

Due: April 1st, 2020

# Assignment #3

**Design and Analysis of Algorithms**

**Submitted by:** Rafsha Mazhar

**Roll Number:** i17-0028

**Section:** E

# GitHub

rafshamazhar / AlgoAssignment Private

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings

No description, website, or topics provided. Edit

Manage topics

1 commit 1 branch 0 packages 0 releases

Branch: master New pull request Create new file Upload files Find file Clone or download

rafshamazhar done	Latest commit 638fbda 6 minutes ago
pAq2.cpp	done 6 minutes ago
pBq1.cpp	done 6 minutes ago
pBq2.c	done 6 minutes ago
pBq4.cpp	done 6 minutes ago

Add a README with an overview of your project. Add a README

## Part A

1. Explain with examples the best and worst case scenario of the Naïve String matching algorithm.

**Marks will only be given for properly writing each step with notations.**

In Naïve String Matching algorithm, we are given a sample text string, say `text []`, and a test pattern string, say `pattern []`. The loop iterates over the `text []` and looks for the `pattern []` string. If any substring of `text []` exactly matches with the `pattern[]` string, pattern is matched. After a match, the loop starts again at  $(found+1)$ th index, *found* being the index of the `text[]` where the pattern matched.

Keeping this in mind,

### Best Case

The best case scenario for Naïve String Matching Algorithm would be when the pattern string is not present in the text string such that, the first character of `pattern[]` string itself is absent in the given `text[]` string.

In such a case, the loop will iterate over the `text[]` string without finding any substring matches, to the very end.

For example, if the given text is:

Text [ ] = aaabcaddeaab , n = 12

Pattern [ ] = feaa , m = 4

n and m being the lengths respectively

In this case, as the first character of the pattern string isn't even present anywhere in the text string, the loop will simply skip the inner loop and just iterate over the text string once.

In such cases, best case complexity of the algorithm would be **O(n)** as the loop simply iterates over the text once.

## Worst Case

There are two worst case scenarios for this approach.

1. If both, the test pattern [] and the given text[] string comprises of a single character repeating throughout, til the very end.

For example,

Text [ ] = rrrrrrrrrrrrrr , n = 15

Pattern [ ] = rrr , m = 3

In this case, a match would be found at each index of the text[] and the loop will have to be restarted at (*found*+1)th index, every time pattern is matched, increasing the complexity of th algorithm significantly.

2. Another worst case scenario is if all the characters of the pattern[] matches to a substring of the text[] except the very last character.

For example,

Text [ ] = qwertyqwertyqwerty , n = 18

Pattern [ ] = qwertyy , m = 7

Here, the pattern matches to the text at various indeces up until 'qwerty' but a last 'y' is missing. So at each of these indeces, the loop will match the pattern to the text index by index, only to prove a mismatch at the last index of the pattern. It will have to start over from the starting index all over again, the same way it did for a match, even though there wouldn't even be a match.

In both these cases i.e. the worst cases, the complexity would be  **$O(m*(n-m+1))$**  .

2. We have studied multiple string matching algorithms in class and have found naive to be an inefficient algorithm.

- Can you suggest a solution to increase the efficiency of the naïve algorithm?

A simple way to improve performance of the Naïve Algorithm is to match pattern to text. If, say  $c$ , characters match and there is a mismatch at  $(c+1)$  index, we move the window to  $c+1$  index. This approach works for some cases, for example,

Text:        **AB****C**AABACAB  
Pattern:    **ABA**

Text:        **AB****C**AABACAB  
Pattern:        **ABA**

However this approach doesn't work in all cases, for example it skips the pattern in this case:

Text:        **AA****A****B**AE  
Pattern:    **AABA**

Text:        **AA****A****B**AE  
Pattern:        **AABA**

A better approach to Naïve Algorithm would be the **Knuth Morris Pratt (KMP) Algorithm**.

In Naïve Search Algorithm, upon a match or mismatch, we restart at the starting index of the window even though we may already have compared the next couple of characters of the window. This increases the complexity, especially in the worst cases of Naïve Search.

In KMP, however, we reduce the worst case complexity to  **$O(n)$**  by making sure we do not waste the next reads, following the starting index, even in the worst cases of Naïve approach.

This however involves some pre-processing. Before matching pattern to given text, we make Longest Proper Prefix LPS array, the same size as the length of the pattern. In this array, at each index, we examine the sub-pattern up until that index and store the length of the maximum matching proper prefix which is also a suffix of the sub-pattern.

Later on, while matching a given pattern to some text, we utilize the pattern's lps such that, whenever there is a match or mismatch, rather than starting from the starting+1 index, we determine how many characters would match anyway, using the LPS and move our cursor to the index accordingly.

For example, for a given pattern and some text,

Pattern [] = abaa

Text [] = ba**ab**ab**ab**ab**ab**a

The LPS [] would be,

LPS [] = [0, 0, 1, 1]

Pattern would be matched to text at indices 6 and 9.

At index 2, given in red, there would be 3 successful matches but the last character of the pattern wouldn't match to the text. In such cases, Naïve Approach would have been to start over from index 3. In KMP however, we will look at the corresponding lps table:

Sub-Pattern	a	ab	aba	abaa
LPS	0	0	1	1

So since *aba* matched, we have to start matching from the 1<sup>st</sup> index of the pattern [], skipping the 0<sup>th</sup> index. In the text, the window will simply move over, without overlapping any already-matched characters, directly to index 5. This will continue happening for each match and mismatch until we have exhausted the text.

Time complexity of KMP is **O (n)**.

- **Will it be feasible for all the cases?**

KMP is feasible for all cases because it reduces the complexity for even those cases which gave the worst results using Naïve approach to  $O(n)$ . This is because, due to the use of LPS, the loop only iterates over the text of length  $n$  once rather than going back over and over again.

- **Explain the working of your suggested solution by writing code.**

### **Pseudo Code for KMP**

This code would be in 2 parts. First, we need to compute the LPS array for any given pattern. Then we have to search for the pattern in the text, using the LPS array.

```
void generateLPS(int m, char* pattern, int*LPS)
{ int suffix = 0;
  for (int c=1; c<m; c++) //start at 1 because LPS[0]=0
  { if (pattern[c]==pattern[m])
    { suffix++;
      LPS[c]=suffix;
    }
    else
    { if (suffix==0)
      { LPS[c]=0; }
      else
      { suffix = LPS[suffix-1];
        c--; //stay on the index
      } } } }
```

What happens here is that given a pattern of length  $m$ , this function creates an array LPS, also of length  $m$ , and initializes it with zero. Then a for-loop is used to readjust LPS values for all the sub-patterns. The LPS value for the first, single-character-long sub-pattern is always zero so the for-loop iterates over from index 1 to  $m-1$  and adjusts their values. It does so by comparing the pattern from the start to the pattern itself, but in reverse (from the last index). For each successful match, it increases the LPS value for that sub-pattern by 1.

When there's a mismatch, if the value for the LPS of that index (suffix) is 0, that means there have been no successful matches. Hence it stores 1 against that sub-pattern in the LPS array.

Now for the actual search,

```
void KMP(char* text, char* pattern)
{ int m = strlen(pattern);
  int n = strlen(text);
  int LPS[m] = {0};
  generateLPS(m, pattern, LPS);
  int t=0;
  int p=0;
  while (t<n)
  { if (pattern[p]==text[t])
    { p++;
      t++;
    }
    if (p==m)
    { cout<<"Pattern found at index: "<<t-p<<endl;
      p=LPS[p-1];
    }
    else if (pattern[p]!=text[t] && t<n)
    { //mismatch after p matches
      if (p!=0)
        p=LPS[p-1];
      else
        t++;
    }
  }
}
```

```
int main()
{ char Text[] = "baabaababaabaaba";
  char Pattern[] = "abaa";
  KMP(Text, Pattern);
  return 0;
}
```

```
Pattern found at index: 2
Pattern found at index: 7
```



## Part B

### 1. KMP

a. Given below is an arbitrary pattern:

“aabaabcab”

You have to show the KMP prefix function for the above pattern.

Index	Sub-Strings	LPS
0	a	0
1	aa	1
2	aab	0
3	aaba	1
4	aabaa	2
5	aabaab	3
6	aabaabc	0
7	aabaabca	1
8	aabaabcab	0

So LPS array would be:

LPS [] = [0,1,0,1,2,3,0,1,0]

b. You are required to write a program in C++ which will show how you achieved the KMP prefix function for part A.

```
void generateLPS(int m, char* pattern, int*LPS)
{ int suffix = 0;
  for (int c=1; c<m; c++) //start at 1 because LPS[0]=0
  { if (pattern[c]==pattern[m])
    { suffix++;
      LPS[c]=suffix;
    }
    else
    { if (suffix==0)
      { LPS[c]=0; }
      else
      { suffix = LPS[suffix-1];
        c--; //stay on the index          } } } }
```

What happens here is that given a pattern of length  $m$ , this function creates an array LPS, also of length  $m$ , and initializes it with zero. Then a for-loop is used to readjust LPS values for all the sub-patterns. The LPS value for the first, single-character-long sub-pattern is always zero so the for-loop iterates over from index 1 to  $m-1$  and adjusts their values. It does so by comparing the pattern from the start to the pattern itself, but in reverse (from the last index). For each successful match, it increases the LPS value for that sub-pattern by 1. When there's a mismatch, if the value for the LPS of that index (suffix) is 0, that means there have been no successful matches. Hence it stores 1 against that sub-pattern in the LPS array.

## 2. Rabin-Karp

- a. For this question, you have to mention the number of spurious hits encountered if we run the Rabin-Karp algorithm on the text given below.

**Text: 3141592653589793**

**Assume you are looking for the pattern: P=26, and working modulo: q=11.**

**Note: Make sure you give a detailed description of your work to support your answer.**

Here, we are given

Text: 3141592653589793,  $n = 16$

Pattern: 26,  $m=2$

Mod =  $q = 11$

First, we calculate hash for the given pattern,  $P = 26$ .

$$\begin{aligned}\text{Hash}(26) &= 2 \cdot 10^1 + 6 \cdot 10^0 \pmod{11} \\ &= 20 + 6 \pmod{11} \\ &= 26 \pmod{11} \\ &= 4\end{aligned}$$

Now, we divide the text into sub-patterns of size  $m = 2$  and calculate hashes for each sub-pattern and match it to the hash of the pattern.

Sub-Pattern	Hash	Hits
31	9	-
14	3	-
41	8	-
15	4	Spurious
59	4	Spurious
92	4	Spurious
26	4	Actual hit– Matches Pattern
65	10	-
53	9	-
35	2	-
58	3	-
89	1	-
97	9	-
79	2	-
93	5	-

This shows that there are 3 spurious hits and 1 actual hit, with the sub-patterns of the given text. Spurious hits means that the hash of the sub-pattern matches with the hash of the pattern but the sub-pattern does not match to the pattern. These spurious hits were for sub-patterns **15**, **59** and **92**.

**b. You are required to write a program in C++ to code your solution of part A.**

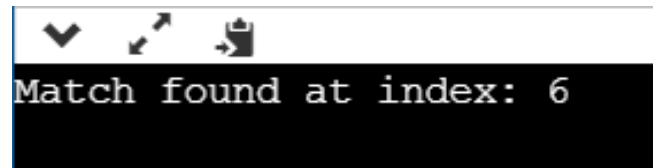
Code for implementing Rabin Karp involves 2 steps. First, we need to compute hashes for the pattern itself and all the sub-patterns of the text of size  $m$  and compare hashes. Next, for each hit, we need to compare the characters of the sub-pattern to the pattern itself to determine whether the hit was a spurious hit or an actual hit.

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

rabinkarp(char pattern[], char text[])
{
    int m = strlen(pattern);
    int n = strlen(text);
    int p = 0, t = 0; // hash value
    int q = 11;
    int units = pow(10, m-1);
    for (int c=0; c<m; c++)
    {
        p = p + pattern[c]*units;
        t = t + text[c]*units;
        units = units/10;
    }
    int x;
    for (int c=0; c<(n-m); c++)
    {
        if (p%q==t%q)
        {
            if (p == t)
                printf("Match found at index: %d", c);
        }
    }
}
```

```
x = t-((pow(10, m-1))*text[c]);  
t = 10*x;  
t = t + text[c+m];  
} }
```

```
int main()  
{ char text[] = "3141592653589793";  
  char pattern[] = "26";  
  rabinkarp(pattern, text);  
  return 0;  
}
```



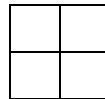
A terminal window screenshot with a black background and white text. The text reads "Match found at index: 6". Above the text, there are three small icons: a downward-pointing chevron, a cursor arrow, and a clipboard icon.

**3. Discuss how to extend the Rabin-Karp method to handle the problem of looking for a given  $m \times m$  pattern in an  $n \times n$  array of characters (The pattern may be shifted vertically and horizontally, but it may not be rotated.)**

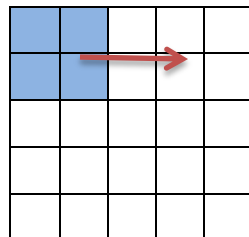
In order to extend Rabin-Karp to two dimensional pattern matching, given a pattern of length  $m \times m$  and text of length  $n \times n$ , we need to use the same technique as that of Rabin Karp. However, instead of the first  $m$  characters being our first window and then sliding over linearly to the next character for each change in window, for this, we need to start at  $0 \times 0$ , such that our first window would be  $0$  to  $(m-1) \times 0$  to  $(m-1)$ .

For example, given a pattern of length  $2 \times 2$  and a text of length  $5 \times 5$ , we start our iterative process with the following first window:

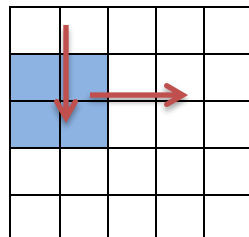
Pattern:



Text with First Window:



There would be a nested for-loop. The window will first move horizontally up until the last index of the window size maps to the  $(m-1)$ th index of the text. Then it will move an index downwards vertically and progress till the horizontal end once again. This will continue until the window has reached  $(m-1) \times (m-1)$ th index.



Last iteration:


The rest of the working would be similar to Rabin-Karp. For each new window of size  $m \times m$ , the loop will compute a hash. Then for those hashes which match the pattern's hash (hits), a loop would compare all the indices of the sub-pattern to the pattern. If ALL the indices match, it would be a MATCH, else it would be considered a spurious hit.

4.

- a. **Study a new string matching algorithm of your choice. Make sure you have not studied it in class before. Explain the working of the algorithm by discussing an example.**

Another method used for pattern matching is Boyer Moore Pattern Searching Algorithm. Similar to KMP, Boyer Moore preprocesses the pattern before comparing it to the text.

Boyer Moore is a combination of two approaches: bad character heuristic and good suffix heuristic. At every step, it uses the best of the two heuristics to decide how farther the window would slide from the current index for the next step.

Another way it differs from other algorithms is that rather than starting at the 0<sup>th</sup> index, it starts matching characters from the last index of the pattern. However, for text, we still start from the 0<sup>th</sup> index such that for a text of length n and pattern of length m, the first window of the text would be 0 to (m-1) however we will start comparing in such a way that we first match the (m-1) index of the text and to the (m-1) index of the pattern and then proceed to (m-2) and so on until we have compared all indices up to the 0<sup>th</sup> index.

**i. Bad Character Heuristic**

When pattern is matched to a window of the text sub-pattern, the first mismatched character is the bad character. We shift the window forward, character by character, until either the mismatched character of the string now matches to some part of the pattern or the window moves past the mismatched character.

For example, in the case that the mismatched character now matches,

Text:           A B A B B A C A B B C  
Pattern:       B A A B

Text:           A B A B B A C A B B C  
Pattern:       B A A B



For case 2, the window moves forward until the it moves past the mismatched character. For example,

**Text:**     A D A B B A C A B B C  
**Pattern:**   B A A B

**Text:**     A D A B B A C A B B C  
**Pattern:**   B A A B

**ii. Good Suffix Heuristic**

Similar to Bad Character Heuristic, this works for 3 cases. First, if a sub-pattern of the pattern matches a sub-pattern of the text and the same sub-pattern occurs in the pattern again, we move the window such that the second sub-pattern in pattern aligns with the matching sub-pattern in text. For example,

**Text:**     C B C A C B A B A A B B C  
**Pattern:**   A B A B A

**Text:**     C B C A C B A B A A B B C  
**Pattern:**   A B A B A

Second case occurs if a prefix sub-pattern in pattern matches a suffix sub-pattern in text. We align them, hence moving the window multiple characters forward.

**Text:**     C B C B C B C A B A A B B C  
**Pattern:**   C A C C C A

**Text:**     C B C B C B C A B A A B B C  
**Pattern:**   C A C C C A

These cases ensure that we do not compare each window to pattern naively and reduces no. of comparisons. If neither of these cases occur, the window is moved completely past the sub-pattern window in text as there are no matches in this window.

For example,

Text: C B C B C B C A B A A B B C  
Pattern: B A A B C

Text: C B C B C B C A B A A B B C  
Pattern: B A A B C

By using the best of these two heuristics, this algorithm helps move the window forward faster.

Time complexity of Bad Character Heuristic is  $O(n/m)$ ,  $n$  being the length of the text and  $m$  being the length of the pattern.

- b. You are required to write a program in C++ to code your algorithm for part A.

```
#include <stdio.h>
#include <string.h>
#include "math.h"
#include <bits/stdc++.h>
using namespace std;

#define characters 256

void badCharacters(string pattern, int m, int badC[characters])
{
    for (int c = 0; c < characters; c++)
        badC[c] = -1;
    for (int c = 0; c < m; c++)
        badC[(int) pattern[c]] = c;
}

void moore( string text, string pattern)
{
    int m = pattern.size();
    int n = text.size();
```

```
int badC[characters];
badCharacters(pattern, m, badC);
```

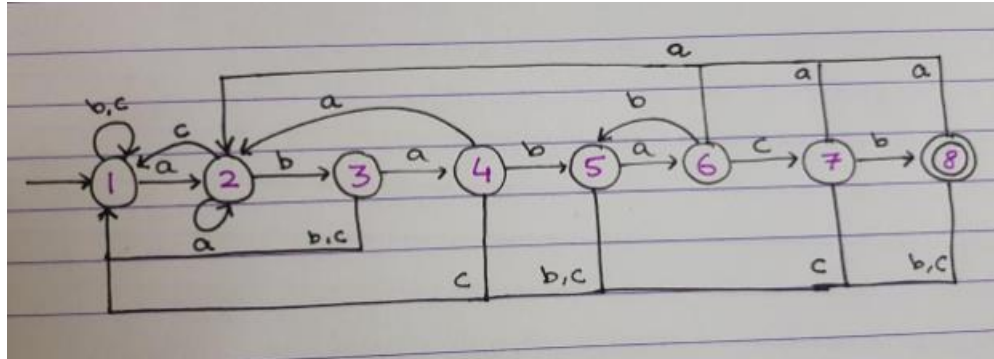
```
int c=0;
while (1)
{
    if (c>(n-m))
        break;
    int d = m-1;
    while (d>=0 && pattern[d]==text[c+d])
        d--;
    if (d<0)
    {
        cout<<"Pattern matches at shift: "<<c<<endl;
        c = c + (c + m < n)? m-badC[text[c + m]] : 1;
    }
    else
    { c = c + max(1, d - badC[text[c + d]]);
    }
}
}
```

```
int main()
{
    string txt= "ABAAABCD";
    string pat = "ABC";
    moore(txt, pat);
    return 0;
}
```

## Part C

- Design a finite automata machine along with state table for string matching that accepts all strings ending the form "ababacb".

Full FA:



State Table:

	a	b	c
1	2	1	1
2	2	3	1
3	4	1	1
4	2	5	1
5	6	1	1
6	2	5	7
7	2	1	8
8	2	1	1

2. Prove that string S is a substring of string A by drawing a state diagram of S, and using Finite Automata.

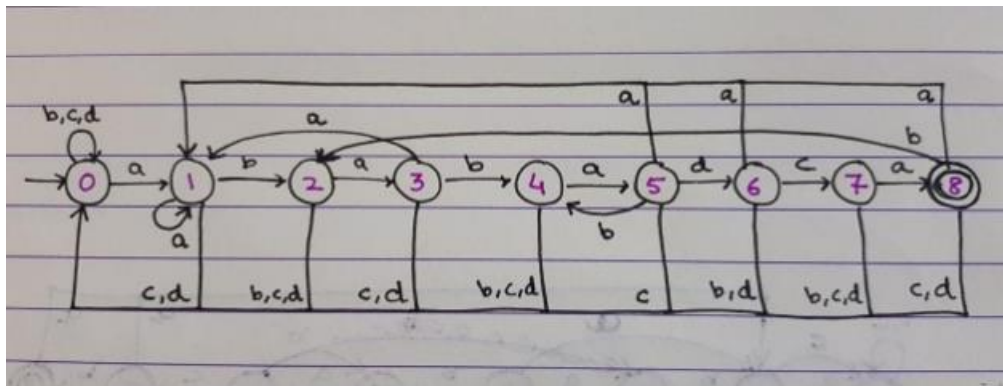
S: a b a b a d c a

- i. A: a b a b a b d c a a b a b a d c a
- ii. A: a b a b a d c a a b a b a d c a d
- iii. A: a b a b a b d c b a b a b a d c b

Answer Format- Answer should be according to the given format:

SOLUTION:

State Diagram:



State Table:

	a	b	c	d
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	0	6
6	1	0	7	0
7	8	0	0	0
8	1	2	0	0

**Tables of Solution Sets:**

**i. A: a b a b a b d c a a b a b a d c a**

<b>A</b>	<b>State</b>
a	1
b	2
a	3
b	4
a	5
b	4
d	0
c	0
a	1
a	1
b	2
a	3
b	4
a	5
d	6
c	7
a	8 (string matched)

**ii. A: a b a b a d c a a b a b a d c a d**

<b>A</b>	<b>State</b>
a	1
b	2
a	3
b	4
a	5
d	6
c	7

a	8 (string matched)
a	1
b	2
a	3
b	4
a	5
d	6
c	7
a	8 (string matched)
d	0

iii. A: a b a b a b d c b a b a b a d c b

A	State
a	1
b	2
a	3
b	4
a	5
b	4
d	0
c	0
b	0
a	1
b	2
a	3
b	4
a	5
d	6
c	7
b	0

No matches here.