CS263

ASSIGNMENT 1

NAME:

ARCHIT AGRAWAL

ROLL NO. :

202052307

SECTION:

A

Task 1: Write the algorithm of Linear Search, Binary Search, Insertion Sort, Selection Sort, and Bubble Sort. Use the Step count method and write the time function for the algorithm.
Create a table and write the complexity in "big-Oh" notations.

Table 1: Time complexity

Algorithm	Best Case	Average Case	Worst Case
Linear Search			
Binary Search			
Insertion Sort			
Selection Sort			
Bubble Sort			

Archit Agrawal
2020052307

CS263 : Assignment 1

Task 1 : Write the algorithm of Linear Search, Binary Search, Bubble Sort, Selection Sort and Insertion Sort. Use the step count method and write the time function for the algorithm. Create a table and write the complexity in "Big-oh" notations.

(i) Linear Search :-

Algorithm :

- Start from leftmost element of arr[] and one by one compare the target element x with each element of arr[].
- If x matches with an element, return the index, else return -1.

Code : (Java)

```
public int linearSearch(int[] arr, int num){
    for(int i=0; i<arr.length; i++){
        if(arr[i]==num) return i;
    }
    return -1;
}
```

Archit Agrawal
2020523O7

Step Count Analysis

here n = arr. length

	Step	Cost	Frequency	Final Cost
1.	for(int i=0; i<arr.length; i++)	c_1	$n+1$	$c_1(n+1)$
2.	if (arr[i] == num)	c_2	n	$c_2 n$
3.	return i	c_3	1	c_3
4.	return -1	c_4	1	c_4

total cost = $c_1(n+1) + c_2 n + c_3 + c_4$

$$T(n) = (c_1+c_2)n + (c_1+c_3+c_4)$$

After neglecting the constant terms, we get the 'Big-Oh' time complexity for linear search. which is O(n).

Best Case: The best case occurs when target element is present at index 0 of arr. In this case, the step 1, 2 will be executed only once and step 3 will also be executed only once and the best case complexity will be O(1).

Worst Case: The worst case occurs when either the element is not present in the array or it is present at last index of array. The complexity in this case

Is $O(n)$.

Average Case: The average case occurs when target element is present in the array at an index other than first and last index of array. The average case complexity can be considered to be $O(n/2)$ because the target element can be considered to be at equal distance from the middle element of array.

(ii) Binary Search: If an element is to be searched in an sorted array, we can use binary search.

Algorithm:

Assume the array is sorted in non-decreasing order.

- Compare x with the middle element.
- If x matches with the middle element, we return the mid index.
- Else if x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So, we recur for the right half
- Else (x is smaller) recur for the left half.

Archit Agrawal
202052307

Code :

```
public int binarySearch(int[] nums, int target){
    int ans = -1;
    int i = 0;
    int j = nums.length - 1;

    while ( j >= i){
        int mid = (i+j)/2;

        if (nums[mid] == target){
            ans = mid;
            break;
        } else if (nums[mid] < target){
            i = mid + 1;
        } else {
            j = mid - 1;
        }
    }

    return ans;
}
```

Archit Agrawal
2020052307

Step-Count Analysis

n = nums.length

	Step	Cost	Frequency	Final Cost
1	int ans = -1	c_1	1	c_1
2	int i = 0	c_2	1	c_2
3	int j = nums.length-1	c_3	1	c_3
4	while (j >= i)	c_4	$\log_2(n+1)$	$(\log_2 n+1)\, c_4$
5	int mid = (i+j)/2	c_5	$\log_2(n)$	$(\log_2 n)\, c_5$
6	if-else if ladder	c_6	$\log_2(n)$	$(\log_2 n)\, c_6$
7	return ans	c_7	1	c_7

total cost $= (c_4 + c_5 + c_6)(\log_2 n) + (c_1 + c_2 + c_3 + c_7)$

$$T(n) = k_1 \log_2 n + k_2$$

where $c_4 + c_5 + c_6 = k_1$ and $c_1 + c_2 + c_3 + c_7 = k_2$

Ignoring the constant terms, the 'big-Oh' time complexity for binary search is $O(\log_2 n)$.

Best Case: The best case occurs when the target element is found at central index of array. Steps 4-5 are executed only once in this case and the best case time complexity of binary search is $O(1)$.

Worst Case: The worst case occurs when either the target element is present at first or last index or if the element is not present in the array. All the steps are executed maximum number of times and the complexity in this case is $O(\log_2 n)$.

Average Case: It occurs when the target element is present at an index other than the first, central and last index of array. The complexity in this case is $O(\log n)$.

(iii) Bubble Sort:

Algorithm:

Swap(a,b) means $a = b$ and $b = a$.

~~start bubbleSort(array):~~

~~for all elements of array~~

- In this algorithm we take passes of the array.
- In each pass, we keep comparing adjacent elements and swap if the element at right is smaller than the element at left.
- the greatest element reaches the end of array this way.
- Each of the next pass is done till (last - pass No) index of array.

Archit Agrawal
2019052307

Code:

```
public static void bubbleSort (int[] arr){
    for (int i=0 ; i < arr.length; i++){
        for (int j= 0; j<arr.length - i -1 ; j++){
            if (arr[j] > arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
```

Archit Agrawal
2020052307

	Step	Cost	Frequency	Final Cost
1.	for(int i=0; i<arr.length; i++)	c_1	$n+1$	$(n+1)c_1$
2.	for(int j=0; j<arr.length-i-1; j++)	c_2	$n(n+1)$	$n(n+1)c_2$
3.	if (arr[j] > arr[j+1])	c_3	$n(n+1)$	$n(n+1)c_3$
4.	int temp = arr[j]	c_4	$n(n+1)$	$n(n+1)c_4$
5.	arr[j] = arr[j+1]	c_5	$n(n+1)$	$n(n+1)c_5$
6.	arr[j+1] = temp	c_6	$n(n+1)$	$n(n+1)c_6$

total cost $= (n+1)c_1 + n(n+1)(c_2 + c_3 + c_4 + c_5 + c_6)$

$$T(n) = k_2 n^2 + k_1 n + k_0$$

where $k_2 = c_2 + c_3 + c_4 + c_5 + c_6$

$k_1 = c_1 + c_2 + c_3 + c_4 + c_5 + c_6$

$k_0 = c_1$

Ignoring the constant terms, the 'big-oh' time complexity for bubble sort is $O(n^2)$.

Best-Case: best case occurs when the array is already sorted. The complexity in this case is $O(n^2)$ only but the number of swaps i.e. step 4-6 are reduced to zero.

Worst-Case: Worst case occurs when the array is reverse sorted. The time

complexity remains $O(n^2)$ but maximum number of swaps occur in this case.

Average Case: It occurs when the input array is neither sorted nor reverse sorted. The number of swaps in this case are between zero and maximum and time complexity remains $O(n^2)$.

(iv) Selection Sort: The selection sort algorithm sorts an array by repeatedly finding the minimum (or maximum) element (considering ascending order) from unsorted part and putting it at the beginning (end). The algorithm maintains two subarrays in a given array.

a) The subarray which is already sorted.
b) Remaining subarray which is unsorted.

Code:

```
public static void selectionSort(int[] arr){

    for(int i=0; i<arr.length; i++){
        int greatest = arr[0];
        int tempJ = 0;
        for(int j=1; j<arr.length-i; j++){
            if(arr[j] > greatest){
                greatest = arr[j];
                tempJ = j;
            }
        }
        int tempA = arr[arr.length-i-1];
```

Archit Agrawal
2020052307

```
        arr[arr.length - i - 1] = greatest;
        arr[tempJ] = tempA;
    }
}
```

Step-Count Analysis:

	Step	Cost	frequency	Final cost
1.	for(int i=0; i<arr.length; i++)	c_1	$n+1$	$(n+1)c_1$
2.	int greatest = arr[0];	c_2	n	nc_2
3.	int tempJ = 0;	c_3	n	nc_3
4.	for(int j=0; j<arr.length-i; j++)	c_4	$n(n+1)$	$n(n+1)c_4$
5.	if (arr[j] > greatest)	c_5	$n(n+1)$	$n(n+1)c_5$
6.	greatest = arr[j]	c_6	$n(n+1)$	$n(n+1)c_6$
7.	tempJ = j	c_7	$n(n+1)$	$n(n+1)c_7$
8.	int tempA = arr[arr.length-i-1]	c_8	n	nc_8
9.	arr[arr.length-i-1] = greatest	c_9	n	nc_9
10.	arr[tempJ] = tempA	c_{10}	n	nc_{10}

$$\text{total cost} = n(c_2+c_3+c_8+c_9+c_{10}) + (n+1)c_1 + n(n+1)(c_4+c_5+c_6+c_7)$$

$$= (c_4+c_5+c_6+c_7)n^2 + (c_1+c_2+\cdots+c_{10})n + c_1$$

$$\boxed{T(n) = k_2 n^2 + k_1 n + k_0}$$

where, $k_2 = c_4+c_5+c_6+c_7$

$k_1 = c_1+c_2+\cdots+c_{10}$

$k_0 = c_1$

Archit Agrawal
2020052307

Best Case: The best case occurs when the input array is already sorted. The time complexity in this case is $O(n^2)$ only but the number of swaps are reduced to zero or $O(1)$. There are still $O(n^2)$ comparisons.

Worst - Case: The worst case / ~~accord~~ average case / Average Case when the array is not already sorted. In such an input, there are $O(n^2)$ comparisons and $O(n)$ swaps and therefore, the time complexity is $O(n^2)$.

(v) **Insertion Sort -:**

Algorithm:
To sort an array of size n in ascending order:

- iterate from arr[1] to arr[n-1] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Archit Agrawal
2020052307

Code:

```
public static void insertionSort(int [] arr){

    int key;
    for(int i=1; i<arr.length; i++){
        key = arr[i];
        int j = i-1;
        while(j>=0 && arr[j]>key){
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

Step-Count Analysis:-

	Step	Cost	Frequency	Final Cost
1.	int key;	C_1	1	C_1
2.	for(int i=1; i<arr.length; i++)	C_2	$(n+1)$	$(n+1)C_2$
3.	key = arr[i]	C_3	n	nC_3
4.	int j = i-1	C_4	n	nC_4
5.	while(j>=0 && arr[j]>key)	C_5	$n(n+1)$	$n(n+1)C_5$
6.	arr[j+1] = arr[j]	C_6	$n(n+1)$	$n(n+1)C_6$
7.	j = j-1	C_7	$n(n+1)$	$n(n+1)C_7$
8.	arr[j+1] = key	C_8	n	nC_8

Archit Agrawal
2020052307

total cost $= n(n+1)(C_5 + C_6 + C_7) + n(C_3 + C_4 + C_8) + (n+1)C_2 + C_1$

$T(n) = (C_5 + C_6 + C_7)n^2 + n(C_2 + C_3 + \cdots + C_8) + C_1 + C_2$

$$T(n) = k_2 n^2 + k_1 n + k_0$$

Best Case: Best case occurs when the input array is already sorted. The comparisons in this case are $O(n)$ and swaps are $O(1)$. This reduces the time complexity to $O(n)$ in best-case.

Worst-Case: It occurs when the input array is reverse sorted. In this case each element is compared to all its preceding elements. Hence, there are $O(n^2)$ comparisons. Hence, the time complexity is $O(n^2)$.

Average Case: The average case is when the array is neither sorted nor reverse sorted. The comparisons lie between $O(n)$ and $O(n^2)$ in this case and swaps lie between $O(1)$ and $O(n)$. The time complexity is $O(n^2)$ in this case.

Archit Agrawal
202052307

n = array size

Algorithm	Best-case	Average case	Worst case
Linear Search	$O(1)$	$O(n/2)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Task 2: Implement the above algorithms in any language. There will be minimum 100000 elements in each array.
Initialize this three arrays: a[], b[] and c[] in following ways. Make sure input is same to all sorting algorithms.

- Elements of a[] should be in increasing order
- Elements of b[] should be decreasing order
- Elements of c[] will be in random order.

Execute your algorithm for each input array and compute the execution time in (microseconds).

CODE

```java
import java.util.*;

public class Search{

    public static int linearSearch(int[] arr, int num){

        for(int i = 0; i < arr.length; i++){
            if(arr[i] == num) return i;
        }

        return -1;
    }

    public static int binarySearch(int[] nums, int target) {
        int ans = -1;
        int i = 0;
        int j = nums.length - 1;

        while(j >= i){
            int mid = (i + j) / 2;

            if(nums[mid] == target){
                ans = mid;
                break;
            } else if(nums[mid] < target){
                i = mid + 1;
            } else {
                j = mid - 1;
            }
        }

        return ans;
    }
```

```
    public static void insertionSort(int[] arr){

        //Insertion Sort Algorithm
        int key;
        for (int i = 1; i < arr.length; i++){
            key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key){
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }

    }

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of array: ");
        int n = sc.nextInt();

        Random rand = new Random();
        int[] arr = new int[n];

        System.out.println("Creating the array using Random class: ");

        for(int i = 0; i < n; i++){
            arr[i] = rand.nextInt(10000);
        }

        System.out.println();
        int x = arr[rand.nextInt(n)];

        double start = System.nanoTime();
        linearSearch(arr, 0);
        double end = System.nanoTime();

        System.out.println("Time taken in Linear Search (best case) in millise
conds is : "+ (end - start)/1000000.0);

        start = System.nanoTime();
        linearSearch(arr, arr.length - 1);
        end = System.nanoTime();

        System.out.println("Time taken in Linear Search (worst case) in millis
econds is : "+ (end - start)/1000000.0);
```

```
        start = System.nanoTime();
        linearSearch(arr, x);
        end = System.nanoTime();

        System.out.println("Time taken in Linear Search (average case) in mill
iseconds is : "+ (end - start)/1000000.0);

        System.out.println();
        System.out.println("Sorting the array..");
        insertionSort(arr);

        System.out.println();

        x = arr[rand.nextInt(n)];

        start = System.nanoTime();
        binarySearch(arr, arr.length/2);
        end = System.nanoTime();

        System.out.println("Time taken in Binary Search (best case) in millise
conds is : "+ (end - start)/1000000.0);

        start = System.nanoTime();
        binarySearch(arr, arr.length - 1);
        end = System.nanoTime();

        System.out.println("Time taken in Binary Search (worst case) in millis
econds is : "+ (end - start)/1000000.0);

        start = System.nanoTime();
        binarySearch(arr, x);
        end = System.nanoTime();

        System.out.println("Time taken in Binary Search (average case) in mill
iseconds is : "+ (end - start)/1000000.0);

    }
}
```

Output

```
Enter the size of array:
100000
Creating the array using Random class:

Time taken in Linear Search (best case) in milliseconds is : 0.285
Time taken in Linear Search (worst case) in milliseconds is : 2.025
Time taken in Linear Search (average case) in milliseconds is : 0.0275

Sorting the array..

Time taken in Binary Search (best case) in milliseconds is : 0.0254
Time taken in Binary Search (worst case) in milliseconds is : 0.0022
Time taken in Binary Search (average case) in milliseconds is : 0.0029
PS C:\Users\Archit\Desktop\cprog>
```

CODE

```java
import java.util.*;

public class Sort{

    public static void bubbleSort(Integer []arr){

        //Bubble Sort Algorithm
        for(int i = 0; i < arr.length ; i++){
            for(int j = 0; j < arr.length - i - 1; j++){
                if(arr[j] > arr[j + 1]){
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp ;
                }
            }
        }
    }

    public static void selectionSort(Integer []arr){

        //Selection Sort Algorithm
        for(int i = 0; i < arr.length; i++){
            int greatest = arr[0];
            int tempJ = 0;
            for(int j = 1; j < arr.length - i ; j++){
                if(arr[j] > greatest ){
                    greatest = arr[j] ;
                    tempJ = j ;
                }
            }
            int tempA = arr[arr.length - i - 1] ;
            arr[arr.length - i - 1] = greatest;
```

```
            arr[tempJ] = tempA;
        }

    }

    public static void insertionSort(Integer []arr){

        //Insertion Sort Algorithm
        int key;
        for (int i = 1; i < arr.length; i++){
            key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key){
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }

    }


    public static void main(String []args){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of array");
        int n = sc.nextInt();

        System.out.println("1. If you want the input array to be randomly arra
nged, enter 1");
        System.out.println("2. If you want the input array to be arranged in i
ncreasing order, enter 2");
        System.out.println("3. If you want the input array to be arranged in d
escending order, enter 3");
        int order = sc.nextInt();

        Integer[] a = new Integer[n];
        Integer[] b = new Integer[n]; //a copy of a[]
        Integer[] c = new Integer[n]; //another copy of a[]
        Random rand = new Random();

        for (int i = 0; i < n; i++) {
            a[i] = rand.nextInt(9000) + 1000;
            //System.out.print(a[i] + " ");
        }

        System.out.println();

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

```
            c[i] = a[i];
        }
        //a[] will be bubble sorted
        //b[] will be selection sorted
        //c[] will be insertion sorted
        // as random will give a new number everytime that is why
        //a copy of a[] is created to ensure that all the sorting
        //methods gets the same array

        if (order == 2) {
            Arrays.sort(a);
            Arrays.sort(b);
            Arrays.sort(c);
        } else if (order == 3) {
            Arrays.sort(a, Collections.reverseOrder());
            Arrays.sort(b, Collections.reverseOrder());
            Arrays.sort(c, Collections.reverseOrder());
        }

        double start = System.nanoTime();
        bubbleSort(a);
        double end = System.nanoTime();
        double time_in_bubble = (end - start)/1000000.0;

        start = System.nanoTime();
        selectionSort(b);
        end = System.nanoTime();
        double time_in_selection = (end - start)/1000000.0;


        start = System.nanoTime();
        insertionSort(c);
        end = System.nanoTime();
        double time_in_insertion = (end - start)/1000000.0;


        System.out.println("**************************************************
**************************************************************************
*****************");
        System.out.printf("Time taken in bubble sort in milliseconds : %.3f ",
time_in_bubble);
        System.out.println();
        System.out.printf("Time taken in selection sort milliseconds : %.3f ",
time_in_selection);
         System.out.println();
        System.out.printf("Time taken in insertion sort milliseconds : %.3f ",
time_in_insertion);
    }
}
```

Output

- ***Input is randomly arranged***

```
Enter the size of array
10000
1. If you want the input array to be randomly arranged, enter 1
2. If you want the input array to be arranged in increasing order, enter 2
3. If you want the input array to be arranged in descending order, enter 3
1

********************************************************************************
Time taken in bubble sort in milliseconds : 351.844
Time taken in selection sort milliseconds : 59.091
Time taken in insertion sort milliseconds : 68.972
PS C:\Users\Archit\Desktop\cprog>
```

- ***Input is non-decreasingly arranged***

```
Enter the size of array
10000
1. If you want the input array to be randomly arranged, enter 1
2. If you want the input array to be arranged in increasing order, enter 2
3. If you want the input array to be arranged in descending order, enter 3
2

********************************************************************************
Time taken in bubble sort in milliseconds : 107.578
Time taken in selection sort milliseconds : 71.329
Time taken in insertion sort milliseconds : 0.638
PS C:\Users\Archit\Desktop\cprog>
```

- ***Input is non-incsreasingly arranged***

```
Enter the size of array
10000
1. If you want the input array to be randomly arranged, enter 1
2. If you want the input array to be arranged in increasing order, enter 2
3. If you want the input array to be arranged in descending order, enter 3
3

********************************************************************************
Time taken in bubble sort in milliseconds : 403.311
Time taken in selection sort milliseconds : 78.558
Time taken in insertion sort milliseconds : 122.901
PS C:\Users\Archit\Desktop\cprog>
```

Execute your algorithm for each input array and compute the execution time in (microseconds).

Task 3: Plot the graph as execution time vs. the input size for each algorithm. Input size varies as 100, 1000, 10000, 100000, 10000000.

Bubble Sort

Size	Time(in milliseconds)
100	0.554
500	8.001
1000	19.526
5000	130.355
10000	365.168

Selection Sort

Size	Time(in milliseconds)
100	0.139
500	3.748
1000	5.276
5000	19.791
10000	58.391

Insertion Sort

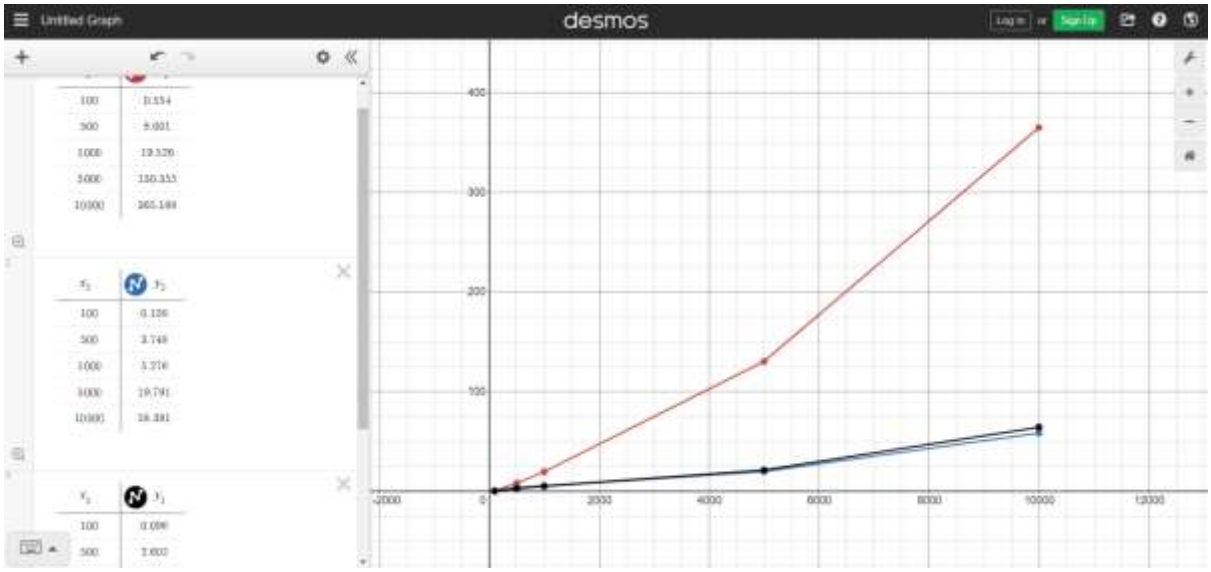
Size	Time(in milliseconds)
100	0.096
500	2.602
1000	5.028
5000	21.168
10000	64.239

Graph

Bubble Sort (in Red)

Selection Sort (in Blue)

Insertion Sort (in Black)



The codes weren’t executing for the input sizes mentioned in the assignment, hence different array sizes are used.