AMRITA
VISHWA VIDYAPEETHAM

Name: Chetan Kumar G

CSE - B

CH.SC.U4CSE24109

Week -2(04/12/2025)

1. Bubble Sort

CODE:

```
//Bubble Sort
#include <stdio.h>
int main(){
int n;
printf("Enter the size of list");
scanf("%d",&n);
int arr[n];
for(int i = 0; i < n; i++) {
scanf("%d",&arr[i]);
}
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;
}
}
}
for(int i = 0; i < n; i++) {
printf("%d ",arr[i]);
}
return 0;
}
```

OUTPUT:

```
chetan@amma07:~/DAA/4dec$ gcc 1.c -o 1
chetan@amma07:~/DAA/4dec$ ./1
Enter the size of list5
2 3 1 5 4
1 2 3 4 5 chetan@amma07:~/DAA/4dec$
```

Time Complexity:

- Best: **$O(n)$** (already sorted, no swaps)
- Average & Worst: **$O(n^2)$**

Justification:
Bubble sort repeatedly compares adjacent elements and swaps them if needed.
For every element, it may compare with all others.
Hence, nested loops give quadratic time.

Space Complexity: $O(1)$
It sorts in-place using only a temporary variable for swapping.

2. Insertion Sort

CODE:

```c
//Insertion Sort
#include <stdio.h>
int main(){
int n;
printf("Enter the size of list");
scanf("%d",&n);
int arr[n];
for(int i = 0; i < n; i++) {
scanf("%d",&arr[i]);
}
for(int i = 1; i < n; i++) {
int key =arr[i];
int j=i-1;
while(j>=0 && arr[j]>key){
arr[j+1]=arr[j];
j--;
}
arr[j+1]=key;
}
for(int i = 0; i < n; i++) {
printf("%d ",arr[i]);
}
return 0;
}
```

OUTPUT:

```
chetan@amma07:~/DAA/4dec$ gcc p2.c -o p2
chetan@amma07:~/DAA/4dec$ ./p2
Enter the size of list5
2 3 1 5 4
1 2 3 4 5 chetan@amma07:~/DAA/4dec$
```

Time Complexity:

- Best: **O(n)** (already sorted)
- Average & Worst: **O(n²)**

Justification:

Each element is compared with all previous elements in the worst case.
In a sorted array, only one comparison per element is needed.
Thus, time varies based on input order.

Space Complexity: O(1)

Sorting is done in-place without extra memory.

3. Selection Sort

CODE:

```
//Selection Sort
#include <stdio.h>
int main(){
int n;
printf("Enter the size of list");
scanf("%d",&n);
int arr[n];
for(int i = 0; i < n; i++) {
scanf("%d",&arr[i]);
}
for(int i = 0; i < n-1; i++) {
int min=i;
for(int j=i+1;j<n;j++){
if(arr[j] < arr[min])
min= j;
}
int temp = arr[min];
arr[min] = arr[i];
arr[i] = temp;
}

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

OUTPUT:

```
chetan@amma07:~/DAA/4dec$
chetan@amma07:~/DAA/4dec$ gcc p3.c -o p3
chetan@amma07:~/DAA/4dec$ ./p3
Enter the size of list5
2 3 1 5 4
1 2 3 4 5 chetan@amma07:~/DAA/4dec$
```

Time Complexity:

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

Justification:

Selection sort always scans the remaining unsorted array to find the minimum.

Number of comparisons does not depend on input order.

Hence, time complexity is always quadratic.

Space Complexity: $O(1)$

Only one temporary variable is used for swapping.

4. Bucket Sort

CODE:

```c
//Bucket Sort
#include <stdio.h>
#include <stdlib.h>
void bucketSort(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    int *bucket = (int *)calloc(max + 1, sizeof(int));
    for (int i = 0; i < n; i++)
        bucket[arr[i]]++;
    int index = 0;
    for (int i = 0; i <= max; i++) {
        while (bucket[i] > 0) {
            arr[index++] = i;
            bucket[i]--;
        }
    }
    free(bucket);
}
```

```c
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]);
    }
    bucketSort(arr, n);
    printf("\nSorted Array: ");
    for (int i = 0; i < n; i++){
        printf("%d ", arr[i]);
    }
    return 0;
}
```

OUTPUT:

```
chetan@amma07:~$ gcc 4.c -o 4
chetan@amma07:~$ ./4
Enter number of elements: 5
Enter 5 elements:
2 3 1 5 4

Sorted Array: 1 2 3 4 5 chetan@amma07:~$
```

Time Complexity:

- Best & Average: **O(n + k)**
- Worst: **O(n²)**

Justification:

Elements are distributed into **k** buckets and sorted individually.
If data is uniformly distributed, sorting is fast.
Worst case occurs when all elements fall into one bucket.

Space Complexity: O(n + k)

Extra space is required for buckets.

5. MAX HEAP

CODE:

```
//HEAP MAX
#include <stdio.h>
void maxHeapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        maxHeapify(arr, n, largest);
    }
}
void heapSortMax(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        maxHeapify(arr, i, 0);
    }
}
```

```c
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]);
    heapSortMax(arr, n);
    printf("\nSorted in Ascending Order (Max Heap): ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT:

```
chetan@amma07:~$ gcc 5.c -o 5
chetan@amma07:~$ ./5
Enter number of elements: 5
Enter 5 elements:
2 3 1 5 4

Sorted in Ascending Order (Max Heap): 1 2 3 4 5 chetan@amma07:~$
```

6. MIN HEAP

CODE:

```c
//MIN HEAP
#include <stdio.h>
void minHeapify(int arr[], int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] < arr[smallest])
        smallest = left;
    if (right < n && arr[right] < arr[smallest])
        smallest = right;
    if (smallest != i) {
        int temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
        minHeapify(arr, n, smallest);
    }
}
void heapSortMin(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--)
        minHeapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        // Move smallest to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        minHeapify(arr, i, 0);
    }
}
```

```c
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]);
    heapSortMin(arr, n);
    printf("\nSorted in Descending Order (Min Heap): ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT:

```
chetan@amma07:~$ gcc 6.c -o 6
chetan@amma07:~$ ./6
Enter number of elements: 5
Enter 5 elements:
2 3 1 5 4

Sorted in Descending Order (Min Heap): 5 4 3 2 1 chetan@amma07:~$
chetan@amma07:~$
```

Time Complexity:

- Best, Average, Worst: **O(n log n)**

Justification:

Building a heap takes **O(n)** time.
Each deletion or insertion takes **O(log n)** and is done **n** times.
So total time becomes **O(n log n)**.

Space Complexity: O(1)

Heap sort is in-place when implemented using arrays.

7. BFS

CODE:

```c
//BFS
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int queue[MAX], front = 0, rear = 0;
void enqueue(int x) {
    queue[rear++] = x;
}
int dequeue() {
    return queue[front++];
}
void BFS(int graph[][MAX], int visited[], int n, int start) {
    enqueue(start);
    visited[start] = 1;
    while(front != rear) {
        int node = dequeue();
        printf("%d ", node);
        for(int i = 0; i < n; i++) {
            if(graph[node][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(i);
            }
        }
    }
}
```

```c
int main() {
    int graph[MAX][MAX] = {
        {0,1,1,0},
        {1,0,1,1},
        {1,1,0,1},
        {0,1,1,0}
    };
    int visited[MAX] = {0};
    BFS(graph, visited, 4, 0);
    return 0;
}
```

OUTPUT:

```
chetan@amma07:~$ gcc 7.c -o 7
chetan@amma07:~$ ./7
0 1 2 3 chetan@amma07:~$
chetan@amma07:~$
```

Time Complexity: O(V + E)

Justification:

Each vertex is visited once and each edge is explored once.

Queue operations take constant time.

Thus, total time depends on vertices and edges.

Space Complexity: O(V)

Queue and visited array store vertices.

8. DFS

CODE:

```c
//DFS
#include <stdio.h>
void DFS(int graph[][4], int visited[], int node) {
    visited[node] = 1;
    printf("%d ", node);
    for(int i = 0; i < 4; i++) {
        if(graph[node][i] == 1 && !visited[i])
            DFS(graph, visited, i);
    }
}
int main() {
    int graph[4][4] = {
        {0,1,1,0},
        {1,0,1,1},
        {1,1,0,1},
        {0,1,1,0}
    };
    int visited[4] = {0};
    DFS(graph, visited, 0);
    return 0;
}
```

OUTPUT:

```
chetan@amma07:~$ gcc 8.c -o 8
chetan@amma07:~$ ./8
0 1 2 3 chetan@amma07:~$
chetan@amma07:~$
```

Time Complexity: O(V + E)

Justification:

Each vertex and edge is visited exactly once.

Recursive or stack-based traversal explores depth-wise.
Hence, time depends on total nodes and edges.

Space Complexity: O(V)
Stack (recursive or explicit) and visited array require extra space.