Knuth-Morris-Pratt Algorithm:

With the abettance of Knuth-Morris-Pratt, it was possible to achieve a linear-time string-matching method in contrast to previously stated polynomial time algorithms by demolishing the need for back-tracking. **Using an auxiliary function π, the matching time achieved is Θ(n), and pre-computation time from the pattern results in time Θ(m) which can be kept in array ***π[1…m]***← (Paraphrased Reference CLRS: 32.4 page: 1002). As a result, total time complexity becomes O(m + n).

The Prefix Function π:

A pattern's prefix function ***π*** contains information about how the pattern compares to shifts of itself, giving it the upper hand by avoiding unnecessary shifts like other algorithms. The ***π*** function has the ability to provide a tabular view which is commonly coined as ***LPS***, or ***Longest Prefix Suffix*** table.

An example (Fig 1.1(a)) of a shift ***s* is shown by the Naive String Matching algorithm when a template containing the pattern ***P*** = *xyxyxzx* is applied against a string ***T*** that results in ***q = 5*** for the characters that have matched successfully. However, the 6th pattern character mismatches with the corresponding text character which practically determines ***q*** characters preceding it has matched, this information allows the determination of the corresponding text character and that certain shifts are invalid.
Since the first pattern character *(x)* in the example in the figure would be aligned with a text character that we know does not match the first pattern character but does match the second pattern character *(y)*, the shift ***s + 1*** is inherently erroneous. However, the shift ***s′ = s + 2*** depicted in part (1.1(b)) of the figure aligns the first three pattern characters with three required text characters.

****The response to the following queries is generally helpful:

Given that pattern characters ***P[1… .q]*** match text characters ***T[s+1…s+q]***,

what is the least shift ***s' > s*** such that for some ***k < q,***

P[1… .k] = T[s' + 1… .s' + k] ***…… (1)***

where ***s' + k = s + q ?***

In other words, knowing that $P_q \sqsupset T_{s+q}$, we want the longest proper prefix P_k of P_q which is also a suffix of T_{s+q}. (Since $s' + k = s + q$, if we are given s and q, then finding the smallest shift s' is tantamount to finding the longest prefix length k.) By adding the difference $q - k$ in the lengths of these prefixes of P to the shift s to arrive at our new shift s', so that $s' = s + (q - k)$. In the best case, $k = 0$, so that $s' = s + q$, and we immediately rule out shifts $s + 1, s + 2, \ldots, s + q - 1$. In any case, at the new shift s' we don't need to compare the first k characters of P with the corresponding characters of T, since *equation (1)* guarantees that they match.

We can precompute the necessary information by comparing the pattern against itself, as Figure 1.1(c) demonstrates. Since $T[s' + 1 \ldots .s' + k]$ is part of the

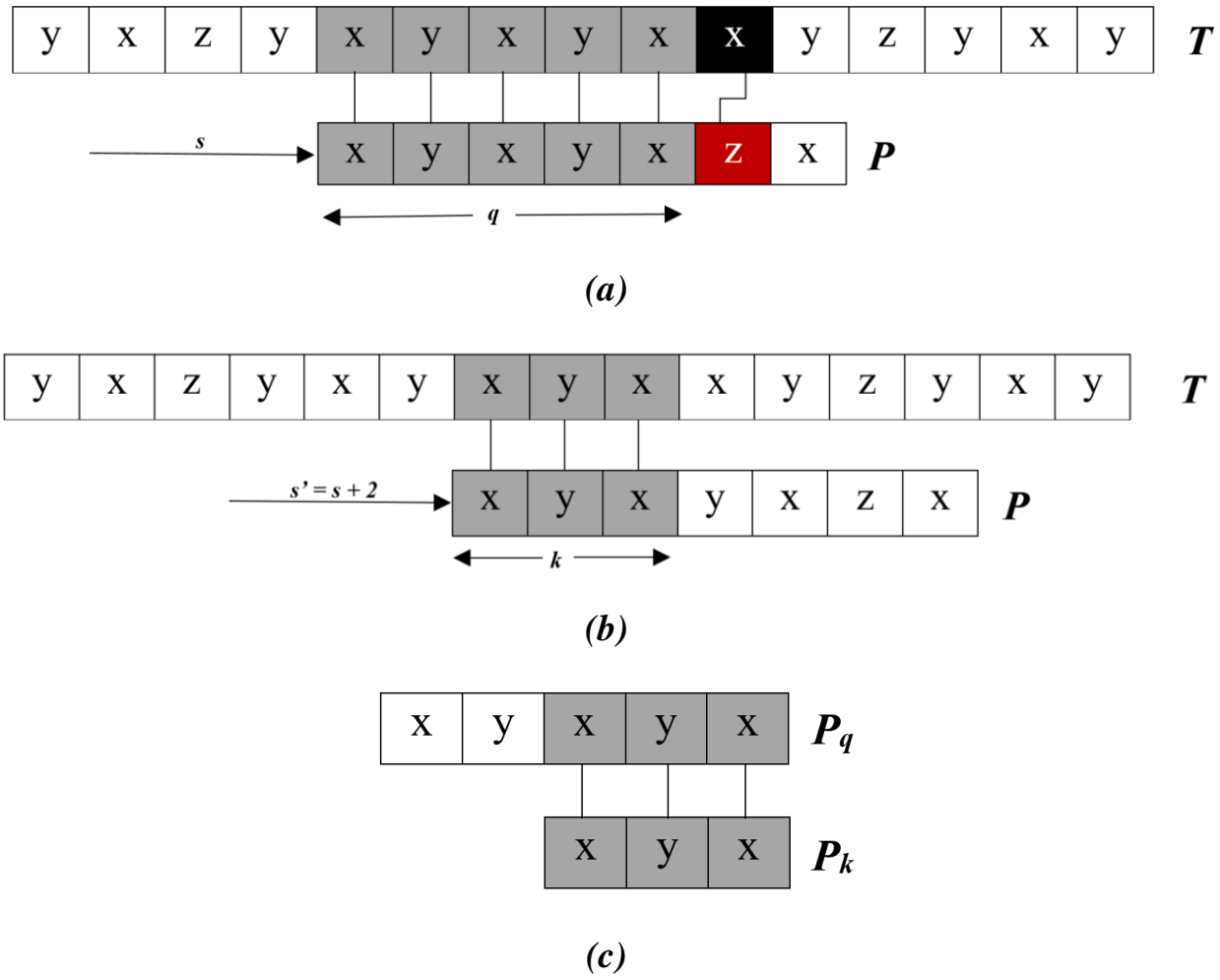


Figure 1.1 The prefix function π. (a) The pattern $P = xyxyxzx$ aligns with a text T so that the first q = 5 characters match. Matching characters, shown shaded, are connected by vertical lines. (b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything we know about the text and therefore is potentially valid. (c) We can precompute useful information for such deductions by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P_5 is P_3. We represent this precomputed information in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shifts, the next potentially valid shift is at $s' = s+(q- \pi[q])$ as shown in part (b).

known portion of the text, it is a suffix of a string P_q. Therefore, we can interpret *equation (1)* as asking greatest **k < q** such that $P_k \sqsupset P_q$. Then, the new shift ***s'=s+(q-k)*** is the next potentially valid shift. We will find it convenient to store, for each value of ***q,*** the number of ***k*** matching characters at the new shift ***s'*** rather than storing say, ***s'-s***.

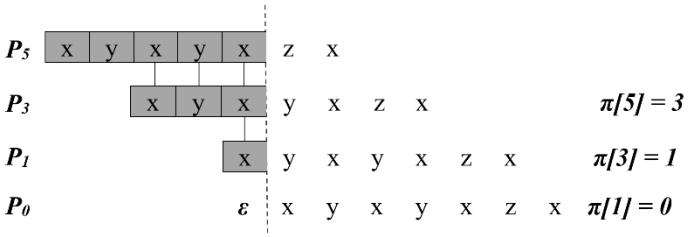
We formalize the information that we precompute as follows. Given a pattern ***P [1…m]***, the prefix function for the pattern ***P*** is the function π : {1,2,…,m}→ ***{0,1,…,m-1}*** such that

$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$.**** ← Section can be redacted

That is, ***π[q]*** is the length of the longest prefix of ***P*** that is proper suffix of P_q. Figure 1.2(a) gives the complete prefix function ***π*** for the pattern *xyxyxzx*. ** ←(Paraphrased Reference CLRS: 32.4 page: 1005).

i	1	2	3	4	5	6	7
P[i]	x	y	x	y	x	z	x
π[i]	0	0	1	2	3	0	1

(a)



(b)

Figure 1.2 An illustration for the pattern $P = xyxyxzx$ and $q = 5$. (a) The π function for the given pattern. Since $\pi[5] = 3$, $\pi[3] = 1$, and $\pi[1] = 0$, by iterating π we obtain $\pi^*[5] = \{3, 1, 0\}$. (b) We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_5; we get matches when $k = 3$, 1, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_5. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_5. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k: k < 5 \text{ and } P_k \sqsupset P_5\} = \{3, 1, 0\}$. Lemma 32.5 claims that $\pi^*[q] = \{k: k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

**The pseudocode below gives the Knuth-Morris-Pratt matching algorithm as the procedure KMP-MATCHER. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

KMP-MATCHER (T, P)

```
1   n = T.length
2   m = P.length
3   π = COMPUTE-PREFIX-FUNCTION (P)
4   q = 0                                   // number of characters matched
5   for i = 1 to n                          // scan text from left to right
6       while q > 0 and P[q+1] ≠ T[i]
7             q = π[q]                      // next character doesn't match
8       if   P[q+1] == T[i]
9             q = q + 1                     // next character matches
10    if q == m                             // is all of P matched?
11          print "Pattern occurs with shift" i – m
12          q = π[q]                        // look for the next match
```

COMPUTE-PREFIX-FUNCTION (P)

```
1   m = P.length
2   let π[1 .. m]  be a new array
3   π[1] = 0
4   k = 0
5   for q = 2 to m
6       while k > 0 and P[k+1] ≠ P[q]
7             k = π[k]
8       if   P[k+1] == P[q]
9             k = k + 1
10        π[q] = k
11    return  π
```

These two procedures have much in common because both match a string against the pattern ***P :*** KMP-MATCHER matches the text ***T*** against ***P***, and COMPUTE-PREFIX-FUNCTION matches ***P*** against itself. **← (CLRS: 32.4 page: 1005).

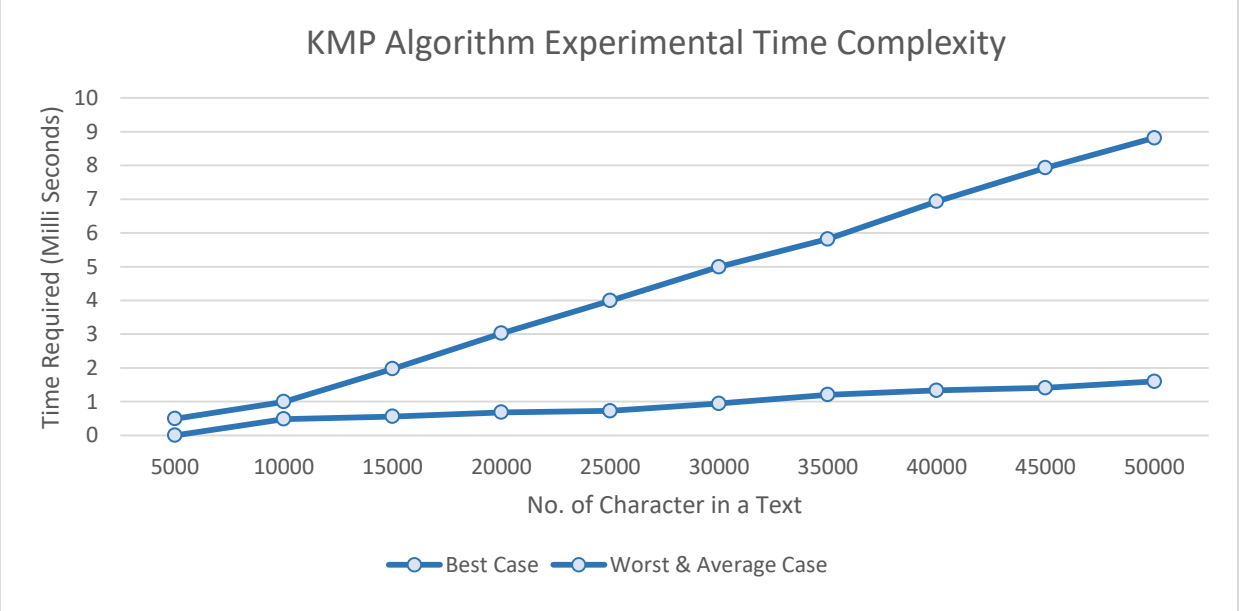
Running-time analysis:

KMP algorithm exploits the degenerating property (a pattern's tendency to repeat the same sub-patterns) to reduce complexity in contrast to others.

Preprocessing entails generating an auxiliary ***LPS*** array with the same size as the pattern string that corresponds to the pattern string. After asymptotic analysis was done it was found that pre-processing requires ***O(m)*** time and space complexity, and searching requires ***O(n + m)*** time complexity (regardless of the size of the alphabet).

When the pattern is absent from the text string, the time complexity is ***O(n)***, which is the best-case scenario.
Average time complexity yields ***O(n)*** ; when ***n > > m***
Theoretically, ***KMP-MATCHER*** has a worst-case and average-case time complexity of ***O(n + m)***.
Note that above, ***n*** is the length of the text string and ***m*** is the length of the pattern to be found.

The outcomes of the theoretical analysis of time complexity are reflected in experimental evidence. Using ***C++*** programming and plotting the running time of the algorithm produced graph retains resemblance to figurative plot of its asymptotic analysis. Hence, it establishes the fact that the KMP algorithm has a linear time complexity.

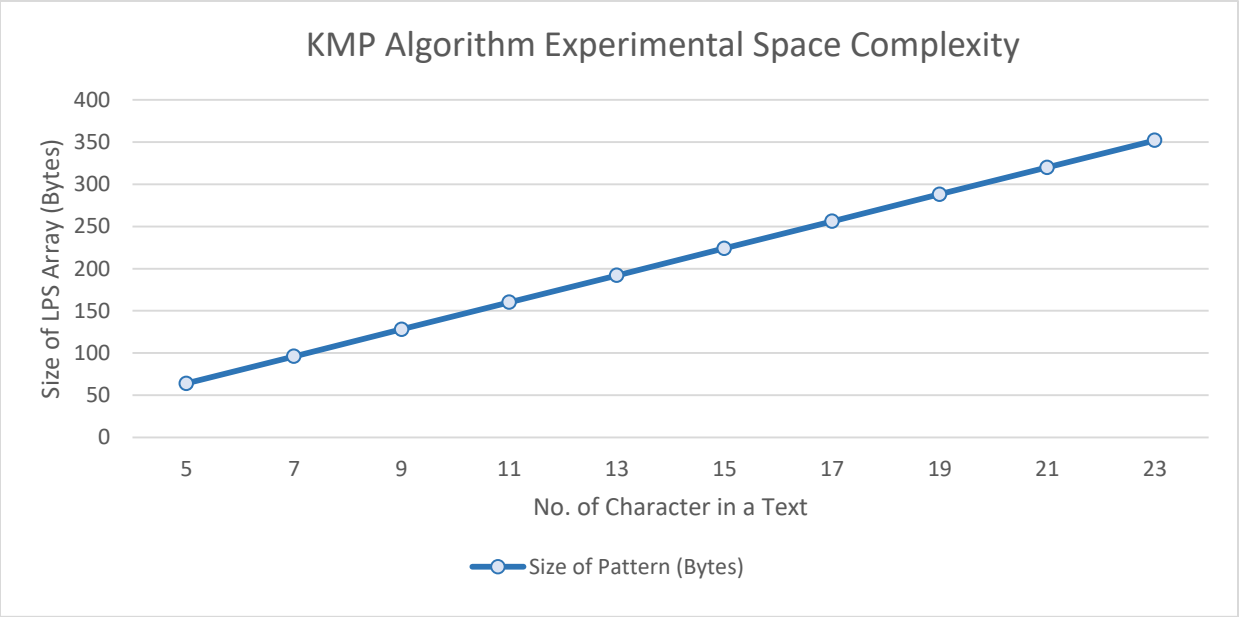


Patterns Used:

Best Case: RRRRRRRRUET
Worst Case: RRRRRRRRUETX

Space Complexity:

Space complexity of KMP is **O(m)**, where **m** is the size of the LPS array.
The experimental Space complexity found mirrors the same result which can be seen from the graph below.



Comparative Analysis between previous Algorithms and KMP :

From the theory part it can be seen that KMP has best and average case complexity **O(n)** in contrast to Rabin Karp having **O(m + n)** which is higher than KMP. Rabin-Karp in worst case has **O(mn)** but can reach to **O(n^2)** if pattern and text string matches, resulting to a quadratic function which is resolved by KMP making its worst case **O(m + n)** i.e. linear time complexity.
Naïve String Matching Algorithm has best case complexity of **O(n-m)** which can be resulted to **O(n)** however, it has worst case complexity **O((n - m + 1) * m)** that can approach to Quadratic time complexity in comparison to KMP matcher.
So by comparing all the complexity theoretically a conclusion can be drawn that KMP is the best among all the algorithms.

But same can't be said for the space complexity because space complexity of KMP is **O(m)** compared to Naïve and Rabin-Karp which is **O(1)** i.e. constant which can significantly large depending on the pattern size.

Now coming to the practical analysis, the graphs drawn from each algorithm mirrors their theoretical analysis (with some errors here and there).
From the graph of KMP we can see, during the best, average and worst case the graph is linear in contrast to Rabin-Karp and Naïve which is linear at best and average case but can extend to quadratic in worst case.
In worst case of previous two algorithms, after 10000 character Text String input for a given pattern the graph starts to gradually show its quadratic nature which is absent in case of KMP matcher.

Coming to the discussion of space complexity, KMP has space complexity graph which is proportional to the size of the pattern however Naïve and Rabin-Karp has constant space complexity mostly parallel to axis which has size of pattern.

Advantages of KMP over other Two string matcher algorithms:

1. Greatest advantage of KMP is worst case linear time complexity.
2. Efficiency of KMP is far higher than other two, it improves efficiency, especially for lengthy texts and intricate patterns, by reducing pointless character comparisons.
3. The KMP algorithm can be used for a range of pattern matching tasks, such as detecting the frequency of a pattern within a text or simultaneously searching for numerous patterns hence it's versatile.
4. Mostly there is no worst case for accidental inputs.

Disadvantages of KMP over other Two string matcher algorithms:

1. KMP Algorithm will be complicated in a scenario where the need for resources, like as CPUs and memory for processing very big datasets is large

Components Used :

Processor : AMD Ryzen 5 3500X (Base: 3.6Ghz Boost: 4.1Ghz)
RAM: PNY XLR8 (8GB * 2 DDR4 3200MT/s)
SSD: Patriot Burst 240GB NVME 2.0
GPU: AMD Radeon RX 5600XT
Language Used: C++
Text Editor: VS Code
Code Borrowed From: Geeks For Geeks https://www.geeksforgeeks.org/