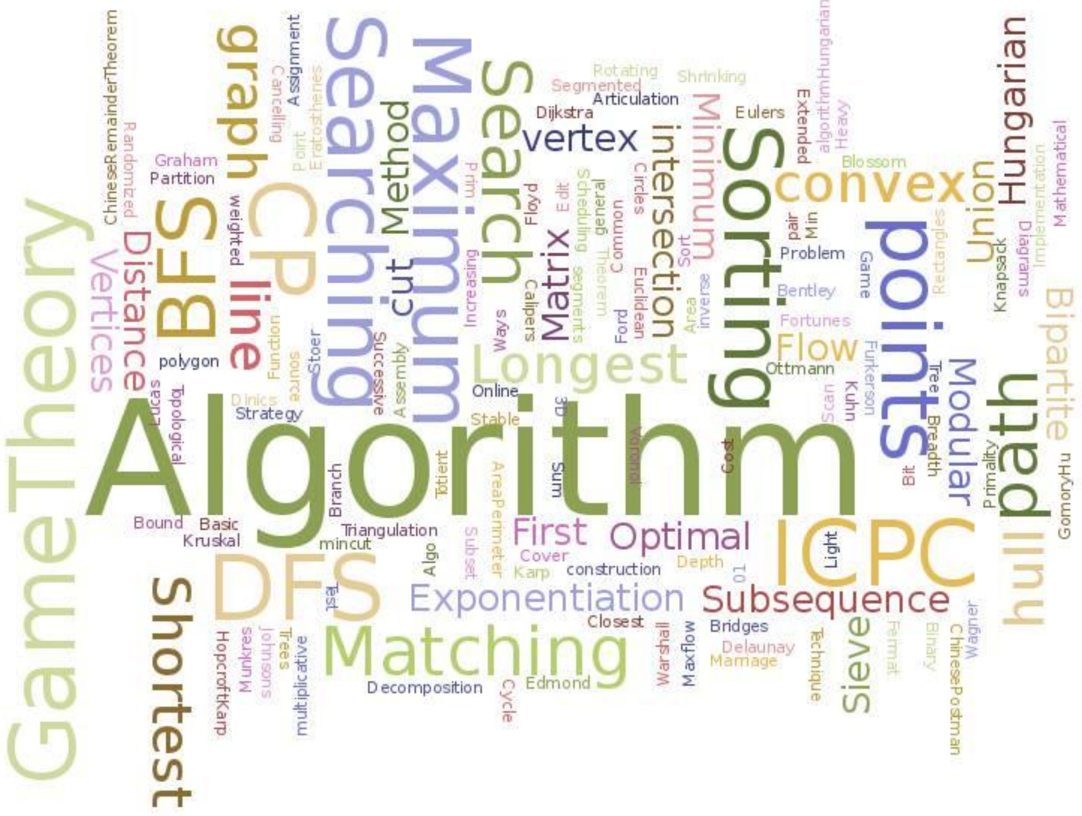
AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY UNDER SECTION 3 OF UGC ACT, 1956

SCHOOL OF COMPUTING

Design and Analysis of Algorithms Lab 4 Report

(Course Code: 23CSE211)



Name: NAGARAMPALLI SARVAN KUMAR

Roll Number: CH.SC.U4CSE24130

Branch/ Section: Computer Science and Engineering / CSE – B

College: Amrita Vishwa Vidyapeetham Chennai Campus

Academic Year: 2025 – 2026

Programs and Their Space Complexities

Date: 27-11-2025

1. Write a program to find the sum of first *n* natural numbers using a function.

2. Write a program to find the sum of squares of first *n* natural numbers.

3. Write a program to find the sum of cubes of first *n* natural numbers.

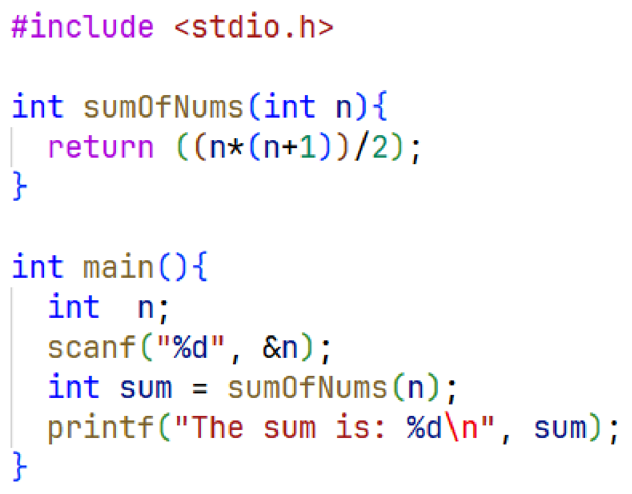
4. Write a program to find factorial of a number using a recursive function.

5. Write a program to find the transpose of a 3×3 matrix.

6. Write a program to print the Fibonacci series using recursion.

Solutions:

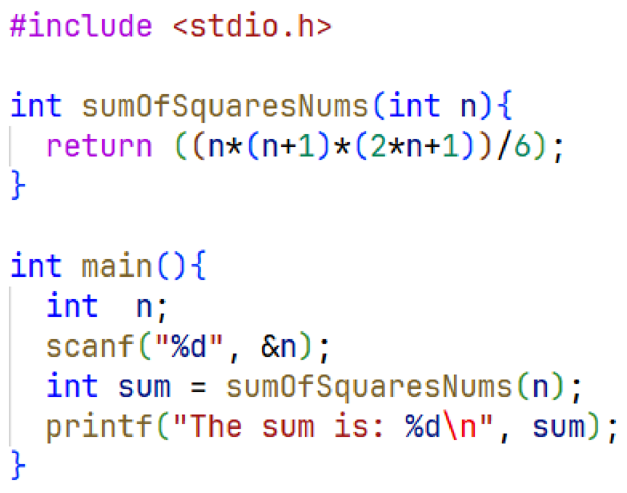
Write a program to find the sum of first *n* natural numbers using a function.



```c
#include <stdio.h>

int sumOfNums(int n){
  return ((n*(n+1))/2);
}

int main(){
  int  n;
  scanf("%d", &n);
  int sum = sumOfNums(n);
  printf("The sum is: %d\n", sum);
}
```

Justification: Uses the formula n(n+1)/2, which calculates the total directly.
Space Complexity: O(1)

Write a program to find the sum of squares of first *n* natural numbers.

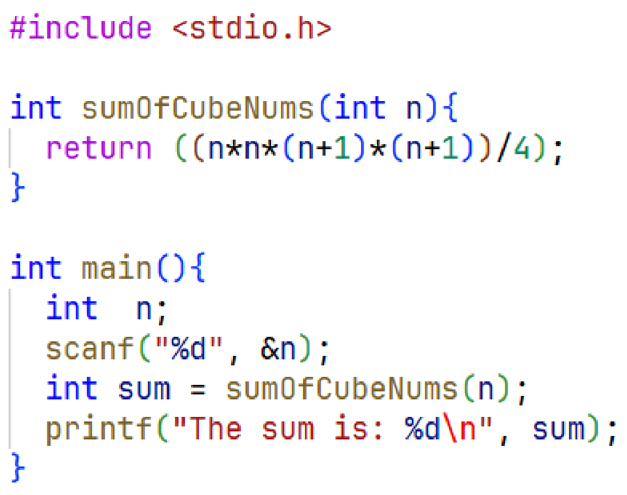


```c
#include <stdio.h>

int sumOfSquaresNums(int n){
  return ((n*(n+1)*(2*n+1))/6);
}

int main(){
  int  n;
  scanf("%d", &n);
  int sum = sumOfSquaresNums(n);
  printf("The sum is: %d\n", sum);
}
```

Justification: Uses the formula n(n+1)(2n+1)/6, computes the sum directly.

Space Complexity: O(1)

Write a program to find the sum of cubes of first *n* natural numbers.

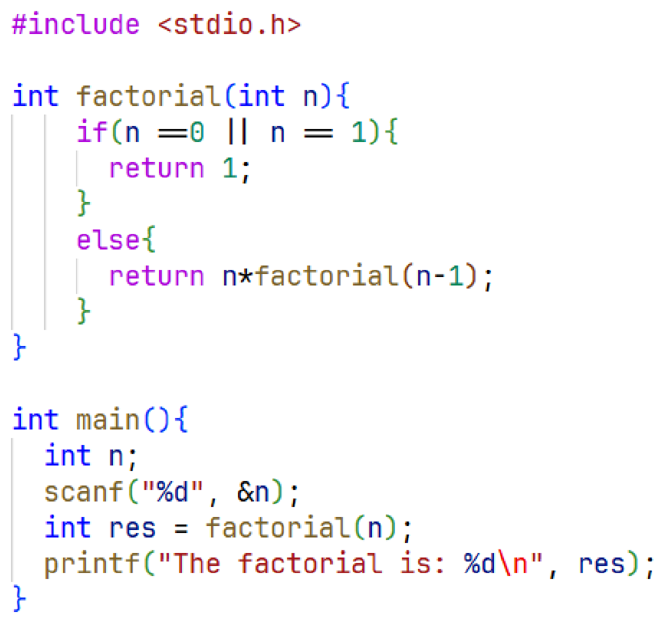


```c
#include <stdio.h>

int sumOfCubeNums(int n){
  return ((n*n*(n+1)*(n+1))/4);
}

int main(){
  int  n;
  scanf("%d", &n);
  int sum = sumOfCubeNums(n);
  printf("The sum is: %d\n", sum);
}
```

Justification: Uses the formula $[(n(n+1))/2]^2$, which calculates the total directly.

Space Complexity: O(1)

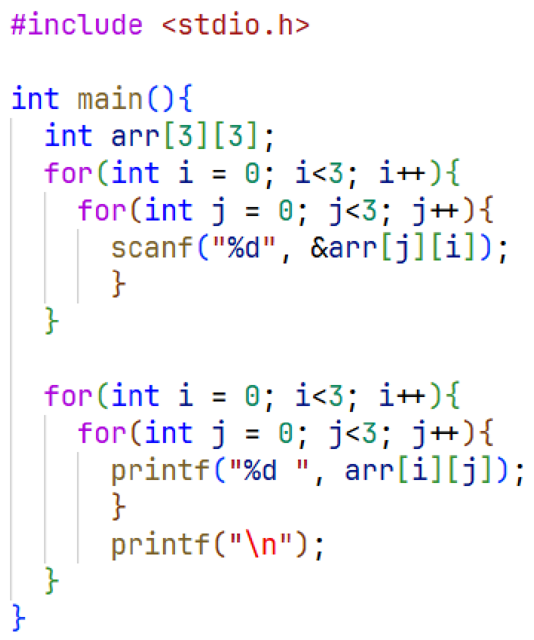
Write a program to find factorial of a number using a recursive function.



```c
#include <stdio.h>

int factorial(int n){
    if(n ==0 || n == 1){
        return 1;
    }
    else{
        return n*factorial(n-1);
    }
}

int main(){
    int n;
    scanf("%d", &n);
    int res = factorial(n);
    printf("The factorial is: %d\n", res);
}
```

Justification: Uses recursive calls that reduce n step-by-step until the base case is reached.
Space Complexity: O(n)

Write a program to find the transpose of a 3×3 matrix.

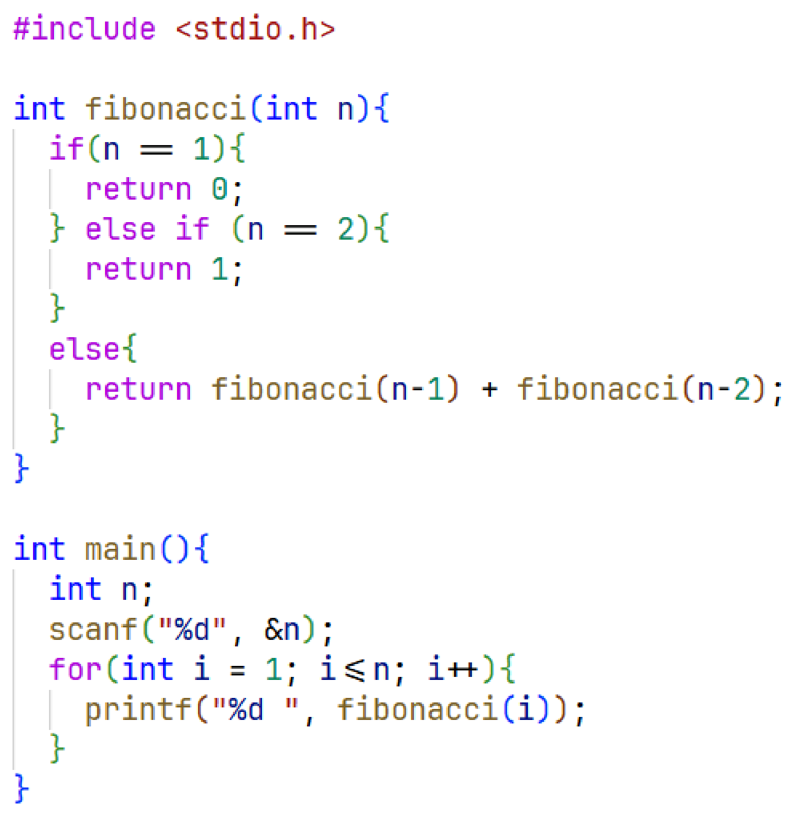


```c
#include <stdio.h>

int main(){
    int arr[3][3];
    for(int i = 0; i<3; i++){
        for(int j = 0; j<3; j++){
            scanf("%d", &arr[j][i]);
        }
    }

    for(int i = 0; i<3; i++){
        for(int j = 0; j<3; j++){
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
```

Justification: Uses a fixed 3×3 array and stores each element directly in its transposed position during input.

Space Complexity: O(1)

Write a program to print the Fibonacci series using recursion.



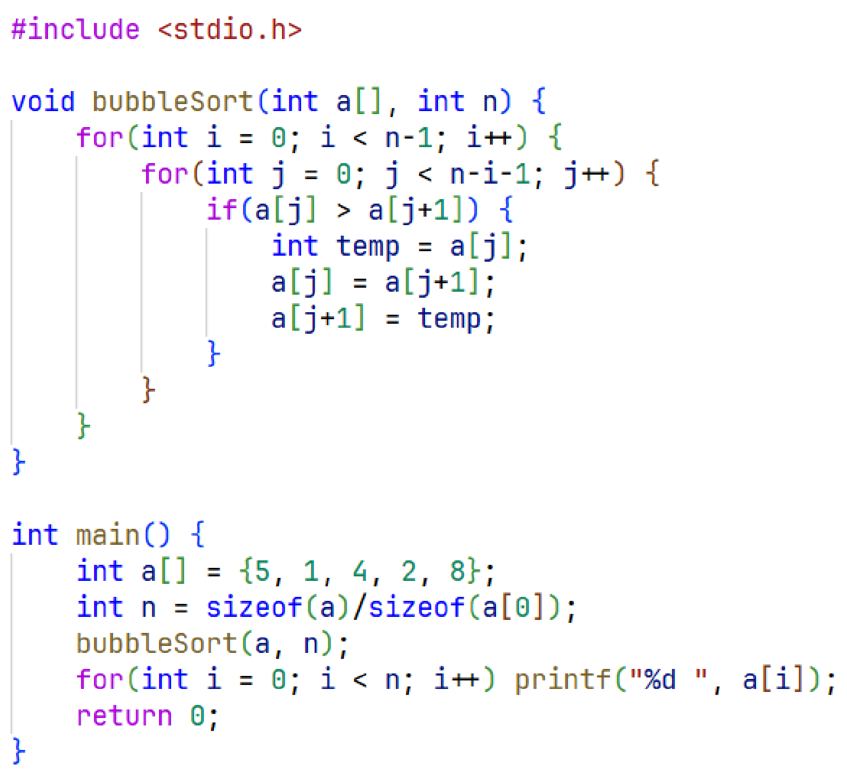
```c
#include <stdio.h>

int fibonacci(int n){
    if(n == 1){
        return 0;
    } else if (n == 2){
        return 1;
    }
    else{
        return fibonacci(n-1) + fibonacci(n-2);
    }
}

int main(){
    int n;
    scanf("%d", &n);
    for(int i = 1; i<=n; i++){
        printf("%d ", fibonacci(i));
    }
}
```

Justification: Uses recursive calls that expand until the base cases are reached.
Space Complexity: O(n)

Sorting Techniques Date: 04-12-2025

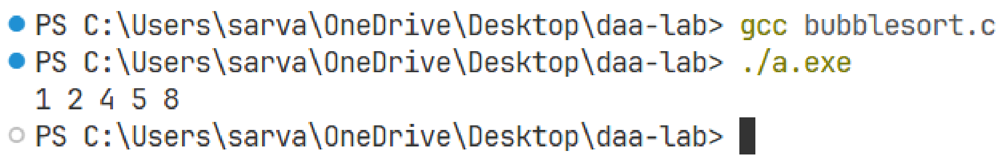
1. Bubble Sort in C
2. Insertion Sort in C
3. Selection Sort in C
4. Bucket Sort in C
5. Heap Sort in C (Max Heap)
6. Heap Sort in C (Min Heap)

Bubble Sort in C



```c
#include <stdio.h>

void bubbleSort(int a[], int n) {
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

int main() {
    int a[] = {5, 1, 4, 2, 8};
    int n = sizeof(a)/sizeof(a[0]);
    bubbleSort(a, n);
    for(int i = 0; i < n; i++) printf("%d ", a[i]);
    return 0;
}
```

Output:



```
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc bubblesort.c
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
  1 2 4 5 8
○ PS C:\Users\sarva\OneDrive\Desktop\daa-lab> 
```

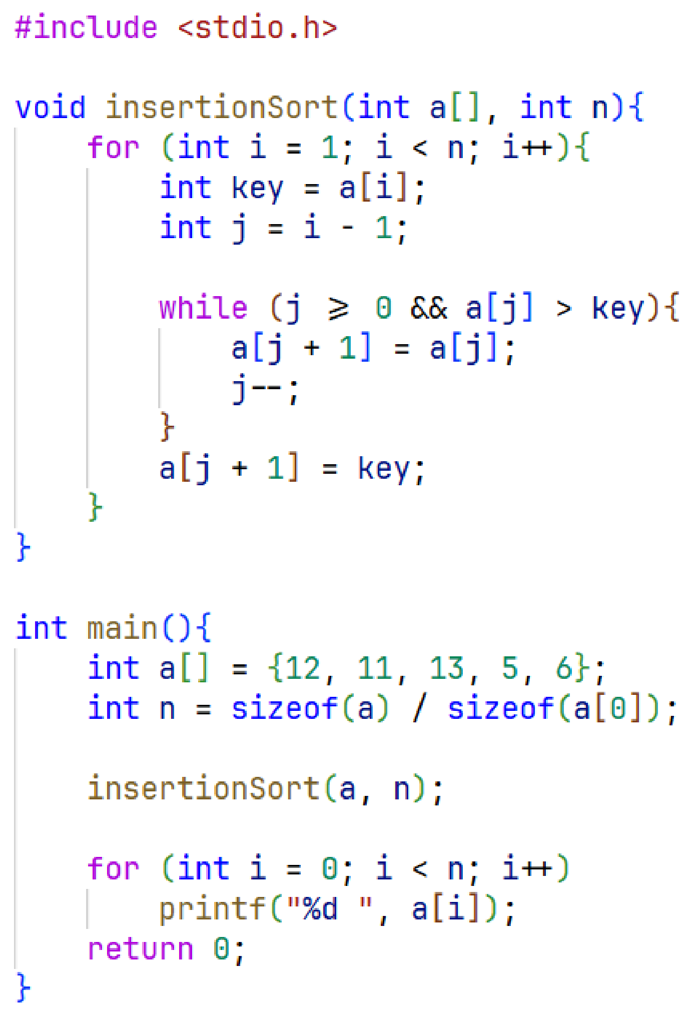
Time Complexity (Worst): O(n^2)

Justification: Every pass compares and swaps almost all elements.

Space Complexity (Worst): O(1)

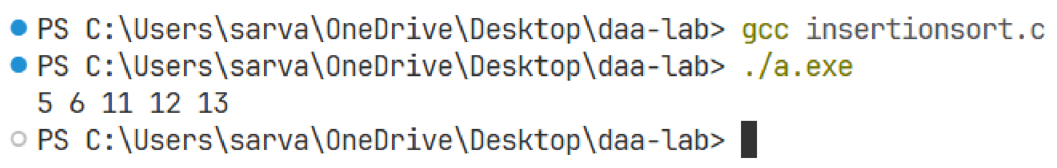
Justification: In-place sorting using only a temporary variable

Insertion Sort in C



```c
#include <stdio.h>

void insertionSort(int a[], int n){
    for (int i = 1; i < n; i++){
        int key = a[i];
        int j = i - 1;

        while (j >= 0 && a[j] > key){
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}

int main(){
    int a[] = {12, 11, 13, 5, 6};
    int n = sizeof(a) / sizeof(a[0]);

    insertionSort(a, n);

    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output:



```
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc insertionsort.c
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
  5 6 11 12 13
○ PS C:\Users\sarva\OneDrive\Desktop\daa-lab> █
```

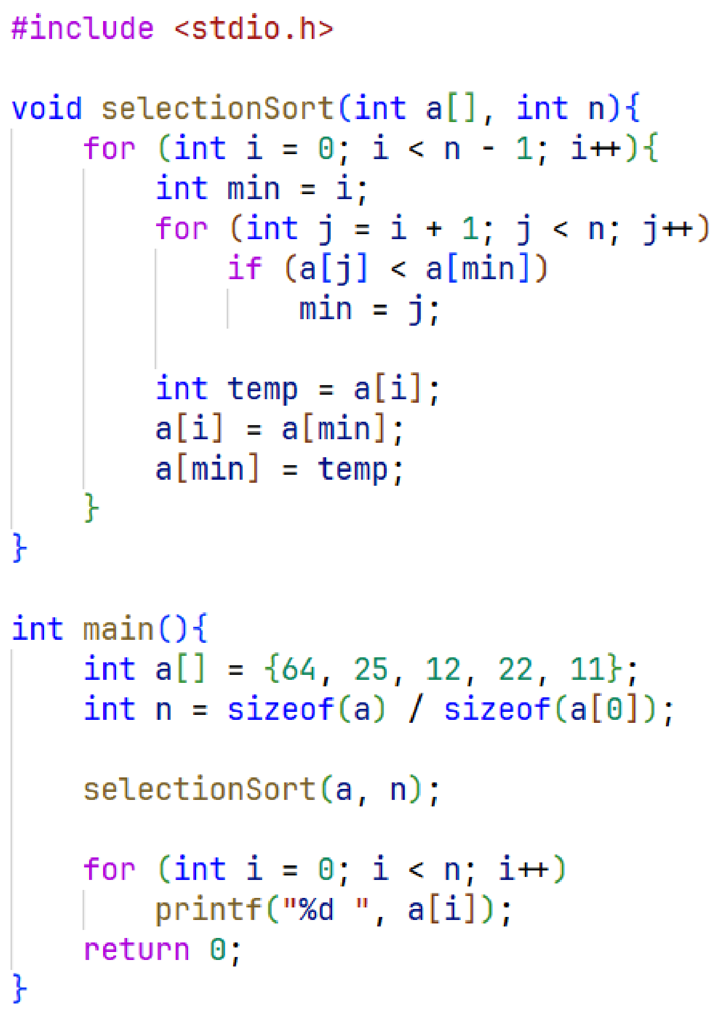
Time Complexity (Worst): O(n^2)

Justification: Each new element may shift through the entire sorted portion.

Space Complexity (Worst): O(1)

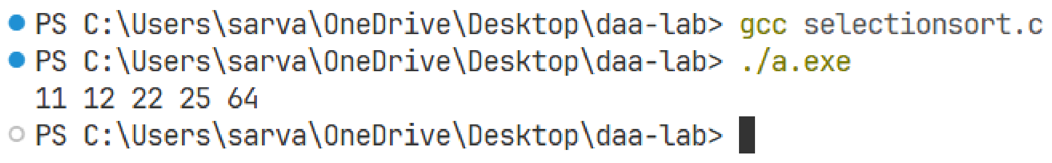
Justification: Uses the same array for sorting.

Selection Sort in C



```c
#include <stdio.h>

void selectionSort(int a[], int n){
    for (int i = 0; i < n - 1; i++){
        int min = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[min])
                min = j;

        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}

int main(){
    int a[] = {64, 25, 12, 22, 11};
    int n = sizeof(a) / sizeof(a[0]);

    selectionSort(a, n);

    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output:



```
PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc selectionsort.c
PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
11 12 22 25 64
PS C:\Users\sarva\OneDrive\Desktop\daa-lab>
```

Time Complexity (Worst): O(n^2)

Justification: Always scans remaining elements to find the minimum.

Space Complexity (Worst): O(1)

Justification: Only one extra variable is used.

Bucket Sort in C

```c
#include <stdio.h>

#define MAX 100

void bucketSort(int a[], int n){
    int bucket[MAX] = {0};

    for (int i = 0; i < n; i++)
        bucket[a[i]]++;

    int k = 0;
    for (int i = 0; i < MAX; i++)
        while (bucket[i]--)
            a[k++] = i;
}

int main(){
    int a[] = {4, 1, 3, 4, 2, 8, 7};
    int n = sizeof(a) / sizeof(a[0]);
    bucketSort(a, n);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output:

```
PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc bucketsort.c
PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
1 2 3 4 4 7 8
PS C:\Users\sarva\OneDrive\Desktop\daa-lab> 
```

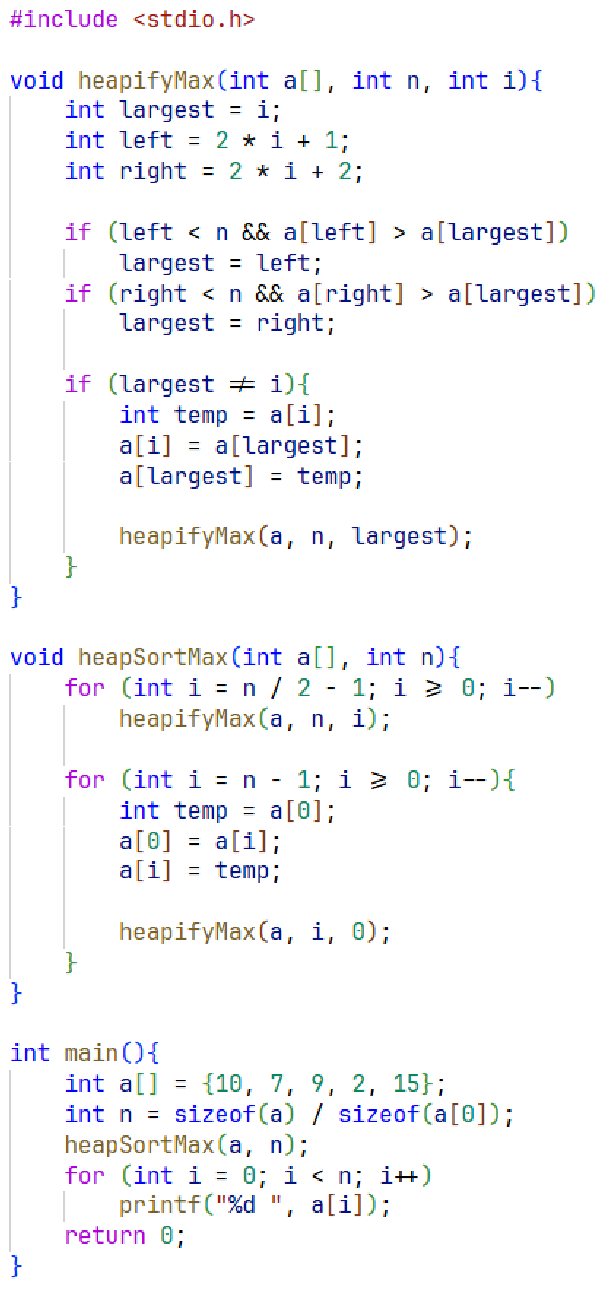
Time Complexity (Worst): O(n^2)

Justification: All elements may fall into a single bucket.

Space Complexity (Worst): O(n + k)

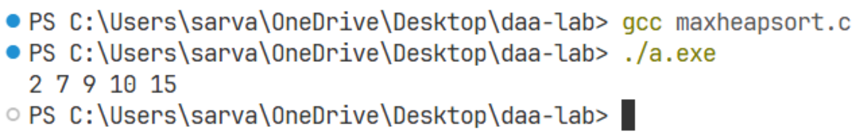
Justification: Requires extra buckets plus storage for all elements.

Heap Sort using Max Heap in C



```c
#include <stdio.h>

void heapifyMax(int a[], int n, int i){
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && a[left] > a[largest])
        largest = left;
    if (right < n && a[right] > a[largest])
        largest = right;

    if (largest != i){
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapifyMax(a, n, largest);
    }
}

void heapSortMax(int a[], int n){
    for (int i = n / 2 - 1; i >= 0; i--)
        heapifyMax(a, n, i);

    for (int i = n - 1; i >= 0; i--){
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapifyMax(a, i, 0);
    }
}

int main(){
    int a[] = {10, 7, 9, 2, 15};
    int n = sizeof(a) / sizeof(a[0]);
    heapSortMax(a, n);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output:



```
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc maxheapsort.c
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
  2 7 9 10 15
○ PS C:\Users\sarva\OneDrive\Desktop\daa-lab> █
```

Time Complexity (Worst): O(n log n)

Justification: Each of the n deletions requires heapify (log n).

Space Complexity (Worst): O(1)

Justification: Array-based heap uses no extra memory.

Heap Sort using Min Heap in C

```c
#include <stdio.h>

void heapifyMin(int a[], int n, int i){
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && a[left] < a[smallest])
        smallest = left;
    if (right < n && a[right] < a[smallest])
        smallest = right;

    if (smallest != i){
        int temp = a[i];
        a[i] = a[smallest];
        a[smallest] = temp;

        heapifyMin(a, n, smallest);
    }
}

void heapSortMin(int a[], int n){
    for (int i = n / 2 - 1; i >= 0; i--)
        heapifyMin(a, n, i);

    for (int i = n - 1; i >= 0; i--){
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapifyMin(a, i, 0);
    }
}

int main(){
    int a[] = {12, 3, 19, 6, 5};
    int n = sizeof(a) / sizeof(a[0]);
    heapSortMin(a, n);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output:

```
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc minheapsort.c
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
  19 12 6 5 3
○ PS C:\Users\sarva\OneDrive\Desktop\daa-lab> █
```

Time Complexity (Worst): O(n log n)

Justification: Same heap operations as max heap.

Space Complexity (Worst): O(1)

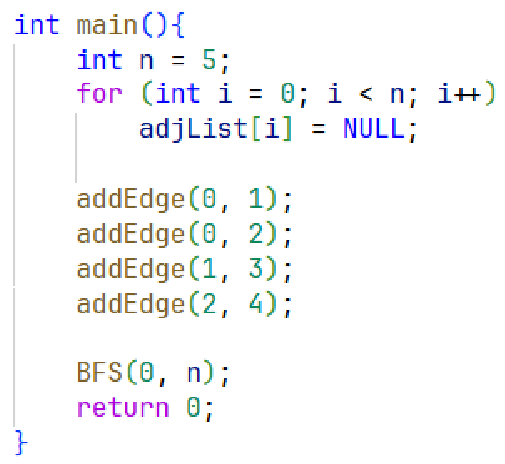
Justification: Sorting is done in-place.

Path Tracing Algorithms Date: 04-12-2025

1. Breadth first search in C
2. Depth first search in C

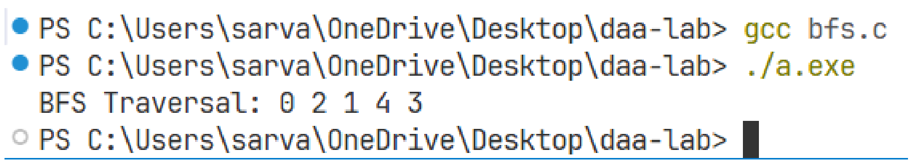
Breadth First Search in C

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

typedef struct Node{
    int vertex;
    struct Node *next;
} Node;

Node *adjList[MAX];
int visited[MAX];
int queue[MAX];
int front = 0, rear = -1;

void addEdge(int u, int v){
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->next = adjList[u];
    adjList[u] = newNode;
}

void BFS(int start){
    visited[start] = 1;
    queue[++rear] = start;

    printf("BFS Traversal: ");

    while (front <= rear){
        int curr = queue[front++];
        printf("%d ", curr);

        Node *temp = adjList[curr];
        while (temp != NULL){
            if (!visited[temp->vertex]){
                visited[temp->vertex] = 1;
                queue[++rear] = temp->vertex;
            }
            temp = temp->next;
        }
    }
}
```



```c
int main(){
    int n = 5;
    for (int i = 0; i < n; i++)
        adjList[i] = NULL;

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(2, 4);

    BFS(0, n);
    return 0;
}
```

Output:



```
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc bfs.c
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
  BFS Traversal: 0 2 1 4 3
○ PS C:\Users\sarva\OneDrive\Desktop\daa-lab> 
```

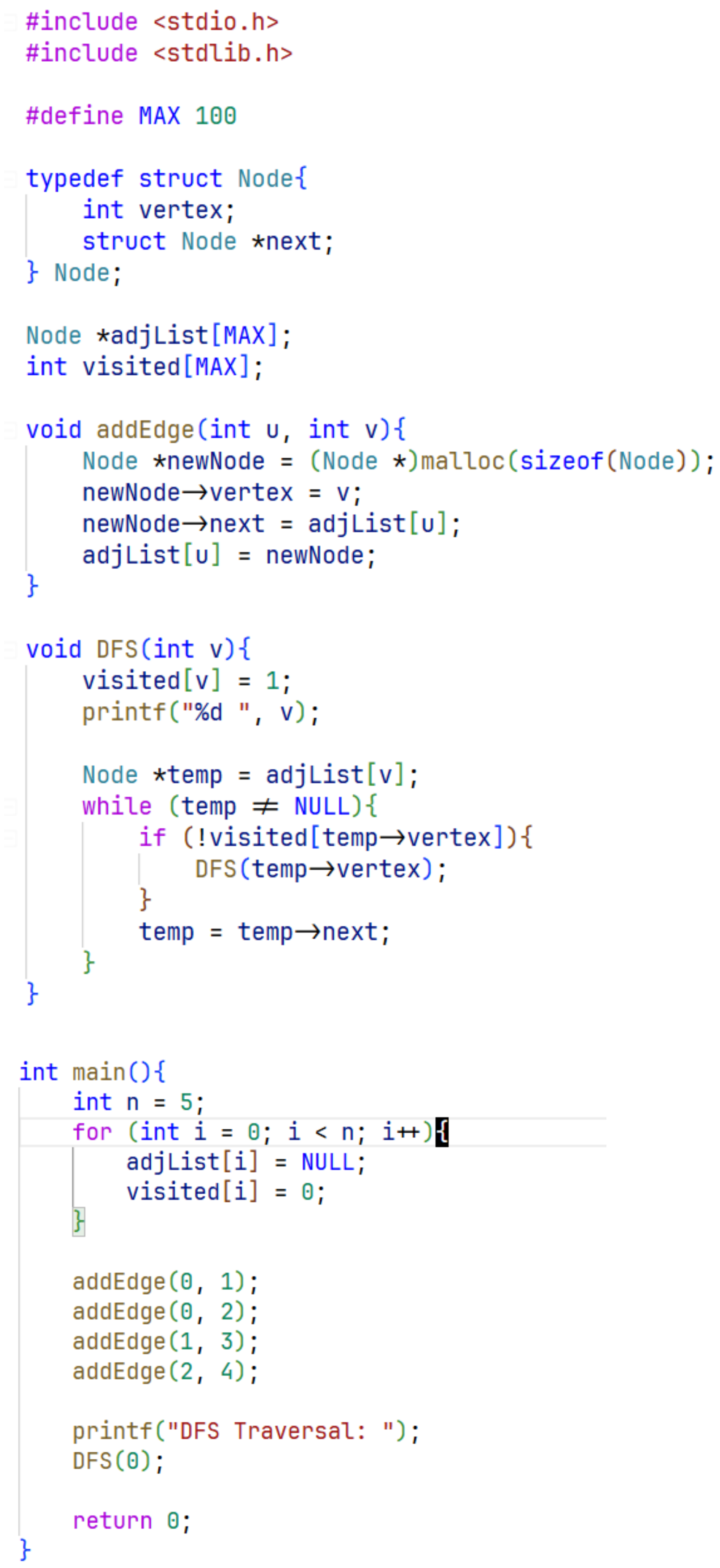
Time Complexity (Worst Case): O(V + E)

Justification for Time Complexity: BFS visits every vertex once and checks every edge once while exploring adjacency lists. Hence total work is proportional to the number of vertices plus edges.

Space Complexity (Worst Case): O(V)

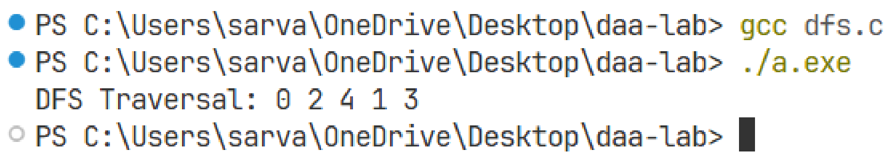
4. Justification for Space Complexity: The queue can hold up to V vertices in the worst case, and the visited[] array also requires O(V). (Adjacency list is part of input storage.)

Depth First Search in C



```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

typedef struct Node{
    int vertex;
    struct Node *next;
} Node;

Node *adjList[MAX];
int visited[MAX];

void addEdge(int u, int v){
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->next = adjList[u];
    adjList[u] = newNode;
}

void DFS(int v){
    visited[v] = 1;
    printf("%d ", v);

    Node *temp = adjList[v];
    while (temp != NULL){
        if (!visited[temp->vertex]){
            DFS(temp->vertex);
        }
        temp = temp->next;
    }
}


int main(){
    int n = 5;
    for (int i = 0; i < n; i++){
        adjList[i] = NULL;
        visited[i] = 0;
    }

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(2, 4);

    printf("DFS Traversal: ");
    DFS(0);

    return 0;
}
```

Output:



```
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> gcc dfs.c
● PS C:\Users\sarva\OneDrive\Desktop\daa-lab> ./a.exe
  DFS Traversal: 0 2 4 1 3
○ PS C:\Users\sarva\OneDrive\Desktop\daa-lab> █
```

Time Complexity (Worst Case): O(V + E)

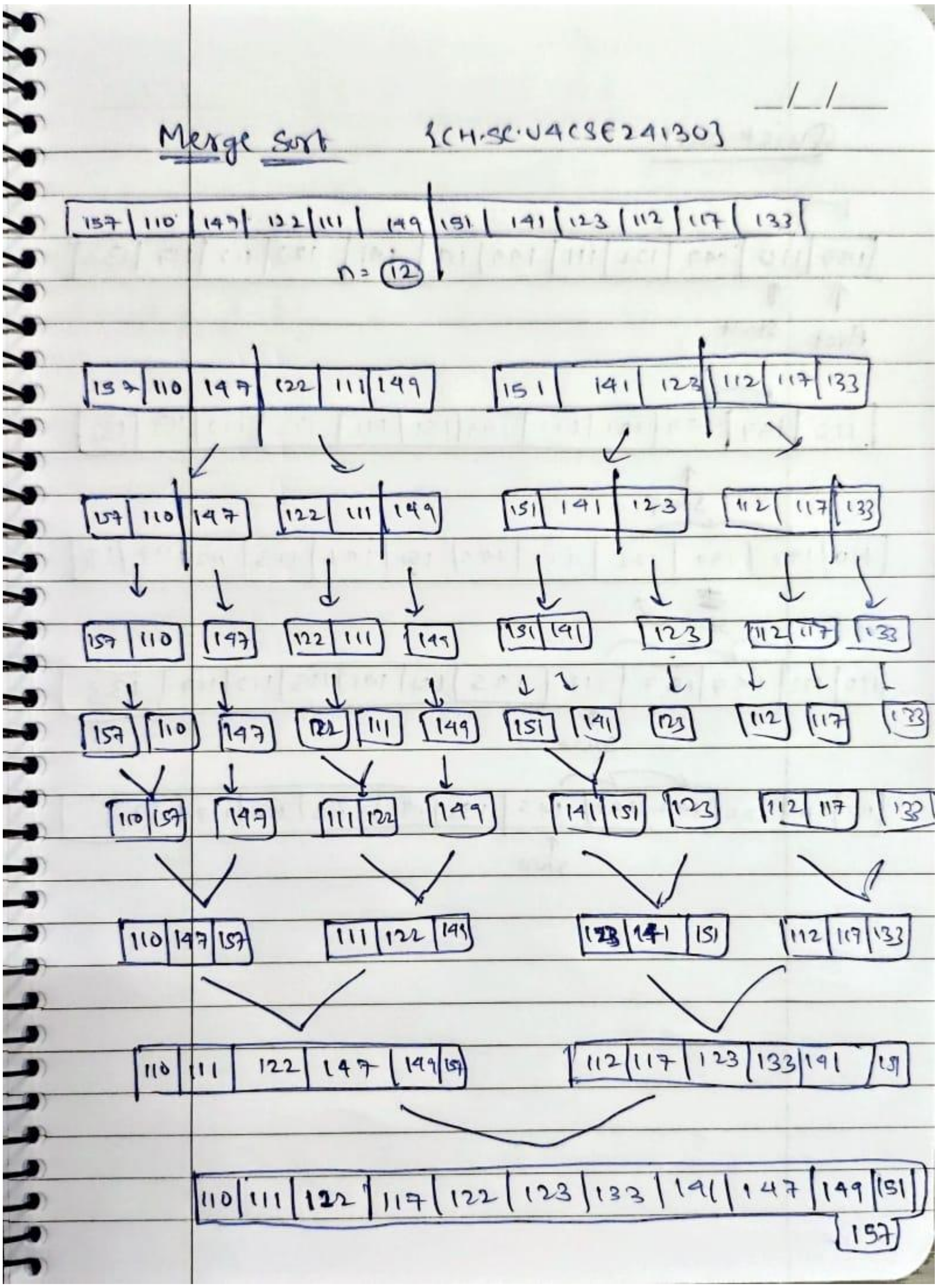
Justification for Time Complexity: DFS recursively explores every vertex and inspects all edges exactly once through the adjacency list, giving a total cost of V + E.

Space Complexity (Worst Case): O(V)

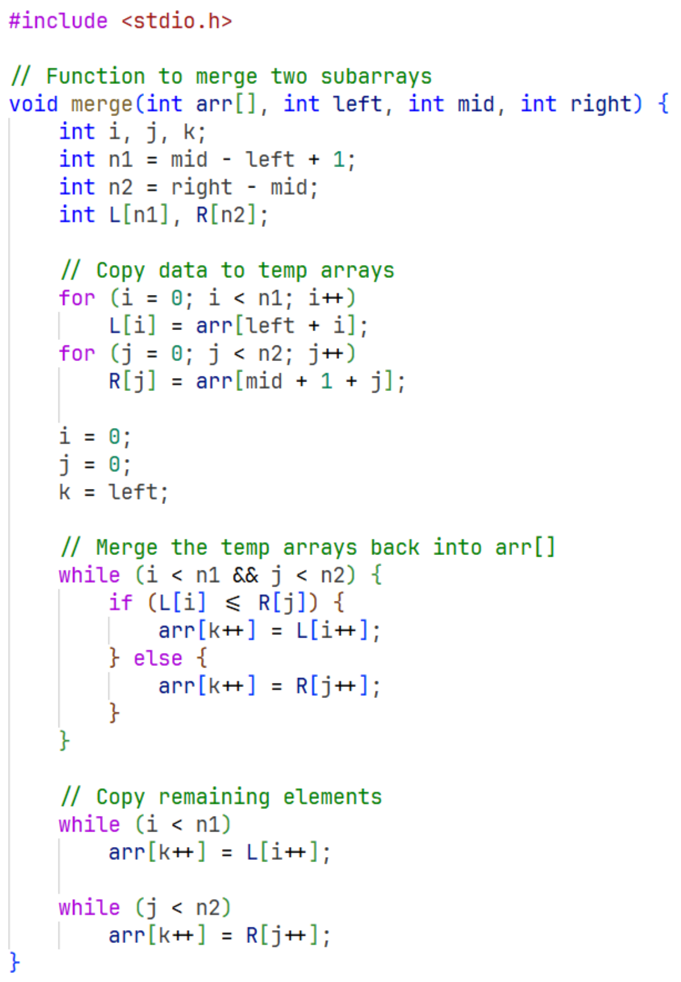
Justification for Space Complexity: The recursion stack can grow up to V levels (in a straight-chain graph), and the visited[] array stores V entries.

Divide & Conquer Techniques Date: 03-01-2026

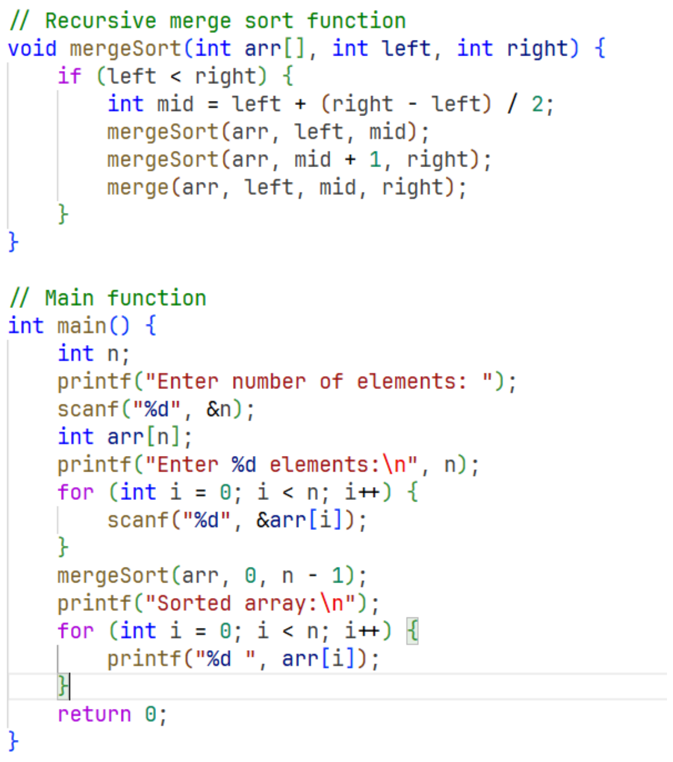
1. Merge Sort
2. Quick Sort



Merge Sort Code:

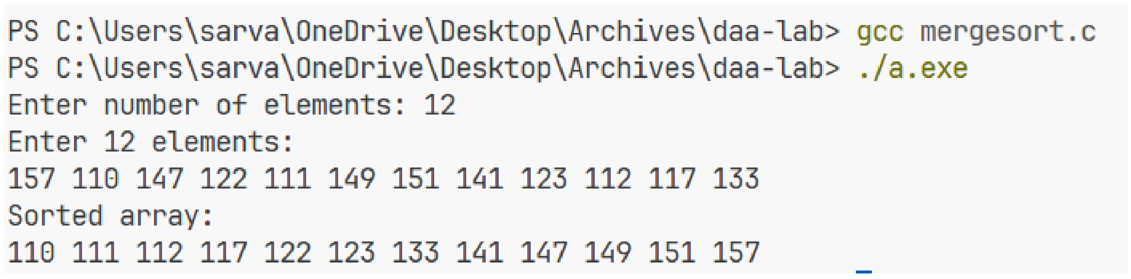


```c
#include <stdio.h>

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];

    // Copy data to temp arrays
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;

    // Merge the temp arrays back into arr[]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    // Copy remaining elements
    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[j++];
}
```



```c
// Recursive merge sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Main function
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    mergeSort(arr, 0, n - 1);
    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:



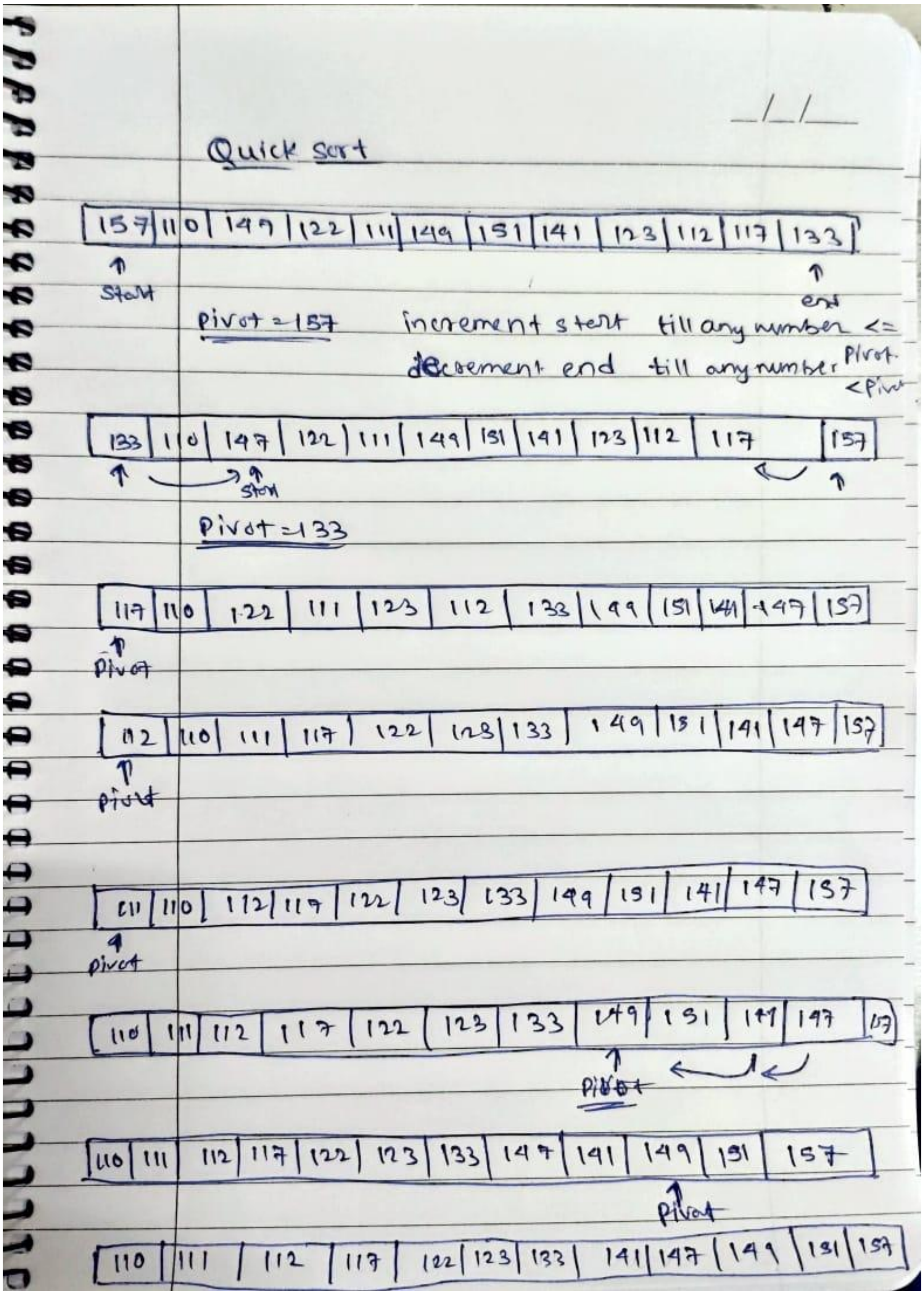
```
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> gcc mergesort.c
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
Enter number of elements: 12
Enter 12 elements:
157 110 147 122 111 149 151 141 123 112 117 133
Sorted array:
110 111 112 117 122 123 133 141 147 149 151 157
```

Justifications:

- **Time Complexity (Best, Average & Worst Case): O(n log n)**
- **Justification for Time Complexity:**

  The array is always divided into two equal parts, and merging checks all elements every time. This process repeats log n times, so the total time is n log n.
- **Space Complexity (Worst Case): O(n)**
- **Justification for Space Complexity:**

  An extra array is used to store elements while merging, which requires space equal to the number of elements.



Quick sort

157	110	147	122	111	149	151	141	123	112	117	133

Start   end

pivot = 157   increment start till any number <= Pivot.
decrement end till any number < Pivot

133	110	147	122	111	149	151	141	123	112	117	157

start

Pivot = 133

117	110	122	111	123	112	133	149	151	141	147	157

Pivot

112	110	111	117	122	123	133	149	151	141	147	157

pivot

111	110	112	117	122	123	133	149	151	141	147	157

pivot

110	111	112	117	122	123	133	149	151	141	147	157

Pivot

110	111	112	117	122	123	133	147	141	149	151	157

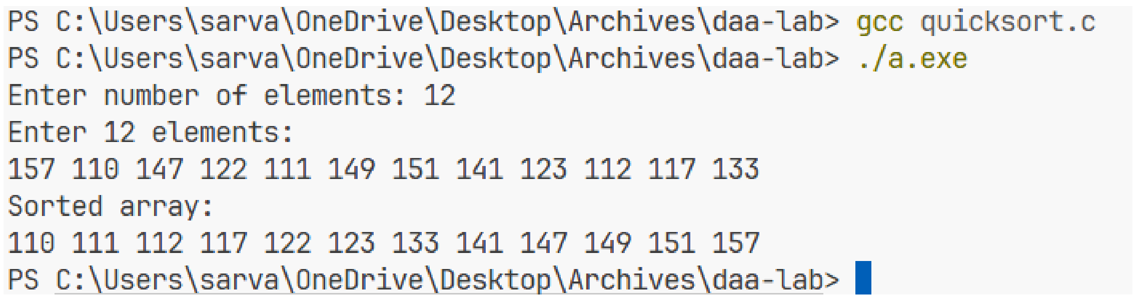
Pivot

110	111	112	117	122	123	133	141	147	149	151	157

Quick Sort Code:

```c
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];    // Choosing last element as pivot
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    quickSort(arr, 0, n - 1);
    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:



```
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> gcc quicksort.c
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
Enter number of elements: 12
Enter 12 elements:
157 110 147 122 111 149 151 141 123 112 117 133
Sorted array:
110 111 112 117 122 123 133 141 147 149 151 157
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab>
```

Justifications:

- **Time Complexity (Worst Case): O(n²)**
- **Justification for Time Complexity:**

  If the pivot is always the smallest or largest element, the array gets divided very unevenly. In this case, comparisons keep increasing for each step, leading to n × n operations.

- **Space Complexity (Worst Case): O(n)**
- **Justification for Space Complexity:**

  In the worst case, recursive calls go one by one, so the recursion stack stores up to n function calls.