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

NHÓM 13

REPORT

LAB 3 - SORTING

CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

Thành Phố Hồ Chí Minh – 2021

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

NHÓM 13

REPORT

LAB 3 – SORTING

|Giáo viên hướng dẫn|

Ts. Nguyễn Thanh Phương

Ths. Bùi Huy Thông

Ts. Nguyễn Ngọc Thảo

CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

Thành Phố Hồ Chí Minh – 2021

INTRODUCTION PAGE

Nhóm 13

- Nguyễn Đình Văn – 20127662
- Trần Thị Bảo Hương – 20127514
- La Gia Bảo – 19127336
- Trần Đại Quốc – 20127609

Selection: Set 2 (11 algorithms):

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shaker Sort
- Shell Sort
- Heap Sort
- Merge Sort
- Quick Sort
- Counting Sort
- Radix Sort
- Flash Sort

THE WORK OF EACH PERSON

- Nguyễn Đình Văn(20127662): Code & Algorithm Presentation(Selection Sort, Counting Sort) + Algorithm Presentation(Flash Sort) Output specifications
- Trần Thị Bảo Hương(20127514): Code & Algorithm Presentation(Quick Sort, Radix Sort, Shaker Sort, Insertion Sort) + 4 Experimental results
- La Gia Bảo(19127336): Code & Algorithm Presentation(Bubble Sort, Heap Sort, Shell Sort) + Code(Flash Sort) + 4 Experimental results
- Trần Đại Quốc(20127609): Code & Algorithm Presentation(Merge Sort) + Report + Chart Draw

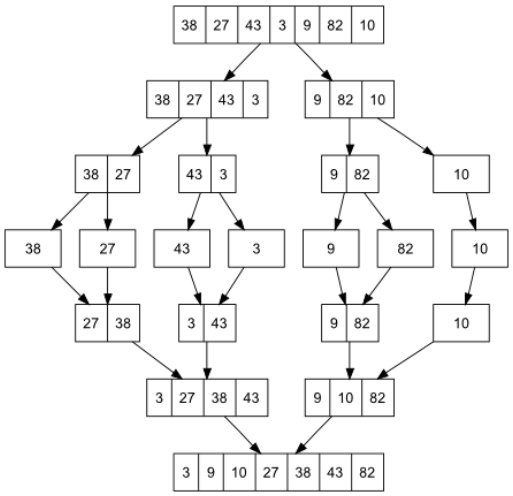
TABLE OF CONTENTS

Contents

ALGORITHM PRESENTATION

1. Merge Sort

- *Ý tưởng:*Chia mảng lớn thành những mảng con nhỏ hơn bằng cách chia đôi mảng lớn và tiếp tục chia đôi các mảng con cho đến khi mảng con nhỏ nhất chỉ còn 1 phần tử. So sánh 2 mảng con có cùng mảng cơ sở (khi chia đôi mảng lớn thành 2 mảng con thì mảng lớn đó gọi là mảng cơ sở của 2 mảng con đó. Khi so sánh chúng vừa sắp xếp vừa ghép 2 mảng con đó lại thành mảng cơ sở, tiếp tục so sánh và ghép các mảng con lại đến khi còn lại mảng duy nhất, đó là mảng đã được sắp xếp.



Hãy xem ý tưởng triển khai code dưới đây để hiểu hơn:

```
mergeSort(arr[], l,  r)
If (r > l)
    1. Tìm chỉ số nằm giữa mảng để chia mảng thành 2 nửa:
         middle m = (l+r)/2
    2. Gọi đệ quy hàm mergeSort cho nửa đầu tiên:
         mergeSort(arr, l, m)
    3. Gọi đệ quy hàm mergeSort cho nửa thứ hai:
         mergeSort(arr, m+1, r)
    4. Gộp 2 nửa mảng đã sắp xếp ở (2) và (3):
         merge(arr, l, m, r)
```

- *Step by Step:*
  Bước 1: khởi tạo ba chỉ số chạy trong vòng lặp i = 0, j = 0, k = 0 tương ứng cho ba mảng A, B và C.
  Bước 2: tại mỗi bước nếu cả hai chỉ số (i<m và j<n) ta chọn min(A[i],B[j]) và lưu nó vào trong C[k]. Chuyển sang Bước 4.
  Bước 3: tăng giá trị k lên 1 và quay về Bước 2.
  Bước 4: sao chép tất cả các giá trị còn lại từ các danh sách mà chỉ số còn vi phạm (tức i<m hoặc j<m) vào trong mảng C
- *Ví dụ minh họa:*
  - Giả sử ta có 2 mảng con lần lượt là:
  - arr1 = [1 9 10 10] , n1 = 4 // chiều dài của mảng con

- arr2 = [3 5 7 9], n2 = 4
- sort_arr = [] // Mảng lưu lại tiến trình gộp
- Khởi tạo i = 0, j = 0 tương ứng là chỉ số bắt đầu của arr1 và arr2
- Xét thấy arr1[i] < arr2[j] => chèn arr1[i] vào cuối mảng sort_arr, tăng i lên 1 đơn vị
  ⇨ sort_arr = [1], i = 1
- Xét thấy arr1[i] > arr2[j] => chèn arr2[j] vào cuối mảng sort_arr, tăng j lên 1 đơn vị
  ⇨ sort_arr = [1, 3], i = 1, j = 1
- Xét thấy arr1[i] > arr2[j] => chèn arr2[j] vào cuối mảng sort_arr, tăng j lên 1 đơn vị
  ⇨ sort_arr = [1, 3, 5], i = 1, j = 2
- Xét thấy arr1[i] > arr2[j] => chèn arr2[j] vào cuối mảng sort_arr, tăng j lên 1 đơn vị
  ⇨ sort_arr = [1, 3, 5, 7], i = 1, j = 3
- Xét thấy arr1[i] = arr2[j] => chèn arr1[i] hoặc arr2[j] vào cuối mảng sort_arr
- Giả sử tôi chọn arr1, tăng i lên 1 đơn vị
  ⇨ sort_arr = [1, 3, 5, 7, 9], i = 2, j = 3
- Xét thấy arr1[i] > arr2[j] => chèn arr2[j] vào cuối mảng sort_arr, tăng j lên 1 đơn vị
  ⇨ sort_arr = [1, 3, 5, 7, 9, 9], i = 1, j = 4
- Do j >= n2, tiếp tục tăng i chừng nào i < n1 thi thêm phần tử ở arr1[i]ư vào sort_arr.
- Sau cùng ta đượ mảng đã sắp xếp là sort_arr = [1, 3, 5, 7, 9, 9, 10, 10]

- *Độ phức tạp thuật toán:*
  - Trường hợp tốt: $O(nlog(n))$
  - Trường hợp trung bình: $O\big(nlog(n)\big)$
  - Trường hợp xấu: $O(nlog(n))$
- *Không gian bộ nhớ sử dụng*: $O(n)$
- *Phức tạp thời gian*: $O\big(nlog(n)\big)$

2. Selection Sort

- *Ý tưởng:* 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

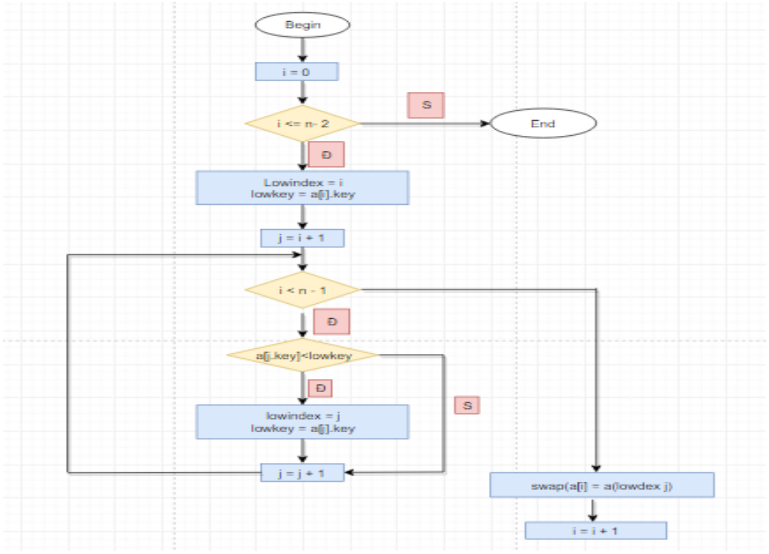
  1. Một mảng con đã được sắp xếp
  2. Một mảng con chưa được sắp xếp

  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.

  Step by Step Selection – Sort:

- Bước 1: i = 0.
- Bước 2: Tìm phần tử a[iMin] trong dãy hiện hành từ a[i] đến a[n-1].
- Bước 3: Đổi chỗ a[iMin] và a[i].
- Bước 4: Nếu i < n – 1 thì lặp lại bước 2 với i = i + 1;

Lưu đồ Selection – Sort:



- *Ví dụ:*

  Array [] = 9 8 15 4 1

// Tìm phần tử nhỏ nhất trong trong arr[0...4]
// và đổi chỗ nó với phần tử đầu tiên

[1] 8 15 4 9

// Tìm phần tử nhỏ nhất trong trong arr[1...4]
// và đổi chỗ nó với phần tử đầu tiên của arr[1...4]

1 [4] 15 8 9

// Tìm phần tử nhỏ nhất trong trong arr[2...4]
// và đổi chỗ nó với phần tử đầu tiên của arr[2...4]

1 4 [8] 15 9

// Tìm phần tử nhỏ nhất trong trong arr[3...4]
// và đổi chỗ nó với phần tử đầu tiên của arr[3...4]

1 4 8 9 [15]

- *Độ phức tạp của thuật toán:*
  - Trường hợp tốt: O(n^2)
  - Trung bình: O(n^2)
  - Trường hợp xấu: O(n^2)
  - Không gian bộ nhớ sử dụng: O(1)

- *Cải tiến*: Dựa trên ý tưởng của Selection – Sort, ta sẽ giảm bớt số lần hoán đổi khi vị trí của phần tử nhỏ nhất trùng với vị trí của phần tử đầu tiên của đoạn chưa được sắp xếp.

  - Code khi cải tiến:

```
void selectionSort( int a[],  int n){
  for ( int i = 0; i < n-1; i++){
     int min = i;
    for ( long long j = i+1; j < n; j++)
      if (a[j] < a[min])
        min = j;
    if (j != min){
          int temp = a[min];
          a[min] = a[i];
          a[i] = temp;
     }
  }
}
```

  - Trường hợp tốt: O(n^2)
  - Trung bình: O(n^2)
  - Trường hợp xấu: O(n^2)

3. <u>Counting Sort</u>

- *Ý tưởng:* Thuật toán Counting – Sort sắp xếp một mảng bằng cách đếm số lần xuất hiện của mỗi phần tử có trong mảng. Sau khi thực hiện đếm số lần xuất hiện của mỗi phần tử ta thu được một mảng đếm số lần xuất hiện, từ mảng đó ta tiến hành cộng đồn theo công thức Arr[i] = Arr[i-1] + Arr[i] (chạy từ vị trí có giá trị min đến vị trí có giá trị max của mảng ban đầu). Từ giá trị của mỗi phần tử mảng ban đầu ta so với mảng vừa có, ta sẽ suy ra được vị trí của phần tử khi được sắp xếp.

Step by step Counting – Sort:

- Bước 1: Tìm Min và Max của mảng A[0..n] có n phần tử.
- Bước 2: Đếm số lần xuất hiện của các phần tử của mảng A lưu vào mảng A_Count[0.. Max]. Theo công thức A_Count[A[i]]++ (nêu phần tử A[i] tồn tại trong mảng A).
- Bước 3: Từ mảng A_Count ta tiến hành biến đổi mảng theo công thức A_Count[i] = A_Count[i – 1] + A_Count[i] (Với i chạy từ Min đến Max)
- Bước 4: Tạo một mảng Arr[0..n] là mảng lưu kết quả sắp xếp mảng A[0..n]
- Bước 5: i = 0.
- Bước 6: Duyệt mảng A để suy ra mảng Arr theo công thức Arr[A_Count[A[i]] = A[i] và A_Count[A[i]] –
- Bước 7: Tăng i = i + 1 và nếu i < n thì quay lại bước 5, nếu không thì qua bước 8.
- Bước 8: Ta thu được mảng Arr[0..n] là mảng lưu kết quả của A[0..n] sau khi sắp xếp.

- *Ví dụ minh họa*:

  Để đơn giản, hãy xem xét dữ liệu trong phạm vi từ 0 đến 9.

  Dữ liệu đầu vào: 1, 4, 1, 2, 7, 5, 2

  1) Lấy một mảng đếm để lưu trữ số lượng của mỗi đối tượng duy nhất.

  Chỉ số: 0 1 2 3 4 5 6 7 8 9

  Đếm: 0 2 2 0 1 1 0 1 0 0

  2) Sửa đổi mảng đếm sao cho mỗi phần tử ở mỗi chỉ mục

  lưu trữ tổng của các lần đếm trước đó.

  Chỉ số: 0 1 2 3 4 5 6 7 8 9

  Đếm: 0 2 4 4 5 6 6 7 7 7

  Mảng đếm đã sửa đổi cho biết vị trí của từng đối tượng trong trình tự đầu ra.

  3) Xuất từng đối tượng từ chuỗi đầu vào, theo sau là giảm số lượng của nó đi 1. Xử lý dữ liệu đầu vào: 1, 4, 1, 2, 7, 5, 2. Vị trí của 1 là

  2. Đặt dữ liệu 1 ở chỉ mục 2 trong đầu ra. Giảm số lượng đi 1 vị trí

  dữ liệu tiếp theo 1 ở chỉ mục 1 nhỏ hơn chỉ mục này.

  Ta thu được dãy: 1 1 2 2 4 5 7

- *Độ phức tạp của thuật toán:*
  - Trường hợp tốt: O(n)
  - Trung bình: O(n)
  - Trường hợp xấu: O(n^2)

4. Radix Sort

- *Ý tưởng thuật toán:*

Cho dãy số nguyên có n phần tử: $a_0, a_1, ... a_{n-1}$;

Giả sử mỗi phần tử a_i trong dãy là một số nguyên có tối đa m chữ số . Phân loại các phần tử này lần lượt theo hàng đơn vị, chục, trăm, ngàn... nói đơn gì hơn chúng ta phân loại các chữ số từ chữ số bắt đầu từ chữ số có giá nhỏ nhất (hàng đơn vị) đến chữ số có nghĩa lớn nhất (hàng nghìn, hàng tỷ,...)

- *Step by Step:*

Bước 1: Tạo một bảng có kích thước là 10 x n: Với số dòng là 10 ứng với các chữ số từ 0 đến 9, n là số phần tử của mảng .

Bước 2: Đưa các giá trị trong mảng vào bảng ứng với chữ số mà bạn đang xét.

Bước 3: Cập nhật lại thứ tự dãy số sau khi phân loại.

Bước 4: Kiểm tra xem đó đã có dấu hiệu kế thúc chưa. Nếu có thì kết thúc, nếu chưa quay lại bước 2

- *Ví dụ*

Mảng A cần sắp xếp

78	149	63	91	124

Phân lô theo hàng đơn vị

0	1	2	3	4	5	6	7	8	9
	91		63	124				78	149

Cập nhật mảng A:

91	63	124	78	149

Phân lô theo hàng chục

0	1	2	3	4	5	6	7	8	9
		124		149		63	78		91

Cập nhật mảng A:

124	149	63	78	91

Phân lô theo hàng trăm

0	1	2	3	4	5	6	7	8	9
63	124								
78	149								
91									

Cập nhật mảng A:

63	78	91	124	149

- *Độ phức tạp của thuật toán:*
    - Vì sắp xếp dựa trên cơ số là một thuật toán không so sánh, nó có lợi thế hơn so với các thuật toán sắp xếp so sánh khác.
    - Đối với sắp xếp dựa theo cơ số sử dụng sắp xếp đếm làm một sắp xếp trung gian, độ phức tạp về thời gian là
    - Ở đây, là số vòng chu kỳ và là độ phức tạp về thời gian của sắp xếp đếm.
    - Do đó, sắp xếp dựa theo cơ số có độ phức tạp về thời gian tuyến tính tốt hơn của các thuật toán sắp xếp so sánh.
    - Nếu chúng ta lấy các số có chữ số rất lớn hoặc số lượng các cơ số khác như số 32 bit và 64 bit thì nó có thể thực hiện trong thời gian tuyến tính tuy nhiên thuật toán sắp xếp trung gian chiếm không gian lớn.
    - Điều này làm cho không gian sắp xếp dựa trên cơ số là không hiệu quả. Đây là lý do tại sao kiểu sắp xếp này không được sử dụng trong các thư viện phần mềm

5. Shaker Sort

- *Ý tưởng thuật toán:*

Shaker sort là thuật toán cải tiến của Buble sort. Đối với Buble sort ở mỗi lần duyện chúng ta đưa phần tử nhỏ nhất lê đầu dãy; còn với Shaker sort ở mỗi lần duyện ta đưa ta đồng thời đưa phần thử nhỏ nhất về đầu dãy và lớn nhất về cuối dãy. Do đó trong 1 lần duyện Shaker sort đưa ít nhất 2 phần thử về đúng vị trí của nó.

- *Step by Step:*

Bước 1: Khai báo kiểu biến và giá trị cho biến cần sử dụng trong đoan chương trình, cụ thể chương trình ở đây sử dụng biến left, right, và k.

+ Do chúng ta đang có 1 mảng 1 chiều và nhiệm vụ trong thuật toán shakersort này ta cần phải gán biến left = 0 (chính là biến ở đầu mảng), right = n-1 (biến ở cuối mảng), k = n-1 (định vị trí bắt đầu lắc).

-Bước 2: Sau đó, ta dùng 1 vòng lặp while với điều kiện lặp là left < right. Trong vòng lặp while này sẽ có 2 vòng lặp for nữa, ta cứ hiểu 2 vòng lặp for này là 1 lượt đi và 1 lượt về.

+ Lượt đi:

- Ta duyệt vòng lặp for từ cuối mảng tới đầu mảng, nếu gặp cặp nghịch thế(số trước lớn hơn số đầu) thì ta gọi hàm hoán vị (hoặc dùng swap(M[i-1] ,M[i]);). Trong lần lặp này sẽ đưa được giá trị nhỏ nhất trong mảng về vị trí đầu tiên.
- Dùng biến k đánh dấu để bỏ qua đoạn đã được sắp xếp thứ tự.

+ Lượt về:

- Lấy k = left .Ta duyệt từ đầu mảng đến cuối mảng bắt đầu từ vị trí k , nếu gặp cặp nghịch thế(số trước lớn hơn số đầu) thì ta gọi hàm hoán vị (hoặc dùng swap(M[i] ,M[i+1]);). Trong lần lặp này sẽ đưa được giá trị lớn nhất trong mảng về vị trí cuối cùng.
- Dùng biến k đánh dấu để bỏ qua đoạn đã được sắp xếp thứ tự. Sau đó gán k = right

-Bước 3: Kiểm tra chiều dài đoạn cần sắp xếp nếu = 1 thì ngưng, còn nếu > 1 thì lặp lại bước 2.

- *Ví dụ:*

Mảng A = {7;101;9;98;85;13}

Bắt đầu với left = 0; right = n -1; k = 0

+ Ở lượt đi:

- Biến j sẽ chạy từ right về left, nếu xuất hiện cặp nghịch thế a[j] < a[j -1], ta:
  - Hoán vị a[j] và a[j -1].
  - Cặp nhật biến k = j -1;
- Sau lượt đi ta được mảng A= {7;9;101;13;98;85}
- Biến K lúc này có giá trị là 1; biến left được cập nhật lại bằng biến k.

+ Ở lượt về:

- Biến j sẽ chạy từ left đến, nếu xuất hiện cặp nghịch thế a[j] > a[j + 1], ta:
  - hoán vị chúng.
  - Cặp nhật biến k = j +1;
- Sau lượt đi ta được mảng A= {7;9;13;98;85; 101}
- Biến K lúc này có giá trị là 5, biến right được cập nhật lại bằng biến k.

Tiếp tục lặp lại vì left < right;(left = 1; right = 5;)

+ Ở lượt đi:

- Biến j sẽ chạy từ right về left, nếu xuất hiện cặp nghịch thế a[j] < a[j -1], ta
  - Hoán vị a[i] và a[j -1].
  - Cặp nhật biến k = j -1;
- Sau lượt đi ta được mảng A= {7;9;13;85;98;101}
- Biến K lúc này có giá trị là 3, biến left được cập nhật lại bằng biến k.

+ Ở lượt về:

- Biến j sẽ chạy từ left đến, nếu xuất hiện cặp nghịch thế a[j] > a[j + 1], ta:
  - Hoán vị chúng.
  - Cặp nhật biến k = j +1;
- Sau lượt về ta được mảng A= {7;9;13;85;98; 101}
- Biến K lúc này có giá trị là 3, biến right được cập nhật lại bằng biến k.

Dừng vòng lặp vì left = right;(left = 3; right = 3;)

Sau khi thực hiện ta nhận đượcc một mảng A được sắp xếp.

- *Độ phức tạp:*
  - Độ phức tạp cho trường hợp tốt nhất là O(n).
  - Độ phức tạp cho trường hợp xấu nhất O(n2).
  - Độ phức tạp trong trường hợp trung bình là O(n2).
- Thuật toán nhận diện được mảng đã sắp xếp.

6. Insertion Sort

- *Ý tưởng thuật toán:*

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

- *Step by Step:*

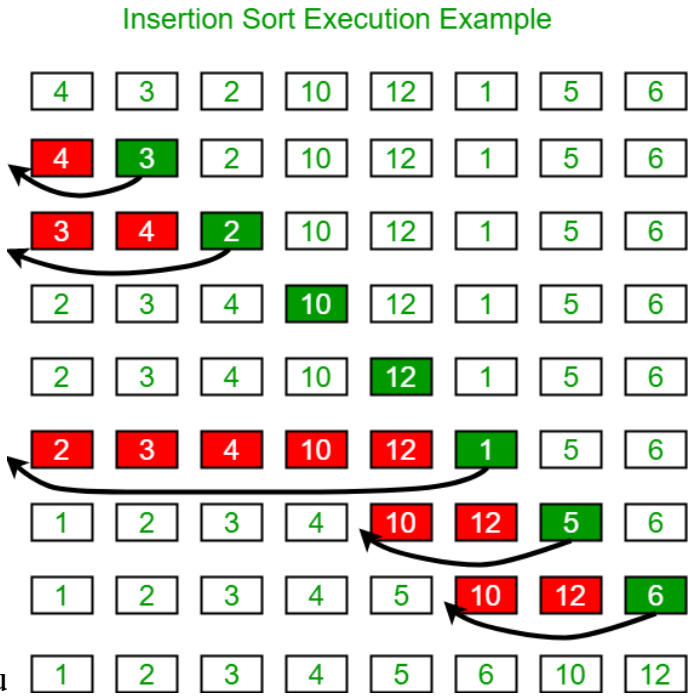
Bước 1: Khởi tạo mảng với dãy con đã sắp xếp có k = 1 phần tử (phần tử đầu tiên, phần tử có chỉ số 0)

Bước 2: Duyệt từng phần tử từ phần tử thứ 2, tại mỗi lần duyệt phần tử ở chỉ số i thì đặt phần tử đó vào một vị trí nào đó trong đoạn từ [0…i] sao cho dãy số từ [0…i] vẫn đảm bảo

tính chất dãy số tăng dần. Sau mỗi lần duyệt, số phần tử đã được sắp xếp k trong mảng tăng thêm 1 phần tử.

Bước 3: Lặp cho tới khi duyệt hết tất cả các phần tử của mảng.

Ví dụ



Hàng đầu tiên là dãy số chưa sắp xếp từ hàng thứ 2 ta chè số đó vào dãy con trước nó sao cho dãy con đó là dãy tăng.

- *Độ phức tạp:*
  - Trường hợp tốt: O(n)
  - Trung bình: O(n^2)
  - Trường hợp xấu: O(n^2)
- Không gian bộ nhớ sử dụng: O(1)

7. Quick Sort

- *Ý tưởng thuật toán:*

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

- Một số cách chọn key:

1. Chọn phần tử đầu
2. Chọn phần tử cuối
3. Chọn phần tử giữa

4. Chọn phần thử ngẫu nhiên
- *Step by Step*

Cho mảng A có n phần tử.

Ta có biến left = 0; right = n – 1 (đánh đấu vị trí của phân đoạn).

Biến key = a[right + left]/ 2; i = left; j = right;

Khi letf nhỏ hơn right

- Khi i < = j ta thực hiện
  1. Phần tử thứ i nhỏ hơn key ta tăng i lên;
  2. Phần tử thứ j lớn hơn key ta giảm j xuống;
  3. Nếu i nhỏ hơn j ta hoán vị a[i] và a[j] đồng thời tăng i và giảm j;

  Lặp lại cho đến khi i > j
- Nếu l < j, ta thực hiện phân đoạn với mảng a từ l tới j.
- Nếu r > i, ta thực hiện phân đoạn với mảng a từ i lới r

Kết quả sau khi ta nhận thực hiện là 1 mảng được sắp xếp.

- Ví dụ: mảng A: 7 83 4 78 9 67

+ Ta có left = 0; right = 5; key = 4; i = 0; j = 5;

- i = 0 ta có a[i] > key ;
- j = 2 ta có a[i] = key;

+ Vì i < j nên ta Hoán vị a[i] và a[j], ta được mảng A = 4 83 7 78 9 67; i = 1; j = 1;

+ Vì i <= j nên tiếp tục so sánh a[i], a[j] với key

- i=1 ta có a[1] = 83 < key
- j= 0 ta có a[0] = 4 = key

+ Vì i > j nên không xảy ra hoán vị giữa a[i] và a[j];

- Do i > j nên vòng lặp kết thúc.

+Vì l = j nên không thực hiện phân đoạn của mảng a từ l tới j;

+Vì r > i nên ta thực hiện phân đoạn của mảng a từ i tới r;

+Ta có left = 1; right = 5; key = 78; A = 4 83 7 78 9 67 i = 1; j = 5;

- i = 1 ta có a[1] = 83 > key ;
- j = 5 ta có a[5] = 67 < key;

+ Vì i < j nên ta Hoán vị a[i] và a[j], ta được mảng A = 4 67 7 78 9 83; i = 2; j = 4;

+ Vì i <= j nên tiếp tục so sánh a[i], a[j] với key

- i=3 ta có a[3] = 78 = key
- j= 3 ta có a[3] = 78 = key

+Vì i = j nên ta hoán vị giữa a[i] và a[j]; i = 4; j = 2

- Do i > j nên vòng lặp kết thúc.

+ Vì l < j nên ta thực hiện phân đoạn của mảng a từ l tới j;

+ Vì r > i nên ta thực hiện phân đoạn của mảng a từ i tới r;

Thực hiện như vậy mảng A ta sẽ thay đổi như sau:

4	83	7	78	9	67
4	83	7	78	9	67
4	67	7	78	9	83
4	67	7	9	78	83
4	7	67	9	78	83
4	7	9	67	78	83
4	7	9	67	78	83
4	7	9	67	78	83

- *Độ phức tạp:*
  - Trường hợp tốt: O(nlog(n))
  - Trung bình: O(nlog(n))
  - Trường hợp xấu: O(n^2)
- Không gian bộ nhớ sử dụng: O(log(n))

8. Bubble Sort

- *Ý tưởng:* Thuật toán sắp xếp bubble sort thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự (số sau bé hơn số trước với trường hợp sắp xếp tăng dần) cho đến khi dãy số được sắp xếp
- *Step By Step:*
  - Bước 1: khai báo 1 biến ở đầu và 1 biến ở cuối dãy

- Bước 2: cho biến ở cuối dãy so với biến trước nó, nếu biến lớn hơn nằm trước thì hoán đổi (a[j] < a[j-1])
- Bước 3: check đến khi tới biến đầu bảng. Nếu tới thì biến đầu bảng sẽ dời lên 1 và biến 2 reset lại nằm ở cuối dãy
- Bước 4: lặp đến khi biến đầu không thể tăng quá n-1

- *Ví dụ minh họa:*
  - Giả sử chúng ta cần sắp xếp dãy số [2 1 9 7 5] này tăng dần
  - Lần lặp đầu tiên:
    (2 1 9 7 5) –> (2 1 9 5 7), Ở đây, thuật toán sẽ so sánh hai phần tử cuối, và đổi chỗ cho nhau do 7 > 5.
    (2 1 9 5 7) –> (2 1 5 9 7), Đổi chỗ do 9 > 5
    (2 1 5 9 7) –> (2 1 5 9 7), Ở đây, hai phần tử đang xét đã đúng thứ tự (1 < 5), vậy ta không cần đổi chỗ
    (2 1 5 9 7) –> (1 2 5 9 7), Đổi chỗ do 2 > 1
  - Lần lặp thứ 2:
    (1 2 5 9 7) –> (1 2 5 7 9) , Đổi chỗ do 9 > 7
    (1 2 5 7 9) –> (1 2 5 7 9)
    (1 2 5 7 9) –> (1 2 5 7 9)
  - Lần lặp thứ 3:
    (1 2 5 7 9) –> (1 2 5 7 9)
    (1 2 5 7 9) –> (1 2 5 7 9)
  - Lần lặp thứ 4:
    (1 2 5 7 9) –> (1 2 5 7 9)

- *Độ phức tạp thuật toán:*
  - Trường hợp tốt nhất: O(n)
  - Trung bình: O(n^2)
  - Trường hợp xấu nhất: O(n^2)

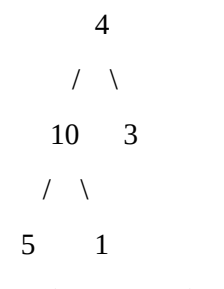
- Bộ nhớ: O(1)

9. Heap Sort

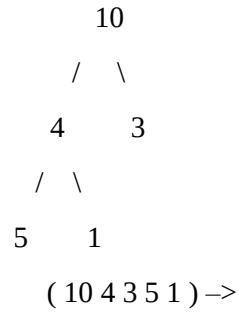
- *Ý tưởng:* Để tìm phần tử nhỏ nhất ở bước i, phương pháp sắp xếp chọn trực tiếp đã không tận dụng được các thông tin đã có được do các phép so sánh ở bước i-1. Phương pháp Heap Sort khắc phục được nhược điểm này

- *Step by Step:*
  - Bước 1: xây array thành max heap (node cha >= node con)
  - Bước 2: đổi vị trí đầu và cuối dãy (swap(a[0],a[n])) sau đó xóa đi node cuối
  - Bước 3: nếu số node còn lại > 1 thì lập lại bước 1

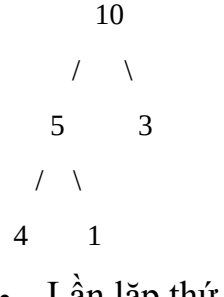
- *Ví dụ minh họa:*
  - Giả sử chúng ta cần sắp xếp dãy số [4 10 3 5 1] này tăng dần
  - Lần lặp đầu tiên:
    (4 10 3 5 1), xây heap



(4 10 3 5 1) –> (10 4 3 5 1), do 10 > 4



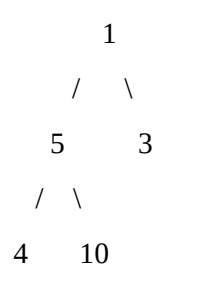
(10 4 3 5 1) –> (10 5 3 4 1), do 5 > 4



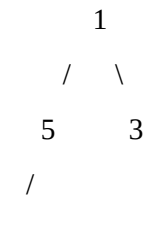
- Lần lặp thứ 2:

  i = n – 1 = 4

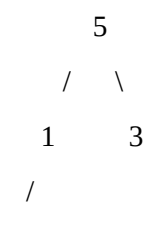
  (10 5 3 4 1) –> (1 5 3 4 10), sau khi đã thành max heap, hoán đổi biến vị trí đầu và cuối (swap(a[0], a[i]))



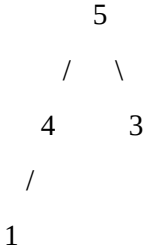
(1 5 3 4 10), xây heap không có phần tử cuối



(1 5 3 4 10) –> (5 1 3 4 10), do 5 > 1



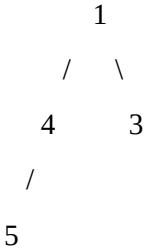
(5 1 3 4 10) –> (5 4 3 1 10), do 4 > 1



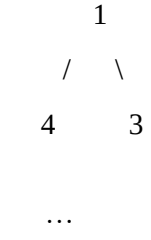
- Lần lặp thứ 3:

  i -= 1 = 3

  (5 4 3 1 10) –> (1 4 3 5 10), sau khi đã thành max heap, hoán đổi biến vị trí đầu và cuối (swap(a[0], a[i]))



(1 4 3 5 10), xây heap không có phần tử cuối



…

Đến khi chỉ còn 1 node 1

- *Độ phức tạp thuật toán:*
  - Trường hợp tốt nhất : dãy đã được sắp xếp sẵn=> độ phức tạp là O(1)
  - Trường hợp xấu nhất : O(nlog2n)
- Bộ nhớ: O(1)

10. Shell Sort

- *Ý tưởng*: Như 1 cách khác biểu diễn Insertion Sort, gap là 1 cách so sánh trên khoảng không gian lớn từ ½ chuỗi đến ¼, … cho đến khi những khoảng cần so sánh chỉ còn nằm kề nhau

- *Step by Step:*
  - Bước 1: đặt vòng lập gap = n/2
  - Bước 2: lưu 1 bảng sao của phần tử tại gạp
  - Bước 3: tại điểm xuất phát i ở gap (i=gap), nếu phần tử ở điểm đó so với điểm từ đó trừ gap lớn hơn thì hoán đổi (f =i, a[f] < a[f-gap]), lập đến khi ko giảm nhỏ hơn gap nữa
  - Bước 4: thế bảng sao tại j cuối cùng và tăng điểm xuất phát lên 1 (i+=1) và lập lại bước 2, 3 đến khi i=n
  - Bước 5: chia dôi gap (gap/=2) và lập lại bước 2, 3, 4 đến khi gap không thể chia nhỏ hơn (gap=1)

- *Ví dụ minh họa:*
  - Giả sử chúng ta cần sắp xếp dãy số [12 34 54 2 4] này tăng dần
  - Lần lặp đầu tiên:
    gap = n/2 = 5/2 = 2.5 = 2 (do int gap)
    i = gap = 2
    Temp = a[2] = 54
    (12 34 54 2 4) –> (12 34 54 2 4), do 12 < 54 nên không hoán đổi
    (12 34 54 2 4) –> (12 34 54 2 4), do temp đè lên cùng vị trí cũ

    i+1 = 3
    Temp = a[3] = 2
    (12 34 54 2 4) –> (12 34 54 34 4), do 34 > temp
    (12 34 54 34 4) –> (12 2 54 34 4), do mình không thể giảm thêm nên temp sẽ đè lên vị trí cuối đã thay đổi a[1]

    i+1 = 4
    Temp = a[4] = 4
    (12 2 54 34 4) –> (12 2 54 34 54),do 54 > temp

(12 2 54 34 54) –> (12 2 12 34 54), do 12 > temp
(12 2 12 34 54) –> (4 2 12 34 54), do mình không thể giảm thêm nên temp sẽ đè lên vị trí cuối đã thay đổi a[0]

- Lần lặp thứ 2:
  gap /= 2 = 2/2 = 1
  i = gap = 1
  Temp = a[1] = 2
  (4 2 12 34 54) –> (4 4 12 34 54), do 4 > temp
  (4 4 12 34 54) –> (2 4 12 34 54), temp đè lên a[0]
  i+1 = 2
  Temp = a[2] = 12
  (2 4 12 34 54) –> (2 4 12 34 54),
  (2 4 12 34 54) –> (2 4 12 34 54),
  (2 4 12 34 54) –> (2 4 12 34 54), do temp đè lên cùng vị trí cũ
  …
  Đến khi i = 4

- *Độ phức tạp thuật toán:*
  - trường hợp tốt nhất: O(nlogn)
  - trung bình: tùy thuộc gap
  - trường hợp xấu nhất: Nhỏ hơn hoặc bằng O(n^2)
- Bộ nhớ: O(1)

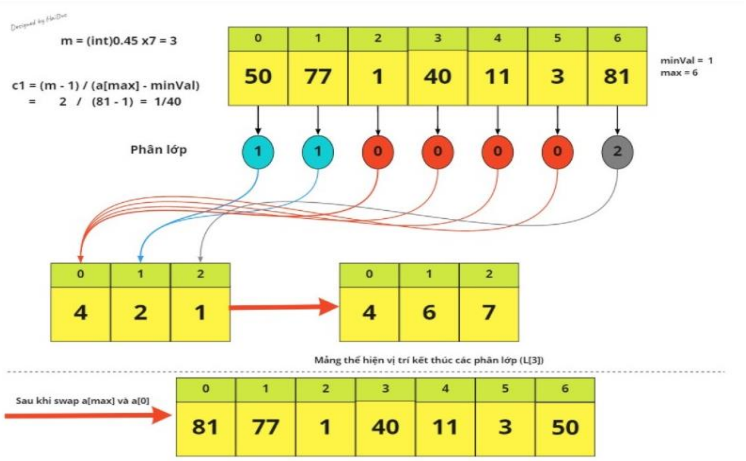
11. Flash Sort

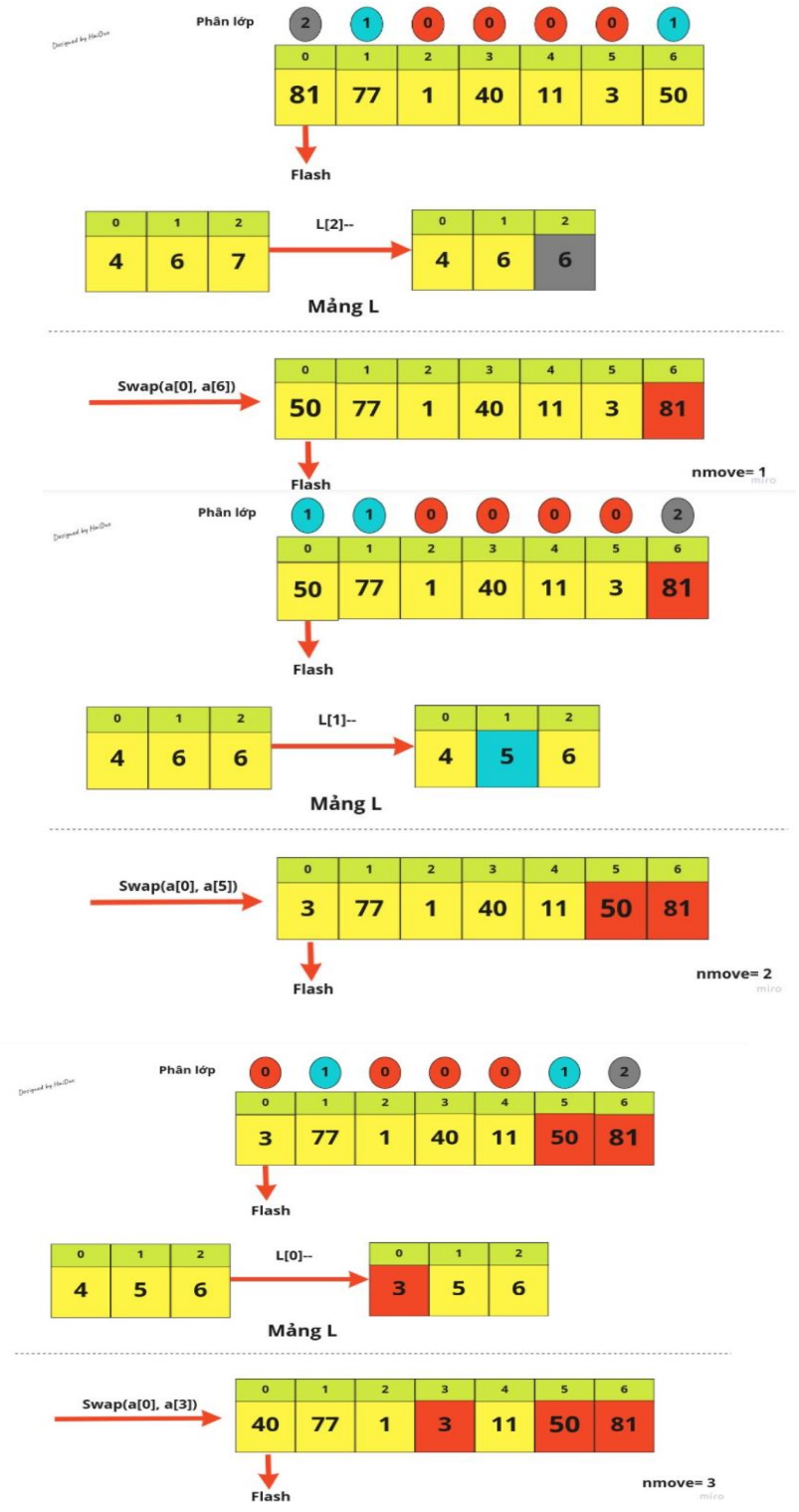
- *Ý tưởng:* Flash Sort bắt nguồn từ Bucket Sort, Flash Sort thực hiện việc sắp xếp một mảng thông qua 3 bước:
  1. Phân lớp dữ liệu, tức là dựa trên giả thiết dữ liệu tuân theo 1 phân bố nào đó, chẳng hạn phân bố đều, để tìm 1 công thức ước tính vị trí (lớp) của phần tử sau khi sắp xếp.
  2. Hoán vị 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.

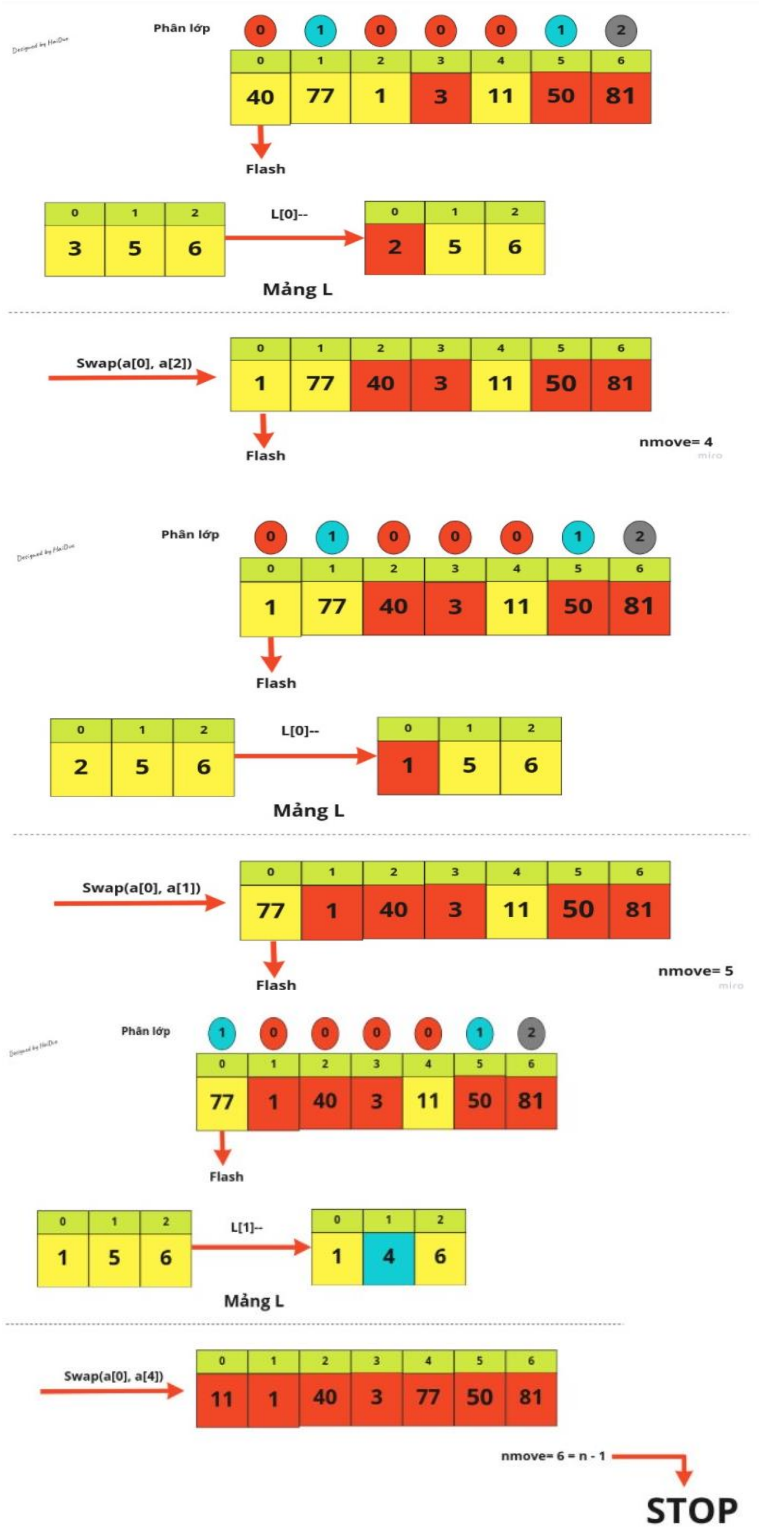
- *Step by Step:*
  - Phân lớp dữ liệu:
- Bước 1: Tính số lớp dữ liệu mảng bằng phần nguyên của 0.45 * chiều dài mảng
- Bước 2: Khởi tạo 1 mảng có chưa m lớp dữ liệu
- Bước 3: Tìm Min và Max của mảng
- Bước 4: Đếm số lượng phần tử các lớp theo quy luật, phần tử a[i] sẽ thuộc lớp k = int((m - 1) * (a[i]-minVal) / (a[max] - minVal)).
- Bước 5: 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).
  - Hoán vị toàn cục:

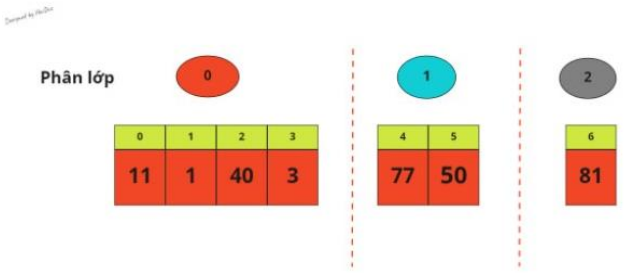
- Bước 6: move = 0(Số lần di chuyển), k = n – 1, c1 = int((m - 1) * (a[i]-minVal) / (a[max] - minVal)).
- Bước 7: Trong khi move < n – 1
- Bước 8: Đếm số K = c1 * (a[0] - Min);
- Bước 9: Hoán đổi giá trị a[0] và a[L[k]];
- Bước 10: L[k] = L[k] – 1, move = move + 1;
- Bước 11: Nếu move < n – 1 thì quay lại bước 8 nếu không thì qua bước 12.
  - Sắp xếp cục bộ:
- Bước 12: các phần tử của mảng sẽ về đúng với phân lớp của mình, ta dùng **Insertion Sort** để sắp xếp lại mảng này.

- *Ví dụ minh họa:*




Phân lớp
2 1 0 0 0 0 1
0 1 2 3 4 5 6
81 77 1 40 11 3 50
Flash
0 1 2
4 6 7
L[2]--
0 1 2
4 6 6
Mảng L
Swap(a[0], a[6])
0 1 2 3 4 5 6
50 77 1 40 11 3 81
Flash
nmove= 1
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
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


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
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
Phân lớp
1 0 0 0 0 1 2
0 1 2 3 4 5 6
77 1 40 3 11 50 81
Flash
0 1 2
1 5 6
L[1]--
0 1 2
1 4 6
Mảng L
Swap(a[0], a[4])
0 1 2 3 4 5 6
11 1 40 3 77 50 81
nmove= 6 = n - 1
STOP



Kết quả sau khi thực hiện Insertion Sort: {1, 3, 11, 40, 50, 77, 81}.

- *Độ phức tạp thuật toán:*
  - trường hợp tốt nhất: O(n)
  - trung bình: O(n + r)
  - trường hợp xấu nhất: O(n^2)
- Bộ nhớ: O(n)

EXPERIMENTAL RESULTS AND COMMENTS

1. Randomized Data

❖ ***Experimental Results Table***

RANDOMIZED DATA												
Data size	**Resulting Static**	**Radix Sort**	**Quick Sort**	**Shaker Sort**	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Counting Sort**	**Heap Sort**	**Selection Sort**	**Flash Sort**	**Shell Sort**
10000	*Running Time*	3	1	242	286	65	3	1	3	100	0	2
	Comparison	260.093	280.458	66.371.371	100.009.999	50.121.717	588.037	70.003	15.001	100.009.999	105.395	392.134
30000	*Running Time*	11	3	2.165	2.765	536	8	1	10	901	2	5
	Comparison	1.050.116	915.201	600.791.512	900.029.999	446.972.090	1.951.881	210.004	45.001	900.029.999	317.008	1.317.541
50000	*Running Time*	18	6	6.012	7.970	1.475	13	1	16	2.509	4	12
	Comparison	1.750.116	1.621.516	1.669.557.982	2.600.049.999	1.2444.458.761	3.405.185	332.770	75.001	2.500.049.999	522.994	2.637.158
100000	*Running Time*	35	11	24.670	32.682	6.277	27	2	34	10.009	8	26
	Comparison	3.500.116	3.379.091	6.656.559.140	10.000.099.999	5.000.669.341	7.210.855	632.771	150.001	10.000.099.999	1.035.570	5.759.217
300000	*Running Time*	106	36	224.616	293.300	54.465	90	5	116	91.908	26	87
	Comparison	10.500.116	10.802.820	60.017.595.481	90.000.299.999	45.051.416.181	23.520.615	1.832.772	450.001	90.000.299.999	3.288.689	198.864.440
500000	*Running Time*	151	62	634.642	826.260	152.146	158	8	204	255.619	43	157
	Comparison	17.500.116	18.964.721	166.742.132.259	250.000.499.999	124.972.045.808	40.626.661	3.032.772	750.002	250.000.499.999	5.546.659	372.900.052

❖ ***Running Time Chart***

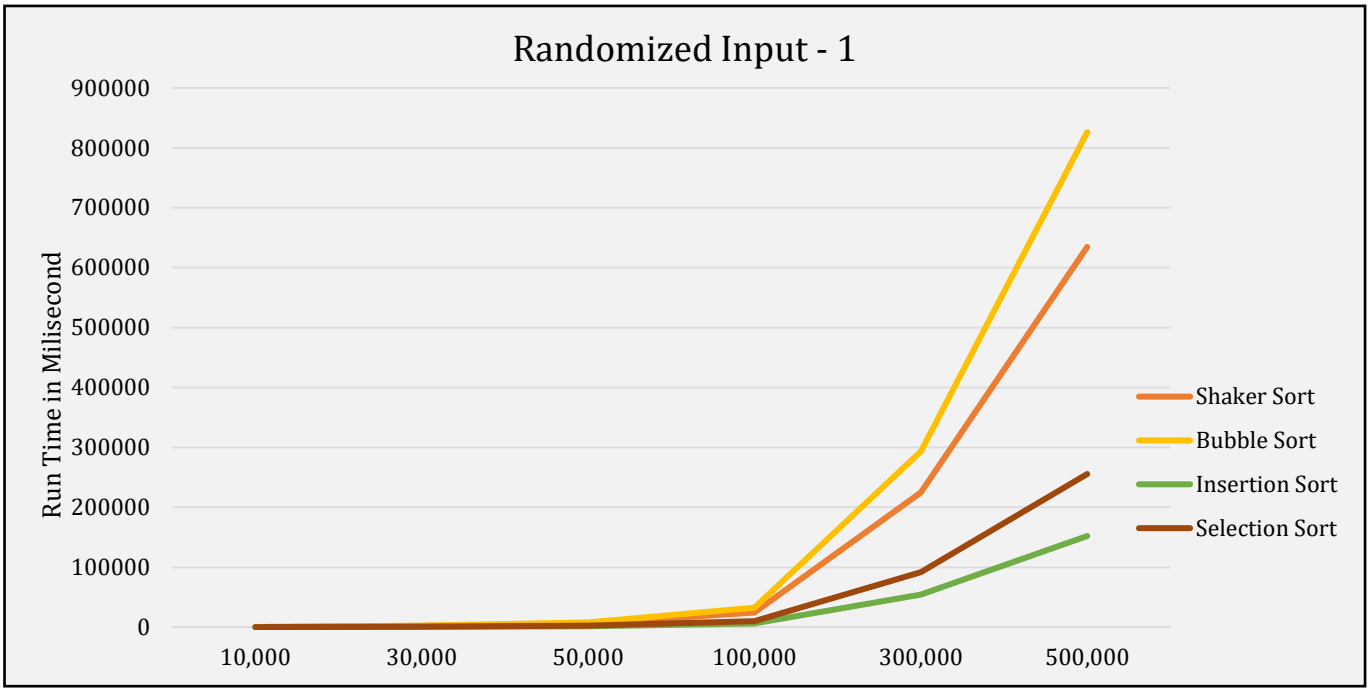


Chart 1. 1: line graph for visualizing the algorithms' running times on randomized input data – 1

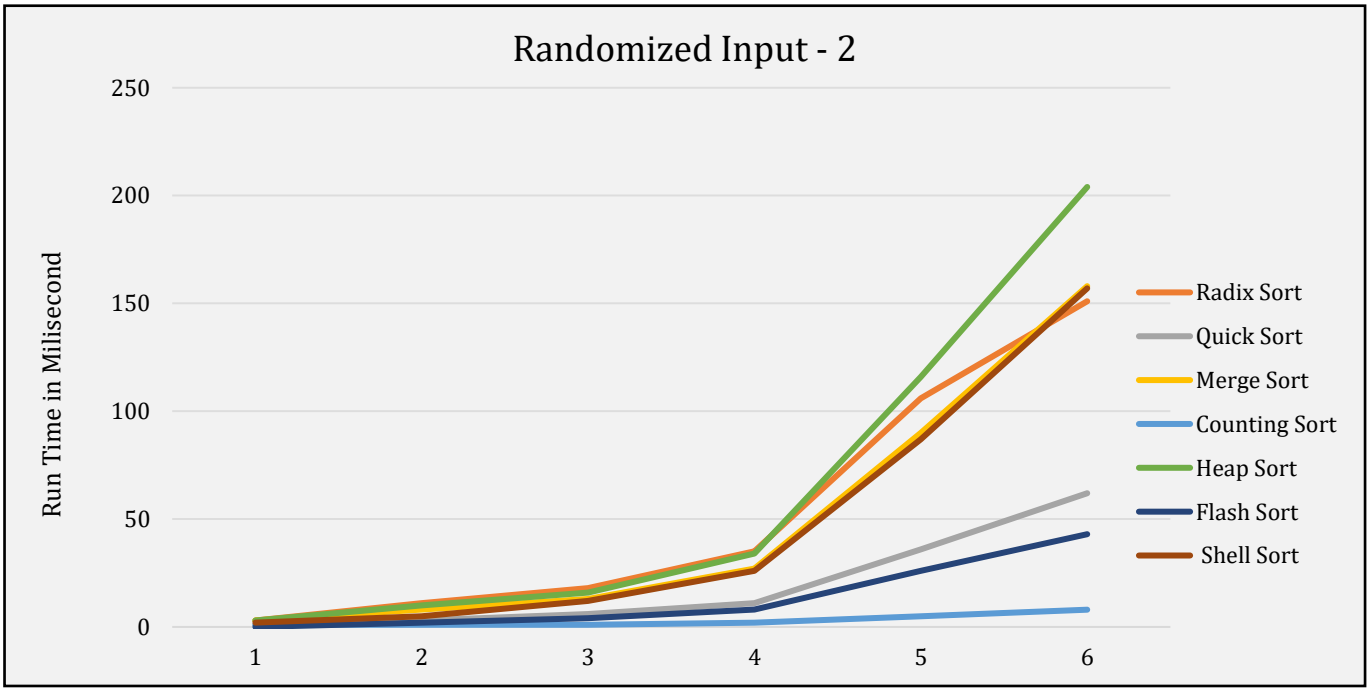


Chart 1. 2:line graph for visualizing the algorithms' running times on randomized input data – 2

❖ ***Comparisons Chart***

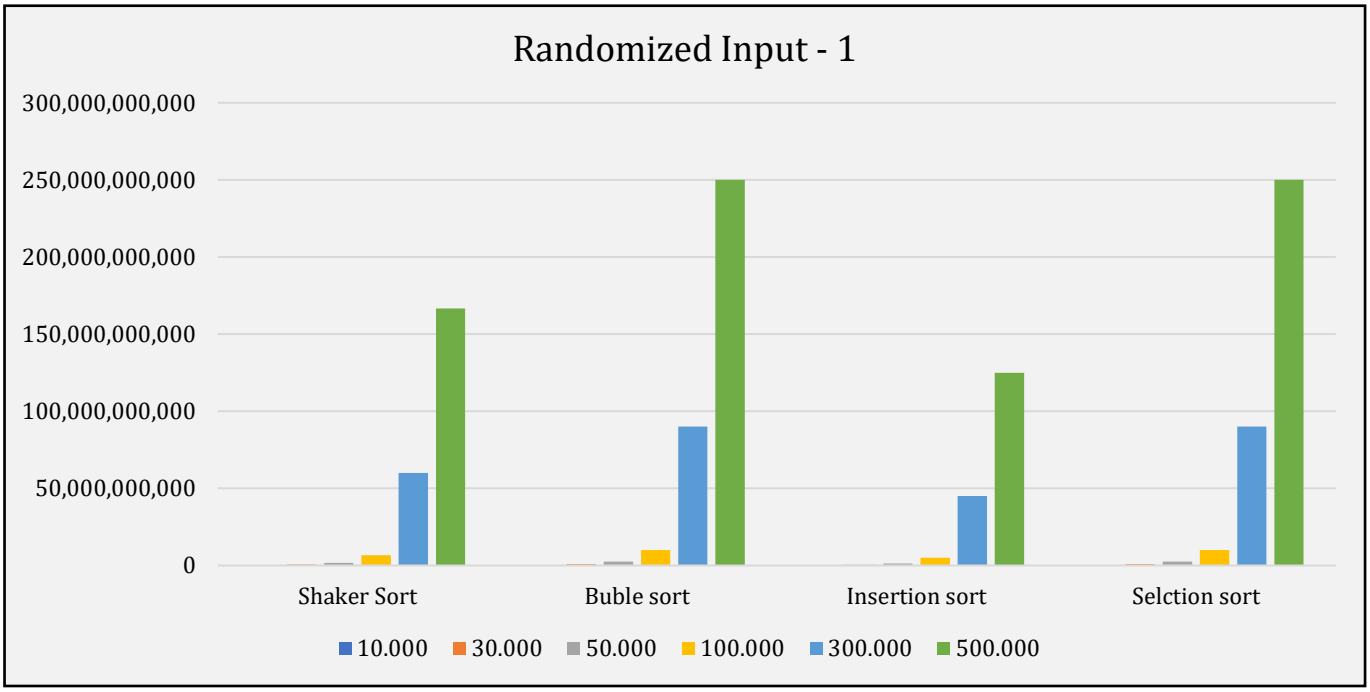


Chart 1. 3:bar chart for visualizing the algorithms' numbers of comparisons on randomized input data – 1

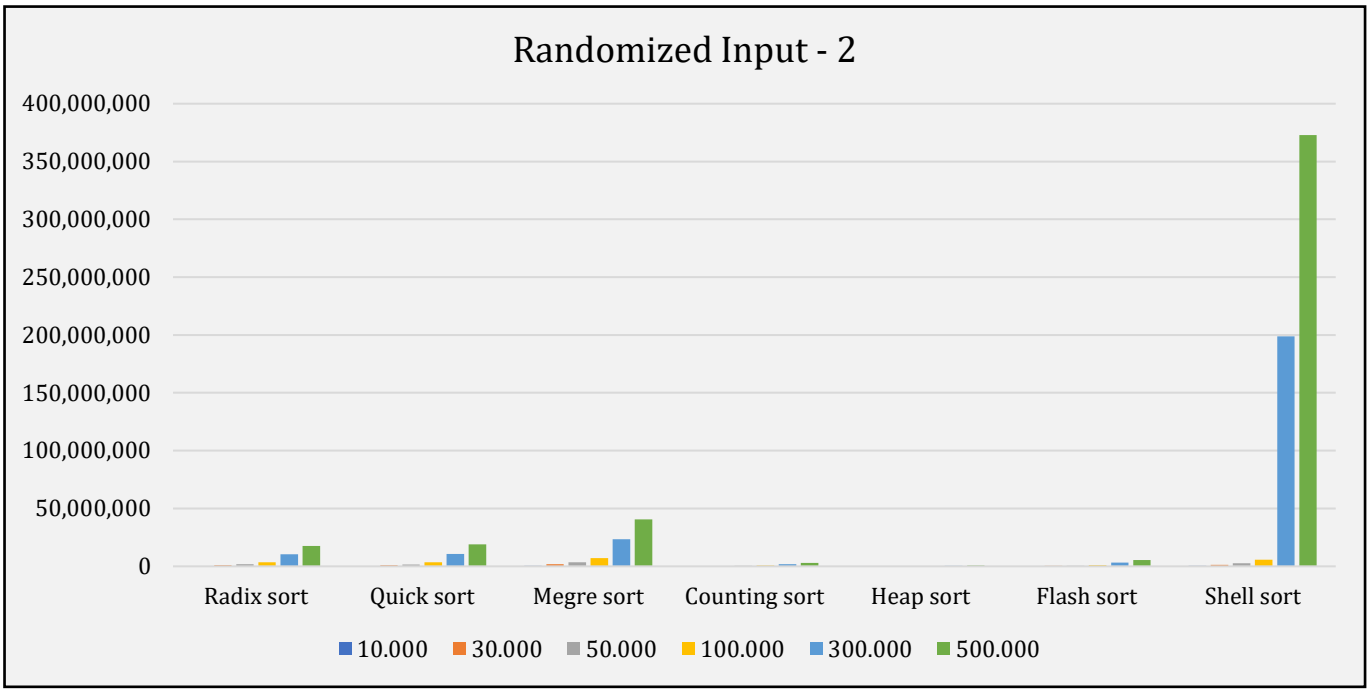


Chart 1. 4:bar chart for visualizing the algorithms' numbers of comparisons on randomized input data-2

❖ **Comment**

- The fastest algorithm: Counting Sort
- The Slowest algorithm: Bubble Sort

Counting Sort, Flash Sort are the fastest running algorithms (<50ms/500,000 Data size). Next is Quick Sort, Shell Sort, Merge Sort, Radix Sort and Heap Sort with relatively fast speed (<= 200 ms/ 500,000 Data size). Finally, the slowest are Insertion Sort, Selection Sort, Shaker Sort, Bubble Sort with runtime (>150,000ms & <830,000ms/500,000 Data size).

- The most comparisons algorithm: Bubble Sort & Selection Sort
- The least comparisons algorithm: Heap Sort

Heap Sort, Counting Sort are the algorithms with the fewest comparisons (<100,000/10,000 Data size). Next is Flash Sort, Quick Sort, Radix Sort, Shell Sort, Merge Sort with an average number of comparisons (<600,000/10,000 Data size). Finally, the most comparisons are Selection Sort, Shaker Sort, Bubble Sort, Insertion Sort with the number of comparisons (>50,000,000/10,000 Data size)

2. Nearly Sorted Data

❖ ***Experimental Results Table***

NEARLY SORTED DATA												
Data size	**Resuilting Static**	**Radix Sort**	**Quick Sort**	**Shaker Sort**	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Counting Sort**	**Heap Sort**	**Selection Sort**	**Flash Sort**	**Shell Sort**
10000	*Running Time*	4	1	1	110	1	2	1	2	99	0	1
	Comparison	260.093	160.895	178.218	100.009.999	23.134	512.187	70.004	15.001	100.009.999	109.989	262.703
30000	*Running Time*	13	1	2	969	1	6	1	8	904	2	2
	Comparison	1.050.116	518.360	608.119	900.029.999	490.738	1.664.119	210.004	45.001	900.029.999	329.991	854.189
50000	*Running Time*	17	4	2	2.697	1	10	1	14	2.518	3	4
	Comparison	1.750.116	1.655.987	958.991	250.049.999	611.086	2.878.999	350.004	75.001	2.500.049.999	549.990	1.510.701
100000	*Running Time*	30	4	3	10.746	2	21	2	30	10.178	6	7
	Comparison	3.500.116	1.993.266	1.716.240	10.000.099.999	857.202	5.942.631	700.004	150.001	10.000.099.999	1.099.991	3.077.581
300000	*Running Time*	117	12	4	98.674	3	64	5	98	91.446	18	25
	Comparison	13.500.139	6.227.188	1.716.240	90.000.299.999	1.451.058	19.049.167	2.100.004	45.001	90.000.299.999	3.299.990	10.270.573
500000	*Running Time*	193	19	5	271.544	4	106	9	170	252.155	31	41
	Comparison	22.500.139	10.572.916	2.516.812	250.000.499.999	1.815.562	32.621.831	3.500.004	75.001	240.000.499.999	5.499.989	17.081.047

❖ ***Running Time Chart***

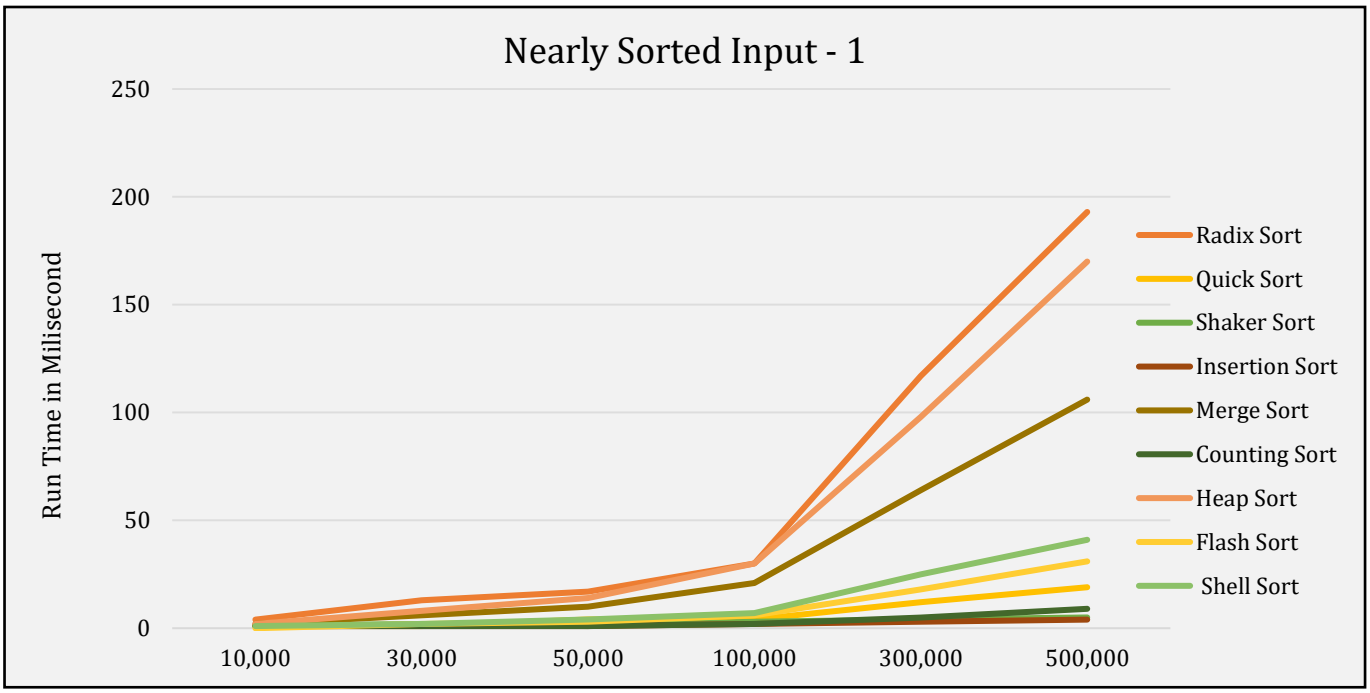


Chart 2. 1: line graph for visualizing the algorithms' running times on Nearly Sorted input data – 1

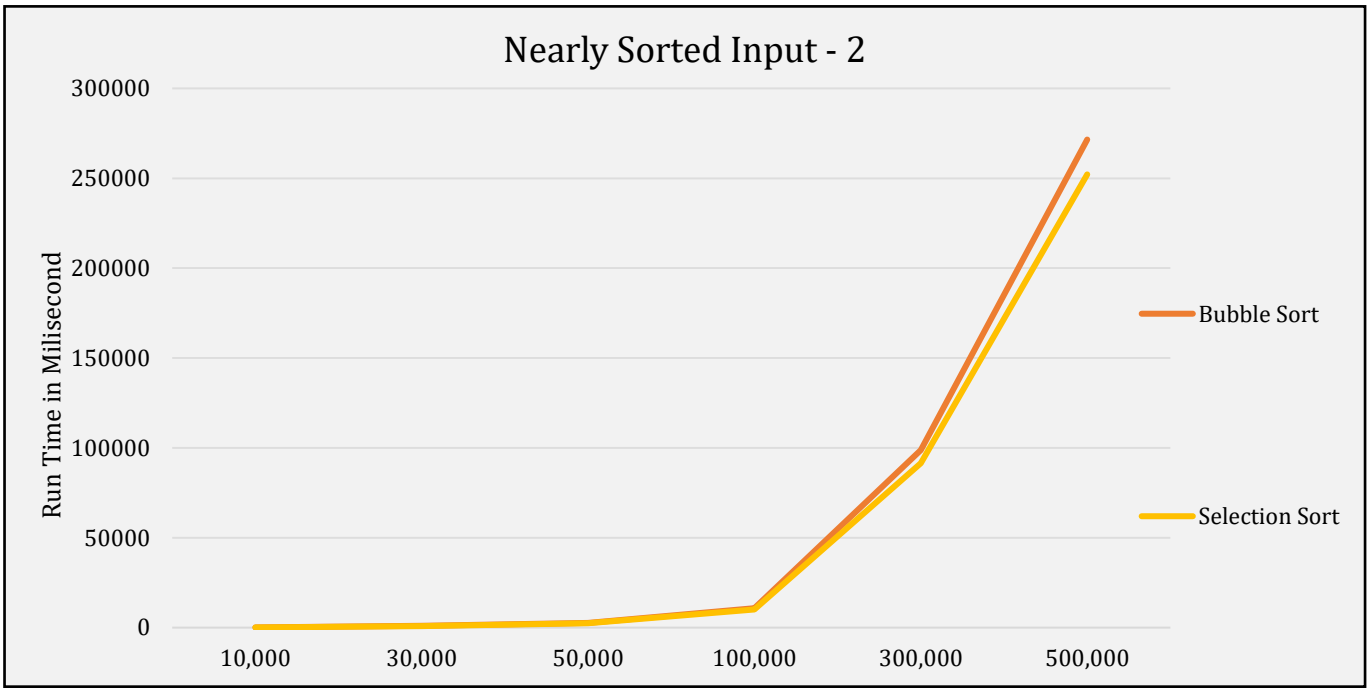


Chart 2. 2: line graph for visualizing the algorithms' running times on Nearly Sorted input data – 2

❖ ***Comparisons Chart***

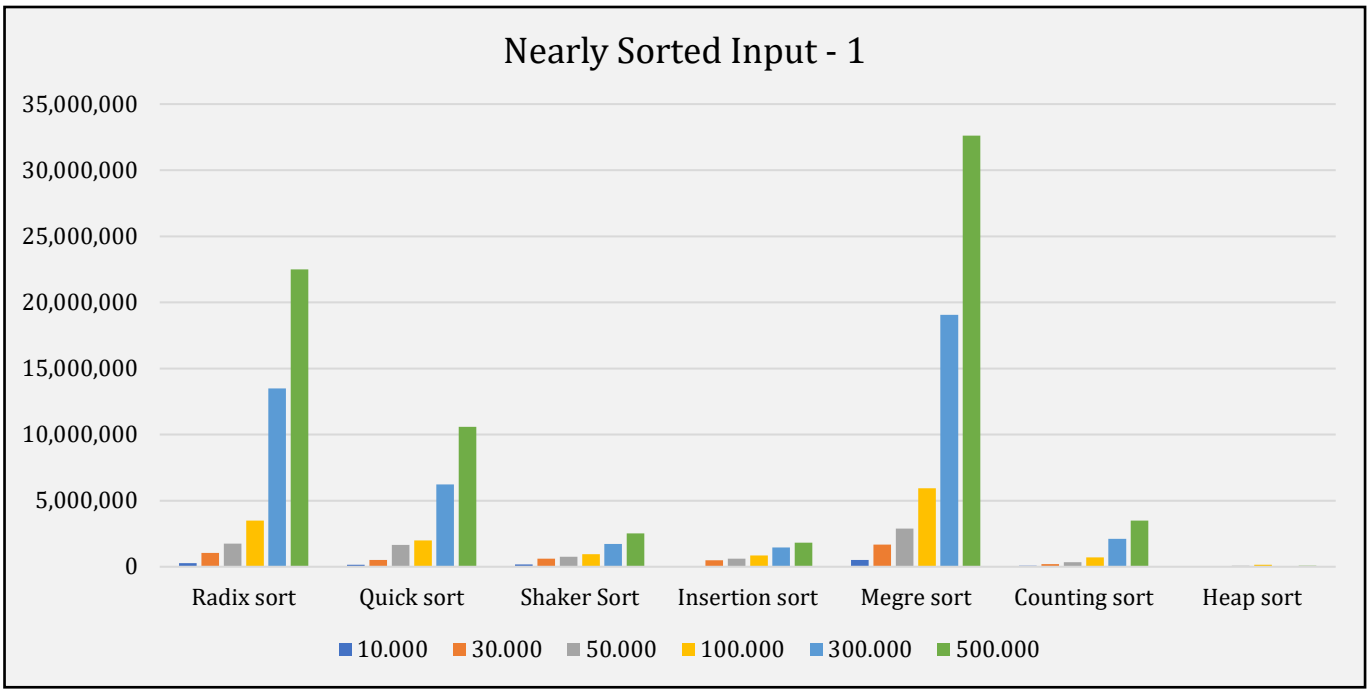


Chart 2. 3: bar chart for visualizing the algorithms' numbers of comparisons on nearly sorted input data – 1

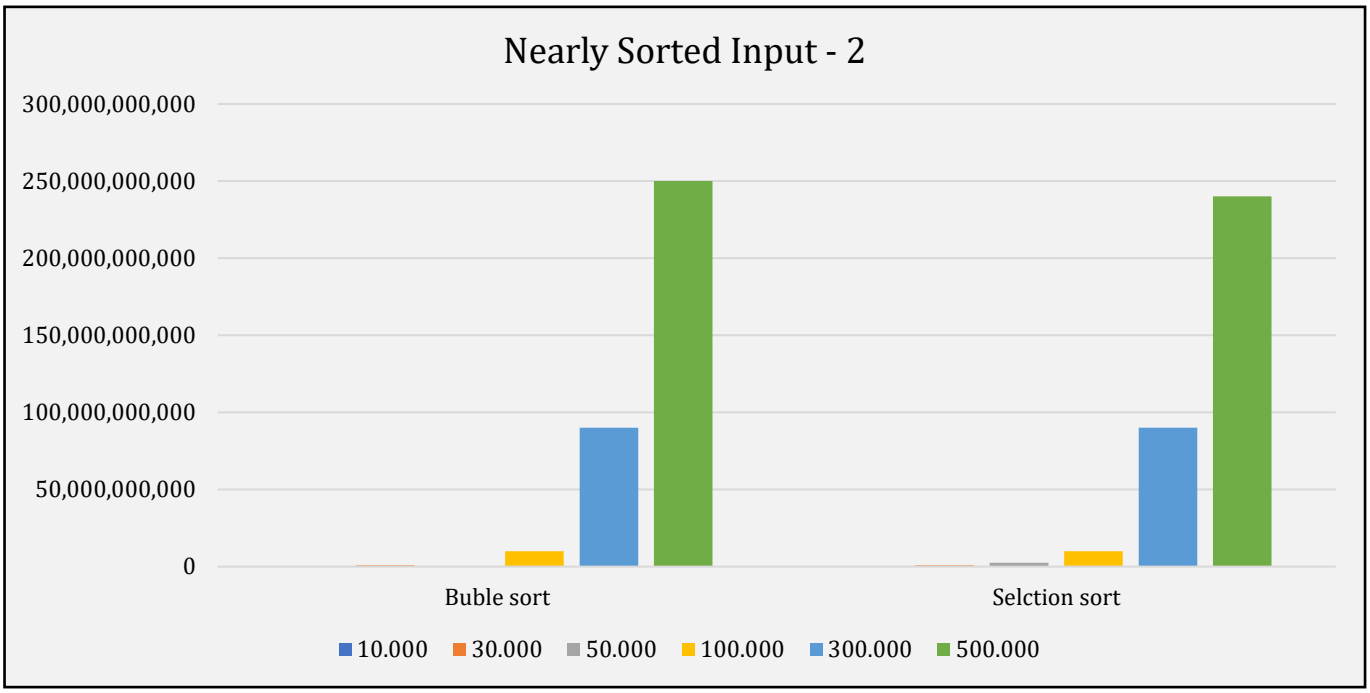


Chart 2. 4: bar chart for visualizing the algorithms' numbers of comparisons on nearly sorted input data – 2

❖ **Comment**

- The fastest algorithm: Insertion Sort
- The Slowest algorithm: Bubble Sort

Insertion Sort, Counting Sort, Shaker Sort, Quick Short, Flash Short, Shell Sort are the fastest running algorithms (<50ms/500,000 Data size). Next is Merge Sort, Heap Sort, Radix Sort with relatively fast speed (< 200 ms/ 500,000 Data size). Finally, the slowest are Selection Sort, Bubble Sort with runtime (>250,000ms/500,000 Data size).

- The most comparisons algorithm: Bubble Sort
- The least comparisons algorithm: Heap Sort

Heap Sort, Insertion Sort, Counting Sort are the algorithms with the fewest comparisons (<100,000/10,000 Data size). Next is Flash Sort, Quick Sort, Radix Sort, Shell Sort, Merge Sort, Shaker Sort with an average number of comparisons (<520,000/10,000 Data size). Finally, the most comparisons are Selection Sort, Bubble Sort with the number of comparisons (>100,000,000/10,000 Data size)

3. Sorted Data

❖ ***Experimental Results Table***

SORTED DATA												
Data size	**Resuilting Static**	**Radix Sort**	**Quick Sort**	**Shaker Sort**	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Counting Sort**	**Heap Sort**	**Selection Sort**	**Flash Sort**	**Shell Sort**
10000	*Running Time*	3	1	0	124	0	2	1	3	122	1	1
	Comparison	260,093	160,903	40	100.009.999	29,998	485,241	7,004	15,001	100.009.999	109,999	240. 037
30000	*Running Time*	10	0	0	1,109	0	7	1	9	1,071	2	2
	Comparison	1.050.116	518,312	120	900.029.999	89,998	1.589.913	210,004	45,001	900.029.999	329,999	780,043
50000	*Running Time*	18	2	0	3,227	0	13	1	15	3,027	3	4
	Comparison	1.750.116	946,617	200	2.500.049.999	149,998	2.772.825	350,004	75,001	2.500.049.999	549,999	1.400.043
100000	*Running Time*	36	4	0	12,429	0	27	2	37	12,327	7	10
	Comparison	3.500.116	1.993.226	400	10.000.099.999	299998	5.845.657	700,004	150,001	10.000.099.999	1.099.999	3.000.045
300000	*Running Time*	141	15	2	120,902	3	91	7	112	108,663	23	25
	Comparison	13.500.139	6.227.156	1.200.000	90.000.299.999	899,998	18.945.945	2.100.004	450,001	90.000.299.999	3.299.999	10.200.053
500000	*Running Time*	253	33	3	518,483	7	146	11	203	307,669	35	45
	Comparison	22.500.139	10.572.876	2.000.000	250.000.499.999	1.499.998	32.517.849	3.500.004	750,001	250.000.499.999	5.320.003	17.000.051

❖ ***Running Time Chart***

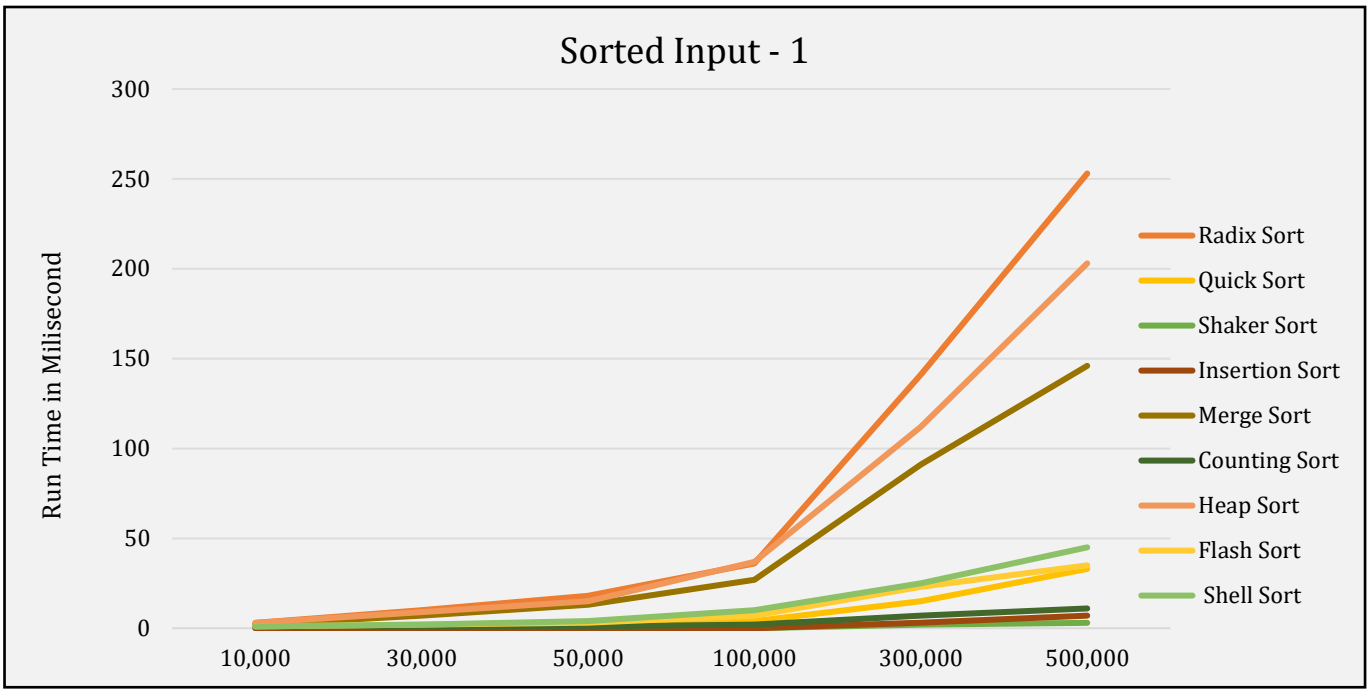


Chart 3. 1: line graph for visualizing the algorithms' running times on Sorted input data – 1

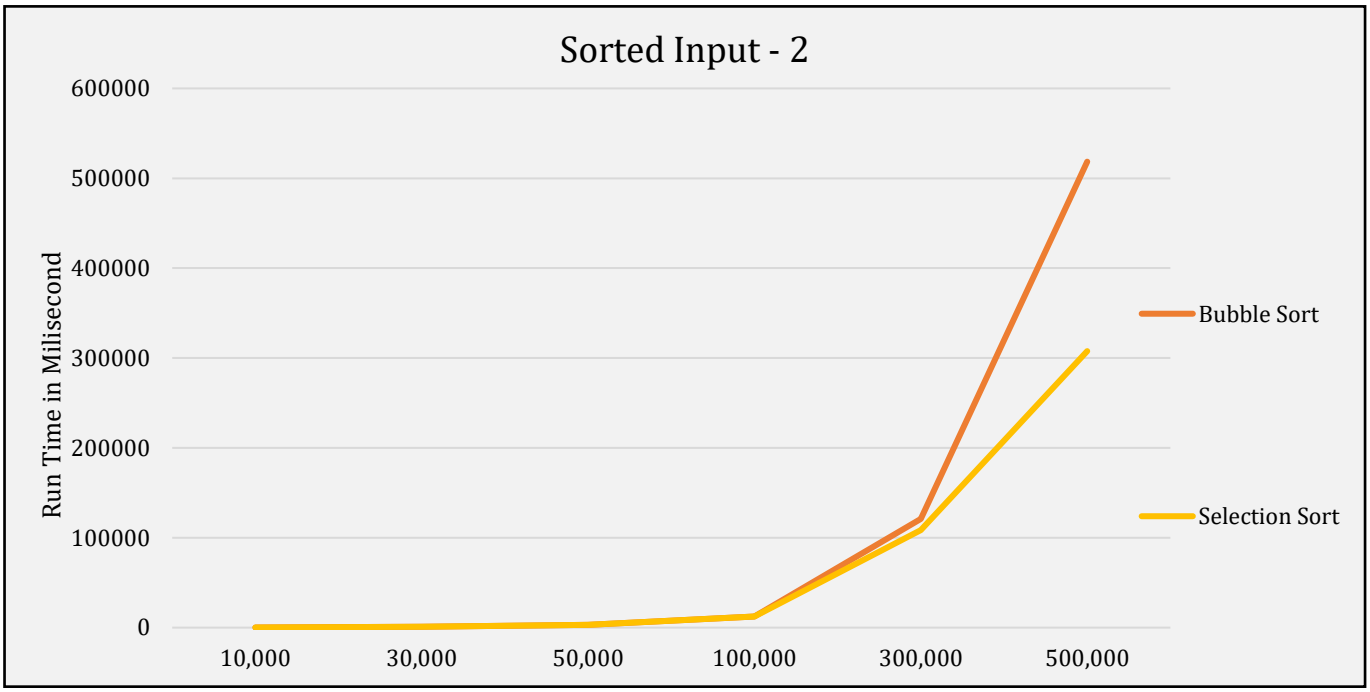


Chart 3. 2: line graph for visualizing the algorithms' running times on Sorted input data – 2

❖ ***Comparisons Chart***

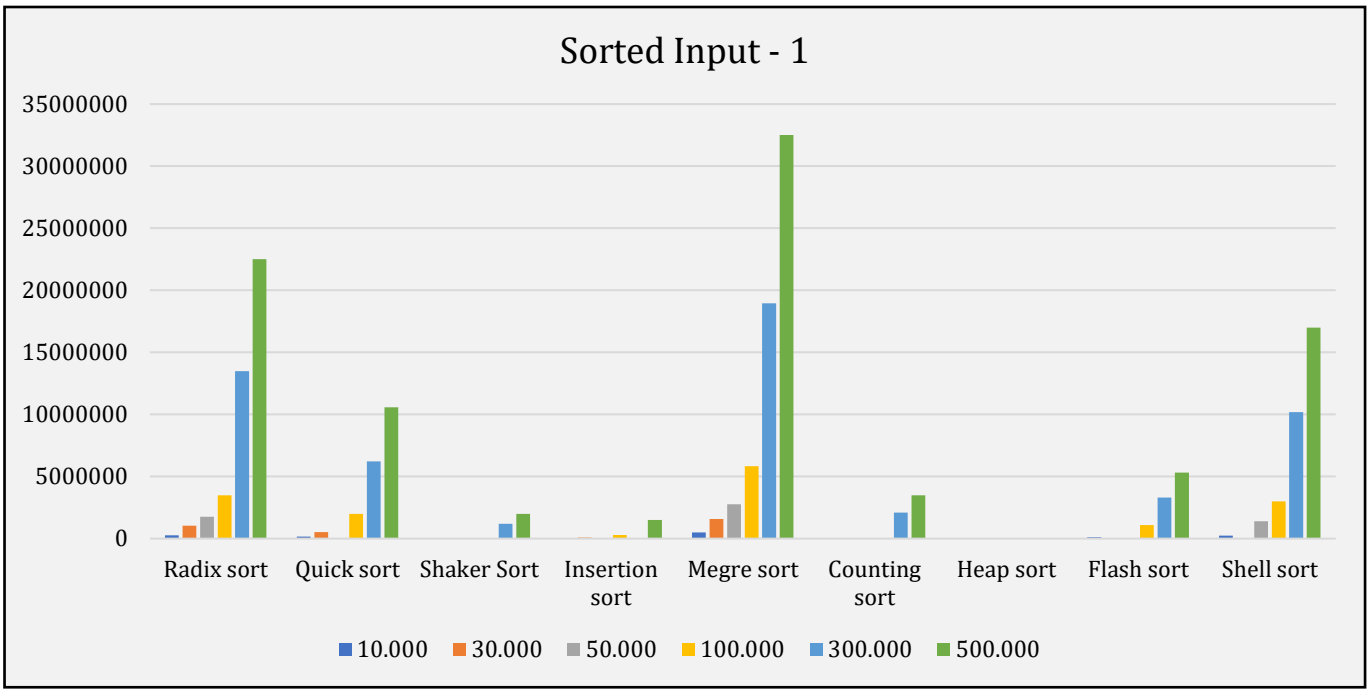


Chart 3. 3: bar chart for visualizing the algorithms' numbers of comparisons on sorted input data – 1

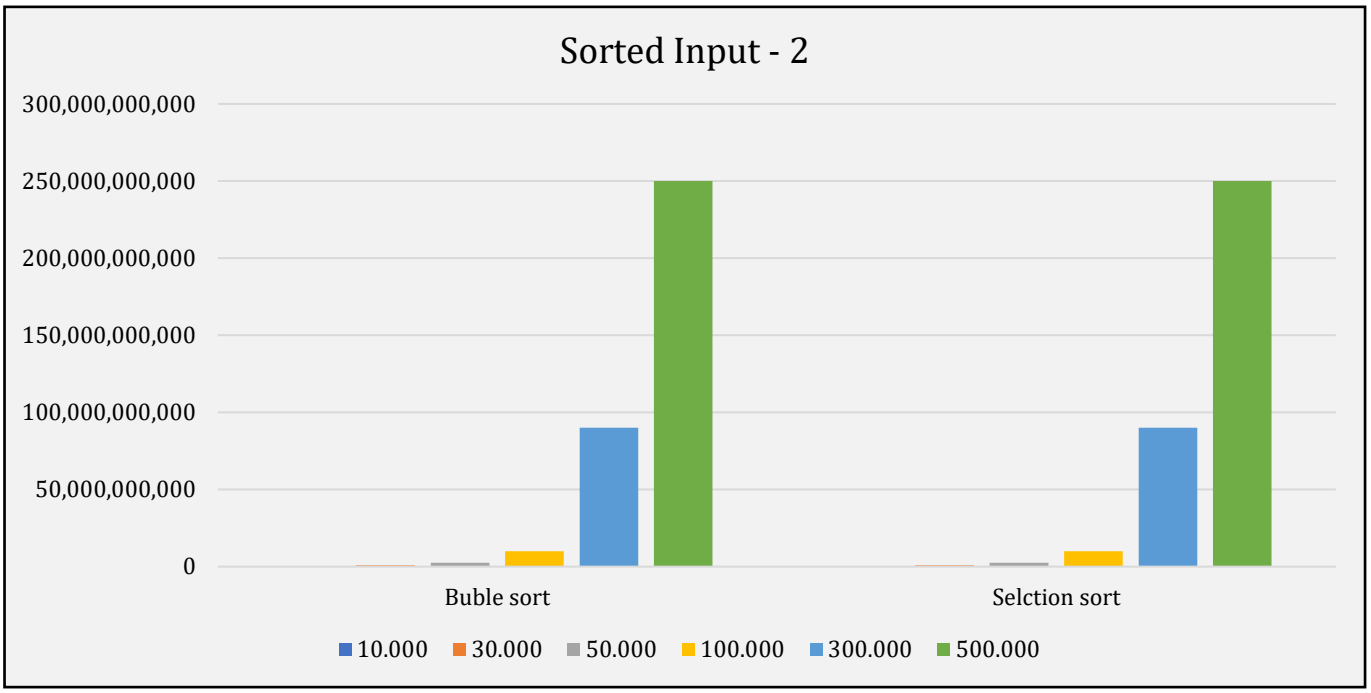


Chart 3. 4: bar chart for visualizing the algorithms' numbers of comparisons on sorted input data – 2

❖ **Comment**

- The fastest algorithm: Shaker Sort
- The Slowest algorithm: Bubble Sort

Shaker Sort, Insertion Sort, Counting Sort, Quick Short, Flash Short, Shell Sort are the fastest running algorithms (<50ms/500,000 Data size). Next is Merge Sort, Heap Sort, Radix Sort with relatively fast speed (<= 200 ms/ 500,000 Data size). Finally, the slowest are Selection Sort, Bubble Sort with runtime (>250,000ms/500,000 Data size).

- The most comparisons algorithm: Heap Sort
- The least comparisons algorithm: Bubble Sort & Selection Sort

Heap Sort, Insertion Sort, Counting Sort are the algorithms with the fewest comparisons (<100,000/10,000 Data size). Next is Flash Sort, Quick Sort, Radix Sort, Shell Sort, Merge Sort, with an average number of comparisons (<520,000/10,000 Data size). Finally, the most comparisons are Selection Sort, Bubble Sort with the number of comparisons (>100,000,000/10,000 Data size). Shaker Sort has a good performance at the stage <100,000 Data size (400/100,000 Data size). But clearly lost to Heap Sort in the later stages. shows that shaker sort runs unstable

4. Reverse Sorted Data

❖ ***Experimental Results Table***

REVERSE SORTED DATA												
Data size	**Resuilting Static**	**Radix Sort**	**Quick Sort**	**Shaker Sort**	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Counting Sort**	**Heap Sort**	**Selection Sort**	**Flash Sort**	**Shell Sort**
10000	*Running Time*	5	0	366	374	144	3	0	3	121	1	1
	Comparison	26,0093	170,88	100.024.999	100.009.999	100.009.999	476,441	70,004	15,001	100.009.999	106,403	302,597
30000	*Running Time*	11	2	3,279	3,064	1,271	7	1	8	1,101	2	3
	Comparison	1.050.116	548,322	900.074.999	900.029.999	900.029.999	1.573.465	210,004	45,001	900.029.999	319,203	987,035
50000	*Running Time*	19	2	9,375	11.005	3,788	13	1	15	3,108	3	5
	Comparison	1.750.116	996,628	2.500.124.999	2.500.049.999	2.500.049.999	2.733.945	350,004	75,001	2.500.049.999	532,003	1.797.323
100000	*Running Time*	39	4	37,328	33,593	16,636	26	2	34	12,477	7	12
	Comparison	3.500.116	2.093.238	10.000.249.999	10.000.099.999	10.000.099.999	5.767.897	700,004	150,001	10.000.099.999	1.064.003	3.844.605
300000	*Running Time*	137	11	332,866	306,942	157,045	85	5	112	110,571	21	35
	Comparison	13.500.139	6.527.178	90.000.749.999	90.000.299.999	90.000.299.999	18.708.313	2.100.004	450,001	90.000.299.999	3.192.003	12.700.933
500000	*Running Time*	258	21	936,576	471,764	458,663	143	10	193	315,072	36	61
	Comparison	22.500.139	11.072.890	250.001.249.999	250.000.499.999	250.000.499.999	32.336.409	3.500.004	750,001	250.000.499.999	5.320.003	21.428.803

❖ ***Running Time Chart***

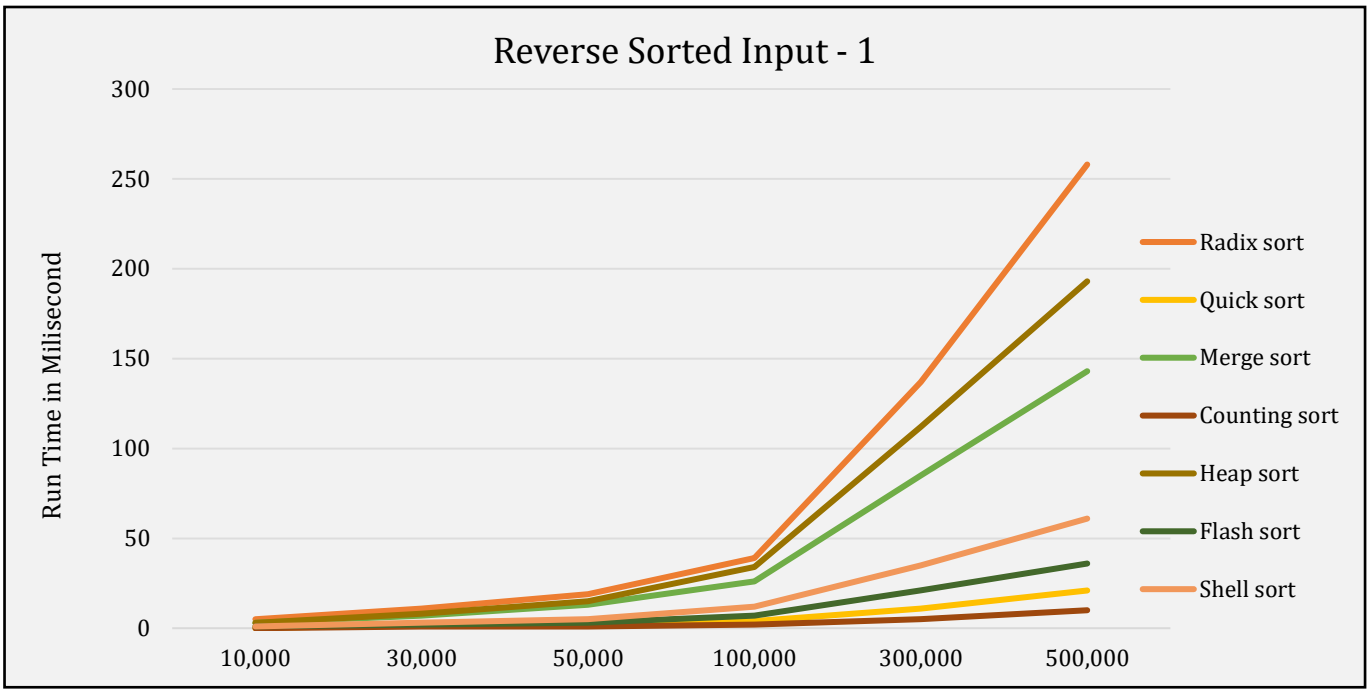


Chart 4. 1: line graph for visualizing the algorithms' running times on Reverse Sorted input data – 1

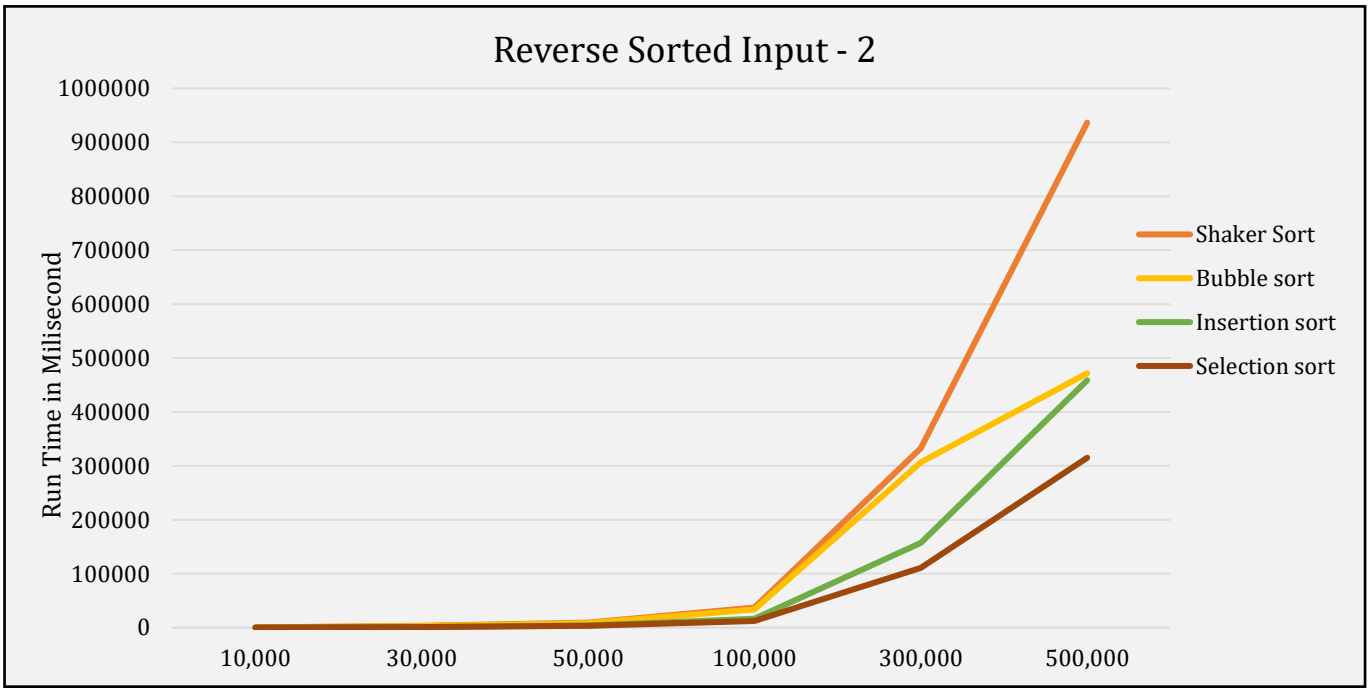


Chart 4. 2: line graph for visualizing the algorithms' running times on Reverse Sorted input data – 2

❖ ***Comparisons Chart***

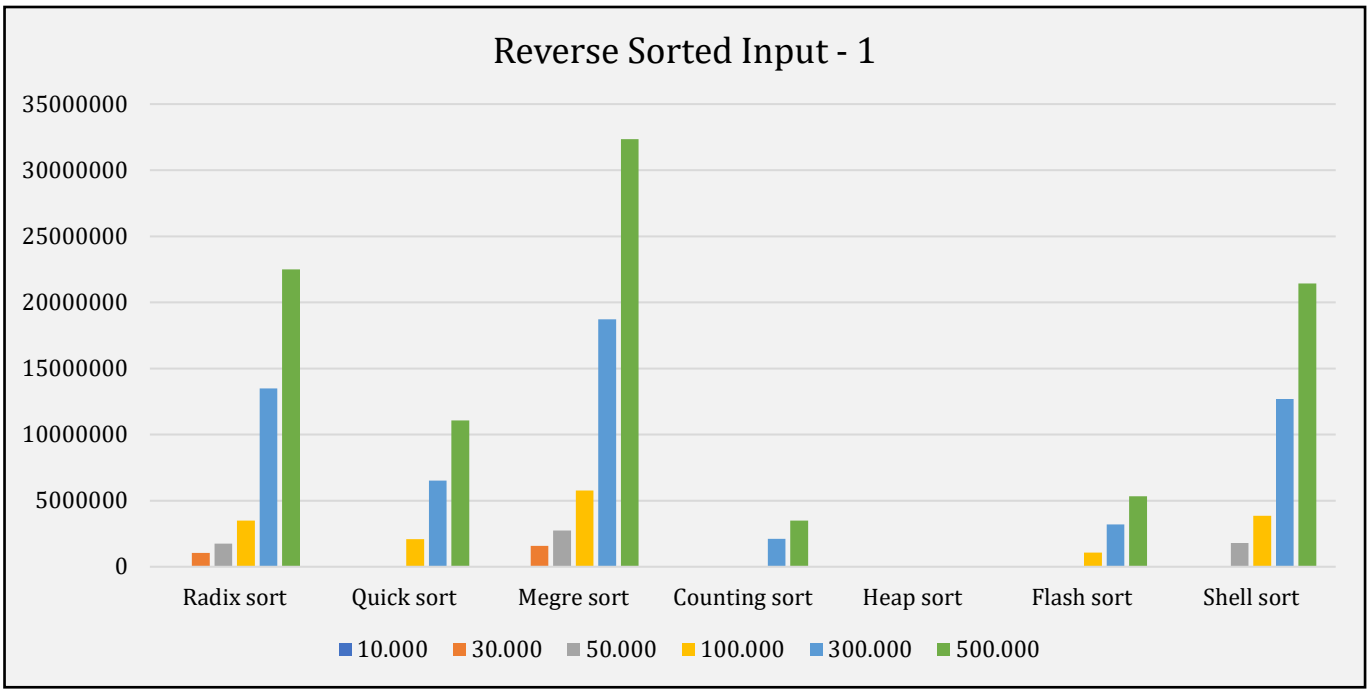


Chart 4. 3: bar chart for visualizing the algorithms' numbers of comparisons on reverse sorted input data – 1

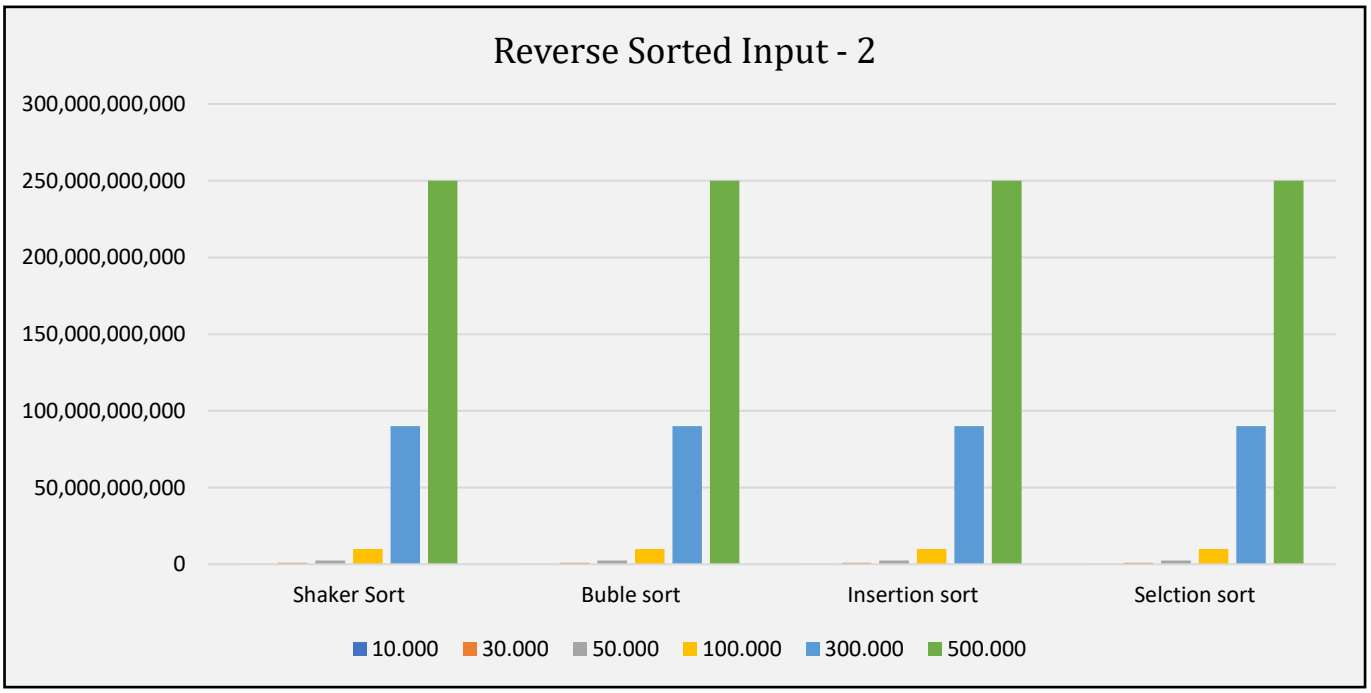


Chart 4. 4: bar chart for visualizing the algorithms' numbers of comparisons on reverse sorted input data – 2

❖ **Comment**

- The fastest algorithm: Counting Sort
- The Slowest algorithm: Bubble Sort

Counting Sort, Quick Short, Flash Short are the fastest running algorithms (<50ms/500,000 Data size). Next is Shell Sort, Merge Sort, Heap Sort, Radix Sort with relatively fast speed (< 200 ms/ 500,000 Data size). Finally, the slowest are Selection Sort, Bubble Sort, Insertion Sort with runtime (>310,000ms/500,000 Data size).

- The most comparisons algorithm: Heap Sort
- The least comparisons algorithm: Bubble Sort, Selection Sort & Insertion Sort

Heap Sort, Counting Sort are the algorithms with the fewest comparisons (<100,000/10,000 Data size). Next is Flash Sort, Quick Sort, Radix Sort, Shell Sort, Merge Sort with an average number of comparisons (<=300,000/10,000 Data size). Finally, the most comparisons are Selection Sort, Bubble Sort, Insertion Sort, Shaker Sort with the number of comparisons (>100,000,000/10,000 Data size)

PROJECT ORGANIZATION AND PROGRAMMING NOTES

15 File:

- DataGenarator.cpp
- Header.h
- Source.cpp
- Main.cpp
- Selection_Sort.cpp
- Bubble_Sort.cpp
- Shaker_Sort.cpp
- Quick_Sort.cpp
- Insertion_Sort.cpp
- Flash_Sort.cpp
- Shell_Sort.cpp
- Merge_Sort.cpp
- Counting_Sort.cpp
- Radix_Sort.cpp
- Heap_Sort.cpp

- **Organized our source code :** *contains the command code and links the sort.cpp files together*
  - Check string is a number *//check, identify between command 1 and command 3*
  - Copy Array *//create 2 arrays to run sort for running time and count comparisons*
  - Create an array
  - Store array to file
  - Run Sort *//run sort to calculate running time and count comparisons*
  - *Output screen*
  - 5 command
- **Organized our main code :**
  - categorize the command and run it using the command line parameter
- **Organized our DataGenarator code :** Taken form Moodle
- **Organized our header code:** Library declaration and code callback
  - **Libraies:**
  - Iostream
  - Fstream
  - Cmath
  - Time.h
  - Ctime
  - String.h
  - Cstring
  - Vector
  - Sstream

❖ Data Structures used: array list

LIST OF REFERENCES

- https://www.stdio.vn/giai-thuat-lap-trinh/merge-sort-u1Ti3U
- https://dnmtechs.com/thuat-toan-sap-xep-tron-merge-sort
- https://nguyenvanhieu.vn/thuat-toan-sap-xep-merge-sort/
- https://www.geeksforgeeks.org/counting-sort/
- https://thuvienkhoahoc.net/giai-thuat-sap-xep-selection-sort-chi-tiet-nhat.html
- https://www.geeksforgeeks.org/selection-sort/
- https://dnmtechs.com/sap-xep-theo-co-so-radix-sort/
- https://www.stdio.vn/giai-thuat-lap-trinh/distribution-sort-radix-sort-vqu1H1
- https://tek4.vn/thuat-toan-sap-xep-theo-co-so-radix-sort/
- https://www.geeksforgeeks.org/radix-sort/
- https://clbketnoitre.wordpress.com/2019/02/25/thuat-toan-sap-xep-shakersort/
- https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U
- https://nguyenvanhieu.vn/thuat-toan-sap-xep-chen/
- https://itzone.com.vn/vi/article/cac-thuat-toan-sap-xep-co-ban/
- https://nguyenvanhieu.vn/thuat-toan-sap-xep-chen/
- https://www.stdio.vn/giai-thuat-lap-trinh/quick-sort-hlmLf1
- https://itzone.com.vn/vi/article/cac-thuat-toan-sap-xep-co-ban/
- https://sites.google.com/site/nhatnamcpt/thu-thuat-tin-hoc-1/lap-trinh/giai-thuat/7-thuat-toan-merge-sort?tmpl=%2Fsystem%2Fapp%2Ftemplates%2Fprint%2F&showPrintDialog=1
- https://nguyenvanhieu.vn/thuat-toan-sap-xep-bubble-sort/
- https://codelearn.io/sharing/cac-thuat-toan-sap-xep-trong-cpp
- https://vi.wikipedia.org/wiki/Sắp_xếp_nổi_bọt
- https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot-nhat
- https://sites.google.com/site/nguoisaigonblog/giai-thuat-sap-xep-tinh-te/giai-thuat-heapsort-dan-tri
- https://www.geeksforgeeks.org/heap-sort/
- https://vimentor.com/vi/lesson/gioi-thieu-ve-heap
- https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot-nhat
- https://tek4.vn/thuat-toan-shell-sort-ung-dung-cua-sap-xep-chen/
- https://en.wikipedia.org/wiki/Shellsort#Gap_sequences
- https://www.geeksforgeeks.org/shellsort/
- https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot-nhat
- https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot-nhat

- https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh?fbclid=IwAR0SCm9wKdFOBHwWyI0gtjFfpYNc-BmUf-LKdSalX12P6FHR_s-3kzUTBtw
- https://en.wikipedia.org/wiki/Flashsort