NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ2001: Algorithms

Project 1: Searching Algorithm

Aurelio Jethro Prahara, Bhargav Singapuri, Nicholas Goh, Timothy Tan

School of Computer Science and Engineering

Introduction

We implemented and analysed two additional algorithms in addition to the brute force algorithm that was required. One of them is the *Knuth-Morris-Pratt* (KMP) algorithm with the other algorithm being *Two-way String Matching*. All of the 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 (Figure 1), then the needle is shifted up by one index, and each character is compared again (Figure 2).

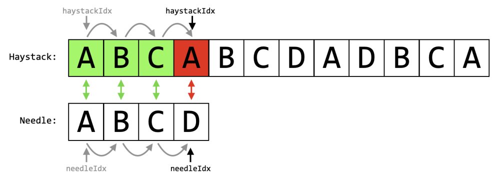


Figure 1: Brute force comparisons up to first mismatch.

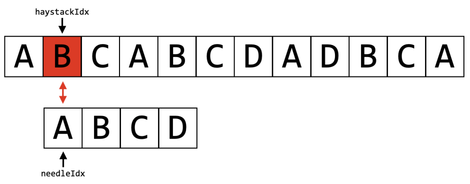
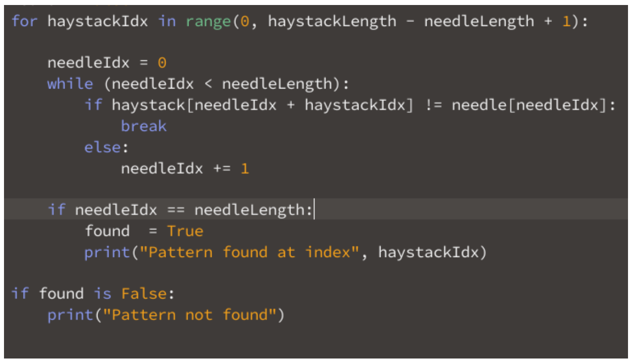


Figure 2: Brute force comparison after first mismatch.

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")
```

Figure 3: Code snippet of our brute force implementation

We first initialised a `haystackIdx` for the substring of the haystack that we are going to compare our needle with. This algorithm will begin by comparing the first letter of the text with each letter of the substring, if they are the same, it will move on to the second letter and so on until the entire needle is traversed successfully. in which case, it is matched or the letters are no longer the same, which results in the *if* case and hence, breaking. Then, the position of the substring will move by one character.

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. 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 character doesn't exist in the haystack. This would just be $\mathcal{O}(1)$ in the inner loop. However, in the worst case, it would have to traverse through the entire length of the needle making the time complexity $\mathcal{O}(n)$. Hence, this means that for this brute force algorithm, the best case time complexity is $\mathcal{O}(h-n+1)$ while the worst case is $\mathcal{O}(h-n+1)(n)$. This worst case scenario can be simplified to being $\mathcal{O}(hn)$ and the best case being $\mathcal{O}(h)$.

There is nothing to analyse for space complexity since we do not need to store any data.

Knuth-Morris-Pratt (KMP) Algorithm

Concept

One of the major drawbacks of the Brute Force Algorithm is that when a mismatch occurs, there is no built in check that prevents the algorithm from double checking characters that have previously been compared. The KMP Algorithm circumvents this inefficiency by comparing the first character of the pattern with the next occurrence of that character in the haystack. Figures 4 and 5 show an example of how KMP handles a mismatch case by skipping over previously compared characters to the next occurrence of the character 'A'.

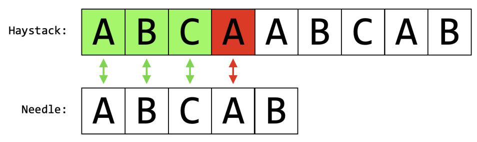


Figure 4: Brute force comparisons up to first mismatch.

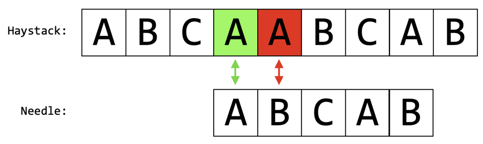


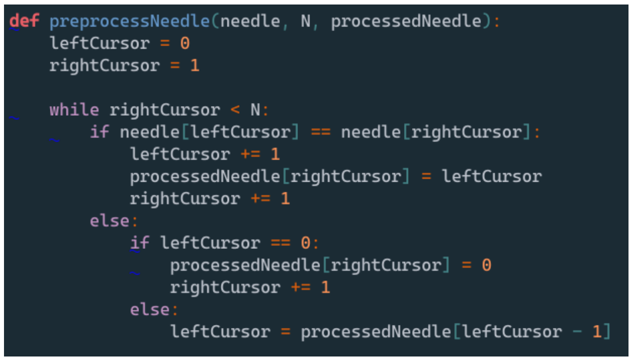
Figure 5: Brute force comparison after first mismatch.

In order to facilitate this, the substring first needs to be preprocessed, to identify 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 is the corresponding mismatch table for the example search pattern in Figures 4 and 5. It indicates that if a mismatch occurs at a 'B' within the haystack, we are to remain at that index, and compare it with index 1 of the needle. If a mismatch occurs at any other character, we are to remain at that index within the haystack, and compare it with index 0 of the needle.

A	B	C	A	B
0	0	0	0	1

Table 1: KMP Mismatch table.

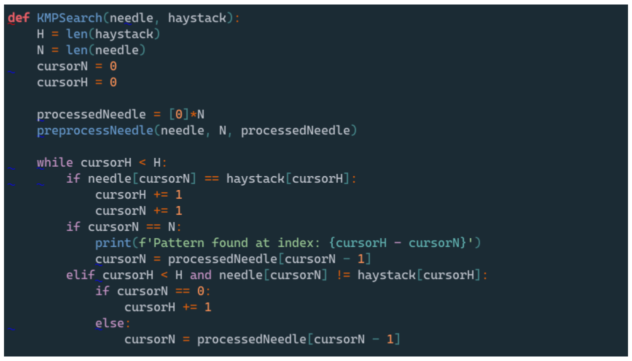
Python Implementation


```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]
```

Figure 6: Preprocess Function for KMP

The function shown in Figure 6 has `processedNeedle`, an array which stores the length of the longest prefix, suffix pairs. We calculate the value such that if the `needle[leftCursor]` and `needle[rightCursor]` matches, we increment `leftCursor` and assigns this value to the `processedNeedle`. If there wasn't a match and the `leftCursor` is pointing to the first character, we reinitialise it. The point of all of this is to find characters which form both the prefix and suffix of the needle as we iterate through the needle.



```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]
```

Figure 7: KMP Search Function

After initialising the variables, we would need to preprocess the needle using the function in Fig 2. Since the length of the haystack and needle are h, n respectively, we would begin comparison of the needle and the haystack and increment the value of the cursor while they keep matching. The *if cursorN == N* indicates that all characters match with each other and the pattern is found. The benefit of KMP comes when there is a mismatch. If the mismatch happens at index a, this means that the first $a-1$ characters of the needle matches with the first $a-1$ characters of the haystack. From the preprocessing, we know that *processedNeedle[a - 1]* is the number of characters that are both the prefix and suffix of that substring of the needle of length a. This means that we do not need to match the $a-1$ characters because we know that these characters will match.

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. Since we need to calculate the value of the prefix suffix pairs for each character in the needle, this would mean that the time complexity is $\mathcal{O}(n)$. Since the preprocessing and the matching happens linearly, we can say that the time complexity of the algorithm is $\mathcal{O}(h+n)$. In the problem that we are given, the best case and worst case time complexity for KMP would remain the same. This is because we want to find all occurences of the needle in the haystack. Hence, we would need to go through all the characters of the string even if a match is found in the first index. This means that the best case and worst case time complexity remains $\mathcal{O}(h+n)$.

The improvement to a linear time complexity in the worst case from $\mathcal{O}(hn)$ to $\mathcal{O}(h+n)$ comes with a tradeoff 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 $\mathcal{O}(n)$.

Analysis

Empirical Analysis

Based on the empirical analysis that we have done above, we can see that for most cases that we tested. Brute force actually take a shorter amount of time compared to KMP even though we have mentioned above that KMP should theoretically be faster since it's time complexity is $\mathcal{O}(h+n)$ in comparison to the worst case time complexity of Brute Force which is $\mathcal{O}(hn)$. However, this worst case scenario would occur only when the comparison for the substring is the same until the last character. For example, a needle of "AAAC" and a haystack filled completely with "A"s. However, this is not the case that we're dealing with in this

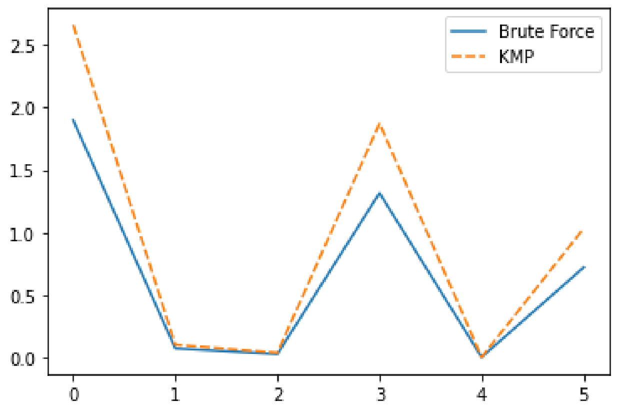
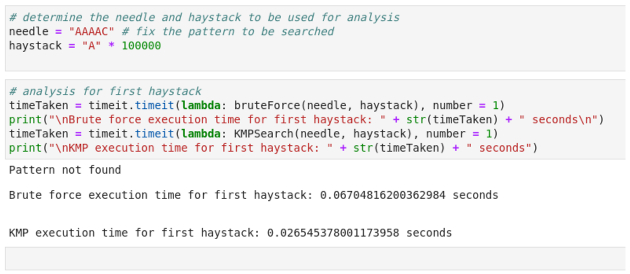


Figure 8: Graph showing differences in timings

problem and neither is this something that will reasonably happen in a real search. On average, the number of comparisons that would have to be done everytime the character shifts is much less than n. This is why it is faster than KMP in the experiment that we have done since the KMP algorithm would still have a time complexity of $\mathcal{O}(n+m)$ even in the best case.



```
# determine the needle and haystack to be used for analysis
needle = "AAAAC" # fix the pattern to be searched
haystack = "A" * 100000
```

```
# analysis for first haystack
timeTaken = timeit.timeit(lambda: bruteForce(needle, haystack), number = 1)
print("\nBrute force execution time for first haystack: " + str(timeTaken) + " seconds\n")
timeTaken = timeit.timeit(lambda: KMPSearch(needle, haystack), number = 1)
print("\nKMP execution time for first haystack: " + str(timeTaken) + " seconds")
```

```
Pattern not found

Brute force execution time for first haystack: 0.06704816200362984 seconds


KMP execution time for first haystack: 0.026545378001173958 seconds
```

Figure 9: Timings

As has been mentioned, in this case, it is certainly faster for KMP than Brute Force to run. However, this is unrealistic since it is extremely improbable to encounter this in any string search scenario for a DNA