AMRITA VISHWA VIDYAPEETHAM | School of Engineering Chennai Campus

23CSE211- DESIGN AND ANALYSIS OF ALGORITHMS

AMRITA VISHWA VIDYAPEETHAM

CHENNAI CAMPUS

Name: SEETHAKA.LAHARI SAI

Roll No: CH.SC.U4CSE24143

Class : CSE-B

Year: 2025-2026

WEEK-2

Program-1: BUBBLE SORT

Code:

```c
#include <stdio.h>

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

int main() {
    int n;

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

    int arr[n];

    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, n);

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

    return 0;
```

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./bubbleSort
Enter number of elements: 5
Enter 5 integers:
9 4 6 2 0
Sorted array:
0 2 4 6 9
```

Time Complexity:

- **Worst & Average Case: $O(n^2)$ because the algorithm always uses two nested loops that compare and swap adjacent elements repeatedly.**
- **Best Case: $O(n^2)$ for this specific code since it does not include a flag to stop early even if the array is already sorted.**

Space Complexity:

- **O(1)** — The algorithm uses only a few extra variables **(`i`, `j`, `temp`)** regardless of input size.
- Sorting is done in-place, so no additional arrays or dynamic memory are used.

Program-2: INSERTION SORT

Code:

```c
#include <stdio.h>

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

        // Move elements greater than key to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
int main() {
    int n;

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

    int arr[n];

    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);
    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./insertionSort
Enter number of elements: 6
Enter 6 integers:
3 8 9 2 4 1
Sorted array:
1 2 3 4 8 9
```

Time Complexity

- **Best Case: O(n)** — when the array is already sorted (only one comparison per element).
- **Average & Worst Case: O(n^2)** — when elements must be shifted many times (e.g., reverse order).

Space Complexity

- **O(1)** — works in-place, using only a few extra variables (`key`, `i`, `j`).

Program-3: SELECTION SORT

Code:

```c
#include <stdio.h>
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        // Find the minimum element in the unsorted part
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap if a smaller element is found
        if (minIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}

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

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./selectionSort
Enter number of elements: 7
Enter 7 integers:
1 9 5 8 3 2 7
Sorted array:
1 2 3 5 7 8 9
```

Time Complexity

- **Best Case: O(n^2)**
- **Average Case: O(n^2)**
- **Worst Case: O(n^2)**

Because Selection Sort always scans the unsorted part to find the minimum, even if the array is already sorted.

Space Complexity

- **O(1)** — Sorting is done in-place with only a constant amount of extra space

Program-4: BUCKET SORT

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
struct Node {
    int data;
    struct Node* next;
};
void insert(struct Node** bucket, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (*bucket == NULL || value < (*bucket)->data) {
        newNode->next = *bucket;
        *bucket = newNode;
    } else {
        struct Node* current = *bucket;
        while (current->next != NULL && current->next->data < value) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}

void bucketSort(int arr[], int n) {
    struct Node* buckets[MAX] = {NULL};
    int maxValue = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > maxValue)
            maxValue = arr[i];
    }
```

```c
    for (int i = 0; i < n; i++) {
        int index = (arr[i] * (MAX - 1)) / maxValue;
        insert(&buckets[index], arr[i]);
    }
    int k = 0;
    for (int i = 0; i < MAX; i++) {
        struct Node* current = buckets[i];
        while (current != NULL) {
            arr[k++] = current->data;
            current = current->next;
        }
    }
}

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

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./bucketSort
Enter number of elements: 5
Enter 5 integers:
1 8 5 9 2
Sorted array:
1 2 5 8 9
```

Time Complexity

- **Best Case: O(n + k)** — when elements distribute evenly among buckets.
- **Average Case: O(n + k)**
- **Worst Case: O(n^2)** — if all elements fall into the same bucket.

(k = number of buckets)

Space Complexity

- **O(n + k)** — uses extra bucket storage plus linked list nodes.

Program-5: HEAP SORT (MAX HEAP & MIN HEAP)

% MAX HEAP :

Code:

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int *arr;
    int size;
    int capacity;
} MaxHeap;
MaxHeap* createHeap(int capacity) {
    MaxHeap* h = (MaxHeap*)malloc(sizeof(MaxHeap));
    h->arr = (int*)malloc(sizeof(int) * capacity);
    h->size = 0;
    h->capacity = capacity;
    return h;
}
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
void heapifyUp(MaxHeap* h, int idx) {
    while (idx > 0) {
        int parent = (idx - 1) / 2;
        if (h->arr[parent] < h->arr[idx]) {
            swap(&h->arr[parent], &h->arr[idx]);
            idx = parent;
        } else break;
    }
}
void heapifyDown(MaxHeap* h, int idx) {
    int left, right, largest;
    while (1) {
        left = 2*idx + 1;
        right = 2*idx + 2;
        largest = idx;
        if (left < h->size && h->arr[left] > h->arr[largest]) largest = left;
        if (right < h->size && h->arr[right] > h->arr[largest]) largest = right;
        if (largest != idx) {
            swap(&h->arr[idx], &h->arr[largest]);
```

```
            idx = largest;
        } else break;
    }
}

void insertHeap(MaxHeap* h, int value) {
    if (h->size == h->capacity) {
        // optional: resize
        h->capacity *= 2;
        h->arr = (int*)realloc(h->arr, sizeof(int) * h->capacity);
    }
    h->arr[h->size] = value;
    heapifyUp(h, h->size);
    h->size++;
}
int extractMax(MaxHeap* h) {
    if (h->size == 0) return -1; // or signal error
    int root = h->arr[0];
    h->arr[0] = h->arr[h->size - 1];
    h->size--;
    heapifyDown(h, 0);
    return root;
}
void buildHeap(MaxHeap* h, int input[], int n) {
    // copy elements
    for (int i = 0; i < n; ++i) h->arr[i] = input[i];
    h->size = n;
    // heapify from last internal node downwards
    for (int i = (h->size / 2) - 1; i >= 0; --i) {
        heapifyDown(h, i);
    }
}
```

```
void printHeap(MaxHeap* h) {
    printf("Heap (array form): ");
    for (int i = 0; i < h->size; ++i) printf("%d ", h->arr[i]);
    printf("\n");
}
int main() {
    int n;
    printf("Enter number of elements: ");
    if (scanf("%d", &n) != 1 || n <= 0) {
        printf("Invalid input.\n");
        return 1;
    }
    int *a = (int*)malloc(sizeof(int) * n);
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; ++i) scanf("%d", &a[i]);
    MaxHeap *h = createHeap(n);
    buildHeap(h, a, n);
    printHeap(h);
    printf("Extracting elements (descending order):\n");
    while (h->size > 0) {
        int mx = extractMax(h);
        printf("%d ", mx);
    }
    printf("\n");
    free(a);
    free(h->arr);
    free(h);
    return 0;
}
```

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./maxheap
Enter number of elements: 5
Enter 5 integers:
4 3 7 5 9
Heap (array form): 9 5 7 4 3
Extracting elements (descending order):
9 7 5 4 3
```

Complexity

- **Insert:** O(log n) (heapify up).
- **ExtractMax:** O(log n) (heapify down).
- **BuildHeap (from array):** O(n).
- **Space Complexity:** O(n) for the heap array (in-place aside from this storage).

% Min HEAP :

Code:

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int *arr;
    int size;
    int capacity;
} MinHeap;
MinHeap* createHeap(int capacity) {
    MinHeap* h = (MinHeap*)malloc(sizeof(MinHeap));
    h->arr = (int*)malloc(sizeof(int) * capacity);
    h->size = 0;
    h->capacity = capacity;
    return h;
}
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
void heapifyUp(MinHeap* h, int idx) {
    while (idx > 0) {
        int parent = (idx - 1) / 2;
        if (h->arr[parent] > h->arr[idx]) {
            swap(&h->arr[parent], &h->arr[idx]);
            idx = parent;
        } else break;
    }
}
void heapifyDown(MinHeap* h, int idx) {
    while (1) {
        int left = 2 * idx + 1;
        int right = 2 * idx + 2;
        int smallest = idx;
        if (left < h->size && h->arr[left] < h->arr[smallest])
            smallest = left;
        if (right < h->size && h->arr[right] < h->arr[smallest])
            smallest = right;
```

```
        if (smallest != idx) {
            swap(&h->arr[idx], &h->arr[smallest]);
            idx = smallest;
        } else {
            break;
        }
    }
}
void insertHeap(MinHeap* h, int value) {
    if (h->size == h->capacity) {
        h->capacity *= 2;
        h->arr = (int*)realloc(h->arr, sizeof(int) * h->capacity);
    }
    h->arr[h->size] = value;
    heapifyUp(h, h->size);
    h->size++;
}
int extractMin(MinHeap* h) {
    if (h->size == 0) return -1;  // or error
    int root = h->arr[0];
    h->arr[0] = h->arr[h->size - 1];
    h->size--;
    heapifyDown(h, 0);
    return root;
}
void buildHeap(MinHeap* h, int input[], int n) {
    for (int i = 0; i < n; i++)
        h->arr[i] = input[i];
    h->size = n;
    for (int i = (n / 2) - 1; i >= 0; i--)
        heapifyDown(h, i);
}
```

```
void printHeap(MinHeap* h) {
    printf("Min-Heap (array form): ");
    for (int i = 0; i < h->size; i++)
        printf("%d ", h->arr[i]);
    printf("\n");
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    MinHeap* h = createHeap(n);
    buildHeap(h, arr, n);
    printHeap(h);
    printf("Extracting elements (ascending order):\n");
    while (h->size > 0) {
        printf("%d ", extractMin(h));
    }
    printf("\n");
    free(h->arr);
    free(h);
    return 0;
}
```

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./minheap
Enter number of elements: 6
Enter 6 integers:
7 1 9 2 3 0
Min-Heap (array form): 0 1 7 2 3 9
Extracting elements (ascending order):
0 1 2 3 7 9
```

Time Complexity

- **Insert: O(log n)**
- **Extract Min: O(log n)**
- **Build Heap (from array): O(n)**
- **Peek Min: O(1)**

Space Complexity

- **O(n)** — heap stored in an array (in-place aside from storage).
-

Program-6: BFS

Code:

```c
#include <stdio.h>

#define MAX 100

int queue[MAX], front = -1, rear = -1;

void enqueue(int x) {
    if (rear == MAX - 1) return;
    if (front == -1) front = 0;
    queue[++rear] = x;
}

int dequeue() {
    if (front == -1 || front > rear) return -1;
    return queue[front++];
}

int isEmpty() {
    return (front == -1 || front > rear);
}

void bfs(int adj[MAX][MAX], int n, int start) {
    int visited[MAX] = {0};

    enqueue(start);
    visited[start] = 1;

    printf("BFS Traversal: ");

    while (!isEmpty()) {
        int node = dequeue();
        printf("%d ", node);
```

```
        for (int i = 0; i < n; i++) {
            if (adj[node][i] == 1 && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }

    printf("\n");
}

int main() {
    int n;
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    int adj[MAX][MAX];

    printf("Enter adjacency matrix (%d x %d):\n", n, n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    int start;
    printf("Enter starting vertex: ");
    scanf("%d", &start);

    bfs(adj, n, start);

    return 0;
}
```

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./bfs1
Enter number of vertices: 4
Enter adjacency matrix (4 x 4):
1 2 3 4
5 6 7 8
3 6 9 1
8 3 6 1
Enter starting vertex: 2
BFS Traversal: 2 3
```

Time Complexity

O(V + E)
BFS visits every vertex once and checks all edges connected to them.

Space Complexity

O(V) for the visited array and the queue.
O(V^2) for adjacency matrix storage (since your code uses a matrix).

Program-7: DFS

Code:

```c
#include <stdio.h>

#define MAX 100

int visited[MAX];

void dfs(int adj[MAX][MAX], int n, int node) {
    visited[node] = 1;
    printf("%d ", node);

    for (int i = 0; i < n; i++) {
        if (adj[node][i] == 1 && !visited[i]) {
            dfs(adj, n, i);
        }
    }
}
```

```c
int main() {
    int n;
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    int adj[MAX][MAX];

    printf("Enter adjacency matrix (%d x %d):\n", n, n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

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

    int start;
    printf("Enter starting vertex: ");
    scanf("%d", &start);

    printf("DFS Traversal: ");
    dfs(adj, n, start);
    printf("\n");

    return 0;
}
```

Output:

```
lahari@DESKTOP-JIGL8HI:~$ ./dfs
Enter number of vertices: 3
Enter adjacency matrix (3 x 3):
1 2 3
4 5 6
7 8 9
Enter starting vertex: 2
DFS Traversal: 2
```

Time Complexity

O(V + E)
DFS visits every vertex once and explores all edges exactly one time.

Space Complexity

O(V) for the recursion stack + visited array.
O(V^2) for adjacency matrix storage (used by your code).

---------- * * * ----------