1.Information page:

Nội dung đồ án: tìm hiểu 11 thuật toán sắp xếp : selection sort, insertion sort, bubble sort, shaker sort, shell sort, heap sort, merge sort, quick sort , counting sort, radix sort và flash sort. HIểu rõ cách cài đặt thuật toán, phân tích cách hoạt động của thuật toán cũng như phân tích về độ phức tạp thời gian và không gian.

2.Introduction page:

Báo cáo bao gồm phần cài đặt của các thuật toán sắp xếp, với các kiểu dữ liệu đầu vào khác nhau, được biểu diễn để có thực hiện được trên 5 loại command nhập xuất các thông tin , báo cáo cũng bao gồm các phân tích về ý tưởng, diễn giải cách hoạt động của thuật toán sắp xếp theo từng bước , đưa ra các biến thể khác nhau của các thuật toán. Đồng thời để tổng quan về độ phức tạp thuật toán , báo cáo cũng sẽ bao gồm các biểu đồ đường và các biểu đồ cột thể hiện một cách trực quan về thời gian thi hành thuật toán với từng kiểu dữ liệu cụ thể, số lượng phép so sánh từ đó đưa ra nhận xét về tất cả các loại thuật toán.

3. Algorithm presentation:

Selection Sort

-Idea:

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 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 descriptions:

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

-Complexity evaluations:

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

-Variants: heap sort chính là 1 thuật toán biến thể của selection sort.

Insertion sort

-Ideas: Ý tưởng của thuật toán là chia dãy thành 2 phần , 1 bên là dãy đã đượt sắp xếp và 1 bên là dãy chưa được sắp xếp. Sau đó ta sẽ chọn từng giá trị từ dãy chưa được sắp xếp và đặt vào đúng vị trí của nó trong dãy đã được sắp xếp.

-Step-by-step descriptions:

```
For j = 2 to length[A]
        Do key <- A[j]
                Insert A[j] into the sorted
                i <- j -1
                while i>0 and A[i] > key
                        do A[i + 1] <- A[i]
                                I <- i-1
                A[i+1] <- key<- key
```

-Complexity evaluations:

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

-Improvements: Phiên bản cải tiến của insertion sort là shell sort.

Bubble Sort

-Ideas:Sắp xếp bong bóng là một thuật toán sắp xếp đơn giản. Thuật toán sắp xếp này là thuật toán dựa trên so sánh, trong đó mỗi cặp yếu tố liền kề được so sánh và các yếu tố được hoán đổi nếu chúng không theo thứ tự. Thuật toán này không phù hợp với các tập dữ liệu lớn vì độ phức tạp trung bình và trường hợp xấu nhất của nó là (n 2) trong đó n là số lượng mục.

-Step-by-step descriptions:

```
// Dãy cần sắp xếp
const input = "5 4 3 2 1";
const numbers = input.split(' ');

// Tối đa chỉ so sánh n - 1 cặp số, vì thế j chỉ chạy tới n - 1
// j là đại diện cho số lượng phần tử đã về đúng vị trí
// mỗi lần chạy hết một lượt trong vòng lặp này
// luôn tối thiểu có 1 số về đúng vị trí
for (let j = 0; j < numbers.length - 1; j++) {
  // Biến đánh dấu có xảy ra sự kiện đổi chỗ không?
  let sw = false;

  // Tiến hành so sánh các cặp số
  // So sánh tối đa n - j - 1 cặp số
  for (let i = 0; i < numbers.length - j - 1; i++) {
    // Nếu phần tử trước lớn hơn phần tử sau
    if (numbers[i] > numbers[i + 1]) {
      // Đổi chỗ cặp số
```

```
      const temp = numbers[i];
      numbers[i] = numbers[i + 1];
      numbers[i + 1] = temp;


      // đánh dấu là có sự kiện đổi chỗ
      sw = true;
    }
  }

  // Sau khi so sánh một lượt các cặp, mà không xảy ra sự kiện đổi chỗ
  // tức là dãy đã được sắp xếp, thì ta không cần chạy vòng lặp nữa
  // break luôn để tiếp kiện thời gian chạy
  if (!sw) {
    break;
  }
}
```

-Optimize:

dừng lặp nếu không có phép so sánh thành công

```
// Optimized implementation of Bubble sort
#include <bits/stdc++.h>
using namespace std;

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
  int i, j;
  bool swapped;
  for (i = 0; i < n-1; i++)
  {
    swapped = false;
    for (j = 0; j < n-i-1; j++)
```

```
    {
      if (arr[j] > arr[j+1])
      {
        swap(arr[j], arr[j+1]);
        swapped = true;
      }
    }

    // IF no two elements were swapped
    // by inner loop, then break
    if (swapped == false)
      break;
  }
}
```

-Biến thể đệ quy:

```
/ C++ code for recursive bubble sort algorithm
#include <iostream>
using namespace std;
void bubblesort(int arr[], int n)
{
   if (n == 0 || n == 1) {
      return;
   }
   for (int i = 0; i < n - 1; i++) {
      if (arr[i] > arr[i + 1]) {
         swap(arr[i], arr[i + 1]);
      }
   }
   bubblesort(arr, n - 1);
}
```

-Complexity evaluations:

Best	Average	Worst	Memory

O(n)	$O(n^2)$	$O(n^2)$	O(1)

-Variants: một biến thể của bubble sort là shaker sort.

Shaker sort

-Ideas:Ý tưởng của shaker sort là biến thể của bubble sort, khác ở chỗ thay vì lặp đi lặp lại sắp xếp dãy từ trái qua phải thì shaker sort luân phiên sắp xếp từ trái qua phải rồi từ phải qua trái. Do đó độ hiệu quả của shaker sort tốt hơn bubble sort.

-Step-by-step descriptions:

```
L <- 0, R <- n – 1
While (L < R)
    For i = L to R
        If (a[i] > a[i+1])
            Swap(a[i],a[i+1])
    R <- R - 1
    For i=R – 1 down to L
        If (a[i]>a[i+1])
            Swap(a[i],a[i+1])
    L <- L + 1
```

-Complexity evaluations:

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

Shell sort

-Ideas:

Shell Sort: là một biến thể chính của Insertion Sort. Trong insertion sort, ta di chuyển các phần tử chỉ một vị trí về phía trước. Khi một phần tử

được di chuyển một khoảng xa về trước thì cần rất nhiều thao tác duy chuyển. Shell sort tránh các trường hợp phải tráo đổi vị trí của hai phần tử xa nhau trong giải thuật sắp xếp chọn (nếu như phần tử nhỏ hơn ở vị trí bên phải khá xa so với phần tử lớn hơn bên trái).
Đầu tiên, giải thuật này sử dụng giải thuật sắp xếp chọn trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn. Khoảng cách này còn được gọi là **khoảng (interval)** – là số vị trí từ phần tử này tới phần tử khác

-Step-by-step descriptions:

```
ShellSort(array, n)
      gap = n / 2;
   while gap is greater than 0 do
            for i = gap; i < n; i = i + 1 do
                  temp = element at position i;
                  Locate j;
                  for j=i; j>=gap and array[j] > array[j-gap]; j=j-gap do
                        element at j = element at (j - gap);
                  end for
                  element at j = temp
            end for
            gap = gap / 2
       end while
end shellsort
```

-Complexity evaluations:

Best	Average	Worst	Memory
O(nlogn)	O(nlogn)	$O(n^2)$	O(1)

Heap sort

-Ideas: Ý tưởng của heap sort dựa trên selection sort là chọn phần tử cực trị rồi đặt về đúng vị trí của nó qua nhiều lần lặp, nhưng heap sort

xây dựng dữ liệu thành Binary Heap (max-heap và min-heap) rồi mới sắp xếp.

-Step-by-step descriptions:

```
Heapify(a,n,i)
        max <- i
        leftchild <- 2*i + 1
        rightchild <- 2*i + 2
        If (leftchild <= n && a[i] < a[leftchild])
                max = leftchild
        else
                max = i
        If (rightchild <= n && a[max] > a[rightchild])
                max = rightchild
        if (max != i)
                swap(a[i],a[max])
                Heapify(a,n,max)
Heapsort(a,n)
        For i = n/2 downto 1
                Heapify(a,n,i)
        For i = n downto 2
                Swap(a[1],a[i])
                Heapify(a,i,0)
```

-Complexity evaluations:

Best	Average	Worst	Memory

O(nlogn)	O(nlogn)	O(nlogn)	O(1)

Merge Sort

-Ideas :Merge sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Hàm merge() được sử dụng để gộp hai nửa mảng. Hàm merge(arr, l, m, r) là tiến trình quan trọng nhất sẽ gộp hai nửa mảng thành 1 mảng sắp xếp, các nửa mảng là arr[l…m] và arr[m+1…r] sau khi gộp sẽ thành một mảng duy nhất đã sắp xếp.

-Step-by-step descriptions:

```
function merge(arr, l, m, r)
{
   var n1 = m - l + 1;
   var n2 = r - m;

   // Create temp arrays
   var L = new Array(n1);
   var R = new Array(n2);

   // Copy data to temp arrays L[] and R[]
   for (var i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (var j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   // Merge the temp arrays back into arr[l..r]

   // Initial index of first subarray
   var i = 0;

   // Initial index of second subarray
```

```
    var j = 0;

    // Initial index of merged subarray
    var k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of
    // L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of
    // R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
// l is for left index and r is
// right index of the sub-array
// of arr to be sorted */
function mergeSort(arr,l, r){
   if(l>=r){
      return;//returns recursively
   }
   var m =l+ parseInt((r-l)/2);
   mergeSort(arr,l,m);
   mergeSort(arr,m+1,r);
   merge(arr,l,m,r);
}
```

-Complexity evaluations:

Best	Average	Worst	Memory
O(nlogn)	O(nlogn)	O(nlogn)	O(n)

Quick Sort

-Ideas :Quick sort là thuật toán sắp xếp, hoạt động theo cách sau: Chọn một phần tử trong mảng làm điểm đánh dấu và sau đó chia mảng thành hai mảng con bằng cách so sánh các phần tử trong mảng với điểm đánh dấu. Mảng 1 sẽ chứ các phần tử nhỏ hơn hoặc bằng điểm đánh dấu và mảng 2 sẽ gồm các phần tử lớn hơn điểm đánh dấu.

-Step-by-step descriptions:

```
/* low  --> Starting index,  high  --> Ending index */
      quickSort(arr[], low, high)
      {
         if (low < high)
         {
            /* pi is partitioning index, arr[p] is now
               at mid place */
            pi = partition(arr, low, high);
```

```
    quickSort(arr, low, pi - 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
  }}
```

-Complexity evaluations:

Best	Average	Worst	Memory
O(nlogn)	O(nlogn)	$O(n^2)$	O(log(n))

Counting Sort

-Ideas :Counting sort là một thuật toán sắp xếp cực nhanh một mảng các phần tử mà mỗi phần tử là các số nguyên không âm; Hoặc là một danh sách các ký tự được ánh xạ về dạng số để sort theo bảng chữ cái. Counting sort là một thuật toán sắp xếp các con số nguyên không âm, không dựa vào so sánh. Trong khi các thuật toán sắp xếp tối ưu sử dụng so sánh có độ phức tạp O(nlogn) thì Counting sort chỉ cần O(n) nếu độ dài của danh sách không quá nhỏ so với phần tử có giá trị lớn nhất.

-Step-by-step descriptions:

```
unction countSort(arr)
  {
  var max = Math.max.apply(Math, arr);
  var min = Math.min.apply(Math, arr);

  var range = max - min + 1;
  var count = Array.from({length: range}, (_, i) => 0);
  var output = Array.from({length: arr.length}, (_, i) => 0);
  for (i = 0; i < arr.length; i++) {
    count[arr[i] - min]++;
  }

  for (i = 1; i < count.length; i++) {
    count[i] += count[i - 1];
  }
```

```
for (i = arr.length - 1; i >= 0; i--) {
   output[count[arr[i] - min] - 1] = arr[i];
   count[arr[i] - min]--;
}

for (i = 0; i < arr.length; i++) {
   arr[i] = output[i];
}}
```

-Complexity evaluations:

Best	Average	Worst	Memory
O(n)	O(n)	O(n)	O(n)

Radix sort

- **-Ideas:** Ý tưởng: Để thực hiện sắp xếp, radix sort phân loại các phần tử theo lần lượt từng chữ số: hàng đơn vị, hàng chục, hàng trăm, hàng nghìn, …
- Giả sử, chúng ta có một mảng gồm các số như sau:
  `[43, 613, 831, 987, 17, 210, 1990, 1234]`

- Đầu tiên, mảng sẽ được chia thành các nhóm dựa vào giá trị của chữ số hàng đơn vị. chúng ta sẽ chia được các nhóm sau:
  ```
  [210, 1990] // nhóm 0
  [831] // nhóm 1
  [43, 613] // nhóm 3
  [1234] // nhóm 4
  [987, 17] // nhóm 7
  ```

- Sau khi chia nhóm theo hàng đơn vị, thứ tự các phần tử trong mảng sẽ như sau:
  `[210, 1990, 831, 43, 613, 1234, 987, 17]`

- Tiếp theo, mảng sẽ được chia thành các nhóm dựa vào hàng chục. Kết quả chúng ta được các nhóm:
  ```
  [210, 613, 17] // nhóm 1
  [831, 1234] // nhóm 3
  [43] // nhóm 4
  ```

```
[987] // nhóm 8
[1990] // nhóm 9
```

- Sau khi chia nhóm theo hàng chục, thứ tự các phần tử trong mảng sẽ như sau:
```
[210, 613, 17, 831, 1234, 43, 987, 1990]
```

- Thuật toán tiếp tục thực hiện với chữ số hàng trăm, chúng ta được các nhóm sau:
```
[17, 43] // nhóm 0
[210, 1234] // nhóm 2
[613] // nhóm 6
[831] // nhóm 8
[987, 1990] // nhóm 9
```

- Sau khi chia nhóm theo hàng trăm, thứ tự các phần tử trong mảng sẽ như sau:
```
[17, 43, 210, 1234, 613, 831, 987, 1990]
```

- Tiếp tục thực hiện với chữ số hàng nghìn, chúng ta được các nhóm sau:
```
[17, 43, 210, 613, 831, 987] // nhóm 0
[1234, 1990] // nhóm 1
```

- Sau khi chia nhóm theo hàng nghìn, thứ tự các phần tử trong mảng sẽ như sau:
```
[17, 43, 210, 613, 831, 987, 1234, 1990]
```

- Đến đây, toàn bộ các số trong mảng đều không có giá trị hàng chục nghìn, nên thuật toán dừng lại, chúng ta có mảng cuối cùng như kết quả bên trên.

-Step-by-step descriptions:

```
function getMax (arr, n):
        mx = arr[0]
        for i :=1 to E do:
          if arr[i] > mx : mx = arr[i]
        return mx
```

```
function countSort( arr,  n,  exp):
        output = [n]
          i, count = [0]*10
          for i := 0 to  n do:
                count[(arr[i] / exp) % 10]++

        for i: = 1 to 10 do :
                count[i] += count[i - 1]

        for i: = n - 1 to  0 do :{
                output [count[(arr[i] / exp) % 10] - 1]= arr[i]
                count[(arr[i] / exp) % 10]--
        }
        for i = 0 to  n do
                arr[i] = output[i]
function radixSort( arr,  n):
          int m = getMax(arr, n)
        for exp: = 1 in range (m / exp > 0) step *10 do:
                countSort(arr, n, exp)
```

-Complexity evaluations:

Best	Average	Worst	Memory
O(nlogk)	O(nlogk)	O(nlogk)	O(logk)

Flash sort

-Ideas: Ý tưởng: là một thuật toán sắp xếp tại chỗ (in-situ, không dùng mảng phụ) có độ phức tạp O(n), 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.

- Với bước **Phân lớp dữ liệu** ta có thể chia nhỏ thêm các bước như sau (với a[] là mảng cần được sắp xếp có n phần tử):
- Bước 1: Tìm giá trị nhỏ nhất của các phần tử trong mảng(minVal) và vị trí phần tử lớn nhất của các phần tử trong mảng(max).
- Bước 2: Khởi tạo 1 vector L có m phần tử (ứng với m lớp, trong source code lần này chọn số lớp bằng 0.45n).
- Bước 3: 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 4: 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). Sở dĩ như vậy bởi ta có thể hình dung như sau:
- Hoán vị toàn cục:
- Trước hết chúng ta đổi chỗ a[max] và a[0] bởi chúng ta khởi tạo giá trị k = m - 1 tương ứng với phân lớp mà a[max] thuộc về, do bắt đầu tại phần tử này nên biến j ban đầu cũng được khởi tạo bằng 0.
- Với tối đa n - 1 lần swap, n phần tử trong mảng sẽ về đúng phân lớp của mình.
- Khi j > L[k] - 1 nghĩa là khi đó phần tử a[j] đã nằm đúng vị trí phân lớp của nó do đó ta bỏ qua và tiếp tục tăng j lên để xét các phần tử tiếp theo.
- Như đã nói ở trên cứ mỗi khi đưa 1 phần tử về đúng phân lớp của nó ta lại giảm vị trí cuối cùng của phân lớp đó xuống (đồng thời tăng biến đếm số lần swap lên 1 đơn vị), quá trình này được thực hiện cho tới khi L[k] = j, điều này cũng tương đương với việc phân lớp k đã đầy.
- Cuối cùng, sau bước **hoán vị toàn cục**, mảng của chúng ta hiện tại sẽ được chia thành các lớp(thứ tự các phần tử trong lớp vẫn chưa đúng) do đó để đạt được trạng thái đúng thứ tự thì khoảng cách phải di chuyển của các phần tử là không lớn vì vậy **Insertion Sort** sẽ là thuật toán thích hợp nhất để sắp xếp lại mảng có trạng thái như vậy.

-Step-by-step descriptions:

```
__L = [];
function flash_sort(a, n) :
        if ( n <= 1) return
          m = n * 0.43
```

```
if (m <= 2) m = 2
for  i: = 0 to m do
        __L[i] = 0
  Mx = a[0], Mn = a[0];
for  i: = 1 to < n do:{
        if  Mx < a[i] : Mx = a[i];
        if  Mn > a[i] : Mn = a[i];
}
if  Mx == Mn:
        return
for  i: = 0 to n do:
        ++__L[getK(a[i])]
for i: = 1 to m do
        __L[i] += __L[i - 1]
count = 0
  i = 0
while ( count < n):
         k = getK(a[i])
        while  i >= __L[k]:
                k = getK(a[++i])
         z = a[i]
        while i is not  __L[k] :
                k = getK(z)
                int y = a[__L[k] - 1]
                a[--__L[k]] = z
                z = y
                ++count
for  k = 1 to m do:
        for  i = __L[k] - 2 to L[k - 1] step --i do:
                if ( a[i] > a[i + 1]):
                        int t = a[i], j = i
                        while ( t > a[j + 1]):
                                a[j] = a[j + 1]
```

++j

a[j] = t

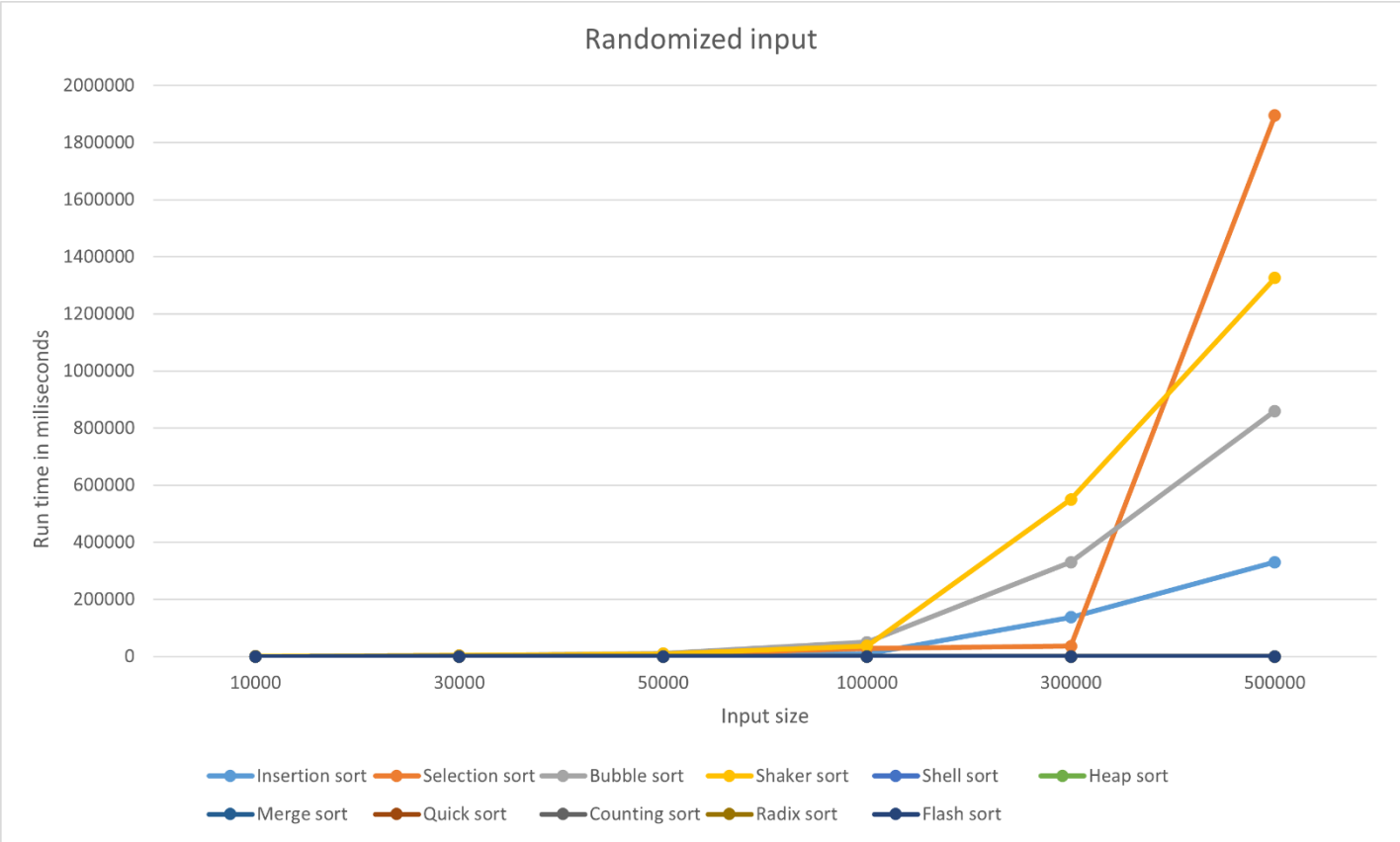
-Complexity evaluations:

Best	Average	Worst	Memory
O(n + m)	O(n+m)	$O(n^2)$	O(m)

Note: m = 0.43n

4. **Experimental results and comments**:

Data order: Randomize input						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	95	50186031	819	452076619	2343	1249976861
Selection sort	172	100019998	2458	900059998	7003	2500099998
Bubble sort	337	100010001	3156	900030001	10523	2500050001
Shaker sort	342	100005001	3152	900015001	9174	2500025001
Shell sort	2	376339	7	1341315	22	4685500
Heap sort	4	637843	10	2150074	12	3771607
Merge sort	4	581368	12	1930309	23	3371821
Quick sort	0	297191	0	928969	4	1630929
Counting sort	0	50589	0	150008	1	232791
Radix sort	2	140056	7	510070	13	850070
Flash sort	0	91552	3	264936	7	442306
Data size	100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	9411	5018893167	137227	44988612131	329986	125094147413
Selection sort	27987	10000199998	35766	90000599996	1895327	250000999999
Bubble sort	49950	10000100001	330018	90000300001	859427	250000500001
Shaker sort	36695	10000050001	549947	90000150001	1325667	250000250001
Shell sort	23	5853210	89	20218291	137	35552844
Heap sort	33	8043249	97	26487055	175	45969312
Merge sort	49	7143229	122	23311685	218	40264306
Quick sort	15	3398183	47	10847024	78	19165685
Counting sort	2	432787	5	1232781	10	2032782
Radix sort	31	1700070	104	5100070	140	8500070
Flash sort	13	854327	49	2827018	139	4408281

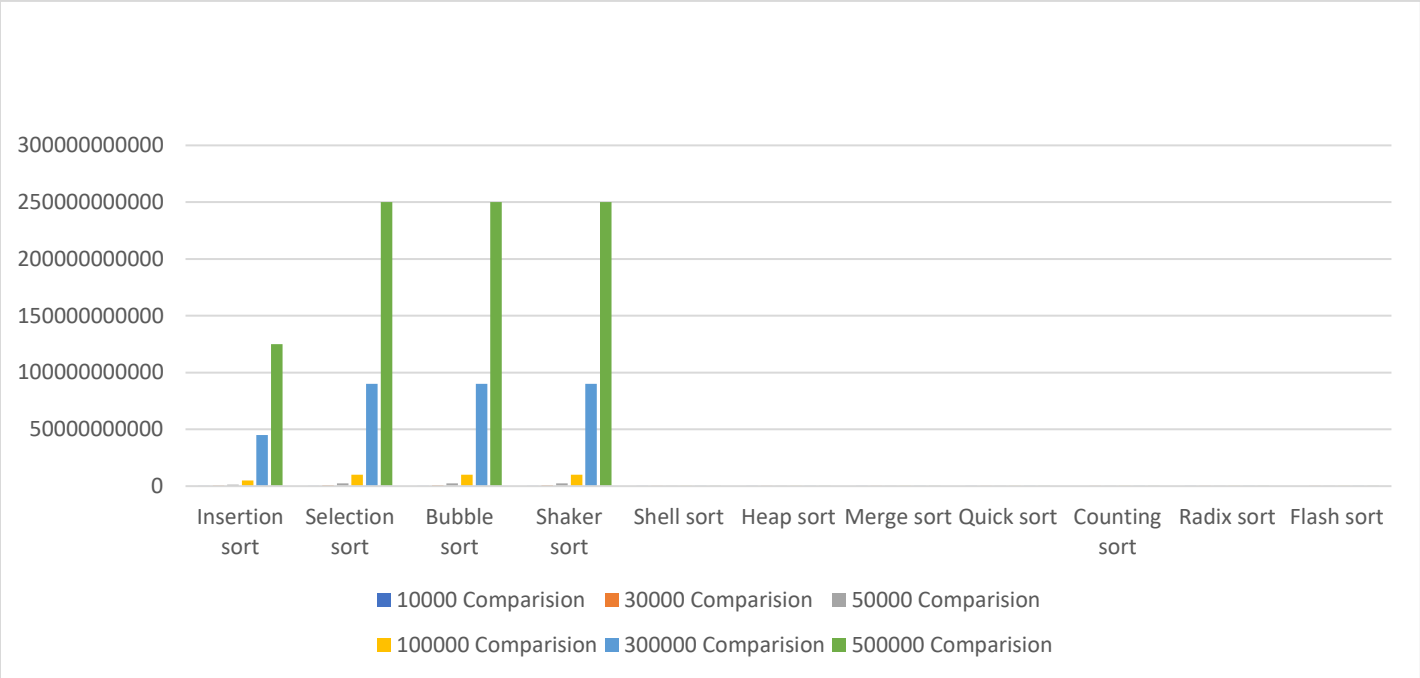


-Randomized input:

+ Nhóm các giải thuật tốn nhiều thời gian chạy là: Insertion sort, Selection sort, Bubble sort, Shaker sort. Ta có thể thấy được đường đồ thị các thuật toán này khá giống với đồ thị hàm $y = x^2$. Thật vậy, vì độ phức tạp trong trường hợp xấu nhất của chúng là O(n^2). Tuy nhiên, với Selection sort, với số phần tử là 300000 trở xuống thì thời gian chạy lại nhanh hơn hẳn so với 3 thuật toán kia, nhưng đến số phần tử 500000 thị chậm một cách đột biến, chậm nhất trong các giải thuật sắp xếp.

+ Nhóm các giải thuật với thời gian chạy khá nhanh là: Shell sort, Heap sort, Merge sort, Quick sort. Ta có thể thấy được đường đồ thị các thuật toán này khá giống với đồ thị hàm y = x*log(x). Thật

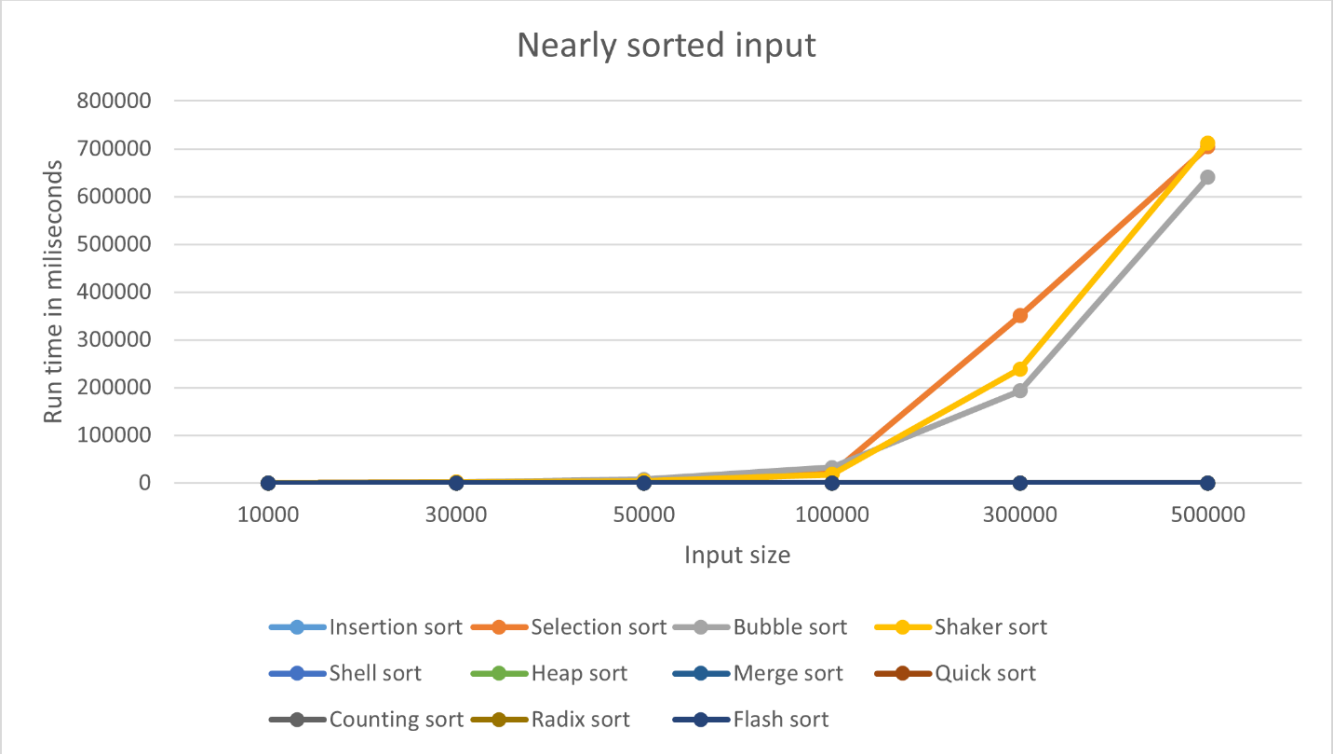
vậy, vì độ phức tạp trong trường hợp xấu nhất của chúng là O(nlogn), ngoại trừ Quick Sort là O(n^2), tuy nhiên thực tế cho thấy Quick Sort lại chạy nhanh hơn 3 thuật toán kia.
+ Nhóm các giải thuật sắp xếp đặc biệt như Counting sort, Radix sort, Flash sort chạy rất nhanh, ít tốn thời gian hơn hẳn 2 nhóm trên, tuy nhiên vì bản chất chúng sẽ áp dụng trong phạm vi hạn chế (chỉ các số nguyên, …). Độ phức tạp là tuyến tính, đường đi đồ thị thời gian chạy tuyến tính. Counting Sort có thời gian chạy nhanh nhất trong 11 thuật toán trên.



Thuật toán có số lượng so sánh lớn là Insertion , Selection, Bubble, Shaker (Thấp nhất là insertion)

Thuật toán có số lượng so sánh nhỏ là: Shell, Heap, Merge, Quick, Counting, Radix, Flash, (Thấp nhất là counting sort)

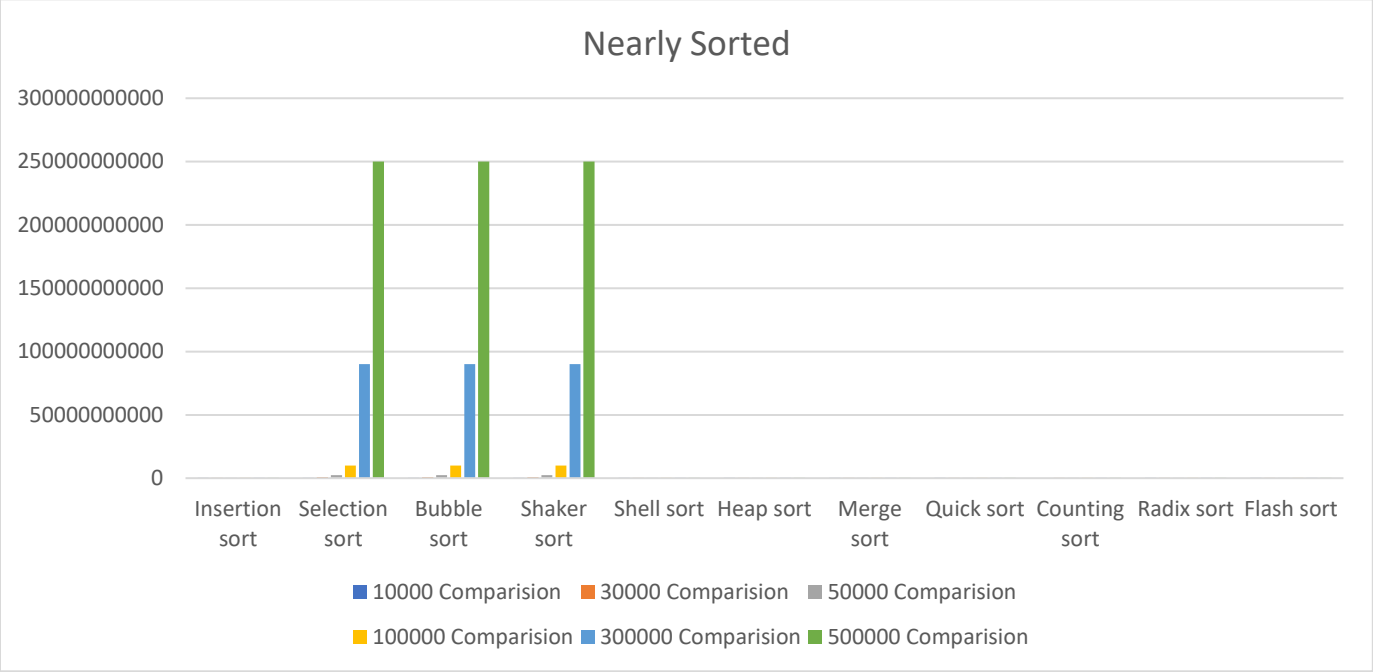
Data order: Nearly sorted input						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	0.18	163895	0.23	431327	0.48	457823
Selection sort	162	100019998	1850	900059998	5136	2500099998
Bubble sort	171	100010001	1463	900030001	7430	2500050001
Shaker sort	168	100005001	1566	900015001	4379	2500025001
Shell sort	0	264537	3	848479	5	1474585
Heap sort	4	670061	5	2236515	20	3925112
Merge sort	3	544963	9	1778420	16	3102809
Quick sort	0	160899	0	518360	2	946665
Counting sort	0	51454	0.7	152145	1	268086
Radix sort	2	140056	8	510070	16	850070
Flash sort	1	112901	3	338701	4	564501
Data size	100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	1	613591	3	1098975	16	1581875
Selection sort	20634	10001999998	350323	90005999998	704211	250009999998
Bubble sort	32491	10000100001	192749	90000300001	640183	250000500001
Shaker sort	18108	10000050001	238344	90000150001	712338	250000250001
Shell sort	7	3082849	23	10284025	38	17076077
Heap sort	25	8364873	80	27413176	145	47404743
Merge sort	41	6510381	84	21060327	142	36249974
Quick sort	12	1993262	15	6227196	21	10572916
Counting sort	2	570429	6	1768063	10	2973766
Radix sort	40	170070	96	6000084	123	10000084
Flash sort	10	1129001	47	3387001	71	5645001



-Nearly sorted input: với dữ liệu gần như đã được sắp xếp, có sự thay đổi lớn về thời gian chạy cũng như tương quan giữa các thuật toán sắp xếp:

+ Các giải thuật tốn nhiều thời gian chạy như Selection sort, Bubble sort, Shaker sort thời gian chạy giảm xuống rõ rệt, từ 1,2 đến 2,0 lần so với trường hợp xấu nhất.
+ Đặc biệt, Insert Sort thời gian chạy giảm sâp sàn, lúc này là nhanh nhất trong các giải thuật (trừ Counting sort). Trong trường hợp đặc biệt này nên dùng Insert Sort là có hiệu quả tốt nhất.
+ Merge sort, Heap sort, Radix sort thời gian chạy xấp xỉ như nhau, mặc dù trong trường hợp xấu nhất, Radix sort chạy nhanh hơn hơn 2 cái kia.
+ Shell sort thời gian chạy nhanh hơn Heap sort, Merge sort. Quick Sort vẫn chạy nhanh chỉ sau Insertion Sort và Counting Sort. Flash Sort chạy chậm hơn 3 cái kia, nhưng chung quy vẫn rất nhanh trong 11 thuật toán.

+ Chậm nhất trong các giải thuật sắp xếp là Shaker sort. Nhanh nhất vẫn là Counting Sort - giải thuật đặc biệt.
+ Phần lớn các thuật toán đều chạy nhanh hơn so với khi dữ liệu ngẫu nhiên, ngoại trừ Radix sort.

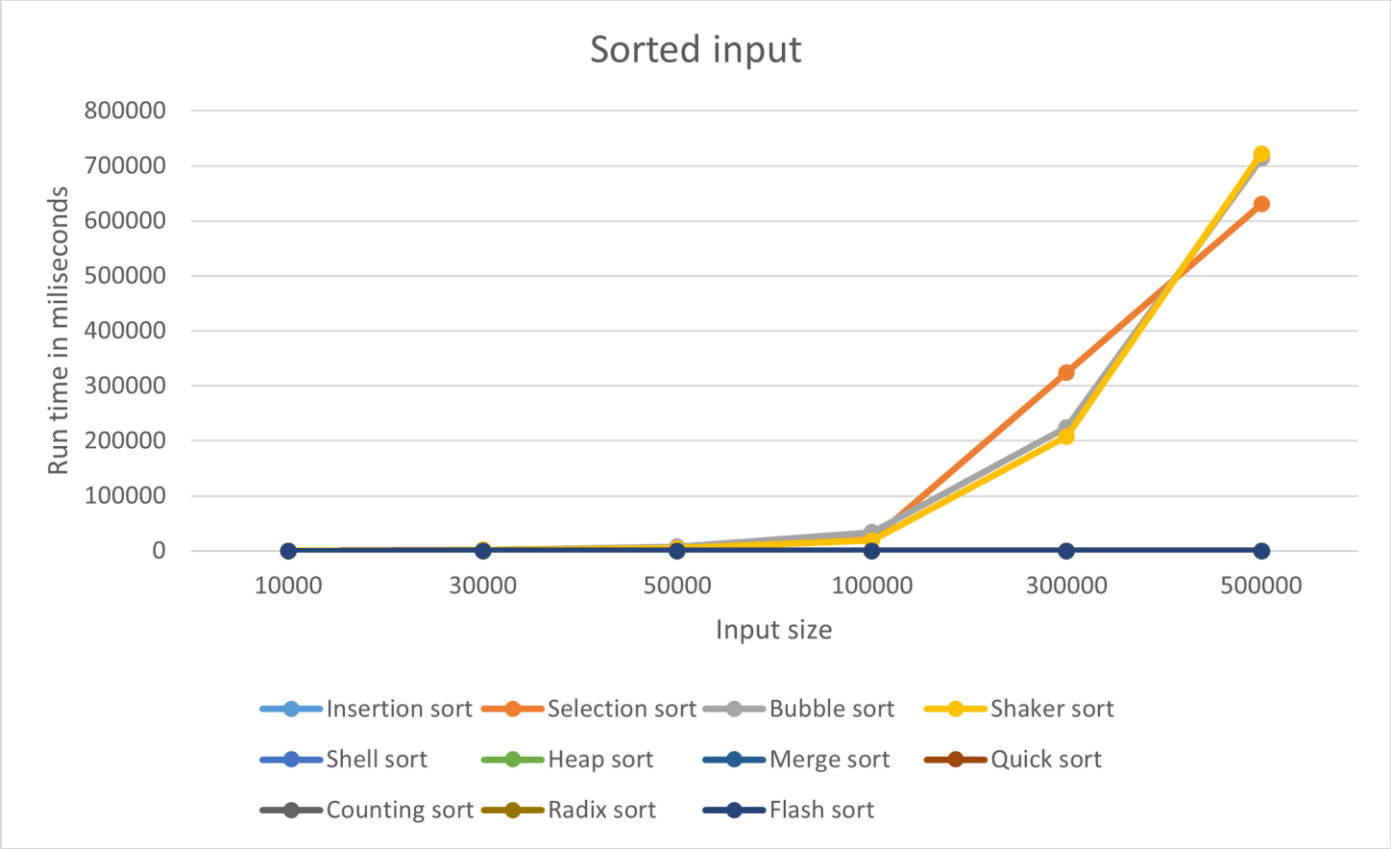


Thuật toán có số lượng so sánh lớn là, Selection, Bubble, Shaker

Thuật toán có số lượng so sánh nhỏ là: Insertion, Shell, Heap, Merge, Quick, Counting, Radix, Flash, (Thấp nhất là inserttion sort)

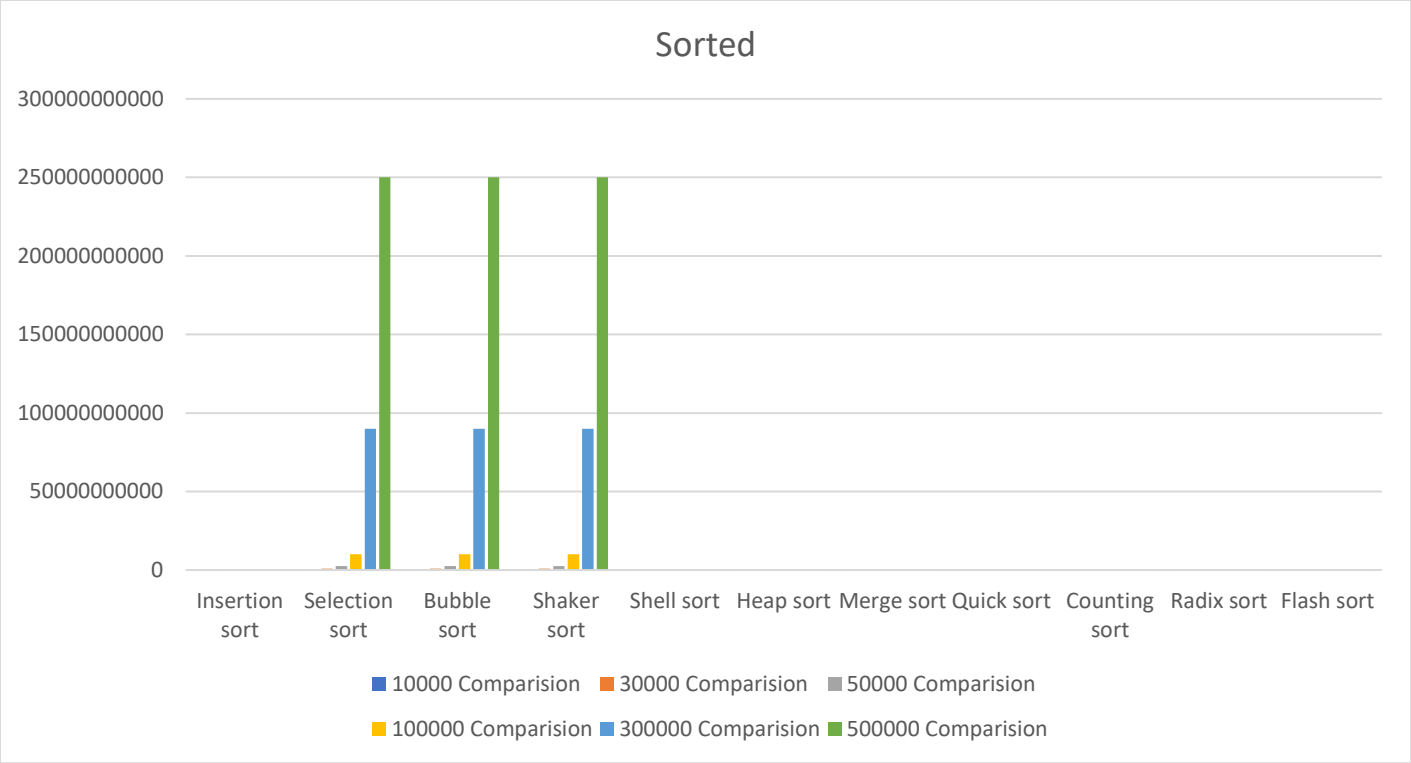
Data order: Sorted input						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	0	19999	0	59999	0	99999
Selection sort	184	100019998	1839	900059998	5120	2500099998
Bubble sort	157	100010001	1485	900030001	7484	2500050001
Shaker sort	169	100005001	1523	900015001	4543	2500025001
Shell sort	1	240037	3	780043	3	1400043
Heap sort	5	670333	6	2236652	10	3925355
Merge sort	3	529911	16	1749481	20	3055406
Quick sort	0	160863	0	518312	2	946617
Counting sort	0	59997	0	179997	1	299997

Radix sort	4	140056	11	510070	16	850070
Flash sort	1	97854	3	293554	6	489254
Data size	100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	1	199999	1	599999	3	999999
Selection sort	20565	10000199998	324353	90005999998	630252	250009999998
Bubble sort	33740	10000100001	224535	90000300001	713644	250000500001
Shaker sort	18628	10000050001	207761	90000150001	722384	250000250001
Shell sort	6	3000045	20	10200053	35	17000051
Heap sort	25	8365084	81	27413234	153	47404890
Merge sort	56	6460754	85	21024472	131	36210425
Quick sort	6	1993226	14	6227156	15	10572876
Counting sort	2	599997	6	1799997	10	2999997
Radix sort	31	1700070	77	6000084	233	10000084
Flash sort	11	978504	30	2935504	36	4892504



- Sorted Input: với dữ liệu gần như đã được sắp xếp, cơ bản không sự thay đổi nhiều về thời gian chạy cũng như tương quan giữa các thuật toán sắp xếp so với trường hợp dữ liệu gần như đã được sắp xếp:

+ Các giải thuật tốn nhiều thời gian chạy như Selection sort, Bubble sort, Shaker sort thời gian chạy khá tương tự so với trường hợp dữ liệu gần như đã được sắp xếp.
+ Merge sort, Heap sort, Radix sort thời gian chạy khá ổn và xấp xỉ như nhau. Ta thấy được với dữ liệu gần như đã được sắp xếp, Radix sort lại chạy chậm hơn trường hợp dữ liệu ngẫu nhiên. Radix sort lần này lại chạy chậm hơn Merge sort, Heap sort.
+ Insert Sort thời gian chạy là nhanh nhất trong 11 thuật toán vì bản chất của nó là rất ít phép so sánh và hoán đổi trong khi dữ liệu đã sắp xếp.
+ Chậm nhất trong các giải thuật sắp xếp vẫn là Shaker sort.
+ Shell sort thời gian chạy nhanh hơn Heap sort, Merge sort. Quick Sort vẫn chạy nhanh chỉ sau Insertion Sort và Counting Sort. Thời gian chạy vẫn ổn định, Flash sort chạy nhanh hơn so với khi trường hợp dữ liệu gần như đã được sắp xếp.
+ Phần lớn các thuật toán đều chạy nhanh hơn so với khi dữ liệu ngẫu nhiên, ngoại trừ Radix sort.

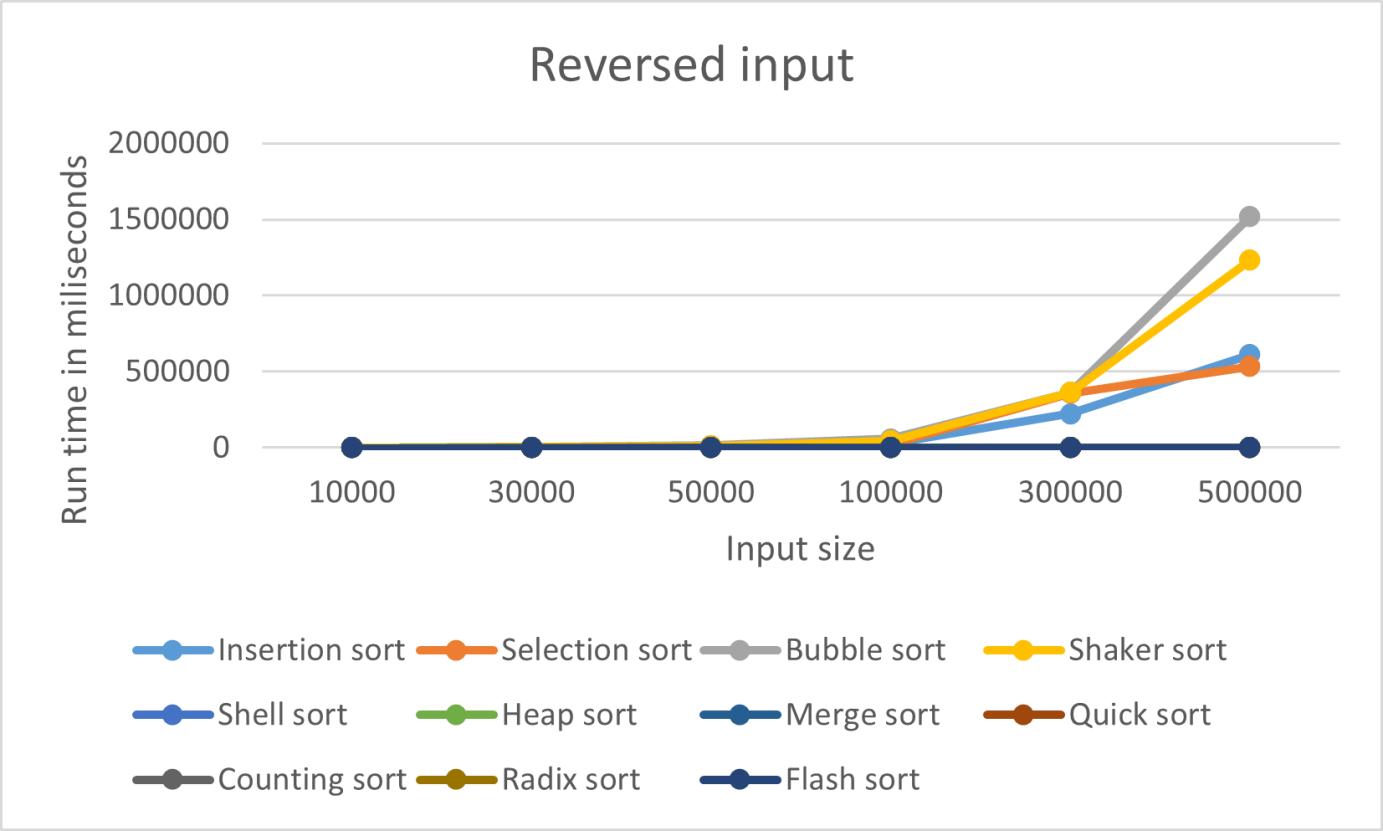


Thuật toán có số lượng so sánh lớn là, Selection, Bubble, Shaker

Thuật toán có số lượng so sánh nhỏ là: Insertion, Shell, Heap, Merge, Quick, Counting, Radix, Flash, (Thấp nhất là inserttion sort)

Data order: Reverse input						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	190	100009999	1645	900029999	5001	2500049999
Selection sort	161	100019998	1858	900059998	5140	2500099998
Bubble sort	282	100010001	3067	900030001	13292	2500050001
Shaker sort	403	100005001	3682	900015001	10982	2500025001
Shell sort	1	262147	2	854525	4	1482179
Heap sort	3	606775	5	2063328	5	3612728
Merge sort	2	535493	8	1771241	24	3085946
Quick sort	0	170880	0	548322	5	996628
Counting sort	0	59997	0	179997	1	299997
Radix sort	2	140056	7	510070	12	850070
Flash sort	1	112877	2	338678	4	564477
Data size	100000		300000		500000	

Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Insertion sort	27592	1000009999 9	223246	9000029999 9	611902	2500004999 99
Selection sort	20681	1000019999 8	358342	9000599999 8	533423	2500099999 98
Bubble sort	59338	1000010000 1	363467	9000030000 1	1517412	2500005000 01
Shaker sort	45332	1000005000 1	364646	9000015000 1	1231231	2500002500 01
Shell sort	5	3844605	28	12700933	35	21428803
Heap sort	29	7718947	83	25569383	135	44483352
Merge sort	32	6521852	102	21205634	169	36619681
Quick sort	12	2093238	15	6527178	34	11072890
Counting sort	2	599997	6	1799997	10	2999977
Radix sort	19	1700070	74	6000084	98	10000084
Flash sort	11	1128978	28	3386975	56	5644975



- Reverse input:

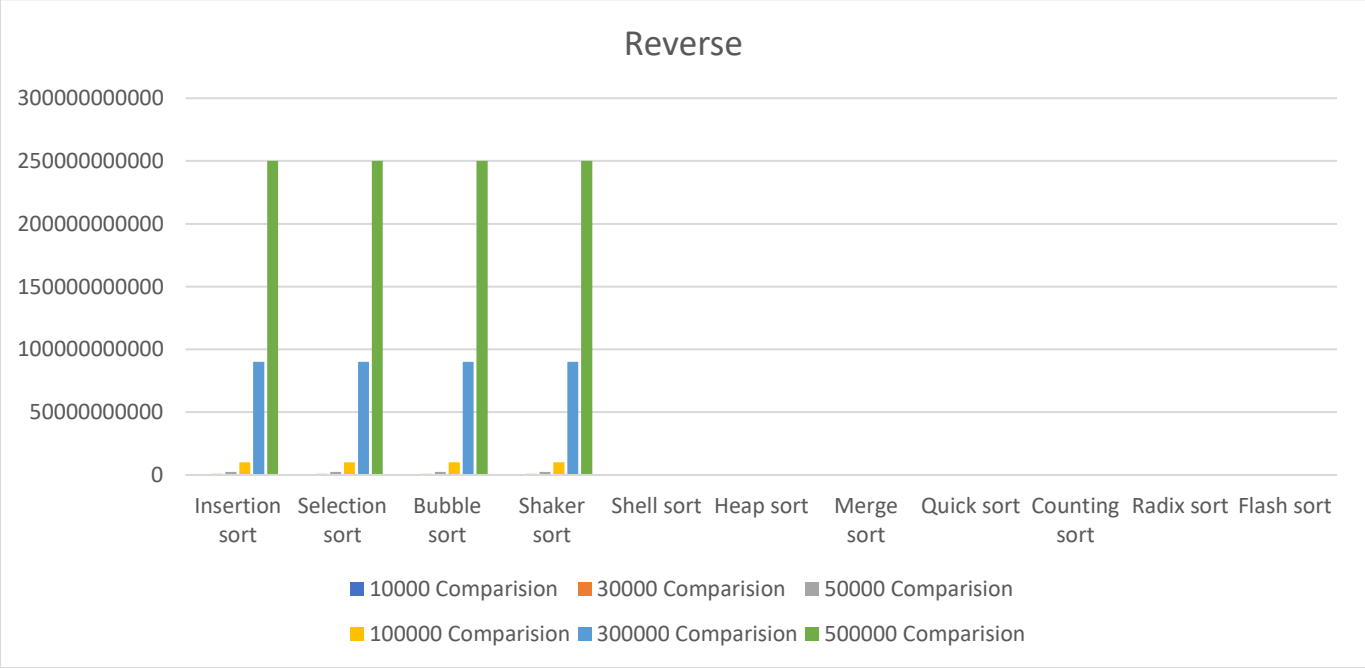
  + Nhóm các giải thuật tốn nhiều thời gian chạy là: Insertion sort, Selection sort, Bubble sort, Shaker sort. Ta có thể thấy được đường đồ thị các thuật toán này khá giống với đồ thị hàm y =

x^2, vì độ phức tạp trong trường hợp xấu nhất của chúng là O(n^2). Tuy nhiên tương quan có sự thay đổi, Insertion sort và Bubble sort tốn nhiều thời gian chạy hơn so với trường hợp dữ liệu ngẫu nhiên, Bubble sort trở thành giải thuật chậm nhất. Selection sort lại cải thiện hơn khi thời gian chạy giảm tới 3 lần so với trường hợp dữ liệu ngẫu nhiên, nhưng chung quy vẫn là rất chậm trong 11 thuật toán.

+ Merge sort, Heap sort vẫn giữ ổn định trong tất cả các trường hợp dữ liệu.

+ Radix sort, Flash sort chạy với thời gian nhanh. Counting sort chạy nhanh nhất trong 11 thuật toán. Nhưng 3 giải thuật này chỉ áp dụng khá hạn chế vì cách sắp xếp khá đặc biệt.

+ Shell sort thời gian chạy nhanh hơn Heap sort, Merge sort. Quick Sort vẫn chạy nhanh chỉ sau Counting Sort và áp dụng rộng rãi đúng như tên gọi của nó.



Thuật toán có số lượng so sánh lớn là Insertion , Selection, Bubble, Shaker (Thấp nhất là insertion)

Thuật toán có số lượng so sánh nhỏ là: Shell, Heap, Merge, Quick, Counting, Radix, Flash, (Thấp nhất là counting sort)

Tổng kết: (4 biểu đồ dạng đường)

- Counting Sort chạy nhanh nhất với 3 trên 4 trường hợp Randomized input, Nearly sorted input, Reverse input. Với sorted Input thì Insertion sort chạy nhanh nhất
- Nhóm các giải thuật chậm nhất phổ biến bao gồm: Selection sort, Bubble sort, Shaker sort.
- Các giải thuật chạy khá nhanh, khá ổn định và áp dụng rộng rãi: Merge sort, Heap sort, Shell sort, Quick Sort. Riêng Insertion sort nên áp dụng cho trường hợp Nearly sorted input, sorted Input.
- Counting sort, Radix sort, Flash sort chạy nhanh .Nhưng 3 giải thuật này chỉ áp dụng khá hạn chế vì cách sắp xếp khá đặc biệt.
- Các giải thuật stable: Bubble sort, Counting sort, Insertion sort, Merge sort, Quick Sort
- Các giải thuật unstable: Heap sort, Shaker sort, Selection sort, Radix sort, Flash sort, Shell sort

Tổng kết: (4 biểu đồ dạng cột)

- Với 4 kiểu dữ liệu đầu vào thì insertion, select, buble, shake đều có số lượng phép so sánh lớn
- Trừ kiểu sake có sô lượng so sánh nhỏ ở kiểu dữ liệu ở dạng sắp sẵn
- Các kiểu sort còn lại đều có số lượng phép so sánh như nhau với mọi kiểu input
- Ngoại trừ quick sort với kiểu input nearly sorted, sorted, reverse thì số lượng phép so sánh cao

5. **Project organization and Programming notes:**

- Source code gồm 1 file header.h chứa các thư viện và khai báo hàm của file DataGenerator.cpp, file DataGenerator.cpp chứa các hàm để khởi tạo mảng theo 4 input order, file main.cpp gồm 11 thuật toán và các hàm giúp xử lý input output của command. Trong hàm main(), truyền vào 2 dữ liệu là int argc (số lượng của argv) và char** argv (prototype của các command). Chia 2 loại command là algorithm mode (3 command đầu) và compare mode (2 command cuối) dựa vào "-a" và "-c" . Phân biệt command 1 với 2,3 dựa vào argv[3] là file input và data size , phân biệt command 2 với 3 là dựa vào argv[4] là input order và output parameter. Phân biệt command 4 và 5 là dựa vào argv[4] là file input và data size.

- Sử dụng thư viện **Chrono** để đo thời gian chạy của thuật toán.Khai báo thời gian bắt đầu và thời gian kết thúc của thuật toán như sau

```
start = high_resolution_clock::now()

end = high_resolution_clock::now()
```

BIến thời gian chạy (đơn vị ms)

```
duration<double, std::milli>time;
```

6. **List of references**

-Các thuật toán sort:

https://www.geeksforgeeks.org/sorting-algorithms/

https://bom.so/lSN4H6

-Thư viện chrono:

https://cplusplus.com/reference/chrono/

-Command line arguments to execute:

https://bom.so/PhnJ3p