# ASSIGNMENT

Submitted To,

Sunu Mary Abraham

Submitted By,

Name: Rena Elza Viju

Roll no : 48

Course : MCA 1st

1. Compute the time complexity of following snippet of codes

a)
```
int i, j, k=0;
for (i=n/2; i<=2; i++)          - n
{
    for (j=2; j<=n; j=j*2) {    - log₂ n
        k= k+n/2
    }
}
```
∴ time complexity = $\underline{O}$ (n log n)

b)
```
int a=0; i=N;
while (i>0) {
    a+=i;
    i/=2;          - log₂ n
}
```
∴ time complexity = $\underline{O}$ (log n)

c)
```
void fun (int n)
{
    for (int i=0; i<n; i++)             - n
        for (int j=n; j>0; j=j/2)       - log₂ n
            printf ("*");
}
```
∴ time complexity - $\underline{O}$ (n log n)

2. Solve the recurrence relation

$T(n) = T(n/2) + \log_2 n$ where $T(1) = 1$

**Solution**

$T(n) = T(n/2) + \log_2 n$

$T(n/2) = T(n/4) + \log(n/2)$

$T(n/4) = T(n/8) + \log(n/4)$

$\therefore T(n) = T(n/2k) + \Sigma \log_2 (n/2i)$

$\log_2 (n/2i) = \log n - \log_2 2i$

$\log n - i$

$T(1) = T(n/2k) \Rightarrow$

$n/2k = 1$

$n = 2^k$

$\log n = k$

$\therefore T(n) = T(1) + \Sigma (\log n - i)$

$= T(1) + \Sigma \log n - \Sigma i$

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

Since $k = \log n \Rightarrow$

$T(n) = 1 + \log n \cdot \log n - \frac{\log n(\log n - 1)}{2}$

$\therefore O(\log^2 n)$

3. Algorithm A performs $10n^2$ basic operations, algorithm B performs 300 log n basic operations. for what value of n does algorithm B start to show its better performance.

<u>Solution</u>

Algorithm A $\Rightarrow$

$$T_A(n) = 10n^2$$

Algorithm B $\Rightarrow$

$$T_B(n) = 300 \log n$$

at which point does algorithm B starts to show its better performance

ie $\Rightarrow 300 \log n < 10n^2$

deuiding through out by 10;

$$30 \log n < n^2$$

| $n$ | $A \Rightarrow n^2$ | $B \Rightarrow 30 \log n$ |
|---|---|---|
| $2^2$ | 16 | 60 |
| $2^3$ | 64 | 90 |
| $2^4$ | 256 | 120 |

$\therefore$ for all $n \geq 24$ algorithm B starts to show bettee performance than A.

## QUESTION -4

Write programs in any programming language to implement the following sorting algorithms:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort

In each implementation:

• Maintain a counter variable to record the number of iterations executed by:
  o the outer loop, and
  o the inner loop.
• Using this counter, determine and compare the number of iterations required for:
  o Best Case
  o Average Case
  o Worst Case

For each sorting algorithm, you must:

1. Clearly specify the input used for best, average, and worst cases.
2. Report the iteration counts for each case.
3. Analyze and compare the time complexity behavior based on the observed counts.
4. Present your results in a table format and give a brief conclusion.

## SOLUTION

Array Size (n) = 5

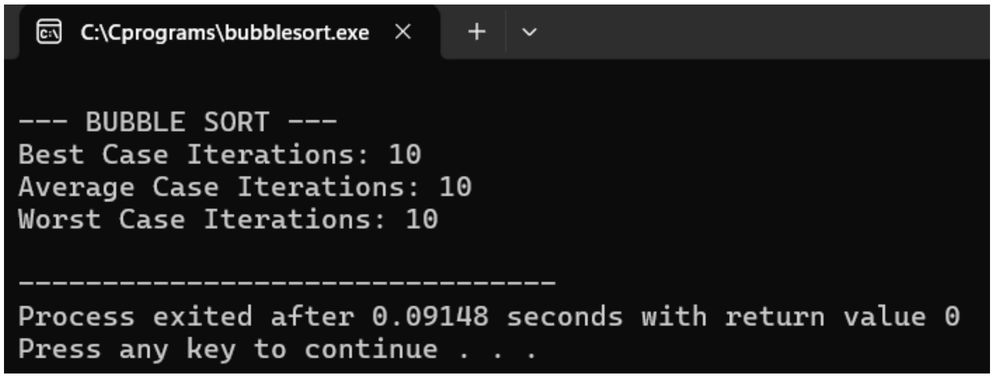
### 1. Bubble Sort

**C Program:**

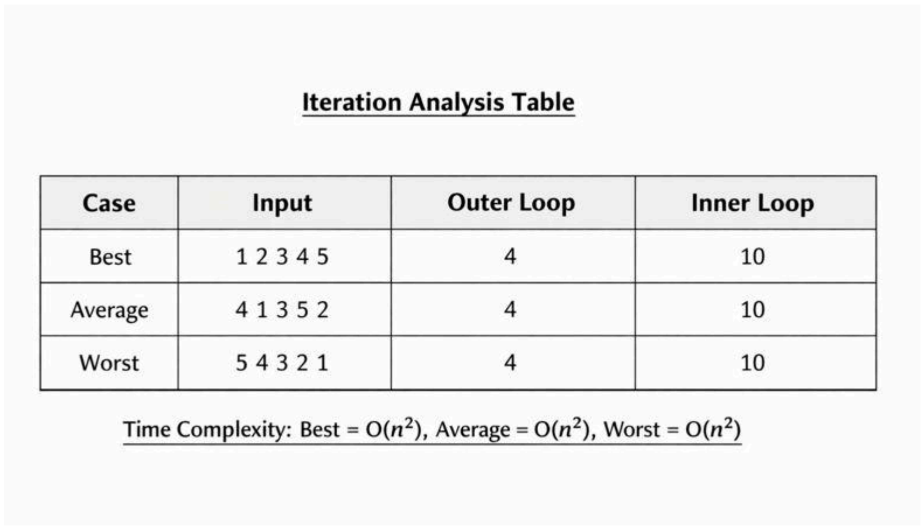
```
#include <stdio.h>

 #define SIZE 5
 int count;

 void copyArray(int dest[], int src[]) {
         for(int i = 0; i < SIZE; i++)
         dest[i] = src[i];
 }

 void bubbleSort(int arr[]) {
         count = 0;
         for(int i = 0; i < SIZE - 1; i++) {
         for(int j = 0; j < SIZE - i - 1; j++) {
         count++;
         if(arr[j] > arr[j + 1]) {
                 int temp = arr[j];
                 arr[j] = arr[j + 1];
                 arr[j + 1] = temp;
         }
         }
         }
 }

 int main() {
         int best[SIZE] = {1, 2, 3, 4, 5};
         int avg[SIZE]  = {4, 1, 3, 5, 2};
         int worst[SIZE] = {5, 4, 3, 2, 1};
         int arr[SIZE];

         copyArray(arr, best);
         bubbleSort(arr);
         printf("Best Case Iterations: %d\n", count);

         copyArray(arr, avg);
         bubbleSort(arr);
         printf("Average Case Iterations: %d\n", count);
```

```
        copyArray(arr, worst);
        bubbleSort(arr);
        printf("Worst Case Iterations: %d\n", count);

        return 0;
}
```

Output:



```
C:\Cprograms\bubblesort.exe

--- BUBBLE SORT ---
Best Case Iterations: 10
Average Case Iterations: 10
Worst Case Iterations: 10

--------------------------------
Process exited after 0.09148 seconds with return value 0
Press any key to continue . . .
```



**Iteration Analysis Table**

| Case | Input | Outer Loop | Inner Loop |
|---|---|---|---|
| Best | 1 2 3 4 5 | 4 | 10 |
| Average | 4 1 3 5 2 | 4 | 10 |
| Worst | 5 4 3 2 1 | 4 | 10 |

Time Complexity: Best = O($n^2$), Average = O($n^2$), Worst = O($n^2$)

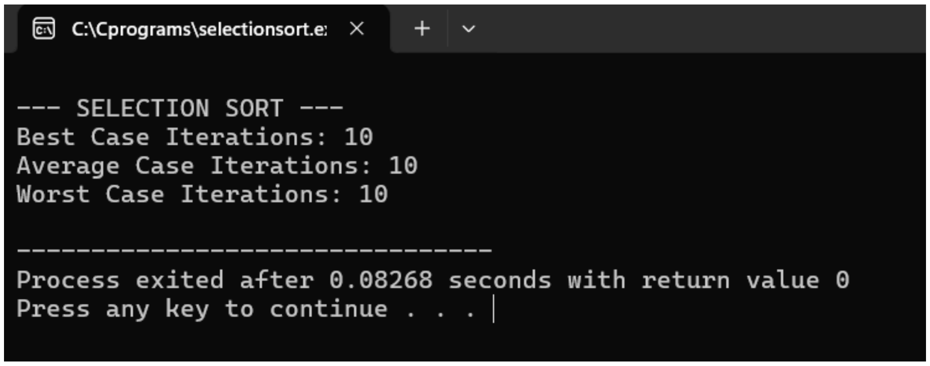
## 2. Selection Sort

**C Program:**

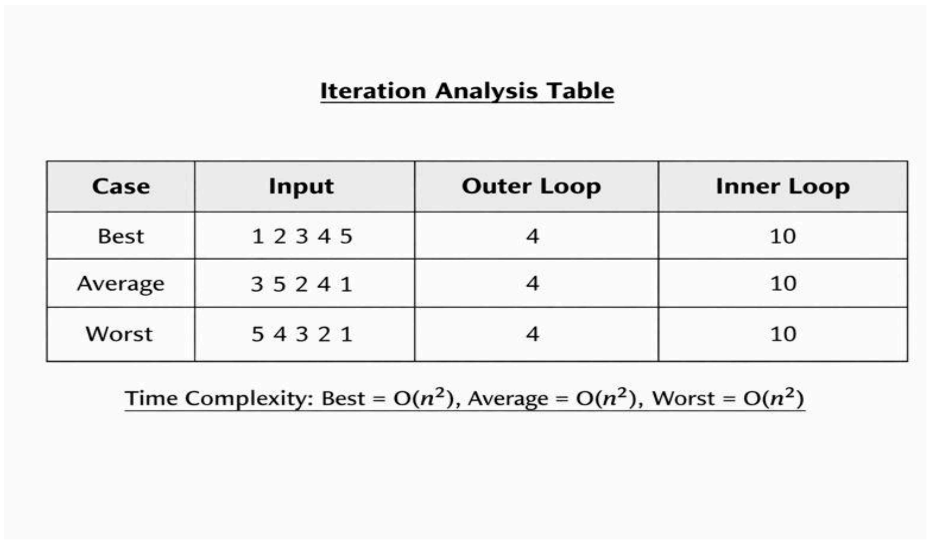
```
#include <stdio.h>

 #define SIZE 5
 int count;

 void copyArray(int dest[], int src[]) {
          for(int i = 0; i < SIZE; i++)
          dest[i] = src[i];
 }

 void selectionSort(int arr[]) {
          count = 0;
          for(int i = 0; i < SIZE - 1; i++) {
          int min = i;
          for(int j = i + 1; j < SIZE; j++) {
          count++;
          if(arr[j] < arr[min])
                   min = j;
          }
          int temp = arr[min];
          arr[min] = arr[i];
          arr[i] = temp;
          }
 }

 int main() {
          int best[SIZE] = {1, 2, 3, 4, 5};
          int avg[SIZE]  = {3, 5, 2, 4, 1};
          int worst[SIZE] = {5, 4, 3, 2, 1};
          int arr[SIZE];

          copyArray(arr, best);
          selectionSort(arr);
          printf("Best Case Iterations: %d\n", count);

          copyArray(arr, avg);
          selectionSort(arr);
          printf("Average Case Iterations: %d\n", count);

          copyArray(arr, worst);
          selectionSort(arr);
          printf("Worst Case Iterations: %d\n", count);
```

```
        return 0;
}
```

**Output:**



```
--- SELECTION SORT ---
Best Case Iterations: 10
Average Case Iterations: 10
Worst Case Iterations: 10

--------------------------------
Process exited after 0.08268 seconds with return value 0
Press any key to continue . . .
```



**Iteration Analysis Table**

| Case | Input | Outer Loop | Inner Loop |
|---|---|---|---|
| Best | 1 2 3 4 5 | 4 | 10 |
| Average | 3 5 2 4 1 | 4 | 10 |
| Worst | 5 4 3 2 1 | 4 | 10 |

Time Complexity: Best = O($n^2$), Average = O($n^2$), Worst = O($n^2$)

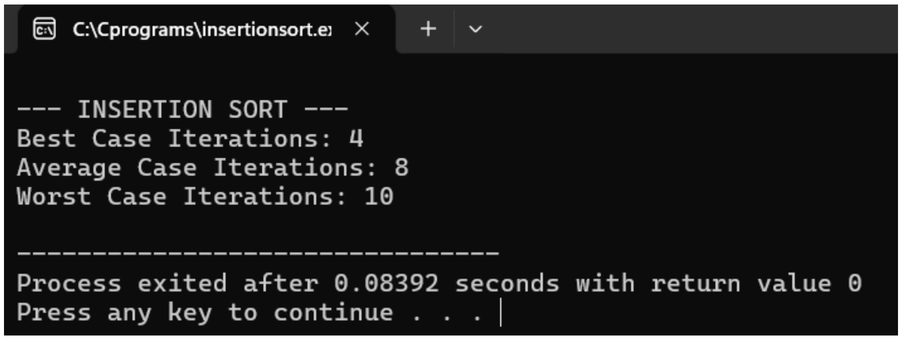
## 3. Insertion Sort

**C Program:**

```
#include <stdio.h>

 #define SIZE 5
 int count;

 void copyArray(int dest[], int src[]) {
          for(int i = 0; i < SIZE; i++)
          dest[i] = src[i];
 }

 void insertionSort(int arr[]) {
          count = 0;
          for(int i = 1; i < SIZE; i++) {
          int key = arr[i];
          int j = i - 1;
          while(j >= 0) {
          count++;
          if(arr[j] > key) {
                   arr[j + 1] = arr[j];
                   j--;
          } else {
                   break;
          }
          }
          arr[j + 1] = key;
    }
 }

 int main() {
          int best[SIZE] = {1, 2, 3, 4, 5};
          int avg[SIZE]  = {4, 2, 5, 1, 3};
          int worst[SIZE] = {5, 4, 3, 2, 1};
          int arr[SIZE];

          copyArray(arr, best);
          insertionSort(arr);
          printf("Best Case Iterations: %d\n", count);

          copyArray(arr, avg);
          insertionSort(arr);
          printf("Average Case Iterations: %d\n", count);
```

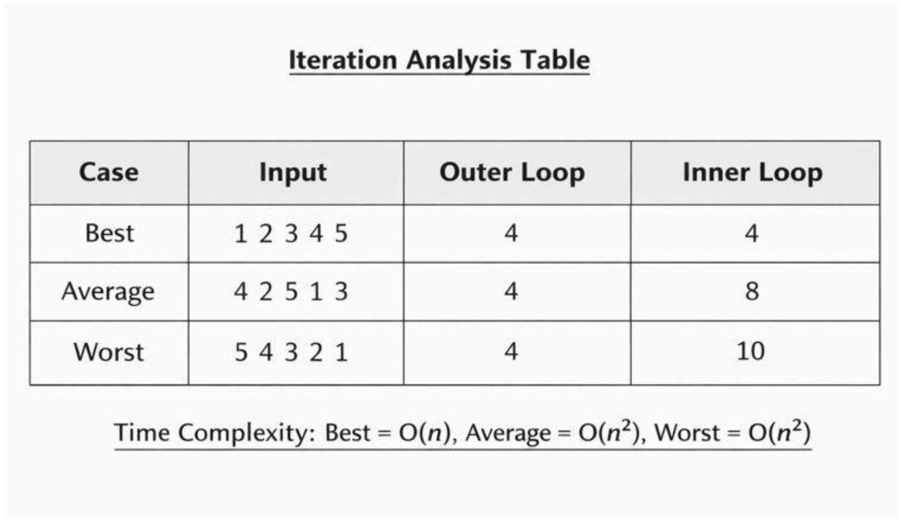
```
        copyArray(arr, worst);
        insertionSort(arr);
        printf("Worst Case Iterations: %d\n", count);

        return 0;
}
```

Output:



C:\Cprograms\insertionsort.e

```
--- INSERTION SORT ---
Best Case Iterations: 4
Average Case Iterations: 8
Worst Case Iterations: 10

--------------------------------
Process exited after 0.08392 seconds with return value 0
Press any key to continue . . .
```



**Iteration Analysis Table**

| Case | Input | Outer Loop | Inner Loop |
|---|---|---|---|
| Best | 1 2 3 4 5 | 4 | 4 |
| Average | 4 2 5 1 3 | 4 | 8 |
| Worst | 5 4 3 2 1 | 4 | 10 |

Time Complexity: Best = O($n$), Average = O($n^2$), Worst = O($n^2$)

## Conclusion

From the iteration analysis, it is observed that Bubble Sort and Selection Sort perform the same number of comparisons in the best, average, and worst cases, regardless of the input order. This makes them inefficient for large datasets, as their time complexity remains O($n^2$) in all scenarios. Insertion Sort, on the other hand, shows a significant improvement in the best case, requiring only O(n) time when the input array is already sorted or nearly sorted. Although its average and worst-case complexities are O($n^2$), its adaptive behavior makes it more efficient than the other two algorithms for small or partially sorted datasets.