Experiment **#6**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Experiment Title: To implement programs on problem solving using Divide & Conquer – Scenario2.

Aim/Objective: To understand the concept and implementation of Basic programs on Divide and Conquer Problems.

Description: The students will understand and able to implement programs on Divide and Conquer Problems.

Pre-Requisites:

Knowledge: Arrays, Sorting, Divide and Conquer in C/C++/Python

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

1. Given an array *arr*, count the number of inversions in the array. Two elements form an inversion if *arr[i]* > *arr[j]* and *i* < *j*. Use the Divide and Conquer method to solve the problem efficiently.

Input Format:

- First line contains an integer `n`, the size of the array.
- Second line contains `n` space-separated integers representing the array.

Output Format:

- Print the total number of inversions.

Example :

Input:

5

2 4 1 3 5

Output: 3

- **Procedure/Program**:

```
#include <stdio.h>

int merge(int arr[], int temp[], int left, int right) {
    if (left >= right) return 0;

    int mid = (left + right) / 2;
    int inv_count = merge(arr, temp, left, mid) + merge(arr, temp, mid + 1, right);

    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
            inv_count += (mid - i + 1);
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (i = left; i <= right; i++) arr[i] = temp[i];

    return inv_count;
}
int count_inversions(int arr[], int n) {
    int temp[n];
    return merge(arr, temp, 0, n - 1);
}

int main() {
    int arr[] = {2, 4, 1, 3, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("%d\n", count_inversions(arr, n));
    return 0;
}
```

- **Data and Results:**

Data

The array contains elements to analyze inversion counts efficiently.

Result

The total number of inversions in the array is calculated.

- **Analysis and Inferences:**

Analysis

Inversions occur when elements are out of their natural order.

Inferences

Efficient algorithms like Merge Sort reduce inversion count calculation time.

2. Find the k^{th} smallest element in an array using the Median of Medians algorithm (a Divide and Conquer-based selection algorithm).

   Input Format:

   - First line contains two integers n and k.
   - Second line contains n space-separated integers representing the array.

   Output Format:

   Print the kth smallest element.

Example :

Input:

6 3

7 10 4 3 20 15

Output:

7

- **Procedure/Program**:

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

int partition(int arr[], int left, int right, int pivotIndex) {
  int pivotValue = arr[pivotIndex];
  int storeIndex = left;
  int temp;

  temp = arr[pivotIndex];
  arr[pivotIndex] = arr[right];
  arr[right] = temp;

  for (int i = left; i < right; i++) {
```

```
        if (arr[i] < pivotValue) {
            temp = arr[storeIndex];
            arr[storeIndex] = arr[i];
            arr[i] = temp;
            storeIndex++;
        }
    }

    temp = arr[storeIndex];
    arr[storeIndex] = arr[right];
    arr[right] = temp;

    return storeIndex;
}

int select(int arr[], int left, int right, int k) {
    if (left == right) {
        return arr[left];
    }

    int pivotIndex = left + (right - left) / 2;
    pivotIndex = partition(arr, left, right, pivotIndex);

    if (k == pivotIndex) {
        return arr[k];
    } else if (k < pivotIndex) {
        return select(arr, left, pivotIndex - 1, k);
    } else {
        return select(arr, pivotIndex + 1, right, k);
```

```
    }
}

int main() {
    int n = 6, k = 3;
    int arr[] = {7, 10, 4, 3, 20, 15};

    int result = select(arr, 0, n - 1, k - 1);
    printf("%d\n", result);

    return 0;
}
```

- **Data and Results:**

Data:

The array contains six integers, and `k` specifies the rank.

Result:

The function returns the 3rd smallest number in the array.

- **Analysis and Inferences:**

Analysis:

The code implements Quickselect to find the k-th element.

Inferences:

Quickselect is efficient, modifying input and using partitioning strategy.

In-Lab:

1. Choose some pivot element, **p**, and partition your unsorted array, **arr**, into three smaller arrays: **left**, **right**, and **equal**, where each element in **left < p**, each element in **right > p**, and each element in **equal = p**.

   Example

   arr = [5, 7, 4, 3, 8]

   In this challenge, the pivot will always be at **arr[0]**, so the pivot is **5**.

   arr is divided into **left = {4,3}**, **equal = {5}**, and **right = {7,8}**. Putting them all together, you get **{4,3,5,7,8}**. There is a flexible checker that allows the elements of **left** and **right** to be in any order. For example, **{3,4,5,8,7}** is valid as well.

   Given **arr** and **p = arr[0],** partition **arr** into **left, right, and equal** using the Divide instructions above. Return a 1-dimensional array containing each element in left first, followed by each element in equal, followed by each element in right.

   Function Description

   Complete the quickSort function in the editor below.

   quickSort has the following parameter(s):

   - int arr[n]: **arr[0]** is the pivot element

   Returns

   - int[n]: an array of integers as described above

   Input Format

   The first line contains **n**, the size of **arr**. The second line contains **n** space-separated integers **arr[i]** (the unsorted array). The first integer, **arr[0],** is the pivot element, **p**.

   Constraints

   - 1 <= n <= 1000
   - -1000 <= arr[i] <= 1000 where 0 <= i < n
   - All elements are distinct.

Sample Input

STDIN	Function
----	------
5	arr[] size n =5

4 5 3 7 2 arr =[4, 5, 3, 7, 2]

Sample Output

3 2 4 5 7

- **Procedure/Program**:

```
#include <stdio.h>

void quickSort(int arr[], int n, int result[]) {
   int pivot = arr[0];
   int left[n], right[n], equal[n];
   int leftCount = 0, rightCount = 0, equalCount = 0;

   for (int i = 0; i < n; i++) {
      if (arr[i] < pivot) {
         left[leftCount++] = arr[i];
      } else if (arr[i] > pivot) {
         right[rightCount++] = arr[i];
      } else {
         equal[equalCount++] = arr[i];
      }
   }

   for (int i = 0; i < leftCount; i++) {
      result[i] = left[i];
   }
   for (int i = 0; i < equalCount; i++) {
      result[leftCount + i] = equal[i];
```

```
    }
    for (int i = 0; i < rightCount; i++) {
        result[leftCount + equalCount + i] = right[i];
    }
}

int main() {
    int n = 5;
    int arr[] = {4, 5, 3, 7, 2};
    int result[n];

    quickSort(arr, n, result);

    for (int i = 0; i < n; i++) {
        printf("%d ", result[i]);
    }
    return 0;
}
```

- **Data and Results:**

Data:

The input array contains five integers to be sorted.

Result:

The array is sorted using a quicksort-like partitioning approach.

- **Analysis and Inferences:**

Analysis:

The algorithm partitions the array into left, equal, and right groups.

Inferences:

This implementation uses a non-recursive, partition-based sorting technique.

2. You are given *k* painters to paint *n* boards. Each painter takes 1 unit of time to paint 1 unit of the board. Find the minimum time required to paint all boards using Divide and Conquer and Binary Search.

Input Format:

- First line contains two integers n (number of boards) and k (number of painters).
- Second line contains n space-separated integers representing the lengths of the boards.

Output:

- Print the minimum time required.

Sample Input:

4 2

10 20 30 40

Output:

60

- **Procedure/Program**:

```
#include <stdio.h>

int isPossible(int boards[], int n, int k, int mid) {

    int painterCount = 1;

    int currentLength = 0;
```

```
    for (int i = 0; i < n; i++) {
        currentLength += boards[i];
        if (currentLength > mid) {
            painterCount++;
            currentLength = boards[i];
        }
        if (painterCount > k) {
            return 0;
        }
    }
    return 1;
}
int findMinTime(int boards[], int n, int k) {
    int start = 0, end = 0, result = 0;


    for (int i = 0; i < n; i++) {
        end += boards[i];
        start = (start < boards[i]) ? boards[i] : start;
    }
while (start <= end) {
        int mid = (start + end) / 2;
```

```
 if (isPossible(boards, n, k, mid)) {
            result = mid;
            end = mid - 1;
        } else {
            start = mid + 1;
        }
    }
    return result;
}

int main() {
    int boards[] = {10, 20, 30, 40};
    int n = 4;
    int k = 2;

    int minTime = findMinTime(boards, n, k);
    printf("%d\n", minTime);

    return 0;
}
```

- **Data and Results:**

Data:

Given 4 boards and 2 painters, determine minimum time for painting.

Result:

The minimum time required to paint all boards is 60.

- **Analysis and Inferences:**

Analysis:

Binary search determines the minimum time by checking possible limits.

Inferences:

Efficient use of binary search optimizes painter allocation and time.

Post-Lab:

Find the longest common prefix among an array of strings using Divide and Conquer.

Input Format:

- First line contains an integer n, the number of strings.
- Next n lines each contain a string.

Output:

- Print the longest common prefix. If there is no common prefix, print an empty string.

Sample Input:

3

flower

flow

flight

Output:

fl

- **Procedure/Program**:

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

char* commonPrefix(char* str1, char* str2) {
   int i = 0, minLen = strlen(str1) < strlen(str2) ? strlen(str1) : strlen(str2);
   while (i < minLen && str1[i] == str2[i]) i++;
   str1[i] = '\0';
   return str1;
}

char* lcp(char* arr[], int left, int right) {
   if (left == right) return arr[left];
   int mid = (left + right) / 2;
   return commonPrefix(lcp(arr, left, mid), lcp(arr, mid + 1, right));
}
```

```
int main() {
    int n;
    scanf("%d", &n);
    char* arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = (char*)malloc(100 * sizeof(char));
        scanf("%s", arr[i]);
    }
    printf("%s\n", lcp(arr, 0, n - 1));
    return 0;
}
```

- **Data and Results:**

Data:

Given an array of strings, find the longest common prefix.

Result:

The longest common prefix among the strings is printed.

- **Analysis and Inferences:**

Analysis:

Divide and conquer approach recursively finds the common prefix.

Inferences:

Efficient use of recursion and string comparison ensures optimal performance.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. Why is the recurrence relation important in analyzing Divide and Conquer algorithms?

- It helps determine the time complexity by relating the problem's overall time to the time for subproblems.

2. What are the limitations of Divide and Conquer? When should it not be used?

- It can introduce overhead, increase memory usage, and may not be efficient for small problems or sequential tasks.

3. What is the Master Theorem? How is it applied in Divide and Conquer?

- The Master Theorem simplifies solving recurrences of the form $T(n) = aT(n/b) + O(n^d)$ to find the time complexity based on values of a, b, and d.

4. What are the differences in space complexity between recursive Divide and Conquer algorithms and iterative solutions.

- Recursive solutions use extra space for call stacks, while iterative solutions typically use less space.

5. Give an example of a Divide and Conquer algorithm that is not recursive.

- Iterative Merge Sort, which uses a bottom-up approach to merge subarrays without recursion.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.