TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐH QG.TP HCM
KHOA CÔNG NGHỆ THÔNG TIN.
---oo0oo---

BÁO CÁO THỰC HÀNH: CÁC THUẬT TOÁN SẮP XẾP

HỌ VÀ TÊN: NGUYỄN TRẦN NGỌC VINH
LỚP: 19CTT4
MSSV: 19120720
MÔN: THỰC HÀNH CTDL& GIẢI THUẬT

CÁC THUẬT TOÁN

1. Selection sort Algorithm (Thuật toán " Chọn trực tiếp")

1.1 Ý tưởng thuật toán:

- Selection sort là một thuật toán sắp xếp đơn giản, dựa trên việc so sánh tại chỗ.
- Thuật toán selection sort sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất (giả sử ta cần sắp xếp mảng tăng dần) trong đoạn dãy chưa được sắp xếp và hoán vị phần tử đó với phần tử đầu tiên trong dãy chưa được sắp xếp (chứ không phải trong toàn bộ dãy). Thuật toán sẽ chia dãy là 2 dãy con:

- Một mảng con đã được sắp xếp.
- Mảng còn lại chưa được sắp xếp.

1.2 Các bước thực hiện:

- Bước 1: i = 0.
- Bước 2: Tìm phần tử nhỏ nhất a[min] trong mảng từ a[i] đến a[n-1].
- Bước 3: Hoán vị a[min] và a[i].
- Bước 4: Nếu i < n thì i = i+1. Lặp lại bước 2.
- Ngược lại, Dừng ở n – 1 phần tử đã nằm đúng vị trí theo thứ tự sắp xếp.

1.3 Ví dụ:

- Giả sử, ta có mảng như sau. Và cần sắp xếp theo thứ tự tăng dần.

12	**7**	**49**	**2**	**25**	**8**
[0]	[1]	[2]	[3]	[4]	[5]

- Bắt đầu tìm phần tử nhỏ nhất tức a[min] = a[3].
- Hoán vị phần tử nhỏ nhất vừa tìm được a[3] với phần tử đầu của mảng a[0].

2	**7**	**49**	**12**	**25**	**8**
[0]	[1]	[2]	[3]	[4]	[5]

- Tiếp tục xét từ phần tử thứ 2 là a[1]. Lúc này, a[min] = a[1] và phần tử này ngay sau a[0] nên ta không cần hoán vị.

2	**7**	**49**	**12**	**25**	**8**
[0]	[1]	[2]	[3]	[4]	[5]

- Hoán vị 8 và 49.

12	**7**	**8**	**12**	**25**	**49**
[0]	[1]	[2]	[3]	[4]	[5]

- Ta thấy mảng đã được sắp xếp thành công.

2	**7**	**8**	**12**	**25**	**49**
[0]	[1]	[2]	[3]	[4]	[5]

1.4 Minh họa chương trình

- SelectionSort (int a[], int n)

+for i = 0 to n – 1

++ min_index = i // vị trí của phần tử nhỏ nhất trong phần chưa sắp xếp.

++for j = i + 1 to n

++if a[j] < a[min_index]

+++ min_index = j

++Hoan vi (a[i], a[min_index]).

1.5 Độ phức tạp:

- Trong mọi trường hợp phép so sánh là:

$(n – 1) + (n – 2)+ …+ 1 = n*(n + 1) /2 = O (n^2)$.

- Số phép hoán vị:

  Trường hợp xấu nhất: O(n)

  Trường hợp tốt nhất: 0 - mảng đã tự sắp xếp tăng dần.

2. Merge sort Algorithm (Thuật toán “Sắp xếp trộn”)

- Là một phương thức sắp xếp dạng “Chia để trị”
- Nguyên tắc “Chia để trị”:

+ Nếu vấn đề nhỏ thì xử lí ngay

+ Nếu vấn đề lớn: chia thành hai vấn đề nhỏ, mỗi vấn đề bằng ½ .

+ Giải quyết vấn đề nhỏ.

+ Kết hợp kết quả của những vấn đề nhỏ với nhau.

2.1 Ý tưởng thuật toán:

- Chia dãy cần sắp xếp thành hai phần, ở vị trí giữa.
- Nếu phần của tử mỗi phần lớn hơn 1 thì: Sắp xếp mỗi phần bằng Merge sort.
- Trộn 2 phần đã được sắp xếp lại với nhau.
- Thuật toán Merge Sort có cài cài đặt bằng đệ quy.

2.2 Các bước thực hiện:

- Bước 1: Tìm chỉ số nằm giữa mảng để chia mảng thành hai nửa.

mid = (left + right) /2.

- Bước 2: Gọi đệ quy hàm MergeSort cho nửa đầu tiên.

MergeSort(array, left, mid).

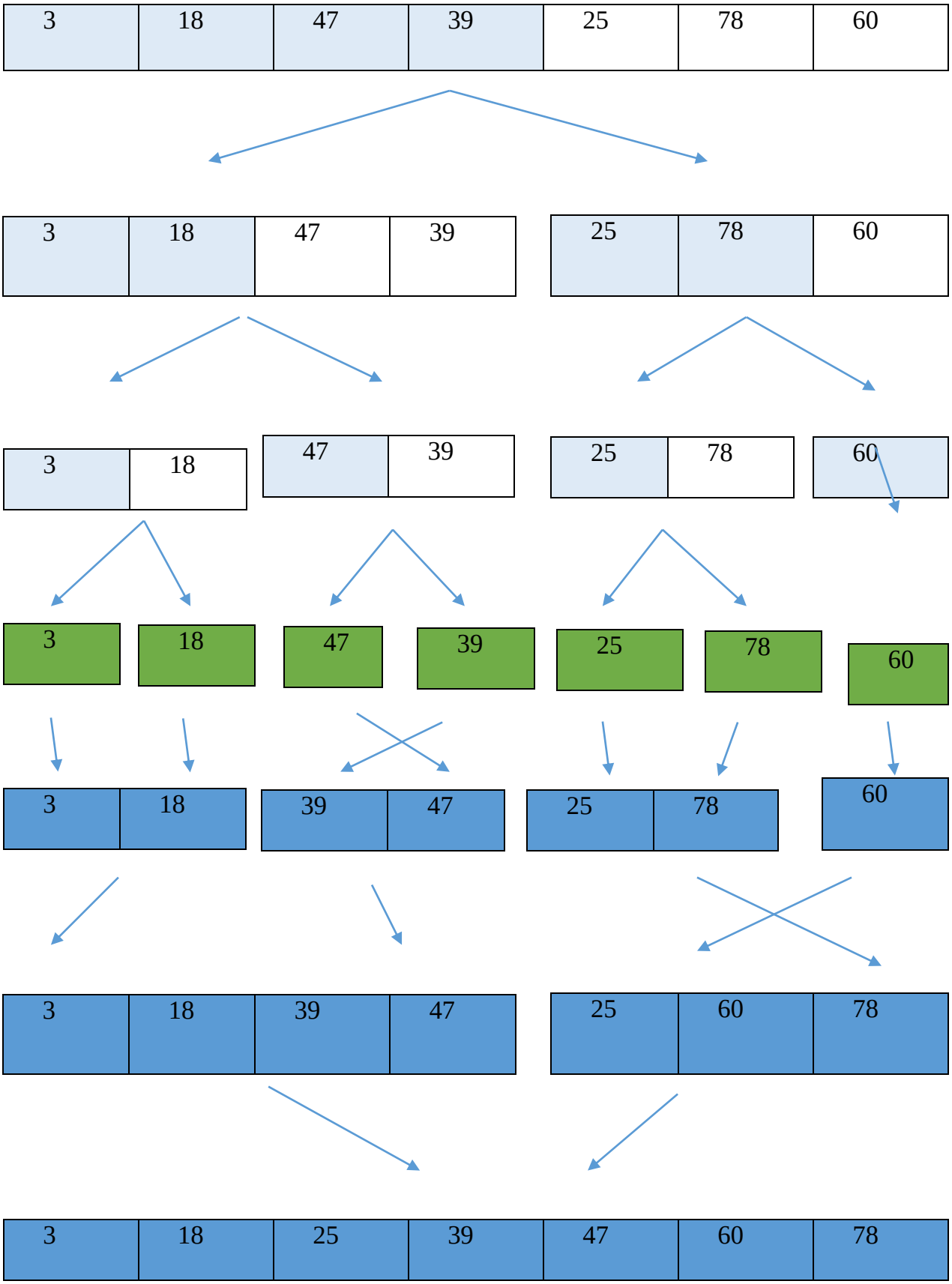
- Bước 3: Gọi đệ quy hàm MergeSort cho nửa thứ hai.

MergeSort(array, mid + 1, right).

- Bước 4: Gộp hai mảng đã sắp xếp ở bước 2 và bước 3 lại với nhau.

Merge(array, 1, mid, right).

2.3 Ví dụ


3 18 47 39 25 78 60
3 18 47 39
25 78 60
3 18
47 39
25 78
60
3
18
47
39
25
78
60
3 18
39 47
25 78
60
3 18 39 47
25 60 78
3 18 25 39 47 60 78

2.4 Minh họa chương trình:

- Gồm hai hàm:

- MergeSort (int array[], int left, int right)

  + if left < right

// mid: phần tử nơi chia mảng thành 2 phần, xử lí trường hợp số lớn.

  ++int mid = left + (right – left) / 2

// tiến hành gọi đệ quy cho các trường hợp tương tự mảng nhỏ sau khi chia.

  ++ MergeSort(array , left , mid)

  ++ MergeSort(array, mid + 1, right)

// sắp xếp mảng theo đúng thứ tự .

  ++ Merge(array , left, mid, right)

- Merge(int array[], int left, int mid, int right)

  + int Temp[], Temp2[].

// Copy phần tử mảng chính ra hai mảng phụ.

  + for i = 0 to mid, i++

    ++Temp[i] = array[left + i].

  + for j = mid + 1 to right, j++

    ++Temp[j] = array[mid + 1 + j].

//sắp xếp phần tử của hai mảng phụ vào mảng chính.

  + int i = 0, j = 0 , t = left.

  + While i < size_temp && j < size_temp2

    ++ If Temp[i] <= Temp2[j]

      array[t] = Temp[i]

      t++, i++.

    ++ else array[t] = Temp2[j]

      t++, j ++

//Copy các phần tử còn lại trong mảng phụ (nếu có) ra mảng chính

+ While(i< size_temp)

array[t]= temp[i]

t++, i++

+ While(j < size_temp2)

array[t]= Temp2[j]

t++, j++

2.5 Độ phức tạp của thuật toán :

- MergeSort là một thuật toán đệ quy và độ phức tạp thời gian có thể được biểu diễn như sau: $T(n) = 2\ t(n/2) + O(n)$.
- Độ phức tạp về thời gian: $O(n*\log_2(n))$ cho cả 3 trường hợp (xấu, trung bình và tốt nhất).
- MergeSort không bị ảnh hưởng bởi thứ tự ban đầu của dữ liệu nên có tính ổn định cao.

3. Heap sort Algorithm

Heap sort là một ứng dụng của cấu trúc dữ liệu heap, ta có thể sử dụng Max Heap hoặc Min Heap để thực hiện heap sort.

3.1 Ý tưởng thuật toán

- Giả sử, ta muốn sắp xếp các phần tử trong mảng A[] theo thứ tự tăng dần. Ta có thể dùng max heap để làm điều này bởi dựa vào đặc tính của max heap, node gốc là node lớn nhất. Như vậy, sau khi tìm được giá trị lớn nhất ta hoán vị nó với phần tử ở cuối mảng và bắt đầu tìm phần tử lớn thứ 2, tiếp tục như thế mảng sẽ được sắp xếp tăng dần.

3.2 Các bước thực hiện

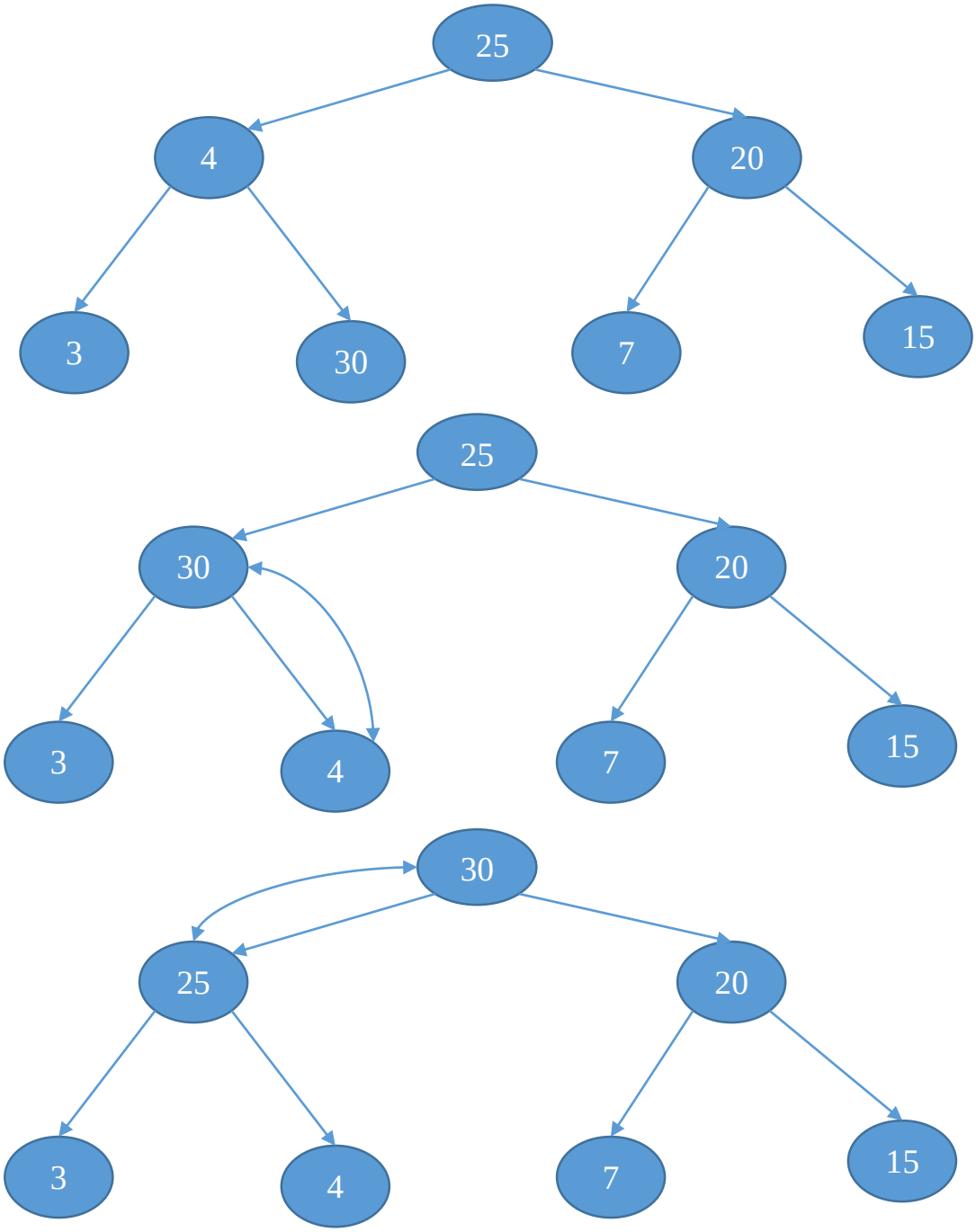
- Bước 1: Ta tạo một max heap của các phần tử trong mảng a[].
- Bước 2: Bây giờ ta thu được phần tử có giá trị lớn nhất rong mảng, chính là node gốc của heap (a[0]) trong mảng. Đổi chỗ phần tử này với phần tử cuối cùng của mảng A.
- Bước 3: Sau khi đổi chỗ, ta tiếp tục tạo max heap mới của các phần tử trong mảng với số phần tử n – 1 (vì phần tử cuối cùng đã được sắp xếp đúng vị trí do vậy kích thước của heap giảm đi 1)
- Bước 4: Lặp lại bước 2 và bước 3 cho đến khi tất cả các phần tử trong mảng đều được sắp xếp theo đúng thứ tự tăng dần.

3.3 Ví dụ:

Cho mảng như sau:

25	**4**	**20**	**3**	**30**	**7**	**15**
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Ta thấy phần tử lớn nhất sẽ ở ngay tại node gốc sau khi ta tạo max heap.(Minh họa tạo max heap ở dưới)



- Mảng sau khi được sắp xếp:

30	**25**	**20**	**3**	**4**	**7**	**15**
[0]	[1]	[2]	[3]	[4]	[5]	[6]

3.4 Minh họa chương trình:

- Gồm 2 hàm

- void heapify (int arr[], int n , int i)

// dua vi tri lon nhat len vi tri i

+ int largest = i

+ int left = 2* i + 1 // vị trí của con bên trái.

+ int right = 2* i + 2 // vị trí của con bên phải.

+ if left < n and arr[largest] < arr[left]

largest = left

+ if right < n and arr[largest] < arr[right]

largest = right

+ if largest != i

hoán vị (arr[i], arr[largest])

heapify (arr, n, largest)

- void heapsort(int arr[], int n)

// tạo max heap

+for i = n / 2 – 1 to i >=0 and i --

++Heapify(arr,n,i)

+ for i = n – 1 to i > 0 and i--

++hoán vị (arr[0], arr[i]) //đưa ra phần tử lớn ra cuối mảng

++ heapyfi(arr,i,0) // tao một max heap mới với số phần tử sẽ là i + 1.

3.5 Độ phức tạp:

- Độ phức tạp về thời gian của heapify là $O(\log_2 n)$.
- Độ phức tạp về thời gian để tạo heap là O(n)
- Heap sort thường có độ phức tạp về thời gian là $O(n^* \log_2 n)$ nhưng trường hợp xấu nhất là có độ phức tạp $O(n^2)$.

4. Quick sort Algorithm

4.1 Ý tưởng thuật toán:

- Quick sort là một thuật toán 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 đến tốc độ sắp xếp. Sau đây, là một số cách để chọn pivot thường được sử dụng:

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

4.2 Các bước thực hiện:

- Bước 1: Chọn phần tử pivot (trong bài là này phần từ này ở giữa).
- Bước 2: Gán giá trị đầu và cuối dãy tương ứng với phần tử left & right.
- Bước 3: Bắt đầu từ phần tử bên trái, di chuyển phần tử này sang bên phải và tìm phần tử lớn hơn giá trị pivot.
- Bước 4: Tương tự bước 3, nhưng bắt đầu từ phần tử bên phải, di chuyển chuyển sang bên trái và tìm phần tử có giá trị lớn nhỏ hơn giá trị pivot.
- Bước 5: Hoán đổi các phần tử được tìm thấy ở bước 3 và bước 4.
- Bước 6: Lặp lại bước 3, 4, 5 cho đến khi left > right.
- Bước 7: Lặp lại toàn bộ cho hai mảng con ở bên trái và bên phải của con trỏ bên trái.

Ví dụ: Cho mảng sau và yêu cầu sắp xếp tăng dần:

5	**9**	**28**	**18**	**14**	**13**	**11**
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Left = 0 , Right = 6, Key = 18 (Key giá trị của pivot)

5	9	11	13	14	18	28
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Vào mảng con bên trái Left = 0, Right = 4, Key = 11(không có sự thay đổi)

5	9	11	13	14	18	28
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Vào mảng con bên trái Left = 0, Right = 1, Key = 5(không có sự thay đổi)

5	9	11	13	14	18	28
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Vào mảng con bên phải Left = 3, Right = 4, Key = 13(không có sự thay đổi)

5	9	11	13	14	18	28
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Vào mảng con bên phải Left = 5, Right = 6, Key = 18(không có sự thay đổi)

5	9	11	13	14	18	28
[0]	[1]	[2]	[3]	[4]	[5]	[6]

4.3 Minh họa thuật toán:

- void quickSort(int array[], int left, int right)

    + if left <= right

        + + int pivot = left + (right – left) / 2 // chọn phần tử pivot

        ++ int i = left

```
++ int j = right
++ while i <= j {
// tìm phần tử lớn hơn pivot từ bên trái.
+++ while array[i] < array[pivot]
i++
// tìm phần tử nhỏ hơn pivot từ bên phải.
+++ while array[j] < array[pivot]
j - -
+++ if  i  <=  j
// đổi vị trí  hai phần tử vừa tìm được ở trên
Hoanvi ( array[i], array[j])
i++, j –
}
// gọi đệ quy cho mảng con tiếp theo.
++ if( left < i )
quickSort ( array, left, i)
++ if( j< right )
quickSort ( array, i, right)
```

4.4 Độ phức tạp:

- Trường hợp xấu nhất: Trường hợp xấu nhất xảy ra khi pivot được chọn sẽ là phần tử lớn nhất hoặc nhỏ nhất, hoặc mảng được sắp xếp theo trường hợp tăng dần hoặc giảm dần.

$T(n) = T(n – 1) + O(n)$

⇨ Độ phức tạp về thời gian lúc này là $O(n^2)$.

- Trường hợp tốt nhất: Trường hợp tốt nhất là khi pivot được chọn là phần tử có giá trị trung bình trong dãy.

$T(n) = 2T(n/2) + (n)$

⇨ Độ phức tạp về thời gian lúc này là $O(n \log_2 n)$.

- Trường hợp bình thường: Độ phức tạp về thời gian là $O(n\log_2 n)$

5. Bubble sort Algorithm

5.1 Ý tưởng thuật toán:

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

5.2 Ví dụ:

- Cho mảng sau: Yêu cầu sắp xếp tăng dần

5	**1**	**4**	**2**	**9**
[0]	[1]	[2]	[3]	[4]

- Lặp lần 1:

+ 1 và 5 đổi chỗ cho nhau (1 < 5)

1	**5**	**4**	**2**	**9**
[0]	[1]	[2]	[3]	[4]

+ 5 và 4 đổi chỗ cho nhau (4 < 5)

1	**4**	**5**	**2**	**9**
[0]	[1]	[2]	[3]	[4]

+5 và 2 đổi chỗ cho nhau (2 < 5)

1	**4**	**2**	**5**	**9**
[0]	[1]	[2]	[3]	[4]

+ 5 < 9 nên không thực hiện hoán vị.

5	**1**	**4**	**2**	**9**
[0]	[1]	[2]	[3]	[4]

- Lặp lần 2

+ 1 < 4 không thực hiện hoán vị

1	**4**	**2**	**5**	**9**
[0]	[1]	[2]	[3]	[4]

+ 2 và 4 đổi chỗ cho nhau (2 < 4)

1	**2**	**4**	**5**	**9**
[0]	[1]	[2]	[3]	[4]

+ 4 < 5 không thực hiện hoán vị

+ 5 < 9 không thực hiện hoán vị

- Lặp lại lần 3

+ 1 < 2 không thực hiện hoán vị

+ 2 < 4 không thực hiện hoán vị

+ 4 < 5 không thực hiện hoán vị

+ 5 < 8 không thực hiện hoán vị

- Lưu ý: Lí do thực hiện lần lặp thứ 3 mặc dù dãy đã được sắp xếp vì thuật toán ta không nhận ra điều này ngay được. Thuận toán cần thêm 1 lần lặp nữa để kết luận dãy đã được sắp xếp bằng cách kiểm tra từ đầu đến cuối dãy mà không thấy sự hoán vị nào xảy ra.

5.3 Minh họa chương trình

- Void bubbleSort(int a[], int n){

  + bool flag = false

  + for i = 0 to n – 1, i++

    ++ flag = false

    ++ for j = 0 to n – i – 1

//kiểm tra & hoan vi hai phần tử cho đúng thứ tự sắp xếp

    +++ if a[j+1] < a[j]{

      hoan vi (a[j+1], a[j])

      flag = true

    }

    ++ if flag = false

      break // dừng chương trình

}

5.4 Đánh giá thuật toán:

- Trường hợp xấu nhất xảy ra khi mảng được sắp xếp ngược lại. Độ phức tạp về thời gian là O(n*n)
- Trường hợp tốt nhất xảy ra khi mảng đã được sắp xếp. Độ phức tạp về thời gian là O(n)

6. Insertion sort Algorithm

6.1 Ý tưởng thuật toán:

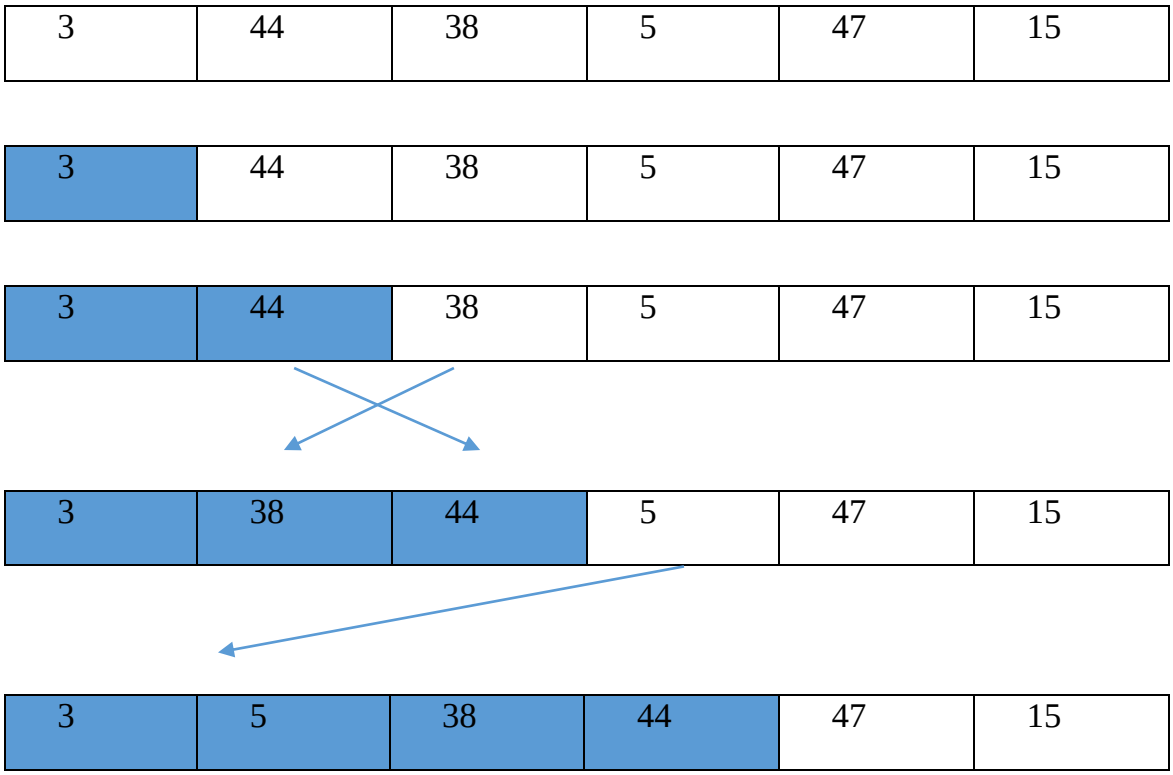
- Thuật toán Insertion sort thực hiện sắp xếp dãy số theo cách duyệt từng phần tử và chèn từng phần tử đó vào đúng vị trí trong mảng con(dãy số từ đầu đến phần tử phía trước nó) đã được sắp xếp sao cho dãy số trong mảng sắp đã xếp đó vẫn đảm bảo tính chất của một dãy số tăng dần.

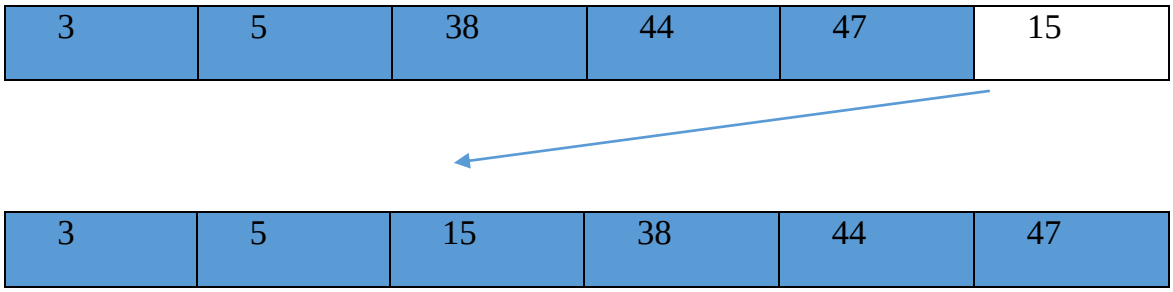
6.2 Các bước thực hiện:

- Bước 1: Khởi tạo mảng với dãy con đã được sắp xếp có k = 1 phần tử (a[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 tròn 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 một phần tử
- Bước 3: Lặp lại cho tới khi duyệt hết tất cả các phần tử của mảng.

6.3 Ví dụ:

- Cho mảng sau: Yêu cầu sắp xếp tăng dần.





6.4 Minh họa chương trình:

- void insertionSort(int arr[], int n)

    + int i, key, j

    + for i = 1 to n

        ++key = arr[i]

        ++j = i – 1

        ++while 0 <= j and key < arr[j]

            +++arr[j+1] = arr[i]

            +++j = j - 1

        ++arr[j+1] = key;

6.5 Độ phức tạp về thời gian của thuật toán

- Trường hợp tốt: O(n)
- Trung bình và xấu nhất: O(n^2). Sắp xếp chèn mất thời gian tối đa để sắp xếp nếu các phần tử được sắp xếp theo thứ tự ngược lại.

7. Binary Insertion sort Algorithm

7.1 Ý tưởng thuật toán:

- Binary Insertion Sort sử dụng tìm kiếm nhị phân để tìm vị trí thích hợp để chèn phần tử đã chọn ở mỗi lần lặp.

Ví dụ: Cho mảng sau là yêu cầu sắp xếp tăng dần

24	**32**	**18**	**26**	**50**	**85**
[0]	[1]	[2]	[3]	[4]	[5]

24	**32**	**18**	**26**	**50**	**98**
[0]	[1]	[2]	[3]	[4]	[5]

24	**32**	**18**	**26**	**50**	**98**
[0]	[1]	[2]	[3]	[4]	[5]

24	**32**	**32**	**26**	**50**	**98**
[0]	[1]	[2]	[3]	[4]	[5]

24	**24**	**32**	**26**	**50**	**98**
[0]	[1]	[2]	[3]	[4]	[5]

12	**24**	**32**	**26**	**50**	**98**
[0]	[1]	[2]	[3]	[4]	[5]

- Tiếp tục như vậy cho đến phần tử cuối cùng.

7.2 Minh họa chương trình:

- Gồm 2 hàm
  - int binarySearch(int a[], int item, int low, int right)

// hàm tìm vị trí thích hợp dựa vào thuật toán tìm kiếm nhị phân

+ if high <= low

return low + 1 nếu item > a[low].Ngược lại, return low.

+ int mid = low + (high – low) / 2

+ if item = a[mid]{

return mid + 1

}

+ if item > a[mid]

return binarySearch(a, item, mid + 1, high)

else return binarySearch(a,item,low, mid - 1)

- void binaryInsertionSort(int a[], int n){

+ int i, location, j , k , selected

+ for i = 1 to n, i++

++ j = i – 1

++ selected = a[i]

// gọi hàm binarySearch để tìm vị trí thích hợp cho việc chèn phần tử.

++ location = binarySearch(a,selection, 0, j);

++ while location <= j {

// di chuyển dữ liệu ra sau 1 vị trí

a[j+1] = a[j]

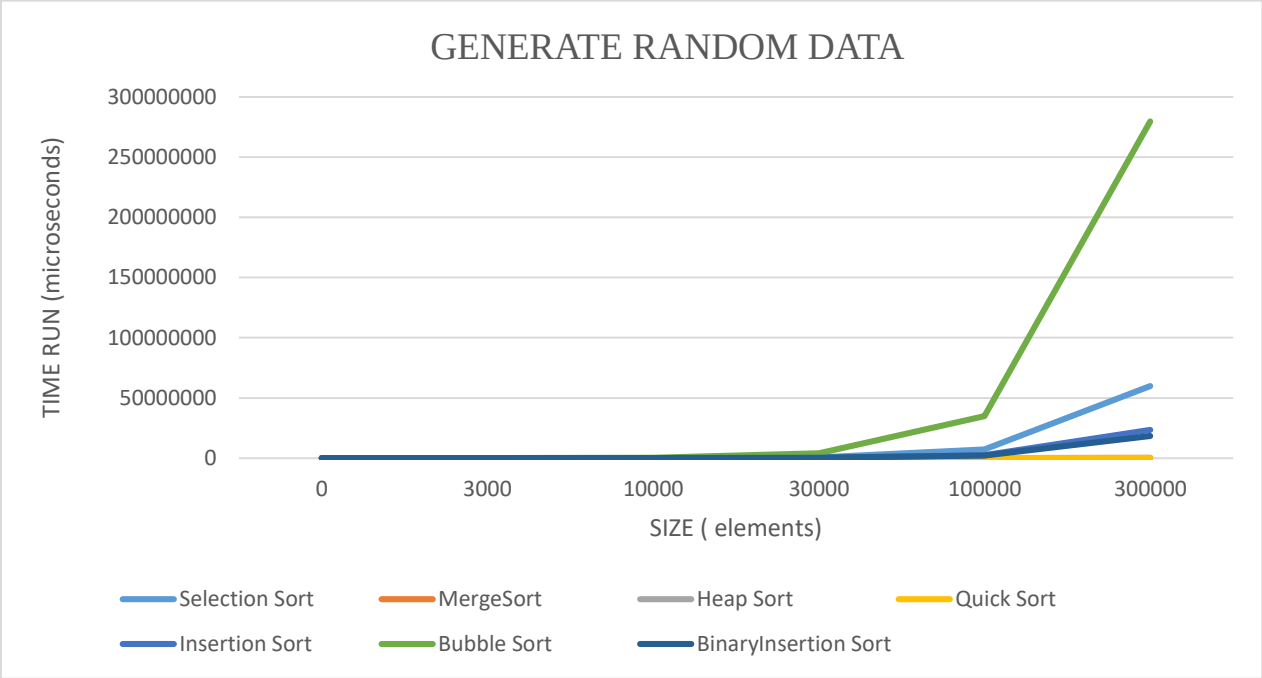
```
            j—
        }
    ++ a[j+1] = selected
}
```

7.3 Đánh giá thuật toán

- Độ phức tạp của thuật toán theo thời gian luôn là O(n2) trong trường hợp xấu nhất là thực hiện hoán đổi vị trị các phần tử trong mỗi lần chèn.

KHẢO SÁT THỰC NGHIỆM

1. Generate Random Data.



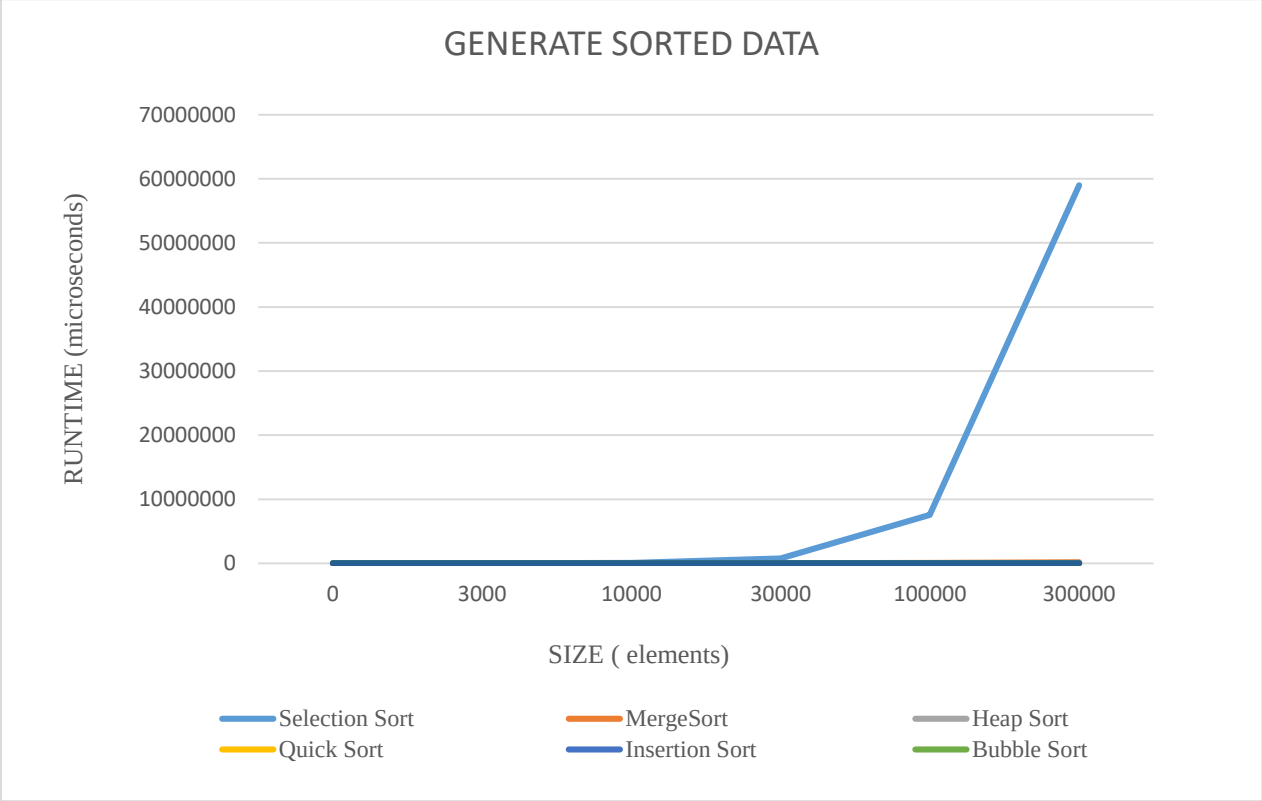
Hình ảnh 1. Biểu đồ thể thời gian chạy của các thuật toán với dữ liệu đầu vào là ngẫu nhiên.

- Dựa trên biểu đồ ta có thể nhìn thấy:

- Quick sort là thuật toán sắp xếp chạy nhanh nhất.
- Bubble Sort là thuật toán sắp xếp chạy chậm nhất.
- Và các thuật toán còn lại được sắp xếp theo thời gian chạy nhanh như sau:

Heap Sort > Merge Sort > Binary Insertion Sort > Insertion Sort > Selection Sort.

2. Generate Sorted Data

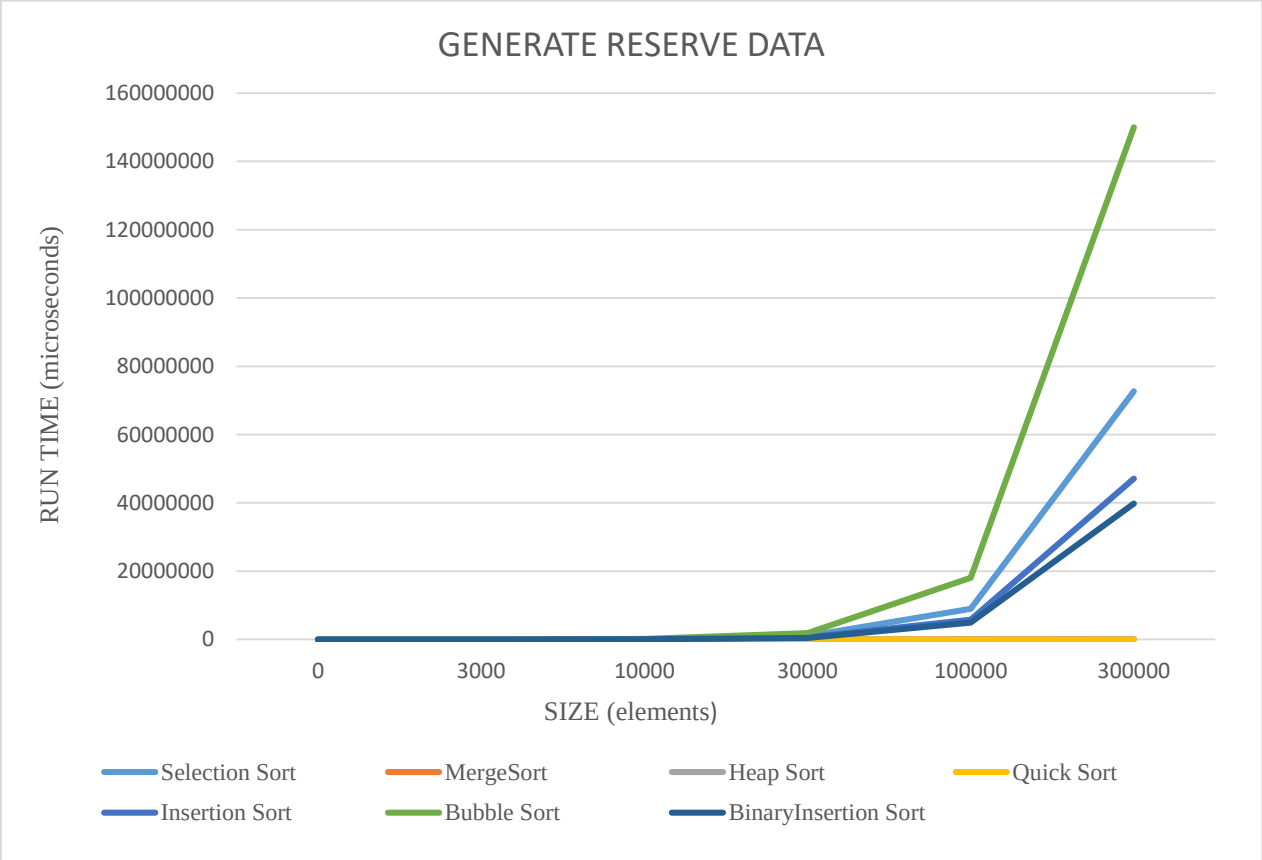


Hình ảnh 2. Biểu đồ thể thời gian chạy của các thuật toán với dữ liệu đầu vào có thứ tự tăng dần.

- Dựa vào biểu đồ ta có thể nhìn thấy:

- Bubble sort là thuật toán chạy nhanh nhất.
- Selection sort là thuật toán chạy chậm nhất.
- Ta thấy rằng Bubble Sort lại chạy nhanh nhất trong trường hợp này vì nó không thực hiện công việc hoán đổi vị trí nào cả, và chỉ thực duyệt qua lần lượt các phần tử một lần nên thời gian chạy của Bubble Sort khá nhanh.

3. Generate Reserve Data

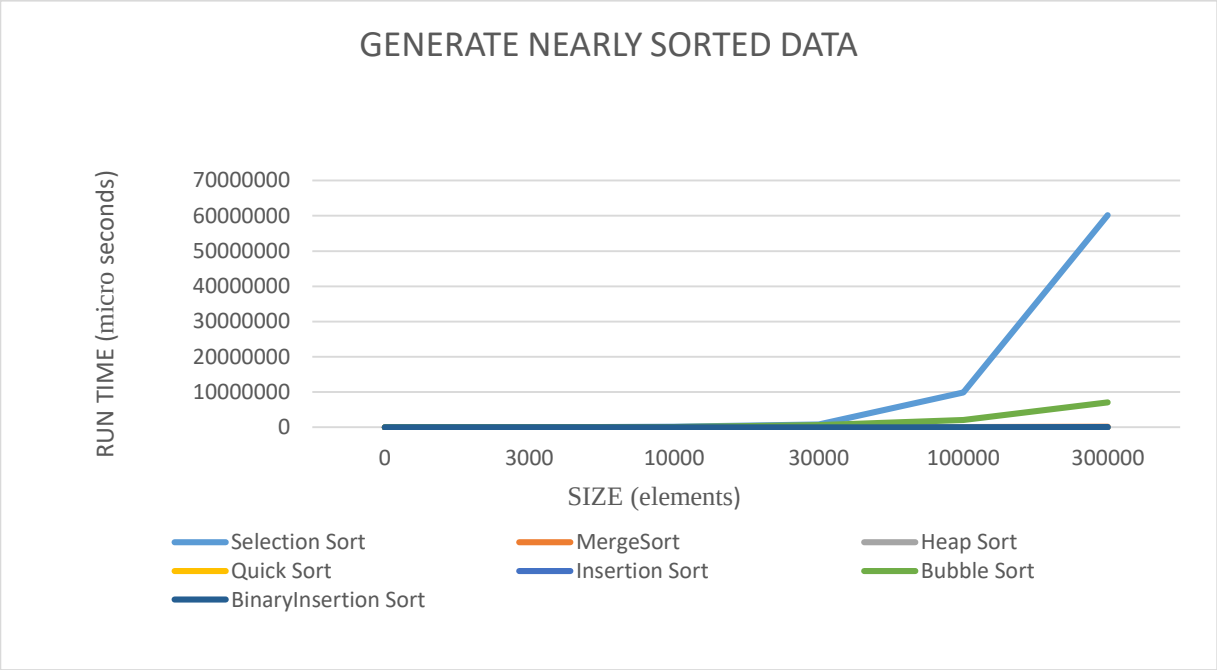


Hình ảnh 3. Biểu đồ thể thời gian chạy của các thuật toán với dữ liệu đầu vào có thứ tự giảm dần.

- Dựa vào biểu đồ ta thấy:

- Thuật toán chạy nhanh nhất chính là Quick Sort.
- Thuật toán chạy chậm nhất chính là Bubble Sort.
- Sắp xếp giảm dần về thời gian chạy nhanh của các thuật toán còn lại như sau: Heap Sort > MergeSort > Insertion Sort > BinaryInsertion Sort > Selection Sort.

4. Generate Nearly Sorted Data



Hình ảnh 4. Biểu đồ thể thời gian chạy của các thuật toán với dữ liệu đầu vào gần như có thứ tự

- Dựa vào biểu đồ ta thấy:

- Thuật toán chạy chậm nhất là Selection Sort.
- Chạy nhanh nhất là Insertion Sort.
- Dựa vào biểu đồ và bản số liệu sắp xếp thời gian chạy nhanh của các thuật toán còn lại theo thứ tự giảm dần như sau:

  Quick Sort > Binary Insertion Sort > Heap Sort > Merge Sort > Bubble Sort.

5. Đánh giá chung:

- Quick Sort và Merge Sort luôn là những thuật toán chạy nhanh nhất.
- Bubble Sort và Selection Sort là những thuật toán chạy chậm nhất.
- Những thuật toán có tính ổn định kém: Quick Sort và Heap Sort. Những thuận toán có tính ổn định cao là Merge Sort, Insertion Sort, Bubble Sort, Binary Insertion Sort.
- Insertion Sort tốt hơn Selection Sort nhất là khi mảng đã có thứ tự sắp xếp.
- Nhìn chung Quick Sort nhanh hơn Heap sort 2 lần nhưng Heap Sort lại có tính ổn định cao hơn trong mọi trường hợp.