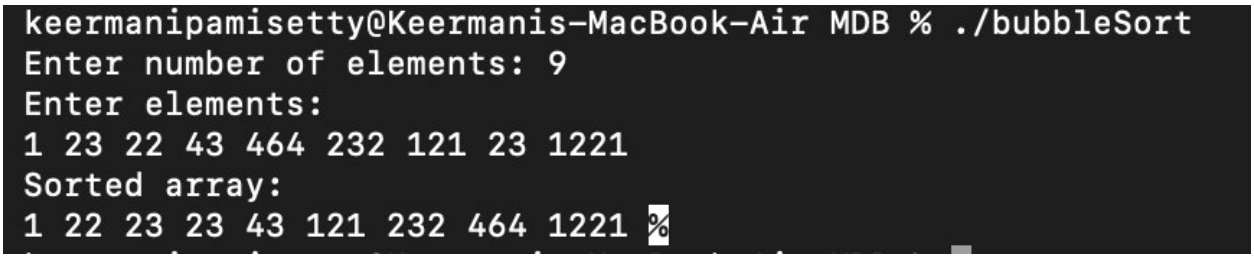
1. Bubble Sort

Code:

```c
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;

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

int main()
{
    int arr[100], n, i;

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

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

    bubbleSort(arr, n);

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

    return 0;
}
```

Output:



Time Complexity:

Bubble sort usually takes $O(n2)$ time because it compares and swaps many pairs of elements again and again, like checking almost every pair in the list.
In the best case, when the list is already sorted and the algorithm stops as soon as it sees no swaps in a pass, it only goes through the list once, so it takes $O(n)$ time.

2. Insertion Sort

Code:

```c
#include <stdio.h>

void insertionSort(int arr[], int n)
{
    int i, key, j;

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

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

        arr[j + 1] = key;
    }
}

int main()
{
    int arr[100], n, i;

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

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

    insertionSort(arr, n);

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

    return 0;
}
```

Output:

```
keermanipamisetty@Keermanis-MacBook-Air MDB % clang insertionSort.c -o insertion
Sort
keermanipamisetty@Keermanis-MacBook-Air MDB % ./insertionSort
Enter number of elements: 6
Enter elements:
1 2 34 31 23 12
Sorted array:
1 2 12 23 31 34 %
```

Time Complexity:

Insertion sort works by taking one element at a time and inserting it into its correct position in the already sorted part of the array. In the **best case**, when the array is already sorted, each element is compared only once and no shifting is needed, so the time complexity is **O(n)**. In the **average case**, the elements are in random order, so for each element, several comparisons and shifts are required, which results in **$O(n^2)$** time. In the **worst case**, when the array is sorted in reverse order, every new element has to be compared with all previous elements and shifted, leading to **$O(n^2)$** time complexity.

3. Selection Sort

Code:

```c
#include <stdio.h>

void selectionSort(int arr[], int n)
{
    int i, j, minIndex, temp;

    for (i = 0; i < n - 1; i++)
    {
        minIndex = i;

        for (j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }

        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

int main()
{
    int arr[100], n, i;

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

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

    selectionSort(arr, n);

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

    return 0;
}
```

Output:

```
keermanipamisetty@Keermanis-MacBook-Air MDB % clang selectionSort.c -o selection
Sort
keermanipamisetty@Keermanis-MacBook-Air MDB % ./selectionSort
Enter number of elements: 6
Enter elements:
1 2 3 43 22 11
Sorted array:
1 2 3 11 22 43 %
```

Time Complexity:

Selection sort repeatedly finds the smallest element from the unsorted part of the array and places it in its correct position. In **every case**, the algorithm must compare each element with all remaining elements to find the minimum, even if the array is already sorted. Because of these repeated comparisons, the **best, average, and worst case time complexity of selection sort is $O(n^2)$**. The number of swaps is small, but the number of comparisons remains the same for all inputs.

4. Bucket Sort

Code:

```c
#include <stdio.h>

void insertionSort(float bucket[], int n)
{
    int i, j;
    float key;

    for (i = 1; i < n; i++)
    {
        key = bucket[i];
        j = i - 1;

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

void bucketSort(float arr[], int n)
{
    int i, j;
    float bucket[10][10];
    int count[10] = {0};

    for (i = 0; i < n; i++)
    {
        int index = arr[i] * 10;
        bucket[index][count[index]++] = arr[i];
    }
```

```c
    for (i = 0; i < 10; i++)
        insertionSort(bucket[i], count[i]);

    int k = 0;
    for (i = 0; i < 10; i++)
        for (j = 0; j < count[i]; j++)
            arr[k++] = bucket[i][j];
}

int main()
{
    int n, i;
    float arr[100];

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

    printf("Enter elements (0 to 1 range):\n");
    for (i = 0; i < n; i++)
        scanf("%f", &arr[i]);

    bucketSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++)
        printf("%.2f ", arr[i]);

    return 0;
}
```

Output:

```
[keermanipamisetty@Keermanis-MacBook-Air MDB % clang bucketSort.c -o bucketSort
[keermanipamisetty@Keermanis-MacBook-Air MDB % ./bucketSort
Enter number of elements: 6
Enter elements (0 to 1 range):
0.15 0.19 0.11 0.8 0.6 0.45
Sorted array:
0.11 0.15 0.19 0.45 0.60 0.80 %
```

Time Complexity:

Bucket sort works by distributing elements into several buckets and then sorting each bucket individually. In the **best and average case**, when the elements are uniformly distributed among the buckets, each bucket contains only a few elements, so sorting inside the buckets takes very little time and the overall time complexity becomes **O(n + k)**, where *n* is the number of elements and *k* is the number of buckets. In the **worst case**, if all elements fall into a single bucket, that bucket has to be sorted using another algorithm (like insertion sort), which leads to a **time complexity of $O(n^2)$**.

5. Heap sort

Code:

Max Heap:

```c
#include <stdio.h>

void maxHeapify(int arr[], int n, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int temp;

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

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

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

        maxHeapify(arr, n, largest);
    }
}

void heapSortMax(int arr[], int n)
{
    int i, temp;

    for (i = n / 2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);
```

```c
    for (i = n - 1; i > 0; i--)
    {
        temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        maxHeapify(arr, i, 0);
    }
}

int main()
{
    int arr[100], n, i;

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

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

    heapSortMax(arr, n);

    printf("Sorted (Ascending using Max Heap):\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
keermanipamisetty@Keermanis-MacBook-Air MDB % clang heap.c -o heap
keermanipamisetty@Keermanis-MacBook-Air MDB % ./heap
Enter number of elements: 5
Enter elements:
23 45 21 19 100
Sorted (Ascending using Max Heap):
19 21 23 45 100 %
```

Min Heap:

```c
#include <stdio.h>

void minHeapify(int arr[], int n, int i)
{
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int temp;

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

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

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

        minHeapify(arr, n, smallest);
    }
}

void heapSortMin(int arr[], int n)
{
    int i, temp;

    for (i = n / 2 - 1; i >= 0; i--)
        minHeapify(arr, n, i);
```

```c
    for (i = n - 1; i > 0; i--)
    {
        temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        minHeapify(arr, i, 0);
    }
}

int main()
{
    int arr[100], n, i;

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

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

    heapSortMin(arr, n);

    printf("Sorted (Descending using Min Heap):\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
keermanipamisetty@Keermanis-MacBook-Air MDB % clang heap.c -o heap
keermanipamisetty@Keermanis-MacBook-Air MDB % ./heap
Enter number of elements: 5
Enter elements:
23 45 21 19 100
Sorted (Descending using Min Heap):
100 45 23 21 19 %
```

Time Complexity:

In both **max heap** and **min heap**, the operations work in a similar way because their structure is the same; only the ordering condition differs. **Building a heap** from an array takes **O(n)** time. **Insertion** of an element takes **O(log n)** time because the element may move up the height of the heap. **Deletion (extract max or extract min)** also takes **O(log n)** time since the heap needs to be heapified from the root to maintain the heap property. Therefore, **max heap and min heap have the same time complexities for all operations**.

6. BFS

Code:

```c
#include <stdio.h>

int visited[10] = {0};
int queue[10];
int front = -1, rear = -1;
void bfs(int graph[10][10], int n, int start)
{
    int i;
    visited[start] = 1;
    queue[++rear] = start;

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

        for (i = 0; i < n; i++)
        {
            if (graph[current][i] == 1 && visited[i] == 0)
            {
                visited[i] = 1;
                queue[++rear] = i;
            }
        }
    }
}
int main()
{
    int n = 5;
    int start = 0;

    int graph[10][10] =
    {
        {0, 1, 1, 0, 0},
        {1, 0, 0, 1, 1},
        {1, 0, 0, 0, 0},
        {0, 1, 0, 0, 0},
        {0, 1, 0, 0, 0}
    };

    printf("BFS Traversal starting from vertex %d:\n", start);
    bfs(graph, n, start);

    return 0;
}
```

Output:

```
[keermanipamisetty@Keermanis-MacBook-Air MDB % clang bfs.c -o bfs
[keermanipamisetty@Keermanis-MacBook-Air MDB % ./bfs
 BFS Traversal starting from vertex 0:
 0 1 2 3 4 %
```

Time Complexity:

In BFS, each vertex is visited once and all its adjacent vertices are checked. When BFS is implemented using an **adjacency matrix**, for every vertex we scan all other

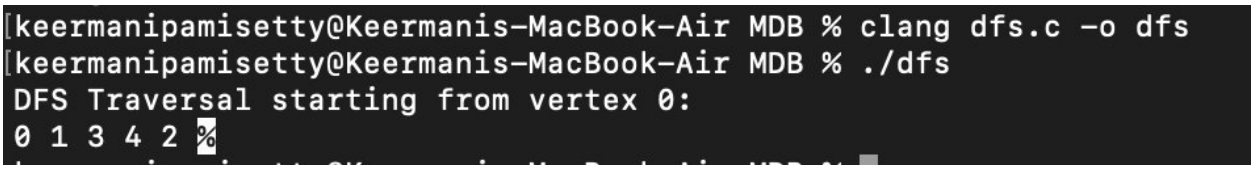
vertices to check for edges, so the time complexity is **O(V^2)**. When BFS is implemented using an **adjacency list**, each vertex and each edge is processed only once, giving a time complexity of **O(V + E)**, where *V* is the number of vertices and *E* is the number of edges.

7. DFS

Code:

```c
#include <stdio.h>
int visited[10] = {0};
void dfs(int graph[10][10], int n, int vertex)
{
    int i;
    visited[vertex] = 1;
    printf("%d ", vertex);

    for (i = 0; i < n; i++)
    {
        if (graph[vertex][i] == 1 && visited[i] == 0)
        {
            dfs(graph, n, i);
        }
    }
}

int main()
{
    int n = 5;
    int start = 0;

    int graph[10][10] =
    {
        {0, 1, 1, 0, 0},
        {1, 0, 0, 1, 1},
        {1, 0, 0, 0, 0},
        {0, 1, 0, 0, 0},
        {0, 1, 0, 0, 0}
    };

    printf("DFS Traversal starting from vertex %d:\n", start);
    dfs(graph, n, start);

    return 0;
}
```

Output:



Time Complexity:

In DFS, each vertex is visited once, and all edges from that vertex are explored recursively. When implemented using an **adjacency matrix**, for each vertex we check all other vertices for edges, resulting in a time complexity of **$O(V^2)$**. When implemented using an **adjacency list**, each vertex and each edge is processed only once, giving a time complexity of **$O(V + E)$**, where *V* is the number of vertices and *E* is the number of edges.