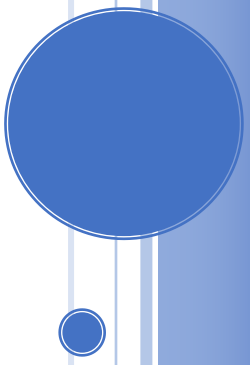
KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REPORT

LAB 2 - SORTING



UNIVERSITY OF SCIENCE

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

INFORMATION TECHNOLOGY

Lecturers:

- **Châu Thành Đức**
- **Phan Thị Phương Uyên**
- **Ngô Đình Hy**
- **Phan Thị Phương**

Student:

- **Nguyễn Đức Huy - 19127422**

Progress:

- **Finish all section in lab-2 : Sorting**

CONTENT

LAB 2:
SORTING:

1 Programming

1.1 Algorithms

I choose **Set 2.** There are 12 algorithms in Set 2: Selection Sort, Insertion Sort, Binary-Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort.

For each algorithms I must implement for ascending order, now I will explain all the information around each algorithms as well (ideas, algorithms, algorithms 's review).

Selection Sort:

Ideas: Selection sort is basically choose smallest for each index of the array. The idea of Selection Sort is that we repeatedly find the smallest element in the unsorted part of the array and swap it with the first element in the unsorted part of the array. For each index, Selection sort find the smallest at the correct index by finding smallest element of the rest element and then swap them. Selection sort loops over indices in the array, the algorithms first selects the smallest element in the array and place it at array position 0, then it selects the next smallest element in the array and place it at array position 1 and so on. It simply continues this proceduce until it places the biggest element in the last position of the array. Selection sort will not require no more than n-1 interchanges (if size of array is n).

Algorithms: Here is few step to explain how selection sort work:

Step 1: Set variable Min to location 0
Step 2: Find the minimum element in the list
Step 3: Swap with this value location Min
Step 4: Increment variable Min to point to next element
Step 5: Repeat until list is sorted

Time complexity: We denote with *n* data items. The two nested loops are an indication that we are dealing with a time complexity* of $O(n^2)$. This will be the case if both loops iterate to a value that increases linearly with *n*.

For each of the first loops we find the smallest element and swapped. Thus, we have, in sum, maximum of n-1 swapping operations.
For the total complexity, only the highest complexity class matters, therefore:
The average, best-case, and worst-case time complexity of Selection Sort is: $O(n^2)$ and a constant space complexity O(1).

Insertion Sort:

Ideas: The original idea behind Insertion Sort is often compared to the way people sort a hand of cards while playing Rummy.
In this card game, the dealer deals out cards to each player. Then, the players take the cards given to them one by one, sorting them in their hand in ascending order by inserting each card in its place. During this entire process, the players hold one sorted pile of cards in their hands, while the unsorted pile from which they draw new cards is in front of them. A very useful property of Insertion Sort is the fact that it doesn't need to know the entire array in advance in order for it to be sorted - it just inserts the given elements one by one. This really comes in handy when we want to add more elements in an already sorted array, because Insertion Sort will add the new elements in their proper places without resorting the entire collection.
The main idea of insertion sort is that array is divided in two parts which left part is already sorted, and right part is unsorted. Then at every iteration sorted part grows by one element which called key. During an iteration, if compared element on the left part is greater than key then you just shift it until right position.

Algorithms: Here is steps on how it works:

Step 1: If it is the first element, it is already sorted.
Step 2: Pick the next element.
Step 3: Compare with all the elements in sorted sub-list.
Step 4: Shift all the the elements in sorted sub-list that is greater than the value to be sorted.
Step 5: Insert the value.
Step 6: Repeat until list is sorted.

Time complexity: Unlike Selection sort has 2 nested loop, insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using for loops, but instead it uses one while loop, which avoids extra steps once the array gets sorted.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

- **Worst Case Time Complexity**: O(n^2)
- **Best Case Time Complexity**: O(n)
- **Average Time Complexity**: O(n^2)
- **Space Complexity**: O(1)

Insertion Sort is a simple, stable, in-place, comparison sorting algorithm.

Binary – Insertion sort:

Ideas: The main idea for binary insertion sort express by your name. Instead of Shifting and compare for each element on the left side that I mention in Insertion sort above, to do find the correct position, unlike original Insertion sort we use binary method for the left side which is already sorted to use binary search. And you just find correct position with for the key first on the other side. Then after you find correct position you update this value in this position like Insertion sort. The difference between Insertion sort and Binary – Insertion sort is finding correct position with difference method.

Algorithms: It similar to Insertion sort

Step 1: If it is the first element, it is already sorted.
Step 2: Pick the next element.
Step 3: Using binary search to find correct position.
Step 4: Shift all elements that are from pos to i - 1 to the right side by one position.
Step 5: Insert the value.
Step 6: Repeat until list is sorted.

Time complexity: Binary search is used to reduce the number of comparisons in Insertion sort. This modification is known as Binary Insertion Sort.
Binary Insertion Sort use binary search to find the proper location to insert the selected item at each iteration. In insertion sort, it takes O(i) (at ith iteration) in worst case. Using binary search, it is reduced to O(log i). Therefore Time complexity is:

- **Worst case time complexity:** Θ(N log N) comparisons and swaps
- **Average case time complexity:** Θ(N log N) comparisons and swaps
- **Best case time complexity:** Θ(N) comparisons and Θ(1) swaps

- **Space complexity:** Θ(1).

Bubble sort:

Ideas: Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Algorithms: Here is few step of bubble sort:

Step 1: Starting with the first element(index = 0), compare the current element with the next element of the array.

Step 2: If the current element is greater than the next element of the array, swap them.

Step 3: If the current element is less than the next element, move to the next element. Repeat step 1.

Time complexity: In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be: n(n-1)/2

Therefore the time complexity of Bubble Sort is $O(n^2)$.

The main convenient of Bubble sort is very simply to understand and it's not hard to implement

The space complexity for Bubble Sort is O(1), because only a single additional memory space is required for temp variable.

Also, the best case time complexity will be O(n), it is when the list is already sorted.

Here is the Time and Space complexity for the Bubble Sort algorithm.

- **Worst Case Time Complexity:** $O(n^2)$
- **Best Case Time Complexity:** O(n)
- **Average Time Complexity:** $O(n^2)$
- **Space Complexity:** O(1)

Shaker sort:

Ideas: Shaker Sort is an improvement of Bubble Sort.

After bringing the smallest part to the beginning of the array, the largest part is returned to the end of the range. By bringing elements to the right positions at both ends, Shaker Sort will help improve the sequence arrangement time by reducing the weight of the array being considered at the next comparison.

Algorithms: Shaker sort similar to bubble sort but each iteration, bubble sort it just find 1 correction position while Shaker sort find 2 element(1 in largest and 1 in smallest in each iteration).

Each iteration of the algorithm is broken up into 2 steps:

Step 1: The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if value on the left is greater than the value on the right, then values are swapped. At the end of first iteration, largest number will reside at the end of the array.

Step 2: The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

Time complexity: In the best case, 2 stages of each iteration don't use swap function and therefore here is complexity of Shaker sort, it similar to Bubble sort right.

- **Worst case time complexity:** $\Theta(n^2)$
- **Average case time complexity:** $\Theta(n^2)$
- **Best case time complexity:** $\Theta(n)$
- **Space complexity:** $\Theta(1)$

Time complexities are same, but Shaker sort performs better than Bubble Sort. Typically cocktail sort is less than two times faster than bubble sort.

As the shaker sort goes bidirectionally, the range of possible swaps, which is the range to be tested, will reduce per pass, thus reducing the overall running time slightly.

Shell sort:

Ideas: Shell Sort is a highly effective sorting algorithm based on the Insertion Sort algorithm. This avoids having to swap the positions of two distant elements in the sorting algorithm (if the smaller one is quite far to the right than the larger one on the left).

The idea of Shell Sort is to allow exchange of far items

First, it uses a sorting algorithm that selects elements that are far apart, then arranges elements with narrower distances. This distance is also known as interval – the number of positions from one to another.

Algorithms: In Shell Sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

Here is a few step to understand this algorithms:

Step 1: You start with a big gap

Step 2: Divide the list into smaller sublists corresponding to big gap (in my implementation I reduce by dividing of 2)

Step 3: Sort these sublists by using the Insertion Sort

Step 4: Repeat until the list has been sorted

Time complexity: Time complexity of above implementation of Shell sort is $O(n^2)$. In my implementation, gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity.

The best-case is when the array is already sorted. The would mean that the inner if statement will never be true, making the inner while loop a constant time operation. Using the bounds you've used for the other loops gives **O(nlogn).** The best case of O(n) is reached by using a constant number of increments. And the **space time complexity** is O(1).

Heap sort

Ideas: The idea behind a heap sort is to find the highest value and place it at the end, repeating the process until everything is sorted. In order to fully understand how a heap sort works, we first have to understand a binary heap, and subsequently, a binary tree.

Algorithms: The Heapsort algorithm is also known as the heap algorithm, which can be seen as an improvement of Selection Sort by dividing elements into two child arrays.

- array of sorted elements.
- array of unsorted arranged elements.

In unsorted array, the largest elements are separated and included in the sorted array. What's improved in Heapsort compared to Selection Sort is the use of heap data structures instead of linear-time searches such as Selection sort to find the largest elements.

Heapsort is an in-place algorithm, which means that there is no need to add any ancies to the data structure during the operation of the algorithm. However, this algorithm does not have stability.

Here is few steps to understand clearly this algorithms:

Step 1: Build a max heap from the input data.
Step 2: At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
Step 3: Repeat step 2 while size of heap is greater than 1.

Time complexity: Time complexity of heapify function is O(Logn). Time complexity of creating and building heap sort is O(n) and overall time complexity of Heap sort is O(nLogn) (best case, worst case and average case) and Space complexity is O(1).

Merge sort:

Ideas: Divide large arrays into smaller smaller arrays by dividing large arrays in half and continuing to split the child arrays until the smallest array has only 1 component.
Compare 2 child arrays with the same base array (when dividing large arrays into 2 child arrays, that large array is called the base array of those two child arrays).
When comparing them, they both arrange and stitch the two child arrays into the base array, continue to compare and stitch the child arrays together until the only array remains, which is the array that has been sorted.

Algorithms: Here is few steps of Merge sort
Step 1: If there is only one part of the list, this list is considered sorted. Returns a list or value.
Step 2: Divide the list in two halves by recursion until it can't be divided.
Step 3: Combine smaller (sorted) lists into new (also organized) lists.

Time complexity: The time complexity in the worst case, the best, the average merge sort is O(nlogn), which makes Merge Sort quite effective. Merge sort is an option when an algorithm is needed to arrange stability, unlike quick sort, which is very unstable.
The downside of merge sort can be mentioned as not very effective in terms of space, when the space complexity in the worst case is O(n), while the quick sort is O(1)

Quick sort:

Ideas: Like merge sort, the quick sort algorithm is a divide and conquer algorithm. It selects an elements in the array as a marker(pivot). The algorithm will perform array divisions into

child arrays based on the selected pivot. The choice of pivot greatly affects the speed of sorting. But the computer can't know when to choose. Here are some ways to choose which pivots are commonly used:

- Always select the first part of the array.
- Always select the last part of the array. (Used in this article)
- Select a random.
- Select an element with a value in the middle of the array (median element).

Algorithms: Here is few steps to understand Partition function of quick sort

Step 1: Select the last elements as the highest indexed elements (the one at the bottom of the list)

Step 2: Declare two variables to point to the left and right side of the list, except for the last element

Step 3: The left variable points to the left array of children

Step 4: The right variable points to the right array of children

Step 5: When the value at the left variable is less than the latching factor, move to the right

Step 6: When the value at the right variable is greater than the latching factor, move left

Step 7: If not in the case of both step 5 and step 6, swap the left and right variable values

Step 8: If left ≥ right, this is the new pivot value

Here is a few steps of quick sort:

Step 1: Take the key pivot as the factor at the bottom of the list

Step 2: Divide the array by using the key pivot

Step 3: Use quick sorting recursively with the left array

Step 4: Use a quick recursive arrangement with the right child array

Time complexity:

We see that the effectiveness of the algorithm depends on selecting the milestone value (or key).

- Best case: each partition we select the median (the one is larger or half the number of elements and is less than or half the other) as a landmark. Then the sequence is divided into two equal parts, and we need to log2(n) the partition time, then arrange it. It is also easy to see that in each partition we need to browse through n elements. So the complexity in the best case belongs to O(nlog2(n)).
- Worst case: each time the part of the plan we choose must be the most valuable or the least valuable as a landmark. Then the sequence is partitioned into two uneven

parts: one has only one, the other has n-1 elements. Therefore, we need to arrange the new partition. So the complexity in the worst case belongs to O(n^2).

In conclusion, we have the complexity of Quick Sort as follows:

- **Best case:** O(nlogn)
- **Worst case:** O(n^2)
- **Average case:** O(nlogn)
- **Space complexity:** O(nlogn)

Counting sort:

Ideas: For each x-part of the input sequence we define its rank as the number of elements less than x. Once we know the r-rank of x, we can put it in r+1 position.
For example, if there are 6 elements less than 17, we can place 17th in 7th place.
Repeat: When there is a type of elements of the same value, we place them in the order that appears in the original sequence to get the most stability of the sort

Algorithms: Here is a few steps to understand this algorithms

Step 1: Consider an input array A having n elements in the range of 0 to k, where n and k are positive integer numbers. These n elements have to be sorted in ascending order using the counting sort technique. Also note that A[] can have distinct or duplicate elements

Step 2: The count/frequency of each distinct element in A is computed and stored in another array, say count, of size k+1. Let u be an element in A such that its frequency is stored at count[u].

Step 3: Update the count array so that element at each index, say i, is equal to -

$$\text{count[i]} = \sum \text{count[u]} \quad \text{where } 0 <= u <= i$$

Step 4: The updated count array gives the index of each element of array A in the sorted sequence. Assume that the sorted sequence is stored in an output array, say B, of size n.

Step 5: Add each element from input array A to B as follows:

- Set i=0 and t = A[i]
- Add t to B[v] such that v = (count[t]-1).
- Decrement count[t] by 1
- Increment i by 1

Repeat steps (1) to (4) till i = n-1

Time complexity: For scanning the input array elements, the loop iterates n times, thus taking O(n) running time. The sorted array B[] also gets computed in n iterations, thus requiring O(n) running time. The count array also uses k iterations, thus has a running time of O(k). Thus the total running time for counting sort algorithm is O(n+k). Space complexity is O(k) Because k = O(n) so the time of the algorithm is O(n) = > the best case Where k = n*n => the algorithm works very badly.

Radix sort:

Ideas: Unlike previous algorithms, the Radix sort is an algorithm that approaches in a completely different direction. If in other algorithms, the basis for sorting is always the comparison of the value of the two elements, then Radix sort is based on the principle of mail classification of the post office. For that reason it is also known as Postmans sort. It has no interest in comparing the value of the elements and the classification itself and the classification order that will create the order for the elements.

It is known that, in order to deliver a large volume of mail to recipients in various localities, electricity often organizes a hierarchy of mail classification. First, letters to the same province or city will be arranged in a batch to be sent to the respective province. The post offices of these provinces do the same work. Letters to the same district will be placed in the same batch and sent to the corresponding district. As such, the letters will be handed out to the recipient systematicly but the work of stacking the message is not too heavy.

Algorithms: Here is few steps of this algorithms:

Step 1: k shows the digit used for the current classification

- k = 0; (k = 0: unit rows; k = 1: dozens)

Step 2: Create batches containing different types of elements

- Initially create 10 lots B0, B1, .. , B9 hollow

Step 3:

- For i = 1 .. n do
- Place someone in a Bt batch with t = who's k-digit;

Step 4:

- Connect B0, B1, .. , B9 again (in the correct order) to a.

Step 5:

- k = k+1;
- If you <, go back to step 2.

- Conversely: Stop

Time complexity: Radix sort will operate on nn dd-digit numbers where each digit can be one of at most bb different values (since bb is the base being used). For example, in base 10, a digit can be 0,1,2,3,4,5,6,7,8 ,or 9.
Radix sort uses counting sort on each digit. Each pass over n d-digit numbers will take O(n + b) time, and there are dd passes total. Therefore, the total running time of radix sort is O(d(n+b)). When d is a constant and b isn't much larger than n (in other words, b=O(n)), then radix sort takes linear time.

Flash sort:

Ideas: The basic idea behind flashsort is that in a data set with a known distribution, it is easy to immediately estimate where an element should be placed after sorting when the range of the set is known. For example, if given a uniform data set where the minimum is 1 and the maximum is 100 and 50 is an element of the set, it's reasonable to guess that 50 would be near the middle of the set after it is sorted. This approximate location is called a class. If numbered 1 to m, the class of an item Ai is the quantile, computed as:

$$K(A_i) = 1 + \mathrm{INT}\left((m-1)\frac{A_i - A_{\min}}{A_{\max} - A_{\min}}\right)$$

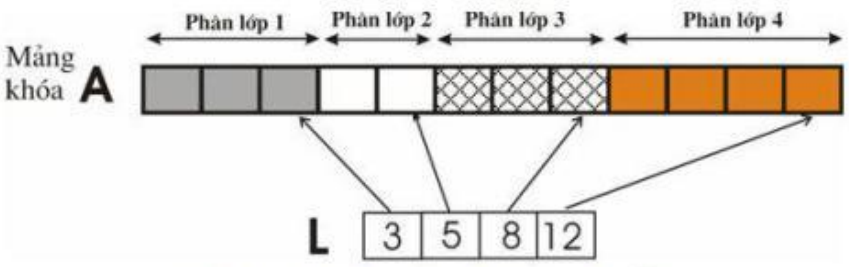
where A is the input set. The range covered by every class is equal, except the last class which includes only the maximum(s). The classification ensures that every element in a class is greater than any element in a lower class. This partially orders the data and reduces the number of inversions. Insertion sort is then applied to the classified set. As long as the data is uniformly distributed, class sizes will be consistent and insertion sort will be computationally efficient.

Algorithm: Here is few step to implementation flashsort and understand clearly
Step 1: Classification elements

- Calculate a classification index based on a formula:

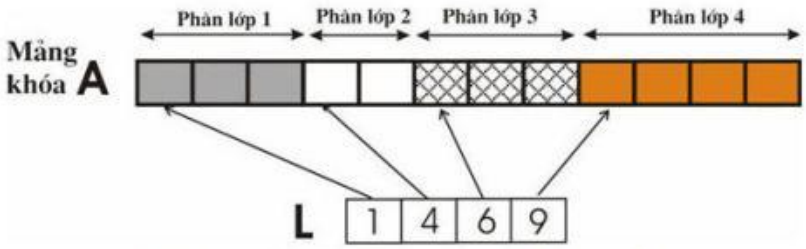
$$k_{a_i} = 1 + Int\left(\frac{(m-1)*(a_i - \min)}{(\max - \min)}\right)$$

- We use the L sub-array to mark the class position of the key array a. The first sublayer is considered hollow when L[i] is the exact position of the end of the class in the key array:



(Hình 1 – mảng phân lớp L với các phân lớp rỗng)

- Class i is considered to be full when L[i] is the exact position of the first part of the class in the key array:



(Hình 2 – mảng phân lớp L với các phân lớp đã đầy)

- Initially, the sublayer is considered hollow, L[i] is initiated by the position of the last part of class i. It is easy to see that the original position of the first subsal will be its size, while that of the m subsal will be n and L[i]=L[i-1]+ the size of the i-class subsize

Step 2: Distribute elements into the correct sublayer (Elements Permutation):

- The next stage is to perform the sorting of elements into their correct sublayer. This will form permutation cycles: every time we take a member somewhere else, we have to lift the current occupying part and continue with the lifted and taken elsewhere until we return to the original position, completing the loop. For each component we calculate where it should be in the sublayer, then put it in the current position of that subsite, and because we put in that subsite 1 part, we have to reverse the position of the sublayer to 1 unit.

Step 3: Arrange the elements in each layer in the correct order (Elements Ordering)

- In the above two stages, we have broken down one array into several small pieces (layers) to make the arrangement faster. The next job is to order the elements in each sublayer. The Straight-Insertion Sort algorithm is a good choice for this requirement

Time complexity: Flash sort is a distribution sorting algorithm showing linear computational complexity O(n) for uniformly distributed data sets and relatively little additional memory requirement.

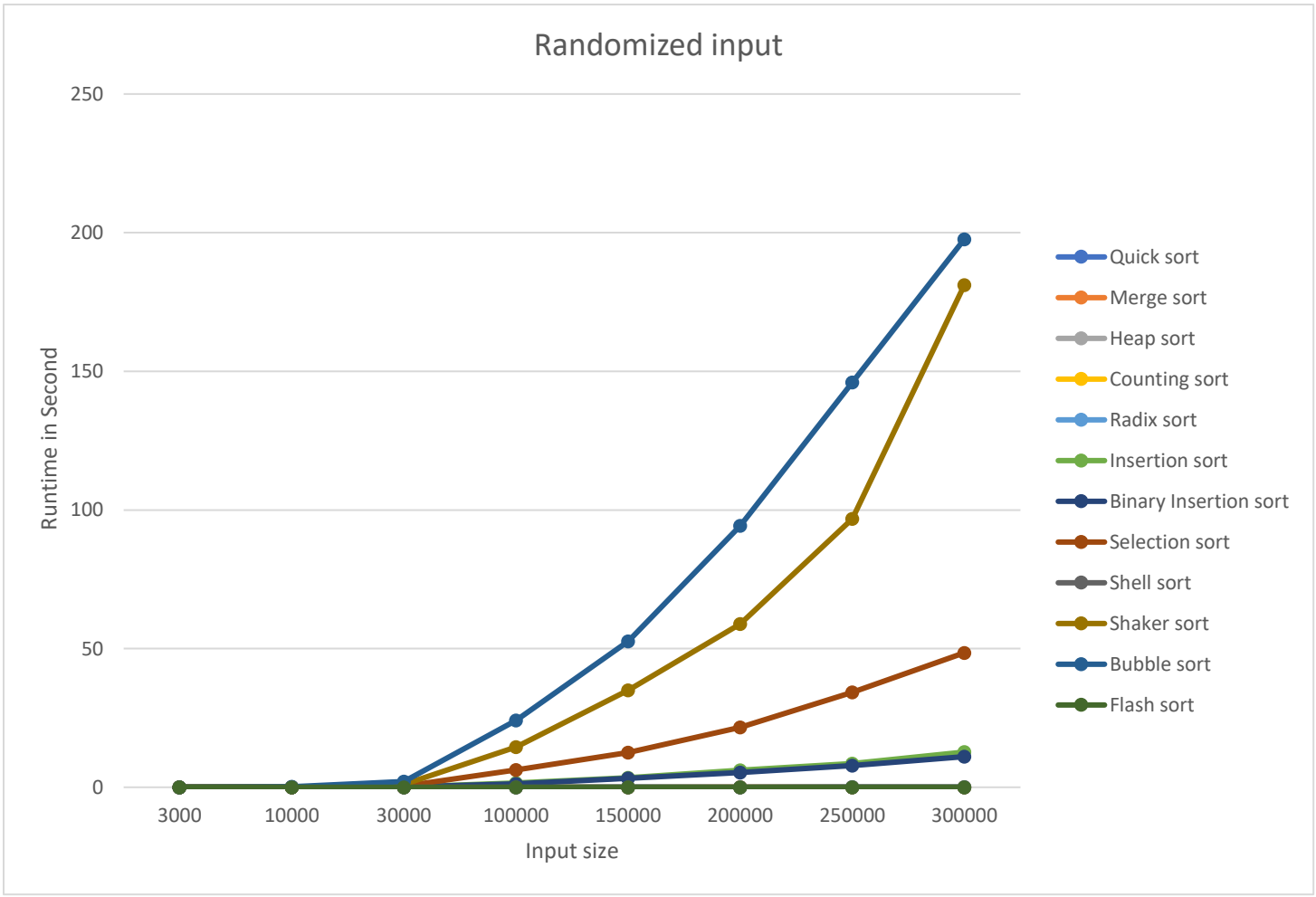
1.2 Overall Time complexity of sorting algorithms:

Sort	Best	Average	Worst	Space	Stable
Quick sort	O(nlogn)	O(nlogn)	O(n^2)	O(nlogn)	No
Merge sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Yes
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	No
Counting sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes
Radix sort	O(n.k)	O(n.k)	O(n.k)	O(n+k)	Yes
Bubble sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No
Insertion sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Binary Insertion sort	O(n)	O(nlogn)	O(nlogn)	O(1)	Yes
Shaker sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Shell sort	O(nlogn)	O(n^2)	O(n^2)	O(1)	No
Flash sort	O(n)	O(n+r)	O(n^2)	O(n)	No

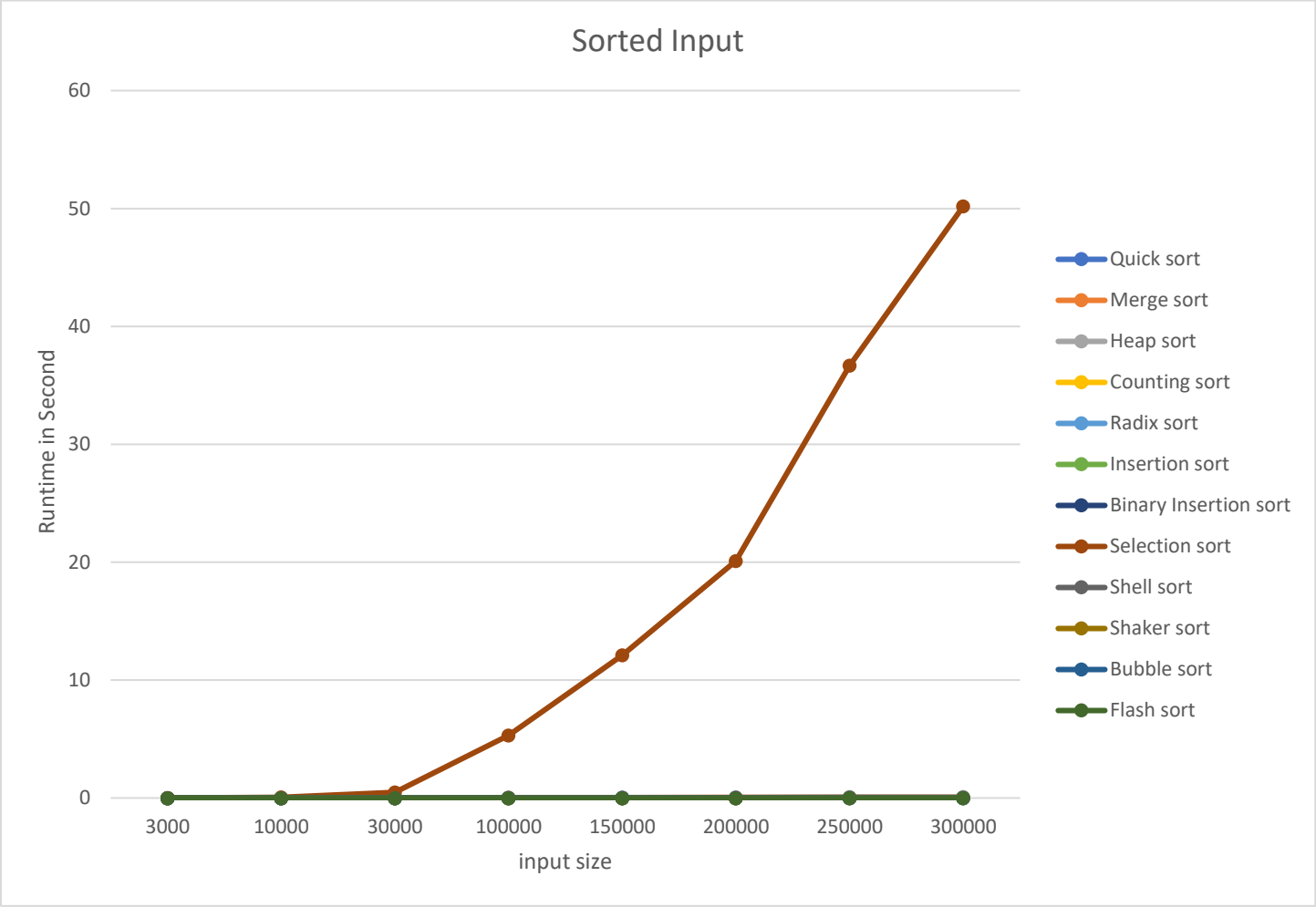
1.3 Presentation on experimental result and comments

1.3.1 Graphs

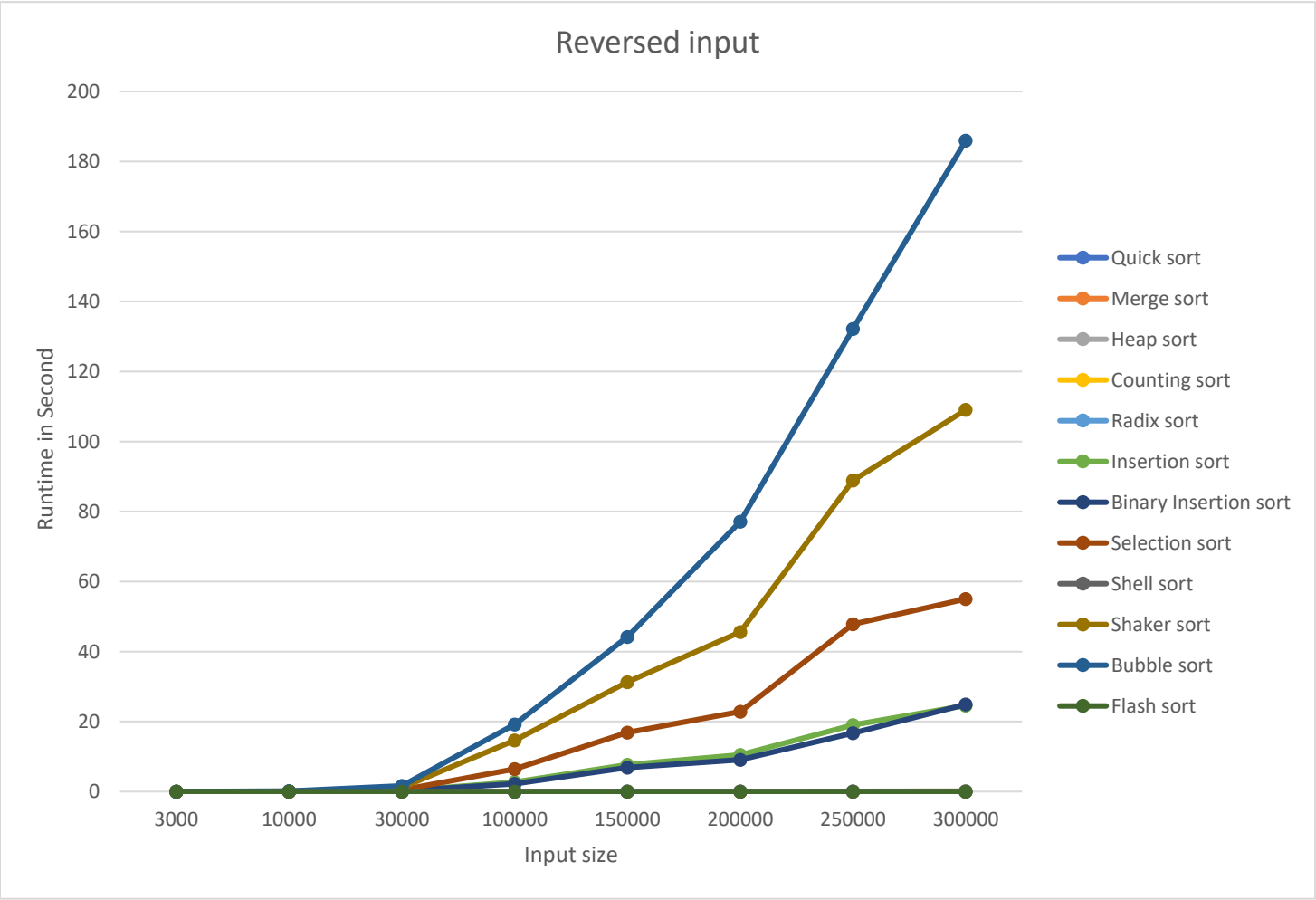
1.3.1.1 First graph



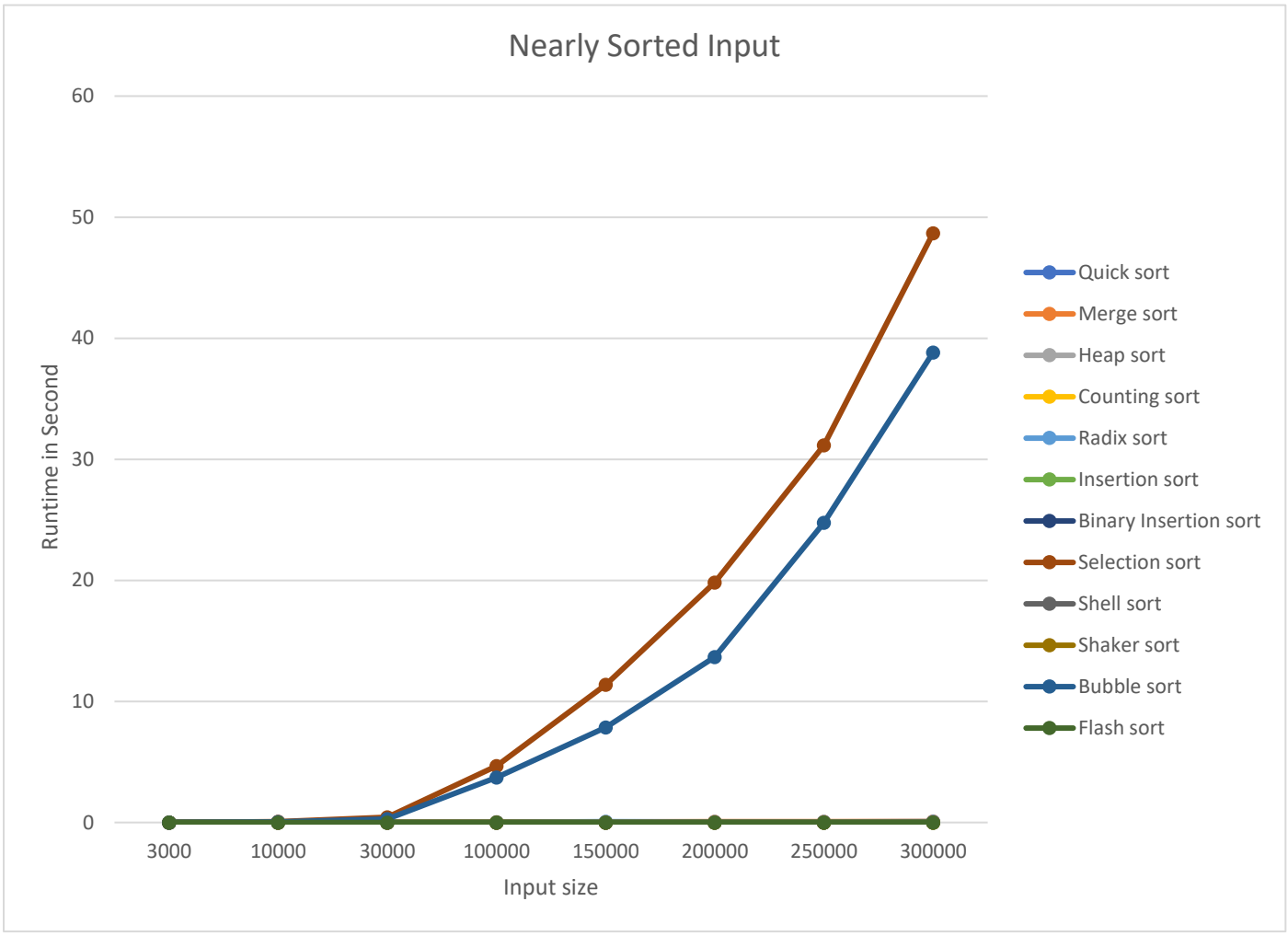
1.3.1.2 Second graph



1.3.1.3 Third graph



1.3.1.4 Fourth graph



1.3.2 Detail on each graph

1.3.2.1 First graph

In the first graph with randomize input. As you can see the complexity of Bubble sort that I mention above is O(n^2) so that is the main reason **bubble sort and Shaker sort(similar to Bubble sort, shaker sort is different a little bit time with bubble sort) which are very lowest** among algorithms in the runtime when the input more and more larger. There are two nested in this algorithm and with a large number input, this algorithm take a lot of time to sort because with randomize input number maybe it could happened in the worst case and you know in each iteration that loop n number input and there n iteration in first loop but Bubble sort is very simple to understand.

Another lowest algorithms but greater than bubble sort is Shaker sort(difference a little bit time rather than Bubble sort) and Selection sort (take a lot of time but still greater than Bubble sort and Shaker sort as well) . Similar to Bubble sort but greater than bubble sort, Shaker sort can sort 2 element at each iteration then can reduce time a half than bubble sort. Selection also has two nested loop but in each iteration it just find a smallest number in sublist the number assignment in each iteration of Selection sort less than the number exchange of pair consecutive wrong number. Therefore, Shaker sort and Selection sort is low as well.

Another sort algorithms save a little bit time than three lowest algorithms I mention above that is Insertion sort and Binary Insertion sort. With the randomize input, Using Big-O to compare algorithms is appropriate, if their Big-O are different. If their Big-O are the same, then you need to look deeper. Insertion Sort and Bubble Sort share the same big-O — O(n^2) for runtime — because as data sets get huge, Insertion Sort execution time and Bubble Sort execution time grow at essentially the same rate. However, having the same Big-O doesn't mean identical performance. Big-O ignores everything except the most-significant growth factor. It ignores constants, lesser factors, even coefficients on that most-significant factor. Specifically, Insertion is faster than Bubble because of what occurs in each pass: Bubble Sort swaps through all remaining unsorted values, moving one to the bottom. Insertion Sort swaps a value up into already-sorted values, stopping at the right place. Once a pass starts, Bubble Sort must see it through to the end. Insertion bails early. They're both Adaptive, as they both sense when they don't need another pass. And Binary insertion sort greater than Insertion sort because for each iteration, it find a correct passion to insert by binary search, it very fast than shifting in traditional insertion sort but it still a little bit time than insertion sort.

The fastest algorithms in this case are Counting sort. Radix sort, Merge sort, Quick sort, Heap and Shell sort are different time with counting sort (maybe the input is'n very large like 1 million to see clearly this difference). Counting sort and Radix sort (difference with small time) with O(n+k) and O(n.k) complexity so that it's very fastest even though with randomized input but if the input and the range may be large it is not advisable for using counting sort. Merge sort and Quick sort lower a little bit than Counting and Radix sort, these algorithms with O(nlogn) complexity . Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets whereas Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets but both of them are very fast in this case. Shell sort with O(n^2) complexity with my code(because I divide a gap each iteration, it's closely to n.(logn)^2) so that it's fast but lower than sort algorithms with O(nlogn) complexity.

1.3.2.2 Second Graph

In the second graph with sorted input number, **the lowest sort in this case is Selection sort** Because each iteration Selection sort must find a smallest number in sublist that take complexity is O(n^2). Why not bubble is lowest time in this case because according to my code, I optimized my bubble sort by putting variable swapped bool that check if the array already ordered and then out of loop, that is the reason why in this case bubble sort' complexity is O(n) (That is best case for my bubble sort in this case).

In this case, as you can witnessed a dramatic change with Bubble sort, Insertion sort and Binary Insertion sort and Shaker sort are fastest . Because of sorted number, Insertion sort and Binary Insertion sort are fastest, it's no need go into while loop due to all number are sorted and Here in such scenario, the condition at while loop always returns false and hence it only iterates for the outer for loop, doing the job in linear time with O(n) time complexity, it's very fast right. The best case for both of them sorting algorithm is when input is already in sorted order.

Bubble sort is very fast in this case because I optimized as my code, unlike traditional Bubble sort always has two nested anyway, in my code I put a variable bool Swapped that checked in each iteration is list already ordered to break out the loop then the complexity of Bubble sort in this case is O(n) (that I mention above) , it's very fast right and it's fast than Counting or Flash sort.

Another fastest sort in this case is Shaker sort, in first loop it traversal from a top of the array to end and no need to exchange then in my code variable right equal to left and next iteration return false to break out loop. Therefore, in this case the complexity of Shaker sort is O(n) which is very fast. Another fastest sorting algorithms are Merge sort Quick sort, Heap sort(three of them are different with a little bit time) and again that is Flash sort, Counting and Radix sort with complexity are O(n) still fast in this case as well.

1.3.2.3 Third Graph

In the third graph with reverse number input**, the lowest algorithm sorting still is Bubble sort and Shaker sort** that I mention above. The bubble sort is an inefficient, but simple, algorithm. It is rarely used in practice, but it is easy to understand. The bubble sort works by comparing successive pairs of elements in a list. The first and second elements are compared, then the second and third, then the third and fourth, and so on through the list. After each comparison, the two elements are swapped if they are out of order. After a single iteration, the largest element will have "bubbled up" to the end of the list. This procedure is repeated over and over until the list is sorted. Therefore, Bubble sort is almost lowest in almost case.

Shaker is very lowest in this case but still greater than Bubble sort because in each iteration instead just traversal once time of bubble sort, shaker sort traversal from top of the array to end and then traversal reverse again that better than Bubble sort so as to save a little bit time.

Another low algorithm in this case is Insertion sort and Binary Insertion sort, for each iteration a loop while return true because every number are ordered descending so that always go into while loop to find right position unlike in the ascending ordered. So the complexity of Insertion also Binary Insertion sort are O(n^2), it take a lot time but still greater than Bubble sort.

The fastest sort algorithm in this case is still Counting sort and Flash sort because Counting sort is very fast with stable step in almost cases with complexity of this algorithm is O(n). Similar to Counting sort with same O(n) therefore, Flash sort is very fast in this case.

1.3.2.4 Fourth Graph

The lowest algorithm in this case still Bubble sort and Selection sort(Bubble sort greater than Selection sort) because Bubble sort always takes one more pass over array to determine if it's sorted. On the other hand, When using selecting sort it swaps n times at most. but when using bubble sort, it swaps almost n*(n-1). And obviously reading time is less than writing time even in memory. The compare time and other running time can be ignored. So swap times is the critical bottleneck of the problem. Therefore, in this case selection is very low as well as (difference few millisecond to second) with Bubble sort

In this case why bubble sort isn't fast than or equal to the ordered input array above, because I mention above, even though my code is optimized but it happened out loop (complexity O(n)) only when the input already sorted and after 1 iteration variable bool return true and out of ancestor loop but in this case there are many elements are mixed with ordered sublist so nothing happened with variable swapped always return false. (Explain why with 2 input structure Bubble sort' result are different).

The fastest algorithm sorting in this case is Shaker sort, Insertion sort and Binary Insertion sort (difference with small time)

In this case Insertion sort and Binary insertion sort again on the top of fast algorithms because best case of both is the ordered array and for each iteration Shifting method of insertion sort and Binary search method just spend a little bit step to find correct position to insert. Therefore, the complexity of both algorithms are closely O(n).

Shaker sort in this case just swap few elements that's wrong position and then the array already sorted quickly, moreover in each iteration of shaker sort traversal 2 direction so shaker sort in this case is very fast right.

Counting sort, Radix sort and Flash sort are very fast in almost case because the complexity of these algorithms is O(n) that very fast for sorting almost case. Quick sort is still fast in this case. Although Quicksort has worst-case O(n^2) behaviour, it is usually fast: assuming random pivot selection, there's a very large chance we pick some number that separates the input into two similarly sized subsets, which is exactly what we want to have. In particular, even if we pick a pivot that creates a 10%-90% split every 10 splits (which is a meh split), and a 1 element n−1 element split otherwise (which is the worst split you can get), our running time is still O(nlogn) (note that this would blow up the constants to a point that Merge sort is probably faster though). And also quick sort fast or low depend on you pick up a pivot, in many case it's hard to pick a edge number in the array which is very low probability. Therefore, in many case quick sort is fast but not stable like merge sort.
Merge sort in many case like this case lower than quick sort(just a little bit)

Counting sort is stable sort in many cases that tested in almost case, it's very fast almost cases as well as Radix sort do, but Radix sort when the number input of array more and more larger it's comparison use for each digit is larger.

1.3.3 Overall graphs

To conclusion all the algorithms, The fastest algorithms for sorting integer structure number with time acceleration is Counting sort. Counting sort always be stable step in almost cases. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K. 2. It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.

Time Complexity Analysis For scanning the input array elements, the loop iterates n times, thus taking O (n) running time. The sorted array B [] also gets computed in n iterations, thus requiring O (n) running time. The count array also uses k iterations, thus has a running time of O (k). To sum of the complexity of Counting sort is O(n+k), it's very fast in almost cases and stable

In computer science, counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for integer number

The lowest sorting algorithms are Bubble sort and Selection sort in many cases even though with the array already ordered, Bubble sort is very fast because I put variable swapped to check in order to break out of loop, but rest of case this algorithm is very low than other sort algorithms because it own two loop, in each iteration this algorithm exchange pair of consecutive number if check back greater than front number, and if in this reverse cases, this algorithm must be swap n-1 times in each iteration to "bubbled" largest number into last index of sub list in the runtime. Therefore, Bubble sort take a lot of time with complexity almost case is O(n^2). Selection sort own two nested loop so anyway, it also go into 2 loop in the runtime and the complexity in almost case of Selection sort is

O(n^2). To conclusion, in the reality life, we usually have a problem with almost nearly sorted. For example, if you have a large information of student maybe somehow to add to this list not very large and then it's not change this list a lot (which is nearly sorted) and in this case bubble is very useful. So bubble is useful in reality and Selection is adaptive with practice and theory.

Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

According to 4 graphs above: Counting sort, Merge sort, Insertion sort, Binary Insertion sort, Radix sort, Bubble sort, Shaker sort are stable Others such as Quicksort, Heapsort, Selection Sort, Shell sort, Flash sort are unstable.

Counting sort and Radix sort are stable sort because non-comparison and non-recursive based sorting. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort. Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only n/2 extra spaces). Insertion sort and Binary insertion sort are stable sorting algorithm, just pick a element and places it in its correct place and in the logic we are only swapping the elements if the element is larger than the key, they are not swapping the element with the key when it holds equality condition. Bubble sort and Shaker sort are a stable algorithm, both of sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Quick sort isn't stable method as it doesn't preserve the order of the elements that have same value before and after sorting. Heap sort is not stable because operations in the heap can change the relative order of equivalent keys. The binary heap can be represented using array-based methods to reduce space and memory usage. Heap sort is an in-place algorithm, where inputs are overwritten using no extra data structures at runtime. Selection sort is not a stable sort because elements which are equal might be re-arranged in the final sort order relative to one another. Selection Sort makes O(N^2) comparisons (every element is compared to every other element). Shell sort is not stable: it may change the relative order of element with equal value.

Due to the in situ permutation that Flash sort performs in its classification process, Flash sort is not stable. If stability is required, it is possible to use a second, temporary, array so elements can be classified sequentially.

2 Reference:

[1] https://drive.google.com/file/d/135PAmaRYkAiMWFHhiLzQDDznsb_R381I/view?usp=sharing (Data confirguation to draw graph)

[2] *Flashsort - Wikipedia*

[3] *https://en.wikipedia.org/wiki/Binary_search_algorithm*

[4] *https://medium.com/smelly-code/binging-binary-search-aa172b5b31c2*

[5] *https://en.wikipedia.org/wiki/Quicksort*

[6] *https://en.wikipedia.org/wiki/Merge_sort*

[7] *https://en.wikipedia.org/wiki/Cocktail_shaker_sort*

[8] *https://en.wikipedia.org/wiki/Shellsort*

[9] *https://en.wikipedia.org/wiki/Heapsort*

[10] *https://www.geeksforgeeks.org/radix-sort/*

[11] The Algorithm Design Manual *2nd ed. 2008 Edition. by* Steven S S. Skiena

[12] *Data Structures and Algorithms [Aho, Alfred, Ullman, Jeffrey, Hopcroft, John]*

[13] *Algorithms and complexity by Herbert S. Wilf*

- *Use Excel to draw graph, table*
- *Font: Calibri(body) and Times New Roman*