ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Thành viên nhóm

20120344	**VƯƠNG TẤN PHÁT**
20120356	**LÊ MINH QUÂN**
20120369	**NGUYỄN THANH TÂN**
20120386	**LÊ PHƯỚC TOÀN**

LAB 03

SORTING

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

Học phần: **Thực hành Cấu trúc dữ liệu và Giải thuật**

Thành phố Hồ Chí Minh – 12/2021

LỜI CẢM ƠN

Báo cáo lab 03 – SORTING là kết quả của quá trình cố gắng không ngừng của cả nhóm và được sự giúp đỡ của các thầy cô, bạn bè và người thân. Qua trang viết này nhóm tác giả xin gửi lời cảm ơn tới những người đã giúp đỡ cả nhóm trong thời gian học tập - nghiên cứu khoa học vừa qua.

Cả nhóm xin tỏ lòng kính trọng và biết ơn sâu sắc đối với cô Phan Thị Phương Uyên đã trực tiếp tận tình hướng dẫn cũng như cung cấp tài liệu thông tin khoa học cần thiết cho lab này. Xin chân thành cảm ơn Lãnh đạo trường Đại học Khoa học Tự nhiên, khoa Công nghệ Thông tin đã tạo điều kiện cho nhóm hoàn thành tốt công việc nghiên cứu khoa học của nhóm.

Cuối cùng, xin gửi lời cảm ơn bạn bè đã giúp đỡ nhóm trong quá trình học tập và thực hiện lab 03. Chúng em xin chân thành cảm ơn!

MỤC LỤC

Phần 1: CÁC THUẬT TOÁN SẮP XẾP

1. Selection Sort:

1.1. Ý tưởng thuật toán:

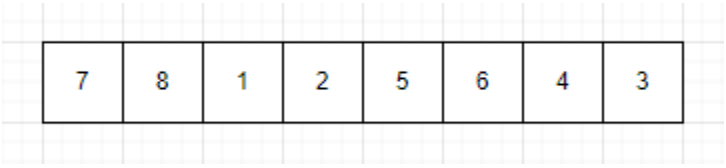
Thuật toán selection sort sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất (giả sử với sắp xếp mảng tăng dần) trong đoạn chưa được sắp xếp và đổi cho phần tử nhỏ nhất đó với phần tử ở đầu đoạn chưa được sắp xếp (không phải đầu mảng). Thuật toán sẽ chia mảng làm 2 mảng con:

- Một mảng con đã được sắp xếp.
- Một mảng con chưa được sắp xếp.

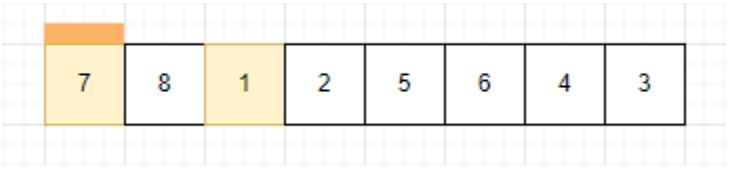
Tại mỗi bước lặp của thuật toán, phần tử nhỏ nhất ở mảng con chưa được sắp xếp sẽ được di chuyển về đoạn đã sắp xếp.

1.2. Thiết lập thuật toán:

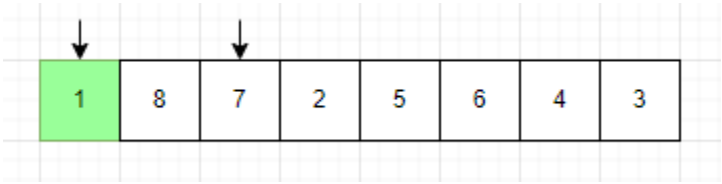
- **Các bước thiết lập thuật toán:**
  - Bắt đầu từ đầu mảng.
  - Duyệt mảng nhằm tìm được phần từ nhỏ nhất mảng.
  - Đổi chỗ của phần tử đó với phần tử đứng ở vị trí đầu tiên của mảng đang duyệt.
  - Sau một lần duyệt mảng thì phần tử đầu của mảng đang duyệt đứng ở đúng vị trí.
  - Trỏ tới phần tử kế tiếp và lặp lại quá trình trên cho đến khi mảng được sắp xếp hết.
- **Ví dụ minh họa:**
  - Cho mảng [8] = {7, 8, 1, 2, 5, 6, 4, 3}



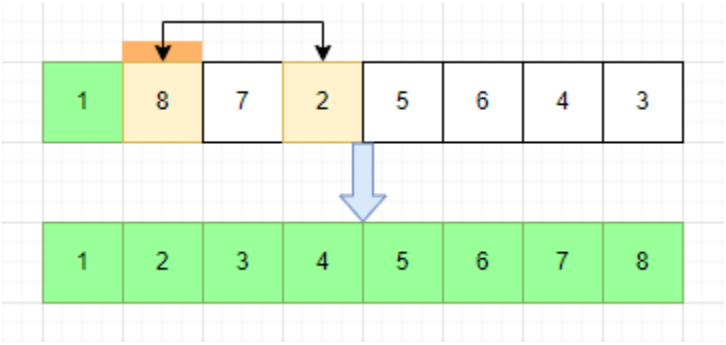
  - Cho i = 0.
  - Tìm phần tử A[min] trong dãy từ A[i] đến A[n-1].



- Đổi chỗ A[i] và A[min].



- Nếu i < n – 1 thì tiếp tục với i++, ngược lại thì dừng.



1.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Thuật toán đơn giản, dễ hiện thực.
  - Có số lần hoán đổi các vị trí ít.
- **Nhược điểm:**
  - Chỉ được áp dụng trong các trường hợp có số lượng phần tử cần so sánh ít.
  - Không nhận biết được mảng đã được sắp xếp.

1.4. Đánh giá độ phức tạp:

- Độ phức tạp về thời gian:

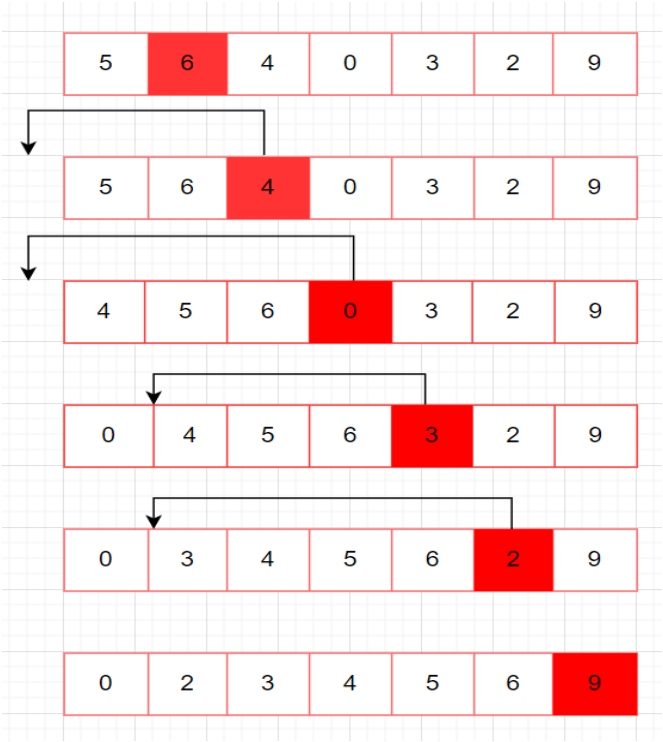
Best case	Average case	Worst case
$O(n^2)$	$O(n^2)$	$O(n^2)$

- Độ phức tạp về không gian: $O(1)$.

2. Insertion Sort:

2.1. Ý tưởng thuật toán:

- Thuật toán sắp xếp chèn thực hiện chia mảng làm hai phần: một phần là mảng con đã được sắp xếp (được tính từ phần tử đầu đến phía trước phần tử đang xét) và phần thứ hai là mảng con chưa được sắp xếp (phần còn lại).



- Thuật toán sắp xếp chèn sẽ tìm kiếm liên tiếp các phần tử của mảng và nếu phần tử đó không có thứ tự (tức là thuộc mảng con chưa được sắp xếp) thì sẽ chèn vào vị trí thích hợp của mảng con đã được sắp xếp.

2.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Khởi tạo mảng con đã được sắp xếp với 1 phần tử (phần tử đầu tiên của mảng).
  - Duyệt từng phần tử của mảng (bắt đầu từ phần tử thứ 2), tại mỗi phần tử thứ i đang duyệt ta sẽ đặt phần tử đó vào đoạn từ [0, i] sao cho dãy số đó đảm bảo tính tăng dần, lúc này mảng con đã được sắp xếp với 1 phần tử ban đầu sẽ được tăng lên thêm một phần tử.
  - Lặp cho tới khi duyệt hết tất cả các phần tử trong mảng.
- **Ví dụ minh họa:**
  - Cho mảng A = [5, 6, 4, 0, 3, 2, 9], n = 7.
  - Ta cho mảng con B đã được sắp xếp có một phần tử là 5, phần tử đầu tiên trong mảng (i = 0).

- Ta xét ở vị trí thứ 2 (i = 1), ta thấy A[1] = 6 > B[0] = 5, mảng con B lúc này có thêm một phần tử là 6, B=[5, 6].
- Ta xét đến vị trí thứ 3 (i = 2), ta thấy A[2] = 4 < B[0] và B[1], mảng con B lúc này được chèn thêm một phần tử là 4, B[4, 5, 6].
- Ta xét đến vị trí thứ 4 (i = 3), ta thấy A[3] = 4 < B[0], B[1] và B[2], mảng con B lúc này được chèn thêm một phần tử là 0, B[0, 4, 5, 6].
- Ta xét đến vị trí thứ 5 (i = 4), ta thấy B[0] = 0 < A[3] = 3 < B[1] và B[2], mảng con B lúc này được chèn thêm một phần tử là 3, B[0, 3, 4, 5 ,6].
- Ta xét đến vị trí thứ 6 (i = 5), ta thấy B[0] = 0 < A[5] = 2 < B[1], B[2], B[3] và B[4], mảng con B lúc này được chèn thêm một phần tử là 2, B[0, 2, 3, 4, 5, 6].
- Ta xét đến vị trí cuối cùng (i = 6), ta thấy A[6] = 9 < B[0], B[1], B[2], B[3], B[4] và B[5], mảng con B lúc này được chèn thêm một phần tử là 2, B[0, 2, 3, 4, 5, 6, 9].
- Mảng sau khi đã được sắp xếp là: 0 2 3 4 5 6 9.

2.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Thuật toán thích hợp đối với mảng đã được sắp xếp một phần hoặc có kích thước nhỏ.
- **Nhược điểm:**
  - Đối với mảng có kích thước lớn thì thời gian chạy sẽ khá lâu.

2.4. Đánh giá độ phức tạp:

- Độ phức tạp về thời gian:

Best case	Average case	Worst case
$O(n)$	$O(n^2)$	$O(n^2)$

- Độ phức tạp về không gian: $O(1)$.

2.5. Các biến thể khác của Insertion Sort:

- Shell Sort
- Binary Insertion Sort

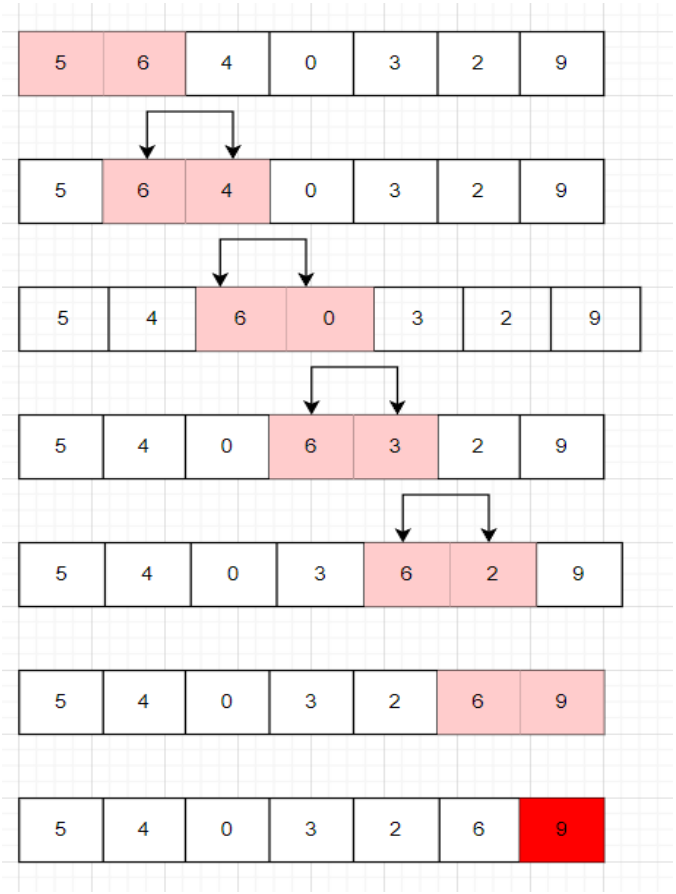
3. Bubble Sort:

3.1. Ý tưởng thuật toán:

- Thuật toán sắp xếp nổi bọt thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự cho đến khi dãy số được sắp xếp.

3.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Ta xét hai phần tử liên tiếp nhau bắt đầu từ đầu mảng, nếu phần tử thứ i lớn hơn phần tử thứ $i + 1$ thì đổi chỗ, sau khi tiến hành xong thì ta sẽ có phần tử lớn nhất ở cuối mảng.
  - Lúc này xem như ta bỏ phần tử cuối cùng ra khỏi mảng và duyệt tiếp tục $n - 1$ phần tử còn lại trong mảng, cứ tiếp tục cho đến khi mảng chỉ còn đúng 1 phần tử.
- **Ví dụ minh họa:**
  - Cho mảng $A = [5, 6, 4, 0, 3, 2, 9]$, $n = 7$.



- Duyệt mảng từ đầu đến kế cuối, xét lần lượt hai phần tử liên tiếp kề nhau, nếu A[i] > A[i+1] thì đổi chỗ hai phần tử.
- Ta thấy A[0] = 5 < A[1] = 6, nên ta tiếp tục xét phần tử kế tiếp.
- Ta thấy A[1] = 6 > A[2] = 4, nên đổi chỗ A[1] và A[2].
- Ta thấy A[2] = 6 > A[3] = 0, nên đổi chỗ A[2] và A[3].
- Ta thấy A[3] = 6 > A[4] = 3, nên đổi chỗ A[3] và A[4].
- Ta thấy A[4] = 6 > A[5] = 2, nên đổi chỗ A[4] và A[5].
- Ta thấy A[5] = 6 < A[6] = 9, kết thúc lần duyệt đầu tiên.
- Sau khi kết thúc lần duyệt đầu tiên thì ta đã đưa được phần tử lớn nhất ra cuối mảng, cho nên những lần duyệt tiếp theo ta chỉ cần duyệt đến vị trí trước vị trí phần tử lớn nhất vừa mới đưa ra cuối mảng, cụ thể trong bài thì trong lần duyệt tiếp theo chỉ cần duyệt đến vị trí i = 5.
- Ta cứ duyệt cho đến khi đưa hết tất cả các phần tử lớn nhất ra cuối mảng đang xét.
- Mảng sau khi đã được sắp xếp là: 0 2 3 4 5 6 9.

3.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Thuật toán dễ hiểu, dễ hình dung, có thể code lại dễ dàng.
  - Thích hợp với những mảng có số lượng phần tử nhỏ.
- **Nhược điểm:**
  - Không nhận diện được tình trạng dãy đã được sắp xếp hay chưa.
  - Các phần tử nhỏ được đưa về vị trí đúng rất nhanh nhưng các các phần tử lớn lại được đưa về vị trí đúng rất chậm.

3.4. Đánh giá độ phức tạp:

- Độ phức tạp về thời gian:

Best case	Average case	Worst case
$O(n)$	$O(n^2)$	$O(n^2)$

- Độ phức tạp về không gian: $O(1)$.

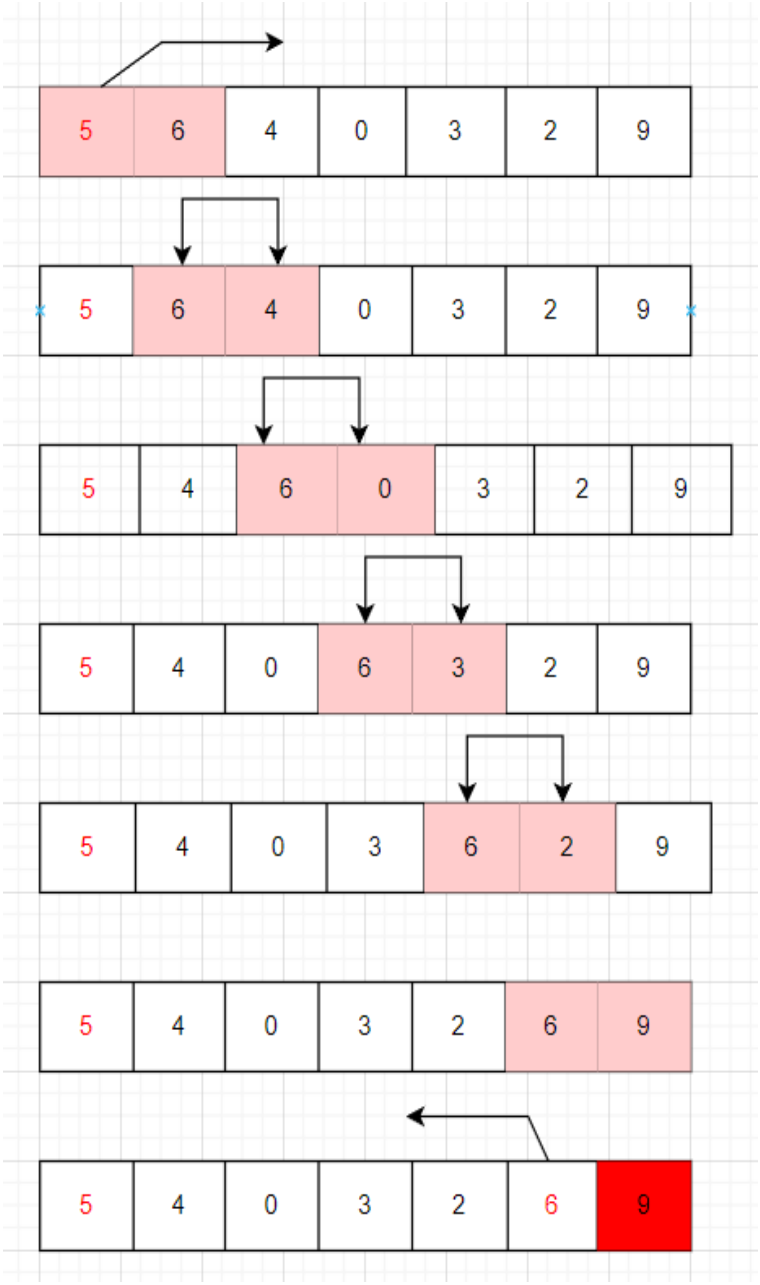
4. Shaker Sort:

4.1. Ý tưởng thuật toán:

- Đây là một phiên bản cải tiến của thuật toán Bubble Sort, ý tưởng cũng tương tự Bubble Sort nhưng thay vì chỉ xét một chiều để đưa phần tử lớn nhất ra cuối dãy thì thuật toán Shaker Sort sẽ xét trên cả hai chiều để tìm phần tử nhỏ nhất đưa ra đầu dãy và lớn nhất đưa ra cuối dãy cứ làm như thế cho đến khi dãy số được sắp xếp.

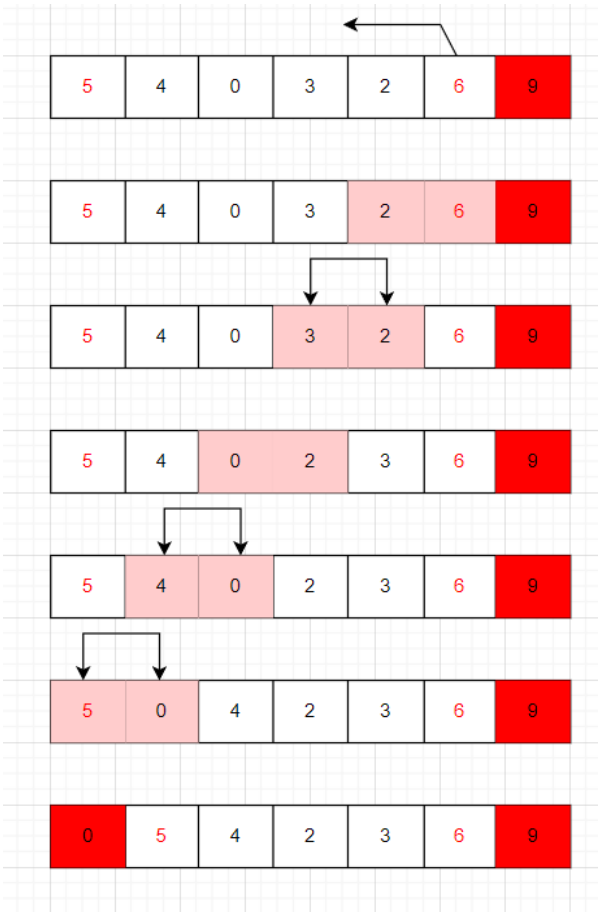
4.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Gán cận trên bằng n - 1, cận dưới bằng 0.
  - Duyệt xuôi chiều dãy số (từ cận dưới đến cận trên - 1) để đưa phần tử lớn nhất ra cuối dãy và cập nhật lại vị trí cận trên của dãy (xem như loại bỏ phần tử lớn nhất đó ra khỏi dãy, chỉ duyệt các phần tử còn lại).
  - Duyệt ngược chiều dãy số (bắt đầu từ cận trên vừa cập nhật đến cận dưới) để đưa phần tử nhỏ nhất ra đầu dãy và cập nhật lại vị trí cận dưới của dãy (xem như loại bỏ phần tử nhỏ nhất đó ra khỏi dãy, chỉ duyệt các phần tử còn lại).
  - Cứ tiếp tục làm nếu cận trên vẫn nhỏ hơn cận dưới và dừng lại khi cận trên bằng cận dưới.
- **Ví dụ minh họa:**



- Cho mảng A = [5, 6, 4, 0, 3, 2, 9], n = 7.
- Duyệt xuôi chiều bắt đầu từ i = 0 đến i = 5, duyệt tương tự như Bubble Sort.
- Xét lần lượt hai phần tử liên tiếp kề nhau, nếu A[i] > A[i+1] thì đổi chỗ hai phần tử.
- Ta thấy A[0] = 5 < A[1] = 6, nên ta tiếp tục xét phần tử kế tiếp.
- Ta thấy A[1] = 6 > A[2] = 4, nên đổi chỗ A[1] và A[2].
- Ta thấy A[2] = 6 > A[3] = 0, nên đổi chỗ A[2] và A[3].

- Ta thấy A[3] = 6 > A[4] = 3, nên đổi chỗ A[3] và A[4].
- Ta thấy A[4] = 6 > A[5] = 2, nên đổi chỗ A[4] và A[5].
- Ta thấy A[5] = 6 < A[6] = 9, kết thúc lần duyệt xuôi chiều đầu tiên, lưu lại vị trí để bắt đầu duyệt trái chiều là vị trí i = 5.



- Duyệt trái chiều chiều bắt đầu từ i = 5 (đã lưu ở trên) về i = 1, duyệt tương tự như Bubble Sort.

4.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Thích hợp với những mảng có số lượng phần tử nhỏ
  - Nhận diện được tình trạng dãy đã được sắp xếp hay chưa
- **Nhược điểm:**
  - Đối với mảng có kích thước lớn thì thời gian chạy sẽ khá lâu.

4.4. Đánh giá độ phức tạp:

- Độ phức tạp về thời gian:

Best case	Average case	Worst case
$O(n)$	$O(n^2)$	$O(n^2)$

- Độ phức tạp về không gian: $O(1)$.

4.5. Các biến thể của Shaker Sort:

Shaker Sort là một phiên bản biến thể của Bubble Sort, cũng có thể nói Shaker Sort là một phiên bản cải tiến của Bubble Sort.

5. Shell Sort:

Shell sort còn được gọi là phương pháp của Shell, là thuật toán sắp xếp do Donald L. Shell công bố trong bài viết A High-Speed Sorting Procedure (tạm dịch: Một phương pháp sắp xếp tốc độ cao) được đăng trên tờ báo Communications of the ACM vào năm 1959.

5.1. Ý tưởng thuật toán:

Shell sort phân hoạch mảng ra thành *những mảng con*, phần tử trong mỗi mảng thường không có địa chỉ liên tục với nhau trên vùng nhớ và cách nhau một khoảng nhất định. Với mỗi mảng con, áp dụng thuật toán insertion sort, lặp lại quá trình với khoảng cách nhỏ hơn (số lượng mảng con cũng nhỏ hơn) cho đến khi khoảng cách còn 0, dùng 1 lần insertion sort nữa thì mảng được sắp xếp.

5.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Với H là dãy số giảm dần của khoảng cách nghĩa là H[i] là khoảng cách giữa 2 phần tử trong mảng con ở mỗi lần chạy, và H[i] không nhỏ hơn 2.
  - Bước 1: Phân hoạch mảng đầu vào thành các mảng con, vị trí (index) của các phần tử trong mảng con cách nhau đúng bằng H[i]..
  - Bước 2: Áp dụng thuật toán insertion sort cho từng mảng con, cho đến khi H[i] < 2 thì dừng.
  - Bước 3: Áp dụng một lần thuật toán insertion sort cho toàn bộ mảng, ta thu được mảng sắp xếp theo yêu cầu.

- **Ví dụ minh họa:**

Sắp xếp mảng A = [7, 9, 2, 4, 5, 7, 8] thành mảng tăng dần.

- Chọn mảng khoảng cách H = {n/2, n/4, n/8,....,1}
- Bước 1: Chọn h = 3. Ta thu được các dãy con là A1 = [7, 4], A2 = [9, 5], A3 = [2, 7, 8]
- Bước 2: Sắp xếp các mảng con lại, ta được A1 = [4, 7], A2 = [5, 9], A3 = [2, 7, 8]. Vậy mảng A của ta sẽ là [4, 5, 2, 7, 9, 7, 8]
- Bước 3: Tiếp tục với h = 6. Ta thu được các dãy con là A1 = [7, 8]

5.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Thời gian sắp xếp nhanh hơn so với Insertion Sort do có thêm cơ chế chia nhỏ mảng rồi sắp xếp.
  - Áp dụng tốt cho mảng với kích thước trung bình nhưng không tốn thêm bộ nhớ.
- **Nhược điểm:**
  - Phụ thuộc vào cách chọn dãy khoảng cách.
  - Trừu tượng vì không có mô hình thực tế để hình dung.

5.4. Đánh giá độ phức tạp:

- Độ phức tạp về thời gian:

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

- Độ phức tạp về không gian: $O(n)$

Các cải tiến của Shell Sort đều dựa vào cách chọn dãy khoảng cách H, như trong bài viết được công bố thì tác giả sử dụng dãy là $\left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{4} \right\rfloor, \ldots, 1$; trong khi nhiều học giả khác như Pratt, Knuth,... sử dụng các dãy phức tạp hơn.

Có thể xem Shell Sort là một cải tiến của Selection Sort, kết hợp thêm các yếu tố chia nhỏ mảng của Merge Sort (cũng như là ý tưởng chia để trị).

6. Heap Sort:

Heap Sort là một thuật toán dựa trên các phép so sánh. Có thể xem nó như một phiên bản cải tiến của Selection Sort. Heap Sort được phát minh bởi J. W. J. Williams năm 1964, cũng là thời điểm ra đời của Heap.

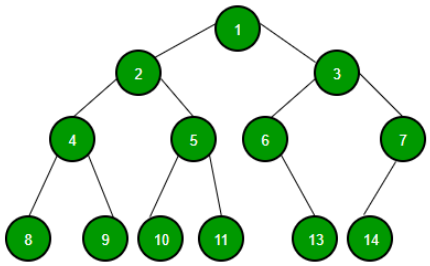
6.1. Ý tưởng thuật toán:

Thuật toán Heap Sort được chia làm hai phần là *xây dựng Max Heap* và *sắp xếp*.

- **Xây dựng max heap:**

Một số khái niệm:

- *Heap* là cây nhị phân hoàn chỉnh.
- *Cây nhị phân hoàn chỉnh* là cấu trúc cây nhị phân hoàn hảo nhưng trừ mức cuối, ở mức cuối các node lá dồn hết qua trái càng xa càng tốt.



- *Max Heap* là một heap sao cho các node đều lớn hơn các node con của nó. Để xây dựng Max Heap, ta sẽ *vun đống* từ giữa mảng. Ta không cần phải đi xây dựng một cấu trúc cây nhị phân trừu tượng mà sẽ thao tác trực tiếp trên mảng dữ liệu đầu vào.
- *Vun đống* (Heapify) là một thao tác kiểm tra xem node đang xét có giá trị lớn hơn giá trị của node con nó hay chưa. Chúng ta sẽ hoán vị một phần tử với node con của nó nếu giá trị của node đó bé hơn node con. Và nếu cả hai node con đều lớn hơn, ta sẽ hoán vị với node con lớn nhất.
- Hai phần tử con của một phần tử **a[i]** bất kỳ sẽ nằm ở vị trí **2i + 1** và **2i + 2** (nếu phần tử đầu là 0) hoặc **2i** và **2i + 1** (nếu phần tử đầu là 1). Các phần tử này gọi là các *phần tử liên đới*. Các số đánh trong hình trên chính là vị trí index của các phần tử khi thể hiện dưới dạng mảng. Nếu duyệt cây theo các mức (Level Order Traversal), ta sẽ thu được một danh sách tăng dần các vị trí index từ 1 đến n (hoặc từ 0 đến n – 1).

- **Sắp xếp:**

Sau khi xây dựng được Max Heap, ta sẽ lặp lại quá trình sắp xếp n lần như sau:

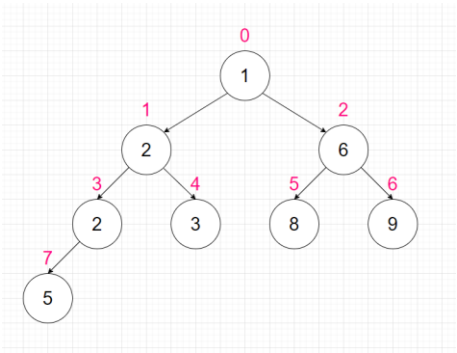
- Hoán vị phần tử đầu với phần tử cuối mảng, sau đó loại bỏ phần tử cuối mảng ra khỏi phạm vi vun đống. Vị trí cuối mảng xem như đã được sắp xếp.
- Tiến hành vun đống cho vị trí đầu tiên trong mảng, do các vị trí còn lại đều đã được vun đống.
- Lặp lại quá trình này khi chỉ còn một phần tử trong mảng hay nói cách khác là tất cả các phần tử đều đã chuyển đến cuối mảng và được sắp xếp.

6.2. Thiết lập thuật toán:

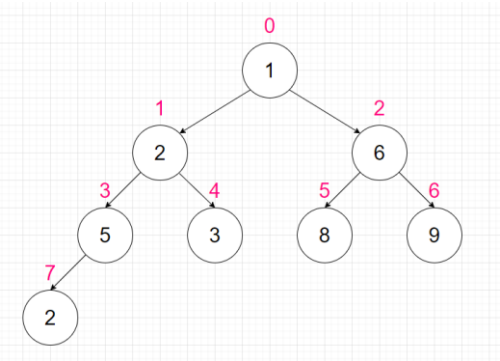
- **Các bước thiết lập thuật toán:**
  - Xét vun đống Max Heap tại vị trí giữa mảng. Lý do chọn vị trí này vì các vị trí sau sẽ không tồn tại bất kỳ node con nào.
  - Khi giá trị của node con lớn hơn node đang xét, hoán vị chúng với nhau. Nếu hai node con đều lớn hơn, chọn node con lớn nhất.
  - Khi xảy ra hoán vị, cần xét vun đống tại vị trí được hoán vị tới của node đang xét. Ví dụ hoán vị giá trị của node tại vị trí **i** với vị trí **j**, thì ta cần phải xét vun đống tại vị trí **j**.
  - Lặp lại quá trình xét cho đến phần tử đầu mảng.
- **Ví dụ minh họa:**
  - Ví dụ cho một mảng như sau:

$$1, 2, 6, 2, 3, 8, 9, 5$$

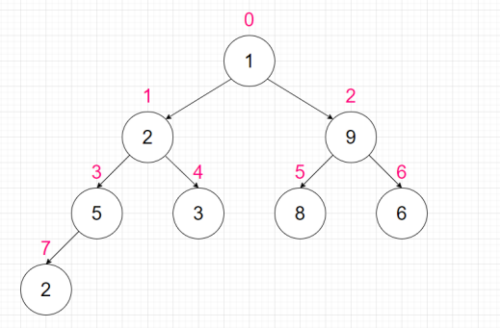
  - Cây nhị phân xây dựng từ mảng này là:



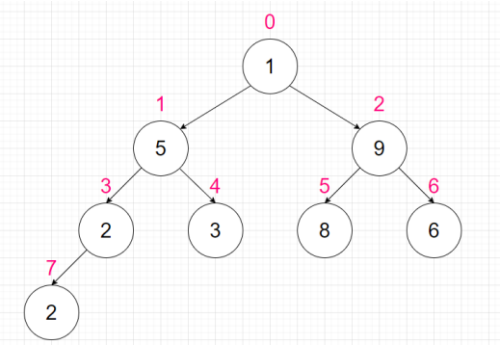
- Để tìm vị trí của phần tử giữa mảng, ta lấy **n/2–1**, với mảng ở trên ta có 8/2 – 1 = 3.
- Vậy ta sẽ xét vun đống ở phần tử **a[3]**, có giá trị là 2.
- Ta hoán vị **a[3]** = 2 với **a[7]** = 5.
- Các phần tử bị hoán vị phải được vun đống ở vị trí mà nó hoán vị đến. Việc này để đảm bảo các node luôn có giá trị lớn hơn node con của nó sau khi có sự thay đổi gây ra bởi công việc hoán vị. Cụ thể, với ví dụ trên ta sẽ xét vun đống tại **a[7]**, nhưng không có gì xảy ra vì nốt này không có node con.



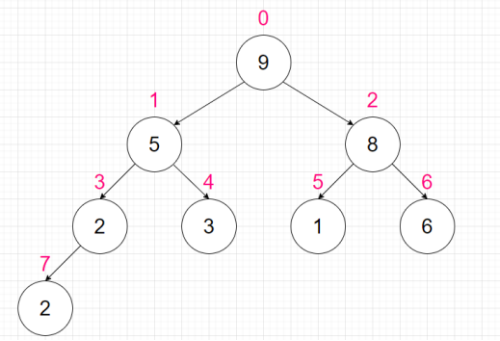
- Sau khi đã xét **a[3]** thì ta tiếp tục xét đến **a[2]**, **a[1]** và **a[0]** (tiến dần về đầu mảng). Khi xét tại **a[2]** = 6, nhận thấy có **a[6]** = 9 lớn hơn, nên ta sẽ hoán vị hai node này. Kết quả là:



- Tương tự cũng phải xét vun đống tại **a[6]**, nhưng vẫn không có gì xảy ra vì tại vị trí này không có node con.



  - Xét **a[1]** = 2 và hoán vị với **a[3]** = 5. Rồi xét vun đống tại **a[3]** nhưng không xảy ra hoán vị vì **a[3]** = **a[7]** = 2.
- Cuối cùng ta xét **a[0]** = 1, hoán vị với **a[2]** = 9 rồi hoán vị tiếp tục với **a[5]** = 8. Ta thu được một Max Heap hoàn chỉnh:



  - Nhận thấy tại mỗi node, giá trị của nó luôn lớn hơn giá trị của các node con.

6.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Một thuật toán Inplace không dùng thêm bộ nhớ phụ.
  - Có độ phức tạp ổn định, sẽ tốt hơn Worst Case của Quick Sort.
  - Nếu dữ liệu tổ chức trên Heap, đây là một thuật toán sử dụng hiệu quả.
  - Có thể sử dụng để giải quyết vấn đề top k phần tử như Selection Sort.
- **Nhược điểm:**
  - Là một thuật toán Non – Stable, thứ tự của các phần tử cùng lớp giá trị bị xáo trộn.
  - Chậm hơn rất nhiều so với Quick Sort hay Merge Sort khi dùng trong thực tế (hoặc khi dữ liệu quá lớn).
  - Khó cài đặt code.

6.4. Đánh giá độ phức tạp:

Phần vun đống, mỗi khi muốn xét một phần tử nào đó xem có thỏa mãn tính chất của Max Heap hay chưa, ta có thể phải đào sâu xuống các node con bên dưới để kiểm tra và vun đống nếu cần thiết. Mà chiều cao của cây nhị phân hoàn chỉnh với n node là $O(\log_2 n)$. Trường hợp xấu nhất khi phải đào đến các lá thì độ phức tạp của phần này $O(\log_2 n)$.

Trong phần xây dựng Max Heap, ta chỉ xét từ giữa mảng, tức là chỉ duyệt qua $n/2$ phần tử. Mỗi phần tử đều phải vun đống và có độ phức tạp thuật toán là $O(\log_2 n)$. như đã tìm ra ở trên. Vì vậy tổng độ phức tạp của việc xây dựng Max Heap là:

$$O(\frac{n}{2} * log_2 n) \approx O(n.log_2 n)$$

Đối với phần sắp xếp, cần phải sử dụng một vòng lặp duyệt qua n phần tử trong mảng để có thể chuyển nó về cuối và vun đống ngay sau đó. Mỗi lần như vậy cần tốn chi phí $O(\log_2 n)$ để vun đống. Từ đó suy ra phần này có độ phức tạp $O(n \log_2 n)$.

Độ phức tạp từng phần:

- Heapify: $O(\log_2 n)$.
- Xây dựng Max Heap: $O(n \log_2 n)$.
- Sắp xếp kèm heapify: $O(n \log_2 n)$.

Độ phức tạp không gian: $O(n)$.

Tổng kết lại:

Best case	Average case	Worst case
$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

6.5. Các biến thể của Heap Sort:

- **Thuật toán Floyd:**

Thuật toán này dùng để xây dựng một Max Heap nhưng lại chỉ có độ phức tạp là $O(n)$ thay vì thông thường là $O(n \log_2 n)$.

Ý tưởng:

- Duyệt Heap từ cuối mảng lên đầu mảng.

- Khi nào gặp một phần tử không thỏa mãn tính chất Max Heap (hoặc Min Heap) thì sẽ hoán đổi giá trị với các node con của nó.
- Tiếp tục xét cho các node con vừa mới bị hoán đổi. Ta gọi bước hai và ba là sàng lọc (percolate).

- **Bottom - up:**

Đây là một phiên bản khác của Heap Sort với phần vun đống (heapify) được tinh chỉnh để giảm thiểu số lần so sánh.

Ý tưởng:

- So sánh hai node con của node đang xét vun đống với nhau và lưu lại node con có giá trị lớn hơn, tạm gọi là node A. Tiếp tục đào xuống để so sánh hai node con của node A và tìm ra node con lớn hơn.
- Cứ như thế cho đến khi đào đến node lá nào đó. Nếu node lá đó có anh em, thì cũng phải so sánh để tìm ra node con lớn nhất, tạm gọi là B. Khi đã có node B này, ta xét từ node gốc (node đang xét vun đống) trở xuống node lá và tìm node đầu tiên lớn hơn node gốc.
- Sao chép node gốc đến node cuối cùng lớn hơn node gốc. Sau đó di chuyển từng phần tử lên một vị trí về phía node gốc.

7. Merge Sort:

Là một thuật toán dựa trên sự so sánh và phương pháp chia để trị do John Von Neumann phát minh ra 1945.

7.1. Ý tưởng thuật toán:

Thuật toán Merge Sort là một thuật toán ứng dụng phương pháp chia để trị, thuật toán này gồm hai phần: *chia mảng* và *trộn mảng*.

- **Chia mảng**

Phần đầu tiên, chia các mảng thành hai không gian con, nếu các không gian con này có nhiều hơn một phần tử thì tiếp tục chia đôi. Ngược lại có duy nhất một phần tử hoặc không có phần tử nào (trong trường hợp dãy lẻ) thì bắt đầu trộn lại (gọi đệ qui).

- **Trộn mảng:**

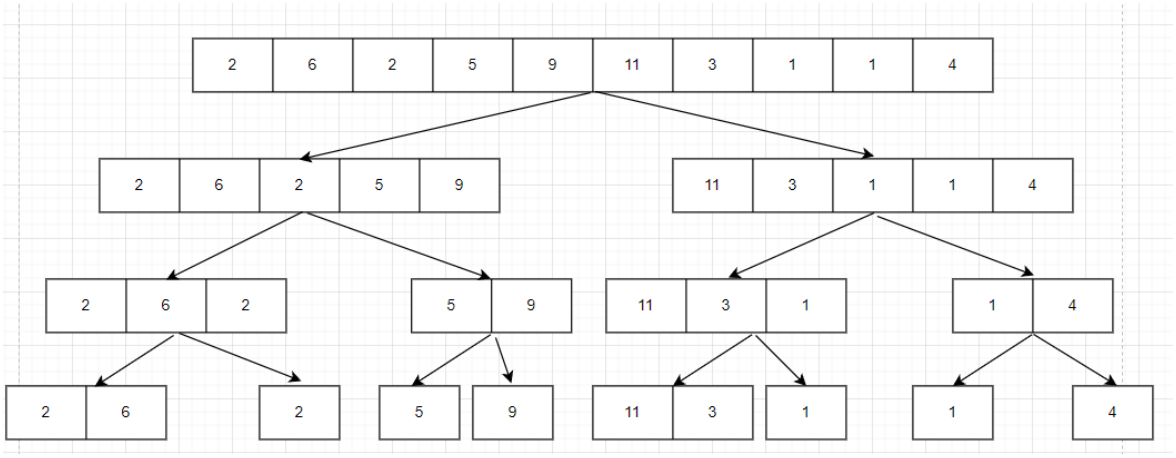
Phần thứ hai chính là quá trình trộn, trong quá trình trộn sẽ kết hợp sắp xếp mảng.

Trộn 2 mảng con được thực hiện như sau:

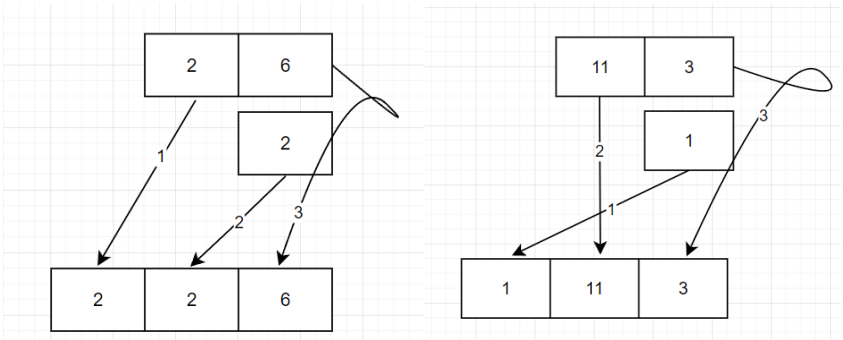
- Chọn phần tử min ở vị trí đầu của một trong hai mảng, xếp vào mảng cần trộn.
- Phần tử nào đã xếp vào thì xóa đi, vị trí đầu của mảng là phần tử tiếp theo.
- Nếu chưa đến cuối mảng thì lặp lại bước 1. Nếu đã đến cuối của một mảng (luôn xảy ra một mảng đã sắp hết và một mảng thì chưa), thì thêm toàn bộ mảng kia vào mảng cần sắp.

7.2. Thiết lập thuật toán:

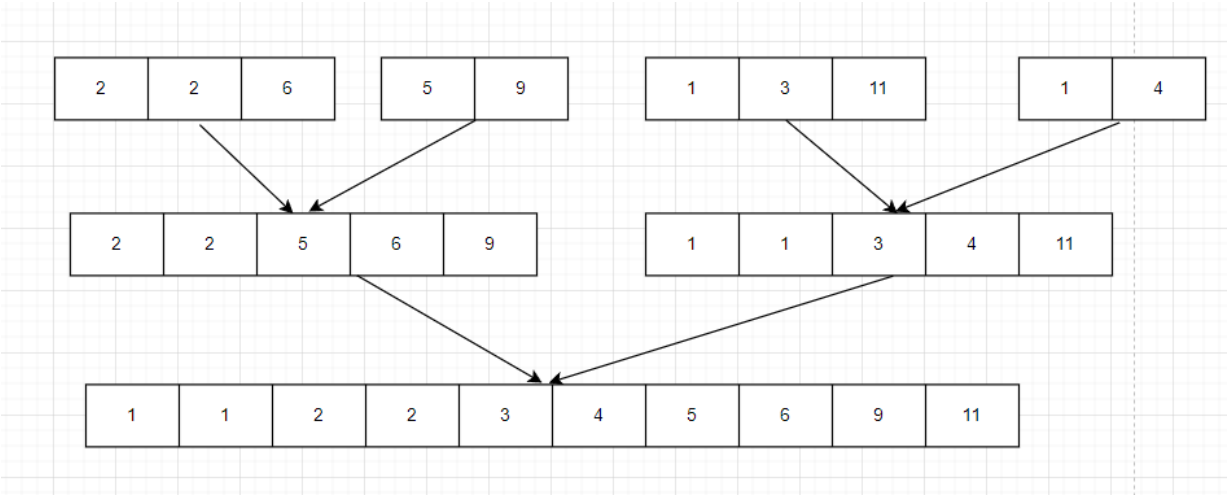
- Xét ví dụ cho mảng dưới đây:



- Ví dụ xét vài mảng con đầu tiên:



- Thứ tự của việc trộn được đánh số như trên hình.
- Tương tự cho đến khi trộn hết tất cả các mảng con:



7.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Khi hai phần tử có cùng giá trị, phần tử nào có chỉ số index nhỏ hơn sẽ đứng trước. Điều này làm thuật toán trở nên Stable trong một số phiên bản.
  - Độ phức tạp ổn định.
  - Nhanh hơn Quick Sort khi dữ liệu đầu vào lớn.
- **Nhược điểm:**
  - Khó cài đặt code.
  - Chậm khi dữ liệu gần như được sắp xếp.

7.4. Đánh giá độ phức tạp:

- Ta có 2^x là lũy thừa của cơ số 2. Với x là số lần nhân tích lũy các số 2 với nhau. Nếu một mảng có n phần tử và ta cho nó bằng 2^x, thì:

$$2^x = n$$

- Nói cách khác, x là số lần nhân đôi số lượng node hiện có để đạt được n node.
- Nếu chúng ta làm ngược lại, tìm số lần để chia một mảng n node thành 1 node đơn lẻ, ta sẽ lấy logarith của hai vế theo cơ số 2:

$$log_2(2^x) = log_2(n)$$

- Như vậy, số lần cần thiết để chia nhỏ mảng ra đến khi không chia được nữa là $\log_2 n$.
- Từ đó, phần chia mảng trong thuật toán Merge Sort có độ phức tạp là $O(\log_2 n)$.
- Ngoài ra, khi trộn mảng, ta phải duyệt qua từng phần tử của hai mảng con bất kỳ. Như vậy độ phức tạp phần trộn là $O(n + m)$, với n và m là kích thước của hai mảng con.
- Bên cạnh đó, không gian phụ mà thuật toán sử dụng là rất lớn phụ thuộc tuyến tính vào số phần tử đầu vào, nên độ phức tạp không gian là $O(n)$. Nếu như dùng Merge Sort trên danh sách liên kết thì sẽ không cần dùng thêm không gian phụ.
- Tuy nhiên không gian bộ nhớ cần dùng cho các lời gọi đệ quy lưu trong Stack ở cả trường hợp dùng mảng và danh sách liên kết đều là $O(\log_2 n)$. Nên không gian tối thiểu mà Merge Sort sử dụng là $O(\log_2 n)$. Chỉ là trong trường hợp dùng Merge

Sort trên mảng, không gian phụ là $O(n)$ áp đảo không gian $O(\log_2 n)$. cho việc lưu Stack khi n tiến về một số rất lớn, nên trường hợp này sẽ là $O(n)$.

Tổng kết lại:

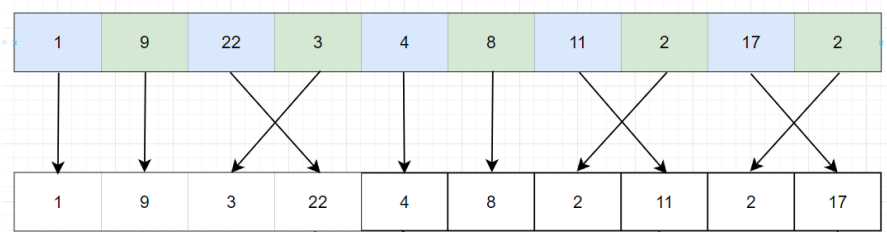
Best case	Average case	Worst case
$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

7.5. Biến thể của Merge Sort:

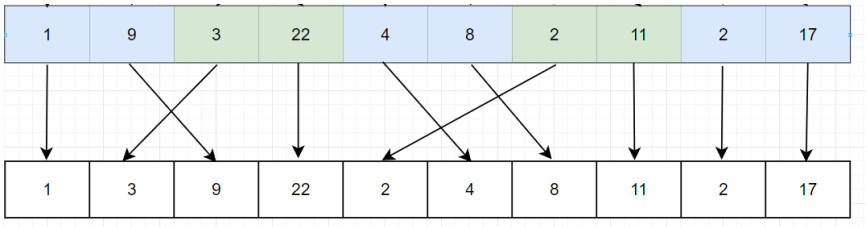
- **Bottom – up:**

Ý tưởng của thuật toán này là thay vì chia mảng ra nhiều mảng con, thì lại xem xét mảng dưới góc độ là chứa nhiều mảng con. Rồi tiến hành trộn các mảng con liền kề lại với nhau.

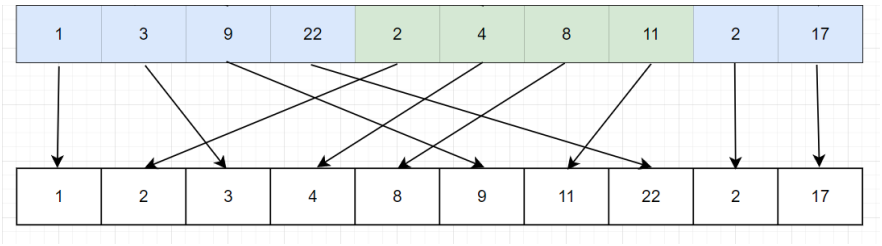
- Xét ví dụ sau đây, ta tiến hành trộn hai các mảng con kích thước là 1 lại với nhau.



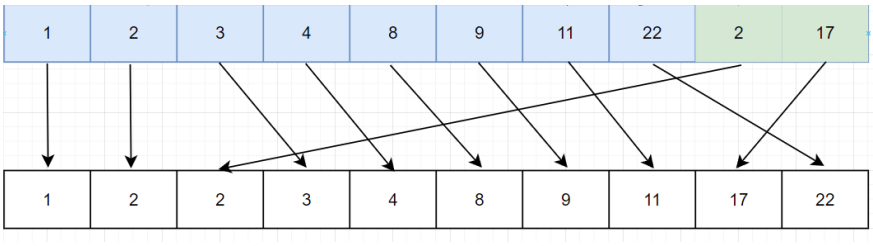
- Sau đó trộn các mảng con kích thước là 2.



- Trộn các mảng con có kích thước là 4.



- Và cuối cùng là trộn các mảng có kích thước là 8 lại với nhau và hoàn thành thuật toán.



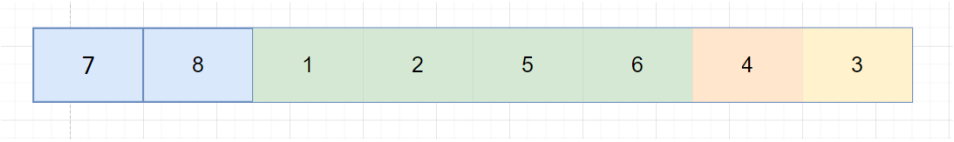
- Chú ý rằng thứ tự của hai số 2 vẫn giữ nguyên, do đó phiên bản này của Merge Sort vẫn là stable.

- **Natural Merge Sort**

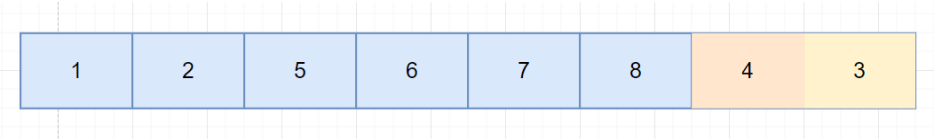
Bởi vì Merge Sort thông thường không nhận biết được một dãy đã sắp xếp nên trong thực tế người ta không dùng Merge Sort thuần để ứng dụng. Thay vào đó họ sử dụng Natural Merge Sort tận dụng các đường chạy để tối ưu thuật toán.

Đường chạy là một dãy tăng không giảm. Ví dụ dãy 7 8 1 2 5 6 4 3 sẽ có bốn đường chạy là (7, 8), (1, 2, 5, 6), (4) và (3). Khi chọn ra được các đường chạy trong cùng một mảng, chúng ta sẽ tiến hành trộn tuần tự các đường chạy đó lại với nhau.

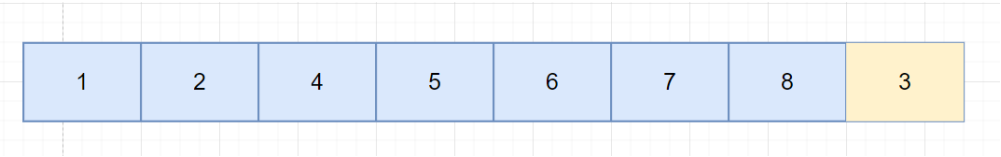
- Ví dụ sắp xếp cho dãy trên, ta có:



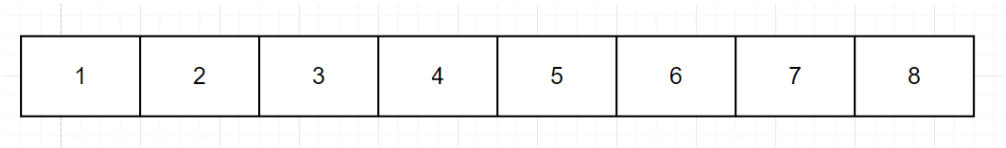
- Trộn hai đường chạy đầu tiên lại với nhau:



- Tiếp tục trộn:



- Trộn đường chạy cuối cùng, ta được kết quả là dãy đã sắp xếp.



Thay vì phân ra quá nhiều mảng con, phiên bản này của Merge Sort chỉ phân ra k đường chạy nhất định rồi trộn chúng lại với nhau.

8. Quick Sort:

8.1. Ý tưởng thuật toán:

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

- Chọn phần tử đầu tiên của mảng.
- Chọn phần tử cuối cùng của mảng.
- Chọn một phần tử random.
- Chọn một phần tử có giá trị nằm giữa mảng.

8.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Bước 1: Chọn một phần tử nằm giữa mảng làm pivot.
  - Bước 2: Đưa những phần tử nhỏ hơn pivot sang mảng bên trái, những phần tử lớn hơn pivot sang mảng bên phải.
  - Bước 3: Thực hiện lại 2 bước trên đối với mảng trái và mảng phải đến khi tất cả các mảng còn đều có một phần tử.
- **Ví dụ minh họa:**

8.3. Đánh giá thuật toán:

- Ưu điểm:
  - Thuật toán có độ phức tạp nhỏ hơn các thuật toán sắp xếp đơn giản, tốc độ xử lý tương đối nhanh.
  - Có thể ứng dụng vào xử lý dữ liệu lớn .
- Nhược điểm:
  - Thuật toán không có tính ổn định, không có tính thích ứng, bị ảnh hưởng bởi dữ liệu đầu vào.
  - Tốn không gian bộ nhớ hơn so với các thuật toán sắp xếp đơn giản.
  - Khó cài đặt thuật toán.
  - Khó khăn trong việc lựa chọn phần tử làm chốt trong phân hoạch. Việc lựa chọn có thể ảnh hưởng rất nhiều đến hiệu suất của thuật toán tuy nhiên ta không thể đưa ra lựa chọn tối ưu nhất.

8.4. Đánh giá độ phức tạp:

Do hiệu quả của thuật toán phụ thuộc rất nhiều vào việc phần tử nào được chọn là phần tử chốt (pivot) nên độ phức tạp thuật toán cũng phụ thuộc vào đó mà khác nhau.

- Phân đoạn không cân bằng: không có phần nào cả, một bài toán con có kích thước $n-1$ và bài toán kia có kích thước là 0. Đó là trường hợp xấu nhất xảy ra khi dãy đã cho là dãy đã được sắp xếp và phần tử chốt được chọn là phần tử đầu của dãy. Suy ra, độ phức tạp thuật toán sẽ là $O(n^2)$.
- Phân đoạn hoàn hảo: phân đoạn luôn thực hiện dưới dạng phân thì đôi, mỗi bài toán con có kích thước là $n/2$. Suy ra, độ phức tạp thuật toán là $O(n \log n)$.

Tổng kết lại:

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

9. Counting Sort:

9.1. Ý tưởng thuật toán:

Với mỗi số trong mảng đầu vào, ta xác định được số phần tử trong mảng nhỏ hơn nó, từ đó ta đặt nó vào vị trí thích hợp trong mảng đầu ra.

9.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**

- Mảng A gồm n phần tử đầu vào.
- Mảng B gồm n phần tử đã được xếp tăng dần.
- Mảng C gồm k phần tử từ 0 đến k là mảng tạm trong quá trình sắp xếp, với k là giá trị của phần tử lớn nhất trong mảng, mọi phần tử trong C mặc định bằng 0. Mảng C đến cuối cùng định vị vị trí (index) của các phần tử trong A ở trong mảng B để sắp xếp
  - Bước 1: Duyệt mảng A, với mỗi phần tử h trong mảng A, tăng giá trị C[h] thêm 1.
  - Bước 2: Với i từ 1 đến k, C[i] = C[i] + C[i-1] (đếm số phần tử nhỏ hơn hoặc bằng)
  - Bước 3: Duyệt mảng A, với mỗi phần tử h trong A, vị trí (index) của h trong mảng B chính là C[h] - 1. Hay nói cách khác, B[C[h] - 1] = h và giảm C[h] xuống 1. Sau khi kết thúc, ta thu được mảng được sắp xếp tăng dần.

- **Ví dụ minh họa:**

Sắp xếp mảng A = [0, 8, 7, 5, 6, 8, 4, 3, 10] theo thứ tự tăng dần

Khai báo mảng C[0,..10] và mảng B[0, 8]

Bước 1:

Ta thu được mảng C như sau:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	1	0	0	1	1	1	1	1	2	0	1

Bước 2:

Cộng dồn value ở các vị trí từ 1 đến 10, giữ nguyên vị trí 0, ta thu được mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	1	1	1	2	3	4	5	6	8	8	9

Bước 3: Xếp vào mảng B lần lượt theo nguyên tắc gán sau: B[C[i] - 1] = A[i] và giảm C[i] xuống 1.

- **Lần 1:**

i = 0, A[i] = 0, C[A[i]] = 1 => C[A[i]] – 1 = 0 và B[C[A[i]] – 1] = 0

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0								

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	3	4	5	6	8	8	9

- **Lần 2:**

i = 1, A[i] = 8, C[A[i]] = C[8] = 8 => C[A[i]] – 1 = 7 và B[C[A[i]] – 1] = B[7] = 8

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0							8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	3	4	5	6	7	8	9

- **Lần 3:**

i = 2, A[i] = 7, C[A[i]] = C[7] = 6 => C[A[i]] – 1 = 5 và B[C[A[i]] – 1] = B[5] = 7

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0					7		8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	3	4	5	5	7	8	9

- **Lần 4:**

i = 3, A[i] = 5, C[A[i]] = C[5] = 4 => C[A[i]] – 1 = 3 và B[C[A[i]] – 1] = B[3] = 5

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0			5		7		8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	3	3	5	5	7	8	9

- **Lần 5:**

i = 4, A[i] = 6, C[A[i]] = C[6] = 5 => C[A[i]] – 1 = 4 và B[C[A[i]] – 1] = B[4] = 6

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0			5	6	7		8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	3	3	4	5	7	8	9

- **Lần 6:**

i = 5, A[i] = 8, C[A[i]] = C[8] = 7 => C[A[i]] – 1 = 6 và B[C[A[i]] – 1] = B[6] = 8

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0			5	6	7	8	8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	3	3	4	5	6	8	9

- **Lần 7:**

i = 6, A[i] = 4, C[A[i]] = C[4] = 3 => C[A[i]] – 1 = 2 và B[C[A[i]] – 1] = B[2] = 4

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0		4	5	6	7	8	8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	2	2	3	4	5	6	8	9

- **Lần 8:**

i = 7, A[i] = 3, C[A[i]] = C[3] = 2 => C[A[i]] – 1 = 1 và B[C[A[i]] – 1] = B[1] = 3

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0	3	4	5	6	7	8	8	

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	1	2	3	4	5	6	8	9

- **Lần 9:**

i = 8, A[i] = 10, C[A[i]] = C[10] = 9 => C[A[i]] – 1 = 8 và B[C[A[i]] – 1] = B[8] = 10

Mảng B:

Index	0	1	2	3	4	5	6	7	8
Value	0	3	4	5	6	7	8	8	10

Mảng C:

Index	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	1	2	3	4	5	6	8	8

9.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Độ phức tạp tuyến tính nên việc thực hiện sẽ nhanh hơn
- **Nhược điểm:**
  - Chỉ xử lý được các kiểu dữ liệu hữu hạn, đếm được, như số nguyên, khó xử lý với số phức.
  - Nếu mảng có số âm, cần có sự điều chỉnh trong thuật toán.
  - Nếu giá trị lớn nhất của mảng rất lớn so với số lượng phần tử trong mảng, độ phức tạp sẽ bị tăng cao.

9.4. Đánh giá độ phức tạp:

- Độ phức tạp thời gian với n là số lượng phần tử mảng và k là phần tử lớn nhất có trong mảng

Best case	Average case	Worst case
$O(n)$	$O(n)$	$O(k+n)$

- Độ phức tạp không gian: $O(n)$ hoặc $O(k)$ tùy thuộc vào số nào lớn hơn.

10. Radix Sort:

Radix sort đã có từ thời điểm sử dụng máy sắp xếp thẻ đục lỗ (card-sorting machine) trước khi máy tính điện tử hiện đại ra đời.

10.1. Ý tưởng thuật toán:

Xét từng chữ số của các phần tử trong mảng, bắt đầu từ chữ số phải cùng rồi phân phối vào các hộp đựng được đánh dấu từ 0 đến n – 1 với n là cơ số, đến khi không còn chữ số nào nữa.

10.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Bước 1: Xét chữ số phải cùng (hàng đơn vị) của tất cả các số có trong mảng.
  - Bước 2: Xếp những số có cùng chữ số thành các nhóm.
  - Bước 3: Theo thứ tự từ chữ số nhỏ nhất đến lớn nhất, đưa các số trong mỗi nhóm vào mảng.
  - Bước 4: Xét chữ số tiếp theo nằm liền kề bên trái của chữ số đang xét và thực hiện lại bước 2 cho đến khi số có nhiều chữ số nhất được xét hết thì ngừng, ta thu được mảng được sắp xếp tăng dần.
- **Ví dụ minh họa:**

Sắp xếp mảng A = [789, 1010, 34, 2, 56, 0, 93, 82, 75] theo thứ tự tăng dần.

Lần chạy đầu tiên:

- Bước 1: Xét chữ số phải cùng của từng phần tử trong mảng A. Nghĩa là [9, 0, 4, 2, 6, 0, 3, 2, 5]. Gom thành nhóm như sau:
  - Nhóm 0: 1010, 0
  - Nhóm 2: 2, 82

    - Nhóm 3: 93
    - Nhóm 4: 34
    - Nhóm 5: 75
    - Nhóm 6: 56
    - Nhóm 9: 789
- Bước 2: Chọn các phần tử từ nhóm nhỏ nhất đến nhóm lớn nhất cho tuần tự vào mảng
    - A = [1010, 0, 2, 82, 93, 34, 75, 56, 789]

Lặp lại quy trình trên cho lần chạy thứ 2:

- Bước 1: Xét chữ số liền kề bên trái của từng phần tử trong mảng A, số nào không có thì coi như là 0. Nghĩa là [1, 0, 0, 8, 9, 3, 7, 5, 8]. Gom thành nhóm như sau:
    - Nhóm 0: 1010, 0, 2
    - Nhóm 3: 34
    - Nhóm 7: 75
    - Nhóm 5: 56
    - Nhóm 8: 82, 789
    - Nhóm 9: 93
- Bước 2: Chọn các phần tử từ nhóm nhỏ nhất đến nhóm lớn nhất cho tuần tự vào mảng
    - A = [1010, 0, 2, 82, 93, 34, 75, 56, 789]

10.3. Đánh giá thuật toán:

- **Ưu điểm:**
    - Thuật toán sắp xếp nhanh.
    - Ứng dụng trong mảng hậu tố (suffix array), có ý nghĩa cốt lõi trong nhiều loại thuật toán
- **Nhược điểm:**
    - Tốn không gian bộ nhớ.
    - Chỉ sử dụng với một số trường hợp đặc thù, không có độ đa dạng cao (không dùng được cho số thực, khó khăn với số âm)

Có nhiều cách để cài đặt Radix Sort sao cho tối ưu không gian, một trong số đó là sử dụng hàng đợi thay vì phải khai báo dư các ô cơ số để chứa các số.

10.4. Đánh giá độ phức tạp:

Best case	Average case	Worst case
$O(n)$	$O(n)$	$O(n)$

11. Flash Sort:

11.1. Ý tưởng thuật toán:

Thuật toán Flash Sort gồm ba công việc: *phân lớp*, *hoán vị* và *sắp xếp cục bộ*.

- Phân lớp giúp xác định kích thước của từng lớp phần tử.
- Hoán vị giúp hoán vị các phần tử giữa các lớp này.
- Sắp xếp cục bộ để sắp xếp các phần tử trong cùng một lớp.

Ý tưởng chung của Flash Sort là chia để trị, tương tự như *Merge Sort* hay *Quick Sort*. Thay vì chia thành hai mảng con đến khi nào không chia được nữa thì Flash Sort chia thành m phân lớp.

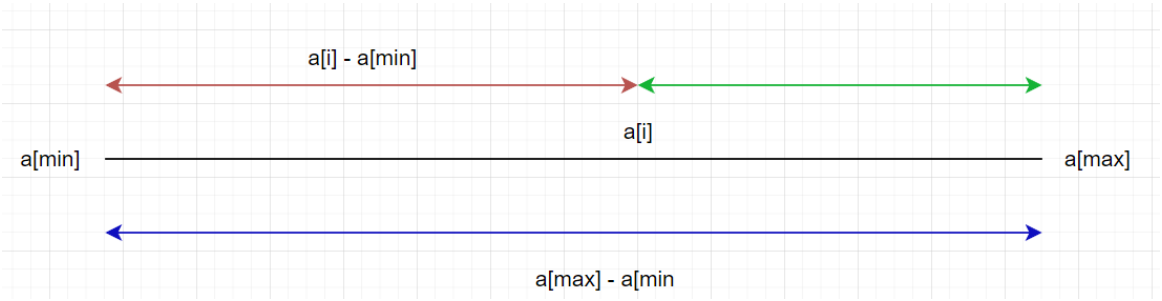
- **Phân lớp:**

Giả sử các phần tử trong danh sách là **a[i]** và được phân bố đều rải rác. Cần tính vị trí của phần tử **a[i]** trong một lớp dữ liệu. Việc tính toán này có thể tính dựa trên giá trị của phần tử đó mà không cần thực hiện các phép so sánh.

- Đầu tiên, ta cần đi tìm hệ số **m**, là số lượng các lớp phần tử. Ta có công thức sau, với **n** là số lượng phần tử:

$$m = C * n$$

- Tiếp theo, ta cần đi tìm giá trị lớn nhất (**a[max]**) và nhỏ nhất (**a[min]**) của danh sách các phần tử. Công việc này cần phải quét hết tất cả các phần tử nên độ phức tạp là O(n).



- Sau đó, ta lấy đoạn màu xanh dương trên hình trừ cho đoạn màu đỏ để ra đoạn màu xanh lá. Nói cách khác, đoạn màu xanh lá có biểu thức:

$$\frac{a[i] - a[min]}{a[max] - a[min]}$$

- Cuối cùng, ta nhân biểu thức này cho (**m – 1**) với **m** là số lượng các phân lớp, lý do trừ 1 vì các phân lớp bắt đầu tại 0.
- Ta suy ra được biểu thức tìm vị trí **k**:

$$k(a[i]) = \left\lfloor \frac{(m-1)(a[i] - a[min])}{a[max] - a[min]} \right\rfloor$$

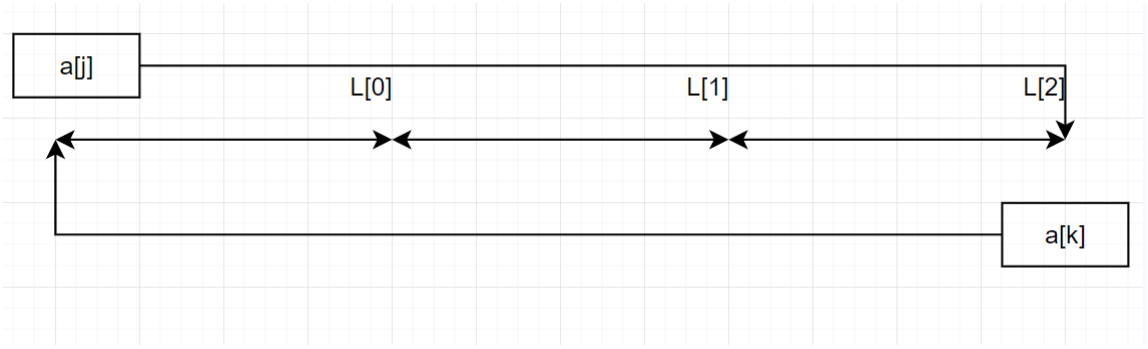
Kết quả sẽ có giá trị từ 0 đến **m**. Nếu muốn index bắt đầu là 1 thì ta chỉ cần cộng thêm 1 vào biểu thức trên. Trung bình sẽ có **n/m** phần tử ở mỗi lớp.

Khi có biểu thức trên, ta cần xét hết tất cả các phần tử và đếm xem mỗi lớp có bao nhiêu phần tử. Rồi chúng ta tính chỉ số index cuối cùng của mỗi lớp, việc này sẽ phục vụ cho công việc hoán vị tiếp theo. Chỉ số này sẽ được lưu trong mảng gồm **m** phần tử, là mảng **L** vốn trước đó lưu số phần tử của mỗi phân lớp. Mảng **L** này sẽ lưu giá trị cuối cùng của các phân lớp.

- **Hoán vị:**

Ý tưởng của công việc hoán vị là di chuyển một phần tử đang ở vị trí không đúng lớp đến nơi mà nó thuộc về.

- Vị trí có thể di chuyển của một lớp phụ thuộc vào mảng **L** ở trên, với **L[0]** là vị trí có thể di chuyển đến cho phần tử **a[j]** (Ta dùng index **j** trong phần hoán vị này) bất kỳ của lớp 0. Tương tự **L[1]** là lớp 1 và **L[k]** là của lớp **k**. Sau khi di chuyển đến, ta giảm giá trị **L[k]** xuống một giá trị để chuyển sang vị trí khác.
- Bất cứ khi nào di chuyển một phần tử **a[j]** đến lớp **m**, ta sẽ hoán vị một phần tử của lớp **m** đó cho phần tử **a[j]** cần di chuyển. Ví dụ dưới đây di chuyển **a[j]** đến lớp 2, ta đẩy **a[k]** sang vị trí trước đó của **a[j]**. Sau đó, giá trị của **L[2]** trừ đi một để chuyển đến vị trí dời chỗ tiếp theo. Việc tính toán xem phần tử **a[j]** thuộc lớp nào cũng phụ thuộc vào biểu thức tính k ở trên.
- Tiếp tục xét xem **a[k]** thuộc lớp nào và đổi chỗ với phần tử thuộc lớp đó.



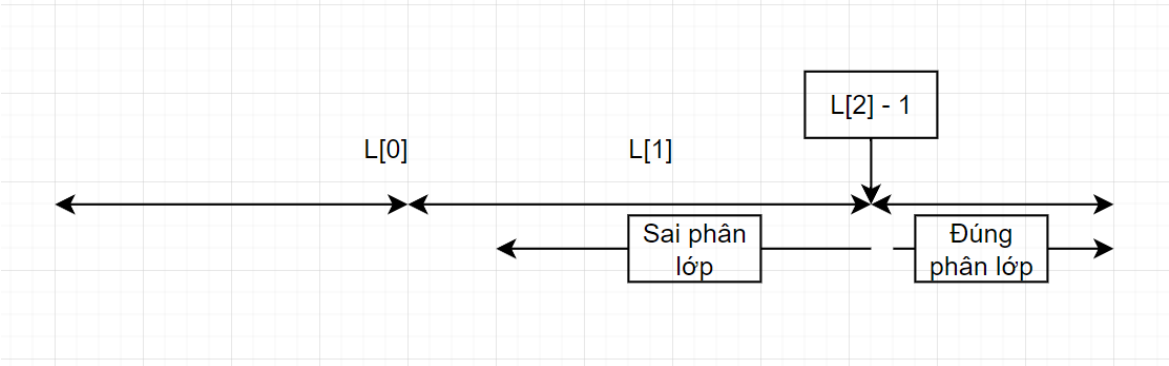
- **Sắp xếp cục bộ:**

Khi đã hoán vị các phần tử về đúng phân lớp của nó, ta sẽ tiến hành sắp xếp cục bộ trong từng phân lớp. Ở đây chúng ta sử dụng sắp xếp chèn (Insertion Sort), và độ phức tạp thuật toán của mỗi phân lớp là:

$$O\left(\left(\frac{n}{m}\right)^2\right)$$

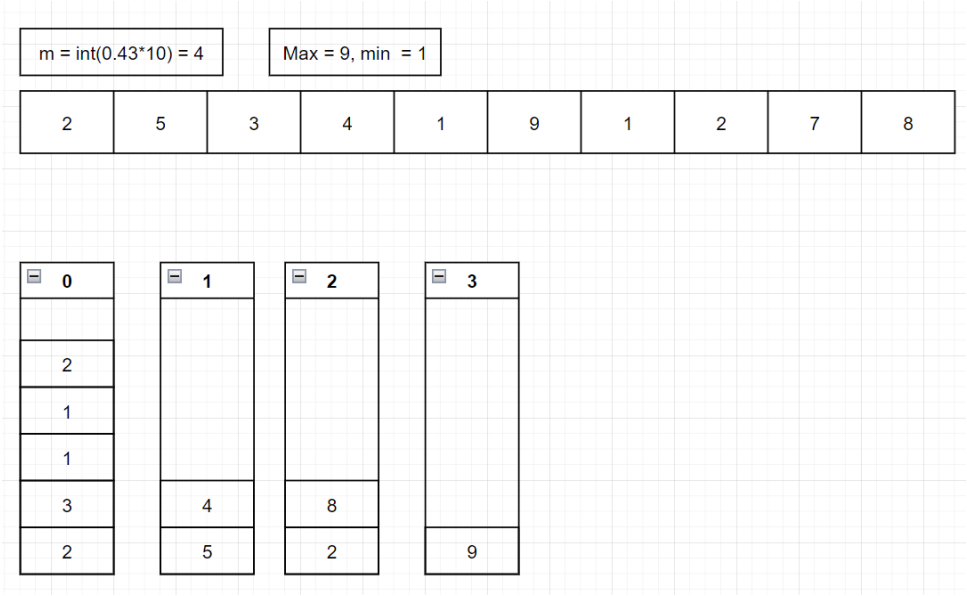
11.2. Thiết lập thuật toán:

- **Các bước thiết lập thuật toán:**
  - Khoảng vị trí của một phân lớp **k** là từ **L[k]** trở đi. Mỗi lần thêm một phần tử vào lớp **k** bất kỳ thì giá trị của **L[k]** giảm đi một. Khi lớp đó đã chứa đầy các phần tử thì giá trị của **L[k]** = **L[k – 1] + 1** (tức là vị trí sau vị trí cuối cùng của phân lớp trước).
  - Từ đó, ta suy ra được rằng với vị trí **j** bất kỳ lớn hơn **L[k] – 1** (**L[k]** trở đi) thì phần tử đó đã thuộc phân lớp của nó, do vậy ta bỏ qua và xét vị trí **j** tiếp theo. Ngược lại, phần tử có vị trí **j** bé hơn hoặc bằng **L[k] – 1** thì nó vẫn chưa nằm đúng phân lớp, ta cần tìm phân lớp của **a[j]** và chuyển nó đến đó.

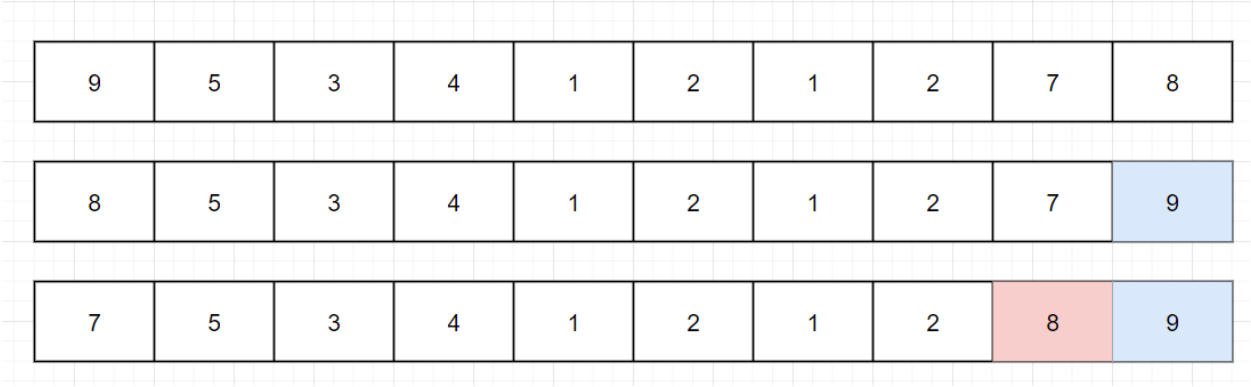


  - Khi đã tìm được vị trí **j** để bắt đầu một chu trình hoán vị đưa các phần tử về đúng phân lớp, ta sẽ lặp đến khi tìm được một vị trí mà ở đó **j** không thỏa vị trí mà phân lớp của nó chỉ định. Tức là **j** khác **L[k]**. Trong vòng lặp này ta sẽ thực hiện đổi chỗ và tăng biến đếm số lần hoán vị lên.
  - Khi số lần hoán vị là **n** thì ta dừng quá trình hoán vị và sang sắp xếp cục bộ. Đặc biệt, khi chỉ còn một phần tử chưa được hoán vị thì ta biết rằng nó đã nằm đúng vị trí, nên điều kiện dừng của bước hoán vị sẽ là **n – 1**.
- **Ví dụ minh họa:**

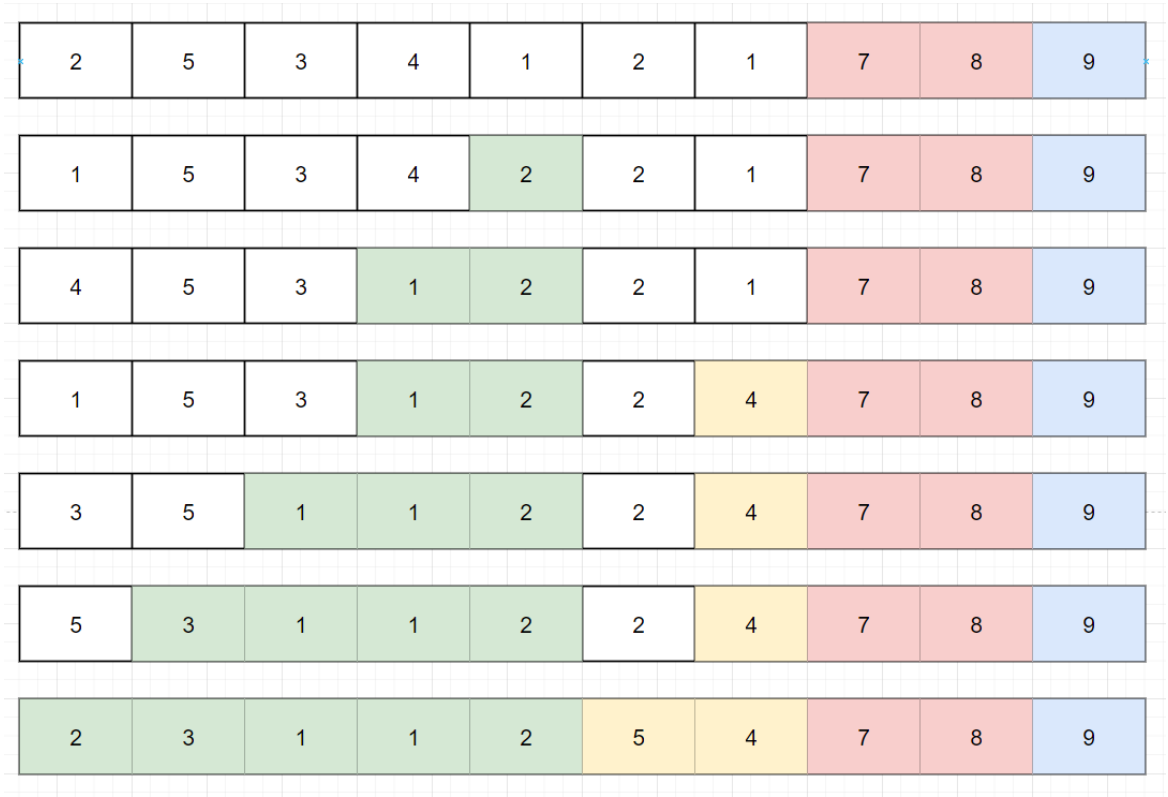
- Cho mảng dưới đây, các số đã được phân vào 4 lớp phù hợp.



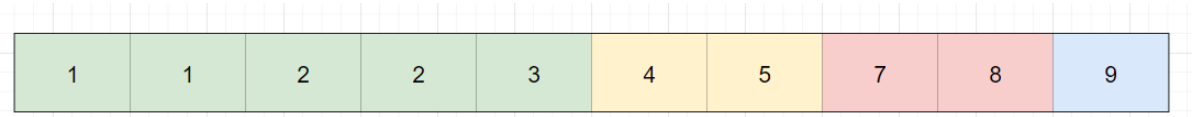
- Tiến hành hoán vị, ta thấy 9 thuộc lớp 3, ta đổi chỗ với 8. Sau đó 8 thuộc lớp 2, đổi chỗ với 7.



- Cứ như vậy đến khi hoán vị tất cả các phần tử về đúng lớp.



- Cuối cùng ta tiến hành Insertion Sort cho từng lớp để được kết quả.



11.3. Đánh giá thuật toán:

- **Ưu điểm:**
  - Có thể dùng để sắp xếp các số không phải số nguyên (non – integer). Để sắp xếp dữ liệu dạng string thì cần chuyển sang hệ cơ số 256.
  - Thời gian chạy thuật toán rất nhanh khi chọn m thích hợp.
- **Nhược điểm:**
  - Nếu dữ liệu không phân bố ngẫu nhiên đều khắp danh sách và phân bố tập trung thành một cụm (skewed), thì số phân lớp sẽ suy biến về $O(1)$. Dẫn đến độ phức tạp là $O(n^2)$.
  - Không phải là thuật toán stable.
  - Khó cài đặt code.

11.4. Đánh giá độ phức tạp:

Như đã phân tích sơ lược ở phần ý tưởng, độ phức tạp thuật toán của các bước là:

$$\begin{aligned} \text{Phân lớp} &= O(n) \\ \text{Hoán vị} &= O(n) \\ \text{Sắp xếp cục bộ} &= O\left(\left(\frac{n}{m}\right)^2\right) \end{aligned}$$

- Nếu số lượng phân lớp là tuyến tính theo $O(n)$, mỗi phân lớp sẽ có kích thước hằng số gần giống nhau, thì việc sắp xếp một phân lớp như vậy với thuật toán có độ phức tạp là $O(n^2)$.như Insertion Sort sẽ có độ phức tạp là $O(1^2) = O(1)$. Như vậy, độ phức tạp thời gian của thuật toán là $O(n)$.
- Độ phức tạp thời gian sẽ tăng lên nếu có ít không gian bộ nhớ được sử dụng và sẽ giảm đi nếu có nhiều không gian bộ nhớ được sử dụng. Nói cách khác, độ phức tạp thời gian và không gian là tỉ lệ nghịch. Mặc dù vậy, nếu dữ liệu đầu vào của thuật toán là một danh sách đã sắp xếp, thì thời gian thực hiện thuật toán cũng không giảm.
- Với $m = 0.1n$, Flash Sort luôn nhanh hơn Heap Sort và nhanh hơn Quick Sort nếu $n > 80$. Hơn thế nữa, với $n = 10000$, Flash Sort sẽ nhanh hơn Quick Sort xấp xỉ hai lần. Mặt khác, với $m = 0.2n$, Flash Sort nhanh hơn khi $m = 0.1n$ xấp xỉ 15%. Với $m = 0.05n$, nó chậm hơn 30% nhưng khi $n > 200$ thì nó vẫn nhanh hơn Quick Sort.
- Số phân lớp $m = O(n)$ chính là độ phức tạp không gian dùng trong thuật toán và độ phức tạp không gian này có thể sai khác phụ thuộc nhiều vào hằng số C được tính ở trên.

Best case	Average case	Worst case
$O(n)$	$O(n)$	$O(n^2)$

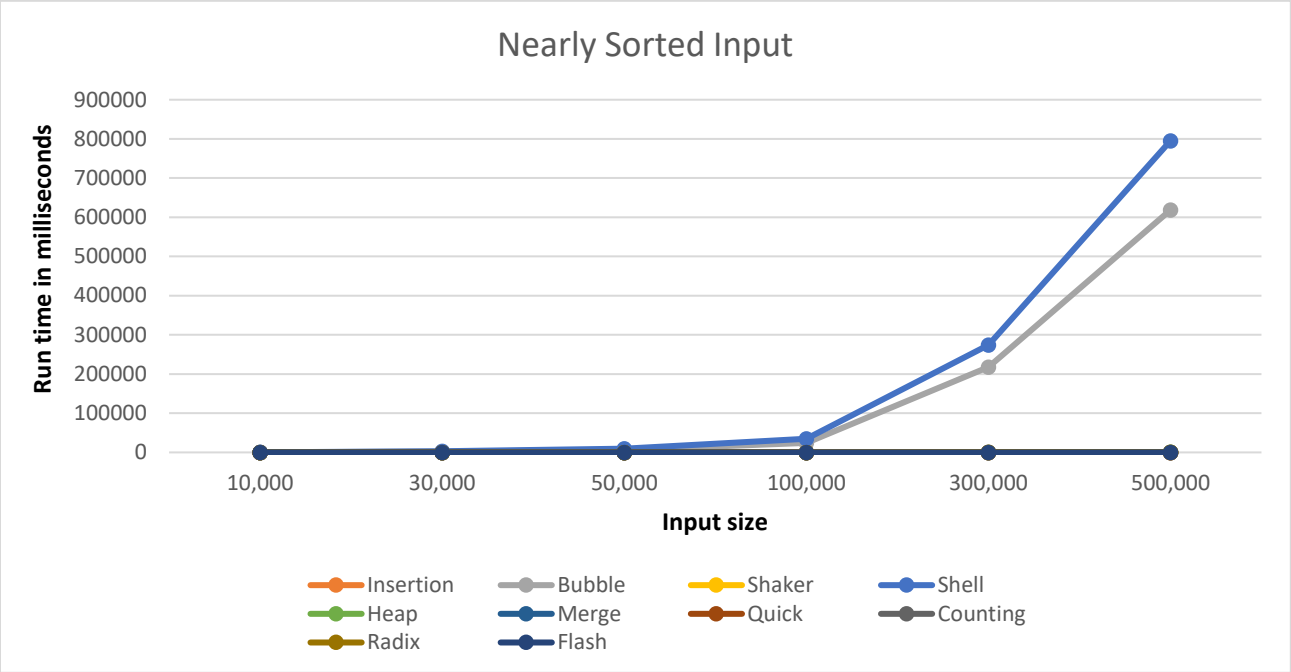
Phần 2: KẾT QUẢ THỰC NGHIỆM

1. Nearly Sorted Input:

1.1. Table:

Data order: Nearly Sorted Data						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	170.543	50005019	1563.69	450015019	4647.72	1250025019
Insertion	0	143858	0.997	380102	1.993	662794
Bubble	239.625	100009999	2181.75	900029999	5971.34	2500049999
Shaker	1.002	143241	2.001	374142	2	643748
Shell	281.259	73354915	2925.28	727686158	9153.83	2083575918
Heap	2.991	669741	10.971	2236668	17.979	3925356
Merge	7.98	489064	20.972	1594594	35.906	2752947
Quick	0.001	176933	0.002	600960	0.005	1050173
Counting	0	51246	0.997	157948	0.998	269580
Radix	8.999	100120	19.982	300120	35.005	500120
Flash	0.998	127964	1.994	383958	1.996	639960
Data order: Nearly Sorted Data						
Data size	**100,000**		**300,000**		**500,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	20612.1	5000050019	169398	45000150019	458130	125000250019
Insertion	2.992	812794	3.988	1412794	4.986	1881998
Bubble	23948	10000099999	217742	90000299999	618530	250000499999
Shaker	1.996	743748	3.998	1143748	4.001	1493991
Shell	34713.6	8332934184	274147	66235031643	794963	201126275534
Heap	38.897	8365043	127.687	27413221	219.44	47404978
Merge	75.811	5725779	242.353	18435351	368.043	31633313
Quick	0.008	2249919	0.033	7424026	0.047	12931216
Counting	1.995	569580	5.984	1769580	9.973	2970669
Radix	62.831	1000120	199.467	3000120	396.951	5000120
Flash	3.99	127.9960	12.967	3839966	21.924	6399964

1.2. Graph:

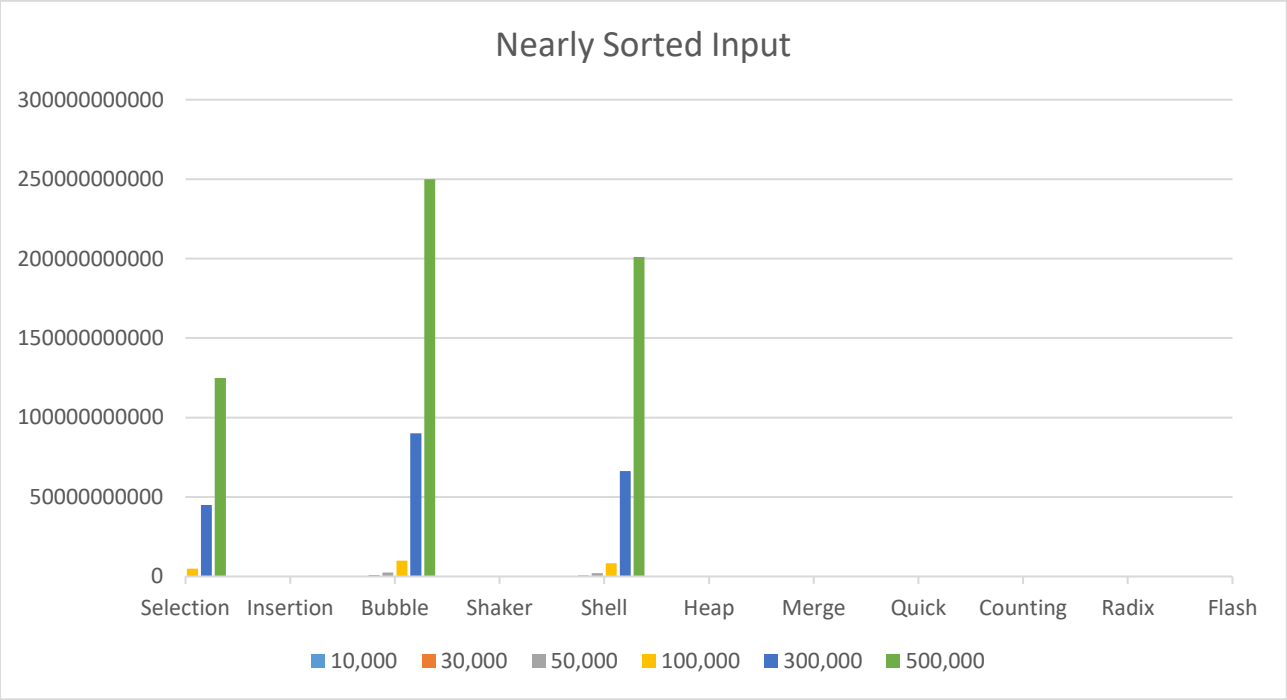


Nhận xét:

- **Đối với các kiểu dữ liệu 10000, 30000, 50000 thì thời gian chạy của tất cả các thuật toán đều nằm trung bình ở mức chưa tới 100000 (ms), nhưng sang các kiểu dữ liệu lớn hơn như 100000, 300000, 500000 thì thuật toán Shell Sort và Bubble Sort có sự biến động mạnh. Cụ thể từ 100000 phần tử đến 300000 phần tử thì thời gian chạy của Bubble Sort là khoảng 300000 (ms) và Shell Sort hơn 200000 (ms), từ 300000 phần tử đến 500000 phần tử thì thời gian chạy của Bubble Sort là khoảng 800000 (ms) và Shell Sort là 600000 (ms).**
- Ai chọn normal giùm Fat trên style đi, đoạn dưới nè.

Tổng quan từ đồ thị cho thấy thuật toán chạy nhanh và ổn định nhất ở kiểu dữ liệu này là Flash Sort, và thuật toán chạy chậm nhất, không ổn định nhất ở kiểu dữ liệu này là Bubble Sort với thời gian chạy có thể lên đến 800000 (ms) (khoảng 13 phút).

1.3. Chart:

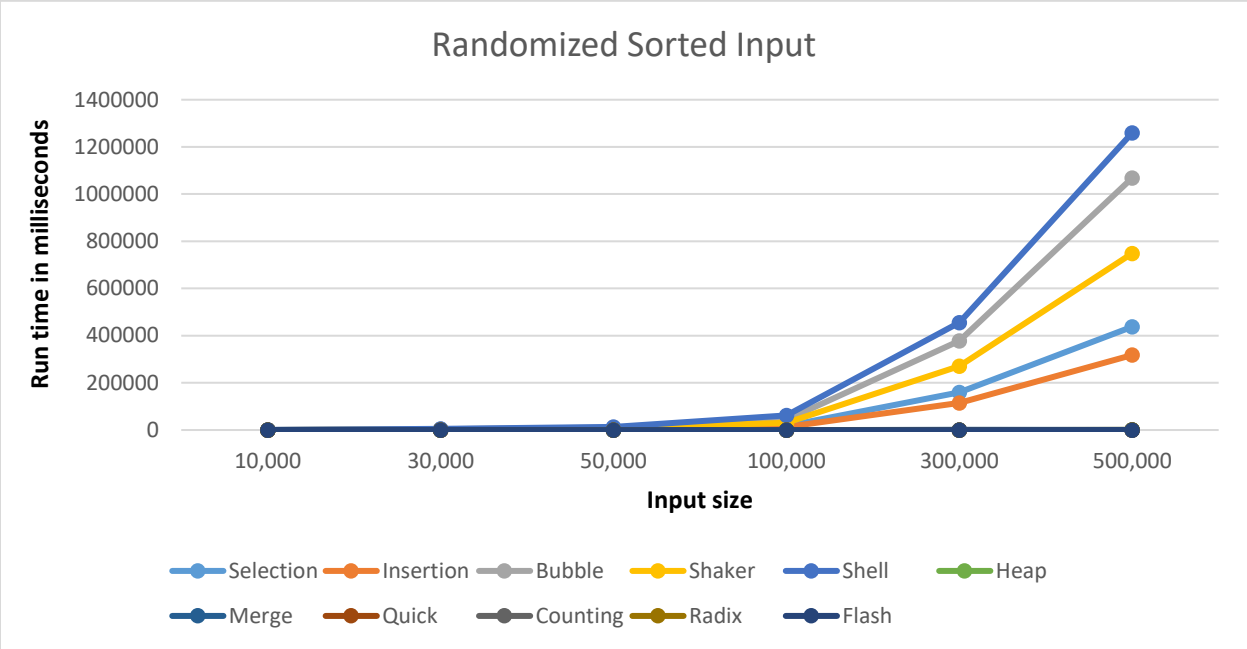

Nearly Sorted Input
300000000000
250000000000
200000000000
150000000000
100000000000
50000000000
0
Selection
Insertion
Bubble
Shaker
Shell
Heap
Merge
Quick
Counting
Radix
Flash
10,000
30,000
50,000
100,000
300,000
500,000

2. Randomized input:

2.1. Table:

Data order: Randomized data						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	254.561	50080672	1875.06	450277706	4431.54	1250481478
Insertion	122.027	50010814	1110.16	451347313	3101.87	1252030661
Bubble	370.083	100009999	3705.89	900029999	10461.2	2500049999
Shaker	279.071	66681501	2674.18	602102447	7611.3	1670702092
Shell	468.266	92574192	4381.93	896854636	12953.7	2597162521
Heap	3.019	638204	10.971	2150609	19.946	3772472
Merge	8.987	573565	22.968	1907363	52.858	3332624
Quick	0.001	214858	0.004	810364	0.007	1367146
Counting	0.001	5006	0.997	150008	0.998	215549
Radix	5.984	100120	19.948	300120	31.422	500120
Flash	0	99761	0.999	304942	2.022	478136
Data order: Randomized data						
Data size	**100,000**		**300,000**		**500,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	17393.3	5000985524	159966	45000985524	437499	125005168851
Insertion	12535.7	4996453191	114679	44961955373	317739	125130680193
Bubble	42407.1	10000099999	378032	90000299999	1068370	250000499999
Shaker	30828.1	6668204177	270412	59971423393	748855	166804982936
Shell	61815.1	10380416917	454895	84163705007	1259520	248085467618
Heap	41.914	8043005	188.497	26491036	248.337	45964799
Merge	84.748	7065043	265.316	23082751	429.878	39882813
Quick	0.014	3029921	0.046	10667975	0.085	20258882
Counting	0.997	365544	3.99	965544	6.982	1565544
Radix	66.822	1000120	187.007	3000120	574.982	5000120
Flash	5.017	908590	16.93	2774595	30.941	4620408

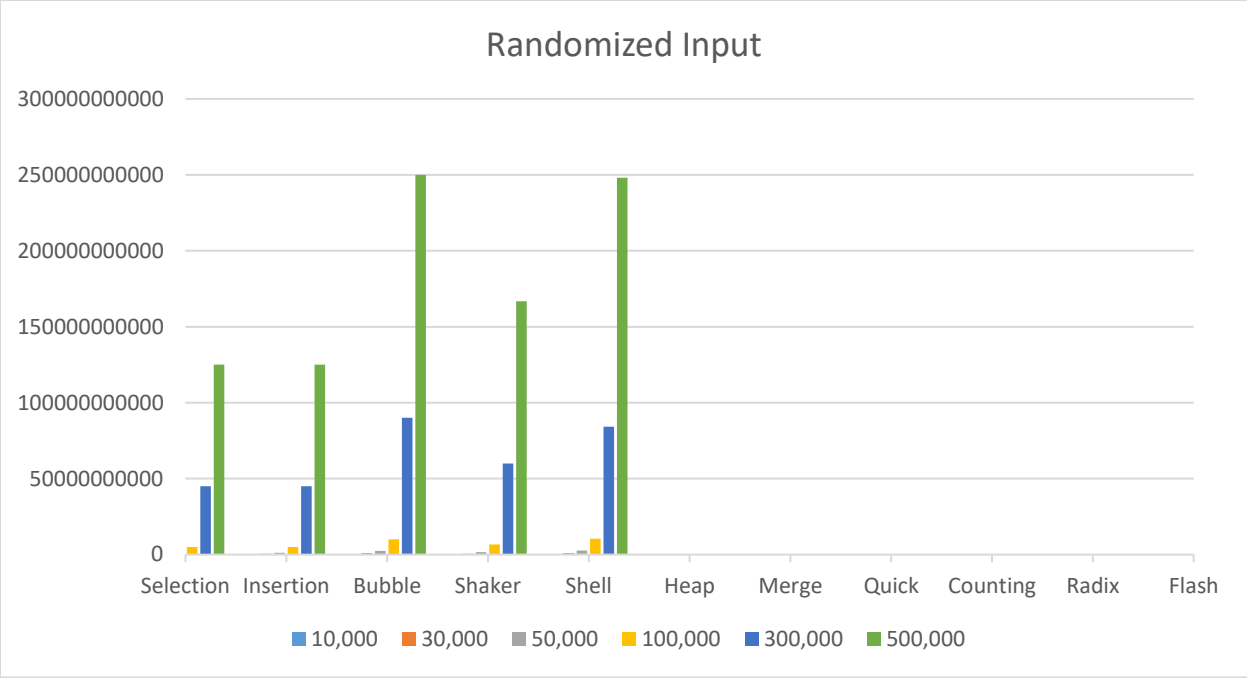
2.2. Graph:



Nhận xét:

- Khi dữ liệu được phân bố ngẫu nhiên thì thuật toán Flash Sort có thời gian chạy ít nhất. Lý do là vì thuật toán này được xây dựng để chạy trên tập dữ liệu phân bố rời rạc.
- Mặt khác, các thuật toán có độ phức tạp $O(n \log_2 n)$ cũng có thời gian chạy rất nhanh và gần như không tăng nhiều khi số lượng dữ liệu tăng lên. Ngược lại, các thuật toán có độ phức tạp bình phương thì lại có tốc độ tăng khá nhanh. Nhanh nhất là Shell Sort, tiếp đến là Bubble Sort, Shaker Sort, Selection Sort và Insertion Sort khi dữ liệu có từ 100000 phần tử trở lên.
- Có thể thấy hai thuật toán dựa vào hoán vị các cặp nghịch thế như Bubble Sort và Shaker Sort đều có thời gian rất lâu khi chạy trên kiểu dữ liệu này. Nhìn tổng quát thì các thuật toán nếu có sự tăng lên của thời gian chạy, thì lại tăng rất đột ngột.

2.3. Chart:

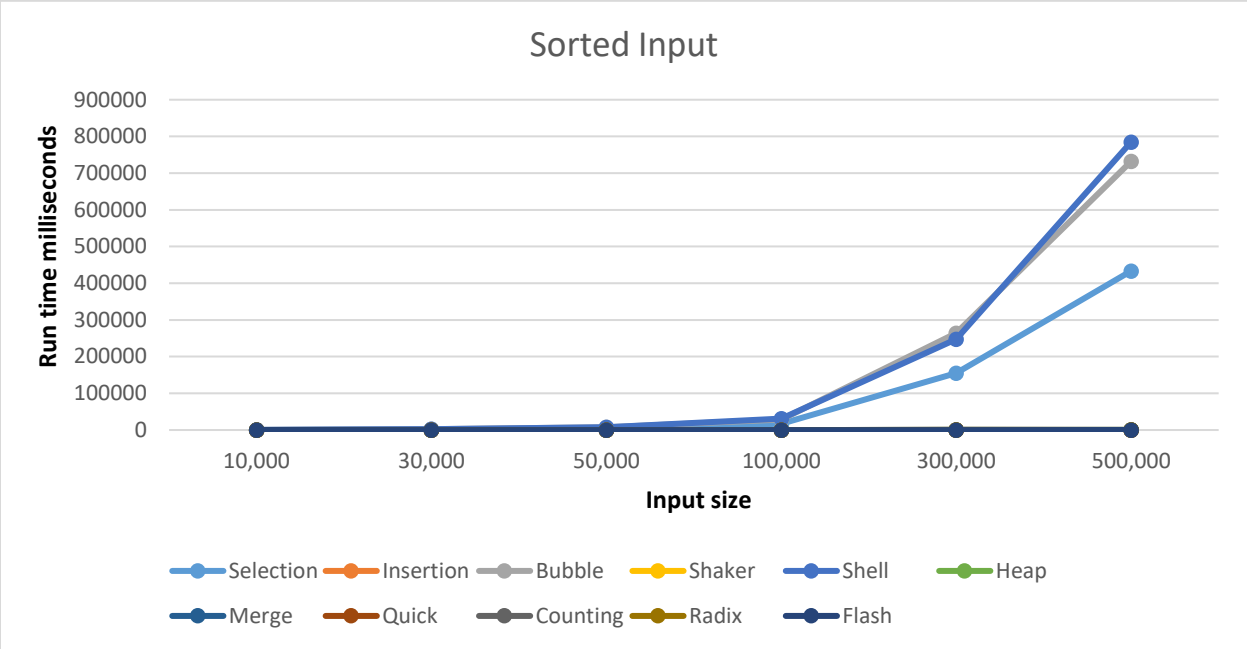

Randomized Input
300000000000
250000000000
200000000000
150000000000
100000000000
50000000000
0
Selection
Insertion
Bubble
Shaker
Shell
Heap
Merge
Quick
Counting
Radix
Flash
10,000
30,000
50,000
100,000
300,000
500,000

3. Sorted input:

3.1. Table:

Data order: Sorted data						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	174.533	50004999	1569.8	450014999	4487.01	1250024999
Insertion	0	29998	0	89998	0.997	149998
Bubble	282.275	100009999	2539.19	900029999	7234.47	2500049999
Shaker	0	20002	0	60002	0	100002
Shell	262.299	73301019	2686.82	727575544	7647.56	2083319524
Heap	2.995	670329	10.971	2236648	17.979	3925351
Merge	7.007	465243	22.932	15.29915	36.902	2672827
Quick	0	174167	0.002	593163	0.003	1049787
Counting	0	59998	0.998	179998	0.998	299998
Radix	5.984	100120	18.456	300120	32.912	500120
Flash	0	127992	1.024	3839992	2.995	639992
Data order: Sorted data						
Data size	**100,000**		**300,000**		**500,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	17116.2	5000049999	155146	4500014999	433430	125000249999
Insertion	0.998	299998	2.992	899998	3.99	1499998
Bubble	28711	10000099999	263563	90000299999	732217	250000499999
Shaker	0	200002	1.994	600002	1.994	1000002
Shell	30700.1	8332677790	247677	66234776337	784912	201126103846
Heap	38.925	8365080	128.683	27413230	220.438	47404886
Merge	74.776	5645659	240.359	18345947	372.035	31517851
Quick	0.007	2249533	0.028	7423143	0.06	12930403
Counting	2.992	599998	5.984	1799998	9.973	2999998
Radix	71.808	1000120	201.966	3000120	317.164	5000120
Flash	4.017	1279992	13.963	3839992	21.968	6399992

3.2. Graph:

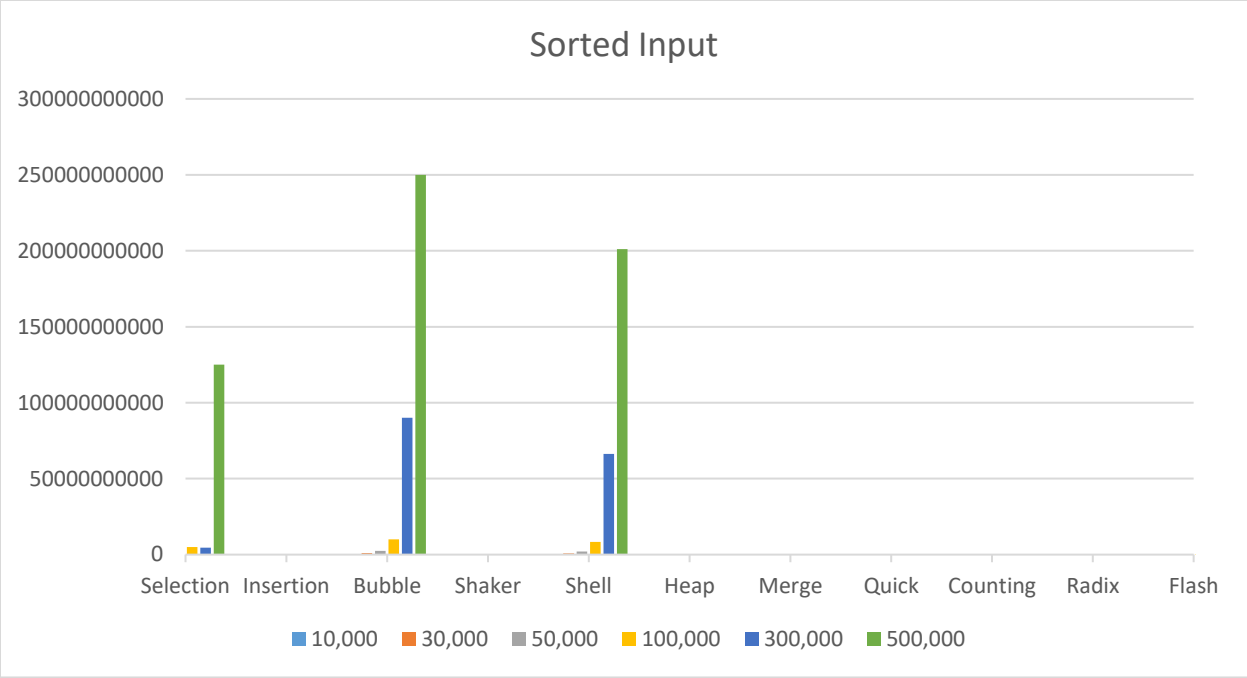


Nhận xét:

Các kiểu dữ liệu từ 10000 đến 100000, các thuật toán chạy với thời gian ổn định, không tăng quá cao. Nhưng từ kiểu dữ liệu 100000 đến 50000, ta thấy các thuật toán như Selection Sort, Shell Sort và Bubble Sort lại có chuyển biến mạnh. Ở kiểu dữ liệu 300000, thì thời gian chạy của Selection Sort là gần 200000 (ms), còn thời gian chạy của Shell Sort và Bubble Sort khá tương đồng nhau đều khoảng 300000 (ms). Ở mốc dữ liệu cao nhất thì thời gian của Selection Sort là hơn 400000 (ms), thời gian chạy Bubble Sort và Shell Sort đều nằm ở khoảng từ 700000 (ms) đến 800000 (ms).

Tổng quan từ đồ thị thì ở loại kiểu dữ liệu này thì thuật toán chạy nhanh và ổn định nhất vẫn là Flash Sort, các thuật toán chạy chậm nhất và không ổn định nhất ở kiểu dữ liệu này là Bubble Sort và Shell Sort với thời gian chạy lên đến 800000 (ms) (khoảng 13 phút).

3.3. Chart:

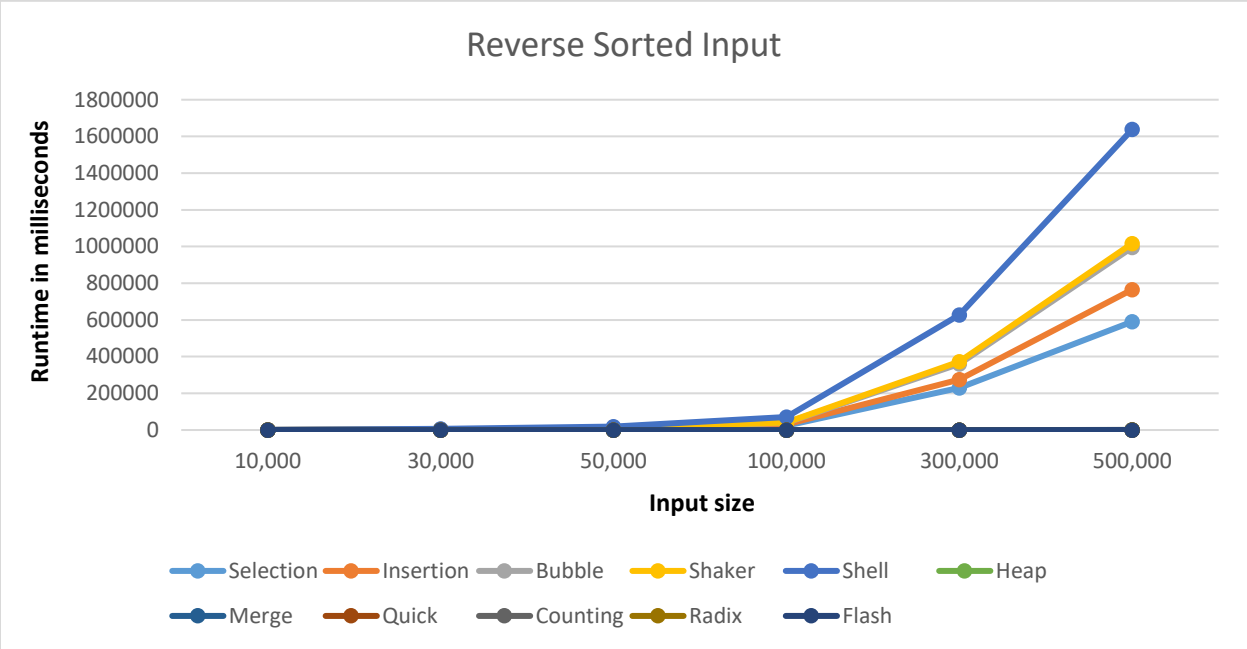

Sorted Input
300000000000
250000000000
200000000000
150000000000
100000000000
50000000000
0
Selection
Insertion
Bubble
Shaker
Shell
Heap
Merge
Quick
Counting
Radix
Flash
10,000
30,000
50,000
100,000
300,000
500,000

4. Reverse Sorted Input:

4.1. Table:

Data order: Reverse sorted data						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	232.379	75004999	2128.31	675014999	5910.77	1875024999
Insertion	292.236	100009999	2620.99	900029999	7269.59	2500049999
Bubble	401.114	100009999	3585.91	900029999	10209	2500049999
Shaker	402.103	100005001	3627.31	900015001	10096.5	2500025001
Shell	967.962	107487811	6589.08	1022777008	18344.2	2997128084
Heap	2.02	606771	9.986	2063324	16.972	3612724
Merge	7.006	466442	20.97	1543466	40.89	2683946
Quick	0.001	175031	0.003	596178	0.005	1046488
Counting	0.999	49999	0.997	149999	0.997	249999
Radix	6.981	100120	34.907	300120	50.865	500120
Flash	0.994	130496	1.973	391496	2.968	652496
Data order: Reverse sorted data						
Data size	**100,000**		**300,000**		**500,000**	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection	24609.2	7500049999	229131	67500149999	590262	187500249999
Insertion	31220.9	10000099999	274186	90000299999	765409	250000499999
Bubble	39830.5	10000099999	360297	90000299999	995206	250000499999
Shaker	40713.2	10000050001	372218	90000150001	1016610	250000250001
Shell	70188.4	11988029782	626873	98618381421	1638010	283513118880
Heap	36.928	7718943	158.577	25569379	209.44	44483348
Merge	73.804	5667898	243.325	18408314	378.987	31836410
Quick	0.009	2242931	0.028	7459112	0.071	12993708
Counting	1.995	499999	4.987	1499999	8.975	2499999
Radix	102.726	1000120	202.965	3000120	331.127	5000120
Flash	3.991	1304996	12.996	3914987	21.941	652996

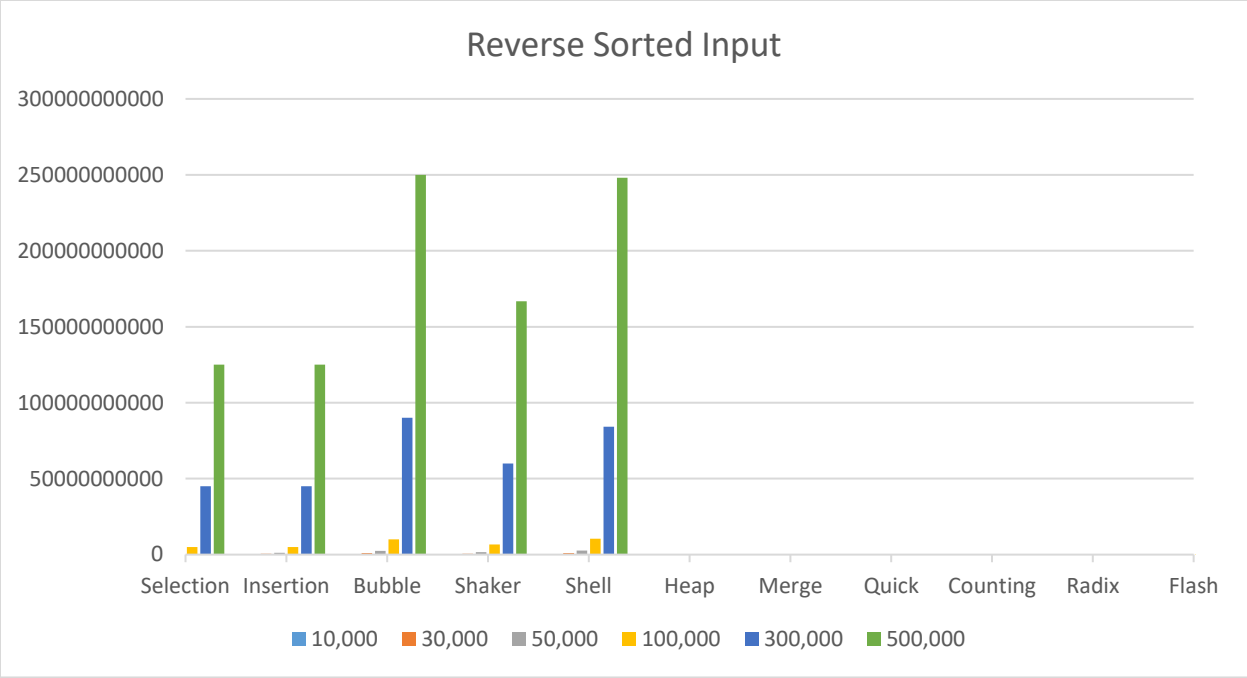
4.2. Graph:



Nhận xét: khi dữ liệu bị sắp xếp đảo ngược như thế này, thời gian chạy của hầu hết các thuật toán đều ở mức rất cao (trên 1000000 ms).

Các thuật toán có độ phức tạp O(n.log2n) đều có thời gian chạy rất ổn định và hầu như không tăng đáng kể. Các thuật toán O(n2) lại có thời gian thực thi khá chậm. Đặc biệt, Shaker Sort và Bubble Sort đều có tốc độ tăng như nhau. Mặc dù vậy thì Shell Sort vẫn là thuật toán có tốc độ tăng nhanh nhất khi dữ liệu từ 100000 phần tử trở lên. Nhìn chung thì các thuật toán không tăng thời gian chạy đột ngột mà tăng dần khi chuyển giao từ 100000 sang 300000 và từ 300000 sang 500000.

4.3. Chart:


Reverse Sorted Input
300000000000
250000000000
200000000000
150000000000
100000000000
50000000000
0
Selection
Insertion
Bubble
Shaker
Shell
Heap
Merge
Quick
Counting
Radix
Flash
10,000
30,000
50,000
100,000
300,000
500,000

Phần 3: BÁO CÁO ĐỒ ÁN

- **Về cấu trúc dữ liệu:**
  - Sử dụng các cấu trúc dữ liệu có sẵn trong STL của C++ như vector, queue để mã nguồn gọn nhẹ hơn và tối ưu hơn.
  - Sử dụng thư viện (cũng như là sub-namespace) chrono để tính toán thời gian chạy.
- **Về tổ chức mã nguồn:**
  - Các hàm phục vụ thực nghiệm được để riêng ra từng file và header “Sorting.h” chứa định nghĩa của các hàm này.
  - File “CommandLineArgument.cpp” là hàm main phục vụ cho việc chạy các lệnh trong file “cmd.cpp” chứa các hàm triển khai từng nội dung bên trong và “cmd.h” chứa các định nghĩa.

TÀI LIỆU THAM KHẢO

[1] S. Nguyễn, "Selection Sort," 18 8 2020. [Online]. Available: https://www.stdio.vn/giai-thuat-lap-trinh/selection-sort-KQi3U. [Accessed 25 11 2021].

[2] N. V. Hiếu, "Thuật toán sắp xếp selection sort minh họa code sử dụng c++," [Online]. Available: https://nguyenvanhieu.vn/thuat-toan-sap-xep-selection-sort/. [Accessed 25 11 2021].

[3] "Thuật toán sắp xếp nhanh – Quick Sort Algorithm C/C++," 5 5 2021. [Online]. Available: https://duongdinh24.com/thuat-toan-quick-sort/#4_Danh_gia_thuat_toan_sap_xep_nhanh. [Accessed 25 11 2021].

[4] cplusplus.com, "Reference <chrono>," [Online]. Available: https://www.cplusplus.com/reference/chrono/duration/duration/.

[5] HaiZuka, "Lý thuyết và bài tập Insertion Sort," [Online]. Available: https://codelearn.io/learning/data-structure-and-algorithms/850321.

[6] N. V. Hiếu, "Thuật toán sắp xếp Bubble Sort minh họa code sử dụng C++," 14 07 2018. [Online]. Available: https://nguyenvanhieu.vn/thuat-toan-sap-xep-bubble-sort/. [Accessed 28 11 2021].

[7] N. V. Hiếu, "Thuật toán sắp xếp chèn - Insertion Sort," 01 09 2019. [Online]. Available: https://nguyenvanhieu.vn/thuat-toan-sap-xep-chen/. [Accessed 28 11 2021].

[8] khanh_48, "Các Thuật Toán Sắp Xếp Trong C++," 23 10 2020. [Online]. Available: https://codelearn.io/sharing/cac-thuat-toan-sap-xep-trong-cpp. [Accessed 28 11 2021].

[9] N. Shiro, "Bubble Sort và Shaker Sort," 18 08 2020. [Online]. Available: https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U. [Accessed 28 11 2021].

[10] N. Shiro, "Insertion Sort," 18 08 2020. [Online]. Available: https://www.stdio.vn/giai-thuat-lap-trinh/insertion-sort-Ri3U12. [Accessed 28 11 2021].

[11] C. A. Shaffer, Data Structures and Algorithm in C++ (3rd Edition), Virginia, USA, 2011.

[12] R. Sedgewick, Algorithms in C, Addison-Wesley , 2006.

[13] L. M. Hoàng, Giải thuật và Lập trình.

[14] J. Edmons, How to think about algorithms, Cambrige University Press, 2008.

[15] N. V. Hiếu, "Counting Sort – Thuật toán sắp xếp đếm phân phối," [Online]. Available: https://nguyenvanhieu.vn/counting-sort/.

[16] D. E. Knuth, The Art of Computer Programming, Addison-Wesley, 1998.

[17] D. L. Shell, "A High Speed Sorting Procedure," *Communications of the ACM,* p. 3, 1959.

[18] "The Flashsort1 Algorithm | Dr Dobb's," 01 02 1998. [Online]. Available: https://www.drdobbs.com/database/the-flashsort1-algorithm/184410496.

[19] "StackOverflow," 6 2014. [Online]. Available: https://stackoverflow.com/questions/24171242/why-is-mergesort-space-complexity-ologn-with-linked-lists.

[20] "Merge Sort Algorithm With Example Program - InterviewBit," [Online]. Available: https://www.interviewbit.com/tutorial/merge-sort-algorithm/.

[21] "Merge Sort Algorithm | Studytonight," [Online]. Available: https://www.studytonight.com/data-structures/merge-sort.

[22] "Merge Sort - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Merge_sort.

[23] "Linear Time BuildHeap," 6 2 2015. [Online]. Available: https://www.youtube.com/watch?v=MiyLo8adrWw.

[24] "Is there an algorithm more efficient in memory and faster than flashsort? - Quora," 8 6 2018. [Online]. Available: https://www.quora.com/Is-there-an-algorithm-more-efficient-in-memory-and-faster-than-flashsort.

[25] "Heapsort – Algorithm, Source Code, Time Complexity," 19 8 2020. [Online]. Available: https://www.happycoders.eu/algorithms/heapsort/#Heapsort_Time_Complexity.

[26] "Heap Sort (with code)," [Online]. Available: https://www.programiz.com/dsa/heap-sort.

[27] "Heap Sort - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Heapsort.

[28] "Flash Sort - Thuật Toán Sắp Xếp Thần Thánh," 18 03 2021. [Online]. Available: https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh. [Accessed 26 11 2021].

[29] "Flash sort - công cụ mới để tối ưu tốc độ giải thuật," 7 12 2004. [Online]. Available: https://www.ddth.com/showthread.php/64851-Flash-sort-c%C3%B4ng-c%E1%BB%A5-m%E1%BB%9Bi-%C4%91%E1%BB%83-t%E1%BB%91i-%C6%B0u-t%E1%BB%91c-%C4%91%E1%BB%99-gi%E1%BA%A3i-thu%E1%BA%ADt.

[30] "Binary Tree Data Structure," 15 11 2021. [Online]. Available: https://www.geeksforgeeks.org/binary-tree-data-structure/.

[31] "[JAVA] MERGE SORT: Thuật toán sắp xếp trộn," 9 9 2020. [Online]. Available: https://niithanoi.edu.vn/merge-sort.html.