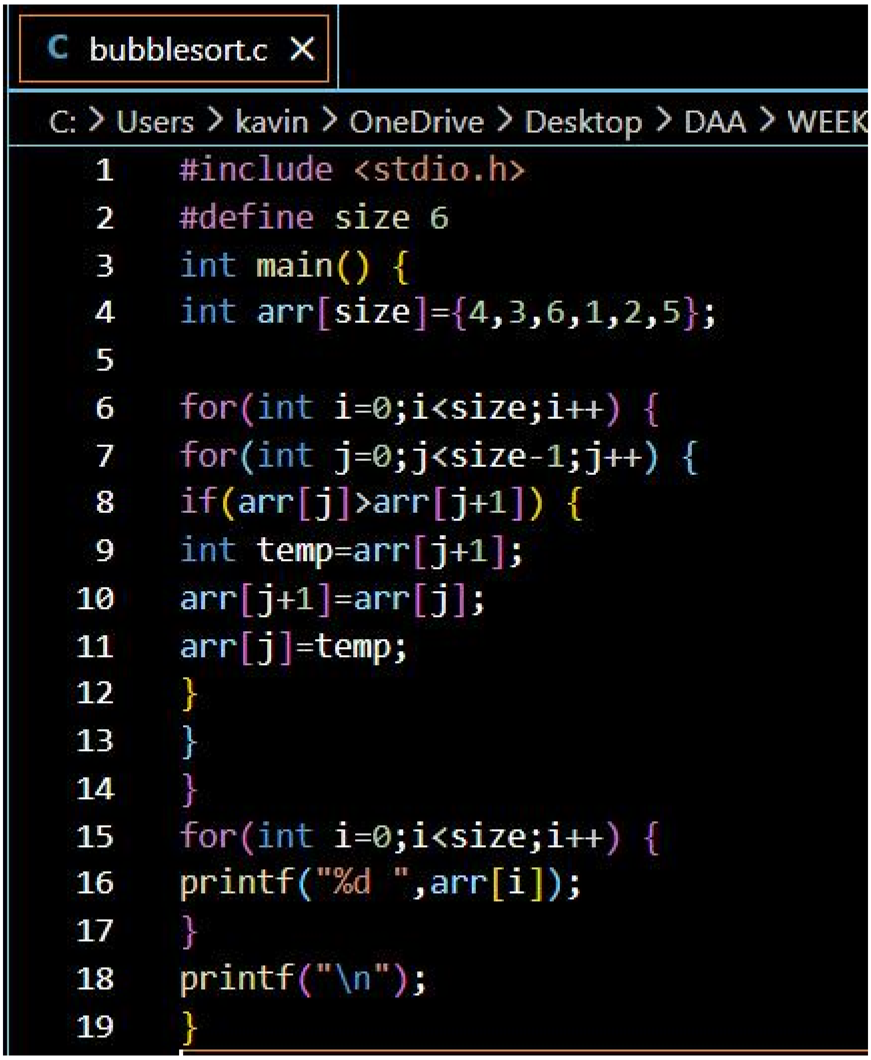
WEEK - 2

Kavin.J.S
CH.SC.U4CSE24119

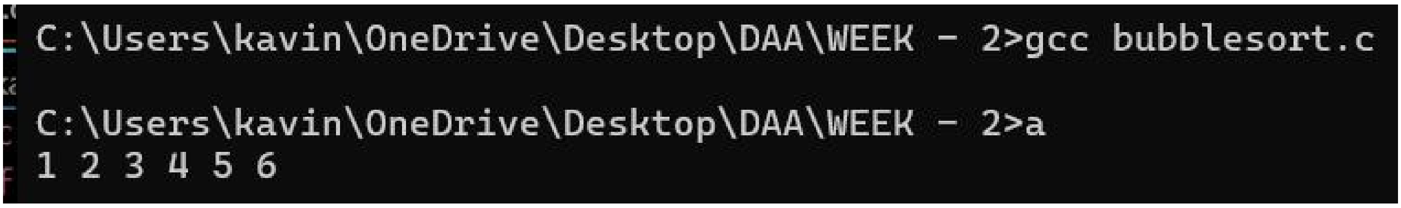
Bubble Sort:

Code:



```c
#include <stdio.h>
#define size 6
int main() {
int arr[size]={4,3,6,1,2,5};

for(int i=0;i<size;i++) {
for(int j=0;j<size-1;j++) {
if(arr[j]>arr[j+1]) {
int temp=arr[j+1];
arr[j+1]=arr[j];
arr[j]=temp;
}
}
}
for(int i=0;i<size;i++) {
printf("%d ",arr[i]);
}
printf("\n");
}
```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc bubblesort.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
1 2 3 4 5 6
```

Time and Space Complexity:

Time Complexity:

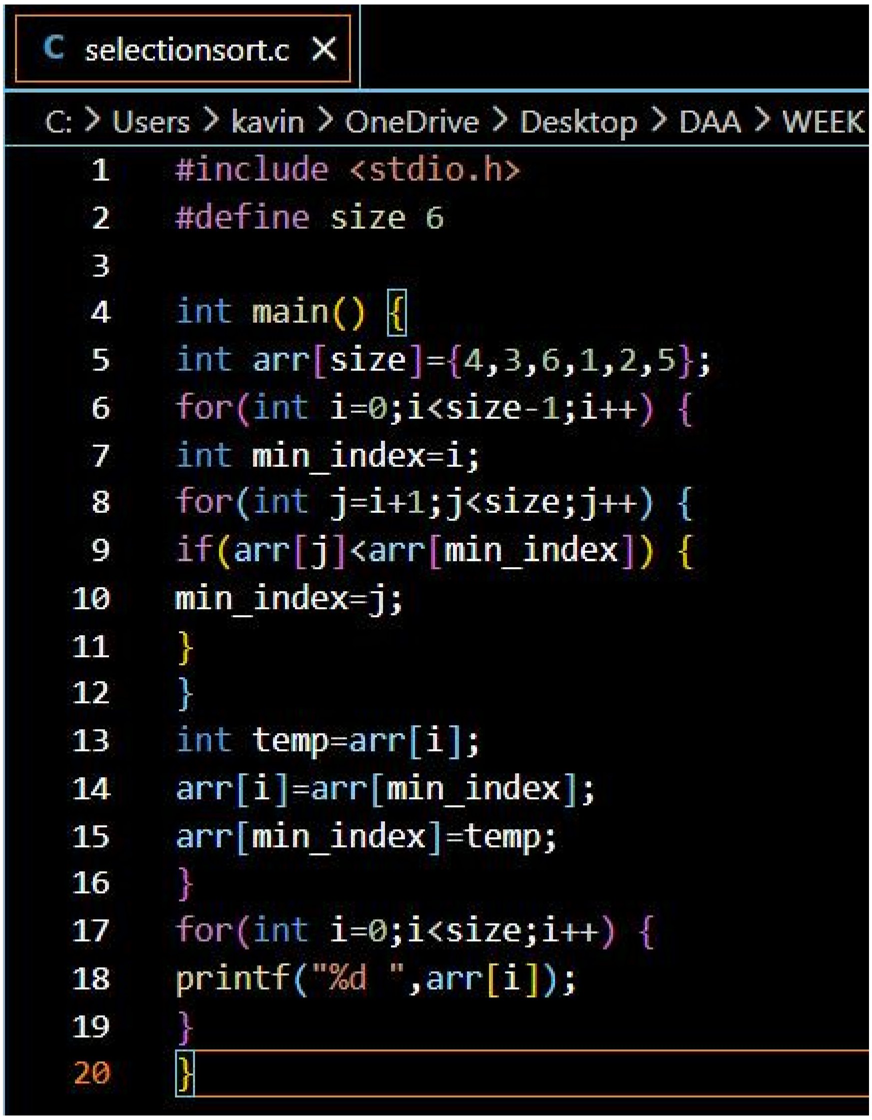
The outer loop runs for n times, while the inner loop runs for n-1 times, So the time taken will be n^2-n, from which we can get,
The time complexity = O(n^2)

Space Complexity:

Only one extra variable is used for swapping(Temp)
No additional arrays or dynamic memomry used
Sorting is done in the same array,
Hence, Space complexity = O(1)

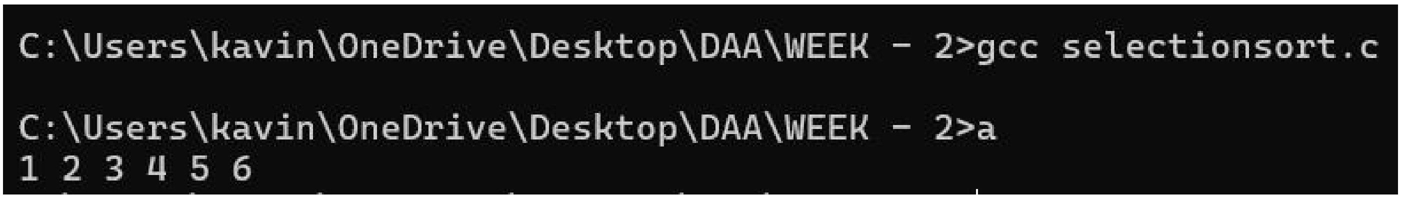
Selection Sort:

Code:



```c
#include <stdio.h>
#define size 6

int main() {
int arr[size]={4,3,6,1,2,5};
for(int i=0;i<size-1;i++) {
int min_index=i;
for(int j=i+1;j<size;j++) {
if(arr[j]<arr[min_index]) {
min_index=j;
}
}
int temp=arr[i];
arr[i]=arr[min_index];
arr[min_index]=temp;
}
for(int i=0;i<size;i++) {
printf("%d ",arr[i]);
}
}
```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc selectionsort.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
1 2 3 4 5 6
```

Time and Space Complexity:

Time Complexity:

The outer loop runs for n-1 times, while the inner loop runs for n-i times for each i ,
Time taken = (n-1)+(n-2)+(n-3)+(n-4)+…….+1 = n(n-1)/2

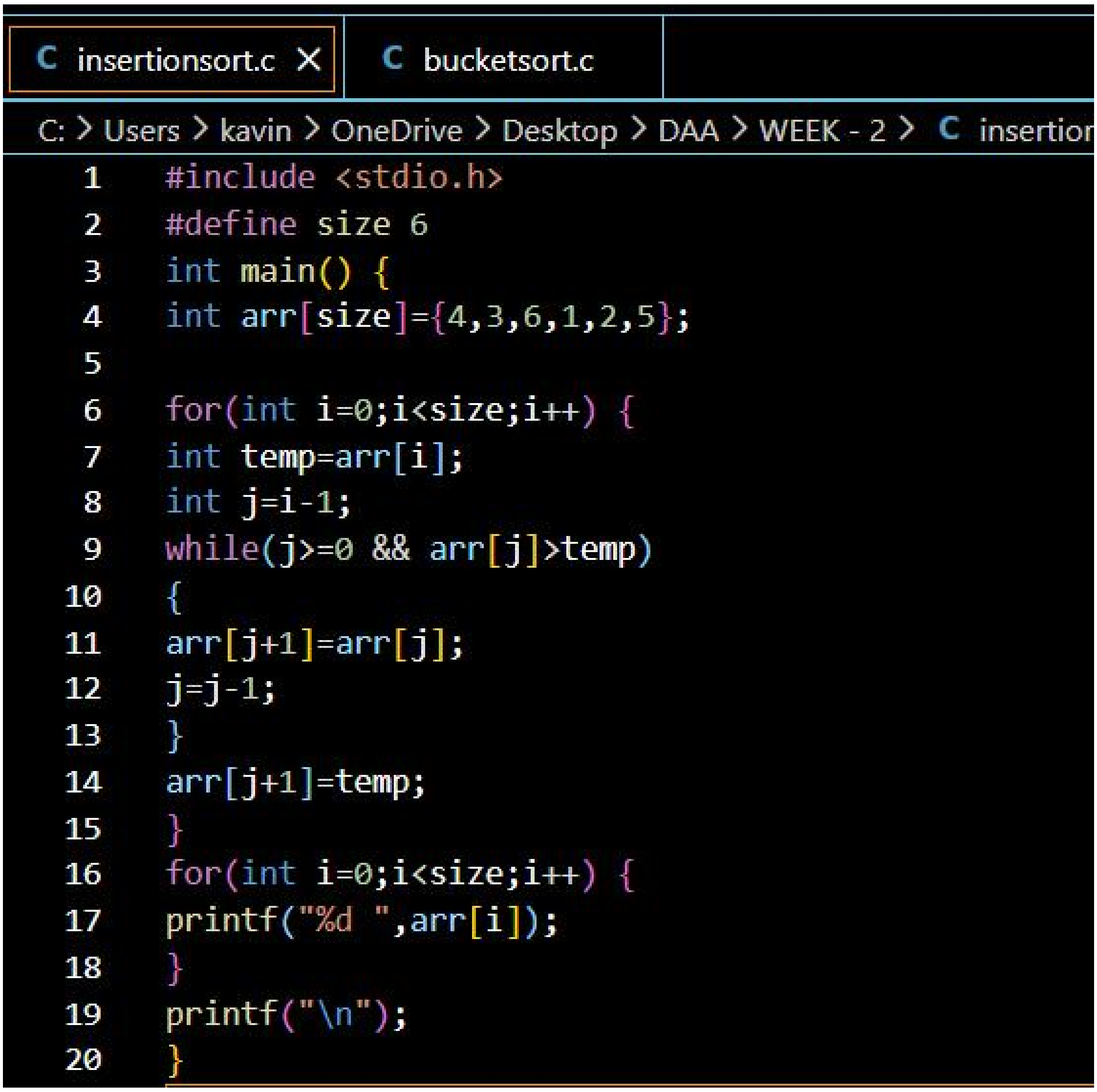
So, Time complexity = O(n^2)

Space Complexity:

Only two constant extra variable(temp, j) are used
No additional arrays or dynamic memomry used
Sorting is done in the same array,
Hence, Space complexity = O(1)

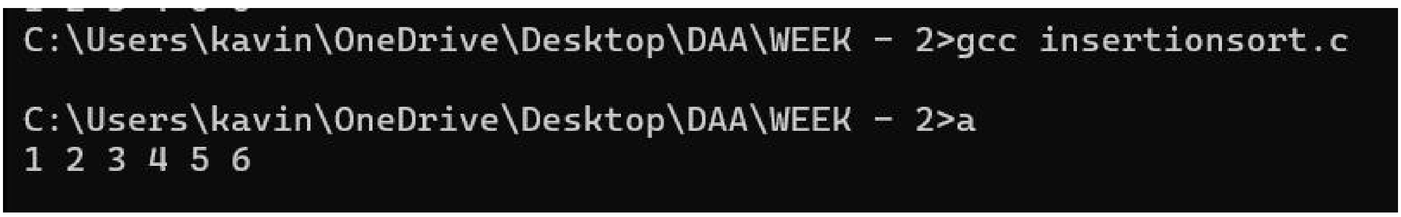
Insertion Sort:

Code:


insertionsort.c
bucketsort.c
C: > Users > kavin > OneDrive > Desktop > DAA > WEEK - 2 > insertior

```c
#include <stdio.h>
#define size 6
int main() {
int arr[size]={4,3,6,1,2,5};

for(int i=0;i<size;i++) {
int temp=arr[i];
int j=i-1;
while(j>=0 && arr[j]>temp)
{
arr[j+1]=arr[j];
j=j-1;
}
arr[j+1]=temp;
}
for(int i=0;i<size;i++) {
printf("%d ",arr[i]);
}
printf("\n");
}
```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc insertionsort.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
1 2 3 4 5 6
```

Time and Space Complexity:

Time Complexity:

The outer loop runs for n times and the inner while loop runs for n times for worst case scenario and 1 times for best case scenario

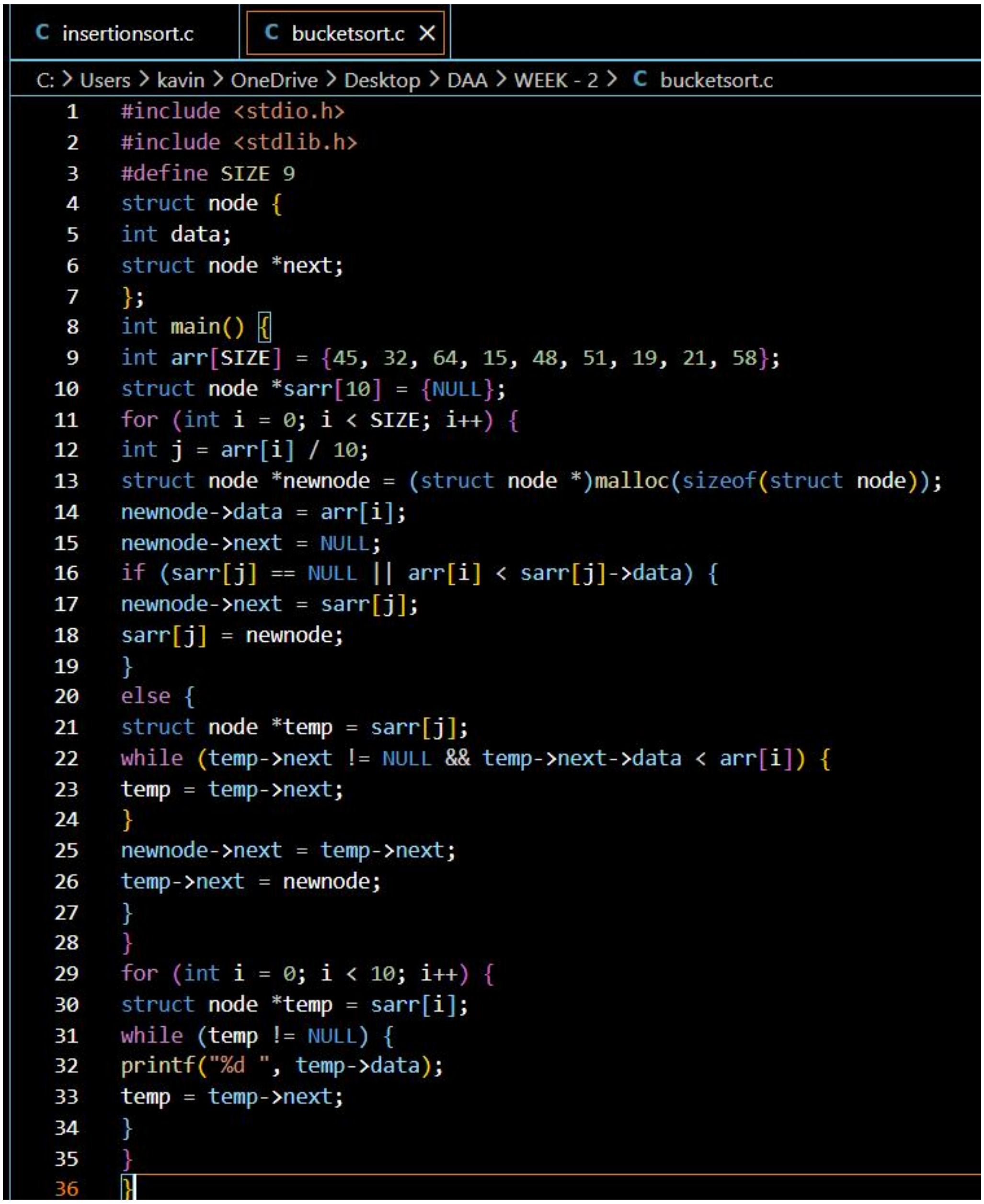
Time taken for worst case scenario = n^2
Time taken for best case scenario = n
So,
For worst case, Time complexity = O(n^2)
For best case, Time complexity = O(n)

Space Complexity:

Only two constant extra variable(temp, j) are used
No additional arrays or dynamic memomry used
Sorting is done in the same array,
Hence, Space complexity = O(1)

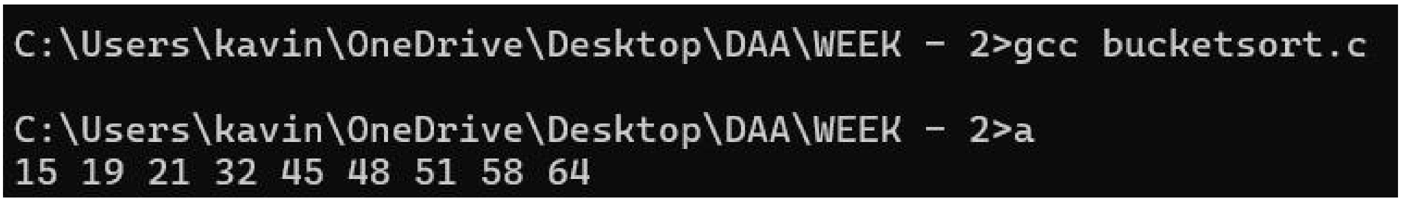
Bucket Sort:

Code:


C insertionsort.c
C bucketsort.c ×
C: > Users > kavin > OneDrive > Desktop > DAA > WEEK - 2 > C bucketsort.c

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 9
struct node {
int data;
struct node *next;
};
int main() {
int arr[SIZE] = {45, 32, 64, 15, 48, 51, 19, 21, 58};
struct node *sarr[10] = {NULL};
for (int i = 0; i < SIZE; i++) {
int j = arr[i] / 10;
struct node *newnode = (struct node *)malloc(sizeof(struct node));
newnode->data = arr[i];
newnode->next = NULL;
if (sarr[j] == NULL || arr[i] < sarr[j]->data) {
newnode->next = sarr[j];
sarr[j] = newnode;
}
else {
struct node *temp = sarr[j];
while (temp->next != NULL && temp->next->data < arr[i]) {
temp = temp->next;
}
newnode->next = temp->next;
temp->next = newnode;
}
}
for (int i = 0; i < 10; i++) {
struct node *temp = sarr[i];
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
}
}
```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc bucketsort.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
15 19 21 32 45 48 51 58 64
```

Time and Space Complexity:

Time Complexity:

The outer loop runs for n times

Best case scenario, bucket is empty while inserting for every element so, insertion takes O(1)

Worst case scenario, all elements fall into the same bucket, so each element have the traverse the list, making its n times for worst case

Hence,

Time Complexity for best case = O(n)

Time Complexity for worst case = O(n^2)

Space Complexity:

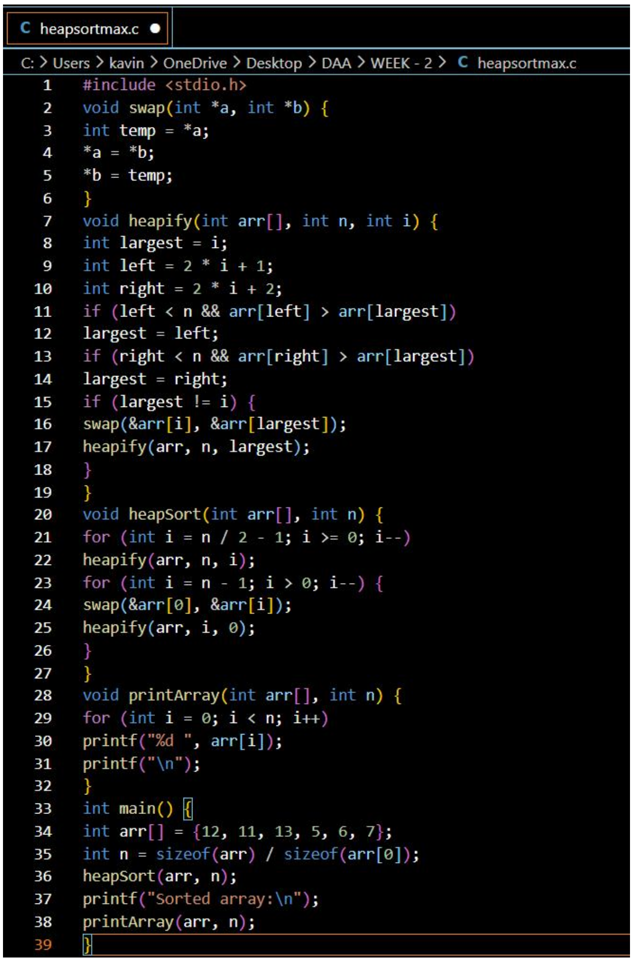
Bucket array uses constant space

n linked list nodes used for every element

So, Space Complexity = O(n)

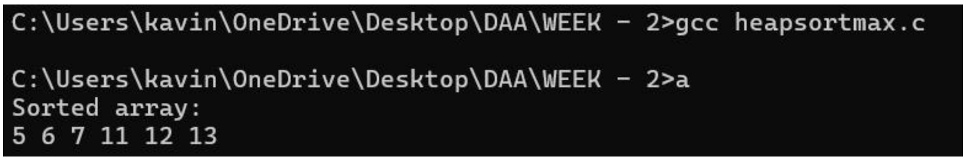
Heap Sort Using Max Heap:

Code:


heapsortmax.c
C: > Users > kavin > OneDrive > Desktop > DAA > WEEK - 2 > heapsortmax.c

```c
#include <stdio.h>
void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}
void heapify(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) {
swap(&arr[i], &arr[largest]);
heapify(arr, n, largest);
}
}
void heapSort(int arr[], int n) {
for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);
for (int i = n - 1; i > 0; i--) {
swap(&arr[0], &arr[i]);
heapify(arr, i, 0);
}
}
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");
}
int main() {
int arr[] = {12, 11, 13, 5, 6, 7};
int n = sizeof(arr) / sizeof(arr[0]);
heapSort(arr, n);
printf("Sorted array:\n");
printArray(arr, n);
}
```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc heapsortmax.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
Sorted array:
5 6 7 11 12 13
```

Time and Space Complexity:

Time Complexity:

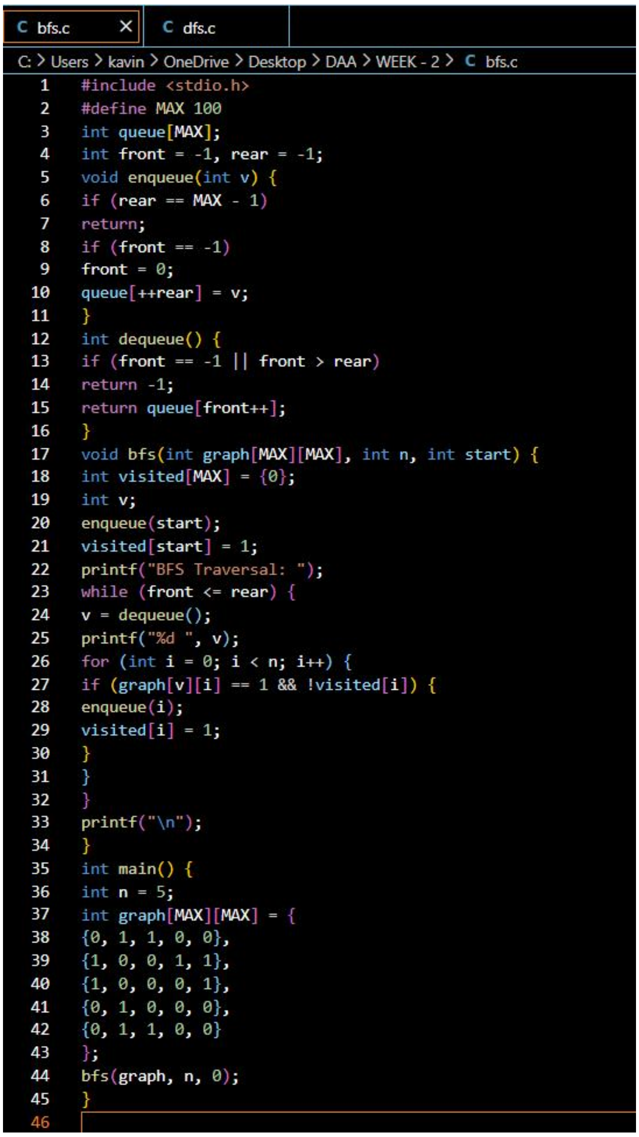
Heapify() on a single node takes O(log n)
Extracting elements from heap runs for n-1 times
Hence, Time Complexity = O(nlog n)

Space Complexity:

Sorting is done in the same array
Recursive calls in heapify() for log n times
Hence, Space Complexity = O(log n)

Breath First Search:

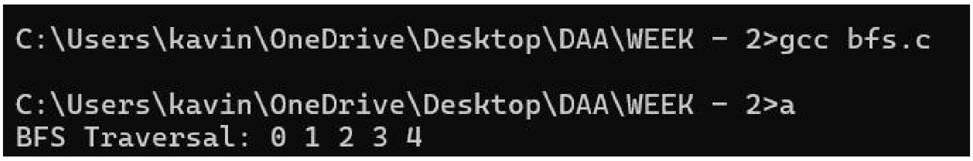
Code:



C bfs.c ×
C dfs.c
C: > Users > kavin > OneDrive > Desktop > DAA > WEEK - 2 > C bfs.c

```c
#include <stdio.h>
#define MAX 100
int queue[MAX];
int front = -1, rear = -1;
void enqueue(int v) {
if (rear == MAX - 1)
return;
if (front == -1)
front = 0;
queue[++rear] = v;
}
int dequeue() {
if (front == -1 || front > rear)
return -1;
return queue[front++];
}
void bfs(int graph[MAX][MAX], int n, int start) {
int visited[MAX] = {0};
int v;
enqueue(start);
visited[start] = 1;
printf("BFS Traversal: ");
while (front <= rear) {
v = dequeue();
printf("%d ", v);
for (int i = 0; i < n; i++) {
if (graph[v][i] == 1 && !visited[i]) {
enqueue(i);
visited[i] = 1;
}
}
}
printf("\n");
}
int main() {
int n = 5;
int graph[MAX][MAX] = {
{0, 1, 1, 0, 0},
{1, 0, 0, 1, 1},
{1, 0, 0, 0, 1},
{0, 1, 0, 0, 0},
{0, 1, 1, 0, 0}
};
bfs(graph, n, 0);
}

```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc bfs.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
BFS Traversal: 0 1 2 3 4
```

Time and Space Complexity:

Time Complexity:

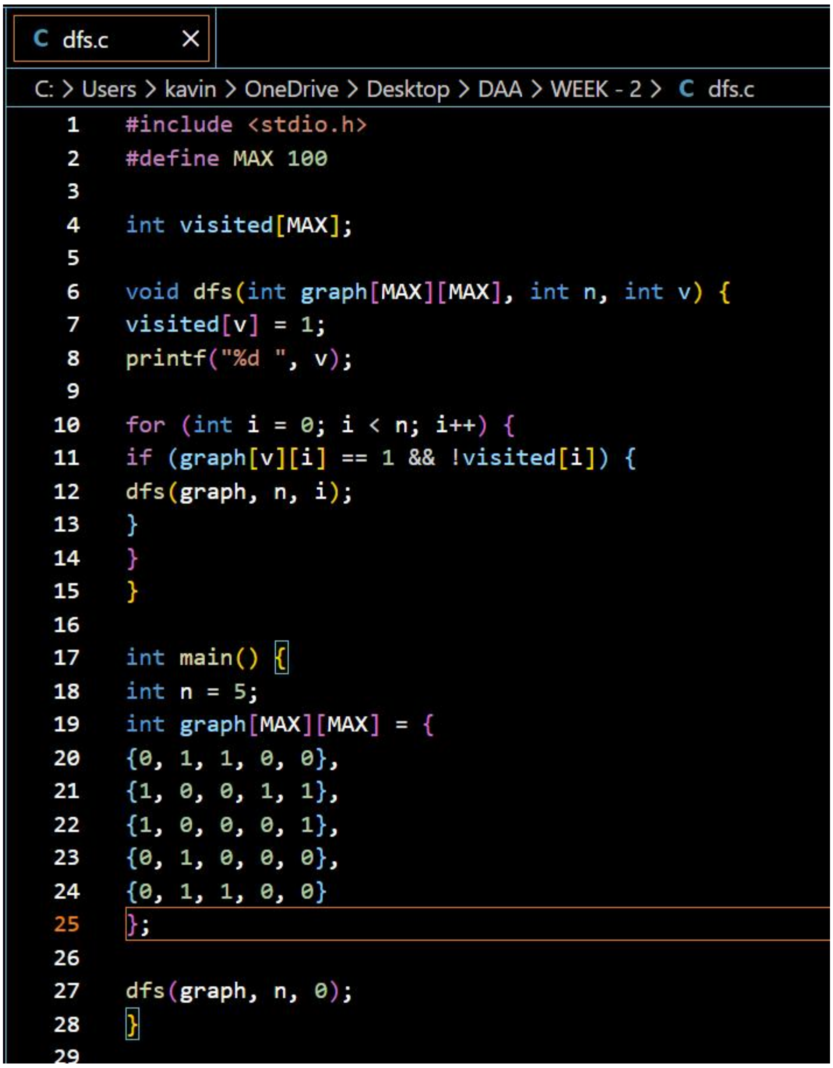
Each vertex is enqueued and dequeued one time, so overall n times
For each dequeued vertex all vertices are scaned, so overall n times
Hence, Time Complexity = O(n^2)

Space Complexity:

Graph storage takes n^2 space
Queue take n space
Visited array takes n space
So, Space Complexity = O(n^2)

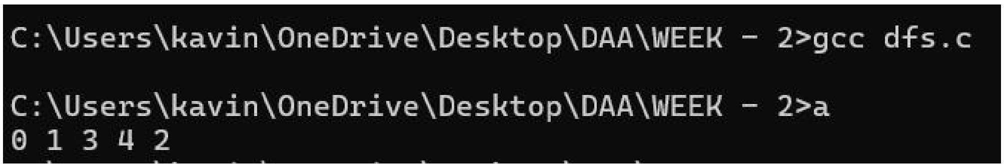
Depth First Search:

Code:


C dfs.c
C: > Users > kavin > OneDrive > Desktop > DAA > WEEK - 2 > C dfs.c

```c
#include <stdio.h>
#define MAX 100

int visited[MAX];

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

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

int main() {
int n = 5;
int graph[MAX][MAX] = {
{0, 1, 1, 0, 0},
{1, 0, 0, 1, 1},
{1, 0, 0, 0, 1},
{0, 1, 0, 0, 0},
{0, 1, 1, 0, 0}
};

dfs(graph, n, 0);
}
```

Output:



```
C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>gcc dfs.c

C:\Users\kavin\OneDrive\Desktop\DAA\WEEK - 2>a
0 1 3 4 2
```

Time and Space Complexity:

Time Complexity:

Each vertex is visited exactly once
For every vertex visited check all the vertex to find the adjacent vertex
Hence, Time Complexity = O(n^2)

Space Complexity:

Graph storage takes n^2 space
Visited array takes n space
Recursion stack takes n space
So, Space Complexity = O(n^2)