In this report, we will be utilising the following three algorithms to help us with string searching in Genome Sequences.

1. Brute Force Algorithm
2. Knuth–Morris–Pratt(KMP) Algorithm
3. Self–Proposed Algorithm

We will provide explanations on the above algorithms, explaining how it works. Then, we will be implementing the said algorithms into our datasets, downloaded from the NCBI website, and analyse its time complexity, as well as providing the correct and complete design and analysis of the algorithms.

In our notation, genome sequence is string S with length N, query sequence is string P with length M.

1 Brute Force Algorithm

The **Brute Force Algorithm** goes through the array of characters from the genome sequence and compares each character from the genome sequence with each character from the query sequence. If the characters match, the algorithm will compare the next character of the genome sequence with the next character of the query sequence. If there is a mismatch, it will compare the next starting character of the genome sequence with the first character of the query sequence. For this assignment, the algorithm will only check $N - M + 1$ substring of the genome sequence.

1.1 Analysis

Worst Case
All the characters in both S and P are same character. The P fully matched on every starting position of S. Since we need to check for all occurrence, the complexity is $\mathcal{O}(N(N - M + 1))$.
Best Case
The characters in S are totally different from P. P mismatch on every first position. The complexity is $\mathcal{O}(N-M+1)$.
Average Case
Suppose we have a random string P and substring of S with the same length as P. Assume there is only 4 types of character in both S and P. Now we calculated the expected number of character matching happen. The probability of i matching happen is $\left(\frac{1}{4}\right)^{i-1}\left(\frac{3}{4}\right)$(Successful matching for $i-1$ characters and mismatch for 1 character. Hence, the expected number of matching is,

$$E = \sum_{i=1}^{M} i\left(\frac{1}{4}\right)^{i-1}\left(\frac{3}{4}\right) + \left(\frac{1}{4}\right)^{M}$$

$$\frac{4}{3}E = 1 + \frac{2}{4} + \frac{3}{4^2} + ... + \frac{M}{4^{(M-1)}} + \frac{4}{3}\left(\frac{1}{4}\right)^{M} \tag{1}$$

$$\frac{1}{3}E = \frac{1}{4} + \frac{2}{4^2} + ... + \frac{M-1}{4^{(M-1)}} + \frac{M}{4^M} + \frac{1}{3}\left(\frac{1}{4}\right)^{M} \tag{2}$$

Subtracting (2) from (1) yields

$$E = 1 + \frac{1}{4} + \frac{1}{4^2} + ... + \frac{1}{4^{(M-1)}} = \frac{4}{3}\left(1 - \left(\frac{1}{4}\right)^{M}\right)$$

When M is positive, the expected number of comparison in each substring in S is smaller than $\frac{4}{3}$, which is $\mathcal{O}(1)$. Hence, the average complexity is equals to the number of substring with length M, which is $\mathcal{O}(N - M + 1)$.

2 KMP Algorithm

The Brute Force algorithm is inefficient as it only takes one character movement in the genome sequence each time when there is a mismatch. The KMP matching algorithm is particularly effective for processing genomic sequences. When a mismatch occurs, it makes use of a preprocessed table known as the Longest proper Prefix which is also Suffix(LPS) table. It will decide the number of characters to be skipped for comparison. The LPS table contains LPS values for each respective index of the query sequence. During a mismatch, the algorithm will check the LPS value of the previous character of the mismatched character in the query sequence. For instance, a value of '0' will resume comparing the first character of the query sequence with the next character of the mismatched character in the genomic sequences. If the value is not '0' the character comparison will start at an index value equal to the LPS value of the previous character to the mismatched character in genomic pattern with the mismatched character in the genome sequence.

```
f[0] = 0; f[1] = 0; //lps array also known as failure function
for(i = 1; i < m; i++){ //pre-compute lps array
    j = f[i];
    while(j && P[i] != P[j]) j = f[j];
    f[i+1] = (P[i] == P[j]) ? (j+1) : 0;
}
for(i = 0, j = 0; i < n; i++){
    while(j && S[i]!=P[j]) j = f[j];
    if(S[i] == P[j]) j++;
    if(j == m){
        cout << "Found at index " << i-m+1 << endl;
        j = f[j];
        found++;
    }
}
```

Listing 1: KMP Algorithm

2.1 Analysis

In the searching state, denote the index of S and P we are currently matching is i and j respectively. The value of i will only increase, and it can at most increase $\mathcal{O}(N)$ times. The value of j will decrease if the mismatch occur. However, j increase only when i is increase. Hence the total number of decrement of j is only bounded by $\mathcal{O}(N)$. For the preprocessing state(calculate *lps* array), the analysis is similar to this, just change the i and j to the index currently checking and the index of longest prefix respectively. Hence, the worst complexity is $\mathcal{O}(\mathcal{N}+\mathcal{M})$. For the best case, we still need to pre-process through the P and searching through the S. So, the complexity is still $\mathcal{O}(N+M)$. So, the worst, best and average complexity is $\mathcal{O}(N+M)$ with a extra space complexity $\mathcal{O}(M)$ to storing the preprocessed value.

3 Self-proposed Algorithm

Our self-proposed algorithm is modified using the baseline of Brute Force Algorithm.

Motivation 1. Brute Force Algorithm results in multiple redundant comparisons as the algorithm searches through every character in the genome sequence in order to match it with the query sequence. It resulting in a terrible worst case time complexity.

Heuristic 1. The self-proposed algorithm will search through the genome sequence, similar to the brute force algorithm, then once the mismatch is found, it will be matched first. After which, the algorithm will once again, continue its search from the front.

Motivation 2. When there are repeating patterns found in the prefix, searching through the front of the sequence will result in significant time wastage.

Heuristic 2. Firstly, the algorithm needs to pre-process the query sequence to know exactly how many characters the algorithm can skip. When a mismatch occurs, we will shift the query sequence to the right and try to match it again. We can notice that characters in the genome sequence before the mismatch character is actually matched to the prefix of the query sequence. Next, our task is to find the maximum character in the prefix we can skip. In preprocessing, for every index in the query sequence, we set it as the starting index and check how long the substring we can match with the prefix.

Searching

Let $|\Sigma|$ be the number of different characters in the query sequence P, in our case is 4(ATCG).

Using $\mathcal{O}(|\Sigma|M)$ time to precompute array in **heuristic 1**(denote by $jump[\,]$), and $\mathcal{O}(M^2)$ time for the longest skip matching(longest prefix) array(denoted by $lp[\,]$). We need to check the matching for all starting index in the genome sequence from $i = 0$ to $i = N - M$.

For the checking, firstly use $lp[\,]$ array to find out the maximum length of the checked character in genome sequence S with the prefix of the query sequence P. To achieve this, we need to store the index of the rightmost substring of the checked genome sequence, noted as $[l : r]$. If the current checking index i is in this region, the number of characters we can skip is $\min(lp[i-l], r-i)$. After that, we can keep matching for the characters left. If the right endpoint of the matching segment r' is larger than r, we update the region to $[i, r']$. If mismatch occurs, we use the $jump[\,]$ array to match the mismatched character in the genome sequence S.

```
//Preprocessing
L = 0, R = 0, i = 0;
while(i < N - M + 1){
    j = 0;
    if(i <= R) j = min(R - i, lp[i - L]);
    while(j < M && S[i+j] == P[j]) j++;
    if(i + j - 1 > R) L = i, R = i + j - 1;

    if(j == M){
        cout << "Found at index " << i << endl;
        found++, i++;
    }
    else i += (j - pre[(int)(S[i + j])][j]);//fail matching at P[j]
}
```

Listing 2: Self-proposed Algorithm

3.1 Analysis

Preprocessing: $\mathcal{O}(M^2 + |\Sigma|M)$ Searching: From the code we can see that the outer while loop will run at most $\mathcal{O}(N - M + 1)$ times. Inside the outer loop, other than the inner loop will run in only $\mathcal{O}(1)$. We just have to consider the inner loop. We claim that this while loop will at most run $\mathcal{O}(N)$ times **in total**.

Proof. When entering the inner while loop, the first character checked is $P[j]$ and $S[i+j]$.
If $i+j < r$: The checking will happen only one time. If $P[j]$ is equal to $S[i+j]$, that means the $lp[\,]$ we calculated is wrong. Hence, $P[j]$ is not equal to $S[i+j]$, which means we will need to check only one time.
If $i+j == r$: There might need several checks in this case since the character in S after index r hasn't been checked. But for every checking, the new value of r will increment. This increment will only happen $\mathcal{O}(N)$ times.
if $i+j > r$: This is impossible since we update j as $min(lp[i-l], r-i)$ *Q.E.D.*

Hence, the worst complexity on searching is $\mathcal{O}(N)$ and the best case is $\mathcal{O}(N - M + 1)$(mismatch for every character). But the preprocessing time is remain the same. So, the worst, best and average complexity is $\mathcal{O}(N + M^2 + |\Sigma|M)$ with a extra space complexity $\mathcal{O}(M^2 + |\Sigma|M)$ to storing the preprocessed value.

4 Empirical Time Analysis

We use multiples datasets with different sizes that downloaded from the NCBI website. This act as the average cases to test the complexity of the algorithms. The result is shown below.

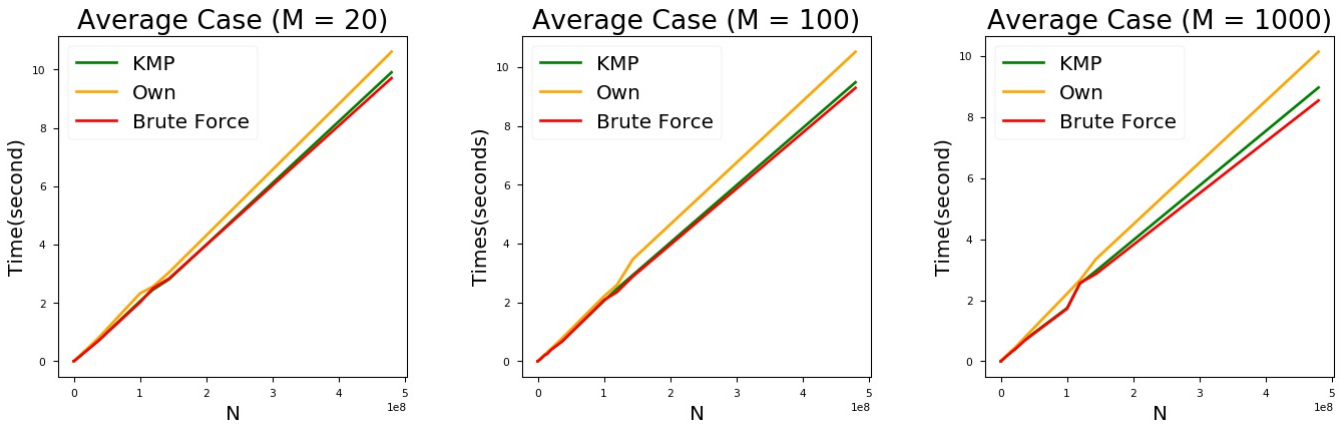
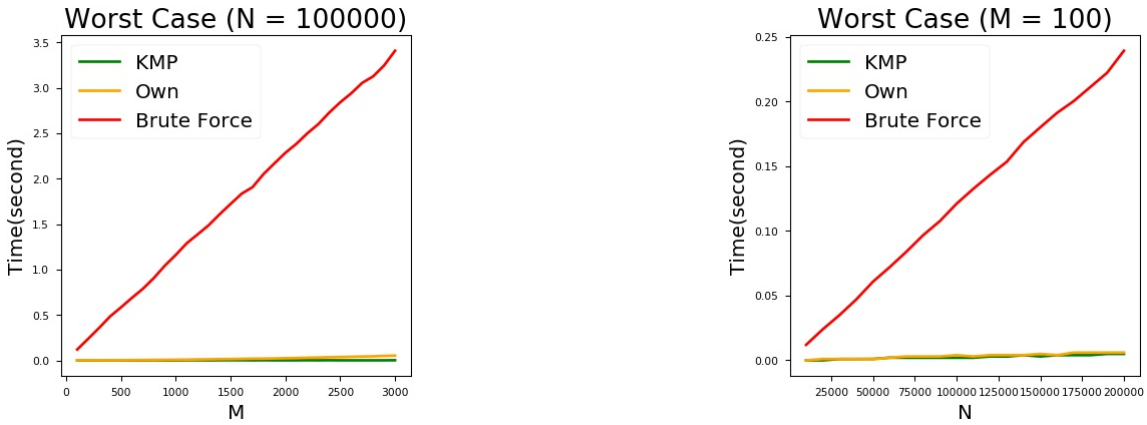


Figure 1: Time Analysis for Average Cases when N Changed

As shown in the graph, all three algorithm complexity is linear proportional to N. Since the operation number is slightly more in self-proposed algorithm, the time consumed is slightly more. When the M increase, the gap between self-proposed algorithm and other algorithms increase as the preprocessing time increase in square. Since in the average cases, the expected number of brute force algorithm is less($< \frac{4}{3}$), hence it performs well.

Next, we generate the string to test for worst case, which both string contains only the same character(all character **A**) with different length. The result is shown below:



(a) M changed

(b) N changed

Figure 2: Time Analysis for Worst Cases

Same as our analysis, the worst complexity of the brute force will be significantly larger than the other two.

5 Conclusion

The brute force algorithm perform quite well in average cases. For genome sequence, the occurrence for characters is quite random(from non-bio major perspective). Also, brute force algorithm does not need any extra spaces to storing the preprocessing value. Hence the brute force is the best choice in this case.

6 References

https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching
https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching
Fast Pattern Matching In Strings(Knuth, Morris, Pratt)