Project Report

CSC 7300 Algorithms Design and Analysis
Fall 2024

Submitted by:

Nurjahan	nurja1@lsu.edu
Nushrat Jahan Ria	nria1@lsu.edu
Tania Khatun	tkhatu1@lsu.edu
Saima Sanjida Shila	sshila1@lsu.edu

Date of Submission: December 04, 2024

Type and Title of the Project

Implementation projects (Burrows-Wheeler Transform Based compression/indexing algorithm)

1 Introduction

1) **Problem Statement:** Searching for patterns within large text datasets—like DNA sequences, Protein structures—comes with significant challenges and these challenges arise due to the high computational and memory demands required to manage and process such vast amounts of data in real life. This success hugely depends a lot on how quickly the search query is being executed and how efficiently memory is being used, and the biggest thing is that many traditional data structures fail in this regard especially as data sizes grow which can be considered as the biggest limitations of the conventional approach. Our goal is to show that large-scale biological and structured datasets can be handled much more efficiently and in much less time, which is particularly important for applications in fields like Bioinformatics, Genomics, Search Engines and Information Retrieval where both speed and memory usage are critical. Our main objectives are as follows:

    - To solve the problems of efficiently managing and searching through large datasets by compressing them using a specialized data structure called the FM-index, which is built using the Burrows-Wheeler Transform (BWT) and a wavelet tree and this proposed algorithm will ensure that we can find an pattern within a text.
    - The primary goal of this project is to create a system that can efficiently search for patterns and count the number of times a pattern appears in the text by optimizing both processing time and memory usage of a system.
    - To compare the performance between the traditional suffix array approach and optimized approach using FM-index.

2) **Input:** DNA sequences, Protein structures datasets from Pizza & Chili corpus.
3) **Output:** Find a pattern and count the number of times a pattern appears in the text.

2 Proposed Algorithmic Methodology

The necessary steps for the proposed algorithm are as follows:

i. **Brute force algorithm (Suffix Array):** A suffix array is an array of all suffixes of the text, sorted in lexicographical order. Once the array is built, substring search is done using binary search over the suffixes, enabling efficient lookup.

ii. **Sophisticated algorithm:** We will create an FM-index (Full-text Minute-space) based on Burrows-Wheeler Transform (BWT) of the text. The FM-index is a compressed self-index, which means that it compresses the data and indexes it at the same time. The steps will be:
   a) Create a Suffix Array in O(n) time.
   b) Build Burrows-Wheeler transform (BWT) from Suffix Array
   c) Construct Wavelet tree over BWT
   d) Query algorithm for the Wavelet Tree
   e) Backward pattern matching by using FMIndex.

3 Background Study

3.1 Suffix Array

A suffix array is an array that contains the starting indices of all suffixes of a string sorted in lexicographical order and is used for efficient text processing and pattern matching in real-life applications. Let us consider the string T[0...n] = "QPRPRP".

Now the suffixes of the string will be:

Suffix[0] = QPRPRP$, Suffix[1] = PRPRP$, Suffix[2] = RPRP$, Suffix[3] = PRP$, Suffix[4] = RP$, Suffix[5] = P$, Suffix[6] = $

After sorting the suffixes in lexicographical order, we get:

Suffix[6] = $, Suffix[5] = P$, Suffix[3] = PRP$, Suffix[1] = PRPRP$, Suffix[0] = QPRPRP$, Suffix[4] = RP$, Suffix[2] = RPRP$

Now we get the suffix array of string T[0…n] as [6,5,3,1,0,4,2], which represents the lexicographically sorted order of all the suffixes and facilitates efficient search operations of string.

If n is the length of the text, then constructing a suffix array requires O(n^2 log n), where substring search requires O(m log n) time if we apply binary search and the space complexity will be O(n). In this project, to decrease the overall time complexity, Karkainnen and Sanders DC3 algorithm has been implemented to compute the suffix array in O(n) time.

3.2 DC3 algorithm

The DC3 algorithm is a fast and efficient method for creating a suffix array which works by breaking the problem into smaller parts and processing them step by step. Radix sort is applied to arrange the suffixes into their correct order. In this algorithm, suffixes are divided into two groups and independent ranking is given to each group. To make the process faster, one group is sorted before the other group, which reduces the number of index comparisons. Finally, the suffix array is generated by combining these two groups, resulting in a time complexity of O(n).

3.3 Burrows-Wheeler Transform

Burrows-Wheeler Transform is a technique that is used as a data compression algorithm in real-life applications. The string of the BWT contains the same characters as the original text S, but the characters are reorganized in a particular order to facilitate compression. If there are n characters, then |n| rotations of the string will be needed to generate the BWT text, where the final array will be lexicographically sorted. Finally, the last column of the lexicographically sorted array is considered as the BWT text.

Step 1: Suffixes of T

We list all the suffixes of T, with their starting positions:

Index	**Suffix**
0	QPRPRP$
1	PRPRP$
2	RPRP$
3	PRP$
4	RP$
5	P$
6	$

Table 1 Suffixes

Step 2: Sort Suffixes Lexicographically

This ordering corresponds to your suffix array:

SA={6,5,3,1,0,4,2}

Step 3: Compute the BWT Using SA

We used the following formula to generate Burrows-Wheeler Transform (BWT) text:

$$BW[\,i\,] = \begin{cases} S[SA[\,i\,] - 1] & \text{if } SA[\,i\,] \mathrel{!}= 1 \\ S[n] & \text{if } SA[\,i\,] = 1 \end{cases}$$

i	SA[i]	SA[i]-1	BWT[i]= T[SA[i]-1]
0	6	5	P
1	5	4	R
2	3	2	R
3	1	0	Q
4	0	6	$
5	4	3	P
6	2	1	P

Table 2 Building BWT

Finally, from table 2, we get the Burrows-Wheeler Transform text of QPRPRP$ is PRRQ$PP. Once the suffix array (SA) is constructed, the BWT can be derived in O(n) time by iterating through the SA.

3.4 Wavelet Tree

For the construction of the Wavelet Tree, we need to consider three variables which are 'currentInteger,' 'bitPosition,' and 'oneCount', where the variable currentInteger stores bits in an array of unsigned integers, bitPosition tracks the position of the current bit (from 0 to 31), and oneCount keeps track of how many have been encountered so far. A lambda function is used to check whether the current character falls beyond the middle of the alphabet for the tree branch or not. If it does, then the value of the currentInteger will be updated by adding $2^{\text{bitPosition}}$, and oneCount will be increased when such characters are found. Additionally, when currentInteger reaches a multiple of 32 or oneCount reaches a multiple of the block size, their values will be saved to the respective arrays.

The time complexity of the Wavelet Tree construction requires O(n log σ), where σ denotes the length of the alphabet because this algorithm organizes the characters into two groups based on the lambda function and then recursively calls the function to build the left and right child nodes.

The Burrows-Wheeler Transform text PRRQ$PP can be compressed using a wavelet tree where the alphabet {$, P, Q, R}is represented with binary encodings such as {$, P} = 0 and {Q, R}=1. At the root, the left child is formed where all characters are encoded as 0, and the right child is formed where all characters are encoded as 1. This process is recursively repeated until each character is uniquely represented by a 0 or a 1 at the leaf level.

After applying this formula, we get the following wavelet tree for the string PRRQ$PP.

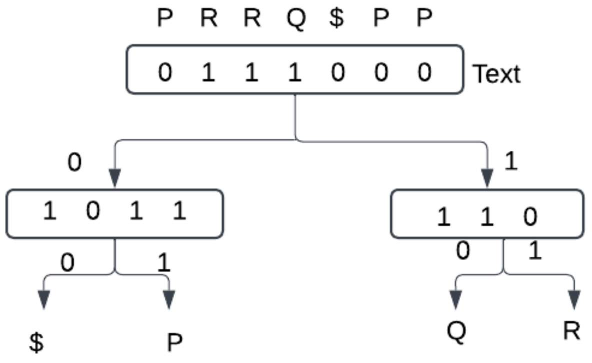


Figure 1 Wavelet Tree for the String PRRQ$PP

3.5 Implementation of Bit Vector Rank

To count the bits in the bit vector up to a given index, we implemented rank1() and rank0() functions by summing up values starting from a specific bit where the rank1 function counts the number of 1s in the bit vector from the beginning to a given index, and rank0 counts the number of 0s. Depending on whether the character is above the middle of the alphabet range for the node, we used rank1() or rank0() functions and then moved to the corresponding child. The whole process requires O(n) time complexity.

Suppose we want to count the occurrences of Q in BWT[2:6] where BWT Text = PRRQ$PP

Step 1: Identify the Range in the Root Node

- Text Range: BWT[2:6]=R,R,Q,$,P
- The root node's bit vector: [0,1,1,1,0,0,0]
- Perform a rank operation to calculate the range corresponding to the right child (Q, R):
    - Compute $\text{rank}_1(1) = 0$ (number of 1s before index 2).
    - Compute $\text{rank}_1 (6) = 3$ (number of 1s up to index 6).
    - The substring BWT[2:6] contains 3 characters in Q, R.

Step 2: Move to the Right Child (Q,R)

- The bit vector for the right child is: [1,1,0]
- Perform a rank operation to calculate the range corresponding to Q:
    - Compute $\text{rank}_0(0)=0$ (number of 0s before the first position).

- Compute $rank_0(3)=1$ (number of 0s up to the third position).
- The substring contains 1 occurrence of Q.

Final Answer: Character Q appears 1 time in BWT[2:6].

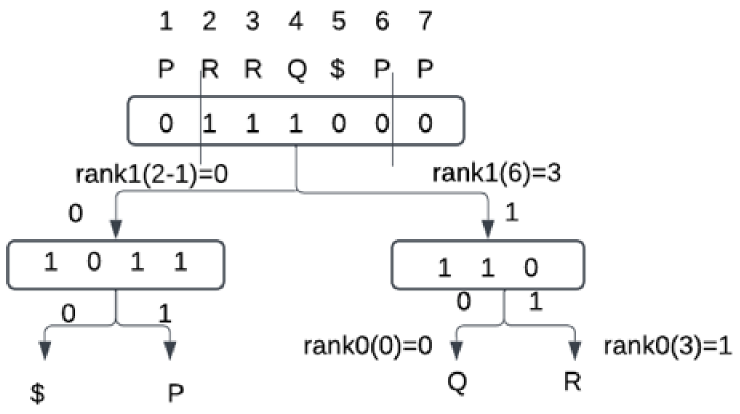


Figure 2 Number of Times Character Q Appears in BWT[2:6]

3.6 Occurrence Table

A lookup table stores all characters in the string S that are lexicographically smaller which requires the O(n) time complexity. Let us consider the string T = QPRPRP$. Now to create the occurrence table for each character, we need to calculate cumulative count of all characters that are lexicographically smaller than that character. For example, if we consider P then we found that only $ is lexicographically smaller than P which occurs only one time that means the cumulative count for P will be 1.

Character	Cumulative Count	Explanation
$	0	No characters are lexicographically smaller than $.
P	1	Only $ is lexicographically smaller than P, and it occurs once.
Q	4	Only $ and P are lexicographically smaller than Q where $ (1 occurrence) and P (3 occurrences). Hence, the cumulative count will be 1+3=4.
R	5	$, P and Q are lexicographically smaller than R where $ (1 occurrence), P (3 occurrences), and Q (1 occurrence). Hence, the cumulative count will be 1+3+1=5.

Table 3 Constructions of Occurrence Table

$	**P**	**Q**	**R**
0	1	4	5

Table 4 Final Occurrence Table

3.7 FM Pattern Matching

In this project, we used the FMIndex pattern-matching algorithm to search for the substrings using the Burrows-Wheeler Transform concept. If p is the length of the pattern, then this algorithm requires O(n * p) time which actually works by going backward through the pattern until the match is found. In this algorithm, two variables, start point and end point, are used where the value of the start point variable is set to 1 and the value of the endpoint variable is set to n. For each character in the pattern, the values of the start and end point variables will be updated based on the previous values. In this way, any suffix within the updated start and end points will match the given pattern.

So, our BWT string is "PRRQ$PP". Let us consider the pattern P=PRP. Backward search processes the pattern from right to left, starting with the last character. From the occurrence table we get cumulative count of the characters are:

$$C[\$]=0,\ C[P]=1,\ C[Q]=4, \text{ and } C[R]=5$$

Now we will follow the following steps to perform backward search using the FM-Index.

Step 1: Initialize

Start with sp=1 and ep=7 (the full range of BWT). [sp depicts start point and ep depicts end point]

Step 2: Process Last Character (P[3]=P)

1. Compute sp: sp=C[P]+Occ(P,sp−1)+1; sp=1+Occ(P,0)+1=1+0+1=2

   [Here, Occ(c,i) gives the number of times the character c appears in BWT[1…i]]

2. Compute ep: ep=C[P]+Occ(P,ep); ep=1+Occ(P,7)=1+3=4

New Range: [sp,ep]=[2,4]

Step 3: Process Second Character (P[2]=R)

1. Compute sp: sp=C[R]+Occ(R,sp−1)+1; sp=5+Occ(R,1)+1=5+0+1=6
2. Compute ep: ep=C[R]+Occ(R,ep); ep=5+Occ(R,4)=5+2=7

New Range: [sp,ep]=[6,7]

Step 4: Process First Character (P[1]=P)

1. Compute sp: sp=C[P]+Occ(P,sp−1)+1; sp=1+Occ(P,5)+1=1+1+1=3
2. Compute ep: ep=C[P]+Occ(P,ep); ep=1+Occ(P,7)=1+3=4

New Range: [sp,ep]=[3,4]

The formula for calculating the number of occurrences is ep−sp+1. For the final range [3,4], the number of occurrences is ep−sp+1=4−3+1=2. Thus, the pattern PRP appears twice in the text: T=QPRPRP$

4 Experimental Results

4.1 Experiment Setup

4.1.1 Dataset Details: Here, we have applied two different datasets from the Pizza&Chili Corpus (https://pizzachili.dcc.uchile.cl//) to test a Burrows-Wheeler Transform-based compression and indexing algorithm.

1) **Real or Synthetic:** The datasets are all real-world collections from respected sources, each prepared to represent a different domain. Each of these datasets has been compiled and maintained with real-world characteristics in their data, making them ideal for analyzing practical performance for compression algorithms. Datasets include the following:

    - PROTEINS: protein sequences from the SwissProt database.
    - DNA: gene sequences from the Gutenberg Project.

2) **Data Size:** Each dataset has been capped at a size of 50MB, keeping experimental variables under control and allowing comparability. This size allows for sufficient data to observe the effect of BWT on compressibility and indexing across text types but not so much that computation overloads which ensure processing times remain reasonable while yielding meaningful results.

4.1.2 Configuration: The proposed algorithm is implemented on a computer featuring an 8-core CPU: MacBook with an M3 chip, containing 4 performance cores and 4 efficiency cores, all supported by 128GB of RAM. The IDE Visual Studio Code was used to carry out the development in C++. This combination was favored for its ease of setup, reliability, and consistency of the development environment, and ease of access to needed libraries.

4.2 Results and Discussion

4.2.1 Using Brute force algorithm (Suffix Array):

We have first applied Brute force algorithm such as suffix array to find the occurrences of the pattern and find that it takes a long time for the intended task.

Dataset	Array Creation Time (ms)	Pattern Matching Time (ms)
Proteins	407109	0
DNA	204321	0

Table 5 Performance of Suffix Array (Brute Force Method)

4.2.2 Using Sophisticated algorithm:

Pattern	No of Occurrence
DCLRR	17
DRLMQQLQDLEE	1
DLPTWEES	1
GNDYEEFGAFGGYGTLT	3
GRG	22297
GAAS	1570
KHTATARF	3

Table 6 Occurrence Count of Some Patterns in Proteins Dataset

Block Size	Array Creation Time (ms)	Wavelet Tree Creation Time (ms)	Pattern Matching Time (ms)
2	19962	9259	226
4	20596	6281	98
8	19812	5092	67
16	20381	4616	50
32	19571	4374	34

Table 7 Comparison of Time: based on Block Size (Protein Dataset)

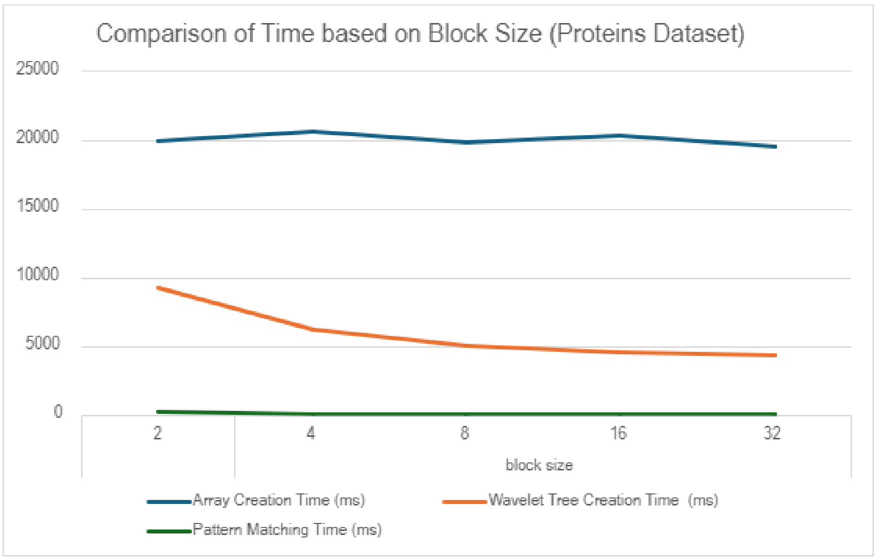


Figure 3 Time for Different Block Sizes (Protein Dataset)

Pattern	No of Occurrence
GGCCAGATA	195
AATCTAGACAGAT	5
GTCTT	59235
GGC	667273
CTTTTTAATTT	189
GGGTTTTCTCCA	12
GAATTGTGCTGCT	14

Table 8 Occurrence Count of Some Patterns in DNA Dataset

Block Size	Array Creation Time (ms)	Wavelet Tree Creation Time (ms)	Pattern Matching Time (ms)
2	17531	7947	202
4	17557	5652	104
8	18263	4636	71
16	17616	4322	57
32	18057	4031	37

Table 9 Comparison of Time: based on Block Size (DNA Dataset)

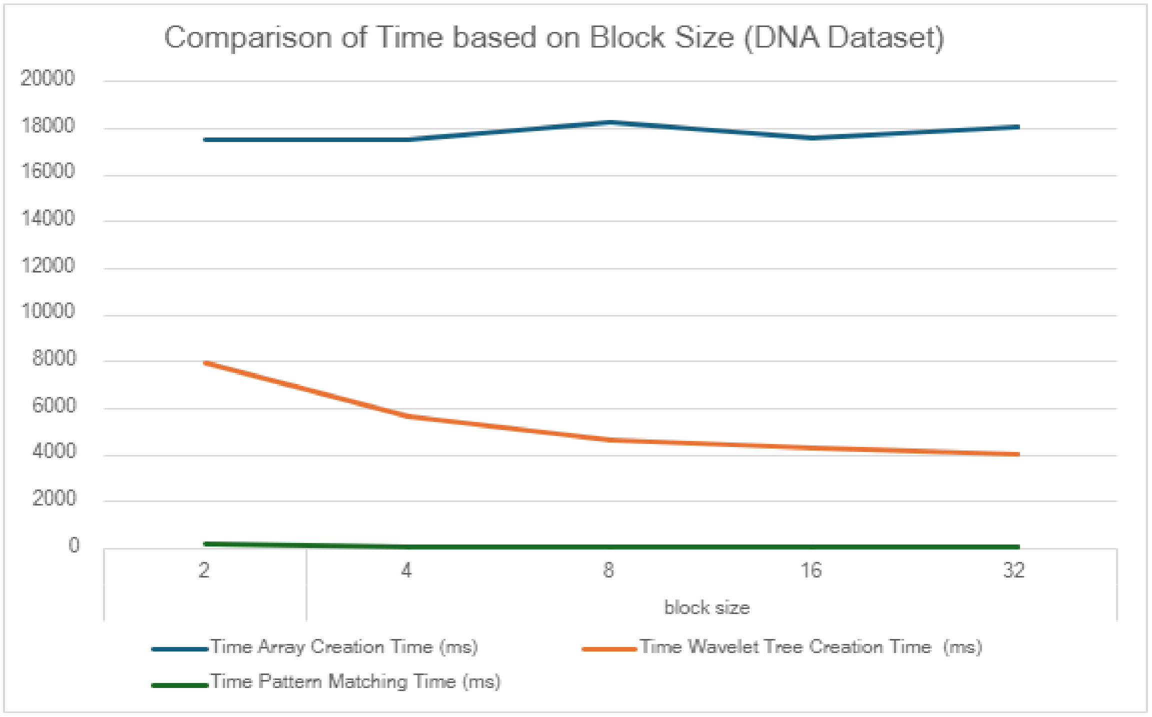


Figure 4 Time for Different Block Sizes (DNA Dataset)

4.2.3 Space Analysis:

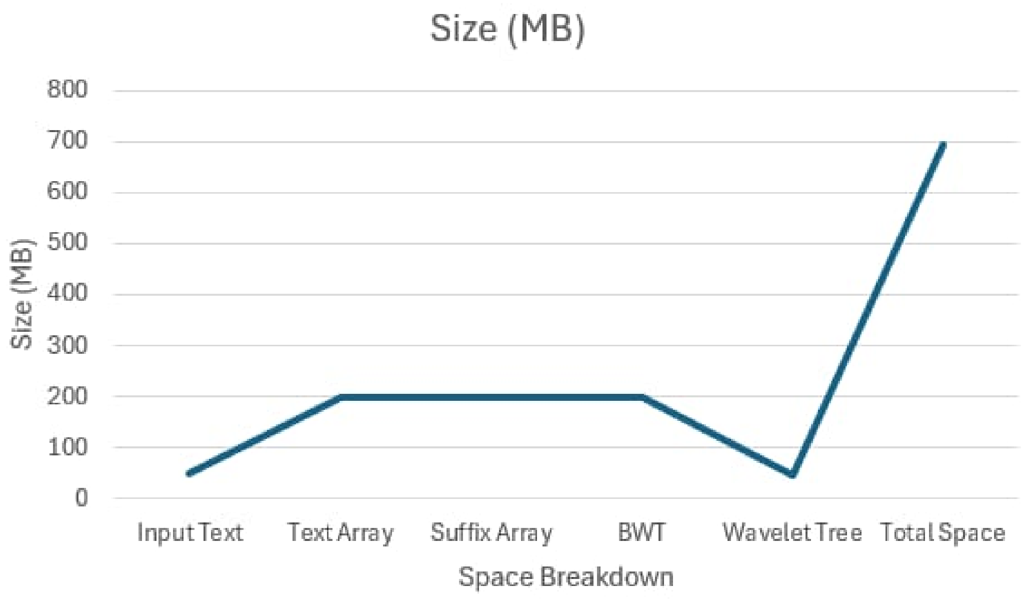
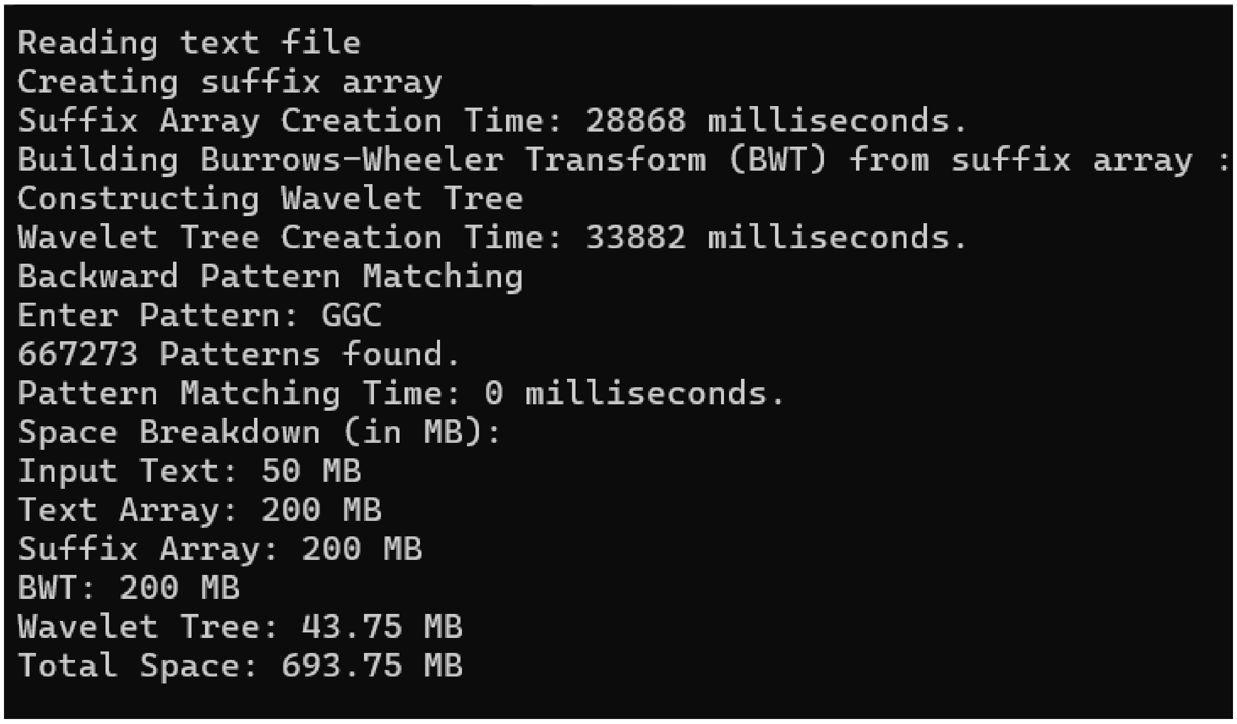


Figure 5 Space Breakdown of Pattern Matching Query



```
Reading text file
Creating suffix array
Suffix Array Creation Time: 28868 milliseconds.
Building Burrows-Wheeler Transform (BWT) from suffix array :
Constructing Wavelet Tree
Wavelet Tree Creation Time: 33882 milliseconds.
Backward Pattern Matching
Enter Pattern: GGC
667273 Patterns found.
Pattern Matching Time: 0 milliseconds.
Space Breakdown (in MB):
Input Text: 50 MB
Text Array: 200 MB
Suffix Array: 200 MB
BWT: 200 MB
Wavelet Tree: 43.75 MB
Total Space: 693.75 MB
```

Figure 6 Output Picture for DNA Dataset (Block Size=32)

Here, if we analysis the result we can see that sophisticated algorithms can create suffix array faster than the brute force algorithm. The array creation time for the brute force algorithm is 407,109 ms and 204,321 ms for the protein and DNA datasets, respectively, while the time for the sophisticated algorithm is 20,064 ms and 17,804 ms for the protein and DNA datasets, respectively.

In addition, sophisticated algorithms require less space than brute force algorithms. While sophisticated algorithms require 693.75 MB indifferent to block size, brute force algorithms require 1050 MB.

4.2.4 Complexity Analysis:

Sophisticated Algorithm (FM Index):

- Time Complexity: Building the FM-index involves computing the Burrows-Wheeler Transform of the text and constructing auxiliary data structures like the occurrence table. Time Complexity for construction is O(nlogn) or O(n) for efficient implementations. The memory efficient FM Index data structure stores text in such a way that a substring query can be answered in optimal asymptotic time, i.e., O(L) time for a substring of length L.

- Space Complexity**:** Near entropy-bounded, typically $O(nH_k)$, where H_k is the k-th order empirical entropy of the text, which makes the FM-index highly space-efficient.

Brute Force Method (Suffix Array):

- Time Complexity:
  - Construction Time: The naive approach for constructing a suffix array (by explicitly sorting all suffixes) takes $O(n^2 \log n)$ where n is the length of the text, because sorting all suffixes is costly.

  - Search Time: Once the suffix array is built, substring search can be done in O(m log n) time, where m is the length of the pattern, using binary search on the sorted suffixes.

- Space Complexity**:** The suffix array requires O(n) space to store the array of suffixes.

5 Conclusions

In this project, by examining two different datasets we have seen how the speed is affected if we change the block size of FMIndex. To search the substring easily from a text FMIndex which is implemented using the Burrows-Wheeler Transform is an efficient algorithm and widely used in bioinformatics for DNA sequence analysis in real-world applications. By utilizing the backwards search technique, FMIndex can find the sequences very quickly, often in constant O (1) time. FM Index not only optimize the search query time, but also compresses the space.