Compare the performance of Rabin-Karp, Knuth-Morris-Pratt and naive stringmatching algorithms.

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <stdlib.h>

// Naive String Matching Algorithm

int naiveSearch(char \*text, char \*pattern) {

int n = strlen(text);

int m = strlen(pattern);

int count = 0;

// Loop through the text

for (int i = 0; i <= n - m; i++) {

int j = 0;

while (j < m && text[i + j] == pattern[j]) {

j++;

}

if (j == m) {

count++; // Pattern found at index i

}

}

return count;

}

// Rabin-Karp Algorithm for Pattern Matching

#define d 256 // Number of characters in the input alphabet

#define q 101 // A prime number for hashing

int rabinKarpSearch(char \*text, char \*pattern) {

int n = strlen(text);

int m = strlen(pattern);

int i, j;

int count = 0;

int p = 0; // Hash value for pattern

int t = 0; // Hash value for text

int h = 1;

// The value of h would be "pow(d, m-1)%q"

for (i = 0; i < m - 1; i++)

h = (h \* d) % q;

// Calculate the hash value of pattern and first window of text

for (i = 0; i < m; i++) {

p = (d \* p + pattern[i]) % q;

t = (d \* t + text[i]) % q;

}

// Slide the pattern over text one by one

for (i = 0; i <= n - m; i++) {

// Check the hash values of current window of text and pattern

if (p == t) {

// If the hash values match, check for characters one by one

for (j = 0; j < m; j++) {

if (text[i + j] != pattern[j])

break;

}

if (j == m)

count++; // Pattern found at index i

}

// Calculate hash value for next window of text: remove leading digit, add trailing digit

if (i < n - m) {

t = (d \* (t - text[i] \* h) + text[i + m]) % q;

if (t < 0)

t = (t + q);

}

}

return count;

}

// Knuth-Morris-Pratt (KMP) Algorithm for Pattern Matching

void computeLPSArray(char \*pattern, int m, int \*lps) {

int len = 0;

int i = 1;

lps[0] = 0;

// Build the LPS array

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 count = 0;

int lps[m]; // Longest Prefix Suffix array

// Compute the longest prefix suffix array

computeLPSArray(pattern, m, lps);

int i = 0, j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

i++;

j++;

}

if (j == m) {

count++; // Pattern found at index i - j

j = lps[j - 1];

} else if (i < n && pattern[j] != text[i]) {

if (j != 0)

j = lps[j - 1];

else

i++;

}

}

return count;

}

// Helper function to run and time the algorithms

void runAndTimeAlgorithms(char \*text, char \*pattern) {

clock\_t start, end;

// Naive String Matching

start = clock();

int naiveCount = naiveSearch(text, pattern);

end = clock();

printf("Naive Algorithm found %d matches in %.6f seconds\n", naiveCount, ((double)(end - start)) / CLOCKS\_PER\_SEC);

// Rabin-Karp Algorithm

start = clock();

int rkCount = rabinKarpSearch(text, pattern);

end = clock();

printf("Rabin-Karp Algorithm found %d matches in %.6f seconds\n", rkCount, ((double)(end - start)) / CLOCKS\_PER\_SEC);

// Knuth-Morris-Pratt Algorithm

start = clock();

int kmpCount = KMPsearch(text, pattern);

end = clock();

printf("KMP Algorithm found %d matches in %.6f seconds\n", kmpCount, ((double)(end - start)) / CLOCKS\_PER\_SEC);

}

int main() {

char text[] = "ABABDABACDABABCABAB";

char pattern[] = "ABAB";

printf("Comparing performance of String Matching Algorithms:\n");

runAndTimeAlgorithms(text, pattern);

return 0;

}