Date of the Session:__/__/_____ **Time of the Session:_____to_____**

EX – 2 Implementation of String Matching Algorithms

Prerequisites:

- Basics of Data Structures and C Programming.
- Basic knowledge about String Data type.

Pre-Lab :

1) Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search (char pat [], char txt[]) that prints all occurrences of pat[] in txt[] using naïve string algorithm?(assume that n > m)

Input
txt[] = "THIS IS A TEST TEXT"
pat[] = "TEST"
Output
Pattern found at index 10
Input
txt[] = "AABAACAADAABAABA"
pat[] = "AABA"
Output
Pattern found at index 0
Pattern found at index 9

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

typedef struct {
   char character;
   int frequency;
} CharFreq;

int compare(const void *a, const void *b) {
   CharFreq *cf1 = (CharFreq *)a;
   CharFreq *cf2 = (CharFreq *)b;
   if (cf1->frequency != cf2->frequency)
      return cf1->frequency - cf2->frequency;
   return cf1->character - cf2->character;
}

void sortString(char str[]) {
   int freq[256] = {0};
```

```
    int n = strlen(str);
    for (int i = 0; i < n; i++) {
        freq[(int)str[i]]++;
    }
    CharFreq charFreqs[256];
    int count = 0;
    for (int i = 0; i < 256; i++) {
        if (freq[i] > 0) {
            charFreqs[count].character = (char)i;
            charFreqs[count].frequency = freq[i];
            count++;
        }
    }
    qsort(charFreqs, count, sizeof(CharFreq), compare);
    int index = 0;
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < charFreqs[i].frequency; j++) {
            str[index++] = charFreqs[i].character;
        }
    }
    str[index] = '\0';
}

int main() {
    char str1[] = "aaabbc";
    char str2[] = "cbbaaa";
    sortString(str1);
    sortString(str2);
    printf("Sorted strings:\n%s\n%s\n", str1, str2);

    char str3[] = "aabbccdd";
    char str4[] = "aabcc";
    sortString(str3);
    sortString(str4);
    printf("Sorted strings:\n%s\n%s\n", str3, str4);

    return 0;
}
```

2) Discuss the Rabin Karp algorithm for string matching and explain time complexity of the algorithm?

The **Rabin-Karp algorithm** is used for string matching. It computes a hash value for the pattern and compares it with hash values of substrings in the text. If the hashes match, it checks the actual strings to avoid collisions.

Time Complexity:

- **Best case: O(n + m)** — No hash collisions, just direct comparisons.
- **Worst case: O(n * m)** — Many hash collisions, requiring string comparisons.
- **Average case: O(n + m)** — With a good hash function, collisions are minimized.

3) Stefan is a guy who is suffering with OCD. He always like to align things in an order. He got a lot of strings for his birthday party as gifts. He likes 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
cbbaaa

Output

aabbcc
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

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

typedef struct {
   char character;
   int frequency;
} CharFreq;

int compare(const void *a, const void *b) {
   CharFreq *cf1 = (CharFreq *)a;
   CharFreq *cf2 = (CharFreq *)b;
   if (cf1->frequency != cf2->frequency)
      return cf1->frequency - cf2->frequency;
   return cf1->character - cf2->character;
}

void sortString(char str[]) {
   int freq[256] = {0};
   int n = strlen(str);
   for (int i = 0; i < n; i++) {
      freq[(int)str[i]]++;
   }
   CharFreq charFreqs[256];
   int count = 0;
```

```
    for (int i = 0; i < 256; i++) {
        if (freq[i] > 0) {
            charFreqs[count].character = (char)i;
            charFreqs[count].frequency = freq[i];
            count++;
        }
    }
    qsort(charFreqs, count, sizeof(CharFreq), compare);
    int index = 0;
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < charFreqs[i].frequency; j++) {
            str[index++] = charFreqs[i].character;
        }
    }
    str[index] = '\0';
}

int main() {
    char str1[] = "aaabbc";
    char str2[] = "cbbaaa";
    sortString(str1);
    sortString(str2);
    printf("Sorted strings:\n%s\n%s\n", str1, str2);

    char str3[] = "aabbccdd";
    char str4[] = "aabcc";
    sortString(str3);
    sortString(str4);
    printf("Sorted strings:\n%s\n%s\n", str3, str4);

    return 0;
}
```

In-Lab:

1) Naive method and KMP are two string comparison methods. Write a program for Naïve method and KMP to check whether a pattern is present in a string or not. Using clock function find execution time for both and compare the time complexities of both the programs (for larger inputs) and discuss which one is more efficient and why?

Sample program with function which calculate execution time:

```
#include<stdio.h>
#include<time.h>
void fun()
{
   //some statements here
}
int main()
{
   //calculate time taken by fun()
clock_t t;
t=clock();
fun();
t=clock()-t;
double time_taken=((double)t)/CLOCK_PER_SEC; //in seconds
printf("fun() took %f seconds to execute \n",time_taken);
return 0;
}
```

Source code:

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

int naiveSearch(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);

   for (int i = 0; i <= n - m; i++) {
      int j = 0;
      while (j < m && text[i + j] == pattern[j]) {
         j++;
      }
      if (j == m) {
         return i;
      }
   }
   return -1;
}

void computeLPSArray(char *pattern, int m, int *lps) {
   int len = 0;
```

```
    int i = 1;
    lps[0] = 0;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int KMPSearch(char *text, char *pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int lps[m];

    computeLPSArray(pattern, m, lps);

    int i = 0;
    int j = 0;

    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }

        if (j == m) {
            return i - j;
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return -1;
}

void calculateExecutionTime(void (*func)(), char *text, char *pattern) {
    clock_t start, end;
    start = clock();

    func(text, pattern);

    end = clock();
```

```
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", time_taken);
}

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

    printf("Naive Search:\n");
    calculateExecutionTime(naiveSearch, text, pattern);

    printf("KMP Search:\n");
    calculateExecutionTime(KMPSearch, text, pattern);

    return 0;
}
```

OUTPUT

```
Naive Search:
Execution time: 0.000002 seconds
KMP Search:
Execution time: 0.000001 seconds
```

time complexities

Naive String Matching:

- **Time Complexity:** O(n * m)
  - **n** = length of text
  - **m** = length of pattern
  - **Worst Case:** O(n * m) (needs to check each substring of length m in the text)

KMP String Matching:

- **Time Complexity:** O(n + m)
  - **n** = length of text
  - **m** = length of pattern
  - **Worst Case:** O(n + m) (preprocessing the pattern in O(m) and matching in O(n))

discuss which one is more efficient and why

KMP is more efficient than the Naive method because:

- **Naive** has a time complexity of **O(n * m)**, requiring redundant comparisons for each possible substring.
- **KMP** preprocesses the pattern in **O(m)** time to create an LPS array and then matches the text in **O(n)** time, resulting in a total complexity of **O(n + m)**.

2) Lisa is a school student teacher gave her an assignment to check whether the pattern is there or not in a given text and also she mentioned that it is have solve by using kmp algorithm so when a mismatch come other some matches in your search if she print the number of letters that we can neglect before then she will get good marks so help her by writing a code.

Sample Input:
ABABDABACDABABCABAB
ABABCABAB
Sample Output:
we don't match before 2 letters because they will match anyway
we don't match before 0 letters because they will match anyway
we don't match before 1 letter because they will match anyway
we don't match before 0 letters because they will match anyway
Found pattern at index 10

Source code:

```
#include <stdio.h>

#include <string.h>


void computeLPSArray(char* pattern, int M, int* lps) {
   int length = 0;
   lps[0] = 0;
   int i = 1;


   while (i < M) {
      if (pattern[i] == pattern[length]) {
         length++;
         lps[i] = length;
         i++;
      } else {
         if (length != 0) {
            length = lps[length - 1];
         } else {
            lps[i] = 0;
            i++;
```

```
        }
      }
    }
}

void KMPSearch(char* text, char* pattern) {
    int N = strlen(text);
    int M = strlen(pattern);
    int lps[M];
    computeLPSArray(pattern, M, lps);
    int i = 0;
    int j = 0;

    while (i < N) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }

        if (j == M) {
            printf("Found pattern at index %d\n", i - j);
            j = lps[j - 1];
        } else if (i < N && pattern[j] != text[i]) {
            if (j != 0) {
                printf("we don’t match before %d letters because they will match anyway\n", j);
                j = lps[j - 1];
            } else {
                printf("we don’t match before %d letters because they will match anyway\n", j);
```

```
            i++;
        }
      }
    }
}

int main() {
    char text[] = "ABABDABACDABABCABAB";
    char pattern[] = "ABABCABAB";
    KMPSearch(text, pattern);
    return 0;
}
```

Post-Lab:

1) Given a pattern of length- 5 window, find the valid match in the given text by step-by-step process using Robin-Karp algorithm
Pattern: 2 1 9 3 6
Modulus: 21
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Text: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7

Source code:

```
#include <stdio.h>

int computeHash(int arr[], int start, int end, int modulus) {
   int hash = 0;
   for (int i = start; i <= end; i++) {
      hash = (hash * 10 + arr[i]) % modulus;
   }
   return hash;
}

void rabinKarp(int text[], int textLen, int pattern[], int patternLen, int modulus) {
   int patternHash = computeHash(pattern, 0, patternLen - 1, modulus);
   int currentHash = computeHash(text, 0, patternLen - 1, modulus);

   printf("Pattern Hash: %d\n", patternHash);
   printf("Step-by-step process:\n");

   for (int i = 0; i <= textLen - patternLen; i++) {
      printf("Index %d-%d, Text Window: ", i, i + patternLen - 1);
      for (int j = i; j < i + patternLen; j++) {
         printf("%d ", text[j]);
      }
      printf("\n");

      printf("Current Hash: %d\n", currentHash);

      if (currentHash == patternHash) {
         int match = 1;
         for (int j = 0; j < patternLen; j++) {
            if (text[i + j] != pattern[j]) {
               match = 0;
               break;
            }
         }
```

```
            if (match) {
                printf("Valid match found at index %d\n", i);
            } else {
                printf("Hash matched, but window doesn't match pattern.\n");
            }
        }

        if (i < textLen - patternLen) {
            currentHash = (currentHash * 10 - text[i] * 10000 + text[i + patternLen]) %
    modulus;
            if (currentHash < 0) {
                currentHash += modulus;
            }
        }
    }
}

int main() {
    int text[] = {9, 2, 7, 2, 1, 8, 3, 0, 5, 7, 1, 2, 1, 2, 1, 9, 3, 6, 2, 3, 9, 7};
    int pattern[] = {2, 1, 9, 3, 6};
    int textLen = sizeof(text) / sizeof(text[0]);
    int patternLen = sizeof(pattern) / sizeof(pattern[0]);
    int modulus = 21;

    rabinKarp(text, textLen, pattern, patternLen, modulus);

    return 0;
}
```

2) James is sharing his information with his friend secretly in a chat. But he thinks that message should not understandable to anyone only for him and his friend. So he sent the message in the following format.

Input

a1b2c3d4e

Output

abbdcfdhe

Explanation:

The digits are replaced as follows:

- s[1] -> shift('a',1) = 'b'
- s[3] -> shift('b',2) = 'd'
- s[5] -> shift('c',3) = 'f'
- s[7] -> shift('d',4) = 'h'

Source code:

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

char* transform_message(const char* s) {
   int len = strlen(s);
   char* result = (char*)malloc(len + 1);
   int result_index = 0;

   for (int i = 0; i < len; i++) {
      if (isdigit(s[i])) {
         int shift_value = s[i] - '0';
         char previous_char = result[result_index - 1];
         char shifted_char = previous_char + shift_value;
         result[result_index++] = shifted_char;
      } else {
         result[result_index++] = s[i];
      }
   }
   result[result_index] = '\0';
   return result;
}

int main() {
   const char* input_str = "a1b2c3d4e";
   char* output_str = transform_message(input_str);
   printf("Output: %s\n", output_str);
   free(output_str);
```

```
    return 0;
}
```

Comments of the Evaluators (if Any)	Evaluator's Observation Marks Secured:_____out of [50]. Signature of the Evaluator Date of Evaluation: