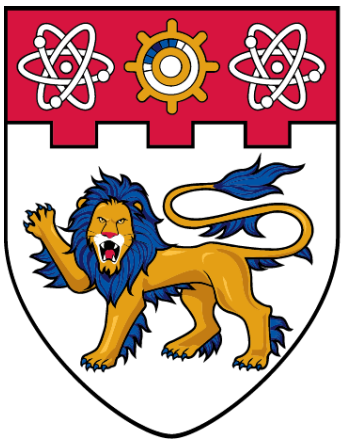

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Project 1 Report: Searching Algorithm

Team Members:

Name	Matriculation Number
Acharya Atul	U1923502C
Anil Ankitha	U1923151C
Arora Srishti	U1922129L
Kothari Khush Milan	U1922279J
Pal Aratrika	U1922069F

Brute Force Algorithm

This is the naive string searching algorithm which uses 2 nested loops. It loops through the entire sequence and in case of a mismatch, it backtracks and shifts the outer-loop variable by 1.

Pseudo-code

```
Counter = 0
For i =0 to (length of sequence) - (length
of query):
    Pos = i
    For j=0 to (length of query - 1)
        If str[i+j]!= query[j]
            Pos = -1
            Break
       If Pos != -1
            Print Pos
            Counter += 1
    If Counter is 0
       Print "Empty"
```

Analysis

Space Complexity: The brute force algorithm requires no auxiliary storage hence the space complexity is none.

Time Complexity: Let the length of the sequence be n and the length of the query be m.

Best Case Scenario: The query is not present in the sequence. Therefore, the first character of the query never matches any character of the sequence. The inner loop is never executed.

Ex. Sequence = "ABCDABCDABCDABCD" and Query = "ZZZZ". Let the compiler perform $c1$ operations in the outer for loop and $c2$ operations in the nested inner for loop. (Where $c1$ and $c2$ are constants)

Total number of operations = $(n - m + 1) * c1$

Time complexity is $O(n - m) \approx O(n)$. (since $m > n$)

Worst Case Scenario: When all characters of genome sequence match the query. E.g. Genome Sequence = "AAAAAAAAAAA" and

Query = "AA". Or when all characters of the genome sequence and query are the same, except the last. Ex: Genome Sequence = "AAAAAAAAAAAB" and Query = "AAAB".

Number of iterations of outer loop = $n - m + 1$

Number of iterations of inner loop = m

Total number of operations = $(n - m + 1) * (m * c2 + c1)$

Time complexity = $O(nm) \approx (m^2 << mn$, therefore neglected)

Average Case Scenario:

The number of times the outer loop runs = $n - m + 1$

The inner loop may run x times where x ∈ [1, m]. Each value of x has an equal probability of 1/m.

Number of iterations of inner loop = $\frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m}(\frac{m(m+1)}{2}) = \frac{m+1}{2}$

Total number of iterations = $\frac{m+1}{2}(n - m + 1)$

$= \frac{(m*n-m2+ m + n-m+1)}{2}$

$= \frac{(m*n - m2+ n + 1)}{2}$

Time complexity is therefore ≈ $O(mn)$

Knuth Morris Pratt Algorithm

One drawback of the brute force algorithm is that it cannot recognize patterns in the genome sequence and it blindly checks all possibilities exhaustively. To prevent that, this algorithm is

used. It includes a pre-processing step where an array of the same length as the query is created. This is called a pi table which stores the length of the longest proper prefix which is also a suffix. Using this array, during searching, when a mismatch occurs, the algorithm knows which position of the genome to jump to.

Pseudo-code

```
algorithm Compute-π-values(P[1,..,m])
input: pattern P of length m
preconditions: 1 ≤ m
output: table π[,....m]
π(1) ← 0;
for(i ← 1 to m-1)
/* π[1,.....,i] is already
calculated;  calculator π[i + 1]
{
algorithm KMP(P[1,...,m], T[1,...n])
input: pattern P of length m and text T of
length n
preconditions: 1 ≤ m ≤ n
output: list of all numbers s, such that P
occurs with shift s in T
q ← 0;
i ← 0;
while (i<n) /* P[1,....q] == T[i - q +
1,....,i]
{
        if( P[q+1] == T[i+1] )
        {
                q ← q + 1;
                i ← i + 1;
                if ( q == m )
                {
                        output i - q;
                        q ← π(q); /*slide
pattern to the right
                }
        }
        else /*a mismatch occurred
        {
                if( q == 0) {i ← i + 1}
                else { q ← π(q) }
        }
}                                       [1]
```

Analysis

The analysis of the KMP algorithm can be divided into two parts- pre-processing and searching function. Both the parts have their own complexities and need to be analysed separately.

The pre-processing function consists of a single while loop with counter-variable i and test condition i < length of the query. Let us assume that the length of the query is m. Hence, the loop will iterate $m-1$ times. The body of the loop consists of an if-else segment. Assuming the worst case, when there is a mismatch and j != 0, i will not be incremented. As a result, the loop will iterate from j = 1 to m. There are other $m-1$ iterations. Thus, total number of iterations = $m-1+m-1=2m-2$. Hence, this function has a linear time and space complexity of $O(m)$.

The search function has the majority of its code inside a single while loop. The initialization of i, j and pos have fixed costs. Let's assume the length of the genome sequence is n and j is the index for query. The while loop has i = 0 as its counter-variable and its test condition is $i < n$. Hence, the loop will iterate n times. However, there is a difference in the if conditions of this algorithm and that of the brute-force algorithm. In this, as long as the characters match, we keep on incrementing i as well as j. We then check if a match has been found and if this is true, we use the longest prefix suffix array to set the next value of j so that we do not need to start from the beginning. If there is a mismatch, the same array helps us to reduce the total number of iterations. Ex: if the search was at genome [i] and genome [$i+j$] != query [j], then the next search must begin at genome[$i+(j-$pre-processed_table[j]).

Therefore, the pre-processed array allows us to avoid starting search from the beginning and reducing the total number of iterations in the process. Hence, the loop can execute a maximum of $2n$ times in the worst case scenario. In other words, whenever there is a mismatch, the genome index remains unchanged and we try to match genome [i] with query [preprocessed_table [j]]. The query index is decreased by a certain amount calculated in

the preprocessed_query() function. Thus, the maximum run-time is $2n$ times and the time complexity of this function is $O(n)$, which is linear.

Combining the two, we find that the overall time complexity of KMP Search is $O(m+n)$ in all scenarios. This proves that the KMP algorithm is much more efficient as compared to the brute-force algorithm. The space complexity of this algorithm is $O(m)$ as we need a preprocessing array of size of the query.

Rabin Karp Algorithm

In Rabin Karp Algorithm, hashing is used for searching. The query is first converted into one single hash value. Then we traverse through the genome and hash each genome substring of the same length as query. If the hash value of the query and that genome substring match, only then we perform character by character comparisons like in Brute force, to see if the query is present. Example of the hash function: AGCT→ [int(A)* 5^3 + int(G)* 5^2 + int(C)* 5^1 + int(T)* 5^0] mod q , q is a prime number, and 5 is the number of unique characters possible.

Pseudocode

```
Compute h_p ( for pattern p)
Compute h_t (for the first substring of t
with m length)
For i = 1 to n - m
     If h_p = h_t
          Match t[i….i + m] with p, if
matched return 1
     Else
          h_t = (d (h_t - t[i+1].d^(m-1)) +
t[m+i+1]) mod q
End                                    [2]
```

Analysis

Space Complexity: In this algorithm, the hash code of the query which is shown in code as hash_q, and the hash code of substrings in the genome sequence, shown in code as hash_g need to be stored. Since these are 2 variables the space complexity is constant or $O(1)$.

Time Complexity: Let the length of the query sequence be m and the length of the genome sequence be n.

***Pre-processing time*:** Calculating the value of h (variable in code) which stores base^{m-1}modq where q is a prime number, needs a for loop running $m-1$ times and hence, has a time complexity of $O(m)$. Calculating the hash value of the query and the first window of m characters in the genome sequence also uses a for loop running m times and hence has a time complexity of $O(m)$. So, the pre-processing time complexity is $O(m)$.

***Searching time*:** The rolling hash function is chosen such that after shifting the window by one character in the genome sequence, it can be recomputed easily with a fixed number of operations taking $O(1)$ time. There is an outer for loop that runs $n-m+1$ times to scan through the genome sequence. An inner for loop runs only if the hash value of the query matches the hash value of a substring in the genome. Hence in the best case: No substring of length m in the genome has the same hash value as the query. Then, the inner for loop is never executed and the outer for loop runs $n-m+1$ times so it is $O(n-m+1) \approx O(n)$. In the average case, the algorithm runs the outer for loop $n-m+1$ times so essentially it runs in $O(n-m+1) \approx O(n)$. In the worst case, all the characters of the genome and query are the same. At every step, the hash value of the genome substring matches the hash value of the query and the inner loop runs m times to compare corresponding characters in the

genome substring and query to find a match. Hence the loops run $(n - m + 1) * m$ times and it has a complexity of $O(mn)$. [m^2 is negligible compared to mn]. On adding the pre-processing and searching time:

Best case: $O(m) + O(n) = O(m + n)$

Average case: $O(m) + O(n) = O(m + n)$

Worst case: $O(m) + O(mn) = O(mn)$

Experimental Analysis

A graph was plotted with the x axis showing the length of the genome sequence n and the y axis showing the number of character comparisons performed in all 3 algorithms.

The number of character comparisons done by the KMP and Rabin Karp (including hash comparisons) Algorithms were significantly less compared to the Brute Force Algorithm.

Fig: Graph of n vs number of comparisons

Conclusion

KMP and Rabin Karp algorithms outperform Brute Force Search in almost every scenario. The KMP algorithm can outperform Brute Force because it remembers substring pattern information through a pi table and hence prevents backtracking. The Rabin Karp algorithm can outperform Brute Force because it changes the string comparisons, which are essentially multiple character comparisons in Brute Force to a single value hash comparison.

Contribution Statement

Atul Acharya	GUI, Experimental Analysis, KMP code, KMP Analysis
Anil Ankitha	Presentation and Report Writing, Brute Force Analysis
Arora Srishti	Brute Force Analysis, Rabin Karp Analysis, Experimental Analysis
Kothari Khush Milan	KMP Analysis, Brute Force Code, Experimental Analysis
Pal Aratrika	Brute Force Analysis, KMP Code, Rabin Karp Code, Rabin Karp Analysis

References

[1] Algorithms in Bioinformatics, Spring, 2016. Retrieved from: https://www.cs.ubc.ca/~hoos/cpsc445/Handouts/kmp.pdf

[2] A Kolokolova, Applied Algorithms, Jan 24, 2012. Retrieved from: http://www.cs.mun.ca/~kol/courses/6783-w12/scribe-rabin-karp.pdf

Notes

External libraries used: tkinter, seaborn, time, pickle, pandas

Constraints: The maximum file size that we have tested our program on is 600MB.