KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
227 Nguyễn Văn Cừ, Phường 4, Quận 5, TP.HCM
Điện Thoại: (08) 38.354.266 - Fax:(08) 38.350.096

Môn học: Cấu trúc dữ liệu và giải thuật
Lớp 20CTT3_N2 - Nhóm 16

Báo cáo: Bài tập thực hành Lab 03
Các thuật toán sắp xếp

Mục lục

Thông tin nhóm

MSSV	Họ tên	Email
18120622	Lê Văn Trung	18120622@student.hcmus.edu.vn
18120299	Trương Công Quốc Cường	18120299@student.hcmus.edu.vn
20120401	Nguyễn Đức Việt	20120401@student.hcmus.edu.vn
20120429	Nguyễn Quốc Anh	20120429@student.hcmus.edu.vn

Giới thiệu

Sắp xếp là hành động được sử dụng nhiều trong đời sống, công việc khác nhau, và công nghệ thông tin cũng không ngoại lệ. Sử dụng thuật toán đúng cách sẽ giúp chúng ta giảm được thời gian và tiết kiệm chi phí khi triển khai ứng dụng.

Tài liệu báo này trình bày bao gồm: Thông tin của nhóm, giới thiệu tài liệu và trình bày các thuật toán thuộc Set 2 của bài thực hành Lab 03.

Có 11 thuật toán được trình bày bao gồm: Selection Sort, Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Radix Sort, Bubble Sort, Shaker Sort, Shell Sort, Counting Sort và Flash Sort.

Các thuật toán dưới đây được miêu tả cụ thể qua ý tưởng, ví dụ và độ phức tạp. Bên cạnh đó, để có cái nhìn rõ ràng, các thuật toán đã được so sánh với nhau, từ đó chúng ta phân tích được điểm mạnh, điểm yếu của từng thuật toán, để đưa ra lựa chọn đúng đắn.

Trong quá trình thực hiện báo cáo, khó tránh khỏi sai sót về kiến thức, vì vậy, rất mong sự nhận xét và đánh giá của thầy cô để giúp chúng em hoàn thiện hơn.

Trình bày thuật toán

1. Thuật toán Selection Sort

a. Ý tưởng

Sắp xếp chọn hay sắp xếp lựa chọn là một thuật toán chọn phần tử nhỏ nhất từ danh sách chưa được sắp xếp trong mỗi lần lặp và đặt phần tử đó ở đầu danh sách chưa được sắp xếp. Quá trình này diễn ra liên tục cho tới khi có được một danh sách các giá trị được sắp xếp hoàn chỉnh.Danh sách chứa các phần tử sẽ được chia làm hai phần. Phần ở bên trái là phần được sắp xếp (sort list) và phần ở bên phải là phần chưa được sắp xếp (unsorted list). Ban đầu chưa được sắp xếp thì phần sorted list sẽ trống và phần unsorted list sẽ chứa tất cả các phần tử ban đầu. Phần tử nhỏ nhất trong list sẽ được chọn và tráo đổi với vị trí đầu tiên trong list, tiếp đến sẽ là vị trí nhỏ thứ hai tiếp tục được tráo đổi ngay sau vị trí nhỏ nhất.

b. Các bước chạy

Hàm selectionSort(a[ ], N)

Bước 1: Khởi tạo i=0

Bước 2: min=i;

Bước 3: Khởi tạo j=i+1

Bước 4: Nếu a[j] < a[min] thì gán min bằng j;

```
if (a[j] < a[min])
    min = j;
```

Bước 5: Nếu j<n quay lại Bước 4 với j=j+1

Ngược lại: qua Bước 6

Bước 6: Hoán vị a[i] và a[min]

Nếu i<N-1: quay lại Bước 1 với i=i+1;

Ngược lại: Dừng!

c. Ví dụ

- Sắp xếp mảng sau với n = 5

5	9	6	7	2

- Khởi tạo i = 0, min = 0, j = 1. Tìm min từ vị trí 0 đến 4
- min = 4, đổi chỗ a[0] và a[4]

2	9	6	7	5

- Khởi tạo i = 1, min = 1, j = 2. Tìm min từ vị trí 1 đến 4
- min = 4, đổi chỗ a[2] và a[4]

2	5	6	7	9

- Khởi tạo i = 2, min = 2, j = 3. Tìm min từ vị trí 2 đến 4
- min = 2, đổi chỗ a[2] và a[2]

2	5	6	7	9

- Khởi tạo i = 3, min = 3, j = 4. Tìm min từ vị trí 3 đến 4
- min = 3, đổi chỗ a[3] và a[3]

2	5	6	7	9

- i = 4 = n-1. Dừng. Mảng đã được sắp xếp.

d. Đánh giá độ phức tạp

_ Số phép so sánh: $(n-1)+(n-2)+(n-3)+.....+1=n(n-1)/2$ gần bằng n^2. Độ phức tạp = $O(n^2)$

_ Ngoài ra, chúng ta có thể phân tích độ phức tạp bằng cách quan sát số vòng lặp. Có 2 vòng lặp nên độ phức tạp là $n \times n = n^2$

	Time	Space
Best case	O(n)	O(1)
Average case	$O(n^2)$	O(1)
Worst case	$O(n^2)$	O(1)

e. Ứng dụng

Thuật toán Selection sort được sử dụng trong các trường hợp:

+ Mảng có ít phần tử
+ Chi phí hoán đổi không thành vấn đề
+ Không quan trọng vấn đề thời gian

2. Thuật toán Insertion Sort

a. Ý tưởng

Sắp xếp chèn là một thuật toán sắp xếp có nhiệm vụ đặt một phần tử chưa được sắp xếp vào vị trí thích hợp của nó trong mỗi lần lặp. Giống như cách sắp xếp quân bài của những người chơi bài. Muốn sắp một bộ bài theo trật tự người chơi bài rút lần lượt từ quân thứ 2, so với các quân đứng trước nó để chèn vào vị trí thích hợp.

b. Các bước chạy

insertionSort(array[ ], int N)

Bước 1: Khởi tạo i=1;
value=array[i];
hole=i;

Bước 2: Nếu hole>0 và array[hole - 1]>value
array[hole]=array[hole-1];
hole =hole-1;
Ngược lại: Qua Bước 3

Bước 3: a[hole]=value;

Bước 4: Nếu i <N: quay lại Bước 1 với i=i+1;
Ngược lại: Dừng!

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- Khởi tạo i = 1, value = 9, hole = 1
- a[0] < value, giữ nguyên vị trí của a[1]

5	9	6	7	2

- Khởi tạo i = 2, value = 6, hole = 2
- a[1] > value nên hole = 1. Dịch 9 sang phải 1 ô, chèn 6 vào vị trí 1

5	6	9	7	2

- Khởi tạo i = 3, value = 7, hole = 3
- a[2] > value nên hole = 2. Dịch 9 sang phải 1 ô, chèn 7 vào vị trí 2

5	6	7	9	2

- Khởi tạo i = 4, value = 2, hole = 4
- a[0] > value nên hole = 0. Từ 5 dịch tất cả phần tử sang phải 1 ô và chèn 2 vào vị trí 0

2	5	6	7	9

d. Đánh giá độ phức tạp

- Độ phức tạp của trường hợp tốt nhất: O(n)

  Khi mảng đã được sắp xếp, vòng lặp bên ngoài chạy trong n số lần trong khi vòng lặp bên trong hoàn toàn không chạy. Vì vậy, chỉ có n số phép so sánh được thực hiện. Do đó, độ phức tạp là tuyến tính.

- Độ phức tạp của trường hợp trung bình: O(n)

  Xảy ra khi các phần tử của một mảng có thứ tự lộn xộn (không tăng dần cũng không giảm dần).

- Độ phức tạp của trường hợp xấu nhất: $O(n^2)$

  Giả sử, một mảng có thứ tự tăng dần và ta muốn sắp xếp nó theo thứ tự giảm dần. Trong trường hợp này, trường hợp xấu nhất sẽ xảy ra.

  Mỗi phần tử phải được so sánh với mỗi phần tử khác, do đó, đối với mỗi phần tử thứ n, (n-1) số phép so sánh sẽ được thực hiện.

  Do đó, tổng số phép so sánh = n×(n−1)n2

	Time	Space
Best case	O(n)	O(1)
Average case	O(n)	O(1)
Worst case	$O(n^2)$	O(1)

e. Ứng dụng

Thuật toán Insertion sort được sử dụng trong các trường hợp:

+ Mảng có ít phần tử
+ Mảng gần như đã được sắp xếp, chỉ một vài phần tử bị đặt sai chỗ

3. Thuật toán Merge Sort

a. Ý tưởng

Merge Sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa và lặp lại tương tự trên các mảng đã chia. Sau đó gộp các nửa thành mảng đã sắp xếp.

b. Các bước chạy

- Hàm mergeSort(array[], left, right):
  Nếu left < right:
  Bước 1:
  Tìm phần tử chính giữa mảng để chia mảng thành 2 nửa:
  mid = left + (right - left)/2;
  Bước 2: Gọi hàm đệ quy mergeSort cho nửa đầu tiên:
  mergeSort(array[], left, mid);
  Bước 3: Gọi hàm đệ quy mergeSort cho nửa sau:
  mergeSort(array[], mid+1, right);
  Bước 4: Gọi hàm merge để gộp 2 mảng đã sắp xếp ở bước 2 và bước 3 thành một mảng sắp xếp:
  merge(array[], left, mid, right);
- Hàm merge(array[], left, mid, right):
  Bước 1:
  Tính số phần tử của 2 mảng con
  n1 = mid -left + 1;
  n2 = right - mid;
  Bước 2:
  Sao chép phần tử vào 2 mảng con: L[n1] và R[n2]
  for(i=0; i<n1; i++)
  L[i] = array[left+i]
  for(j=0; j<n2; j++)
  R[j] = array[mid+j+1)
  Bước 3:
  Gộp 2 mảng con thành mảng đã sắp xếp:

```
i=0, j=0, k=left;
while(i<n1 và j<n2):
        if(L[i] < R[j]):
                array[k] = L[i];
                i++;
        else
                array[k] = R[j];
        k++;
while(i<n1):
        array[k] = L[i];
        i++;
        k++;
while(j<n2):
        array[k] = R[j];
        i++;
        k++;
```

c. Ví dụ:

- Cho mảng sau với n = 5

5	9	6	7	2

- Chia đôi mảng thành 2 mảng con: L11[3] và R11[2]

5	9	6

7	2

- Chia đôi mảng L11 thành: L21[2] và R21[1]

5	9

6

- Chia đôi mảng R11 thành: L22[1] và R22[1]

7

2

- Chia đôi mảng L21 thành: L31[1] và R31[1]

5

9

- Gộp mảng L31 và R31:

5	9

- Gộp mảng L22 và R22:

2	7

- Gộp mảng L21 và R21:

5	6	9

- Gộp mảng L11 và R11:

2	5	6	7	9

d. Đánh giá độ phức tạp

- Để gộp 2 nửa mảng với nhau ta cần duyệt qua mỗi phần tử một lần nên sẽ mất O(n) để gộp mảng. Ta cần đệ quy tiếp với các nửa mảng cho đến khi mảng được sắp xếp hoàn toàn. Do đó độ phức tạp trong cả 3 trường hợp là O(nlog(n)).
- Merge sort sử dụng mảng phụ có độ lớn n để lưu dữ liệu đã gộp.

	Time	Space
Best case	O(n log(n))	O(n)
Average case	O(n log(n))	O(n)
Worst case	O(n log(n))	O(n)

e. Ứng dụng

Thuật toán Merge sort được sử dụng trong các trường hợp:

+ Đếm nghịch đảo
+ Sắp xếp ngoài mảng
+ Ứng dụng thương mại điện tử

4. Thuật toán Heap Sort

a. Ý tưởng

Heap Sort là một kỹ thuật sắp xếp so sánh dựa trên cấu trúc dữ liệu heap nhị

phân. Ta xây dựng max-heap hoặc min-heap từ mảng dữ liệu đã cho và đưa phần tử cực trị vào vị trí đúng trong mảng sắp xếp, thực hiện tương tự cho đến khi tất cả được sắp xếp.

b. Các bước chạy

- Hàm heapSort(array[], n):

Bước 1:
Xây dựng cấu trúc heap nhị phân với dữ liệu đầu vào: Max-heap cho sắp xếp tăng dần và Min-heap cho sắp xếp giảm dần.
Trong phạm vi bài thực hành, ta xây dựng max-heap.
Bước 2:
Khi đó, phần tử root có giá trị lớn nhất. Ta đổi chỗ phần tử root cho phần tử cuối cùng của heap và giảm kích thước của heap xuống 1.
Bước 3:
Xây dựng lại max-heap tại vị trí root.
Bước 4:
+ Nếu kích thước của heap lớn hơn 1: Quay lại bước 2.
+ Nếu kích thước của heap bằng 1: Mảng đã được sắp xếp.

- Hàm heapify(array[], n, i): Xây dựng max-heap

Bước 1:
Tìm vị trí của các node lá trái và phải:
left = 2*i+1;
right = 2*i+2;
Bước 2:
Tìm phần tử lớn nhất trong các node: left, root, right và gọi hàm đệ quy heapify tại node con nếu bị thay đổi.

```
largest = i;
if(left < n và array[left] > array[largest])
        largest = left;
if(right < n và array[right] > array[largest])
        largest = right;
```

Bước 3:
Nếu node cha nhỏ hơn node con, đổi chỗ node cha và node con lớn nhất. Sau đó gọi hàm đệ quy heapify tại vị trí node con có sự thay đổi.

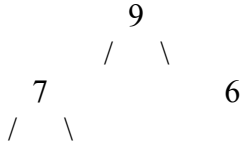
```
if(largest != i)
        HoanVi(array[i], array[largest]);
        heapify(array[], n, largest);
```

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- Xây dựng max-heap:



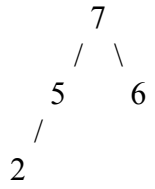
5 2

9	7	6	5	2

- Đổi chỗ a[0] và a[n-1]

2	7	6	5	9

- Xây dựng lại max-heap với các phần tử từ vị trí 0 đến 3:

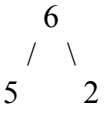


7	5	6	2	9

- Đổi chỗ a[0] và a[3]

2	5	6	7	9

- Xây dựng lại max-heap với các phần tử từ vị trí 0 đến 2:

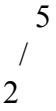


6	5	2	7	9

- Đổi chỗ a[0] và a[2]

2	5	6	7	9

- Xây dựng lại max-heap với các phần tử từ vị trí 0 đến 1:



5	2	6	7	9

- Đổi chỗ a[0] và a[1]

2	5	6	7	9

- Mảng đã được sắp xếp.

d. Đánh giá độ phức tạp

- Quá trình lấy một phần tử ra khỏi cấu trúc heap có n phần tử mất O(log(n)). Tổng thời gian để thực hiện hết n phần tử là O(log(n)) + O(log(n-1)) + O(log(n-2)) + … + O(log(1)) = O(nlog(n)).

- Heap sort là một thuật toán sắp xếp tại chỗ nên không sử dụng đến không gian lưu trữ mảng phụ.

	Time	Space
Best case	O(n log(n))	O(1)
Average case	O(n log(n))	O(1)
Worst case	O(n log(n))	O(1)

e. Ứng dụng

Thuật toán Heap sort được sử dụng trong các trường hợp:

+ Thuật toán lựa chọn: có thể sử dụng đống để tìm phần tử lớn nhất, nhỏ nhật, trung vị, phần tử lớn thứ k, trong thời gian tuyến tính.
+ Thuật toán cho đồ thị: nhiều thuật toán cho đồ thị sử dụng cấu trúc dữ liệu đống như thuật toán Dijkstra, hay thuật toán Prim.

5. Thuật toán Quick Sort

a. Ý tưởng

- Quick Sort là một thuật toán chia để trị. Thuật toán sẽ chọn một phần tử làm điểm đánh dấu (pivot), sau đó chia mảng thành 2 phần: các phần tử lớn hơn pivot và các phần tử nhỏ hơn pivot. Thực hiện tương tự với các mảng con cho đến khi mảng được sắp xếp.
- Phần tử đánh dấu sẽ quyết định đến tốc độ chạy của chương trình, tuy nhiên, không có phương pháp để xác định đâu là phần tử tốt nhất. Người ta thường dùng các cách sau để chọn phần tử đánh dấu:
  + Chọn phần tử đầu tiên
  + Chọn phần tử cuối cùng
  + Chọn phần tử đứng giữa
  + Chọn phần tử ngẫu nhiên
- Trong phạm vi bài thực hành này, chúng ta chọn phần tử chính giữa làm pivot.

b. Các bước chạy

- Hàm partition(array[], low, high): Đưa các phần tử nhỏ hơn pivot về bên trái, các phần tử lớn hơn pivot về bên phải và trả về vị trí của pivot.

  Bước 1:

  Tìm phần tử chính giữa mảng:

  pivot = array[(low+high)/2];

  Bước 2:

  Tìm phần tử lớn hơn bên trái pivot và phần tử nhỏ hơn bên phải pivot và đổi chỗ cho nhau. Nếu i > j, dừng và trả về vị trí i là vị trí chia mảng thành 2 phần.

  i = low, j = high;

  while(i<j):

  ```
      while(a[i]<pivot):
          i++;
  ```

```
            while(a[j]>pivot):
                    j--;
            if(i<=j):
                    swap(a[i], a[j]);
                    i++;
                    j--;
    return i;
```

- Hàm quickSort(array[], low, high):

  Bước 1:

  Đưa các phần tử nhỏ hơn pivot về bên trái, các phần tử lớn hơn pivot về bên phải và trả về vị trí của pivot bằng cách gọi hàm partition:

  pi = partition(array[], low, high);

  Bước 2:

  Nếu low nhỏ hơn pi-1, gọi hàm đệ quy quickSort với low và high = pi-1:

```
if(low < pi-1):
        quickSort(array, low, pi-1);
```

  Bước 3:

  Nếu pi nhỏ hơn high, gọi hàm đệ quy quickSort với low = pi và high:

```
if(pi < high):
        quickSort(array, pi, high);
```

  Bước 4: Mảng ban đầu đã được sắp xếp.

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- Xét các phần tử từ vị trí 0 đến 4, chọn pivot = 6
- Chuyển các phần tử nhỏ hơn 6 sang trái và các phần tử lớn hơn 6 sang bên phải

5	2	6	7	9

- Xét các phần tử ở vị trí từ 0 đến 1, chọn pivot = 5, chuyển các phần tử lớn hơn 5 sang phải và nhỏ hơn 5 sang trái

2	5	6	7	9

- Xét các phần tử ở vị trí từ 3 đến 4, chọn pivot = 7, chuyển các phần tử lớn hơn 7 sang phải và nhỏ hơn 7 sang trái

2	5	6	7	9

- Kết quả: mảng đã được sắp xếp

d. Đánh giá độ phức tạp

- Trường hợp xấu nhất xảy ra khi mẫu dữ liệu đã được sắp xếp nhưng ta chọn

pivot là phần tử trái nhất hoặc mảng sắp xếp ngược nhưng ta chọn phần tử phải nhất làm pivot. Khi đó, các phần tử sau mỗi lần chia mảng thành 2 thì đều nằm cùng 1 phía.

- Trường hợp trung bình và tốt nhất là ta chọn phần tử ở giữa làm pivot, khi đó các phần tử được chia vào 2 nửa, ta sẽ mất O(n) để duyệt qua các phần tử và O(log(n)) cho các lần đệ quy.

	Time	Space
Best case	O(nlog(n))	O(log(n))
Average case	O(nlog(n))	O(1)
Worst case	$O(n^2)$	O(n)

e. Ứng dụng

Thuật toán Quick sort được sử dụng trong các trường hợp:

+ Được sử dụng ở mọi nơi mà không cần đến sự ổn định
+ Chi phí hoán đổi không nhiều
+ Tiết kiệm thời gian

6. Thuật toán Radix Sort

a. Ý tưởng

Radix Sort là một kỹ thuật sắp xếp các phần tử bằng cách nhóm các chữ số riêng lẻ của một giá trị có cùng một vị trí. Sau đó, sắp xếp các phần tử theo thứ tự tăng hoặc giảm. Radix Sort dựa trên nguyên tắc phân loại thư của bưu điện. Nó không quan tâm đến việc so sánh giá trị của phần tử và bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

b. Các bước chạy

radixSort(a[ ], N)

Bước 1: Khởi tạo mảng "output" với N phần tử (int* output = new int[n]) và max = a[0] và exp = 1;

Bước 2: Duyệt mảng a[i] với (i chạy từ 0 đến N) .

Nếu a[i] lớn hơn max thì cập nhật lại max

max=a[i];

Bước 3: Khi max / exp > 0 :

Khởi tạo mảng bucket[ ] 10 phần tử (bucket[10] = { 0 })

Duyệt mảng a[i] với (i chạy từ 0 đến N). Đếm các số hàng đơn vị từ 0,..,9

```
for (int i = 0; i < n; i++)
bucket[a[i] / exp % 10]++;
```

Cộng dồn trên mảng bucket để xác định vị trí sắp xếp lại cho từng chữ số

```
for (int i = 1; i < 10; i++)
bucket[i] += bucket[i - 1];
```

Sắp xếp các phần tử từ mảng bucket sang mảng output

```
for (int i = n - 1; i >= 0; i--)
output[--bucket[a[i] / exp % 10]] = a[i];
```

Duyệt và gán phần tử trong mảng a[i] = output [i] (với i chạy từ 0 đến N)

```
for (int i = 0; i < n; i++)
a[i] = output[i];
```

Gán lại exp gấp 10 lần

```
exp=exp * 10;
```

Bước 4: Nếu max / exp > 0: quay lại Bước 3

Ngược lại: Xóa mảng output[ ], Dừng!

c. Ví dụ

- Cho mảng sau với n = 5

567	90	61	7	212

- Vị trí hàng đơn vị

90	61	212	567	7

- Vị trí hàng chục

7	212	61	567	90

- Vị trí hàng trăm

7	61	90	212	567

d. Đánh giá độ phức tạp

- Xét một mảng gồm n phần tử, các phần tử trong mảng có tối đa m chữ số thì:
- Số lần chia nhóm các phần tử: m lần
- Trong mỗi lần chia nhóm và gộp lại thành mảng, các phần tử chỉ được xét đúng 1 lần. Vì vậy, độ phức tạp của thuật toán Radix Sort là O(2mn) = O(n)

7. Thuật toán Bubble Sort

a. Ý tưởng

Sắp xếp nổi bọt là một thuật toán kiểu so sánh đơn giản, với thao tác cơ bản là sắp xếp 2 phần tử kề nhau, nếu chúng chưa đứng đúng vị trí thì đổi chỗ cho nhau.

b. Các bước chạy

bubbleSort(a[ ], N)

Bước 1: Khởi tạo i=0;

Bước 2: Gán swapped = false

Bước 3: Khởi tạo j=0;

Bước 4: Nếu a[j] > a[j+1] thì hoán vị hai phần tử với nhau và gán swapped=true

```
if (a[j] > a[j + 1])
{
        HoanVi(a[j], a[j + 1]);
```

```
                    swapped = true;
                }
```

Bước 5: Nếu j<n-i-1: quay lại Bước 4 với j=j+1;
Ngược lại: qua Bước 6

Bước 6: Nếu swapped = false thì Dừng!

```
if (swapped == false)
    break;
```

Nếu i<n-1: quay lại Bước 2 với i=i+1
Ngược lại: Dừng!

c. Ví dụ

- Cho mảng sau với n = 5

5	**9**	6	7	2
5	**6**	**9**	7	2
5	6	**7**	**9**	2
5	6	7	**2**	**9**
5	**6**	2	7	9
5	**2**	**6**	7	9
2	5	**6**	**7**	9

-

d. Đánh giá độ phức tạp

_ Số phép so sánh: $(n-1)+(n-2)+(n-3)+\cdots+1=n(n-1)/2$ gần bằng n^2 Độ phức tạp: $O(n^2)$

_ Ngoài ra, chúng ta có thể phân tích độ phức tạp bằng cách quan sát số vòng lặp. Có 2 vòng lặp nên độ phức tạp là $n \times n = n^2$

	Time	Space

Best case	O(n)	O(1)
Average case	$O(n^2)$	O(1)
Worst case	$O(n^2)$	O(1)

e. Ứng dụng

Thuật toán Bubble sort được sử dụng trong các trường hợp:

+ Mảng có ít phần tử
+ Chi phí hoán đổi không thành vấn đề
+ Kiểm tra từng cặp phần tử là bắt buộc
+ Không quan trọng vấn đề thời gian

8. Thuật toán Shaker Sort

a. Ý tưởng

Shaker Sort là một cải tiến của Bubble Sort. Sau khi đưa phần tử nhỏ nhất về đầu dãy, thuật toán sẽ giúp chúng ta đưa phần tử lớn nhất về cuối dãy. Do đưa các phần tử về đúng vị trí ở cả hai đầu nên thuật toán sắp xếp cocktail sẽ giúp cải thiện thời gian sắp xếp dãy số.

b. Các bước chạy

shakerSort(a[ ], N)

Bước 1: Khởi tạo các giá trị left, right và k

```
int left = 0;
int right = n - 1;
int k = 0;
```

Bước 2: Khi left < right

Duyệt các phần tử a[i] (với i đi từ left đến right). Nếu a[i] > a[i+1] (Phần tử bên trái lớn hơn bên phải) thì Hoán vị a[i] với a[i+1] và gán k bằng i

```
for (int i = left; i < right; i++)
{
    if ( a[i] > a[i + 1])
    {
        HoanVi(a[i], a[i + 1]);
        k = i;
    }
}
```

Gán right bằng k

```
right = k;
```

Duyệt các phần tử a[i] (với i đi từ right đến left). Nếu a[i] < a[i-1] (Phần tử bên trái lớn hơn bên phải) thì Hoán vị a[i] với a[i-1] và gán k=i

```
for (int i = right; i > left; i--)
{
    if ( a[i] < a[i - 1])
```

```
                {
                    HoanVi(a[i], a[i - 1]);
                    k = i;
                }
            }
        Gán left bằng k
            left = k;
```

Bước 3: Nếu left < right: quay lại Bước 2
Ngược lại: Dừng!

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- Chạy tiến

5	**9**	6	7	2
5	**6**	**9**	7	2
5	6	**7**	**9**	2
5	6	7	**2**	**9**

- Chạy lùi

5	6	**2**	**7**	9
5	**2**	**6**	7	9
2	**5**	6	7	9

- Mảng đã được sắp xếp, tuy nhiên thuật toán không có cơ chế nhận biết nên sẽ tiếp tục chạy mà không đổi chỗ các phần tử

2	**5**	**6**	7	9
2	5	**6**	**7**	2

2	**5**	**6**	7	2

d. Đánh giá độ phức tạp

- Ta thấy sự chuyển dời của phần tử không có gì là cải tiến (từ vị trí ban đầu của nó, để đi đến vị trí đúng đều mất chi phí như Bubble Sort)
- Chỉ có số phép so sánh là được cải tiến, nhưng chưa tìm được công thức tính số phép so sánh, mặt khác ta thấy chi phí chuyển dời luôn cao hơn chi phí hoán vị thường mất 3 phép gán so sánh nên cải tiến có thể xem như không đáng kể.
- ShakerSort là một dạng nâng cao của BubbleSort nên nó có thể nhận biết được mảng đã được sắp xếp.

	Time	Space
Best case	O(n)	O(1)
Average case	$O(n^2)$	O(1)
Worst case	$O(n^2)$	O(1)

e. Ứng dụng

Thuật toán Shaker sort được sử dụng trong các trường hợp

+ Khắc phục được nhược điểm của Bubble sort khi duyệt mảng theo hai lượt từ hai phía khác nhau
+ Mảng có ít phần tử
+ Chi phí hoán đổi không thành vấn đề
+ Không quan trọng vấn đề thời gian

9. Thuật toán Shell Sort

a. Ý tưởng

Shell sort là một thuật toán sắp xếp các phần tử cách xa nhau ban đầu và liên tiếp giảm khoảng cách giữa các phần tử được sắp xếp. Nó là một phiên bản tổng quát của sắp xếp chèn.

b. Các bước chạy

shellSort(a[ ], N)

Bước 1: Khởi tạo gap = n/2

Bước 2: Khởi tạo i = gap

Bước 3: Khởi tạo biến temp bằng a[i] và j

```
int temp = a[i];
int j;
```

Bước 4: Duyệt mảng a[j] (với j=i từ j đến j-gap). Gán a[j] bằng a[j-gap] chỉ khi j lớn hơn hoặc bằng gap và a[j - gap] lớn hơn temp

```
for (j = i; j >= gap && a[j - gap] > temp; j -= gap)
    a[j] = a[j - gap];
```

Bước 5: Gán a[j] bằng temp

a[j] = temp;

Nếu i < N quay lại Bước 3 với i = i + 1

Ngược lại: qua bước 6

Bước 6: Nếu gap > 0 quay lại Bước 2 với gap = gap / 2

Ngược lại: Dừng!

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- Interval = 2
- Vì 5 < 6 nên giữ nguyên vị trí

5	9	**6**	7	2

- Đổi chỗ 9 và 7

5	**7**	6	**9**	2

- Đổi chỗ 6 và 2

5	7	**2**	9	**6**

- Đổi chỗ 5 và 2

2	7	**5**	9	6

- Interval = 1, làm tương tự sắp xếp chèn

2	7	5	9	6

2	**7**	5	9	6

2	**5**	**7**	9	6

2	**5**	**7**	**9**	6

2	**5**	**6**	**7**	**9**

d. Đánh giá độ phức tạp

	Time	Space
Best case	O(n log(n))	O(1)
Average case	O(n log(n))	O(1)
Worst case	$O(n^2)$	O(1)

10. Thuật toán Counting Sort

a. Ý tưởng

Sắp xếp bằng phép đếm phân phối là 1 trong những giải thuật sắp xếp thuộc nhóm Distribution Sort. Đây là một thuật toán sắp xếp đơn giản cho trường hợp đặc biệt các giá trị trong mảng cần sắp xếp đều là số nguyên và biết được giá trị của mảng nằm trong khoảng nào đó.

b. Các bước chạy

countingSort(a[ ], N)

Bước 1: Khởi tạo mảng output với N phần tử và max bằng a[0]

```
int* output = new int[n];
int max = a[0];
```

Bước 2: Duyệt mảng a[i] (với i từ 1 đến N) để tìm giá trị lớn nhất trong mảng đã cho

```
for (int i = 1; i < n; i++)
{
        if ( a[i] > max)
                max = a[i];
}
```

Bước 3: Khởi tạo mảng count với max + 1 phần tử

```
int* count = new int[max + 1];
```

Bước 4: Gán cho mảng count từ phần tử đầu đến phần tử cuối bằng 0

```
for (int i = 0; i <= max; i++)
        count[i] = 0;
```

Bước 5: Lưu trữ số đếm của từng phần tử tại chỉ số tương ứng của chúng trong mảng count

```
for (int i = 0; i < n; i++)
        count[a[i]]++;
```

Bước 6: Lưu trữ tổng tích lũy của các phần tử của mảng count

```
for (int i = 1; i <= max; i++)
        count[i] += count[i - 1];
```

Bước 7: Tìm chỉ số của từng phần tử của mảng ban đầu trong mảng count. Điều này sẽ cho biết số đếm tích lũy. Sau khi đặt mỗi phần tử vào đúng vị trí của nó, ta sẽ giảm số đếm của nó đi một đơn vị

```
for (int i = n - 1; i >= 0; i--)
{
```

```
            output[count[a[i]] - 1] = a[i];
            count[a[i]]--;
        }
```

Bước 8: Xóa mảng count và mảng output, Dừng!

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- Đếm số lần xuất hiện của các phần tử

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

- Cập nhật vị trí xuất hiện của các phần tử trong mảng

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

- Sắp xếp lại mảng ban đầu
- Phần tử 5 ở vị trí thứ 1

0	1	2	3	4	**5**	6	7	8	9
0	0	1	1	1	**1**	3	4	4	5

	5			

- Phần tử 9 ở vị trí 4

0	1	2	3	4	5	6	7	8	**9**
0	0	1	1	1	1	3	4	4	**4**

	5			9

- Phần tử 6 ở vị trí 2

0	1	2	3	4	5	6	7	8	**9**
0	0	1	1	1	1	3	4	4	**4**

	5	6		9

- Phần tử 7 ở vị trí 3

0	1	2	3	4	5	6	**7**	8	9

0	0	1	1	1	1	3	**3**	4	4

	5	6	7	9

- Phần tử 2 ở vị trí 0

0	1	**2**	3	4	5	6	7	8	9
0	0	**0**	1	1	1	3	3	4	4

2	5	6	7	9

- Kết quả: mảng đã được sắp xếp

d. Đánh giá độ phức tạp

- Có bốn vòng lặp chính. (Việc tìm giá trị lớn nhất có thể được thực hiện ngoài hàm). Độ phức tạp là: O(max)+O(size)+O(max)+O(size)=O(max+size)
- Trong tất cả các trường hợp trên, độ phức tạp là như nhau vì cho dù các phần tử được đặt trong mảng như thế nào thì thuật toán cũng trải qua n+k lần.
- Không có sự so sánh giữa bất kỳ phần tử nào, vì vậy nó tốt hơn so với kỹ thuật sắp xếp dựa trên so sánh. Tuy nhiên, sẽ là sai lầm nếu các số nguyên là rất lớn vì mảng phải có kích thước như vậy sẽ phải được tạo ra.
- Độ phức tạp không gian của sắp xếp đếm là O(max). Phạm vi phần tử càng lớn thì độ phức tạp về không gian càng lớn.

	Time	Space
Best case	O(n+k)	O(max)
Average case	O(n+k)	O(max)
Worst case	O(n+k)	O(max)

e. Ứng dụng

Thuật toán Counting sort được sử dụng trong các trường hợp:

+ Mảng có ít số nguyên khác nhau hơn so với kích thước (Có nhiều phần tử trùng trong 1 mảng)
+ Độ phức tạp là tuyến tính

11. Thuật toán Flash Sort

a. Ý tưởng

Flash sort là một thuật toán sắp xếp tại chỗ (không dùng mảng phụ), không đệ quy, gồm có 3 bước:

1. Phân lớp dữ liệu, tức là dựa trên giả thiết dữ liệu tuân theo 1 phân bố nào đó, chẳng hạn phân bố đều, để tìm 1 công thức ước tính vị trí (lớp) của phần tử sau khi sắp xếp.
2. Hoán vị toàn cục, tức là dời chuyển các phần tử trong mảng về lớp của mình.
3. Sắp xếp cục bộ, tức là để sắp xếp lại các phần tử trong phạm vi của từng lớp.

b. Các bước chạy

flashSort(array[], n): nhận vào mảng array cần sắp xếp và số lượng phần tử n.

Bước 1: Phân lớp dữ liệu

- Tìm giá trị của phần tử nhỏ nhất trong mảng và vị trí của phần tử lớn nhất của các phần tử trong mảng:

```
minVal = a[0];
max = 0;
for(i=1; i<n; i++):
        if(array[i] < minVal):
                minVal = array[i];
        if(array[i] > array[max]:
                max = i;
if(a[max] == minVal) return;
```

- Khởi tạo 1 vector có m phần tử, trong đó m là số lượng lớp được tính bằng công thức m = 0.45*n:

```
m = (int) n*0.45;
l[m];
for(i=0; i<m; i++):
        l[i] = 0;
```

- Đếm số lượng phần tử của các lớp, phần tử a[i] sẽ thuộc lớp k được tính theo công thức k = (int) (m - 1) * (a[i]-minVal) / (a[max] - minVal):

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

- Tính vị trí kết thúc của phân lớp thứ j theo công thức a[j] = a[j] + a[j-1] (1 < j < m-1):

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

Bước 2: Hoán vị toàn cục

```
nmove = 0, j = 0, k = m-1;
```

- Hoán vị phần tử đầu tiên và phần tử lớn nhất bởi vì ta khởi tạo k = m-1 mà a[max] thuộc phân lớp cuối cùng:

```
HoanVi(a[max], a[0]);
```

- Đưa phần tử a[i] về đúng phân lớp của nó:

```
while(nmove < n-1):
```

```
while(j > l[k] - 1):
        j++;
        k = (int) (c1*array[j] - minVal);
flash = array[j];
if(k<0) break;
while(j != l[k]):
        k = (int) (c1*flash - minVal);
        --l[k];
        swap(flash, a[l[k]]);
        nmove++;
```

Bước 3: Hoán vị cục bộ

- Các phần tử đã nằm đúng phân lớp của mình, ta sử dụng thuật toán insertionSort để sắp xếp lại các phần tử trong mỗi phân lớp:
  insertionSort(array[], n);

c. Ví dụ

- Cho mảng sau với n = 5

5	9	6	7	2

- m = (int) 0.45*5 = 2, minVal = 2, max = 1
- l[2] = {3, 2}

5 (0)	9 (1)	6 (0)	7 (1)	2 (0)

- Tính vị trí kết thúc:
  l[2] = {3, 5}
- Hoán vị toàn cục

9 (1)	**5 (0)**	6 (0)	7 (1)	2 (0)

2 (0)	5 (0)	6 (0)	7 (1)	**9 (1)**

2 (0)	5 (0)	6 (0)	7 (1)	9 (1)

2 (0)	**5 (0)**	6 (0)	7 (1)	9 (1)

2 (0)	5 (0)	**6 (0)**	7 (1)	9 (1)

2 (0)	5 (0)	6 (0)	**7 (1)**	9 (1)

d. Đánh giá độ phức tạp

- Thuật toán flash sort là thuật toán sắp xếp tại chỗ, không dùng mảng phụ nên độ phức tạp không gian là O(1).

	Time	Space

Best case	O(n)	O(1)
Average case	O(n)	O(1)
Worst case	O(n)	O(1)

- Trong tất cả các trường hợp ta sẽ cần tối đa nmove < n-1 bước để thực hiện sắp xếp các phần tử nên độ phức tạp trong tất cả trường hợp có thể ước lượng là O(n)

Bảng số liệu và biểu đồ

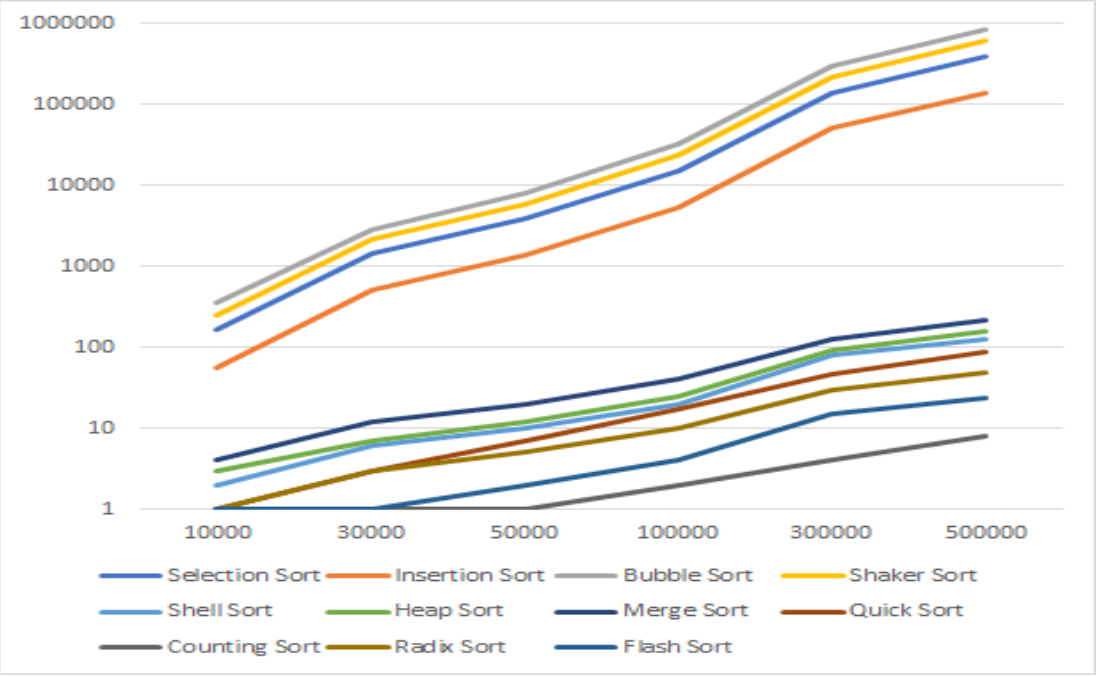
1. Random data

a. Bảng số liệu

Data order: Random data

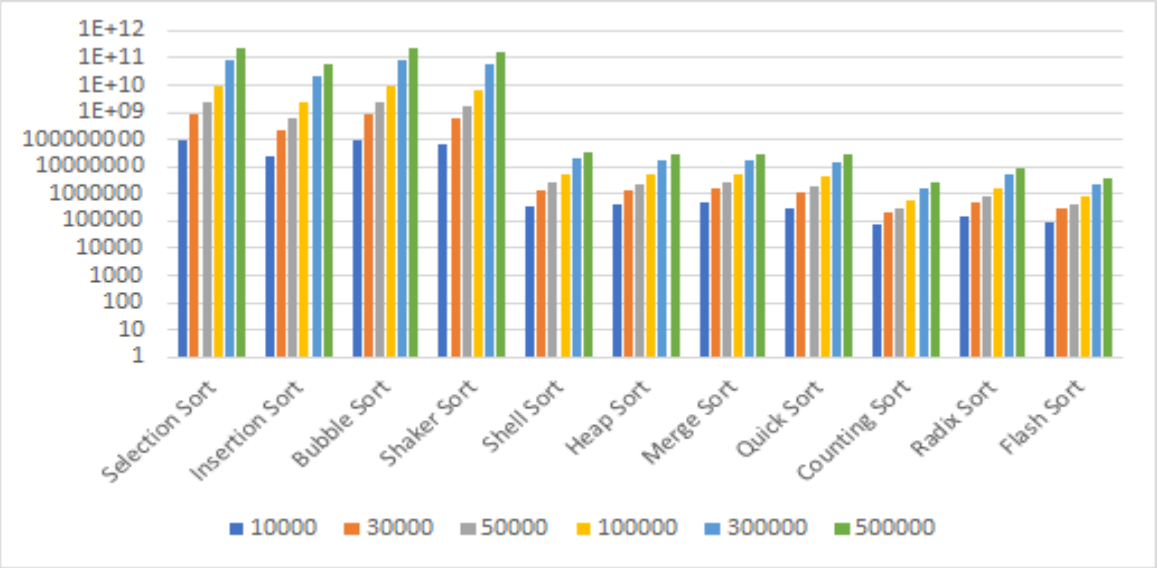
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Time	Time	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare
Selection Sort	164	100009999	1425	900029999	3906	2500049999	15389	10000099999	138694	90000299999	386652	2.5E+11
Insertion Sort	57	24900906	508	226068272	1349	621526634	5434	2493467651	50625	22510272287	140133	62385001862
Bubble Sort	355	99994720	2876	900054376	8099	2500065032	32512	10000024440	301357	90000059776	829531	2.5E+11
Shaker Sort	245	66222065	2180	602203495	5972	1657416341	23926	6658076103	219061	60045180472	613206	1,66E+11
Shell Sort	2	380602	6	1357844	10	2518807	20	5720778	79	20143355	126	36159222
Heap Sort	3	392805	7	1319011	12	2313311	25	4924557	93	16187779	158	28074017
Merge Sort	4	457607	12	1512959	20	2637024	40	5574359	128	18136196	216	31289026
Quick Sort	1	299289	3	1107634	7	1987346	17	4184941	47	14686511	87	28463320
Counting Sort	0	70003	1	210001	1	315539	2	565539	4	1565539	8	2565539
Radix Sort	1	140058	3	510072	5	850072	10	1700072	30	5100072	49	8500072
Flash Sort	0	83071	1	272753	2	429536	4	877071	15	2388095	24	3963737

b. Biểu đồ thời gian chạy



- Thuật toán nhanh nhất: Counting sort. Các thuật toán theo sau là Flash sort, Radix sort, Quick sort, Shell sort, Heap sort, Merge sort cũng có thời gian chạy chênh lệch không nhiều so với counting sort. Độ phức tạp của các thuật toán này trong trường hợp trung bình là O(nlogn) hoặc O(n).
- Thuật toán chậm nhất: Bubble sort. Theo sau đó là các thuật toán Shaker sort, Selection sort. Các thuật toán này đều có độ phức tạp O(n^2) trong trường hợp trung bình, do đó với dữ liệu ngẫu nhiên sẽ cho tốc độ rất chậm.

c. Biểu đồ phép so sánh



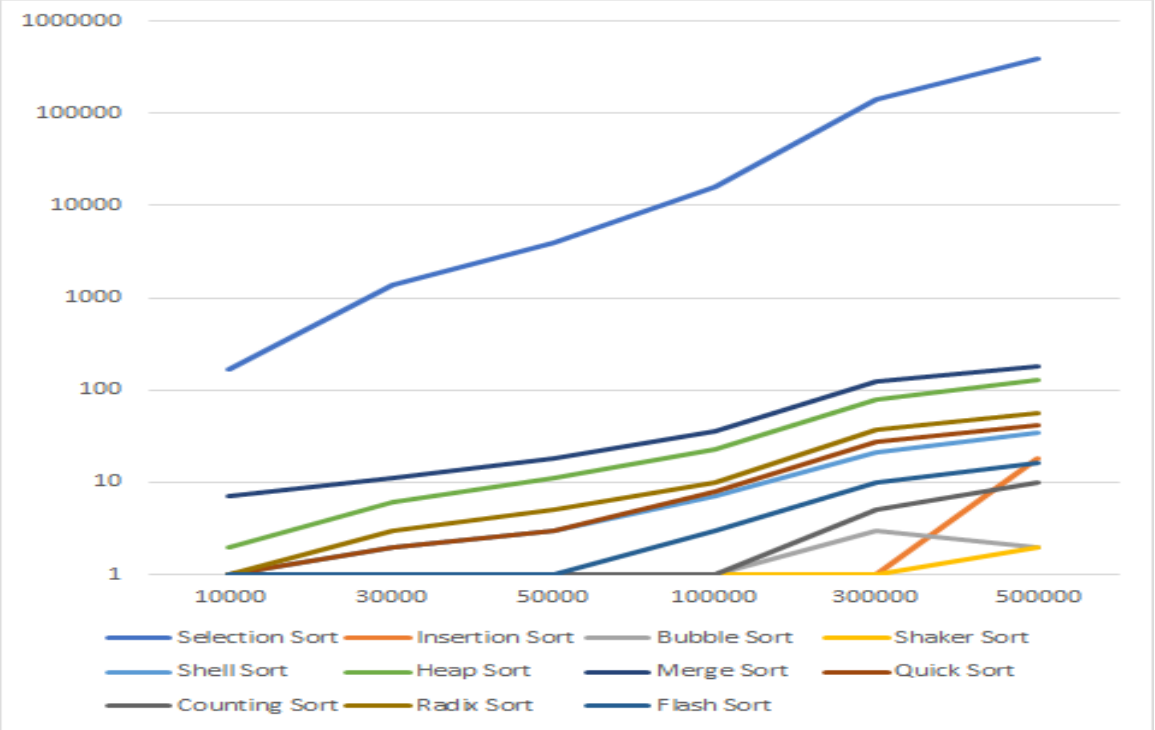
2. Sorted data

a. Bảng số liệu

Data order: Sorted data

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare
Selection Sort	170	100009999	1403	900029999	3916	2500049999	15741	10000099999	140889	90000299999	389669	2.5E+11
Insertion Sort	0	19999	0	59999	0	99999	1	199999	1	599999	18	999999
Bubble Sort	0	20001	0	60001	0	100001	0	200001	3	600001	2	1000001
Shaker Sort	0	20002	0	60002	0	100002	0	200002	1	600002	2	1000002
Shell Sort	0	240037	2	780043	3	1400043	7	3000045	21	10200053	35	17000051
Heap Sort	2	415869	6	1380301	11	2419913	23	5152563	79	16833481	128	29067101
Merge Sort	7	406234	11	1332186	18	2320874	36	4891754	125	15848682	179	27234634
Quick Sort	1	233166	2	790879	3	1401729	8	3003427	28	9920392	41	17213607
Counting Sort	0	7003	1	210003	1	350003	1	700003	5	21000063	10	3500003
Radix Sort	1	140058	3	510072	5	850072	10	1700072	37	6000086	57	10000086
Flash Sort	0	107995	1	323995	1	539995	3	1079995	10	3239995	16	5399995

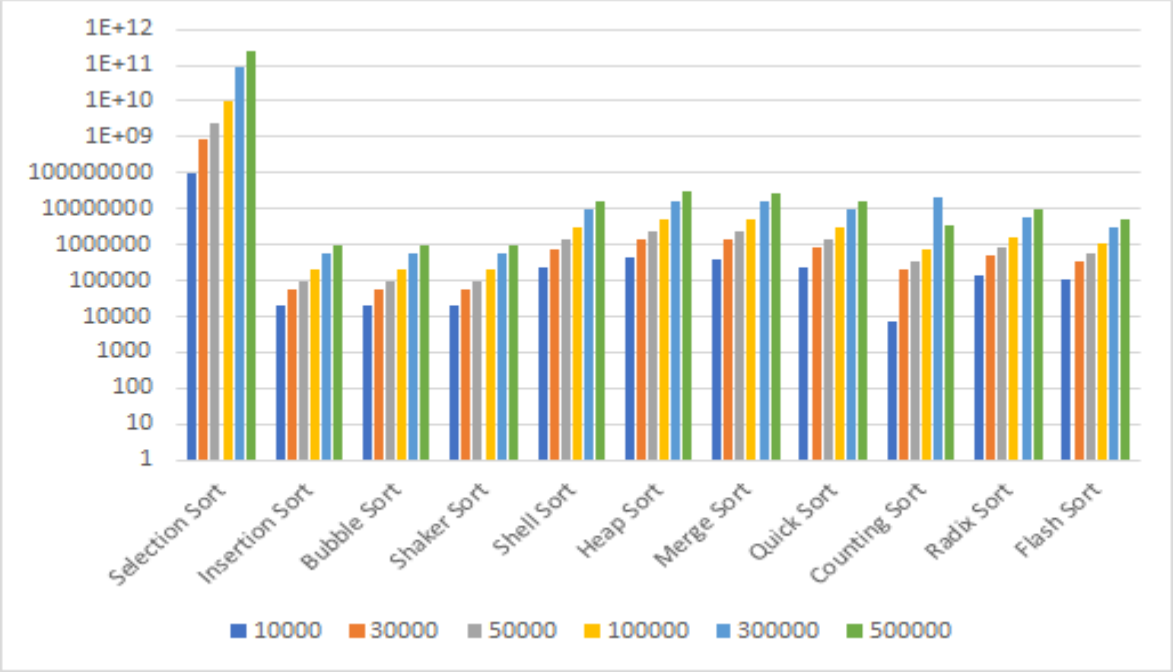
b. Biểu đồ thời gian chạy



- Nhận xét

- Thuật toán nhanh nhất: Shaker sort. Các thuật toán sau cũng cho tốc độ tốt với dữ liệu đã được sắp xếp: bubble sort, insertion sort, counting sort, flash sort, shell sort, quick sort, radix sort, heap sort, merge sort.
- Thuật toán chậm nhất: Selection sort.
- Giải thích: Shaker sort là một thuật toán cải tiến của bubble sort. Bởi vì dữ liệu đầu vào đã được sắp xếp nên thuật toán này chỉ cần so sánh 2 phần tử kề nhau mà không cần thực hiện phép đổi chỗ nào. Thuật toán Selection sort luôn luôn có độ phức tạp O(n^2) cho dù dữ liệu đã được sắp xếp.

c. Biểu đồ về phép so sánh



- Nhận xét

- Thuật toán ít phép so sánh nhất: Bubble sort và Shaker sort. 2 thuật toán này chỉ chênh lệch 1 phép so sánh.
- Thuật toán nhiều phép so sánh nhất: Selection sort
- Giải thích: Shaker sort là một thuật toán cải tiến của bubble sort. Bởi vì dữ liệu đầu vào đã được sắp xếp nên thuật toán này chỉ cần so sánh 2 phần tử kề nhau mà không cần thực hiện phép đổi chỗ nào. Thuật toán Selection sort luôn luôn có độ phức tạp O(n^2) cho dù dữ liệu đã được sắp xếp.

3. Reverse data

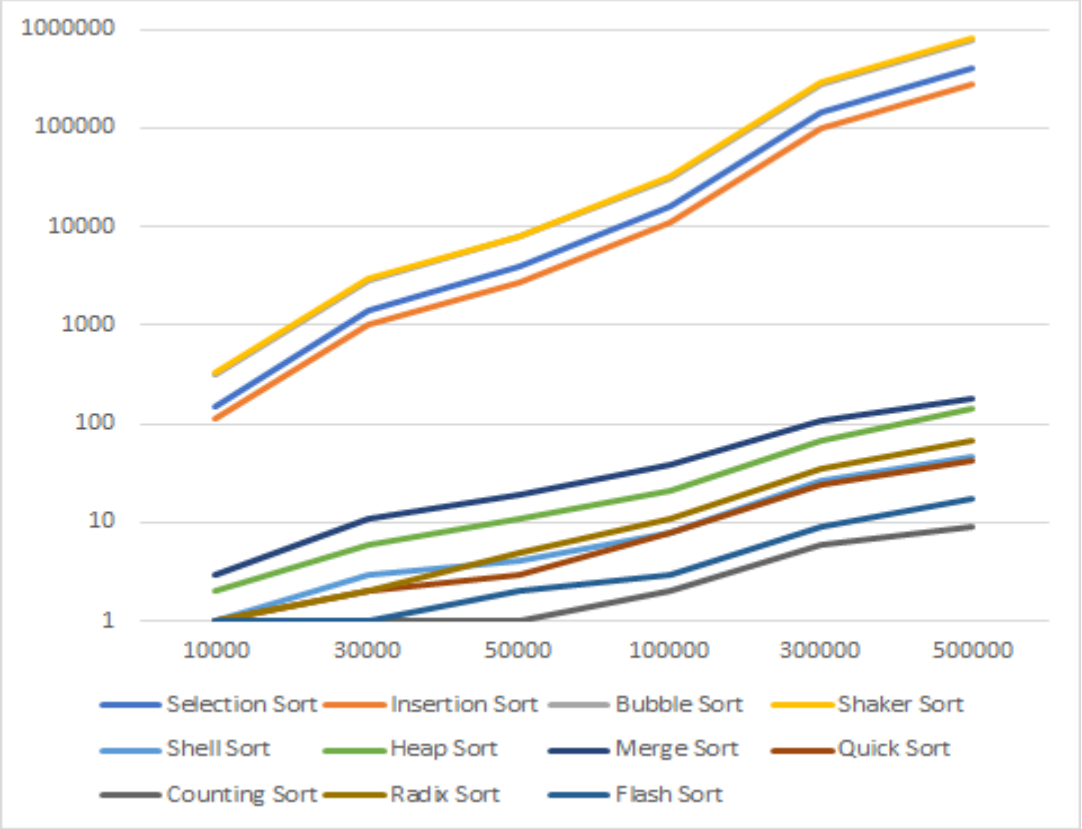
a. Bảng số liệu

Data order: Reserve data

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare
Selection Sort	152	100009999	1431	900029999	3989	2500049999	15913	10000099999	142443	90000299999	396180	2.5E+11
Insertion Sort	112	50014999	1003	450044999	2745	1250074999	11060	5000149999	99630	45000449999	276835	1,25E+11
Bubble Sort	315	100019998	2863	900059998	7926	2500099998	31368	10000199998	281206	90000599998	788486	2,50E+11
Shaker Sort	327	100005001	2915	900015001	8070	2500025001	31978	10000050001	287935	90000150001	805567	2.5E+11
Shell Sort	1	302597	3	987035	4	1797323	8	3844605	27	12700933	46	21428803
Heap Sort	2	370089	6	1257637	11	2196677	21	4692303	69	15528739	140	27005351
Merge Sort	3	401834	11	1323962	19	2301434	38	4852874	110	15729866	180	27143914
Quick Sort	1	235002	2	800207	3	1398325	8	2996614	24	9988669	42	17405058
Counting Sort	0	70003	1	210003	1	350003	2	700003	6	2100003	9	3500003

Radix Sort	0	140058	2	510072	5	850072	11	1700072	35	6000086	67	10000086
Flash Sort	1	94502	1	283502	2	472502	3	945002	9	2835002	17	4725002

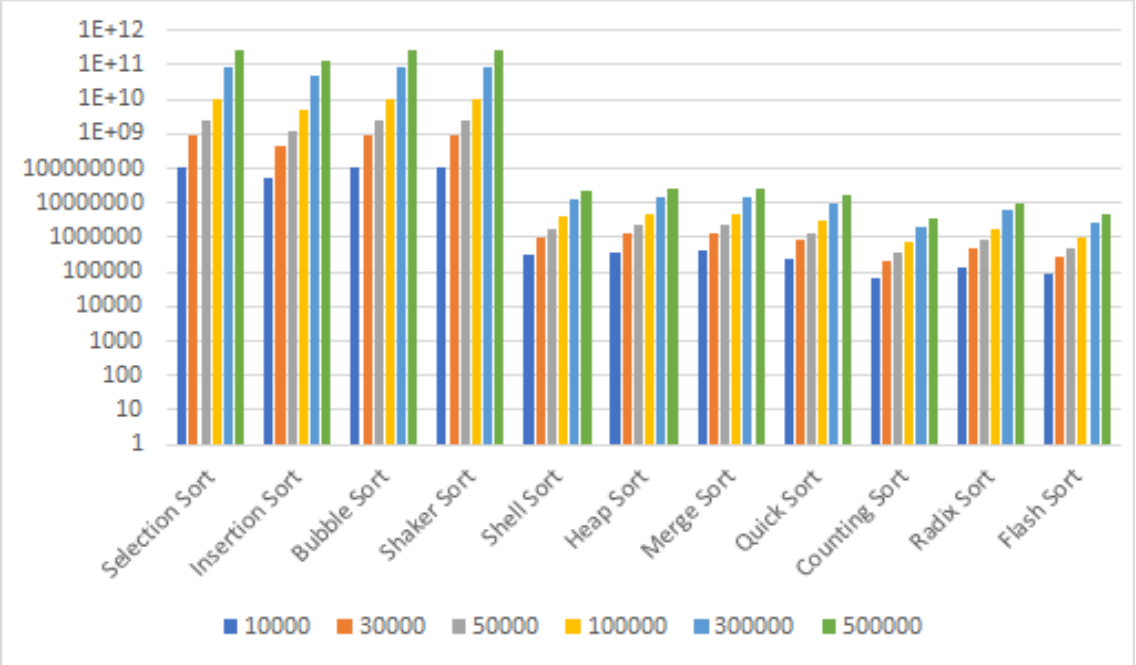
b. Biểu đồ thời gian chạy



- Nhận xét:

- Từ biểu đồ trên ta thấy có thuật toán Selection Sort, Insertion Sort, Bubble Sort và Sharker Sort có thời gian chạy rất lớn vì nhìn thấy khoảng cách giữa 4 thuật toán trên với các thuật toán khác cách xa nhau, phần tử trong mảng càng lớn thời gian chạy càng lâu.
- Còn các thuật toán còn lại thời gian chạy rất nhanh khoảng 100ms trừ Merge Sort và Heap Sort nhưng cũng không quá lớn. Trong đó Counting Sort có thời gian chạy là ngắn nhất không qua 10ms.
- Giải thích: Selection sort có độ phức tạp O(n^2) trong mọi trường hợp. Bubble sort, Shaker sort và Insertion sort hoạt động tốt nhất với dữ liệu đã được sắp xếp. Tuy nhiên, đối với dữ liệu sắp xếp ngược các thuật toán cho tốc độ rơi vào trường hợp xấu nhất.

c. Biểu đồ phép so sánh



- Nhận xét:

- Từ biểu đồ trên ta thấy ba thuật toán Selection Sort, Insertion Sort, Bubble Sort và Sharker Sort có số phép so sánh rất lớn, lớn hơn nhiều so với các thuật toán khác. Trong bốn thuật toán đó ta thấy không có sự chênh lệch quá lớn.
- Còn các thuật toán còn lại các phép so sánh không có sự chênh lệch lớn với nhau.
- Giải thích: Selection sort có độ phức tạp O(n^2) trong mọi trường hợp. Bubble sort, Shaker sort và Insertion sort hoạt động tốt nhất với dữ liệu đã được sắp xếp. Tuy nhiên, đối với dữ liệu sắp xếp ngược các thuật toán cho tốc độ rơi vào trường hợp xấu nhất.

4. Nearly sorted data

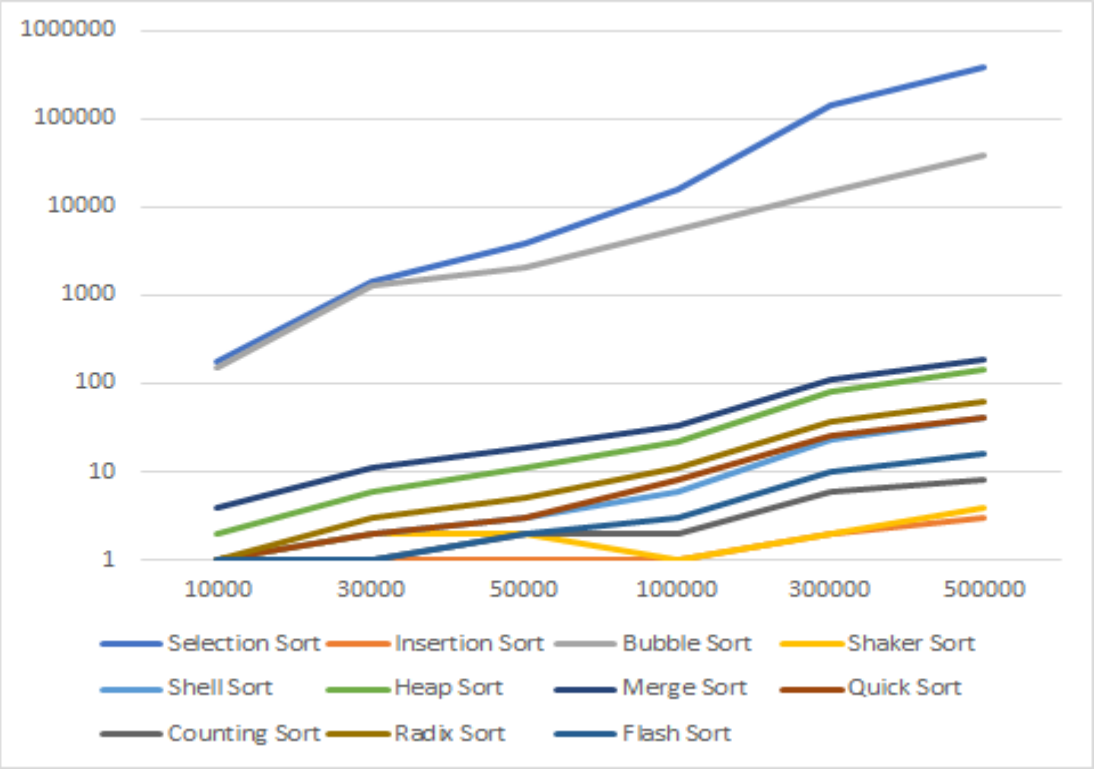
a. Bảng số liệu

Data order: Nearly sorted data

Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare	Time	Compare
Selection Sort	173	100009999	1411	900029999	3931	2500049999	15608	10000099999	140684	90000299999	392223	2.5E+11
Insertion Sort	1	97739	1	344025	1	297667	0	358987	2	758263	3	1296587
Bubble Sort	150	98814397	1300	889652925	2106	1416044376	5448	3649225752	14902	9984031617	38539	25590361601
Shaker Sort	1	190343	2	667358	2	547278	1	570688	2	985185	4	1629681
Shell Sort	1	262509	2	849529	3	1483165	6	3079639	23	10260091	40	17081203
Heap Sort	2	415419	6	1380217	11	2419715	22	5152155	80	16833631	140	29067071
Merge Sort	4	423989	11	1383543	19	2367715	34	4945076	110	15900384	190	27287564
Quick Sort	0	233438	2	792235	3	1401655	8	3003416	25	9920479	41	17214236

Counting Sort	1	70003	1	210003	2	350003	2	700003	6	2100003	8	3500003
Radix Sort	1	140058	3	510072	5	850072	11	1700072	36	6000086	61	10000086
Flash Sort	1	107973	1	323971	2	539973	3	1079983	10	3239971	16	5399977

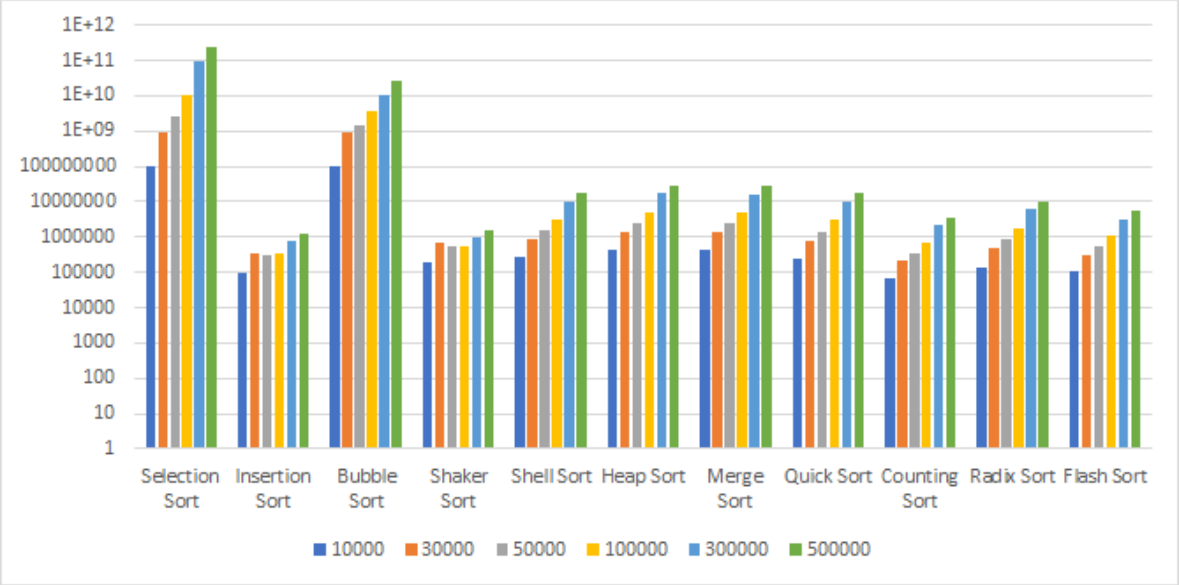
b. Biểu đồ thời gian chạy



- Nhận xét:

- Từ biểu đồ trên ta thấy thuật toán Selection Sort và Bubble Sort có thời gian chạy rất lớn vì thấy rõ sự tách nhóm của hai thuật toán trên với các thuật toán còn lại.
- Những thuật toán còn lại không có sự chênh lệch quá lớn về thời gian chạy nhưng thuật toán Insertion Sort, Shaker Sort và Counting sort có thời gian chạy khá nhanh khi mảng có từ 10000 đến 500000 cũng không đến 10ms.

c. Biểu đồ phép so sánh



- Nhận xét:

- Từ biểu đồ trên ta thấy thuật toán Selection Sort và Bubble Sort có phép so sánh rất lớn có khoảng cách rất lớn với các hàm Sort còn lại trong đó Selection Sort có phép so sánh lớn nhất vì thuật toán là duyệt từng phần tử.
- Các hàm sort khác thì có phép so sánh nhỏ hơn hai hàm trên và tăng đều trừ Insertion Sort và Shaker Sort lúc tăng lúc giảm nhưng chênh lệch không quá lớn.

5. Nhận xét tổng quan các thuật toán sắp xếp:

- Các thuật toán cho tốc độ tốt trong phần lớn các trường hợp: Quick sort, Flash sort, Shell sort, Radix sort, Counting sort và Heap sort.
- Thuật toán cho tốc độ chậm nhất trong mọi trường hợp: Selection sort.
- Đối với từng thuật toán:
  + Selection sort, Bubble sort: Có tốc độ chậm nhất và nhiều phép so sánh nhất trong các trường hợp. Thuật toán này chỉ phù hợp với trường hợp cỡ mẫu nhỏ và yêu cầu sử dụng ít bộ nhớ.
  + Insertion sort, Shaker sort: Có tốc độ tốt với mẫu dữ liệu đã được sắp xếp một phần.
  + Quick sort, Flash sort, Shell sort, Radix sort, Counting sort và Heap sort: Phù hợp với trường hợp mẫu dữ liệu có kích cỡ lớn với giá trị ngẫu nhiên.

Tổ chức Project và ghi chú

Tất cả các file mã nguồn được lưu trong thư mục Source bao gồm:

- **Header file:**
  - sort.h
  - dataGenerator.h
  - inputHandler.h
- **Cpp file:**
  - sort.cpp
  - dataGenerator.cpp
  - inputHandler.cpp
  - main.cpp

Các file trong thư mục gốc:

- **Thư mục Release:**
  - **File thực thi:** 16.exe
- **File báo cáo:** Report.pdf
- **File Checklist:** Checklist.xlsx

Để chạy chương trình, ta sử dụng Terminal để chạy chương trình bằng dòng lệnh. Nếu nhập lệnh **16.exe** chương trình xảy ra lỗi, sửa lại lệnh bằng **.\16.exe**

Cách build File thực thi 16.exe:

- Chương trình được viết trên IDE Visual Studio.
- Chuyển IDE sang chế độ Release và chạy tổ hợp phím Ctrl + F5.
- File thực thi được biên dịch ra thư mục Release.

Tham Khảo

Selection sort:
https://nguyenvanhieu.vn/thuat-toan-sap-xep-selection-sort
Merge sort:
https://www.geeksforgeeks.org/merge-sort
Insertion sort:
https://cafedev.vn/thuat-toan-insertion-sort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh
Heap sort:
https://www.geeksforgeeks.org/heap-sort
Quick sort:
https://www.geeksforgeeks.org/quick-sort
Bubble sort:
https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U
https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat/thuat-toan-sap-xep-theo-co-so-radix-sort
Radix sort:
https://www.geeksforgeeks.org/radix-sort
Shaker sort:
https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U
Counting sort:
https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat/thuat-toan-sap-xep-dem-counting-sort
Shell sort:
https://en.wikipedia.org/wiki/Shellsort
Flash sort:
https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh?fbclid=IwAR23axX75k_8VL9CL0iN-s18MFy2gX3q28G3BZ97B5EhJywjYTO46wzYPL4
Đo lường thuật toán:
https://stackoverflow.com/questions/24844600/how-can-i-measure-the-times-of-my-sorting-algorithms-in-a-single-step