AMRITA VISHWA VIDYAPEETHAM

Amrita School of Computing

23CSE211

Design and Analysis of Algorithms

WEEK – 2 (04 December 2025)

Susendran M

CH.SC.U4CSE24154

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF COMPUTING

CHENNAI

1. Bubble Sort

Code :

```
#include <stdio.h>
void bubble_sort(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 the size of the array : ");
scanf("%d",&n);
int arr[n];
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);}
```

```
printf("Array before Bubble sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
bubble_sort(arr,n);
printf("\nArray after Bubble sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
printf("\n");
return 0;
}

// Fixed part = 12 bytes
// Variable part = 4*n bytes
// Total Space complexity = 12 + 4*n bytes
// Space complexity = O(n)
/* Since the variable part comes from arr[], which
grows linearly with input size n */
// Time complexity = O(n^2)
/* The worst case comes from inner loop performs
comparison and swaps *\
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc selection.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
Enter the size of the array : 5
1
3
5
4
2
Array before Selection sort
1       3       5       4       2
Array after Selection sort
1       2       3       4       5
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(n)
Since the variable part comes from arr[], which grows linearly with input size n
Time complexity = O(n^2)
The worst case comes from inner loop performs comparison and swaps

2. Insertion Sort

Code :

```c
#include <stdio.h>
void insertion_sort(int arr[],int n)
{
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;
}
}
int main()
{
int n;
printf("Enter the size of the array : ");
scanf("%d",&n);
int arr[n];
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
```

```c
printf("Array before Insertion sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
insertion_sort(arr,n);
printf("\nArray after Insertion sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
printf("\n");
return 0;
}

// Fixed part = 12 bytes
// Variable part = 4*n bytes
// Total Space complexity = 12 + 4*n bytes
// Space complexity = O(n)
/* Since the variable part comes from arr[],
which grows linearly with input size n */
// Time complexity = O(n^2)
/* The worst case comes from inner loop performs
comparison the element and shifts of each element */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc insertion.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
Enter the size of the array : 5
1
3
5
4
2
Array before Insertion sort
1       3       5       4       2
Array after Insertion sort
1       2       3       4       5
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(n)

Since the variable part comes from arr[], which grows linearly with input size n

Time complexity = O(n^2)

The worst case comes from inner loop performs comparison the element and shifts of each element

3. Selection Sort

Code :

```c
#include <stdio.h>
void selection_sort(int arr[],int n)
{
int min_index,temp;
for(int i=0;i<n-1;i++)
{
min_index=i;
for(int j=i+1;j<n;j++)
{
if(arr[j]<arr[min_index])
{
min_index=j;
}
}
if(min_index != i)
{
temp=arr[i];
arr[i]=arr[min_index];
arr[min_index]=temp;
}
}
}
int main()
{
int n;
```

```
printf("Enter the size of the array : ");
scanf("%d",&n);
int arr[n];
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
printf("Array before Selection sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
selection_sort(arr,n);
printf("\nArray after Selection sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
printf("\n");
return 0;
}
```

```
// Fixed part = 8 bytes
// Variable part = 4*n bytes
// Total Space complexity = 8 + 4*n bytes
// Space complexity = O(n)
/* Since the variable part comes from arr[],
which grows linearly with input size n */
// Time complexity = O(n^2)
/* The worst case comes from inner loop performs
comparison and proportional to the square of the
array size */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc selection.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
Enter the size of the array : 5
1
3
5
4
2
Array before Selection sort
1       3       5       4       2
Array after Selection sort
1       2       3       4       5
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(n)

Since the variable part comes from arr[], which grows linearly with input size n

Time complexity = O(n^2)

The worst case comes from inner loop performs comparison and proportional to the square of the array size

4. Bucket Sort

Code :

```
#include <stdio.h>
#include <stdlib.h>
void bucket_sort(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);
}
int main()
{
int n;
printf("Enter the size of the array : ");
scanf("%d",&n);
int arr[n];
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
printf("Array before Bucket sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
bucket_sort(arr,n);
printf("\nArray after Bucket sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
```

```c
printf("\n");
return 0;
}

// Fixed part = 20 bytes
// Variable part = 4*n + 4(max+1) bytes
// Total Space complexity = 20 + 4*n + 4(max+1) bytes
// Space complexity = O(n+max)
/* Since the variable part comes from arr[] which grows
linearly with input size n and space required to store the
bucket array based on largest value in the input array */
// Time complexity = O(n+max)
/* The worst case comes from both the size of input array
n and the size of bucket array max */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc bucket.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
Enter the size of the array : 5
1
3
5
4
2
Array before Bucket sort
1       3       5       4       2
Array after Bucket sort
1       2       3       4       5
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(n+max)

Since the variable part comes from arr[] which grows linearly with input size n and space required to store the bucket array based on largest value in the input array

Time complexity = O(n+max)

The worst case comes from both the size of input array n and the size of bucket array max

5. Max Heap Sort

Code :

```
#include <stdio.h>
void maxheap(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;
maxheap(arr,n,largest);
}
}
void heap_sort(int arr[],int n)
{
for(int i=n / 2 - 1;i>=0;i--)
```

```
{
maxheap(arr,n,i);
}
for(int i=n-1;i>=0;i--)
{
int temp=arr[0];
arr[0]=arr[i];
arr[i]=temp;
maxheap(arr,i,0);
}
}

int main()
{
int n;
printf("Enter the size of the array : ");
scanf("%d",&n);
int arr[n];
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
printf("Array before Max Heap sort \n");
for(int i=0;i<n;i++)
```

```
{
printf("%d\t",arr[i]);
}
heap_sort(arr,n);
printf("\nArray after Max Heap sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
printf("\n");
return 0;
}

// Space complexity = O(1)
/* Heap sort is in-place when implemented
using arrays */
// Time complexity = O(n log n)
/* The worst case comes from where building a heap
takes O(n) and each deletion or insertion takes
O(log n) and is done n times */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc maxheap.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
Enter the size of the array : 5
1
3
5
4
2
Array before Max Heap sort
1       3       5       4       2
Array after Max Heap sort
1       2       3       4       5
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(1)
Heap sort is in-place when implemented using arrays
Time complexity = O(n log n)
The worst case comes from where building a heap takes O(n) and each deletion or insertion takes O(log n) and is done n times

6. Min Heap Sort

Code :

```c
#include <stdio.h>
void minheap(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;
minheap(arr,n,smallest);
}
}
void heap_sort(int arr[],int n)
{
for(int i=n / 2 - 1;i>=0;i--)
```

```
{
minheap(arr,n,i);
}
for(int i=n-1;i>=0;i--)
{
int temp=arr[0];
arr[0]=arr[i];
arr[i]=temp;
minheap(arr,i,0);
}
}

int main()
{
int n;
printf("Enter the size of the array : ");
scanf("%d",&n);
int arr[n];
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
printf("Array before Min Heap sort \n");
for(int i=0;i<n;i++)
```

```
{
printf("%d\t",arr[i]);
}
heap_sort(arr,n);
printf("\nArray after Min Heap sort \n");
for(int i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
printf("\n");
return 0;
}

// Space complexity = O(1)
/* Heap sort is in-place when implemented
using arrays */
// Time complexity = O(n log n)
/* The worst case comes from where building a heap
takes O(n) and each deletion or insertion takes
O(log n) and is done n times */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc minheap.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
Enter the size of the array : 5
1
3
5
4
2
Array before Min Heap sort
1       3       5       4       2
Array after Min Heap sort
5       4       3       2       1
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(1)
Heap sort is in-place when implemented using arrays
Time complexity = O(n log n)
The worst case comes from where building a heap takes O(n) and each deletion or insertion takes O(log n) and is done n times

7. BFS

Code :

```
#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 \t",node);
for(int i=0;i<n;i++)
{
if(graph[node][i]==1 && !visited[i])
{
visited[i]=1;
```

```
enqueue(i);
}}
}
printf("\n");
}
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;
}

// Space complexity = O(V)
// Queue and visited array store vertices
// Time complexity = O(V+E)
/* The worst case comes from where each vertex
is visited once and each edge is explored once,
Queue operations take constant time */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc bfs.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
0       1       2       3
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(V)

Queue and visited array store vertices

Time complexity = O(V+E)

The worst case comes from where each vertex is visited once and each edge is explored once, Queue operations take constant time

8. DFS

Code :

```c
#include <stdio.h>
void DFS(int graph[][4],int visited[],int node)
{
visited[node]=1;
printf("%d \t",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);
printf("\n");
return 0;
}

// Space complexity = O(V)
// Stack and visited array require extra space
// Time complexity = O(V+E)
/* The worst case comes from where each vertex and
edge is visited exactly once */
```

Output :

```
susendran@susendran-VirtualBox:~/Desktop/DAA$ gcc dfs.c
susendran@susendran-VirtualBox:~/Desktop/DAA$ ./a.out
0       1       2       3
susendran@susendran-VirtualBox:~/Desktop/DAA$
```

Justification :

Space complexity = O(V)

Stack and visited array require extra space

Time complexity = O(V+E)

The worst case comes from where each vertex and edge is visited exactly once