SCHOOL OF COMPUTING

M M VASANTH

CH.SC.U4CSE24227

Week – 2

Date - 05/12/2025

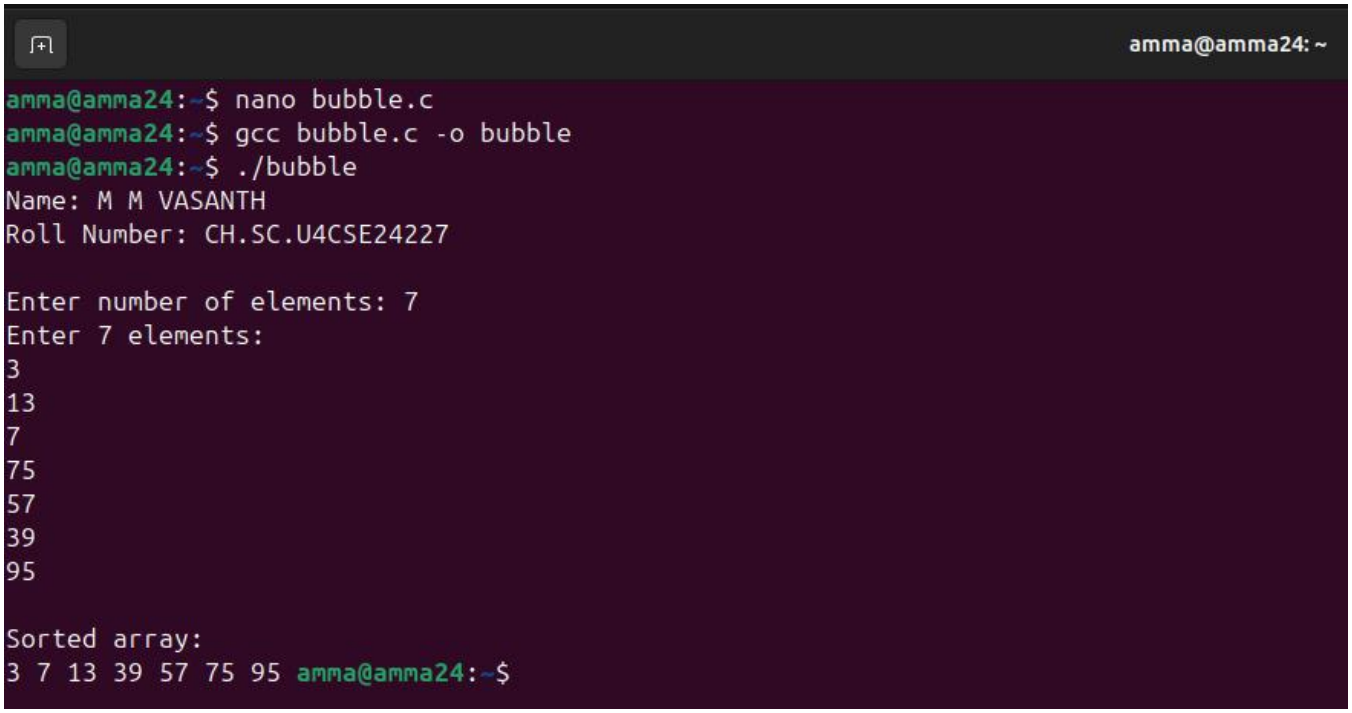
Design and Analysis of Algorithm(23CSE211)

1. BUBBLE SORT

Code:

```
#include <stdio.h>
int main() {
    int a[100], n, i, j, temp;
    printf("Name: M M VASANTH\n");
    printf("Roll Number: CH.SC.U4CSE24227\n\n");
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    for(i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if(a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
    printf("\nSorted array:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

Output:



- **Time Complexity: O(n^2)**

The implementation uses two nested for loops. The outer loop runs n times, and the inner loop runs n-i-1 times. In the worst case (reverse sorted array), this results in a quadratic number of comparisons and swaps.

- **Space Complexity: O(1)**

The sorting happens in-place. I only used a single integer variable temp for swapping, so no extra memory proportional to the input size is required.

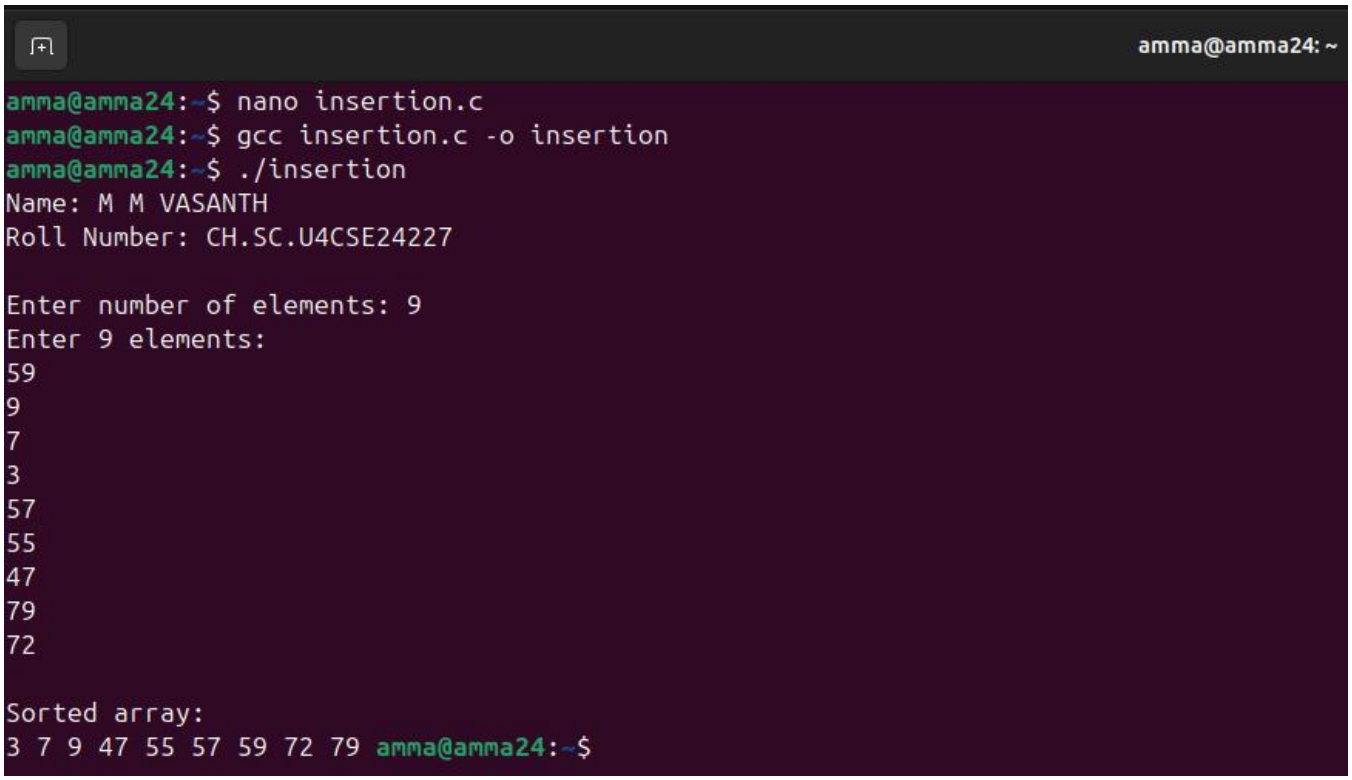
2. INSERTION SORT

Code:

```
#include <stdio.h>
int main() {
    int a[100], n, i, j, key;
    printf("Name: M M VASANTH\n");
    printf("Roll Number: CH.SC.U4CSE24227\n\n");
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    for(i = 1; i < n; i++) {
        key = a[i];
        j = i - 1;
        while(j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
    printf("\nSorted array:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);    }
```

```
    return 0;}
```

Output:



- **Time Complexity: O(n^2)**

The code uses a while loop nested inside a for loop. In the worst case (reverse order), the while loop shifts every element to the right for every iteration of i. However, for nearly sorted data, it performs much faster (close to O(n)).

- **Space Complexity: O(1)**

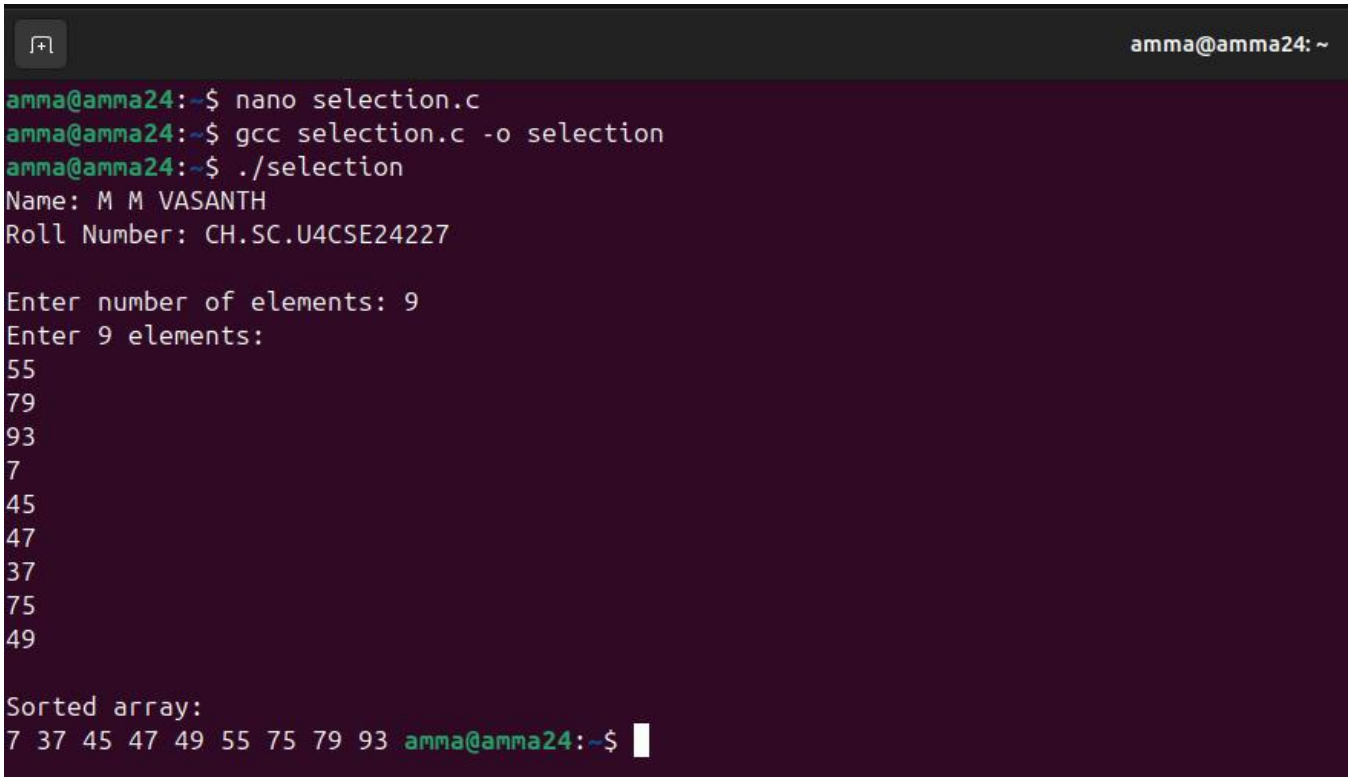
The sorting is performed by shifting elements within the original array. Only a single variable key is used to hold the value being inserted, keeping memory usage constant.

3.SELECTION SORT

Code:

```
#include <stdio.h>
int main() {
    int a[100], n, i, j, minIndex, temp;
    printf("Name: M M VASANTH\n");
    printf("Roll Number: CH.SC.U4CSE24227\n\n");
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    for(i = 0; i < n-1; i++) {
        minIndex = i;
        for(j = i+1; j < n; j++) {
            if(a[j] < a[minIndex]) {
                minIndex = j;
            }
        }
        temp = a[i];
        a[i] = a[minIndex];
        a[minIndex] = temp;
    }
    printf("\nSorted array:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

Output:



- **Time Complexity: O(n^2)**

The logic relies on finding the minimum element in the unsorted subarray for every position i. This requires scanning the remaining n-i elements. Summing these scans results in a quadratic time complexity, regardless of whether the array is already sorted or not.

- **Space Complexity: O(1)**

This is an in-place algorithm. I used variables like minIndex and temp for tracking and swapping, but no additional data structures or arrays were allocated.

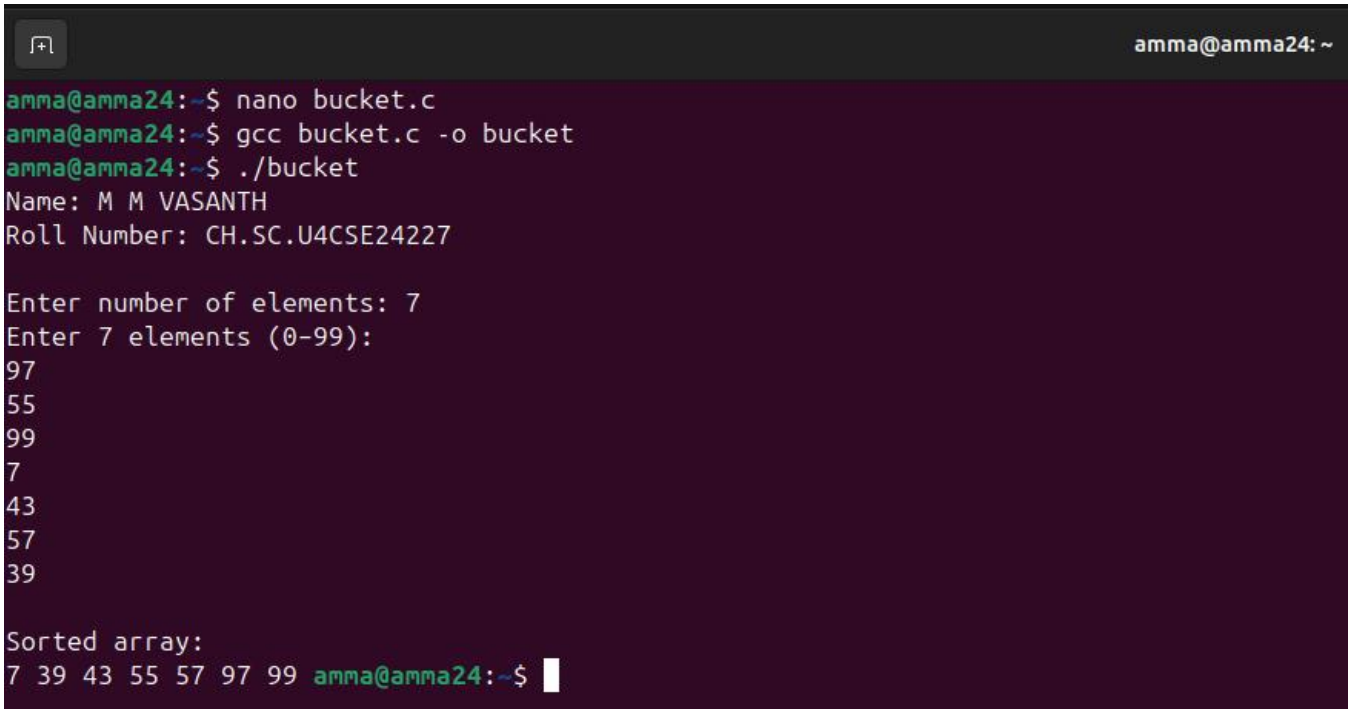
4. BUCKET SORT

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* insertSorted(struct Node* head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL || value < head->data) {
        newNode->next = head;
        return newNode;
    }
    struct Node* temp = head;
    while (temp->next != NULL && temp->next->data < value) {
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    return head;
}
int main() {
    int n, i, j;
    int a[100];
    printf("Name: M M VASANTH\n");
    printf("Roll Number: CH.SC.U4CSE24227\n\n");
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements (0â€“99):\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    struct Node* buckets[10] = { NULL };
    for (i = 0; i < n; i++) {
        int index = a[i] / 10;
        buckets[index] = insertSorted(buckets[index], a[i]);
    }
    printf("\nSorted array:\n");
    for (i = 0; i < 10; i++) {
        struct Node* temp = buckets[i];
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
```

```
        }
    }
    return 0;
}
```

Output:



- **Time Complexity: O(n + k)**

The logic distributes n elements into k buckets (where k=10 in my code). Since the elements are inserted into linked lists in sorted order using insertSorted, the performance depends on the distribution. For uniformly distributed data, this runs in linear average time.

- **Space Complexity: O(n + k)**

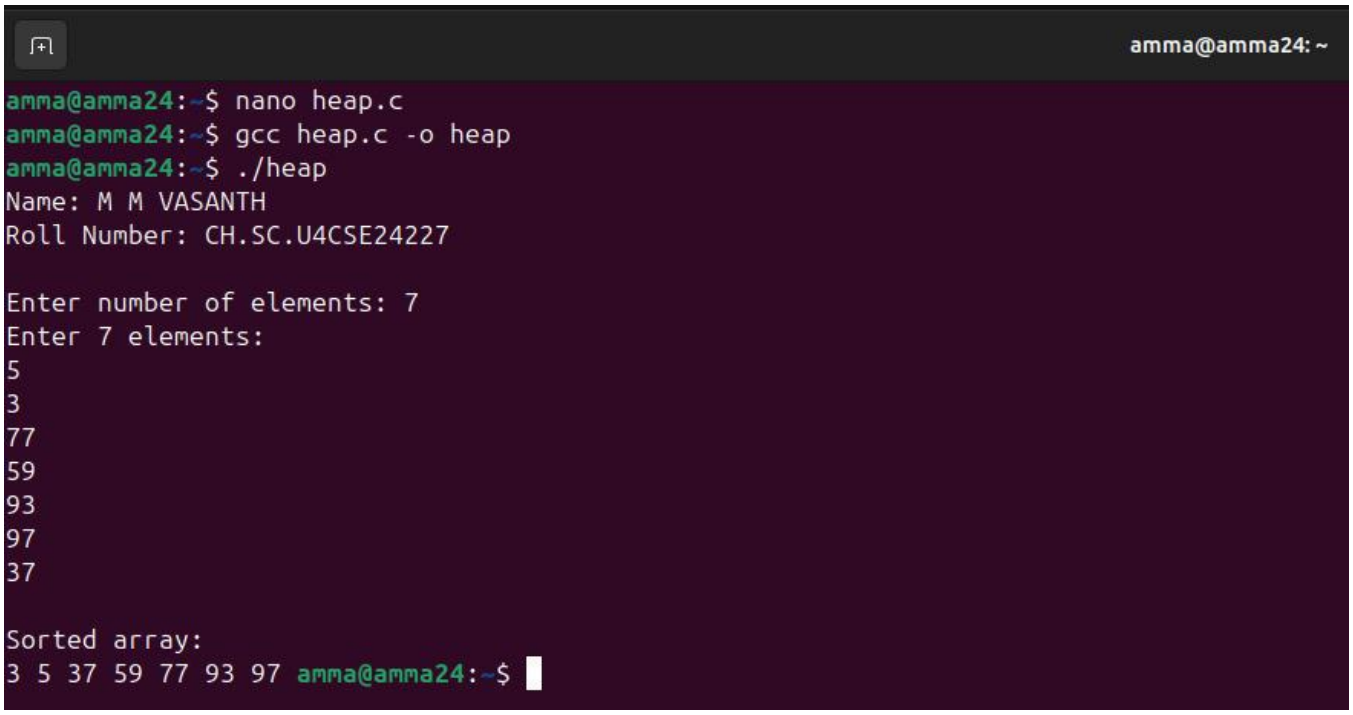
Unlike the other algorithms here, this is not in-place. The code dynamically allocates memory using malloc for every element to store them in linked list nodes, plus the array for the 10 bucket heads.

5. HEAP SORT(MAX HEAP)

Code:

```
#include <stdio.h>
void heapify(int a[], int n, int i) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if(left < n && a[left] > a[largest])
        largest = left;
    if(right < n && a[right] > a[largest])
        largest = right;
    if(largest != i) {
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        heapify(a, n, largest);
    }
}
int main() {
    int a[100], n, i;
    printf("Name: M M VASANTH\n");
    printf("Roll Number: CH.SC.U4CSE24227\n\n");
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    for(i = n/2 - 1; i >= 0; i--) {
        heapify(a, n, i);
    }
    for(i = n - 1; i >= 0; i--) {
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i, 0);
    }
    printf("\nSorted array:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

Output:



- **Time Complexity: O(n log n)**

The heapify function operates on the height of the tree, taking O(log n) time. Since we build the heap and then extract elements n times, the total time complexity is n * log n. This is efficient because it guarantees this performance even in the worst case.

- **Space Complexity: O(log n)**

While the sorting is performed on the array itself (in-place), the provided code uses a recursive heapify function. This recursion consumes stack memory proportional to the height of the binary tree, which is log n.

6.HEAP SORT(MIN HEAP)

Code:

```
#include <stdio.h>
void heapify(int a[], int n, int i) {
    int smallest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if(left < n && a[left] < a[smallest])
        smallest = left;
    if(right < n && a[right] < a[smallest])
        smallest = right;
    if(smallest != i) {
        int temp = a[i];
        a[i] = a[smallest];
        a[smallest] = temp;
        heapify(a, n, smallest);
    }
}
int main() {
    int a[100], n, i;
    printf("Name: M M VASANTH\n");
    printf("Roll Number: CH.SC.U4CSE24227\n\n");
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    for(i = n/2 - 1; i >= 0; i--) {
        heapify(a, n, i);
    }
    for(i = n - 1; i >= 0; i--) {
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i, 0);
    }
    printf("\nSorted array (descending order):\n");
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

Output:

```
vasanth@VasanthMM:/mnt/c/Users/vasan$ nano heap2.c
vasanth@VasanthMM:/mnt/c/Users/vasan$ gcc heap2.c -o heap2
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./heap2
Name: M M VASANTH
Roll Number: CH.SC.U4CSE24227

Enter number of elements: 7
Enter 7 elements:
45
56
22
13
37
7
95

Sorted array (descending order):
95 56 45 37 22 13 7 vasanth@VasanthMM:/mnt/c/Users/vasan$
```

- **Time Complexity: O(n log n)**

The code first builds a min-heap, which takes linear time. The sorting logic then iterates n times to extract the root element. In each iteration, the heapify function is called to restore the heap property. Since heapify traverses the height of the tree—which is logarithmic relative to the number of nodes—the total time complexity is n * log n.

- **Space Complexity: O(log n)**

Although the sorting is performed in-place within the array a[], the heapify function is implemented recursively. This recursion creates a stack of function calls proportional to the height of the binary tree (log n), consuming auxiliary stack memory.