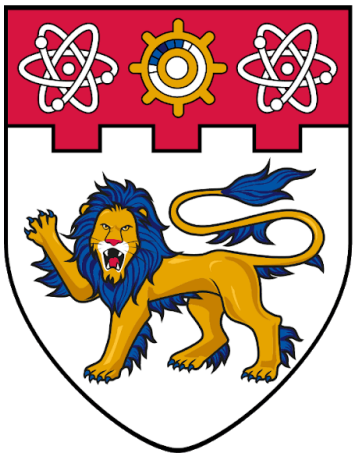

NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

# CZ2001: Algorithms (Lab Project 1)

**Team Members:** Low Yu Benedict,
Leong Kah Wai Alex,
Ong Jing Hong Elliott,
Sunny Pek Yee Chong,
Loe Kit Leong Daniel

**Lab Group:** SSP2

## Objectives

1. Analyse the time complexity of a brute force sequential search on exact occurrences of a query sequence in a genome sequence
2. Design a 'better' algorithm processing the same query sequence as the brute force method
3. Perform algorithm analysis on the proposed algorithm
4. Analyse the time complexity for the proposed algorithm

Programming Language Used: **Python**

## Design of Brute Force Sequential Search Algorithm

How Brute Force Sequential Search works:

1. Traverses the main string from start (left) to end (right)
2. Check every character from the main string to match against the first character of the substring
3. If the first character matches, keep matching while iterating through both the main string and substring till the last character of the substring. If there is a mismatch at any point, continue the traversal of the main string after where the first character matched previously and repeat from step 2
4. If there is no mismatch between the main string and substring throughout the subsequent iterations after the first character matches, substring is found

## Implementation & Analysis of Brute Force Sequential Search Algorithm

Given the following implementation of the brute force sequential search algorithm, we say that:

1. The main string containing all genome sequence data contains **n** elements (problem size)
2. The substring to be searched for in the main string contains **m** elements (2nd problem size)
3. Any primitive operation (e.g. assignment of value) utilises a constant time **c** (Primitive operations performed in function loops are defined by their problem size, thus omitted in our analysis)

[For Brute Force best case, it is when every single character in the string does not match with the first character in the substring.]

```
def brute_force(substring, string):
  substring_index = []                                  #1
  string_length = len(string)                           #2
  substring_length = len(substring)                     #3

  for i in range (string_length):                       #4
    substring_occurrence_complete = True                #5
    if substring[0] == string[i]:                       #6

        for j in range (0, substring_length):           #7
          if i+1 >= string_length:                      #8
            substring_occurrence_complete = False       #9
          elif substring[j] != string[i+j]:             #10
            substring_occurrence_complete = False       #11
            break
        if substring_occurrence_complete:               #12
          substring_index.append(i+1)                   #13

  return substring_index
```

Analysis of time complexity for brute force sequential search (**Best Case**):

1. **#1 To #3** assignment of value in constant time, **c**
4. Traversal of main string, **n**
5. Set flag (substring_occurence_complete) to True, **n**
6. Check if first character of substring matches, **n**

Time complexity = 3c + n + n + n
= 3n + ~~3c~~ (Eliminate constant time c)
= **O(n)**

Analysis of time complexity for brute force sequential search (**Worst Case**):

1. **#1 To #3** assignment of value in constant time, **c**
4. Traversal of main string, **n**
5. Set flag (substring_occurence_complete) to True, **n**
6. Check if first character of substring matches, **n**
7. Inner loop traversing the substring, **n * m**
8. Check if we are nearing end of string traversal, to avoid premature confirmation of substring position, **n * m**
9. Setting flag if to False if we are at the second last index of main string traversal, **n * m**
10. Comparison between main string and substring characters, **n * m**
11. Set flag to False if there is a mismatch, **n * m**
12. Comparison to check if the exact substring has been found, **n**
13. Appending index of found substring to list, **n**

Time complexity = 3c + n + n + n + (nm) + (nm) + (nm) + (nm) + (nm) + n + n
= 5n + 5nm + ~~3c~~ (Eliminate constant time c)
= **O(nm)**

## Design of Proposed Algorithm

How **Knuth–Morris–Pratt (KMP)** Algorithm works:

1. (Pre-Processing) Build an array containing information of the longest common prefix and suffix which do not overlap in the substring.
2. Traverses the main string and the substring from start (left) to end (right) while at each traversal step, matching the characters from both main and substring.
3. If there is a mismatch of character at any point before the traversal completes iterating through the complete substring, a secondary step has to be performed on the substring where the largest common prefix and suffix **before** the mismatched character have to be identified using the constructed array.
4. Once the common prefix and suffix have been identified in the substring, the traversal of the main string will not move to the next character but remain at its current index and try to match the mismatched character again but this time with the character after the prefix in the substring. If both characters do not match, the main string's search index will be moved to the next index while traversal of the substring restarts at its first character to begin matching again.
5. If the previously mismatched character in the main string and character after the prefix in substring matches, traversal and matching for both the main string and substring will continue either till the successful matching of the last character of the substring which indicates that the substring has been found or until there is a mismatch once again where the process flow will repeat from step 2.

## Implementation & Analysis of the Proposed Algorithm

Given the following implementation of the Knuth–Morris–Pratt search algorithm, we say again that:

1. The main string containing all genome sequence data contains **n** elements (problem size)
2. The substring to be searched for in the main string contains **m** elements (2nd problem size)
3. Any primitive operation (e.g. assignment of value) utilises a constant time **c** (Primitive operations performed in function loops are defined by their problem size, thus omitted in our analysis)

```
def KMP_Preprocess(substring, substring_len):
  latest_lpps_idx = 0                                    # 1
  lpps = [0]*substring_len                               # 2
  i = 1                                                  # 3

  while i < substring_len:                               # 4
    if substring[i] == substring[latest_lpps_idx]:       # 5
      latest_lpps_idx += 1                               # 6
      lpps[i] = latest_lpps_idx                          # 7
      i += 1                                             # 8

    else:
      if latest_lpps_idx != 0:                           # 9
        latest_lpps_idx = lpps[latest_lpps_idx-1]        # 10
      else:
        lpps[i] = 0                                      # 11
        i += 1                                           # 12
  return lpps
```

Analysis of time complexity of KMP preprocessing:

1. #1 To #3 Assignment of value in constant time. **c**
4. Traversal of substring. **m**
5. Comparison between character of iterating substring and character of substring with LPPS (Largest Prefix Suffix) index . **m**
6. Updating of LPPS index. **m**
7. Assignment of Latest LPPS index to LPPS table. **m**
8. Increment i by +1. **m**
9. Check if Latest LPPS index is zero or not. **m**
10. Assignment of new LPPS value . **m**
11. Assignment of LPPS[i] to zero. **m**
12. Increment i by +1. **m**

Time complexity = 3c + m + m + m + m + m + m + m + m + m

= 9m + ~~3~~c (Eliminate constant time c)

**= O(m)**

**[For KMP best case, it is when an LPS table already exists and the process to generate an LPS tables are no longer required. Code #6 is no longer required]**

```
def KMP_Search(string, substring):
  substring_len = len(substring)                          # 1
  string_len = len(string)                                # 2
  substring_position = []                                 # 3
  j = 0                                                   # 4
  i = 0                                                   # 5
  lpps = KMP_Preprocess(substring, substring_len)         # 6

  while i < string_len:                                   # 7
    if substring[j] == string[i]:                         # 8
      i += 1                                              # 9
      j += 1                                              # 10

    if j == substring_len:                                # 11
      substring_position.append(i-j+1)                    # 12
      j = lpps[j-1]                                       # 13

    elif i < string_len and substring[j] != string[i]:    # 14
      if j != 0:                                          # 15
        j = lpps[j-1]                                     # 16
      else:
        i += 1                                            # 17
  return substring_position, lpps
```

Analysis of time complexity for KMP Search (**Best Case**):

1. **#1 To #5** Assignment of value in constant time, **5c**
6. ~~Generate LPPS Table, **m**~~
7. Traversal of string, **n**
8. Check if string[i] matches substring[j], **n**
9. Increment i by +1, **n**
10. Increment j by +1, **n**
11. Check if j reaches the end of substring, **n**
12. Append starting index of substring, **n**
13. Assignment of new j Value, **n**
14. Check if i has reached the end of string and check if there is a mismatch of character between main string and substring, **2n**
15. Check if j has reached back to the starting character of substring, **n**
16. Assignment of new j value, **n**
17. Increment i by +1, **n**

Time complexity = 5c + n + n + n + n + n + n + n + 2n + n + n + n
= 12n + ~~5c~~ (Eliminate constant time c)
= **O(n)**

Analysis of time complexity for KMP Search (**Worst Case**):

1. **#1 To #5** Assignment of value in constant time, **5c**
6. Generate LPPS Table, **m**
7. Traversal of string, **n**
8. Check if string[i] matches substring[j], **n**
9. Increment i by +1, **n**
10. Increment j by +1, **n**
11. Check if j reaches the end of substring, **n**
12. Append starting index of substring, **n**
13. Assignment of new j Value, **n**
14. Check if i has reached the end of string and check if there is a mismatch of character between main string and substring, **2n**
15. Check if j has reached back to the starting character of substring, **n**
16. Assignment of new j value, **n**
17. Increment i by +1, **n**

Time complexity = 5c + m + n + n + n + n + n + n + n + 2n + n + n + n
= m + 12n + ~~5c~~ (Eliminate constant time c)
= **O(m + n)**

**Illustration of Comparison between Brute Force Algorithm and KMP Pattern Search Algorithm**

A comparison between the Brute Force, KMP Pattern Search Algorithm (inclusive of pre-processing time) and KMP Pattern Search Algorithm (without accounting for pre-processing time) is shown in the graphs below.

Figure 1 feeds the two algorithms with existing substrings of varying length from 3 to 193 characters. As such all algorithms will find positions matching these substrings.

Figure 2 feeds the two algorithms with non-existent substrings of varying lengths from 3 to 193 characters. As such none of the algorithms will be able to find matching substrings.

The KMP Algorithm that does not time the pre-processing time (Green line) is taken into account as it is possible to optimize the KMP Search by first having the longest-prefix-that-is-also-a-proper-suffix array available prior to the search.

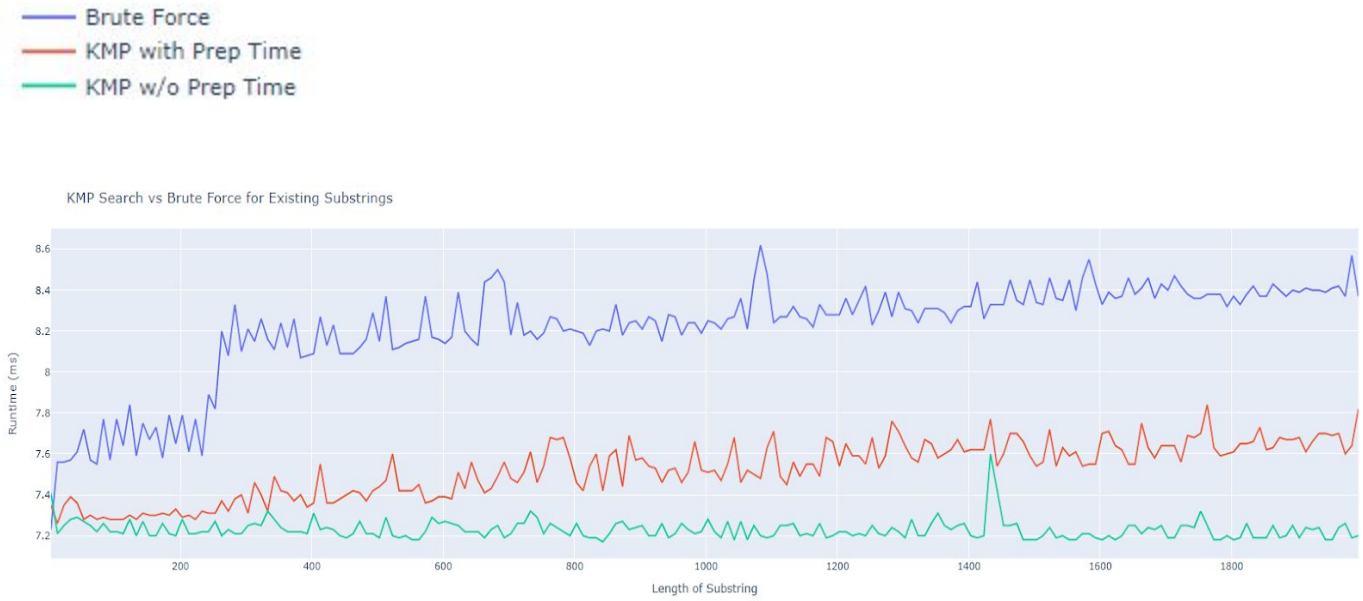


Figure 1: Algorithms given existing lengths of substrings

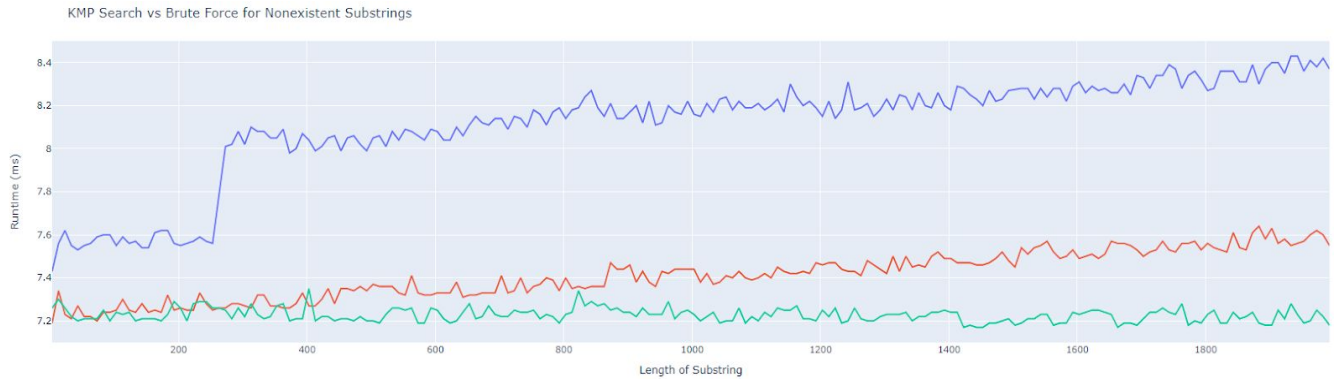


Figure 2: Algorithms given non-existing lengths of substring

**Contributions**

Benedict: Coding, Documentation
Alex: Research, Presentation, Documentation
Elliott: Research, Presentation, Documentation
Sunny: Research, Presentation, Documentation
Daniel: Coding, Documentation

**References:**

KMP Algorithm for Pattern Searching. (2019, May 20). GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/