String Matching algorithms

Group: 1	
Name: تسنيم علي الزهراني	**ID: 439008642**
Name: جمانة محمد الوافي	**ID: 439000464**
Name: منار محمد الامام	**ID: 439004660**
Name: ريم هاشم الشريف	**ID: 439006592**

We had **implemented** both the KMP pattern matching algorithm and the Longest Common Subsequence . And we **tested out** the algorithms on real datasets which are text files and biological sequence data.

Here **comparison** for the performances of each algorithm against naive algorithms for the same problems

❖ KMP pattern matching algorithm

Algorithm	**Preprocessing time**	**Matching time**	**Space**
Naive	0	O((n-m+1)m)	O(n+m)
Knuth-Morris-Pratt	O(m)	O(n)	O(m)

Naive string-matching algorithm:

The naive algorithm finds all valid shifts using a loop that checks the condition P[1 .. m] = T[s+1 .. s+m] for each of the n-m+1 possible values of s.

Input: pattern P of length m and text T of length n

Output: list of all numbers s, such that P occurs with shift s in T

NAIVE-STRING-MATCHER(T, P)

```
1   n = T.length
2   m= P.length
3   for s = 0 to n-m
4       if P[1.. m] == T[s+1 .. s+m]
5           print "Pattern occurs with shift" s
```

Procedure NAIVE-STRING-MATCHER takes time O((n-m+1)m) , and this bound is tight in the worst case. For example, consider the text string a^n (a string of n a's) and the pattern a^m. For each of the n-m+1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus O((n-m+1)m), which is O(n^2) , if m = [n/2]. Because it requires no preprocessing, NAÏVE-STRING-MATCHER's running time equals its matching time.

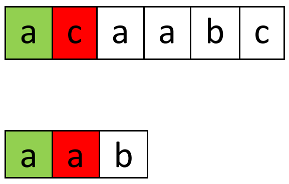
T is the text and P is the pattern, How the Algorithm Works, by finding all valid shifts with which a given pattern P occurs in each text T

Consider the following Text and Pattern

Text: acaabc n(length of T) = 6

Pattern: aab m(length of P) = 3

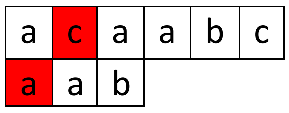
(a)



First character matches P[0]==T[0]

Second character does not match P[1]!=T[1] **(shift)**

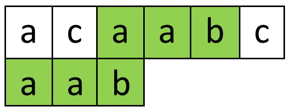
(b)



character does not match P[0]!=T[1]

So, check pattern from T[2] **(shift)**

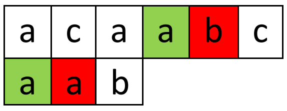
(c)



P[0]==T[2] , then Check for next character p[1] == T[3] , Check for next character p[2] == T[4]

All character matches, the pattern P is present in Text T with **2 shifts**

(d)



Check for the next occurrence of pattern P, p[0] == T[3] , p[1] != T[4] So will not go further because there will be no character left in Text T for Pattern P to match, so the pattern in Text only found **once** with **shift 2**

The Knuth-Morris-Pattern algorithm:

Input: pattern P of length m and text T of length n

Output: list of all numbers s, such that P occurs with shift s in T

KMP- MATCHER (T , P)

```
1   n = T.length
2   m = P.length
3   let π[1 .. m] be a new array
4   COMPUTE-PREFIX-FUNCTION (P, m, π)
5   i = 0, j =0
6   for i < n
7       if P[j] == T[i]
8           j++ , i++
9       if j == m
10          print "Found pattern at index" i - j
11          j =  π[j - 1]
12      elseif i < n && P[j] != T[i]
13              if j != 0
14                  j = π[j - 1]
15              else
16                  i++
```

From 1 to 5 take constant time, the for loop beginning in 6 runs n times as long as the length of the text T, so the running time of matching function is O(n)

Input: pattern P of length m

Output: table π[1 .. m]

COMPUTE-PREFIX-FUNCTION(P, m, π)

```
1   π[0] = 0
2   l = 0
3   i = 1
4   for  i < m
5         if P[i] == P[l]
6               l++
7               π[i] = l
8              i++
9        else
10               if  l != 0
11                    l = π[l - 1]
12               else
13                     π[i] = 0
14                     i++
15   return π
```

From 1 to 5 take constant time, for computing the prefix function, for in 6 runs m times, so the running time of compute prefix function is O(m)

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. This algorithm avoids computing the transition function , and its matching time is O(n) using just an auxiliary function π, which we precompute from the pattern in time O(m) and store in an array π[1 .. m].

Consider the previous example

Text: acaabc n(length of T) = 6

Pattern: aab m(length of P) = 3

Step1: Create π table (prefix table) for the following pattern

0	1	2
a	a	b

π table

0	1	2
0	1	0

(a)

a	c	a	a	b	c

0	1	2
a	a	b

Second character doesn't match P[1]!=T[1] , consider π[0] value and since its 0 we must compare first character in pattern with next character in text

(b)

a	c	a	a	b	c
a	a	b			

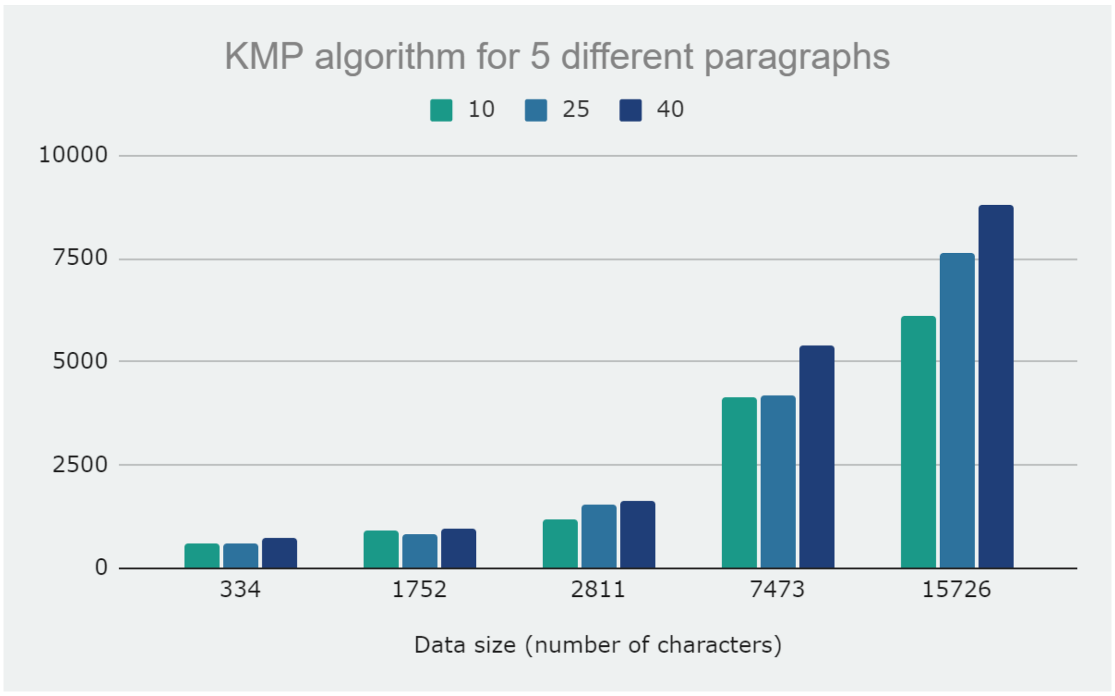
So, the pattern in Text found **once** with **1 shift** at **index 3**

KMP practical time analysis

We have seen in the code the time analysis that the KMP algorithm takes at worst case O(m+n) where m is the length of the pattern (the string we want to search for) and n are the length of the text (which is the input file we have), we want to find string match between them.

To examine this fact, we took a sample from five different paragraphs, the set represent the size of each paragraph:{15726, 7473, 2811, 1752, 334}, we also took a 3 different data in each paragraph, the set represent the size of each data:{10, 25, 40}.

After running the code to find the string match, here is the chart which summarizes the experiment results



The chart shows that the growth of time and data is an addition of the sizes of two strings, surly, it is not the exact addition value since these are practical results, but it follows the same logic and is accurate.

Advantages and Disadvantages of KMP algorithm:

Advantages	Disadvantages
• KMP is one of the most useful algorithms in this field. • One of The best algorithms for linear time for exact matching. • Good for handling large files. • Efficient implementation.	Doesn’t work well if the size of the alphabets increased, more chances of mismatching.

❖ The LCS algorithm

Algorithm	**Space Complexity**	**Time complexity**
Naive	O(1)	O(m*2^n)
The Longest Common Subsequence	O(mn)	O(mn)

Naïve algorithm:

Input: X, Y arrays of lengths n and m

Output: LCS

NAIVE-ALGORITHM(X, Y)

```
1  S 🡐 all subsequences in X
2  maxLength 🡐 0
3  for s ∈ S
4      if isSubsequence(s, Y)
5          maxLength 🡐 max(maxLength, length(s))
6      end
7  end
8  return maxLength
```

Time complexity = all subsequences in X * the time to test if a given sequence is a subsequence of Y. The total number of subsequences in X is 2^n. To check whether a sequence is a subsequence of Y takes O(m) time in the worst-case. Then, the naive algorithm would take O(m*2^n) time. Memory Complexity: The algorithm uses one extra variable to keep the current maximum length. Thus, it has O(1) memory complexity.

The Longest Common Subsequence algorithm:

Let **X** = x1, x2, .. xm and **Y** = y1, y2, .. yn

c[i, j]: the length of an LCS between x1, x2, .. xi and Y = y1, y2, .. yj

c[i, j] can be computed with developing a recursive formula by converting it to an iterative one as follows:

$$c[i,j] = \begin{cases} \mathrm{c[i-1, j-1]+1} & \text{if i = 0 and j = 0} \\ \mathrm{max(c[i,k]+c[k,j]+1)} & \text{if i != 0 and j != 0} \end{cases}$$

Input: X, Y arrays of lengths n and m

Output: LCS

LCS-LENGTH(X, Y)

```
m = X.length
n = Y.length
let LCS[0 ..m,0 ..n] be new tables
for i = 1 to m
    LCS[i, 0] = 0
for j = 0 to n
    LCS[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if xi == yj
            LCS[i, j] = LCS[i – 1, j - 1] + 1
        elseif LCS[i – 1, j] >= LCS[i, j - 1]
            LCS[i, j] = LCS[i – 1, j]
        else LCS[i, j] = LCS[i, j - 1]
return LCS
```

The running time of the procedure is O(mn) at worst, best and average case, since each table entry takes O(1) time to compute.

Input: X, Y arrays of lengths n and m

Output: LCS

```
PRINT-LCS(X, Y, m, n, LCS[][])
1   longest = LCS[m][n]
2   lcs[longest + 1]
3   i = m, j = n
4   while i > 0 && j > 0
5       if X[i - 1] == Y[j - 1]
6           lcs[longest - 1] == X[i - 1]
7           i-- , j-- , longest--
8       elseif  lcs[I - 1][j] > LCS[i][j - 1]
9           i--
10      else
11          j--
12  print lcs
```

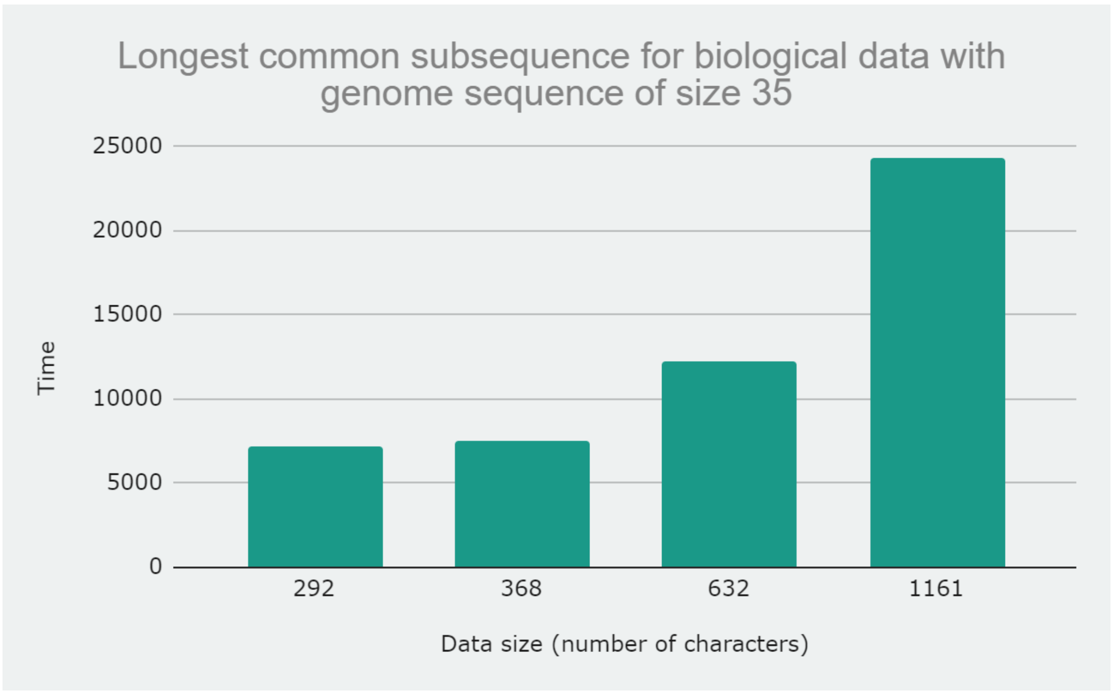
The procedure takes time O(m + n) since it decrements at least one of i and j in each recursive call.

In conclusion, we viewed two algorithms – one naive and one efficient algorithm. The **naive algorithm** checks each subsequence of X if it is a subsequence of Y and has exponential time complexity. **The efficient algorithm** uses the dynamic programming algorithmic scheme. It stores the intermediate subproblems' results in a table and has polynomial time complexity.

Longest common subsequence practical time analysis

It has been shown through code time analysis that the LCS algorithm takes at worst case O(m*n) where m and n are the lengths of the string, we want to find the longest common subsequence between them.

To examine this fact, we took a sample biological genome sequence of size 35 and run the code to find the LCS between our sample and multiple genome sequences of various sizes, the chart below summarizes the experiment results:



The chart shows indeed that the growth of time and data is a multiplication of the sizes of two strings, surly, it is not the exact multiplication value since these are practical results and can be affected by other factors, but it follows the same logic and is accurate.

why using iteration and not recursive method?

Advantages and Disadvantages of LCS algorithm using iteration:

Advantages	Disadvantages
● iteration is usually faster than recursion. ● we don't need to initialize the matrix to all -1's as in recursive approach. ● iteration through matrix save us work since we do not need to spend extra time making sure which cell have been computed and what is not yet computed	Iteration fills in the entire array even when it might be possible to solve the problem by looking at only a fraction of the array's cells.

Credits/References:

-Introduction to Algorithms, Third Edition (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein).
- Longest Common Subsequence (LCS) Problem - DP Algorithm (simplecodehints.com)

-Longest common subsequences: Identifying the stability of individuals' travel patterns Adrian C. Prelipcean∗1, Yusak O. Susilo1 and Gy˝oz˝o Gid´ofalvi2 1 Department of Transport Science, KTH, Sweden 2Department of Urban Planning and Environment, KTH, Sweden

-KMP Algorithm in detail, algo-en, https://labuladong.gitbook.io/algo-en/i.-dynamic-programming/kmpcharactermatchingalgorithmindynamicprogramming

-Knuth-Morris-Pratt Algorithm, BSC, http://www.btechsmartclass.com/data_structures/knuth-morris-pratt-algorithm.html