Experiment **#3**		Student ID	
Date		Student Name	[@KLWKS_BOT THANOS]

Experiment Title: Implementation of programs on Sorting and Searching problems.

Aim/Objective: To understand the concept and implementation of programs on Sorting and Searching problems.

Description: The students will understand and able to implement programs on Sorting and Searching problems.

Pre-Requisites:

Knowledge: Sorting, Searching, Arrays in C/C++/PythonTools: Code Blocks/Eclipse IDE

Pre-Lab:

Implementing and Analyzing the Rabin-Karp String Matching Algorithm.

- **Procedure/Program**:

Rabin-Karp Algorithm

Input: Text `T` of length `n`, Pattern `P` of length `m`
Output: Indices where `P` is found in `T`.

Steps:

1. Calculate hash of `P` and first window of `T`.
2. Slide the window over `T`:
   - If hashes match, compare characters.
   - If match, print index.
3. Update hash for next window:
   `hash_window = (d * (hash_window - text[i] * h) + text[i + m]) % q`.

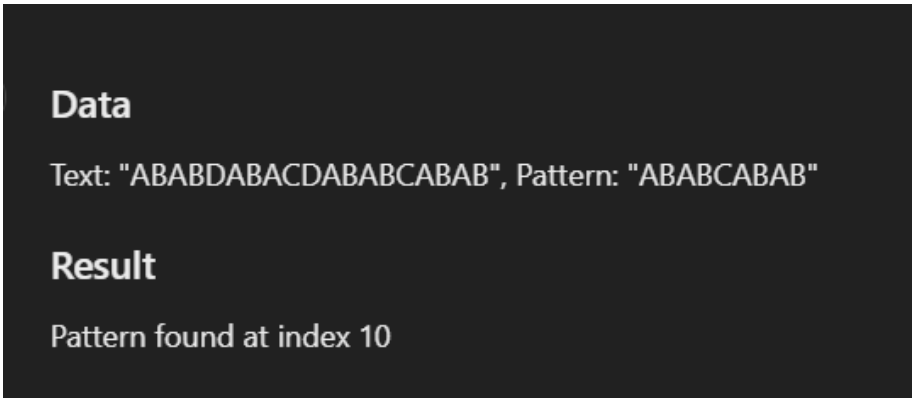
Time Complexity:

- **Best Case**: O(n + m)
- **Worst Case**: O(n * m)
- **Average Case**: O(n + m)

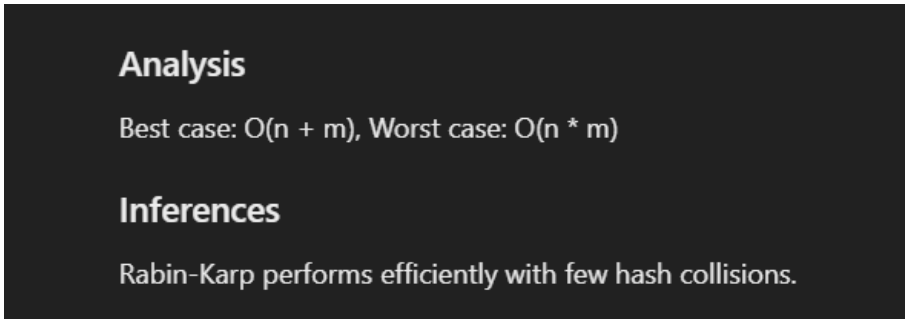
```
#include <stdio.h>
#include <string.h>

#define d 256
#define q 101

void rabinKarp(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   int pHash = 0;
   int tHash = 0;
   int h = 1;

   for (int i = 0; i < m - 1; i++) {
      h = (h * d) % q;
   }

   for (int i = 0; i < m; i++) {
      pHash = (d * pHash + pattern[i]) % q;
      tHash = (d * tHash + text[i]) % q;
   }

   for (int i = 0; i <= n - m; i++) {
      if (pHash == tHash) {
         int match = 1;
         for (int j = 0; j < m; j++) {
            if (text[i + j] != pattern[j]) {
               match = 0;
               break;
            }
         }
         if (match) {
            printf("Pattern found at index %d\n", i);
         }
      }

      if (i < n - m) {
```

```
            tHash = (d * (tHash - text[i] * h) + text[i + m]) % q;
            if (tHash < 0) {
               tHash = tHash + q;
            }
         }
      }
   }

int main() {
   char text[] = "ABABDABACDABABCABAB";
   char pattern[] = "ABABCABAB";

   rabinKarp(text, pattern);

   return 0;
}
```

- **Data and Results:**



- **Analysis and Inferences**



In-Lab:

Alice uses a particular algorithm to systematically sort a sequence of N unique positive integers. Determine the final value of the variable comparison count, which tracks the total number of comparisons made, after using the provided Quicksort algorithm to sort a given permutation of integers in the list A.

Input

The first line of input consists of a single integer N. The second line of input contains a permutation of N numbers.

Output

Output the number of comparisons on the first and only line of the output.

Example

Input:

5

4 3 5 1 2

Output:

11

- **Procedure/Program:**

```
#include <stdio.h>

int comparisons = 0;

int partition(int A[], int low, int high) {
   int pivot = A[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      comparisons++; // Count comparison
      if (A[j] < pivot) {
         i++;
         int temp = A[i];
         A[i] = A[j];
         A[j] = temp;
      }
   }
   int temp = A[i + 1];
   A[i + 1] = A[high];
   A[high] = temp;
   return i + 1;
}

void quicksort(int A[], int low, int high) {
   if (low < high) {
      int pi = partition(A, low, high);
      quicksort(A, low, pi - 1);
      quicksort(A, pi + 1, high);
   }
}

int main() {
   int N;
   scanf("%d", &N);
   int A[N];
```

```
    for (int i = 0; i < N; i++) {
        scanf("%d", &A[i]);
    }

    quicksort(A, 0, N - 1);

    printf("%d\n", comparisons);
    return 0;
}
```

- **Data and Results:**

Data:

Input consists of N and a permutation of N integers.

Result:

Total comparisons made during Quicksort on given input is 11.

- **Analysis and Inferences**

Analysis:

Quicksort's partitioning and recursion determine the total comparison count.

Inferences:

Comparison count depends on array order and Quicksort's partitioning strategy.

Post-Lab:

Stefan is a guy who is suffering with OCD. He always like to align things in an order. He got a lotof strings for his birthday party as gifts. He like to sort the strings in a unique way. He wants his strings to be sorted based on the count of characters that are present in the string.

Input

aaabbc

aabbcc

Output

cbbaaa

aabbcc

If in case when there are two characters is same, then the lexicographically smaller one will be printed first

Input:

aabbccdd

aabcc

Output:

aabbccdd

baacc

- **Procedure/Program:**

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

int compare(const void *a, const void *b) {
   const char *str1 = *(const char **)a;
   const char *str2 = *(const char **)b;

   int len1 = strlen(str1);
   int len2 = strlen(str2);

   if (len1 != len2) {
      return len1 - len2;
```

```
    }

    return strcmp(str1, str2);
}

void sort_strings_by_char_count(char *input) {
    char *strings[100];
    int count = 0;

    char *token = strtok(input, " ");
    while (token != NULL) {
        strings[count++] = token;
        token = strtok(NULL, " ");
    }

    qsort(strings, count, sizeof(char *), compare);

    for (int i = 0; i < count; i++) {
        printf("%s", strings[i]);
        if (i < count - 1) {
            printf(" ");
        }
    }
    printf("\n");
}

int main() {
    char input1[] = "aaabbc aabbcc";
    printf("Input: %s\n", input1);
    printf("Output: ");
    sort_strings_by_char_count(input1);

    char input2[] = "aabbccdd aabcc";
    printf("Input: %s\n", input2);
    printf("Output: ");
    sort_strings_by_char_count(input2);
```

```
    return 0;
}
```

- **Data and Results:**

Data

Input strings are sorted based on length and lexicographical order.

Result

Strings sorted by length and lexicographical order successfully printed.

- **Analysis and Inferences:**

Analysis

Strings are divided into tokens and sorted using `qsort` function.

Inferences

Sorting is effective for sorting strings by length and alphabetically.

- **Sample VIVA-VOCE Questions:**

1. How does the time complexity of the heap sort algorithm compare to other sorting algorithms?

O(n log n), better than bubble/insertion sort ($O(n^2)$), but slower than quicksort.

2. What is the role of the partition function in quicksort? How does it affect the overall sorting process?

Divides array by pivot, ensuring left < pivot, right > pivot.

3. Explain the difference between internal and external sorting.

Internal: Data fits in memory.
External: Data is too large for memory.

4. What is the time complexity of the bubble sort algorithm? Can it be improved?

$O(n^2)$, can be improved with a flag to check sorted state.

5. Compare the time complexity of insertion sort and selection sort. Which one is more efficient?

Both $O(n^2)$; insertion is better for nearly sorted data.

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.