CZ2001: Lab Project 1

A) INPUT DATA

Source: NCBI (National Center for Biotechnology Information) website
Input Data Format - Fasta (.fna or .txt)
Maximum Input File Size : 1 GB, with the below device specifications.
Device Specifications (RAM : 8 GB, Processor : Intel Core I5, CPU : 64-Bit Architecture)

We managed to successfully run the algorithm on the Human Genome Sequence (1 GB). However, running the algorithm on the Homo Sapien Genome Sequence (1.7 GB) resulted in memory errors. Hence, we can definitively state that the maximum file size for the algorithms on this device is 1 GB.

We would also like to state that throughout the report, we will take **n as the length of the genome sequence and m as the length of the query sequence.**

B) ALGORITHMS

1. Brute-Force Search

For the Brute Force algorithm, the first character of the query sequence is checked for a match with the first character of the genome sequence. If they match, subsequent checking is done for the rest of the characters in the query sequence. If the entire string of query sequence matches part of the genome sequence, we have a match and the index of the first character will be stored. Otherwise, the query sequence is shifted over to the next character so that the first character of the query sequence is now checked against the second character of the genome sequence. This continues until the last character for comparison is reached.

TIME COMPLEXITY:

Best Case: We want to minimize the number of comparisons per iteration. To achieve this, the first character in the query sequence must not match any character in the genome sequence. As a result, we will have only one comparison before shifting the query sequence to the next position. Therefore, sum of comparisons = $((n-m+1) \times 1) = n-m+1 < n$, = $\mathbf{O(n)}$

Worst Case: We want to maximize the number of comparisons per iteration. To achieve this, all characters in the query sequence must match up to the last character in the genome sequence. The last character may or may not be a match. As a result, we will then have a maximum of m comparisons before shifting the query sequence to the next position for a total of (n-m+1) positions. Therefore, sum of comparisons = $(n-m+1) * m = nm - m^2 + m < nm+m$ = $\mathbf{O(nm)}$

2. BOYER-MOORE HORSPOOL SUNDAY (BMHS)

The Boyer-Moore Horspool algorithm works by comparing characters of the substring to the main string from right to left. As soon as a mismatch is found, it looks for the last occurrence of the mismatched character (bad character) from the main string in the substring, and identifies the number of characters to shift using a lookup table, such that the mismatch in the main string is aligned with the rightmost occurrence of its corresponding match in the substring.

The Sunday variation (BMHS) goes a step ahead in optimising the algorithm as it uses the character directly right of the query sequence in the main genome sequence, to check if in any case this symbol is involved in the next possible match of the pattern. Between the Sunday Heuristic and the Bad Character Heuristic, the one that returns a larger shift value is used to determine the shift length in the current iteration.

TIME COMPLEXITY:

Preprocessing:
The number of characters to process is equal to the length of the substring = m.
Number of primitive instructions per iteration of preprocessing = C, where C is a constant.
Total number of characters in preprocessing = Cm. So, the time complexity = **O(m)**

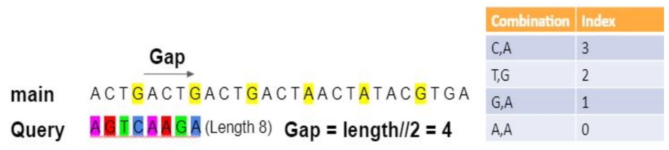
Best Case:

- For the number of mainstring iterations to be minimized, the substring shift per iteration should be maximized. Hence, shift = (m + 1) characters per iteration.
- So the number of iterations over the mainstring would be (n - m) / (m + 1)
- The number of iterations over the substring should also be minimized, hence, the first character during comparison should always be a mismatch.
- Number of comparisons for substring in each iteration in the best case = 1 comparison for bad character heuristic + 1 comparison for Sunday heuristic = 2.
- Total number of iterations = [(n - m) / (m + 1)] x 2 ∈ **O(n/m).**

Worst Case:

- Substring shifts by 1 character at a time for (n - m + 1) iterations over the mainstring.
- For the number of substring comparisons to be maximized, the first (m - 1) characters of the substring should match. The last character should be a mismatch so that the Sunday heuristic is triggered and one extra comparison occurs. So the number of comparisons per iteration would be (m + 1).
- Time complexity = (n - m + 1)(m + 1) ∈ **O(nm).**

3. ORIGINAL ALGORITHM



Combination	Index
C,A	3
T,G	2
G,A	1
A,A	0

Fig 1. Preprocessing

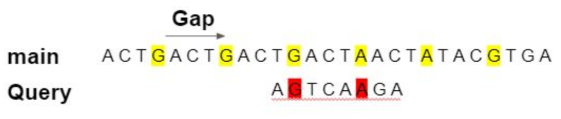


Fig 2. Comparison on match

The original algorithm makes use of fixed checkpoints (denoted in yellow in Fig 1) which are evenly distributed by a constant gap on the genome sequence. The purpose of these checkpoints is to allow us to align our substring properly for comparison to the main string if both the checkpoints exist in the substring configuration.

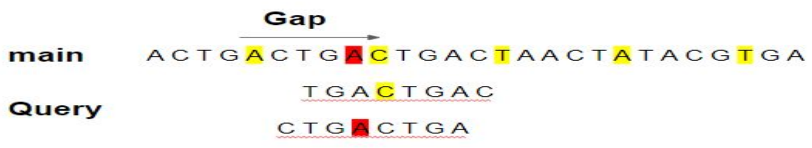
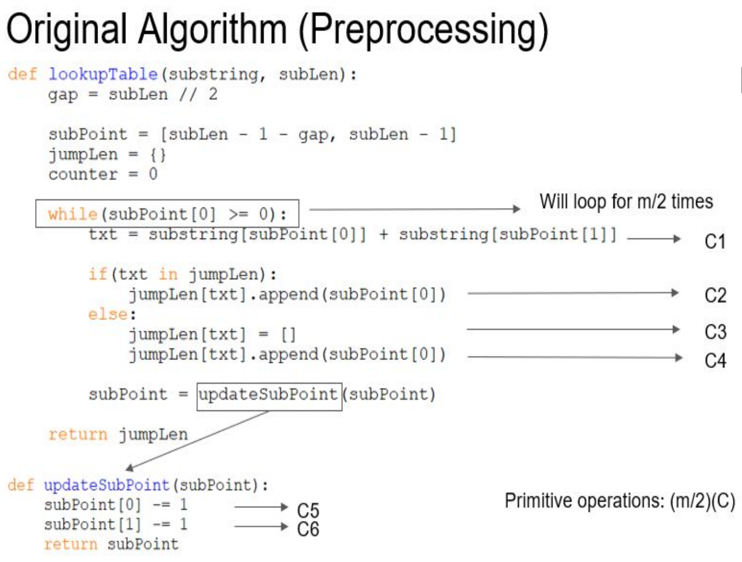


Fig 3. Unmatched characters

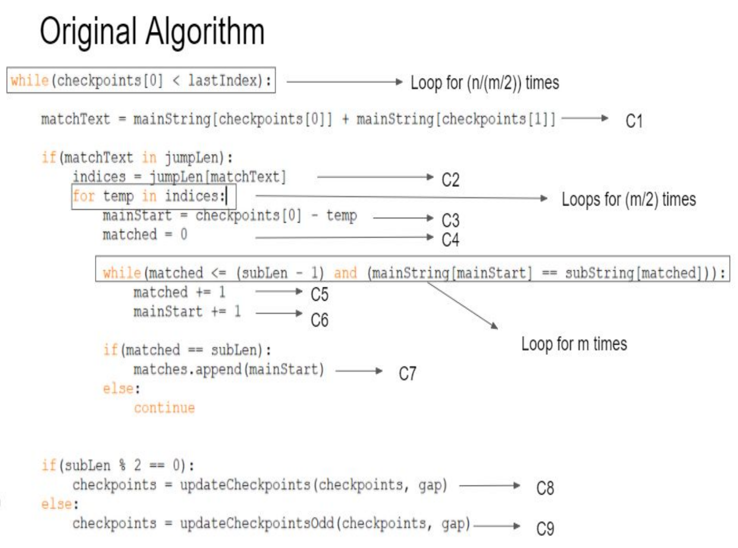
The maximum gap between two checkpoints is obtained by length//2 and this is the optimum gap that we could afford to shift the checkpoints by without compromising on accuracy. If the gap was any wider there might be hidden matches in the genome sequence in between the 2 checkpoints(shown in Fig 3). We will make use of a dictionary to store all character pairs spaced out by the predetermined gap in the query sequence(denoted by the highlighted pairs in Fig 1). We will be matching our character pairs in the dictionary to the checkpoints in the genome sequence. If the character pairs do not match the checkpoints, we will shift our query sequence to the next set of checkpoints. If there is a match as shown in fig 2, we will align our query sequence with the character pairs that match our checkpoint (denoted in red in Fig 2) before carrying out the comparison to see if the whole sequence matches.

TIME COMPLEXITY:



Original Algorithm (Preprocessing)

```
def lookupTable(substring, subLen):
    gap = subLen // 2

    subPoint = [subLen - 1 - gap, subLen - 1]
    jumpLen = {}
    counter = 0

    while(subPoint[0] >= 0):
        txt = substring[subPoint[0]] + substring[subPoint[1]]

        if(txt in jumpLen):
            jumpLen[txt].append(subPoint[0])
        else:
            jumpLen[txt] = []
            jumpLen[txt].append(subPoint[0])

        subPoint = updateSubPoint(subPoint)

    return jumpLen

def updateSubPoint(subPoint):
    subPoint[0] -= 1
    subPoint[1] -= 1
    return subPoint
```



Original Algorithm

```
while(checkpoints[0] < lastIndex):

    matchText = mainString[checkpoints[0]] + mainString[checkpoints[1]]

    if(matchText in jumpLen):
        indices = jumpLen[matchText]
        for temp in indices:
            mainStart = checkpoints[0] - temp
            matched = 0

            while(matched <= (subLen - 1) and (mainString[mainStart] == subString[matched])):
                matched += 1
                mainStart += 1

            if(matched == subLen):
                matches.append(mainStart)
            else:
                continue


    if(subLen % 2 == 0):
        checkpoints = updateCheckpoints(checkpoints, gap)
    else:
        checkpoints = updateCheckpointsOdd(checkpoints, gap)
```

In both the best case and worst case, the preprocessing section would iterate (m/2) times.

Best Case: The algorithm assumes its best case if all the character pairs in the dictionary do not match any of the checkpoints placed on the main string till the end of the dataset. We will then be able to traverse through the dataset of size n skipping through (m/2) characters at once while having the least comparisons.

Then the algorithm just performs (m/2) + (n/(m/2)) = **O(n/m)**

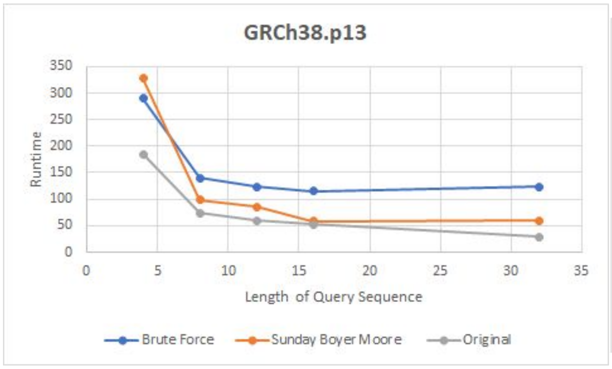
Worst Case: The algorithm assumes its worst case when both the dataset and sub string is made up of a repeated letter resulting in matches at every checkpoint possible with every single character pair in our dictionary. We will end up checking if the query sequence matches the main string at every index possible, making our algorithm only as efficient as the naive algorithm.

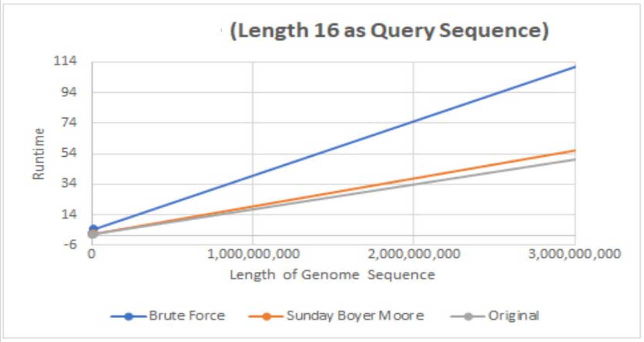
Then the algorithm just performs (m/2) + (n/(m/2))*(m/2)*(m) = (m/2) + (nm) = **O(n*m)**

D) EMPIRICAL ANALYSIS

		Run Time(Seconds) when length of Query Sequence is:				
Algorithm used	**Genome Sequence (Characters)**	**4 (CATG)**	**8(GTAT CACT)**	**12(AACGT AAAAGTT)**	**16(CCACCCT CTATCTTAT)**	**32(TGAAACGCTAA CAAATGATCGTAAA TAACACA)**
1. Brute Force	**ASM694v2(Bacteria) 4,951,383**	2.49	1.48	2.00	1.83	1.90
	R64(Fungi) 12,157,105	4.76	4.73	5.30	4.65	4.62
	GRCh38.p13(Human) 3,099,706,404	290	139	129	114	124
2. Sunday Boyer Moore	**ASM694v2(Bacteria) 4,951,383**	3.04	1.93	1.20	0.78	0.97
	R64(Fungi) 12,157,105	4.60	2.93	2.23	1.59	1.46
	GRCh38.p13(Human) 3,099,706,404	327	98	85	58	59
3. Original	**ASM694v2(Bacteria) 4,951,383**	1.97	1.19	0.97	0.87	0.57
	R64(Fungi) 12,157,105	4.44	2.39	1.86	1.36	0.81
	GRCh38.p13(Human) 3,099,706,404	185	74	60	52	29

For our test sets, we chose GRCh38.p13(Human), ASM694v2(Bacteria) and R-64(Fungi) as the genome sequences. We then ran queries of length 4, 8, 12, 16, and 32 against the genome sequences. We recorded the runtime varying the query or genome lengths while keeping the other constant. This allowed us to identify a trend for each algorithm and enabled us to compare the efficiency of each algorithm.





Graph 1

Graph 2

1) Query length vs. Runtime : Varying the length of the query sequence, and comparing the performance of each algorithm at these lengths in the same genome sequence.
2) Genome Sequence Length vs. Runtime : Varying the length of the genome sequence while keeping the query length constant, and comparing the performance of each algorithm under these conditions.

As we can see from the table, the runtime of the **Brute Force algorithm does not significantly vary at different query lengths**. The performance of this algorithm saturates when the query length crosses 8 characters.As for the **Boyer Moore Horspool Sunday (BMHS) algorithm, we see an overall decrease in the runtime as the query length increases**.

Finally, **in the original algorithm, we observe a stable and constant decrease in the runtime as the length of the query sequence increases**.

E) CONCLUSION:

1. These observations on the basis of the 1st comparison stated above indicates that the Original algorithm and BMHS would perform better on longer queries, where the original algorithm constantly outperforms BMHS.
2. The observations made on the basis of the 2nd comparison stated above indicates that the Original algorithm shows the best performance in larger genome sequences, followed by BMHS and Brute-Force respectively.

REFERENCES:

1. http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/Docs/Boyer-Moore-variants.pdf
2. https://www.tutorialspoint.com/python-support-for-gzip-files-gzip
3. https://www.inf.fh-flensburg.de/lang/algorithmen/pattern/sundayen.htm

CONTRIBUTION STATEMENT:

”We all declare that we have contributed equally towards researching, implementing, testing, reporting and presenting the algorithms. We have also referenced all the work we took inspiration from, to the best of our abilities.”

Signed By:

Jaheez Aneez Ahmed

Lim Jin Yang

Taneja Parthasarthi

Tan Hu Soon

Wu Jun Yan