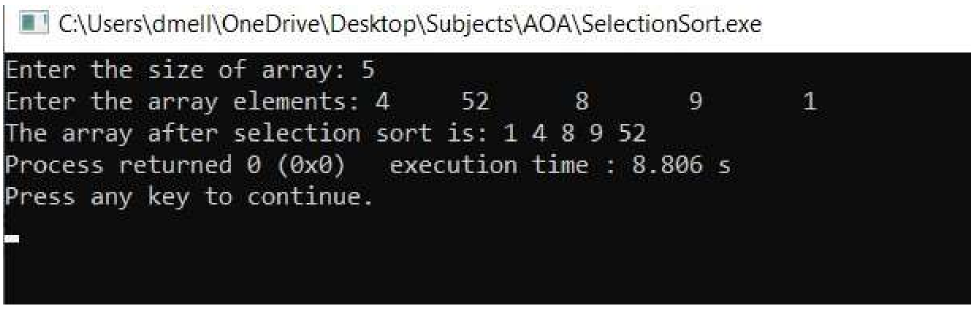
Name : Vanessa D’mello
Roll No. : 8863
Branch : SE Computer A (Batch A)
Practical No. : 1

Selection Sort

```
#include<stdio.h>
int main()
{
	int a[50], i,j,n,temp,min;
	printf("Enter the size of array: ");
	scanf("%d", &n);
	printf("Enter the array elements: ");
	for(i=0; i<n; i++)
		scanf ("%d", &a[i]);
	for (i=0; i<n-1; i++)
	{
		min = i;
		for (j = i+1; j<n; j++)
		{
			if (a[j] < a[min])
				min=j;
		}
		temp = a[i];
		a[i] = a[min];
		a[min] = temp;
	}
	printf ("The array after selection sort is: ");
	for (i=0; i<n; i++)
		printf("%d ", a[i]);
	return 0;
}
```



```
C:\Users\dmell\OneDrive\Desktop\Subjects\AOA\SelectionSort.exe
Enter the size of array: 5
Enter the array elements: 4     52      8       9       1
The array after selection sort is: 1 4 8 9 52
Process returned 0 (0x0)   execution time : 8.806 s
Press any key to continue.
```

Insertion Sort

```
#include <stdio.h>
int main()
{
	int n, array[1000], i, j, temp, flag = 0;
	printf("Enter the size of the array: ");
	scanf("%d", &n);
	printf("Enter the elements of the array: ");
	for (i = 0; i < n; i++)
		scanf("%d", &array[i]);
	for (i = 1 ; i <= n - 1; i++)
	{
		temp = array[i];
		for (j = i - 1 ; j>= 0; j--)
		{
			if (array[j] > temp)
			{
				array[j+1] = array[j];
				flag = 1;
			}
			else
				break;
		}
		if (flag)
			array[j+1] = temp;
	}
	printf("Array after insertion sort: ");
	for (i = 0; i<= n - 1; i++)
		printf("%d ", array[i]);
	return 0;
}
```

C:\Users\dmell\OneDrive\Desktop\Subjects\AOA\Insertionsort.exe

```
Enter the size of the array: 5
Enter the elements of the array: 5      25      8       1       9
Array after insertion sort: 1 5 8 9 25
Process returned 0 (0x0)   execution time : 10.755 s
Press any key to continue.
```

8863

Space complexity for Insertion and Selection Sort

It performs all computations in original array and no other array is used, hence space complexity of selection and insertion sort is $O(1)$.

Time complexity of Selection sort.

For all cases, the complexity of selection sort is same which can be derived as

$$f(n) = \sum_{pass=1}^{n-1} \sum_{i=pass}^{n-1} 1$$

$$= \sum_{pass=1}^{n-1} (n-1-pass+1)$$

$$= \sum_{pass=1}^{n-1} (n-pass)$$

$$= (n-1) + (n-2) + \cdots + 1$$

$$= \sum_{i=1}^{n-1} i$$

$$= \frac{(n-1)(n)}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2}$$

$\therefore$ By ignoring lower order terms and constant coeffient of higher order, we get

$$f(n) = O(n^2)$$

$\therefore$ Time complexity is quadratic

PAGE No.
DATE / /

8863

Time Complexity for Insertion Sort

From above code, there are 3 cases for this algorithm

① Best Case

This occurs when array is completely sorted

eg. | 1 | 2 | 3 | 4 | 5 |

For every pass, there is will be 1 comparison.

Passes	Comparison
1	1
2	1
⋮	⋮
(n-1)	1

$$f(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1)$$

∴ By ignoring lower order terms, we get

$$f(n) = \Omega(n)$$

∴ Best case time complexity is linear

② Worst Case

It happens when array is in reverse order

eg. | 5 | 4 | 3 | 2 | 1 |

Passes	Comparisons
1	1
2	2
⋮	⋮
(n-1)	(n-1)

$$f(n) = 1 + 2 + \cdots + (n-2) + (n-1)$$

$$f(n) = \sum_{i=1}^{n-1} i$$

$$= \frac{(n)(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

8863

By ignoring lower order terms and constant coeffient of higher order we get

$$f(n) = O(n^2)$$

$\therefore$ Worst case time complexity is quadratic

③ Average Case

Here we consider that each element is inserted half way in order

$\therefore$ complexity is given as

Passes	Comparisons
1	1
2	2
⋮	⋮
(n-1)	(n-1)

$$f(n) = \frac{1}{2}\left[1 + \cdots + (n-2) + (n-1)\right]$$

$$= \frac{1}{2}\left[\sum_{i=1}^{n-1} i\right]$$

$$= \frac{1}{2}\left[\frac{n^2}{2} - \frac{n}{2}\right]$$

$$= \frac{n^2}{4} - \frac{n}{2}$$

By ignoring lower order terms and constant coefficient of higher order we get

$$f(n) = \Theta(n^2)$$

$\therefore$ Average time complexity is quadratic

8863

DATE / /

Postlab Questions

Q1] Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. These notations are.

1] Big - Oh Notation : Big Oh (O) Notation gives an upper bound for a function f(n) within a constant factor, we write $f(n) = O(g(n))$, if there are positive constants no and c such that, to the right of n_0, f(n) always lies on or below $c * g(n)$

$$O(g(n)) = \{ f(n) : \text{There exists positive constant } n_0 \text{ and } c \text{ such that } 0 \leq f(n) \leq c * g(n) \text{ for all } n \geq n_0 \}$$

2] Big - Omega Notation: Big - Omega (Ω) Notation gives a lower bound for a function f(n) within a constant factor, we write $f(n) = \Omega(g(n))$, if there are positive constants n_0 and c such that to the right of n_0, the f(n) always lie on or above $c * g(n)$

$$\Omega(g(n)) = \{ f(n) : \text{There exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq c * g(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

3] Big - Theta Notation : Big Theta (θ) notation gives bound for a function f(n) within a constant factor. We write $f(n) = \theta(g(n))$, if there are positive constants no, c_1 and c_2 such that, to the right of n_0 the f(n) always lies between $c_1 * g(n)$ and $c_2 * g(n)$ inclusive.

$$\theta(g(n)) = \{ f(n) : \text{There exist positive constant } c_1, c_2 \text{ and no such that } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0 \}$$

8863

PAGE No.	
DATE	/ /

Q 2 The rate at which running time increases as a function of input is called rate of growth. That is as the amount of data gets bigger, how much more resources does the ~~algo rithm~~ algorithm ~~require~~ requires

commonly used rates of growth

Time Complexity	Name
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	linear logarithmic
n^2	quadratic
n^3	cubic
2^n	exponential