Burrows-Wheeler Transform
Executive Summary

Eddie Li (grc4rz), Luke Del Giudice (wcc5ub), Siddharth Premjith (rqf8pe)
University of Virginia

1 Introduction

A core challenge in data compression is reshaping strings so that repeated patterns are easier to exploit. Many natural strings, such as English text or DNA sequences, contain repetitions that are not efficiently handled by traditional compression. The Burrows-Wheeler Transform (BWT) addresses this by rearranging the string to group similar characters together. This property can then be effectively taken advantage of by compression algorithms. Additionally, BWT has the feature that it is easily reversible. In particular, as long as you have the transformed string, you can easily get the original string without storing extra data. This simplicity contrasts with other methods that may require more complex metadata or models to achieve reversibility.

The naive forward implementation of BWT has $O(n^2 log(n))$ complexity. When fully optimized and using a suffix tree to conduct the string encoding, it has a runtime complexity of $O(n)$ and a space complexity of $O(n)$. The naive complexity to inverse the BWT requires $O(nlog(n))$ time.

2 Overall Intuitive Approach / Solution

The BWT algorithm can be broken down into a few steps. First, we add a character at the end, such as \$, to mark the end of the string. The idea is that this character comes alphabetically before anything in our possible character universe. Then, we will generate all cyclical rotations of our string, which are just all possible permutations of the string maintaining the same order. After, we sort these rotations lexicographically. Finally, we will concatenate all of the last characters of the lexicographically sorted strings to get our transformed string.

Here is a specific example to clarify this process:

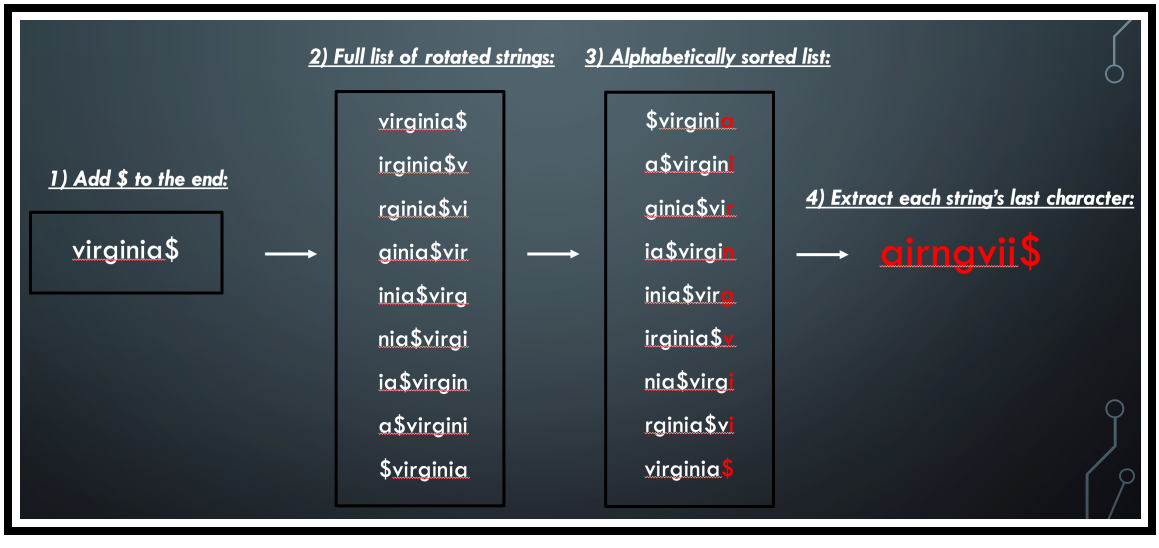

Figure 1: Full Forward BWT Process

On the left hand side, we first add the \$. Then, under step 2, we create all the cyclical rotations, as seen in the black box. We then transform this into the lexicographically sorted list, as seen in the box on the right. Then, we take the last character in each of the strings (highlighted in red), and concatenate them.

This string can then be inputted into a string compression algorithm, and once the original string needs to be retrieved, the BWT output can be retrieved through decompression, then turned into the original string in the following manner:

1. First, assign an index for each repeated character relative to where it appears in the BWT. For instance, the first a should be a_0, the second a should be a_1, and so on.

   - $a_0 i_0 r_0 n_0 g_0 v_0 i_1 i_2 \$$

2. Then, create a sorted version of this list of characters, including the indices.

   - $\$ a_0 g_0 i_0 i_1 i_2 n_0 r_0 v_0$

3. Then, create a one-to-one correspondence of the characters. Since the BWT is ordered by the last character of the sorted strings, that means that each character of the sorted Burrows-Wheeler transform is the corresponding first character of the unsorted BWT. So in the image below, the left hand side is the string of the sorted list of characters, and the righthand side is the BWT-transformed string with the indicies. We create a one-to-one correspondance as shown by the arrows:

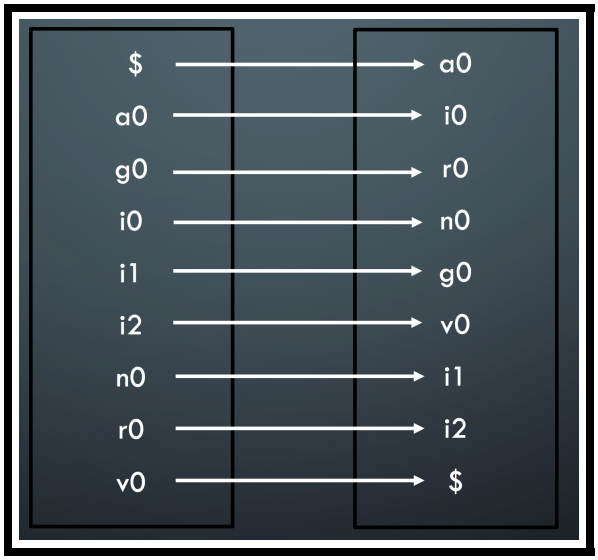

Figure 2: Mapping Backward BWT Process

4. Then, we can use the knowledge that the character at the end of a string must be before the character at the beginning of the string, due to the cyclical nature of how the BWT works. Thus, for \$, the character it points to must be the last character of the original string. Thus, add that to the end of the string. Repeat this step until the special starting character \$ has been reached.

   - $\$ \rightarrow a_0$: a_0
   - $a_0 \rightarrow i_0$: $i_0 a_0$
   - $i_0 \rightarrow n_0$: $n_0 i_0 a_0$
   - $n_0 \rightarrow i_1$: $i_1 n_0 i_0 a_0$
   - $i_1 \rightarrow g_0$: $g_0 i_1 n_0 i_0 a_0$
   - $g_0 \rightarrow r_0$: $r_0 g_0 i_1 n_0 i_0 a_0$
   - $r_0 \rightarrow i_2$: $i_2 r_0 g_0 i_1 n_0 i_0 a_0$
   - $i_2 \rightarrow v_0$: $v_0 i_2 r_0 g_0 i_1 n_0 i_0 a_0$
   - $v_0 \rightarrow \$$: $\$ v_0 i_2 r_0 g_0 i_1 n_0 i_0 a_0$

5. Remove the special starting character and the indices. The resulting string should be the original string pre-BWT.

- $\$v_0i_2r_0g_0i_1n_0i_0a_0 \rightarrow$ virginia

A note about why step 4 works for repeated characters is as follows: if the sorted set of cyclical strings has the first two characters be "ab", "ag", and "at", that means that in the sorted set of the strings, the cyclical strings starting with "a" must be in order of the 2nd character. From this observation, it can also be gathered that these 2nd characters must be at the start of the cyclical string at some point. In that situation, the corresponding "a" ends up going to the end of the string, and each "a" will appear in order of the 2nd character, like as follows: "b___a", "g___a", "t___a". Thus, the order is maintained across both the sorted set of strings and the Burrows-Wheeler transform string since in both cases, it depends on the character immediately following the repeated character, which is why the sorting can maintain the order present within the BWT.

3 Implementation

Implementing the basic form of the BWT through following the sorted rotations method requires two components: conducting the BWT on an input string and reversing the BWT on a transformed string. Both are described below.

3.1 Conduct the BWT

1. Ensure that the input string has a "$" added to the end of the string in order to track the original order of the string characters.
2. Find all the rotations of this string. Doubling the original string and indexing each set of n characters, n referring to the length of the original string, is one approach to simplify finding the rotations.
3. Sort the rotations through string comparison.
4. Set a variable for the BWT string. Iterate through the sorted rotations in order and concatenate the last character of each rotation to this variable. This resulting string is the final BWT string.

3.2 Reverse the BWT

1. Given a transformed BWT string, find two pieces of information:
   (a) Find the rank of each character of the BWT string in order, with the rank referring to how many times the character has appeared previously in the string.
   (b) Find the total appearances of each unique character.
2. Use the total appearances of each character to determine the first index of the sorted rotations each character prefixes.
   (a) Start with the lowest value character and set its index to 0.
   (b) For each character, increase the index by the amount of appearances of this character.
   (c) The subsequent character should start prefixing rotations after all appearances of this character, so the new index represents when the subsequent character starts prefixing rotations. If a subsequent character exists, set its index to the current index value. An example is if 'a' prefixes indices 0-3, then 'a' appears 4 times, and if the subsequent character is 'b', then it starts prefixing at index 4.
   (d) Repeat the last two steps for each new character.
3. Create the original string by utilizing the knowledge that the character at the end of an instance of the sorted rotations string is a part of the BWT string. $bw[rowi]$ represents what character is at the end of index $rowi$ of the sorted rotations. Thus, starting with the end of the BWT string, \$, which prefixes 0, we know the character at $bw[0]$ is the last character, and this can be prepended to \$. Then, for each character being prepended, we can recalculate $rowi$ based on the first instance in the sorted rotations the character prefixes, $first[c]$ and its rank in the original BWT string, $ranks[rowi]$. This is summarized in the code snippet below.

```
rowi = 0
t = '$'
while bw[rowi] != '$':
    c = bw[rowi]
    t = c + t
    rowi = first[c] + ranks[rowi]
```

(a) It is worth noting that the while-loop ends when $bw[rowi]$ equals \$, as this means that the beginning character of the original string is reached, as \$ is cyclically before the beginning character.

3.3 A brief note on the suffix-array optimization

One observation you can make is that if you look at the matrix, and you look at each substring up to the dollar sign, you notice that each suffix of the original string appears once. From this, you can then notice that the order of the strings in the matrix is the same as if you were to order all the suffixes of the string. Then the last column is just found by getting the character before the first letter of the suffix.

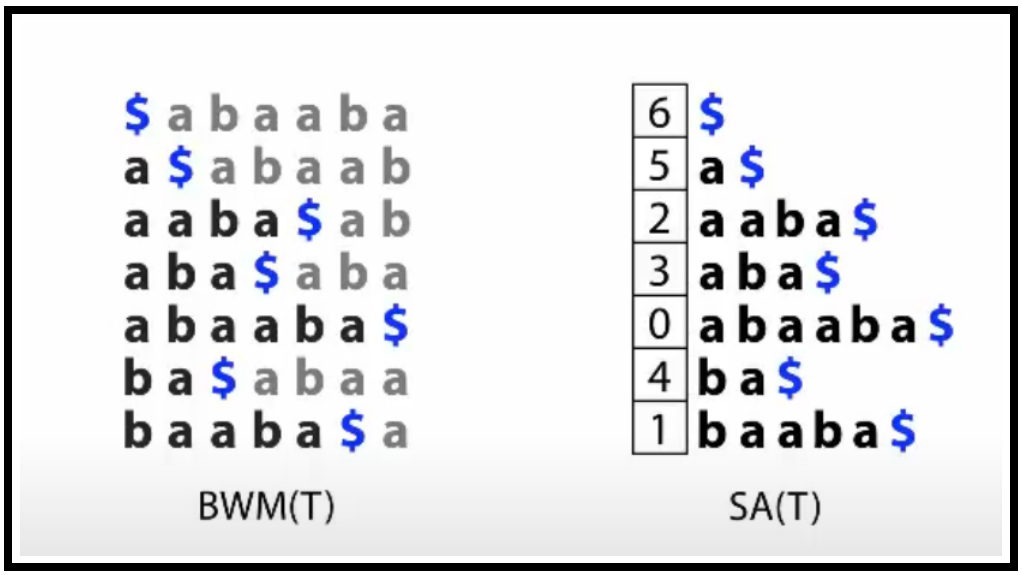

Figure 3: Equivalence to Suffix Array

MAKE SURE TO CREDIT THIS IMAGE

So in this image, the left side is the matrix of the sorted cyclic shifts. And on the right side, it's the substring up to the dollar sign, and this is just the suffixes in sorted order. And then to create the final string, and you go through each suffix, and get the character that would come before the suffix.

This then reduces to finding the suffix array, which is basically just a list, where each index is a starting index of the suffix, if you were to sort the suffixes. This can be done naively in $nlogn$ time. However, rather than paying an extra log-factor to sort all n suffixes directly, the DC3 algorithm builds the suffix array in linear time by exploiting a simple divide and conquer trick:

1) Divide the suffixes into two groups: those starting at positions congruent to 1 or 2 mod 3, and those at positions 0 mod 3.

2) Conquer by recursively sorting the mod 1 and mod 2 suffixes using a radix sort on their first three characters to form a smaller problem of size roughly $2n/3$. Because each comparison is on a fixed-length tuple, each round is $O(n)$.

3) Induce the order of the mod 0 suffixes by one more linear pass: each mod 0 suffix is compared against the already sorted mod 1 suffix that follows it, again using fixed-length tuple comparisons.

4) Merge the two sorted lists (mod 0 and mod 1,2) in one final linear scan to produce the full suffix array.

Every step (fixed-length radix sorts, recursive calls on two-thirds of the data, and one linear merge) takes $O(n)$ overall. The result is the complete suffix array, which you can then use directly to generate the Burrows–Wheeler Transform by pulling the character immediately before each sorted suffix. Some supplemental visuals for this process can be found in the lecture slides, but any further depth on this approach is outside the scope of this project.

4 Programming Challenge

4.1 Summary of Programming Challenge

The essential challenge for the programming problem is to determine how many numbers can replace a placeholder number in a BWT-transformed array such that it can be inverted. The primary learning objective for this problem is for the solver to get a better understanding of how the BWT algorithm work, and how the inversion process corresponds to it. In order to solve this problem, the solver will likely go through a few examples of transforming an array with BWT and inverting it. By going through these examples, the solver should be able to make observations relating to limiting the number of values to check, as they will not need to check values that don't change the relative order of the numbers.

4.2 Key Ideas for Solving Programming Challenge

The naive method to solve this would be to brute force all possible numbers that the 0 could be. However, because these numbers can go to up 10^9, brute forcing is not feasible. So, more observations are needed to be made!

In order for a number to replace the 0, and the array to be valid, you need to draw a graph between the nodes on the lefthand side and righthand side, such that it hits all nodes and goes back to the first one, as this is essentially how the reverse algorithm for BWT works. But, if you increment one value on the left hand side, it isn't very clear what happens to the graph, as the relative order of nodes swap, and the edges get completely rearranged. However, one thing to notice is that when the order on the lefthand side stays the same, there is no change to the graph. So for example, if the numbers on the lefthand side are $-1, 3, 7, 10, 15$, and you incremented 3 to 4, the relative order of all the value stay the same, and the graph will stay the same. So if the graph is valid with 3, the graph will also be valid with 4. And similarly, if the graph is not valid with 3, it is also not valid with 4. This brings us to the observations that for each gap between consecutive sorted numbers, we only need to test 1 number for all the numbers in the gap, which is on the order of n. And for each test, we can it in $O(nlogn)$ time.

The images in Figure 4 give a good visual representation of this.

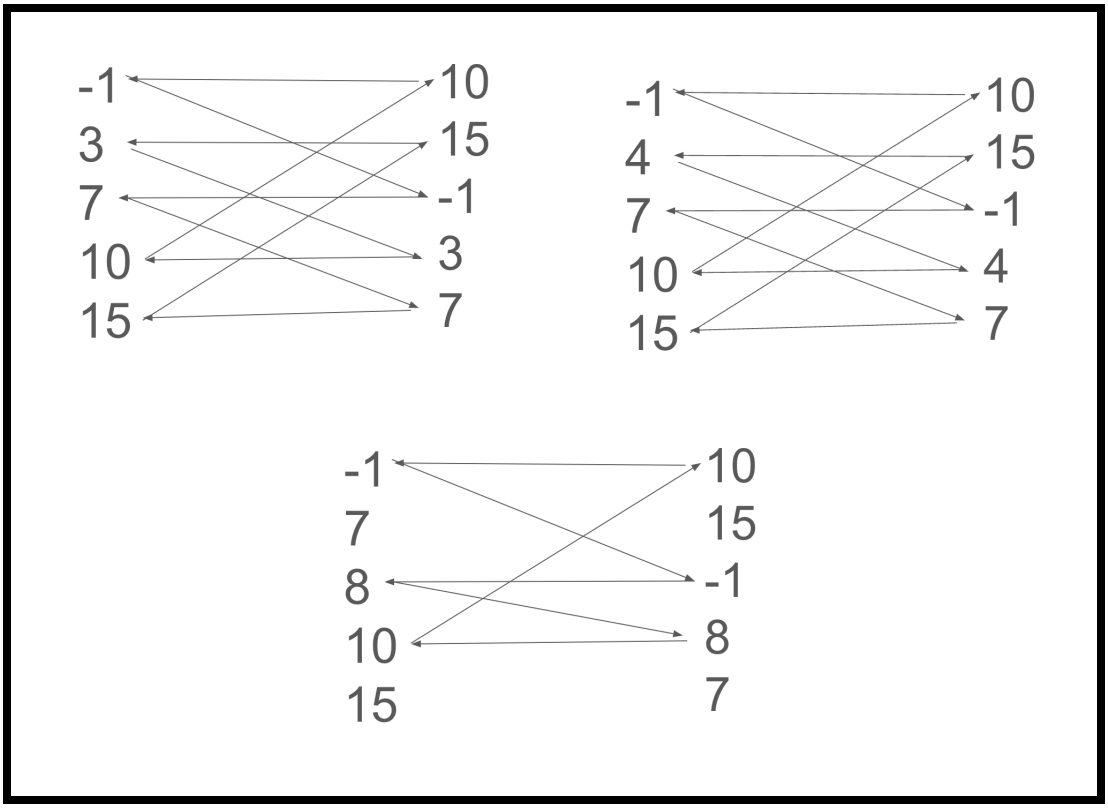

Figure 4: Programming Challenge Visual

So between the first image and the second image, we change the 3 to a 4, but the relative order of the numbers stay the same (the left hand side and the right hand side stay the same, aside from a relabelling of 3 to 4), and hence the graph of the arrows also stays the same. But, when we change from the first image to the third image, we change 3 to 8, and because the order changes on the left side, the arrows get completely changed. This brings us to the observation that changing the value 3 to anything between its neighbors (between 1 and 7 will not change the graph, but changing it outside this range might change the graph, and may make the BWT no longer valid).

So, we only need to change the 0 in our problem on the order of number of gaps between consecutive sorted numbers, which is on the order of n. And for each test, it takes $nlogn$ time to sort the lefthand side, and n time to get the mapping. So our solution is $n^2log(n)$. Additionally, we can get rid of this log factor if we don't resort the lefthand side for every change, and just swap values in the array.

We believe that this is a good challenge, as it initially feels very difficult to solve this without testing every possible number. But, it requires the solver to play around with how the BWT algorithm works, and draw a few of these graphs to realize the observations with the gaps. And after playing around with the problem for a bit, we believe that this observation is within reach of most peoples' ability.

5 Conclusion

The Burrows-Wheeler Transform (BWT) algorithm is a string preprocessing algorithm designed to enhance data compressibility by rearranging characters to group similar symbols together. It works by generating all cyclic shifts of the input string, sorting them lexicographically, and taking the last column of the sorted matrix as the transformed output. Though the naive implementation has a time complexity of $O(n^2logn)$, optimized versions using suffix arrays or suffix trees can achieve linear time and space complexity. Crucially, the BWT is reversible, enabling lossless compression and decompression.

The inverse BWT reconstructs the original string using a method called Last-First (LF) mapping, which leverages the relationship between sorted and unsorted versions of the transformed string. Using this method, we can get the original string of the BWT-transformed string $O(nlogn)$ time.

Our programming challenge explores the relationship between the transformation algorithm and the inverse algorithm and requires the solver to make observations relating to how the transformation/inverse changes when changing the values in the string.

To conclude this Executive Summary, we include two visualizations illustrating BWT's impact. Table 1 compares the compression ratios of five algorithms with and without a BWT pre-step. These measurements were taken on 5,000 random strings of length 5,000 generated either by uniform character selection or by sampling with a bias toward repeating substrings. As you can see, some methods (most notably Huffman Coding) gain little or ever incur extra overhead from metadata. Additionally, MTF, Arithmetic Coding, and Lempel-Ziv already natively capture similar substring structure. Therefore, the most pronounced benefit under BWT falls to RLE.

Figure 5 shows RLE's specific behavior: as the average run-length of similar substrings increases, the compressed size falls in an inverse relationship. This pattern is exactly what we expect; RLE encodes each run of identical characters as a single symbol plus a count, so it performs poorly on mixed substrings. BWT, however, rearranges the string so that repeated substrings become long runs of the same character, making RLE extremely efficient.

In summary, while BWT isn't the right preprocessing step for every compression algorithm, it can deliver substantial gains for any compression algorithm that relies on contiguous runs of identical symbols.

Compression Algorithm	**Uniform Selection**		**Substrings Likely Repeated**	
	Without BWT	With BWT	Without BWT	With BWT
Run-Length Encoding (RLE)	99.85	99.85	99.87	6.61
Huffman Coding	11416.55	11482.77	34375.87	34369.04
Move-to-Front Encoding	62.50	62.51	62.50	62.51
Arithmetic Coding	58.72	58.74	26.81	26.84
Lempel–Ziv (LZ77 / LZ78)	62.45	62.50	5.49	5.21

Table 1: Compression with and without BWT

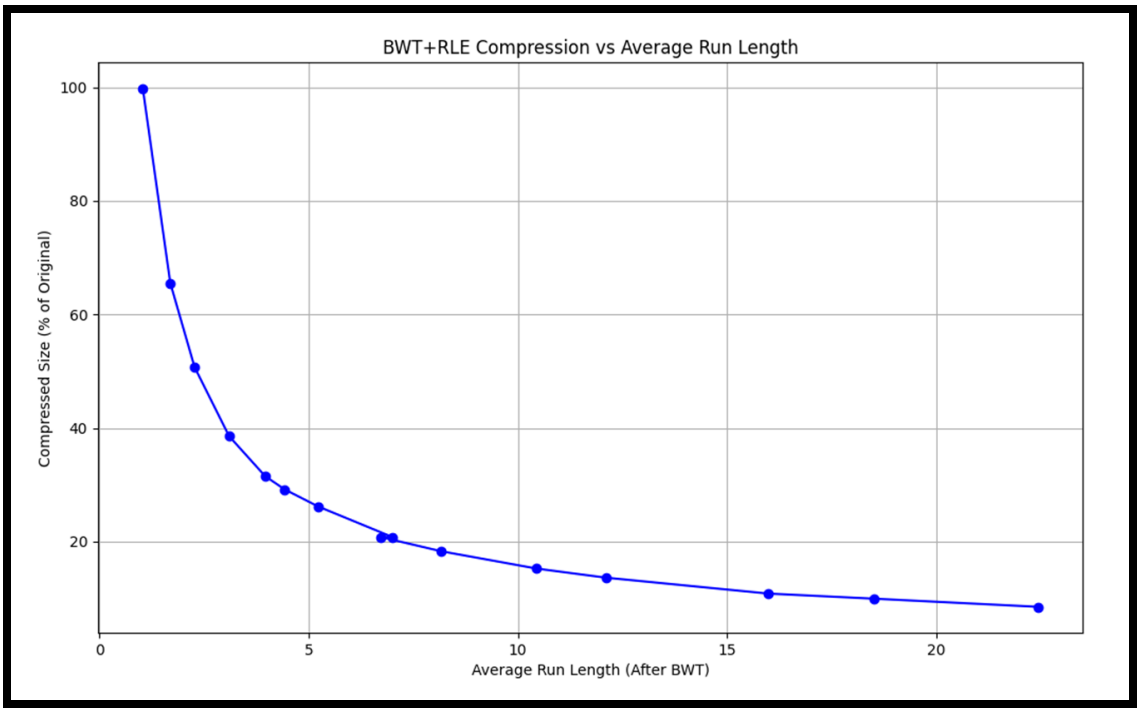

Figure 5: Redundancy Visualized

References

[1] Langmead, B. *Burrows–Wheeler Transform*. YouTube Video, published ca. Sept. 2014. Available: https://www.youtube.com/watch?v=4n7NPk5lwbI. [Accessed: 24-Apr-2025]

[2] Langmead, B. *CG_BWT_Reverse.ipynb*. Jupyter Notebook (rendered 24 Apr. 2025), nbviewer.org. Available: https://nbviewer.org/github/BenLangmead/comp-genomics-class/blob/master/notebooks/CG_BWT_Reverse.ipynb. [Accessed: 24-Apr-2025]

[3] Wikipedia contributors. *Burrows–Wheeler transform. Wikipedia, The Free Encyclopedia.* Last edited 23 April 2025. Available: https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform. [Accessed: 24-Apr-2025]

[4] GeeksforGeeks. *Burrows–Wheeler Data Transform Algorithm*. Last Updated: 08 Dec. 2023. Available: https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/. [Accessed: 24-Apr-2025]