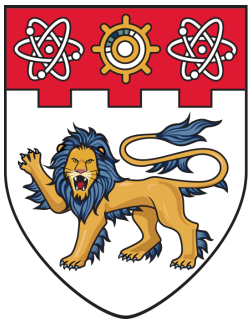

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ2001: Algorithms

# Project 1: Searching Algorithm


Aurelio Jethro Prahara (U1921390C)

Bhargav Piyushkumar Singapuri (U1920026B)

Nicholas Goh (U1920857G)

Timothy Tan Choong Yee (U1920534C)

School of Computer Science and Engineering

# Introduction

The Brute Force Algorithm is often taken as the most basic search algorithm. For this assignment, we have chosen to implement and analyse Brute Force and the *Knuth-Morris-Pratt* (KMP) Algorithm. All implementation was done in Python.

# Brute Force Algorithm

## Concept

The Brute Force Algorithm works by comparing every character in the text (haystack) and the search pattern (needle). If every character in the needle is compared successfully, then the pattern has been found successfully. If a mismatch occurs at a particular index, then the needle is shifted up by one index, and each character is compared again.

## Python Implementation

```
for haystackIdx in range(0, haystackLength - needleLength + 1):

    needleIdx = 0
    while (needleIdx < needleLength):
        if haystack[needleIdx + haystackIdx] != needle[needleIdx]:
            break
        else:
            needleIdx += 1

    if needleIdx == needleLength:
        found  = True
        print("Pattern found at index", haystackIdx)

if found is False:
    print("Pattern not found")
```

Fig. 1: Code snippet of our brute force implementation

The variables `haystackIdx` and `needleIdx` are first initialised to traverse both strings. If a mismatch between these two indices were to occur, then `needleIdx` would reset to 0, and this character would be compared with `haystack[needleIdx + haystackIdx]`. When `needleIdx` traverses the entire needle, then the search pattern has been found and the function prints the index of the haystack at that point.

## Analysis

Let $h, n$ be the length of the haystack and the needle respectively. For a complete search, we would need to go through all characters of the haystack. This means that we need to check $h - n + 1$ substrings. In the inner loop, we would need to compare each character of the needle to each character of the substring of the haystack.

### Best Case Time Complexity

In the best case scenario, in the inner loop, the first character is never the same as the first character of the substring, i.e. the first character of the needle does not exist in the haystack. This results in a time complexity of $\mathcal{O}(1)$ in each inner loop, and an overall time complexity of $\mathcal{O}(h - n)$.

### Average Case Time Complexity

In the case of searching for DNA patterns, there are only four available characters to choose from. The probability of a match is $\frac{1}{4}$. This can be modelled by a geometric distribution $P(X = r) = p(1 - p)^{r-1}$,

where $X$ is the random variable denoting the number of comparisons that have been after encountering the first mismatch. The probability of having a mismatch is given by $p = 3/4$ and the expected number of mismatches is $E(X) = \frac{1}{p} = \frac{4}{3}$. The Brute Force makes these comparisons $h - n + 1$ times. Hence, the average case time complexity can be given by $\frac{4}{3}(h - n + 1)$, which is simplified to $\mathcal{O}(h - n)$.

### Worst Case Time Complexity

In the worst case, the Brute Force Algorithm would have to traverse through the entire length of the needle making the time complexity $\mathcal{O}(n)$. Hence, the worst case time complexity for brute force is $\mathcal{O}(h-n+1)(n) = \mathcal{O}(hn)$. There is nothing to analyse for space complexity since we do not need to store any additional data since there is no preprocessing done.

## Knuth-Morris-Pratt (KMP) Algorithm

### Concept

Figures 2 and 3 show an example of how KMP handles a mismatch case by skipping over irrelevant characters to the next occurrence of 'A'. This is a key improvement over Brute Force, which only shifts the search pattern by one index with each mismatch.

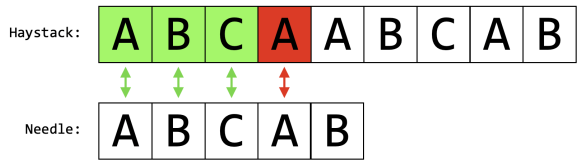


Fig. 2: Comparisons up to first mismatch.

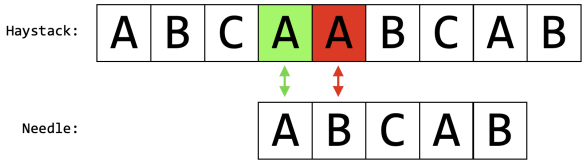


Fig. 3: Comparison after first mismatch.

In order to facilitate this, the substring first needs to be preprocessed to identify the length of longest recurring patterns within the needle (referred to as prefixes and suffixes). This allows the algorithm to form a table of values that indicate the next index in the haystack that the needle should be compared to. Table 1 depicts the corresponding lookup table for the example pattern above. If a mismatch occurs at a 'A' within the haystack, we are to remain at that index, and compare it with index 1 of the needle. In the same way, if the mismatch occurs at 'B', we compare it with index 2 of the needle. In all other cases, we just shift the search pattern right by one increment.

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| A | B | C | A | B |
| **0** | **0** | **0** | **1** | **2** |

Table 1: KMP Lookup table.

### Python Implementation

The function shown in Figure 4 generates `processedNeedle`, which is the KMP lookup table. We calculate the value such that if the `needle[leftCursor]` and `needle[rightCursor]` matches, we increment `leftCursor` and assign this value to the `processedNeedle`, and then increment `rightCursor`. If a mismatch occurs and `leftCursor` is pointing to the first character, we reinitialise it. This constructs an array storing the lengths of all prefix/suffix pairs in the needle. If a mismatch occurs at index $a$, `processedNeedle[a - 1]` gives the index number of the needle that should be used to make the next comparison.

```python
def preprocessNeedle(needle, N, processedNeedle):
    leftCursor = 0
    rightCursor = 1

    while rightCursor < N:
        if needle[leftCursor] == needle[rightCursor]:
            leftCursor += 1
            processedNeedle[rightCursor] = leftCursor
            rightCursor += 1
        else:
            if leftCursor == 0:
                processedNeedle[rightCursor] = 0
                rightCursor += 1
            else:
                leftCursor = processedNeedle[leftCursor - 1]
```

Fig. 4: Preprocess Function for KMP

```python
def KMPSearch(needle, haystack):
    H = len(haystack)
    N = len(needle)
    cursorN = 0
    cursorH = 0

    processedNeedle = [0]*N
    preprocessNeedle(needle, N, processedNeedle)

    while cursorH < H:
        if needle[cursorN] == haystack[cursorH]:
            cursorH += 1
            cursorN += 1
        if cursorN == N:
            print(f'Pattern found at index: {cursorH - cursorN}')
            cursorN = processedNeedle[cursorN - 1]
        elif cursorH < H and needle[cursorN] != haystack[cursorH]:
            if cursorN == 0:
                cursorH += 1
            else:
                cursorN = processedNeedle[cursorN - 1]
```

Fig. 5: KMP Search Function

## Analysis

### Preprocessing Time

Since we do not need to do the comparison for those that are already matched, we only need to do one comparison for each character in the haystack. This means that the time complexity of the matching will be $\mathcal{O}(h)$. However, time is taken to do the preprocessing. This is done in $\mathcal{O}(n)$ time, because the preprocessing function traverses through the needle to calculate the value of prefix/suffix pairs.

### Time Complexity

Since the preprocessing and the matching happens linearly, the time complexity of the algorithm is $\mathcal{O}(h+n)$. In the given problem, the best case and worst case time complexity for KMP remains the same, because we want to find all occurrences of the needle in the haystack. Hence, the entire text needs to be traversed, even if a match has previously been found. This means that the best, worst, and average case time complexity remains $\mathcal{O}(h+n)$.

### Space Complexity

The improvement to a linear time complexity in the worst case from $\mathcal{O}(hn)$ to $\mathcal{O}(h+n)$ comes with a trade-off in space complexity. In the case of brute force, there is no need to store any data. However, the efficiency in KMP comes from the existence of the lookup table which is the array which tells us how many characters we do not need to match. The size of this array is equivalent to the length of the needle that we are searching for. Thus, the space complexity for KMP would be $\mathcal{O}(n)$.

## Empirical Analysis

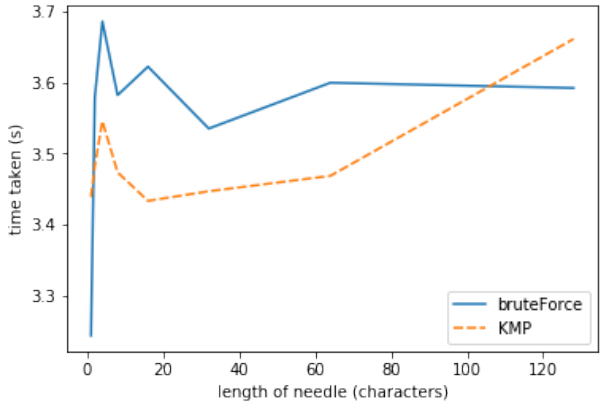


Fig. 6: Fixed haystack and varied needle length

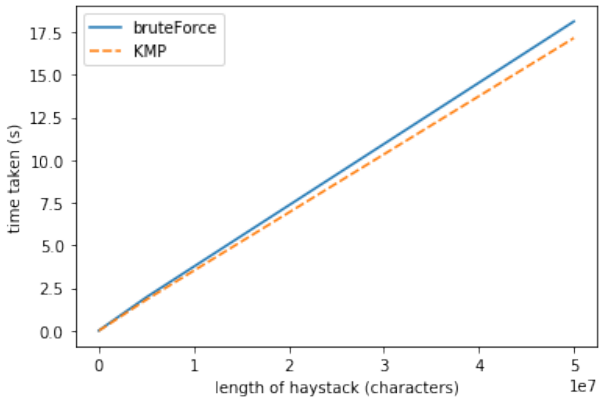


Fig. 7: Fixed needle, varied haystack length

Figure 6 shows the performance of both algorithms with varied needle lengths. When the randomly generated needle reaches a certain length, it has a lower likelihood of being present in the haystack. This means adding more characters to needle does not have a large effect on the the index of mismatch, resulting in time taken for brute force to plateau as seen. However, preprocessing time for KMP still increases linearly as length of needle increases.

Looking at Figure 7, there is little difference in timings of the algorithms, suggesting that they both take O(h). This does make sense since $n$ is negligible as $h$ increases so $\mathcal{O}(h+n) \approx \mathcal{O}(h-n)$.

## References

Kmp algorithm for pattern searching. (2019). `https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/`

Knuth–morris–pratt algorithm. (2020). Wikimedia Foundation. `https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm`

Naive algorithm for pattern searching. (2019). `https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/`

## Contributions

Aurelio Jethro Prahara: Algorithm analysis, including empirical analysis and report writing.

Bhargav Piyushkumar Singapuri: Research and analysis for the *Two Way String Matching* Algorithm, presentation.

Nicholas Goh: Implementation of KMP Algorithm, as well as algorithm analysis, presentation.

Timothy Tan Choong Yee: Implementation of Brute Force Algorithm, as well as report writing.