



## 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  $3 \times 3$  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.

```
#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.

```
#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.

```
#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.

```
#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  $3 \times 3$  matrix.

```
#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  $3 \times 3$  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.

```
#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

```
#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

```
#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

```
#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

```
#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]-->0)
            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

```
#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

```
#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

```
#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;
        }
    }
}
```

```

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

```
#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):  $O(V)$**

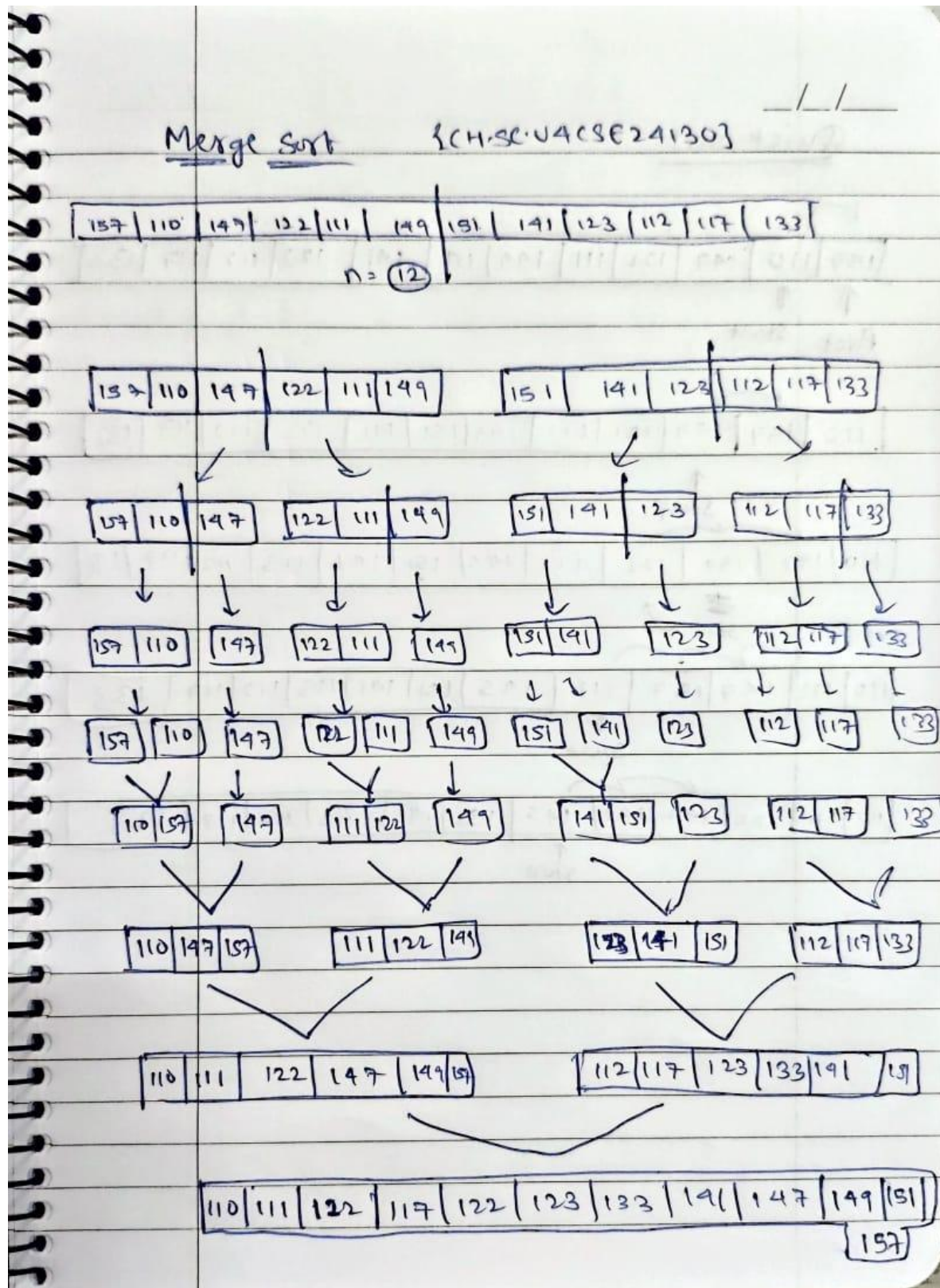
**Justification for Space Complexity:** The recursion stack can grow up to  $V$  levels (in a straight-chain graph), and the visited[] array stores  $V$  entries.

---

## Divide & Conquer Techniques

Date: 03-01-2026

1. Merge Sort
2. Quick Sort



## Merge Sort Code:

```
#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++];
}

// 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 | 149 | 122 | 111 | 149 | 151 | 141 | 123 | 112 | 117 | 133

↑  
start

↑  
end

pivot = 157

increment start till any number  $\leq$  pivot  
decrement end till any number  $<$  pivot

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

↑      → ↑  
start

←      ↑

pivot = 133

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

↑  
pivot

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

↑  
pivot

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

↑  
pivot

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

↑  
pivot      ←      ←

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

↑  
pivot

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

## Quick Sort Code:

```
#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):  $O(n^2)$**
- **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  $n \times n$  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.

# Balancing BST

Date: 08-01-2026

1. Balancing BST (AVL Rotations)
2. Balancing BST (Red Black Tree)

AVL Code:

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

/* ----- NODE STRUCTURE ----- */
struct Node{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

/* ----- UTILITY FUNCTIONS ----- */
int max(int a, int b){
    return (a > b) ? a : b;
}

int height(struct Node *n){
    if (n == NULL)
        return 0;
    return n->height;
}

/* ----- CREATE NODE ----- */
struct Node *newNode(int key){
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return node;
}

/* ----- RIGHT ROTATION ----- */
struct Node *rightRotate(struct Node *y){
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}
```

```

/* ----- LEFT ROTATION ----- */
struct Node *leftRotate(struct Node *x){
    struct Node *y = x→right;
    struct Node *T2 = y→left;

    // Perform rotation
    y→left = x;
    x→right = T2;

    // Update heights
    x→height = max(height(x→left), height(x→right)) + 1;
    y→height = max(height(y→left), height(y→right)) + 1;

    // Return new root
    return y;
}

/* ----- BALANCE FACTOR ----- */
int getBalance(struct Node *n){
    if (n == NULL)
        return 0;
    return height(n→left) - height(n→right);
}

/* ----- INSERT ----- */
struct Node *insert(struct Node *node, int key){
    // 1. Normal BST insertion
    if (node == NULL)
        return newNode(key);
    if (key < node→key)
        node→left = insert(node→left, key);
    else if (key > node→key)
        node→right = insert(node→right, key);
    else
        return node; // Duplicate keys not allowed

    // 2. Update height
    node→height = 1 + max(height(node→left), height(node→right));
    // 3. Get balance factor
    int balance = getBalance(node);
    // 4. Rotations
    // LL Case
    if (balance > 1 && key < node→left→key)
        return rightRotate(node);
    // RR Case
    if (balance < -1 && key > node→right→key)
        return leftRotate(node);
    // LR Case
    if (balance > 1 && key > node→left→key){
        node→left = leftRotate(node→left);
        return rightRotate(node);
    }
    // RL Case
    if (balance < -1 && key < node→right→key){
        node→right = rightRotate(node→right);
        return leftRotate(node);
    }
    return node;
}

```

```

/* ----- INORDER TRAVERSAL ----- */
void inorder(struct Node *root){
    if (root != NULL){
        inorder(root→left);
        printf("%d ", root→key);
        inorder(root→right);
    }
}

/* ----- MAIN ----- */
int main(){
    struct Node *root = NULL;

    int keys[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++)
        root = insert(root, keys[i]);

    printf("Inorder traversal of AVL Tree:\n");
    inorder(root);

    return 0;
}

```

Output:

```

PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> gcc avl_rotations.c
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
Inorder traversal of AVL Tree:
110 111 112 117 122 123 133 141 147 149 151 157

```

**Time & Space Complexities table with justifications:**

Operation	Time Complexity	Justification
Search	$O(\log n)$	Height is logarithmic
Insert	$O(\log n)$	BST insert + constant rotations
Delete	$O(\log n)$	Rebalancing bounded by height
Traversal	$O(n)$	Visit every node
Rotation	$O(1)$	Constant pointer updates
Space	$O(n)$	Store all nodes

## Red Black Tree Code:

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

/* ----- COLORS ----- */
#define RED 1
#define BLACK 0

/* ----- NODE STRUCTURE ----- */
struct Node{
    int data;
    int color;
    struct Node *left, *right, *parent;
};

struct Node *root = NULL;

/* ----- CREATE NODE ----- */
struct Node *createNode(int data){
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->data = data;
    node->color = RED; // New node is always RED
    node->left = node->right = node->parent = NULL;
    return node;
}

/* ----- LEFT ROTATION ----- */
void leftRotate(struct Node *x){
    struct Node *y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

/* ----- RIGHT ROTATION ----- */
void rightRotate(struct Node *y){
    struct Node *x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == NULL)
        root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}
```

```

/* ----- FIX VIOLATIONS ----- */
void fixInsert(struct Node *z){
    while (z != root && z->parent->color == RED){
        if (z->parent == z->parent->parent->left){
            struct Node *y = z->parent->parent->right; // Uncle
            // Case 1: Uncle is RED
            if (y != NULL && y->color == RED){
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            }
            // Case 2 & 3
            else{
                if (z == z->parent->right){
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(z->parent->parent);
            }
        }
        else{
            struct Node *y = z->parent->parent->left; // Uncle
            if (y != NULL && y->color == RED){
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            }
            else{
                if (z == z->parent->left){
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}

```

```

/* ----- INSERT ----- */
void insert(int data){
    struct Node *z = createNode(data);
    struct Node *y = NULL;
    struct Node *x = root;
    while (x != NULL){
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if (y == NULL)
        root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;
    fixInsert(z);
}

```

```

/* ----- INORDER TRAVERSAL ----- */
void inorder(struct Node *node){
    if (node != NULL){
        inorder(node->left);
        printf("%d(%s) ", node->data,
            node->color == RED ? "R" : "B");
        inorder(node->right);
    }
}

/* ----- MAIN ----- */
int main(){
    int keys[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++)
        insert(keys[i]);

    printf("Inorder Traversal of Red-Black Tree:\n");
    inorder(root);

    return 0;
}

```

Output:

```

PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> gcc red_black.c
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
Inorder Traversal of Red-Black Tree:
110(B) 111(R) 112(R) 117(B) 122(R) 123(B) 133(R) 141(B) 147(R) 149(R) 151(B) 157(R)
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> █

```

**Time & Space Complexities table with justifications:**

Operation	Time Complexity	Justification
<b>Search</b>	$O(\log n)$	Search follows a single root-to-leaf path. Height of Red-Black Tree is $O(\log n)$ , so comparisons are logarithmic.
<b>Insertion</b>	$O(\log n)$	BST insertion takes $O(\log n)$ due to bounded height. Fix-up involves recoloring and at most two rotations, each $O(1)$ .
<b>Deletion</b>	$O(\log n)$	BST deletion is $O(\log n)$ . Fix-up may propagate up the tree but height is logarithmic and each step is constant time.
<b>Traversal</b>	$O(n)$	Each node is visited exactly once during traversal.
<b>Rotation</b>	$O(1)$	Rotations involve a constant number of pointer changes, independent of tree size.
<b>Height</b>	$O(\log n)$	Red-Black properties limit height to at most $2 \log(n + 1)$ .
<b>Space</b>	$O(n)$	One node is stored per element in the tree.

## Quick Operations

Date: 22-01-2026

1. Quick Sort
2. Quick Select

Quick Sort Code:

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

void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

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

/* -----
   1) QUICK SORT : FIRST ELEMENT AS PIVOT
   ----- */

int partitionFirst(int arr[], int low, int high){
    int pivot = arr[low];
    int i = low + 1;
    int j = high;

    while (i ≤ j){
        while (i ≤ high && arr[i] ≤ pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i < j)
            swap(&arr[i], &arr[j]);
    }

    swap(&arr[low], &arr[j]);
    return j;
}

void quickSortFirst(int arr[], int low, int high){
    if (low < high){
        int p = partitionFirst(arr, low, high);
        quickSortFirst(arr, low, p - 1);
        quickSortFirst(arr, p + 1, high);
    }
}
```

```

/* -----
2) QUICK SORT : LAST ELEMENT AS PIVOT (LOMUTO)
----- */

int partitionLast(int arr[], int low, int high){
    int pivot = arr[high];
    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;
}

void quickSortLast(int arr[], int low, int high){
    if (low < high){
        int p = partitionLast(arr, low, high);
        quickSortLast(arr, low, p - 1);
        quickSortLast(arr, p + 1, high);
    }
}

/* -----
3) QUICK SORT : RANDOM ELEMENT AS PIVOT
----- */

int partitionRandom(int arr[], int low, int high){
    int randomIndex = low + rand() % (high - low + 1);
    swap(&arr[randomIndex], &arr[high]);
    return partitionLast(arr, low, high);
}

void quickSortRandom(int arr[], int low, int high){
    if (low < high){
        int p = partitionRandom(arr, low, high);
        quickSortRandom(arr, low, p - 1);
        quickSortRandom(arr, p + 1, high);
    }
}

```

```

/* -----
MAIN FUNCTION
----- */

int main()
{
    srand(time(NULL));

    int arr1[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int arr2[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int arr3[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};

    int n = sizeof(arr1) / sizeof(arr1[0]);

    printf("Original Array:\n");
    printArray(arr1, n);

    quickSortFirst(arr1, 0, n - 1);
    printf("\nSorted using First Element as Pivot:\n");
    printArray(arr1, n);

    quickSortLast(arr2, 0, n - 1);
    printf("\nSorted using Last Element as Pivot:\n");
    printArray(arr2, n);

    quickSortRandom(arr3, 0, n - 1);
    printf("\nSorted using Random Element as Pivot:\n");
    printArray(arr3, n);

    return 0;
}

```

Output:

```

PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
Original Array:
157 110 147 122 111 149 151 141 123 112 117 133

Sorted using First Element as Pivot:
110 111 112 117 122 123 133 141 147 149 151 157

Sorted using Last Element as Pivot:
110 111 112 117 122 123 133 141 147 149 151 157

Sorted using Random Element as Pivot:
110 111 112 117 122 123 133 141 147 149 151 157
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab>

```

Time and Space Complexities:

Pivot Selection	Time Complexity	Justification (TC)	Space Complexity	Justification (SC)
First Element as Pivot	$O(n^2)$	If array is already sorted (ascending/descending), pivot becomes smallest/largest $\rightarrow$ partitions (0, $n-1$ ) repeatedly $\rightarrow$ recurrence: $T(n)=T(n-1)+\Theta(n)$	$O(n)$	Recursion depth becomes $n$ due to skewed partitions
Last Element as Pivot	$O(n^2)$	Same as first pivot: sorted or reverse-sorted array causes pivot to be extreme $\rightarrow$ unbalanced partitions at every step	$O(n)$	Call stack grows linearly to $n$
Random Element as Pivot	$O(n^2)$ ( <i>rare</i> )	Worst case still possible if random pivot repeatedly chooses extreme element	$O(n)$	Worst case recursion depth can still reach $n$

Quick Select Code:

```
#include <stdio.h>

// Swap function
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];
    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 Select function
int quickSelect(int arr[], int low, int high, int k){
    if (low ≤ high){
        int pi = partition(arr, low, high);

        if (pi == k)
            return arr[pi];
        else if (pi > k)
            return quickSelect(arr, low, pi - 1, k);
        else
            return quickSelect(arr, pi + 1, high, k);
    }

    return -1; // Invalid case
}

// Driver code
int main(){
    int arr[] = {7, 10, 4, 3, 20, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2; // 0-based index (2 means 3rd smallest)
    int result = quickSelect(arr, 0, n - 1, k);
    printf("K-th smallest element: %d\n", result);
    return 0;
}
```

Output:

```
● PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> gcc quickselect.c
● PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
  K-th smallest element: 7
○ PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> █
```

Time and Space Complexities:

### **Time Complexity**

Worst Case:  $O(n^2)$

Justification:

- Pivot is always the smallest or largest element.
- Each partition reduces problem size by only 1.

### **Space Complexity**

Worst Case:  $O(n)$

Justification:

- Unbalanced recursion leads to  $n$  recursive calls on the stack.

## Job Scheduling

Date: 12-02-2026

Q) Let there be 14 Jobs with the profits {22, 19, 29, 28, 30, 21, 27, 25, 24, 26, 14, 27, 19, 11} and with job completion times {3,3,8,6,7,5,10,4,6,12,13,2,14,1}

Job Scheduling C Code:

```
#include <stdio.h>

int main() {
    int n = 14;

    int profit[14] = {22,19,29,28,30,21,27,25,24,26,14,27,19,11};
    int deadline[14] = {3,3,8,6,7,5,10,4,6,12,13,2,14,1};
    int job[14];

    int slot[15]; // slots from 1 to 14
    int i, j;

    // Initialize job numbers
    for(i = 0; i < n; i++)
        job[i] = i + 1;

    // Initialize slots as empty
    for(i = 1; i ≤ 14; i++)
        slot[i] = -1;

    // Step 1: Sort jobs by profit (Descending) using simple bubble sort
    for(i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if(profit[j] < profit[j+1]) {
                // Swap profit
                int temp = profit[j];
                profit[j] = profit[j+1];
                profit[j+1] = temp;

                // Swap deadline
                temp = deadline[j];
                deadline[j] = deadline[j+1];
                deadline[j+1] = temp;

                // Swap job number
                temp = job[j];
                job[j] = job[j+1];
                job[j+1] = temp;
            }
        }
    }
}
```

```

int totalProfit = 0;

// Step 2: Assign jobs to slots
for(i = 0; i < n; i++) {
    for(j = deadline[i]; j > 0; j--) {
        if(slot[j] == -1) {
            slot[j] = job[i];
            totalProfit = totalProfit + profit[i];
            break;
        }
        else {
            // Slot already filled, try previous slot
        }
    }
}

// Step 3: Display result
printf("Selected Jobs:\n");
for(i = 1; i ≤ 14; i++) {
    if(slot[i] ≠ -1)
        printf("Slot %d → Job %d\n", i, slot[i]);
}

printf("\nMaximum Profit = %d\n", totalProfit);

return 0;
}

```

Output:

```

• PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> gcc job_scheduling.c
• PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab> ./a.exe
Selected Jobs:
Slot 1 -> Job 6
Slot 2 -> Job 12
Slot 3 -> Job 1
Slot 4 -> Job 8
Slot 5 -> Job 9
Slot 6 -> Job 4
Slot 7 -> Job 5
Slot 8 -> Job 3
Slot 10 -> Job 7
Slot 12 -> Job 10
Slot 13 -> Job 11
Slot 14 -> Job 13

Maximum Profit = 292
• PS C:\Users\sarva\OneDrive\Desktop\Archives\daa-lab>

```

## **Time Complexity**

Let  $n$  be the number of jobs.

### **1. Sorting Phase (Bubble Sort)**

Two nested loops are used to sort jobs in decreasing order of profit.

- Outer loop runs  $(n - 1)$  times.
- Inner loop runs up to  $(n - 1)$  times.

In the worst case, total comparisons are proportional to:  $n \times n = n^2$

Therefore, **Time complexity of sorting =  $O(n^2)$**

Justification: Bubble sort compares each element with others using nested loops.

### **2. Scheduling Phase**

Again, two nested loops are used:

- Outer loop runs  $n$  times (for each job).
- Inner loop may run up to  $n$  times (checking slots backward).

In the worst case, for every job all previous slots are checked.

Thus,  $n \times n = n^2$

Therefore, **Time complexity of scheduling =  $O(n^2)$**

## **Overall Time Complexity**

Sorting:  $O(n^2)$

Scheduling:  $O(n^2)$

Total:  $O(n^2) + O(n^2) = O(n^2)$

Final Answer: **Time Complexity =  $O(n^2)$**

## **Space Complexity**

The program uses:

- $\text{profit}[n]$ ,  $\text{deadline}[n]$ ,  $\text{job}[n]$ ,  $\text{slot}[n]$

All arrays are of size proportional to  $n$ .

No recursion and no additional dynamic memory is used.

Therefore,

**Space Complexity =  $O(n)$**

# Huffman Coding

Date: 19-02-2026

Q) Perform Huffman Coding on the following sentence “Data Analytics and Intelligence Laboratory”. Build the Huffman Tree and print the codes.

Code:

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

#define MAX 256

// Huffman Tree Node
struct Node{
    char data;
    int freq;
    struct Node *left, *right;
};

// Create new node
struct Node *newNode(char data, int freq){
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->data = data;
    node->freq = freq;
    node->left = node->right = NULL;
    return node;
}

// Find two minimum frequency nodes
void findTwoMin(struct Node *arr[], int size, int *min1, int *min2){
    *min1 = 0;
    *min2 = 1;
    if (arr[*min1]->freq > arr[*min2]->freq){
        int temp = *min1;
        *min1 = *min2;
        *min2 = temp;
    }

    for (int i = 2; i < size; i++){
        if (arr[i]->freq < arr[*min1]->freq){
            *min2 = *min1;
            *min1 = i;
        }
        else if (arr[i]->freq < arr[*min2]->freq){
            *min2 = i;
        }
    }
}
```

```

// Print Huffman Codes
void printCodes(struct Node *root, int arr[], int top){
    if (root->left){
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    if (root->right){
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    if (!root->left && !root->right){
        printf("%c : ", root->data);
        for (int i = 0; i < top; i++){
            printf("%d", arr[i]);
        }
        printf("\n");
    }
}
}

```

```

int main(){
    char text[] = "Data Analytics and Intelligence Laboratory";
    int freq[MAX] = {0};

    // Count frequency
    for (int i = 0; text[i] != '\0'; i++){
        freq[(int)text[i]]++;
    }

    struct Node *nodes[MAX];
    int size = 0;

    // Create nodes for each character
    for (int i = 0; i < MAX; i++){
        if (freq[i] > 0){
            nodes[size++] = newNode((char)i, freq[i]);
        }
    }

    // Build Huffman Tree
    while (size > 1){
        int min1, min2;
        findTwoMin(nodes, size, &min1, &min2);
        struct Node *left = nodes[min1];
        struct Node *right = nodes[min2];
        struct Node *new = newNode('$', left->freq + right->freq);
        new->left = left;
        new->right = right;
        if (min1 > min2){
            nodes[min1] = nodes[size - 1];
            nodes[min2] = new;
        }
        else{
            nodes[min2] = nodes[size - 1];
            nodes[min1] = new;
        }

        size--;
    }

    int arr[MAX], top = 0;
    printf("Huffman Codes:\n");
    printCodes(nodes[0], arr, top);

    return 0;
}

```

Output:

```
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa\DAA-LAB\Lab 8 (19-02-2026)> gcc huffman_coding.c
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa\DAA-LAB\Lab 8 (19-02-2026)> ./a.exe
Huffman Codes:
t : 000
b : 00100
d : 00101
c : 0011
s : 01000
g : 01001
r : 0101
n : 011
i : 1000
l : 1001
a : 101
e : 1100
   : 1101
A : 111000
D : 111001
y : 11101
o : 11110
I : 111110
L : 111111
PS C:\Users\sarva\OneDrive\Desktop\Archives\daa\DAA-LAB\Lab 8 (19-02-2026)> |
```

### Time Complexity

1. **Counting frequency**  $\rightarrow O(n)$   
(We scan the sentence once)
2. **Building Huffman Tree**  $\rightarrow O(k^2)$   
(Each time we search for two minimum nodes manually)
3. **Printing codes**  $\rightarrow O(k)$

### Total Time Complexity:

$O(n + k^2)$

If characters are ASCII (max 256), then  $k$  is constant.

So overall it becomes:

$O(n)$

### Space Complexity

- Frequency array  $\rightarrow O(1)$  (fixed 256 size)
- Tree nodes  $\rightarrow O(k)$

### Total Space Complexity:

$O(k)$

For ASCII characters:

$O(1)$