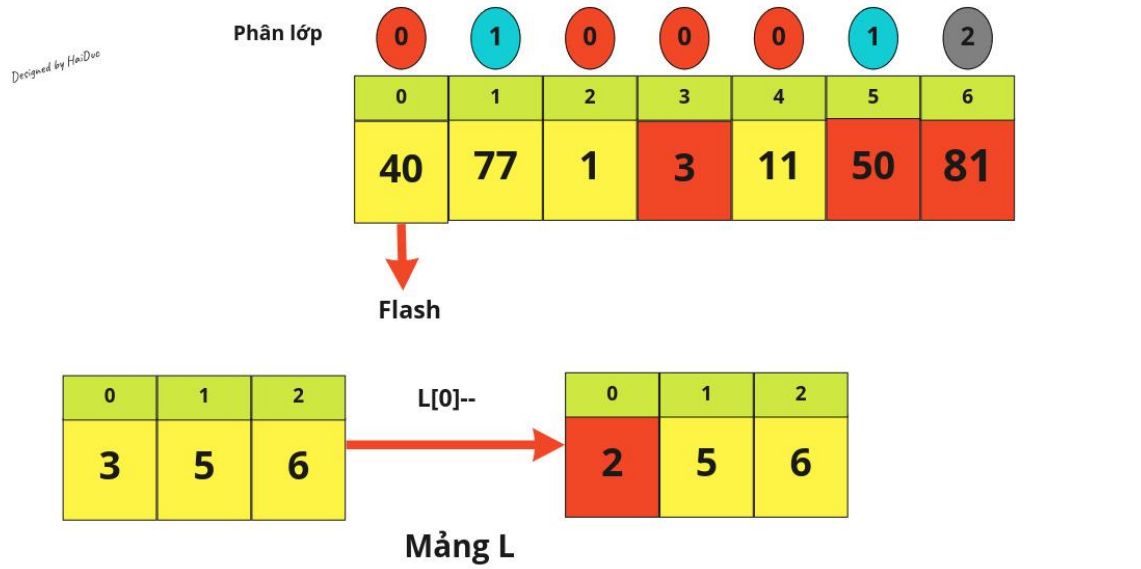


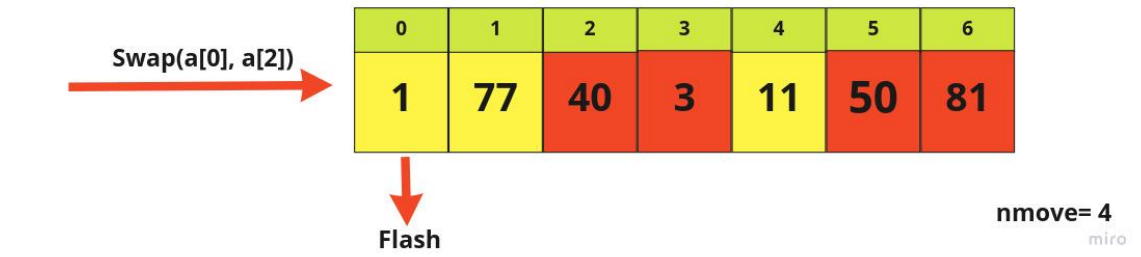

Designed by HaiDuc
Phân lớp
1 1 0 0 0 0 2
0 1 2 3 4 5 6
50 77 1 40 11 3 81
Flash
0 1 2
4 6 6
L[1]--
0 1 2
4 5 6
Mảng L

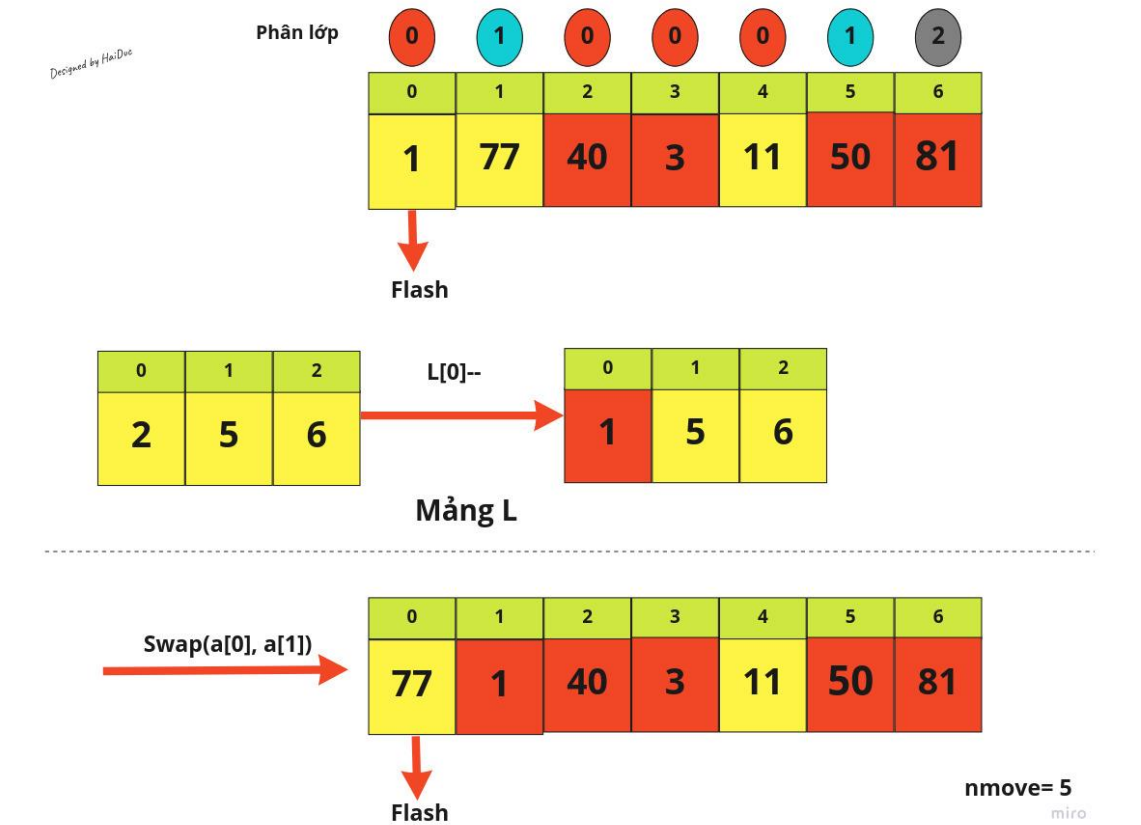

Swap(a[0], a[5])
0 1 2 3 4 5 6
3 77 1 40 11 50 81
Flash
nmove= 2
miro


Designed by HaiDuc
Phân lớp
0 1 0 0 0 1 2
0 1 2 3 4 5 6
3 77 1 40 11 50 81
Flash
0 1 2
4 5 6
L[0]--
0 1 2
3 5 6
Mảng L

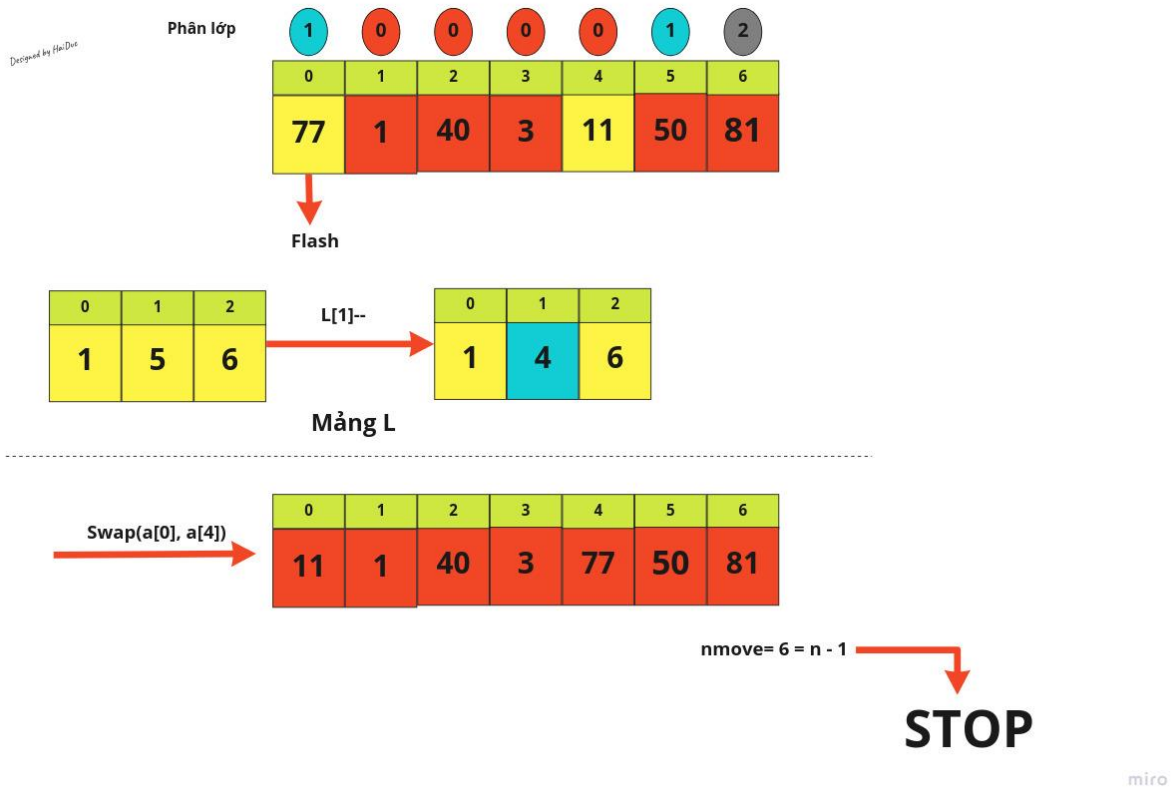
Swap(a[0], a[3])
0 1 2 3 4 5 6
40 77 1 3 11 50 81
Flash
nmove= 3
miro

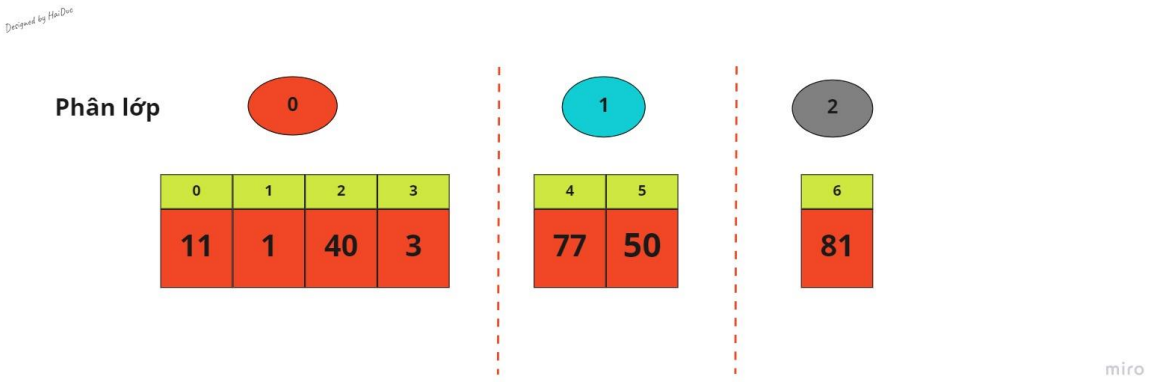

Designed by HaiDuc
Phân lớp
0 1 0 0 0 1 2
0 1 2 3 4 5 6
40 77 1 3 11 50 81
Flash
0 1 2
3 5 6
L[0]--
0 1 2
2 5 6
Mảng L


Swap(a[0], a[2])
0 1 2 3 4 5 6
1 77 40 3 11 50 81
Flash
nmove= 4
miro


Designed by HaiDuc
Phân lớp
0 1 0 0 0 1 2
0 1 2 3 4 5 6
1 77 40 3 11 50 81
Flash
0 1 2
2 5 6
L[0]--
0 1 2
1 5 6
Mảng L

Swap(a[0], a[1])
0 1 2 3 4 5 6
77 1 40 3 11 50 81
Flash
nmove= 5
miro





Đến đây, các phần tử của mảng sẽ về đúng với phân lớp của mình, do đó theo phân tích ở trên, ta dùng **Insertion Sort** để sắp xếp lại mảng này.

11.4. Đánh giá

Độ phức tạp: Tốt nhất O(n), xấu nhất $O(n^2)$,

Điểm mạnh:

- Flash sort không bị ảnh hưởng với thứ tự của dãy.
- Flashsort nhanh hơn heapsort với mọi n và nhanh hơn quicksort với n > 80 . Nó trở nên nhanh gấp đôi so với quicksort ở n = 10000 (Lưu ý rằng các phép đo

này được thực hiện vào cuối những năm 1990, khi hệ thống phân cấp bộ nhớ ít phụ thuộc hơn vào bộ nhớ đệm).

- Yêu cầu bộ nhớ bổ sung duy nhất là vectơ phụ L để lưu trữ giới hạn nhóm và số lượng không đổi của các biến khác được sử dụng. Hơn nữa, mỗi phần tử được di chuyển (thông qua bộ đệm tạm thời, vì vậy hai hoạt động di chuyển) chỉ một lần.

Điểm yếu:

- Tuy nhiên, hiệu quả bộ nhớ này đi kèm với nhược điểm là mảng được truy cập ngẫu nhiên, do đó không thể tận dụng bộ nhớ cache dữ liệu nhỏ hơn toàn bộ mảng.
- Do hoán vị tại chỗ mà flashsort thực hiện trong quá trình phân loại của nó, flashsort không ổn định . Nếu cần sự ổn định, có thể sử dụng mảng thứ hai để các phần tử có thể được phân loại tuần tự. Tuy nhiên, trong trường hợp này, thuật toán sẽ yêu cầu thêm O (n) bộ nhớ.

IV. Experimental results and comments

1. Experimental results

DATA ORDER: RANDOMIZED

Data order: Randomized						
Data size	10,000		30,000		50,000	
Resulting status	Time(ms)	Comparisons	Time(ms)	Comparison	Time(ms)	Comparisons
Selection Sort	159	10^8	1433	9.10^8	3823	25.10^8
Insertion Sort	86	5.10^7	749	$4{,}5.10^8$	2109	$1{,}25.10^9$
Bubble Sort	238	10^8	2395	9.10^8	6856	25.10^8
Heap Sort	2	310,998	6	$1{,}05.10^6$	11	$1{,}84.10^6$
Quick Sort	1	334,137	3	$1{,}23.10^6$	6	$1{,}96.10^6$
Merge Sort	5	583,879	17	$1{,}93.10^6$	31	$3{,}38.10^6$
Shaker Sort	1	$6{,}67.10^7$	5	$5{,}98.10^8$	7	$1{,}66.10^9$
Shell Sort	1	658,552	5	$2{,}24.10^6$	10	$4{,}5.10^6$
Radix Sort	247	140,056	2120	510,070	5954	850,070
Counting Sort	1	70,003	1	210,003	1	315,539
Flash Sort	0.3	87,775	1	268,301	2	422.375

Data order: Randomized						
Data size	100,000		300,000		500,000	
Resulting status	Time(ms)	Comparisons	Time(ms)	Comparison	Time(ms)	Comparisons
Selection Sort	15768	10^{10}	143340	9.10^{10}	401145	25.10^{10}
Insertion Sort	8596	$4{,}98.10^{9}$	81094	$4{,}5.10^{10}$	224732	$12{,}5.10^{10}$
Bubble Sort	27931	10^{10}	258797	9.10^{10}	718366	25.10^{10}
Heap Sort	35	$3{,}94.10^{6}$	81	$1{,}3.10^{7}$	152	$2{,}25.10^{7}$
Quick Sort	18	$4{,}4.10^{6}$	37	$1{,}6.10^{7}$	59	3.10^{7}
Merge Sort	66	$7{,}1.10^{6}$	196	$2{,}33.10^{7}$	318	4.10^{7}
Shaker Sort	16	$6{,}65.10^{9}$	53	$5{,}99.10^{10}$	101	$16{,}6.10^{10}$
Shell Sort	23	10^{7}	78	$3{,}4.10^{7}$	128	$6{,}4.10^{7}$
Radix Sort	24416	$1{,}7.10^{6}$	220819	$5{,}1.10^{6}$	610426	$8{,}5.10^{6}$
Counting Sort	2	565,539	6	$1{,}56.10^{6}$	7	$2{,}56.10^{6}$
Flash Sort	4	840,183	13	$2{,}4.10^{6}$	30	$4{,}17.10^{6}$

DATA ORDER: SORTED

Data order: Sorted						
Data size	10,000		30,000		50,000	
Resulting status	Time(ms)	Comparisons	Time(ms)	Comparison	Time(ms)	Comparisons
Selection Sort	160	10^8	1412	$9.\ 10^8$	3967	$25.\ 10^8$
Insertion Sort	1	19,999	1	59,999	1	99,999
Bubble Sort	146	10^8	1318	$9.\ 10^8$	3652	$25.\ 10^8$
Heap Sort	1	319,779	2	$1{,}07.10^6$	4	$1{,}89.\ 10^6$
Quick Sort	1	10^8	4	$9.\ 10^8$	6	$25.\ 10^8$
Merge Sort	5	475,242	18	$1{,}55.\ 10^6$	32	$2{,}72.\ 10^6$
Shaker Sort	84	20,002	452	60,002	6604	100,002
Shell Sort	1	360,462	6	$1{,}17.\ 10^6$	8	$2{,}1.\ 10^6$
Radix Sort	1	140,056	1	510,070	2	850,070
Counting Sort	0	70,003	1	210,003	1	350,003
Flash Sort	1	117,994	1	353,994	1	584,994

Data order: Sorted						
Data size	100,000		300,000		500,000	
Resulting status	Time(ms)	Comparisons	Time(ms)	Comparison	Time(ms)	Comparisons
Selection Sort	15756	10^{10}	144086	$9.\ 10^{10}$	405570	$25.\ 10^{10}$
Insertion Sort	1	199,999	2	599,999	2	999,999
Bubble Sort	14638	10^{10}	133283	$9.\ 10^{10}$	377586	$25.\ 10^{10}$
Heap Sort	7	4.10^{6}	24	$13{,}3.\ 10^{6}$	42	$23.\ 10^{6}$
Quick Sort	12	10^{10}	44	$9.\ 10^{10}$	73	$25.\ 10^{10}$
Merge Sort	59	$5{,}7.\ 10^{6}$	166	$18{,}6.\ 10^{6}$	321	$32.\ 10^{6}$
Shaker Sort	31126	200,002	306280	600,002	850777	10^{6}
Shell Sort	17	$4{,}5.\ 10^{6}$	53	$15{,}3.\ 10^{6}$	94	$25{,}5.\ 10^{6}$
Radix Sort	3	$1{,}7.\ 10^{6}$	4	$5{,}1.\ 10^{6}$	4	$8{,}5.\ 10^{6}$
Counting Sort	1	700,003	4	$2{,}1.\ 10^{6}$	9	$3{,}5.\ 10^{6}$
Flash Sort	3	589,994	9	$1{,}18.\ 10^{6}$	15	$5{,}9.\ 10^{6}$

DATA ORDER: REVERSE

Data order: reverse						
Data size	10,000		30,000		50,000	
Resulting status	Time(ms)	Comparisons	Time(ms)	Comparisons	Time(ms)	Comparisons
Selection Sort	146,000	10^8	1.312,000	$9.\ 10^8$	3.696,000	$25.\ 10^8$
Insertion Sort	162,000	10^8	1.530,000	$9.\ 10^8$	4.274,000	$25.\ 10^8$
Bubble Sort	195,000	10^8	1.765,000	$9.\ 10^8$	4.789,000	$25.\ 10^8$
Heap Sort	1,000	299528	4,000	1018662	8,000	1791300
Quick Sort	224,000	10^8	1.999,000	$9.\ 10^8$	5.856,000	$25.\ 10^8$
Merge Sort	5,000	476441	15,000	1573465	25,000	2733945
Shaker Sort	0,296	10^8	2,596	$9.\ 10^8$	7,167	$25.\ 10^8$
Shell Sort	1,000	475175	2,000	1554051	4,000	2844628
Radix Sort	1,000	140056	3,000	510070	6,000	850070
Counting Sort	0,000	70003	0,000	210003	1,000	350003
Flash Sort	0,000	100504	1,000	301504	1,000	502504

Data order: reverse						
Data size	100,000		300,000		500,000	
Resulting status	Time(ms)	comparisons	Time(ms)	comparisons	Time(ms)	comparisons
Selection Sort	15.208,000	10^{10}	143.493,000	$9.\ 10^{10}$	415.488,000	$25.\ 10^{10}$
Insertion Sort	19.095,000	10^{10}	168.080,000	$9.\ 10^{10}$	450.610,000	$25.\ 10^{10}$
Bubble Sort	20.334,000	10^{10}	199.102,000	$9.\ 10^{10}$	539.323,000	$25.\ 10^{10}$
Heap Sort	17,000	3824442	51,000	12657736	85,000	22033346
Quick Sort	23.525,000	10^{10}	212.706,000	$9.\ 10^{10}$	616.568,000	$25.\ 10^{10}$
Merge Sort	66,000	5767897	154,000	18708313	259,000	32336409
Shaker Sort	29,691	10^{10}	263,253	$9.\ 10^{10}$	699,937	$25.\ 10^{10}$
Shell Sort	9,000	6089190	32,000	20001852	54,000	33857581
Radix Sort	12,000	1700070	43,000	6000084	71,000	10000084
Counting Sort	2,000	700003	5,000	2100003	7,000	3500003
Flash Sort	3,000	1005004	9,000	3015006	14,000	5025006

DATA ORDER: NEARLY SORTED

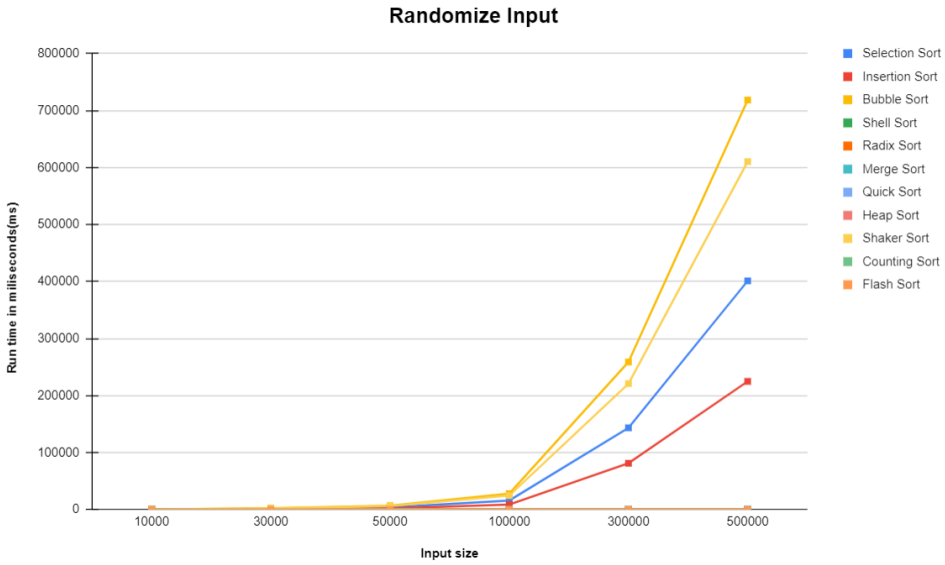
Data order: nearly sorted						
Data size	10,000		30,000		50,000	
Resulting status	Time(ms)	comparisons	Time(ms)	comparisons	Time(ms)	comparisons
Selection Sort	153,000	10^8	1.315,000	$9.\ 10^8$	3.610,000	$25.\ 10^8$
Insertion Sort	0,000	150131	1,000	539935	1,000	795603
Bubble Sort	137,000	10^8	1.194,000	$9.\ 10^8$	3.310,000	$25.\ 10^8$
Heap Sort	1,000	320028	7,000	1079517	13,000	1894148
Quick Sort	167,000	40060155	299,000	296761781	4.599,000	1678692069
Merge Sort	5,000	499378	16,000	1655487	37,000	2831764
Shaker Sort	0,001	182412	0,002	457700	0,003	639922
Shell Sort	1,000	403050	2,000	1279143	5,000	2288362
Radix Sort	1,000	140056	6,000	510070	10,000	850070
Counting Sort	0,000	70003	0,000	210003	1,000	350003
Flash Sort	0,000	117961	1,000	353965	1,000	589969

Data order: nearly sorted						
Data size	100,000		300,000		500,000	
Resulting status	Time(ms)	comparisons	Time(ms)	comparisons	Time(ms)	comparisons
Selection Sort	14.518,000	10^{10}	130.925,000	$9.\ 10^{10}$	371.484,000	$25.\ 10^{10}$
Insertion Sort	1,000	711331	2,000	1040699	3,000	1373575
Bubble Sort	13.310,000	10^{10}	119.800,000	$9.\ 10^{10}$	337.787,000	$25.\ 10^{10}$
Heap Sort	20,000	4038178	59,000	13312229	101,000	23125927
Quick Sort	27.492,000	10^{10}	284.087,000	$8{,}92.10^{10}$	896.166,000	$25.\ 10^{10}$
Merge Sort	67,000	5832489	168,000	18737598	266,000	32114597
Shaker Sort	0,004	505703	0,004	1104490	0,005	1514851
Shell Sort	7,000	4729090	23,000	15460679	38,000	25661417
Radix Sort	19,000	1700070	42,000	6000084	66,000	10000084
Counting Sort	1,000	700003	6,000	2100003	8,000	3500003
Flash Sort	3,000	1179969	11,000	3539967	14,000	5899967

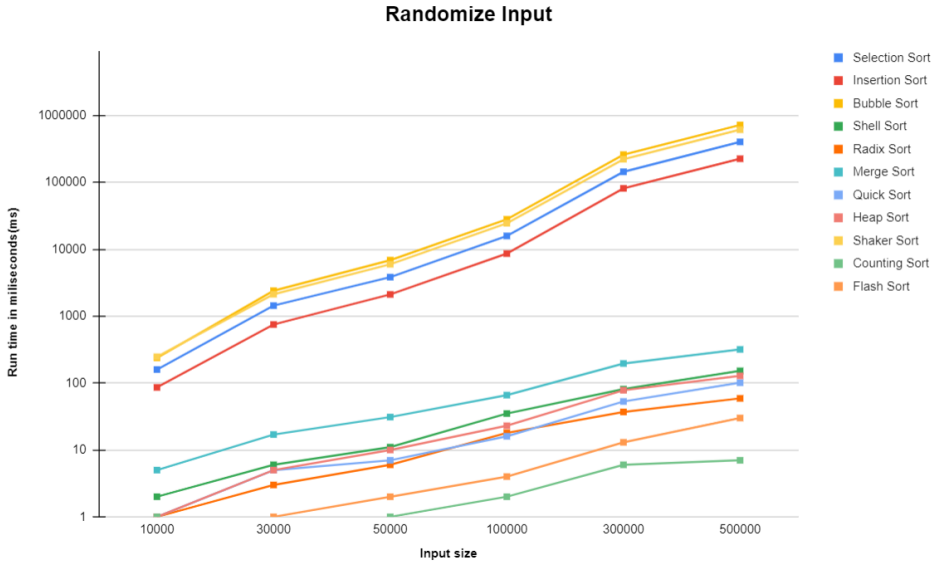
2. Charts and comments

LINE GRAPHS(TIME)

RANDOMIZED SORTED GRAPHs



- Graph theo thang logarit:

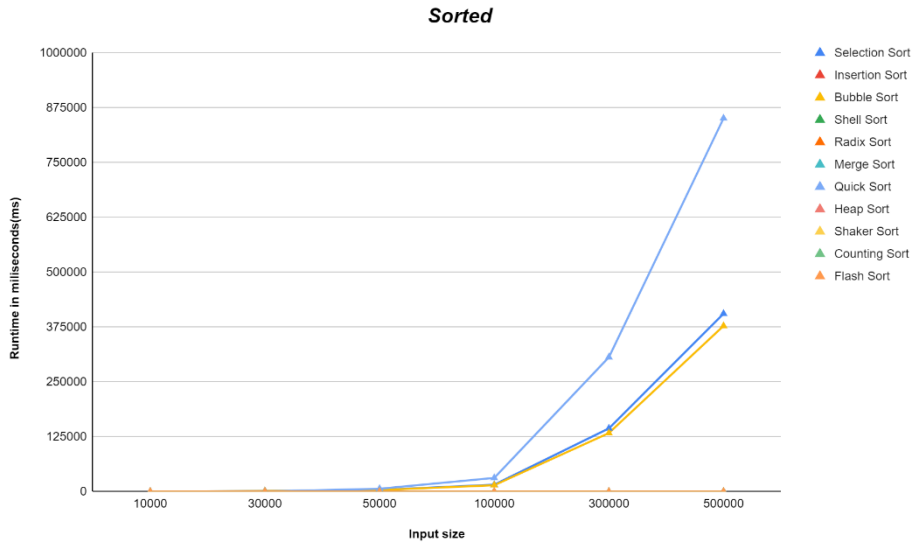


- **Nhận xét:** Những thuật toán có thời gian lớn nhất là Quick, Bubble, Selection, Insertion. Bubble và Selection sort cần nhiều thời gian vì là thuật toán cơ bản nhất, độ phức tạp về thời gian cao. Quick sort và Insertion sort có thời gian chạy lớn vì không còn có thể tận dụng các ưu

thế của chính nó (có thứ tự đặc biệt,…) được nữa khi mảng dữ liệu đầu vào là ngẫu nhiên.

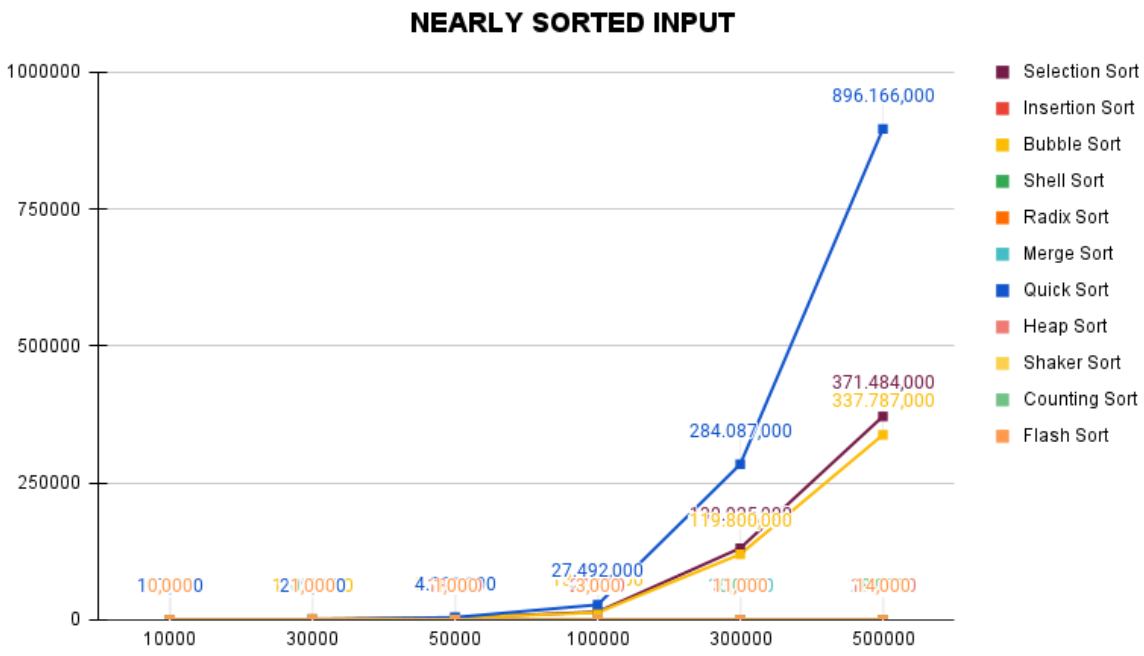
- Các thuật toán không bị ảnh hưởng với thứ tự như Heap, Merge, Radix, Counting, Flash sort hoạt động ổn định như các trường hợp khác.

SORTED GRAPHs



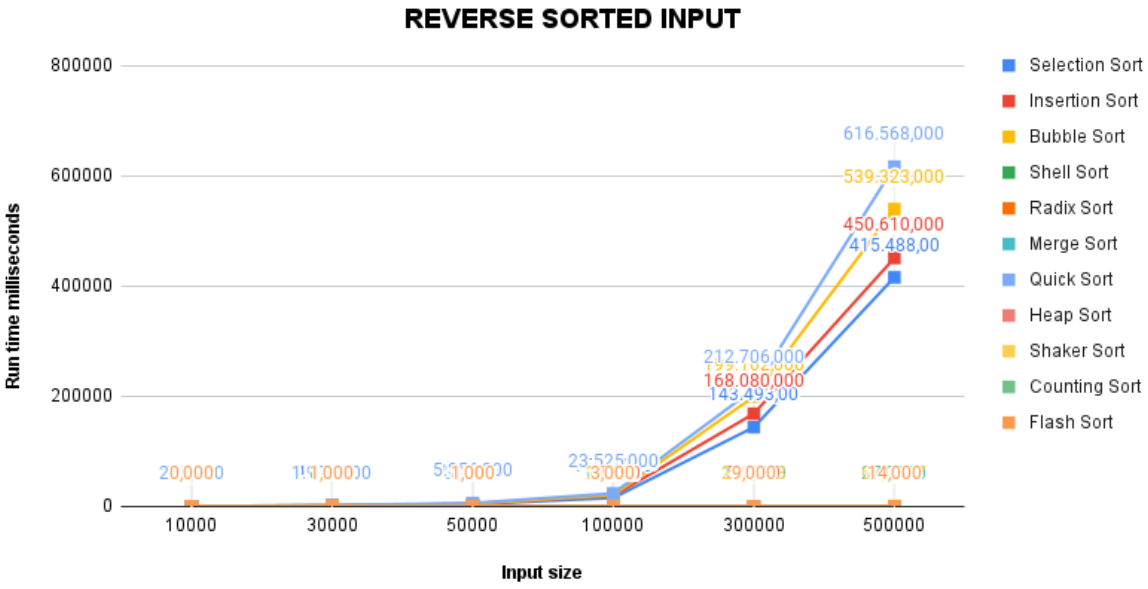
- **Nhận xét:** Thuật toán có thời gian chạy lâu nhất là Quick sort. Bubble và Selection sort dù tốn nhiều thời gian như mọi khi vì là thuật toán cơ bản nhất, độ phức tạp về thời gian cao nhưng thời gian chạy có sự giảm đáng kể do tận dụng được việc dữ liệu đầu vào đã được sắp xếp sẵn. Những thuật toán cũng tận dụng được ưu thế đó là Insertion sort,… Các thuật toán không bị ảnh hưởng với thứ tự vì cách hoạt động khác hoặc hoạt động theo nguyên tắc riêng biệt như Heap, Merge, Radix, Counting, Flash sort vẫn có thời gian chạy ổn đinh, không có sự chênh lệch nhiều như các trường hợp khác.

NEARLY SORTED GRAPHs



- **Nhận xét:** 3 thuật toán có thời gian lớn nhất nhất là Quick, Selection, Bubble sort. Bubble và Selection sort vẫn luôn cần nhiều thời gian vì thuộc tính của nó, Quick sort là vì đã rơi vào trường hợp pivot là phần tử đầu mảng hoặc cuối mảng, gây nên bất lợi lớn (pivot không phân hoạch hiệu quả).
- Các thuật toán tận dụng được ưu thế dãy có thứ tự sẽ thể hiện rõ ràng ưu thế của mình hơn như Insertion sort, Shaker Sort.
- Các thuật toán không bị ảnh hưởng với thứ tự như Heap, Merge, Radix, Counting, Flash sort hoạt động ổn định như các trường hợp khác.

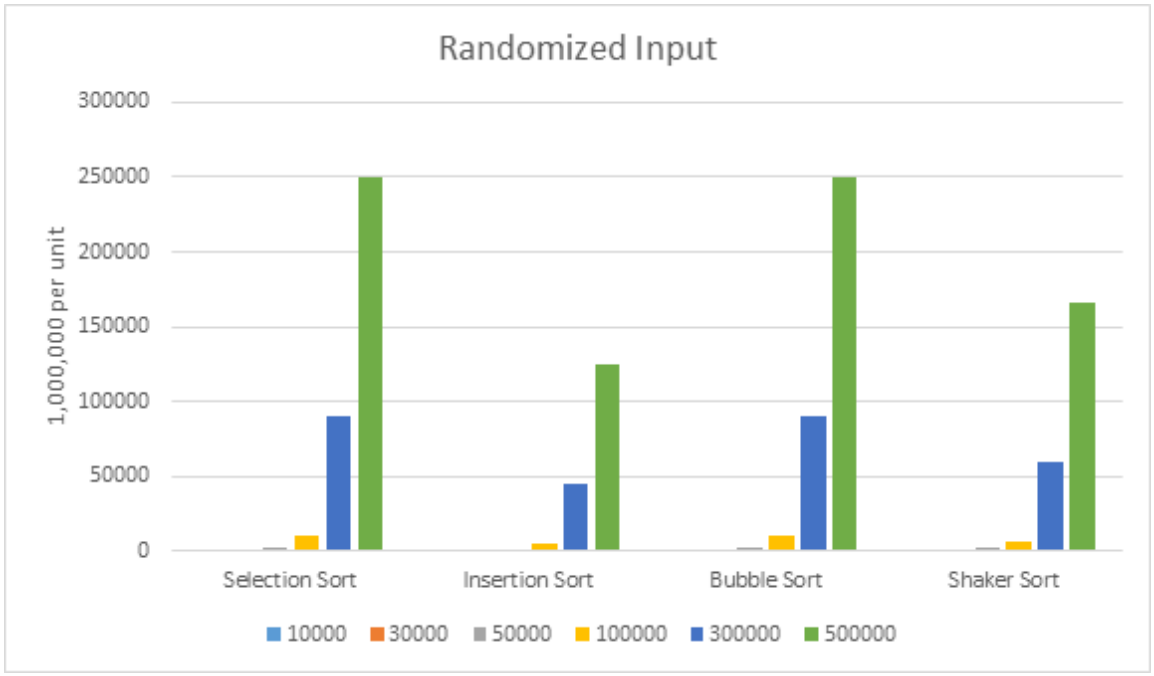
REVERSE GRAPHs

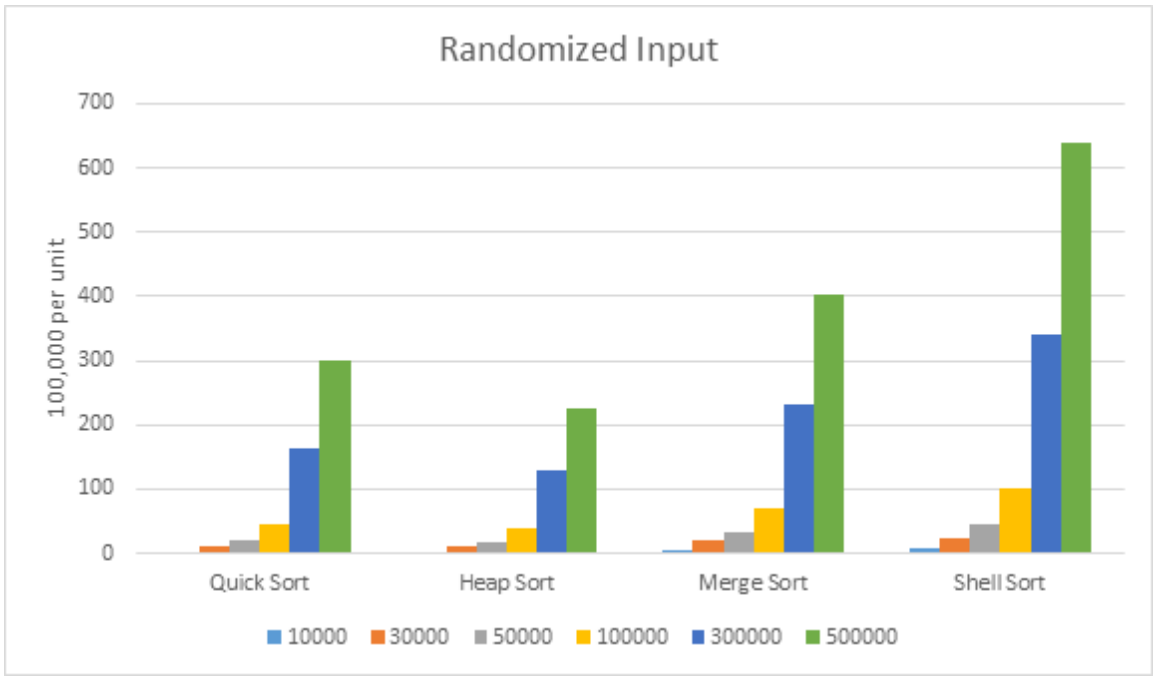


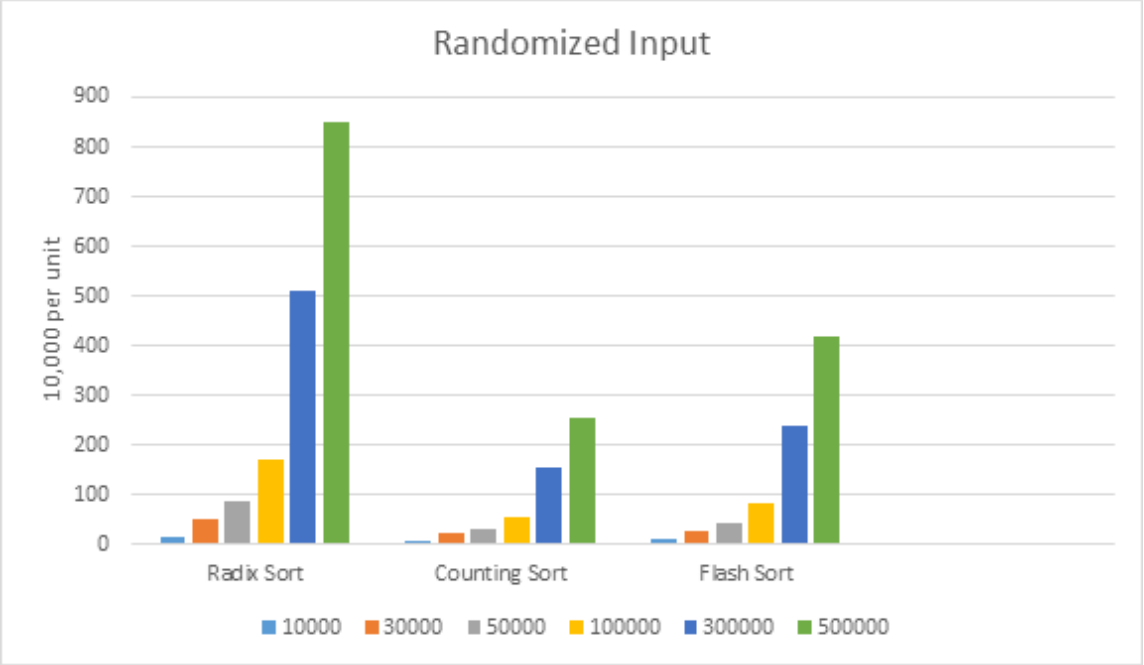
- **Nhận xét:** Những thuật toán có thời gian lớn nhất nhất là Quick, Bubble, Selection, Insertion. Bubble và Selection sort vẫn luôn cần nhiều thời gian vì thuộc tính của nó, vì không còn ưu thế dãy có thứ tự nên Insertion sẽ rơi vào trường hợp xấu nhất của nó. Quick sort là vì đã rơi vào trường hợp pivot là phần tử đầu mảng hoặc cuối mảng, gây nên bất lợi lớn (pivot không phân hoạch hiệu quả).
- Các thuật toán tận dụng được ưu thế dãy có thứ tự sẽ không còn phát huy được như Insertion sort, Shaker Sort.
- Các thuật toán không bị ảnh hưởng với thứ tự như Heap, Merge, Radix, Counting, Flash sort hoạt động ổn định như các trường hợp khác.

BAR CHARTS (COMPARISONS)

RANDOMIZED INPUT CHARTs:



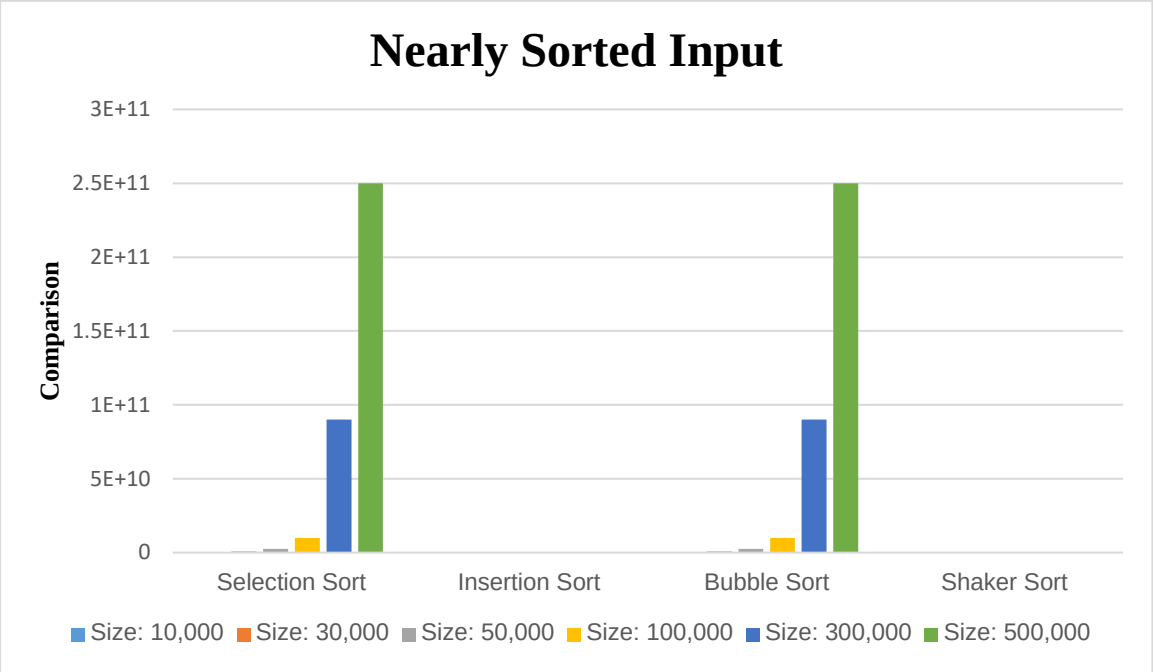


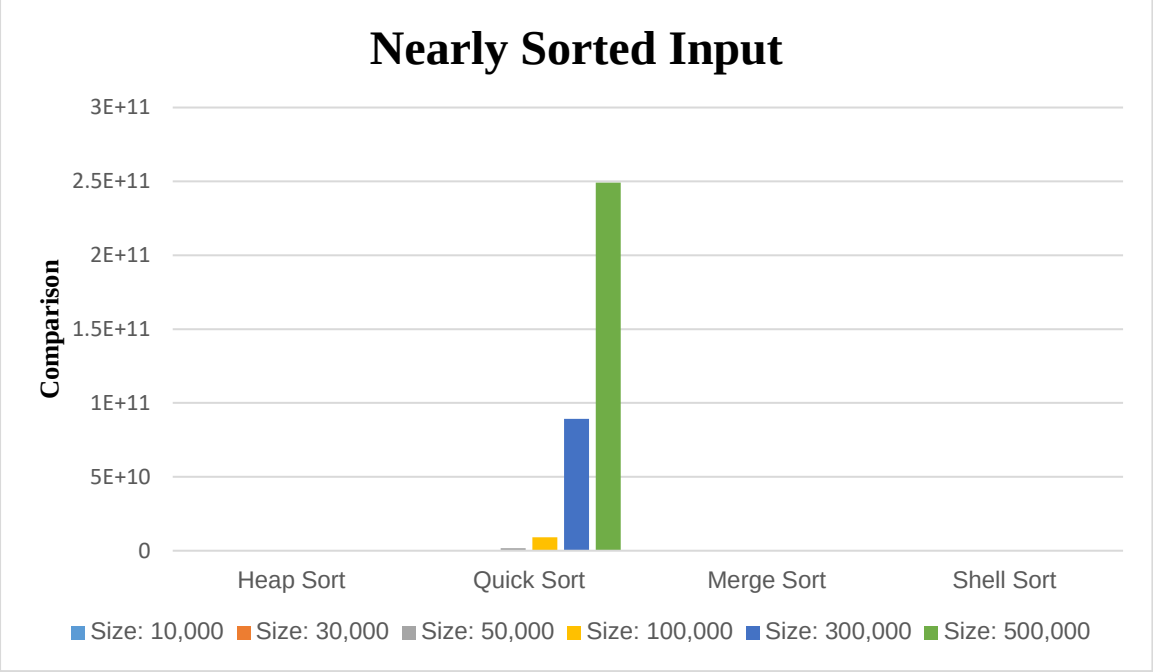


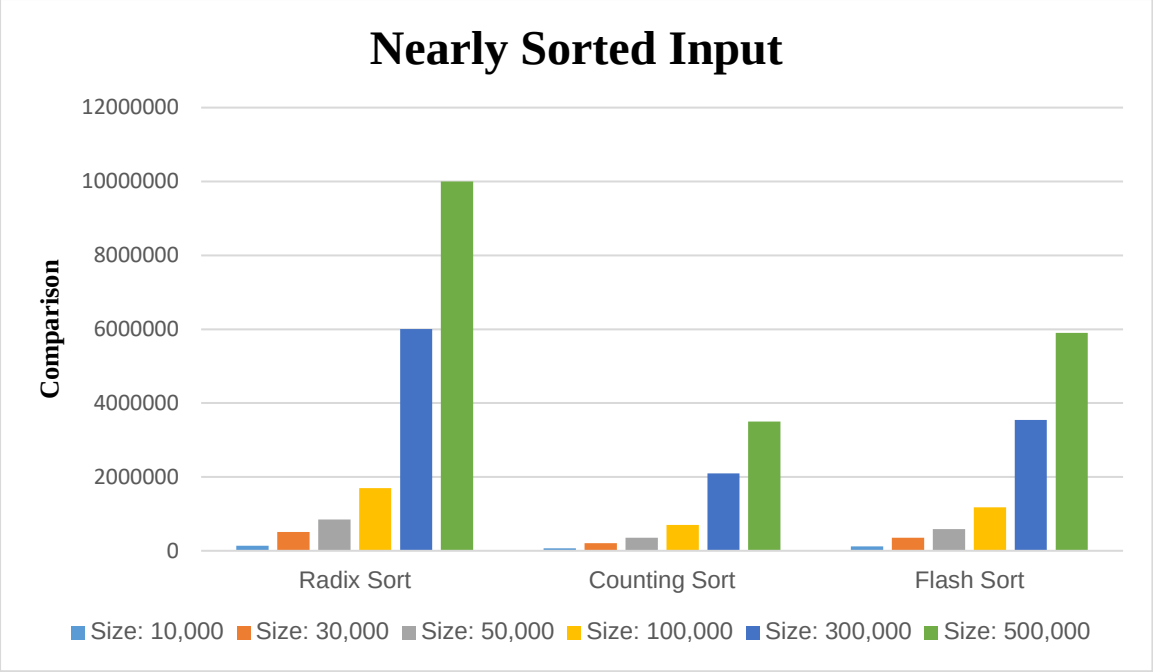
Nhận xét:

- Trong tất cả các datasize Selection sort và Bubble sort luôn có tổng số phép so sánh lớn nhất so với các thuật toán còn lại. (nguyên nhân chủ yếu là do các thuật toán này dựa hoàn toàn trên việc so sánh để sắp xếp).
- Thuật toán có số phép so sánh nhỏ nhất là Counting sort, vì thuật toán này dựa trên việc đếm số lượng của mỗi phần tử rồi sau đó gán lại để tạo thành mảng tăng dần (không cần xây dựng các kiểu cấu trúc đặc biệt hay quá phụ thuộc vào so sánh).
- Từ biểu đồ có thể thấy tốc độ tăng trưởng của Selection, Bubble, Radix Sort là lớn nhất, các thuật toán còn lại tăng trưởng ở mức trung bình (1 số tăng trưởng không ổn định như Quick Sort, Insertion Sort, Shaker Sort vì dữ liệu đầu vào/tính đặc biệt của thuật toán có thể ảnh hưởng khác nhau với các dữ kiện khác nhau.

NEARLY SORTED INPUT CHARTs:



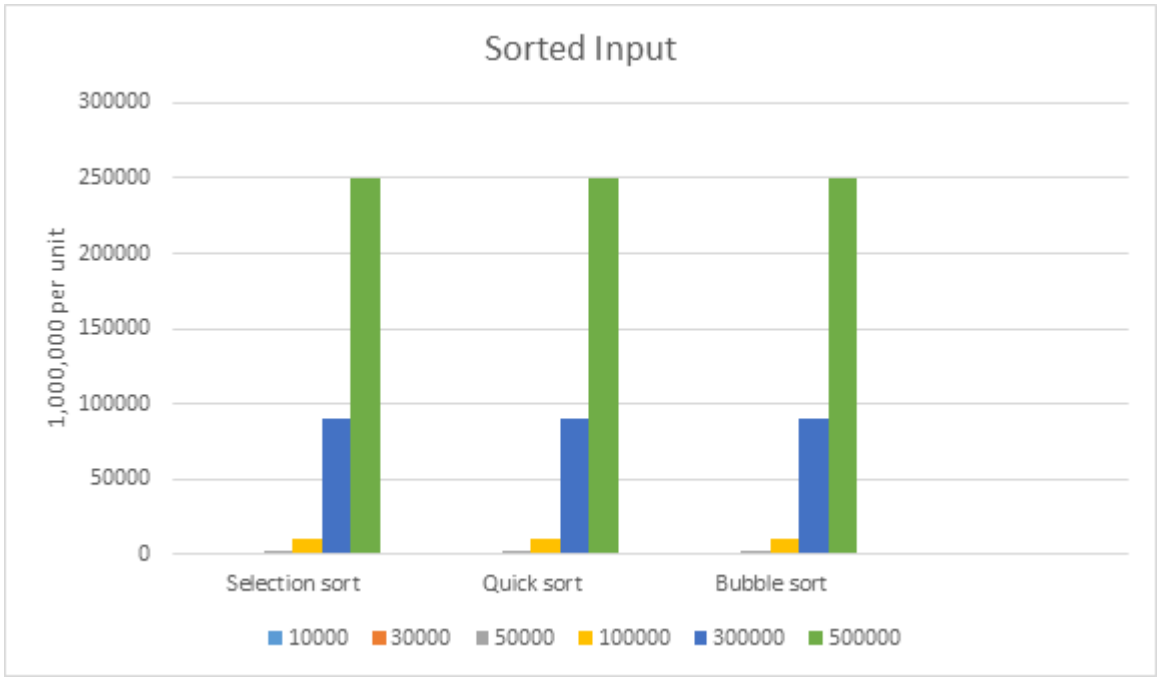


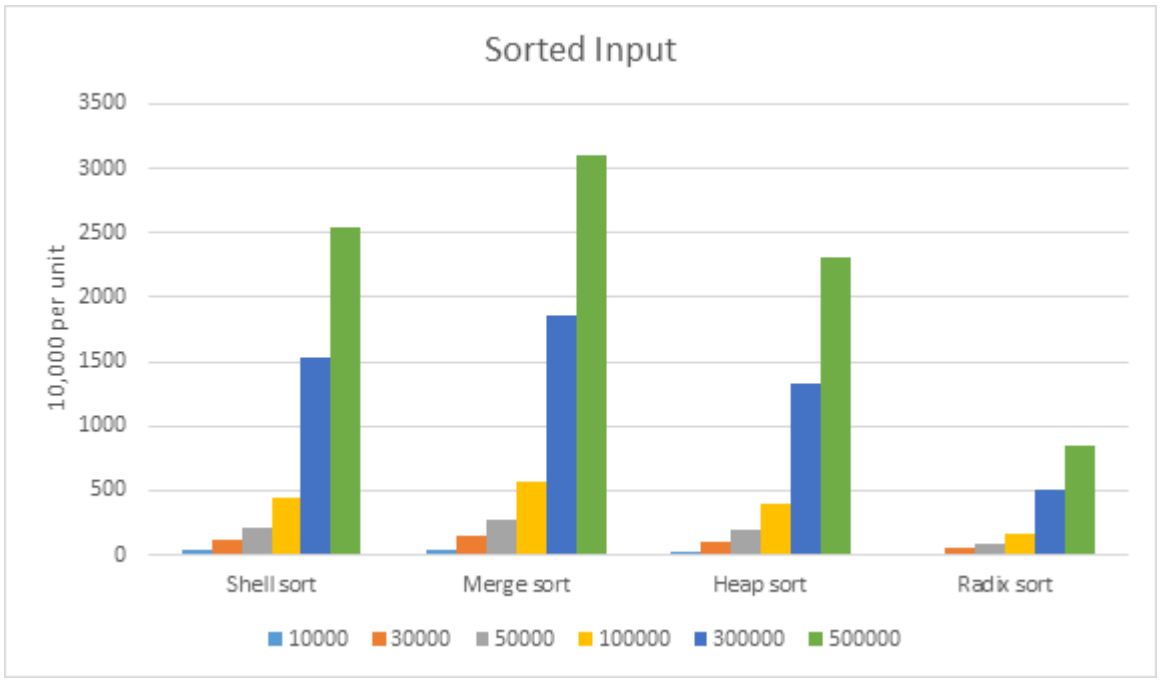


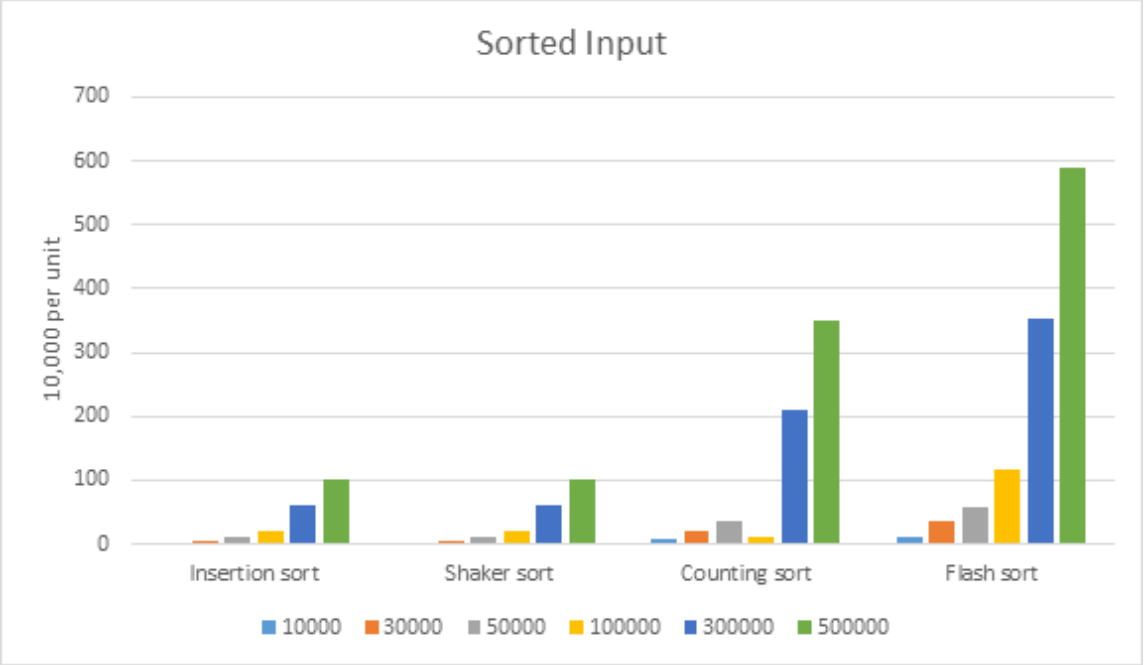
Nhận xét:

- Các thuật toán Bubble, Selection vẫn là các thuật toán có số phép so sánh lớn nhất (đặc tính vốn có), Quick sort vẫn gặp khó khăn khi thuật toán được triển khai với việc chọn pivot là phần tử đầu mảng (gây khó khăn cho việc phân hoạch).
- Các thuật toán làm việc tốt với dãy có thứ tự: Insertion sort, Shaker sort thể hiện rõ điểm mạnh của mình.
- Các thuật toán Heap, Merge, Radix, Counting, Flash sort hoạt động ổn định như các trường hợp khác.

SORTED INPUT CHARTs:



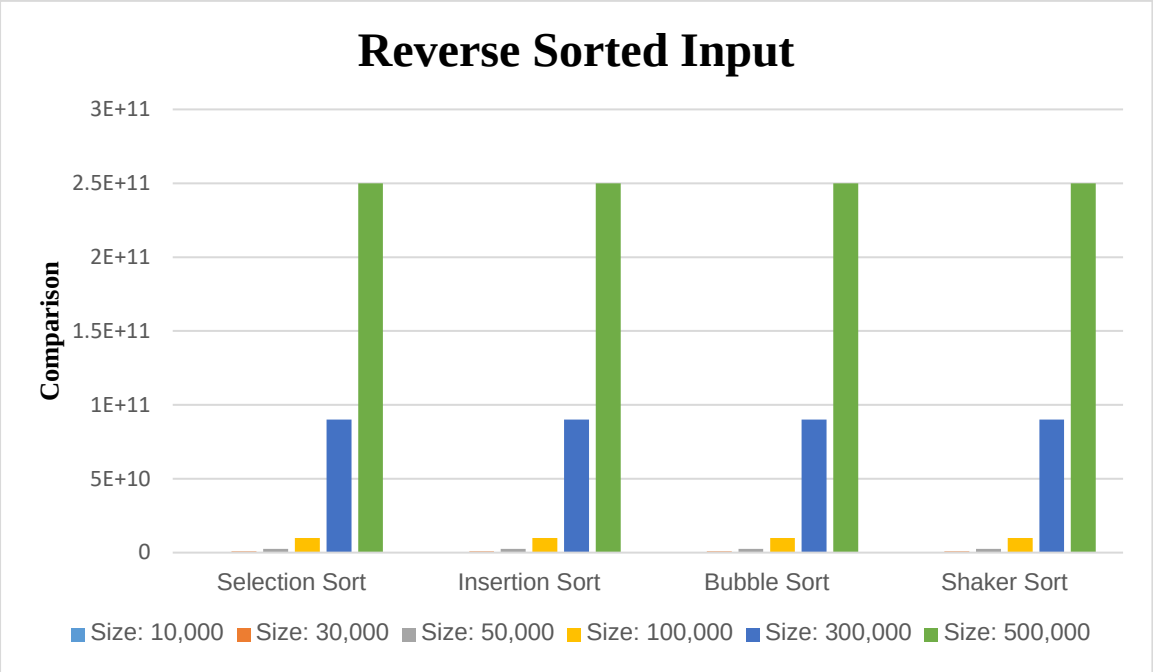


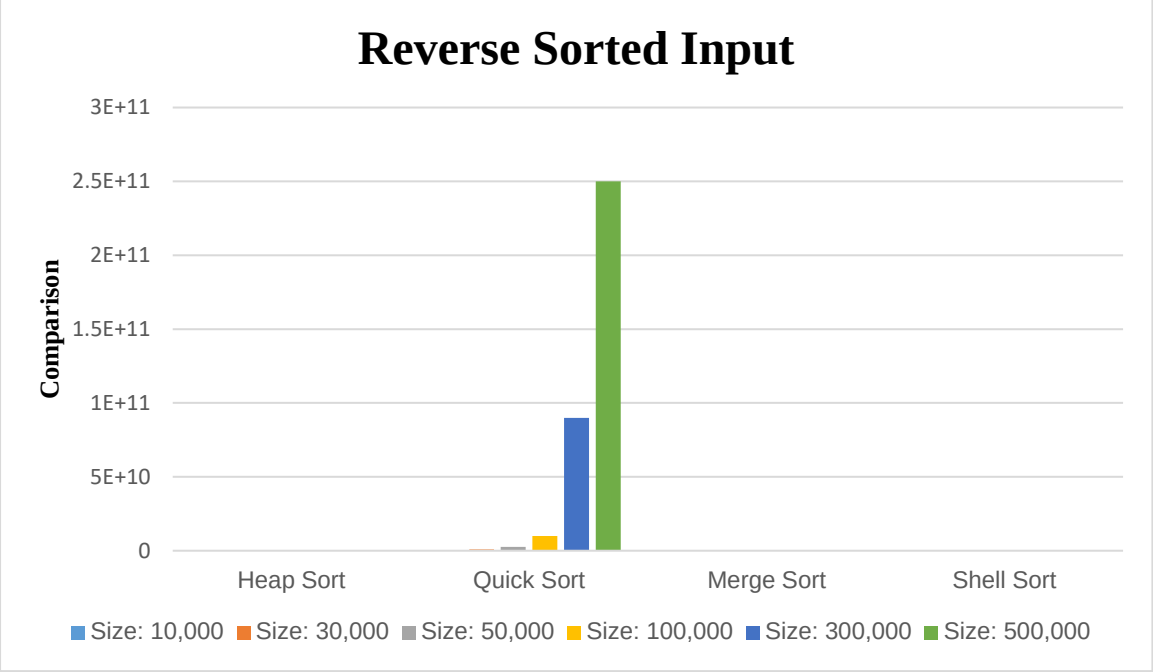


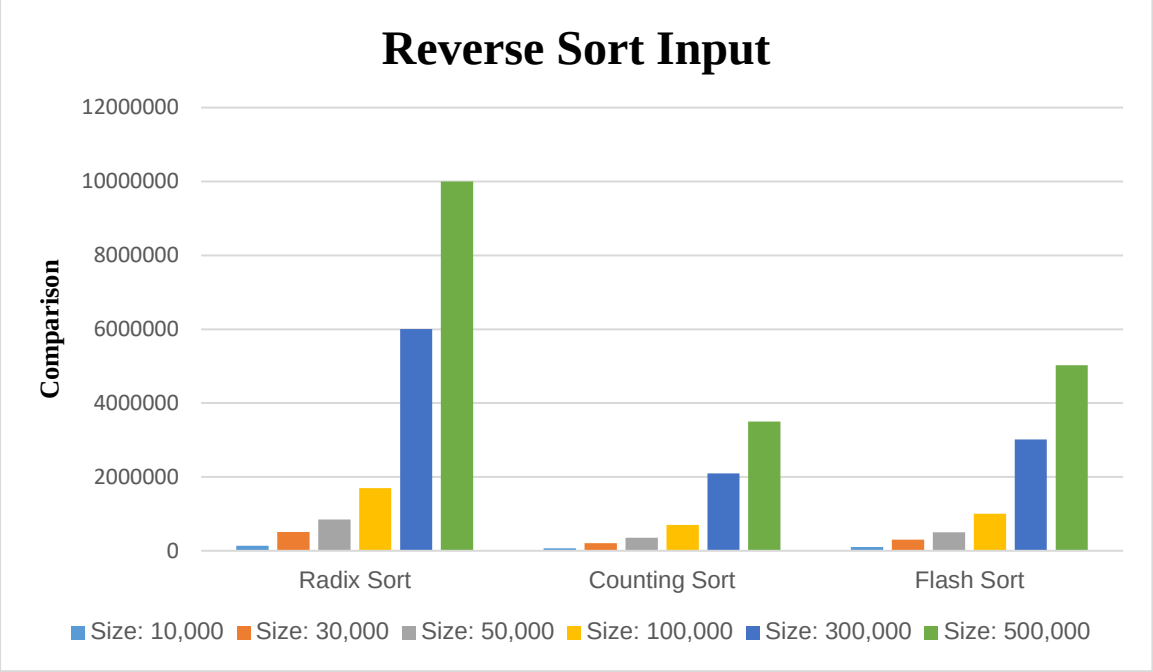
Nhận xét:

- 3 thuật toán có số phép so sánh lớn nhất là Bubble, Quick, Selection sort, Bubble và Selection đã giải thích ở trên, Quick sort là vì đã rơi vào tình hướng chọn pivot gây nên bất lợi (pivot không phân hoạch hiệu quả.
- Các thuật toán Insertion, Shaker có số phép so sánh ít nhất vì 2 thuật toán này tận dụng được ưu thế mảng đã gần đúng với thứ tự.
- Tốc độ tăng trưởng (Insertion và Shaker thấp nhất), đến các thuật toán tầm trung rồi đến Bubble, Quick, Selection có tốc độ tăng trưởng cao nhất.
- Các thuật toán như Merge sort, Heap sort, Radix sort hoạt động rất ổn định vì chúng hoạt động theo 1 quá trình nhất định (không thay đổi khi gặp các yếu tố có lợi/hại với các thuật toán khác.

REVERSE SORTED INPUT CHARTs:







Nhận xét:

- Nhìn chung các đặc điểm, biểu hiện của các thuật toán đối với sorted data giống với nearly sorted data nhưng được thể hiện rõ nét hơn, các thuật toán tận dụng được ưu thế dãy có thứ tự sẽ thể hiện rõ ràng ưu thế của mình hơn, dãy gặp bất lợi Quick sort(nếu pivot phần từ đầu hoặc cuối) thì sẽ càng tệ hơn. Các thuật toán không ảnh hưởng bởi ưu thế đầu vào vẫn hoạt động ổn định.
- Thuật toán Insertion hoạt động tốt với dãy có thứ tự nhưng lại hoạt động ở mức trung bình với dãy ngược thứ tự.

Tổng kết:

- Thuật toán nhanh nhất: Radix Sort (tuy không nhanh bằng 1 số thuật toán khi rơi vào điểm mạnh như Insertion sort) nhưng nó hoạt động ổn định mang lại hiệu quả chắc chắn hơn (tuy có nhiều thuật toán có cùng số phép so sánh)
- Thuật toán chậm nhất là Bubble sort, mặc dù có nhiều thuật toán có cùng số phép so sánh nhưng Bubble sort lại thực hiện theo tác hoán vị nhiều lần nên thời gian chạy lâu nhất.
- Các thuật toán chạy ổn định: Heap Sort, Merge Sort, Selection Sort, Bubble Sort, Radix Sort, Counting Sort, các thuật toán này luôn hoạt động trên chuỗi quy trình định sẵn nên luôn hoạt động như nhau với đầu vào khác nhau.

V. Project organization and Programming notes

- Nếu Quick Sort được triển khai như phần trên sẽ có hạn chế rất lớn đối với các thiết bị không đủ năng lực vì có thể gọi đệ quy quá nhiều lần (rơi vào trường hợp xấu của Quick Sort) nên Quick sort trong phần lập trình được thay đổi qua Non-recursive Quick Sort (Iterative Quick Sort). Phiên bản cải tiến này ứng dụng Stack để loại bỏ việc gọi đệ quy.

VI. References

1. Lecture Notes học phần lý thuyết DSA: DSA-02-Sorting Algorithms-Eng
2. The Advantages & Disadvantages of Sorting Algorithms | Sciencing
3. DAA Merge Sort - javatpoint
4. Quick Sort in C++ (Code with Example) | FavTutor
5. Heap Sort Algorithm: Explanation, Implementation, and Complexity (interviewkickstart.com)
6. Flash Sort - Thuật Toán Sắp Xếp Thần Thánh (codelearn.io)

7. Counting Sort (With Code in Python/C++/Java/C) (programiz.com)
8. Radix Sort - GeeksforGeeks
9. Bubble Sort và Shaker Sort — Giải Thuật Lập Trình (iostream.vn)
10. [Basic-DSAA] Giải thuật sắp xếp - Shell sort. (codelearn.io)
11. [JAVA] SELECTION SORT: Thuật toán sắp xếp chọn (niithanoi.edu.vn)
12. [JAVA] INSERTION SORT: Thuật toán sắp xếp chèn (niithanoi.edu.vn)
13. Flashsort - Wikipedia