DESIGN AND ANALYSIS OF ALGORITHM

23CSE211-MERGE & QUICK

NAME : UPPU MANJU SRI NATH

ROLL NO : CH.SC.U4CSE24249

1) **MERGE SORT :**

CODE:

```c
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    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++;
    }
}
```

```c
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, size);

    mergeSort(arr, 0, size - 1);

    printf("Sorted array:\n");
    printArray(arr, size);

    return 0;
}
```

OUTPUT:

```
Original array:
157 110 147 122 111 149 151 141 123 112 117 133
Sorted array:
110 111 112 117 122 123 133 141 147 149 151 157
```

Time Complexity

• **Worst Case: O(n log n)** – the array is divided and merged at every step.
• **Average Case: O(n log n)** – same operations happen for any input order.
• **Best Case: O(n log n)** – even if the array is already sorted, it still divides and merges.

Space Complexity

• **O(n)** – extra arrays are used while merging.
• Sorting is **not in-place** because additional memory is required.

2) QUICK SORT:

CODE:

```c
#include <stdio.h>


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


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

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

```c
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);
    }
}


void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

```
Original array:
157 110 147 122 111 149 151 141 123 112 117 133
Sorted array:
110 111 112 117 122 123 133 141 147 149 151 157
```

Time Complexity

- **Worst Case: O(n²)** – happens when the pivot always gives unbalanced partitions.
- **Average Case: O(n log n)** – array is divided into fairly equal parts.
- **Best Case: O(n log n)** – pivot divides the array into two equal halves.

Space Complexity

- **O(log n)** – space used by recursive calls.
- Sorting is **in-place** because no extra arrays are used.