Lab 1 :

Write and Execute the following also analyze the time complexity
1. Sort a list of N integers using
(a) Selection sort

Open | *sorting_program.c ~/Documents/ADSANEW_LAB | Save

```c
#include <stdio.h>
int main() {
    int n, i, j, position, swap, swapc = 0, cmpc = 0;

    printf("Enter the number of elements:");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    for (i = 0; i < (n - 1); i++) {
        position = i;
        for (j = i + 1; j < n; j++) {
            cmpc++;
            if (arr[position] > arr[j]) {
                position = j;
            }
        }
        if (position != i) {
            swap = arr[i];
            arr[i] = arr[position];
            arr[position] = swap;
            swapc++;
        }
    }

    printf("Comparison count : %d\n", cmpc);
    printf("Swap count : %d\n", swapc);

    for (i = 0; i < n; i++) {
        printf("%d\t", arr[i]);
    }

    return 0;
}
```

rohithsaidatta@rohithsaidatta-VirtualBox: ~/Documents/ADSANEW_LAB

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ gcc -o sorting sorting_program.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ ./sorting
Enter the number of elements:5
Enter the elements:
9 3 7 1 5
Comparison count : 10
Swap count : 3
1       3       5       7       9       rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$
```

Analysis:

- For each position, find the smallest number in the remaining part.
- If a smaller number is found, swap it with the current position.
- Display comparison and swap counts.
- Show the sorted numbers.

Time Complexity

1. Best Case: O(n^2) - Although no swaps are needed, comparisons still occur.
2. Average Case: O(n^2) - Roughly (n^2)/2 comparisons and n swaps on average.
3. Worst Case: O(n^2) - Maximum comparisons and swaps when the array is reverse sorted.

(b) Bubble sort

*bubblesort_program.c
~/Documents/ADSANEW_LAB

```c
#include <stdio.h>

void BubbleSort(int arr[], int n, int *countercomp, int *counterswap) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            (*countercomp)++;
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                (*counterswap)++;
            }
        }
    }
}

int main() {
    int arr[100], n, countercomp = 0, counterswap = 0;
    printf("Enter size of array\n");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Original array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t", arr[i]);
    }
    BubbleSort(arr, n, &countercomp, &counterswap);
    printf("\nSorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t", arr[i]);
    }
    printf("\nNumber of comparisons: %d\n", countercomp);
    printf("Number of swaps: %d\n", counterswap);

    return 0;
}
```

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ gcc -o bsort bubblesort_program.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ ./bsort
Enter size of array
5
Enter array elements:
4 2 7 1 5
Original array:
4       2       7       1       5
Sorted array:
1       2       4       5       7
Number of comparisons: 10
Number of swaps: 5
```

Analysis

- The program uses Bubble Sort to arrange input numbers.
- Bubble Sort has a slow time complexity of O(n^2) for sorting.
- It counts comparisons and swaps during sorting.
- For already sorted input, Bubble Sort is best with O(n) time.
- For reverse order input, it's worst with O(n^2) time.
- On average, it's still O(n^2) due to multiple comparisons and swaps.

Time Complexity:
Bubble Sort has an average and worst-case time complexity of O(n^2), making it inefficient for large datasets.

1. Best Case:
   Best-case time complexity is O(n) when the array is already sorted. Minimal comparisons and no swaps are needed.

2. Worst Case:
   Worst-case time complexity is O(n^2) when the array is in reverse order, leading to maximum comparisons and swaps.

3. Average Case:
   Average-case time complexity is also O(n^2), as Bubble Sort typically performs a similar number of comparisons and swaps as the worst case.

2. Binary search technique over a list of integers.

Open | *binary_search.c ~/Documents/ADSANEW_LAB | Save

```c
#include <stdio.h>

int binarySearch(int a[], int beg, int end, int val, int *counter) {
    int mid;
    while (beg <= end) {
        (*counter)++;
        mid = beg + (end - beg) / 2;
        if (a[mid] == val) {
            return mid + 1;
        } else if (a[mid] < val) {
            beg = mid + 1;
        } else {
            end = mid - 1;
        }
    }
    return -1;
}

int main() {
    int val, nu;
    printf("Enter the number of elements: ");
    scanf("%d", &nu);
    int a[nu];
    printf("Enter the elements:\n");
    for (int i = 0; i < nu; i++) {
        scanf("%d", &a[i]);
    }
    printf("Enter the element to be searched: ");
    scanf("%d", &val);
    int counter = 0;
    int res = binarySearch(a, 0, nu - 1, val, &counter);
    if (res == -1)
        printf("\nElement is not present in the array");
    else
        printf("\nElement is present at position %d in the array", res);
    printf("\nNumber of comparisons: %d", counter);
    return 0;
}
```

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ gcc -o bsearch binary_search.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ ./bsearch
Enter the number of elements: 5
Enter the elements:
10 20 30 40 50
Enter the element to be searched: 30

Element is present at position 3 in the array
Number of comparisons: 1rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$
```

Analysis

- Binary search algorithm is applied to locate a target value in an array.
- Recursion is used to repeatedly narrow down the search range.
- Comparison occurs between the target and the middle element of the range.
- Once found or exhausted, it reports if the target is present.
- The program displays the number of comparisons made during the search.
- Overall, the code efficiently finds a target in an array and reports comparisons.

Time Complexity:

Binary search takes a time proportional to the logarithm of the number of elements in the array. Written as O(log n), where 'n' is the number of elements.

1. Best Case:
   When the target is right in the middle, you find it in just one step.
   The best case time is constant and is O(1).

2. Worst Case:
   If the target is at an end or absent, you keep splitting the array until there's only one element left.
   The worst-case time is still pretty good, it's O(log n).

3. Average Case:
   On average, binary search takes about $\log_2(n)$ steps to find the target.
   The average case time is also O(log n).

3. Stack operations

(i) Stack-empty(S)

(ii) Push(S, x)

(iii) Pop(S)

*stack.c
~/Documents/ADSANEW_LAB

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100

struct Stack {
    int data[MAX_SIZE];
    int top;
};

void initialize(struct Stack *stack) {
    stack->top = -1;
}

bool isEmpty(struct Stack *stack) {
    return (stack->top == -1);
}

bool isFull(struct Stack *stack) {
    return (stack->top == MAX_SIZE - 1);
}

void push(struct Stack *stack, int x, int *countpush) {
    if (isFull(stack)) {
        printf("Stack is full, cannot push %d\n", x);
        return;
    }
    stack->data[++stack->top] = x;
    (*countpush)++;
    printf("Element %d pushed into stack\n", x);
}

int pop(struct Stack *stack, int *countpop) {
    if (isEmpty(stack)) {
        printf("Stack is empty, cannot pop\n");
        return -1;
    }
    int popped = stack->data[stack->top--];
    (*countpop)++;
```

```c
    printf("Popped element: %d\n", popped);
    return popped;
}

int main() {
    struct Stack stack;
    initialize(&stack);
    int countpush = 0, countpop = 0;

    int choice, element;

    do {
        printf("\nStack Operations:\n");
        printf("1. Check if Stack is Empty\n");
        printf("2. Push Element\n");
        printf("3. Pop Element\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Stack is %s\n", isEmpty(&stack) ? "empty" : "not empty");
                break;
            case 2:
                printf("Enter an element to push: ");
                scanf("%d", &element);
                push(&stack, element, &countpush);
                break;
            case 3:
                pop(&stack, &countpop);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 4);

    printf("Total number of pushes: %d\n", countpush);
    printf("Total number of pops: %d\n", countpop);
    return 0;
}
```

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ gcc -o stk stack.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ ./stk

Stack Operations:
1. Check if Stack is Empty
2. Push Element
3. Pop Element
4. Exit
Enter your choice: 1
Stack is empty
```

```
Stack Operations:
1. Check if Stack is Empty
2. Push Element
3. Pop Element
4. Exit
Enter your choice: 2
Enter an element to push: 10
Element 10 pushed into stack

Stack Operations:
1. Check if Stack is Empty
2. Push Element
3. Pop Element
4. Exit
Enter your choice: 2
Enter an element to push: 20
Element 20 pushed into stack

Stack Operations:
1. Check if Stack is Empty
2. Push Element
3. Pop Element
4. Exit
Enter your choice: 3
Popped element: 20

Stack Operations:
1. Check if Stack is Empty
2. Push Element
3. Pop Element
4. Exit
Enter your choice: 4
Exiting...
Total number of pushes: 2
Total number of pops: 1
```

Analysis

- Stack is implemented using an array.
- Struct named Stack holds array and top index.
- Initialization sets top to -1.
- isEmpty checks if top is -1.
- push adds element if not full.
- pop removes element if not empty.
- User interface in main function.
- Tracks push and pop counts.
- Push and pop counters passed as pointers.

Time complexity

1. Push Operation:
   - Best Case: O(1)
     When the stack is not full, the push operation involves a constant time array assignment and incrementing the top pointer.
   - Worst Case: O(1)
     Even in the worst case (stack being just one element away from full), the push operation is still a constant time operation.
   - Average Case: O(1)
     The average time complexity remains constant since each push operation takes a constant amount of time.

2. Pop Operation:
   - Best Case: O(1)
     When the stack is not empty, the pop operation involves a constant time array access and decrementing the top pointer.
   - Worst Case: O(1)
     Similar to the push operation, the worst case time complexity for popping is also constant.
   - Average Case: O(1)
     The average time complexity remains constant because each pop operation takes constant time regardless of the stack's size.

3. isEmpty Operation:

   - Best Case: O(1)
     Checking if the stack is empty involves a single comparison (constant time) of the top pointer.
   - Worst Case: O(1)
     Similarly, the worst and average cases involve a single constant time comparison.

4. Queue Q also analyze the time complexity.

(i) Enqueue(Q, x)

(ii) Dequeue(Q)

Open | queue.c ~/Documents/ADSANEW_LAB | Save

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100

struct Queue {
    int data[MAX_SIZE];
    int front, rear;
};

void initialize(struct Queue *q) {
    q->front = q->rear = -1;
}

bool isEmpty(struct Queue *q) {
    return (q->front == -1);
}

bool isFull(struct Queue *q) {
    return ((q->rear + 1) % MAX_SIZE == q->front);
}

void enqueue(struct Queue *q, int x, int *countin) {
    if (isFull(q)) {
        printf("Queue is full, cannot enqueue %d\n", x);
        return;
    }
    if (isEmpty(q)) {
        q->front = q->rear = 0;
    } else {
        q->rear = (q->rear + 1) % MAX_SIZE;
    }
    q->data[q->rear] = x;
    (*countin)++;
    printf("Enqueued element %d\n", x);
}

int dequeue(struct Queue *q, int *countout) {
    if (isEmpty(q)) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    }
    int x = q->data[q->front];
    if (q->front == q->rear) {
        initialize(q);
    } else {
        q->front = (q->front + 1) % MAX_SIZE;
    }
    (*countout)++;
    printf("Dequeued element: %d\n", x);
    return x;
}

int main() {
    struct Queue q;
    int countin = 0, countout = 0;
    initialize(&q);

    int choice, element;
    while (1) {
        printf("Choose an operation:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Quit\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to enqueue: ");
                scanf("%d", &element);
                enqueue(&q, element, &countin);
                printf("Number of insertions for enqueues: %d\n", countin);
                break;
            case 2:
                dequeue(&q, &countout);
                printf("Number of deletions for dequeues: %d\n", countout);
                break;
            case 3:
                printf("Quitting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Try again.\n");
        }
    }

    return 0;
}
```

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ gcc -o que queue.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB$ ./que
Choose an operation:
1. Enqueue
2. Dequeue
3. Quit
1
Enter element to enqueue: 10
Enqueued element 10
Number of insertions for enqueues: 1
Choose an operation:
1. Enqueue
2. Dequeue
3. Quit
1
Enter element to enqueue: 20
Enqueued element 20
Number of insertions for enqueues: 2
Choose an operation:
1. Enqueue
2. Dequeue
3. Quit
2
Dequeued element: 10
Number of deletions for dequeues: 1
Choose an operation:
1. Enqueue
2. Dequeue
3. Quit
3
Quitting the program.
```

Analysis
(i) Enqueue operation:
The enqueue operation involves inserting an element at the rear of the queue. Here's how the time complexity breaks down:

- Checking if the queue is full (isFull): O(1)
- Inserting the element into the queue: O(1)
- Updating the rear pointer: O(1)
- Incrementing the countin variable: O(1)
- Printing a message: O(1)
- Overall, the enqueue operation has a constant time complexity of O(1).

1. Enqueue operation:

Best/Average/Worst-case: Always O(1)
Enqueueing is fast and constant-time because it involves simple array updates and checks.

(ii) Dequeue operation:
The dequeue operation involves removing an element from the front of the queue. Here's how the time complexity breaks down:

- Checking if the queue is empty (isEmpty): O(1)
- Retrieving the element from the queue: O(1)
- Updating the front pointer: O(1)
- Incrementing the count out variable: O(1)
- Printing a message: O(1)
- Overall, Dequeue operation also has a constant time complexity of O(1).

2. Dequeue operation:

Best/Average/Worst-case: Always O(1)
Dequeueing is also fast and constant-time due to straightforward array updates and checks.