SCHOOL OF COMPUTING

M M VASANTH

CH.SC.U4CSE24227

Week – 5

Date - 02/01/2026

Design and Analysis of Algorithm(23CSE211)

1. QUICK SORT

Code:

```
#include <stdio.h>

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```c
int main() {
    int arr[100], n, i;
    printf("Name: M M VASANTH\n");
    printf("Roll No: 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", &arr[i]);
    }

    quickSort(arr, 0, n - 1);

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

Output:

```
vasanth@VasanthMM:/mnt/c/Users/vasan$ nano quick.c
vasanth@VasanthMM:/mnt/c/Users/vasan$ gcc quick.c -o quick
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./quick
Name: M M VASANTH
Roll No: CH.SC.U4CSE24227

Enter number of elements: 5
Enter 5 elements:
34
56
63
92
35

Sorted array:
34 35 56 63 92 vasanth@VasanthMM:/mnt/c/Users/vasan$
```

Time Complexity: O(N^2)

The array is partitioned log N times (divide step). In each level of recursion, we iterate through the array segments to compare with the pivot (O(N)). Thus, the total time is N * log N. However, in the worst case (already sorted array), it degrades to O(N^2).

Space Complexity: O(N)

Quick Sort is an in-place sorting algorithm (it doesn't create a new array for sorting). However, it uses stack memory for recursion. In the worst case, the height of the recursion tree is N.

2. MERGE SORT

Code:

```
#include <stdio.h>


void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[50], R[50];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
```

```
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++; k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++; k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

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

    printf("Name: M M VASANTH\n");
    printf("Roll No: 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", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

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

Output:

```
Name: M M VASANTH
Roll No: CH.SC.U4CSE24227

Enter number of elements: 7
Enter 7 elements:
23
34
64
54
79
92
95

Sorted array:
23 34 54 64 79 92 95 vasanth@VasanthMM:/mnt/c/Users/vasan$
```

Time Complexity: O(N log N)

Similar to Quick Sort, the array is divided log N times. The merge operation takes linear time O(N) to combine the subarrays. Since the split is always perfectly balanced (halved), Merge Sort guarantees O(N log N) time complexity even in the worst case.

Space Complexity: O(N)

Unlike Quick Sort, Merge Sort is not in-place. The merge function requires temporary arrays (L and R) to store the split elements before combining them. This consumes auxiliary space proportional to the input size N.

3. BINARY SEARCH TREE (BST)

Code:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key;
    struct Node *left, *right;
};

struct Node* newNode(int item) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

struct Node* insert(struct Node* node, int key) {
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
```

```
    return node;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->key == key)
       return root;

    if (root->key < key)
       return search(root->right, key);

    return search(root->left, key);
}

int main() {
    struct Node* root = NULL;
    int n, i, val, key;
    printf("Name: M M VASANTH\n");
```

```
    printf("Roll No: CH.SC.U4CSE24227\n\n");

    printf("Enter number of elements to insert: ");
    scanf("%d", &n);
    printf("Enter elements:\n");
    for(i=0; i<n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }

    printf("Inorder Traversal (Sorted): ");
    inorder(root);
    printf("\n");

    printf("Enter element to search: ");
    scanf("%d", &key);
    if(search(root, key) != NULL)
        printf("Element Found\n");
    else
        printf("Element Not Found\n");

    return 0;
}
```

Output:

```
Name: M M VASANTH
Roll No: CH.SC.U4CSE24227

Enter number of elements to insert: 5
Enter elements:
24
43
56
95
94
Inorder Traversal (Sorted): 24 43 56 94 95
Enter element to search: 94
Element Found
vasanth@VasanthMM:/mnt/c/Users/vasan$
```

Time Complexity: O(N)

- Insertion/Search: In a balanced tree, we discard half the nodes at each step, making the height log N.
- Worst Case: If the tree becomes "skewed" (like a linked list, e.g., inserting 1, 2, 3, 4, 5 in order), the height becomes N, degrading operations to O(N).

Space Complexity: O(N)

The space is required to store the N nodes of the tree. Additionally, the recursion stack for insertion or traversal can grow up to O(N) in the worst-case (skewed tree).