CZ2001 ALGORITHMS

Project 1: Searching Algorithms

Bairi Sahitya
Loh Xin Yi
Satini Sankeerthana
Shauna Tan
Ong Li Wen

Algorithms analysed:
Brute Force
Knuth-Morris-Pratt (KMP)
Rabin-Karp

General Notes

Our programme can only handle text files with sizes up to around 3.2GB.

Brute Force Algorithm

The brute force algorithm works by searching every possible index in the text for the pattern, where the search window moves down by one index each time. A summarised version of the code used for the current problem is as follows, where m=pattern size and n=text size:

```
for start in range(0, n-m+1): # to move the search window down by one index each time
        count=0
        # searching within each search window
        for i in range(0, m):
                if text[start+i] == pattern[i]:
                        count+=1
                else:
                        break
        if count == m:
                list_of_indices.append(start)
return list_of_indices
```

The algorithm takes the text and pattern strings as inputs (not shown above) and prints out the exact occurrences in the text at which the pattern appears. The brute force algorithm does not pre-process the pattern string in any way and is essentially a 'blind' search in the sense that it is very basic with no theory involved.

Time Complexity Analysis:

For the sake of simplicity, we will only be analysing the number of comparisons done to determine the time complexity of the algorithm.

Best case:

The best case occurs when the first index of every search window is a mismatch. This line of code "if (text[i+j] == pattern[j])" will be executed (n-m+1) number of times, which is the total number of comparisons. Thus, the best-case time complexity is O(n).

Average case:

The average-case time complexity for brute force is very complicated and tedious, so we will not consider it in our analysis. Moreover, the worst-case time complexity is usually more commonly used to compare algorithms as opposed to the average-case time complexity.

Worst case:

The worst case occurs when every index within every search window is a match. This line of code "if (text[i+j] == pattern[j])" will be executed (n-m+1)(m) number of times and "if (count == m)" will be executed (n-m+1)(m) times as well. The total number of comparisons done is 2m(n-m+1). Thus, the worst-case time complexity is $O(mn-m^2) = O(mn)$.

Knuth-Morris-Pratt (KMP) Algorithm

KMP minimizes the amount of backtracking when a mismatch occurs. When a mismatch is detected, instead of searching through every index, lps table is used to determine from which index to search next.

Creation of LPS table:

```
m = len(pat)
lps = [0]*m
```

```
    i = 1
    j = 0 #length of prev longest prefix suffix
    while i<m:
  #match occurs, length of lps will  increase and index i will be incremented by 1 to compare the next char.
        if pat[i] == pat[j]:
            lps[i] = j + 1
            i += 1
            j += 1
        else:
            if j == 0:
                lps[i] = 0
                i += 1
            else:
                j = lps[j-1] #check the index of prev possible prefix
                return lps
```

To search query pattern in text:

```
    n = len(text)
    m = len(pat)
    lps = computeLPS(pat) #creation of lps table of query pattern
    i=0 #index for text
    j=0 #index for pat
    while i<n:
        #if matches, increment 1 for both index to continue comparing
        if pat[j] == text[i]:
            i += 1
            j += 1
        else:
            if j == 0:
                i += 1 #cannot decrement as j=0, increment i
            else:
                j = lps[j-1] #lps table will tell from where to compare next
        if j == m:
            print ("Query pattern is at index " +  str(i-j))
            j = lps[j-1] #to continue finding more patterns
```

A lps table will first be created and will be looked up in the searching of the query pattern. Based on the above, before we search the query pattern, the lps table will first be computed based on the query pattern. It will then search through the text and when a mismatch occurs after a few matches, we will check if there is a longest prefix that is also a suffix by looking up the lps table. And through this, we can determine where to compare next and avoid matching a character we know will match anyway.

Time Complexity Analysis

Creation of LPS table - Preprocessing Time Complexity O(m): When there is a mismatch, j may backtrack a certain amount of times. j cannot roll back more than the times i travelled forward, which is the no. of times each character has been checked with no mismatch. The no. of times can at most be m times where m is length of query pattern. m+m=2m. Time complexity will be O(m).

Best Case: No matches found O(n) + Preprocessing O(m)

The best case is when there are no matches found. i and j will keep increasing till the end of the string which means the execution takes place n times. Therefore, the searching time complexity will be O(n).

Average Case: O(n) + Preprocessing O(m)
Total runtime = Runtime for successful matches: <=N + Runtime for unsuccessful matches: <=N = 2N

Worst Case: search through all the characters in the text and pattern at least once O(n) + Preprocessing O(m)
This is similar to creation of lps table, where the amount of times j backtracks, it must first go n times forward to backtrack n times. Hence, it will be n + n = 2n. Therefore searching time complexity is O(n).

Rabin-Karp Algorithm

To minimise confusion we have used the term 'needle' to refer to the Pattern to be searched and 'haystack' to refer to the sequence/text to be searched in. Also 'n' is the length of haystack and 'm' is the length of the needle.The hash number of the needle is sequentially compared against the hash number of each of the m-length patterns in the haystack. If the number matches, each of the bases is checked to confirm the match. Thus the number of times the "inner for loop" is only executed when there is a match in the hash numbers,and its occurrences are reduced, thus reducing the time complexity significantly.

Pseudo Code

```
//Implementing a switcher in Python for each of the 5 values (A,T,C,G and U (for RNA)) in the for loop
def convert(sequence):
        for i in range(len(sequence)):
                Switcher = {'G' : 1; 'A' : 2; 'C' : 3; 'T' : 0; 'U' : 0; }
        hash_num = (hash_num*5 + switcher.get(sequence[i], 4))%101                    O(1)
Return hash_num;
def rolling_hash(text_hash):
        return ((text_hash-switcher.get(haystack[i-1],4)*multiplier) * 5 +
switcher.get(haystack[i+size_m-1],4))%q

//Rabin Karp Algorithm
def rabinkarp(haystack,needle):
        Needle_hash = convert(needle_hash);                                    O(m)
        Results = [ ]
        For i in range(m-n+1):
                text _hash = rolling_hash(text_hash)                           O(1)
                Find hash value of m-lengthed pattern in haystack
                If hash value of pattern in haystack == hash value of needle   O(1)
                    Valid = True
                        For j in range(n):
                                if bases of needle and pattern in haystack are different:   O(1)
                                valid = False
                                Break
                if valid:
                  results.append(i)
 return results
```

Calculating the Hash Value:

General Formula: Sum of (ASCII Value * (large_prime^index_position))
Modified: Sum of (DNA_Base_Value * (5^index_position)) % prime number

As seen above, each of the 5 letters are first assigned a numerical value from 0 to 3. If the input letter does not fall within the range, it is assigned a value of 4. Thus, we have taken 5 as the base value (as there are 5 distinct characters), which is raised to the power of the index position of each of the DNA Base in the sequence string. To this, we multiply the value assigned to the specific DNA base in the switcher block. Eg: CCA = (2 * (5^0)) + (3 * (5^1)) + (3 * (5^2)) = 92
We then multiplied the hash value by mod 101 at each of the multiplications so the number will always remain within the range of 100. Hence CCA = ((((3 %101*5 + 3)%101*5 + 2)%101 = 92. The addition of the modulo is to account for large hash numbers which will require larger computation time. By using a rolling hash, the new hash number is created from the previous hash number, thus removing the need for recomputation from scratch at each letter in the text.

Time complexity Analysis:

Preprocessing time:

The preprocessing includes converting the needle and the first comparison window to their respective hash value. The conversion is done in constant time (O(1)) for each of the letters. Thus the total preprocessing time is O(m). (see convert function).

For rabinkarp(haystack,needle):

Best and Average Case:.

In the rolling hash steps, computation of text_hash and comparison of hash numbers is in O(1). A total of (n-m)k computations and comparisons are done and the time complexity is O(n), ignoring constants. Every time there is a match in the hash numbers, the inner loop is executed to confirm the pattern match which is in O(m). Thus the time complexity average case is O(n+m). Best case occurs when there is no match in hash numbers and the inner loop is never executed giving a complexity of O(n).

Worst case scenario: Searching for AAA in AAAAAAAAAAAAA.

The complexity of the worst case will still be O(mn). This will happen if there is a match at every pattern or when there is false positive in each step of the hash number matching, causing the inner loop to be executed each time.

BIBLIOGRAPHY:

https://www-igm.univ-mlv.fr/~lecroq/string/node3.html#:~:text=The%20brute%20force%20algorithm%20requires,in%20an%20for%20instance)
https://brilliant.org/wiki/rabin-karp-algorithm/#:~:text=The%20best%2D%20and%20average%2Dcase,collision%20and%20therefore%20must%20check
https://arxiv.org/ftp/arxiv/papers/1401/1401.7416.pdf
https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
https://www.programiz.com/dsa/rabin-karp-algorithm
https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
https://www.educative.io/edpresso/what-is-the-knuth-morris-pratt-algorithm

Work Distribution

Shauna Tan - Brute Force algorithm implementation, report, slides, GUI and import .fna file implementation
Bairi Sahitya and Satini Sankeerthana - Rabin Karp algorithm implementation, report, slides
Loh Xin Yi and Ong Li Wen - KMP algorithm implementation, report, slides