REPORT

A Report submitted for completion of training

By
Dhiresh Kumar Singh
Associate Software Engineer

Ground Floor, UAS Alumni Association Building,
Veterinary College Campus, Bellary Rd,
Hebbal, Bengaluru, Karnataka 560024
July 2023

Contents

1. Color Blindness

Color blindness (also called color vision deficiency) is a visual disorder that makes it difficult to see certain colors or distinguish between different color combinations. Color blindness is caused by a gene mutation that affects the photoreceptors in the retina (the part of the eye that catches and processes light). Because of this, people with color blindness can't see certain colors or, in extreme cases, they can only see grayscale.

1.1 How is it tested?

There are a variety of tests that can be used to diagnose color vision problems, with some of the most common being listed below:

1.1.1 Ishihara Color Plates Test

One of the most commonly used tests for detecting color blindness is the Ishihara color plates test. This test involves placing a series of colored plates on a test subject. Each plate contains a pattern of colored dots in different sizes and colors. The test subject will be presented with a set of colored plates that contain hidden numbers and symbols. These hidden numbers and symbols are visible to the normal person, but invisible to the person with color vision deficiency. By asking the test subject to identify these hidden numbers and symbols, the eye care professionals will be able to determine whether or not the person has color vision deficiency.

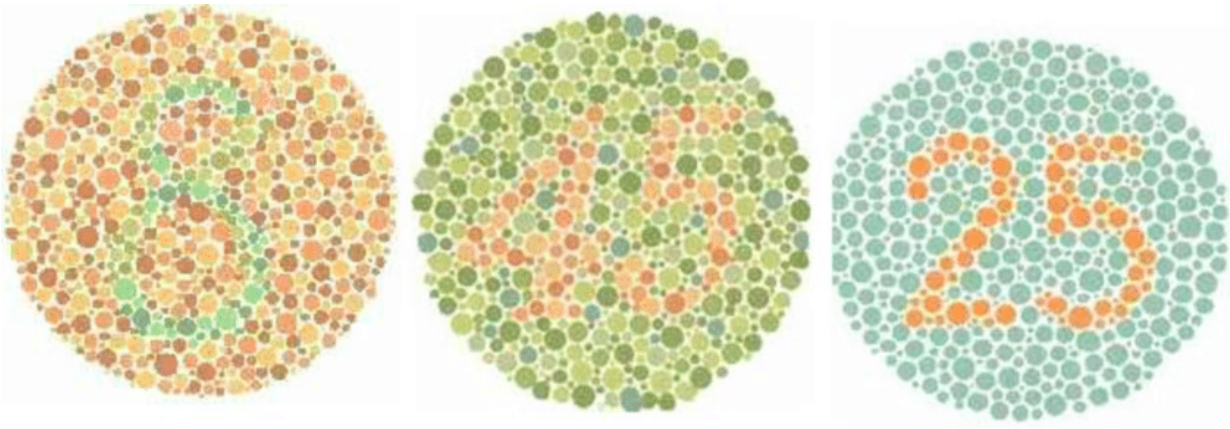


Fig1: *This image shows three (of the thirty eight) Ishihara test plates. If you are not color blind, you should be able to see the number 8 inside the left circle, 45 in the middle circle, and the number 25 on the right.*

1.1.2 Farnsworth D-15 Test

This test looks at how well a person can arrange their colored caps in the correct order based on the colors they're looking at. It's used by optometrists to diagnose certain kinds of color blindness by comparing the person's arrangement to the correct order.

1.1.3 Anomaloscope

Anomaloscope is a diagnostic tool designed to aid in the diagnosis of the most prevalent type of color blindness, red-green light blindness. It displays a combination of red light and green light, which the test subject is required to match by varying the intensity of these colors. Those with general color vision are likely to be able to accurately match the colors, whereas those with red green color blindness will have difficulty doing so.

1.1.4 Genetic Testing

In some cases, it is possible to test for specific gene mutations that cause color blindness. While genetic testing is not a common part of routine screening, it can be beneficial for people who have a family history of color blindness or who want to gain a better understanding of their condition.

1.2 What do we believe is the cause?

Color blindness is mainly caused by genetics, as it's often an inherited condition that's caused by specific gene mutations that target the photoreceptors in the retina. The photoreceptors, also known as cones, detect and react to various wavelengths of light, enabling us to see a wide range of colors.

Cones are composed of three distinct types, each of which is capable of perceiving different wavelengths of light. The short-wavelength cones (S) are able to detect blue light, while M cones are sensitive to green light, and L cones are able to detect red light. By combining signals from these three types, a wide range of colors and shades can be distinguished.

Color blindness can also be caused by mutations that affect the blue sensitive (S) cone, resulting in blue-yellow. This form of color vision disorder is less common in the general population.
On rare occasions, color blindness can also be acquired. The retina or optic nerve can be damaged by certain medical conditions, injuries or exposure to toxins, resulting in temporary or permanent loss of color vision. Acquired color blindness is much rarer than the congenital form.

1.3 How is the cause linked to our genomes?

The genetic basis of color blindness reveals an interesting relationship between genes and visual perception. X-linked genetic polymorphisms on chromosome X cause different degrees of color blindness.
Red-green color blindness is the most common form of color blindness. It is mainly caused by an X chromosome genetic mutation. The X chromosome is where the red and green cone pigments are located. A mutation in the X chromosome can cause the absence or alteration of the function of either of these pigments, making it difficult to differentiate between them. Males who have only one X chromosome are more likely to develop this condition if they inherited the defective gene from their mothers. Females who have two X chromosomes may carry the gene, but may not develop the condition themselves because they have a healthy copy on the other X chromosome.

1.4 What sort of genomic event do we suspect to be the cause of this problem?

Color blindness is caused by a genetic mutation. When it comes to color vision, the genomic event that causes color blindness is a mutation in the genes that make the light sensitive pigments in the cone cells of your retina. These mutations change the structure or the function of the pigments, making them less sensitive to certain wavelengths of light.
Color blindness can be caused by a variety of genetic mutations, and the type of genomic event can differ depending on the severity of the color vision deficiency.

1. **Gene deletion or alteration:** In some situations, a gene that controls the production of a particular cone pigment can be removed or altered, causing the pigment to be missing or not working as well as it should. For instance, in red and green color blindness, mutations in the genes that control the red and green cone pigment can cause them to not develop properly or not work at all.

2. **Gene Duplication:** On rare occasions, gene duplication can cause overproduction of some cone pigments, resulting in overlapping spectral sensitivity and color confusion.

3. **Point Mutation:** Point mutations are single-nucleotide polymorphisms (SNPs) in a gene that alter the structure or function of the cone pigments. These small changes can affect how cone pigments react to different wavelengths of light.

4. **Frame-shift Mutation:** Frame-shift mutations are caused by the insertion or deletion of nucleotides in the reading frame of a gene. These mutations can result in non-functional or shortened cone pigments.

2. Genome Sequencing

DNA sequencing, also referred to as genomics, is a cutting-edge laboratory technique employed to precisely sequence the sequence of nucleotide base sequences in a person's DNA. Human DNA, commonly referred to as a genetic blueprint, consists of a trillion nucleotide base sequences, which are represented by the nucleotide letters A(adenine), T(thymine), C(cytosine), and G(guanine), respectively.

The process of genomic sequencing enables researchers to analyze and interpret the entirety of an organism's DNA sequence, including human, animal, plant, or microbial DNA. This information provides invaluable insights into an individual's genetic makeup, uncovering the unique genes, genetic variants, and mutations found in their DNA.

Genome sequencing technologies come in a variety of forms, each with its own set of benefits in terms of precision, coverage, and price.

2.1 How does it work?

Genome sequencing involves breaking down an organism's DNA into smaller pieces, determines the sequence of nucleotides in each piece, and then rebuilds the entire DNA sequence. There are several steps involved in the sequencing process, some of which may be slightly different depending on the sequencing technology being used. However, the basic principles remain the same. Here's a brief overview of genomic sequencing:

1. **DNA Extraction:** The first step in the process is to take a sample that contains the DNA that you want to sequenced. You can take a blood sample or a tissue biopsy. You can also take a saliva sample or any other type of biological material that has DNA in it. The DNA will then be extracted and purified.
2. **DNA Fragmentation:** The DNA extracted is then broken down into smaller fragments using different techniques. These fragments can be hundreds of base pairs long or thousands of base pairs long.
3. **Library Preparation:**In order to make the DNA suitable for sequencing, special adaptors, or barcodes, are affixed to the end of the DNA fragment. These adaptors act as identifiers and are required for subsequent steps in the sequencing chain.

4. **Sequencing:** The pre-assembled DNA fragments are inserted into the sequencing machines. Different sequencing technologies employ different techniques to read the DNA base sequence. For instance, in NGS (Next-Generation sequencing), a common technique is called “sequencing by synthesis”, where fluorescently labeled nucleotides are incorporated into the DNA fragments. The sequencing machine reads each base in the DNA fragment and records the corresponding nucleotide sequence.
5. **Data Analysis:**Once the sequencing is complete, a large amount of raw sequencing data is generated. This raw data contains a large number of short DNA reads. The next step is to sort, filter, and sequence these reads to match a reference genome (for whole genome sequencing) or to sequence them together (for de novo sequencing).
6. **Genome Reconstruction:** To reconstruct the full DNA sequence of an organism, the sequence reads are aligned with a reference genome, or larger contigs are assembled from the sequence reads.
7. **Variant Calling:** Once the genome has been reconstructed, it’s compared with the reference genome to detect genetic variants, such as SNPs (single base pairs) or indels (insertion/deletion). These variants can play a role in determining susceptibility to disease or other characteristics.
8. **Interpretation:** The last step is to interpret the genomic information in the light of the relevant research or clinical inquiries. This could be the identification of disease-promoting mutations, genetic determinants of characteristics, or personalized medical therapies based on a patient’s genetic makeup.

2.2 What does the data generated look like?

Genome sequencing data is a large amount of non-structured genetic data in DNA sequences. Genomic data is usually expressed as a sequence of nucleotide base pairs (NBS). Each base pair is represented by one or more of the four letters A, T, C, or G (adenine, thymidine, cytosine, and guanine respectively). The sequence data is written as a long sequence of these letters. The length of the sequence data is determined by the sequencing technology and the genome size being sequenced.

The raw data may be vast, comprising billions of nucleotide base pairs, as it encapsulates the entirety of the genome of the sequenced organism. For instance, the entire human genome is composed of more than 3 billion nucleotide base pairs. Due to the vastness of the raw data, digital files are often used to store it, and file sizes can range from gigabytes to terabytes.

2.3 What is the specific data that has been given for this project?

The data given for the project is as following

2.3.1. chrX.fa

The reference sequence for chromosome x in fasta format. First line is the header line followed by the sequence (ignoring the newlines).

2.3.2. chrX_last_col.txt

The last column of the BWT. This sequence contains one more character than the reference sequence as it also contains the special character $ which is appended to its end.

2.3.3. chrX_map.txt

Contains mapping of indexes in BWT with index in reference. Line number i (0-based) has the starting index in the reference of the ith sorted circular shift
eg. First line containing 3435 means that the string starting at 3435 (0-based) is the first in sort order of BWT rotated strings.

2.3.4. reads

Contains about 3 million reads, one read per line, reads are roughly of length 100, but may not be exactly so. And also each read could come from the reference sequence or its reverse complement.

2.4 What are the sizes of each of the data elements?

File	Number of lines(in Millions)	Number of chars(in Millions)	Size of the (in MBs)
chrX.fa	1.511007	151.10056	152.6
chrX_last_col.txt	1.511007	151.100561	152.6
chrX_map.txt	151.100561	-	1400
reads	3.06672	308.449838	311.5

Table1: *Number of lines, character and size for each of the data elements*

3. Read Alignment

3.1 Suffix Tree and associated algorithm

Suffix tree is a powerful data structure used in string processing. It represents all suffixes of a string in compressed form. This allows for fast and efficient substring search, string comparison, and other string operations.

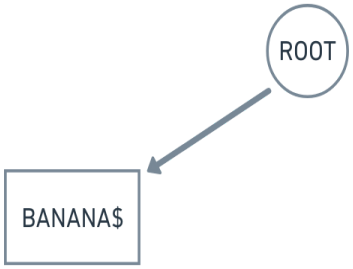
For example, a string S with length n is represented by a suffix tree. Each edge of the suffix tree represents an S substring, and the way from the root node to the leaf node represents an S suffix. The structure of the suffix tree is similar to a trie data structure, with no branches containing only one character.

3.1.1 Construction of suffix tree

Let's construct a suffix tree for the string "BANANA".

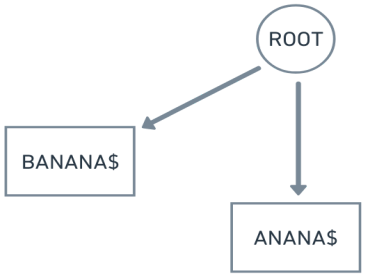
Step 1: Initialize by creating a new empty tree containing only the root node named 'R'.

Step 2: Adding the First Suffix "BANANA$".

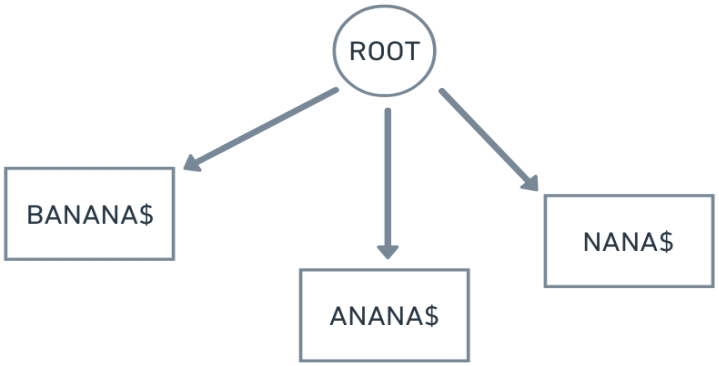


Step 3: Adding the Second Suffix "ANANA$".

Now, add the second suffix of the string "BANANA$", which is "ANANA$".

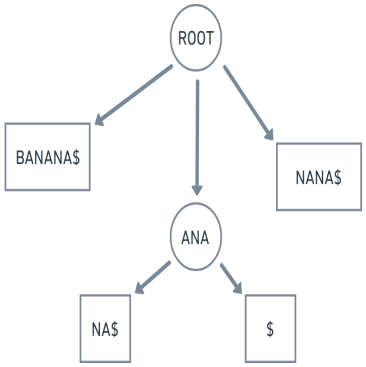


Step 4: Adding the Third Suffix "NANA$".



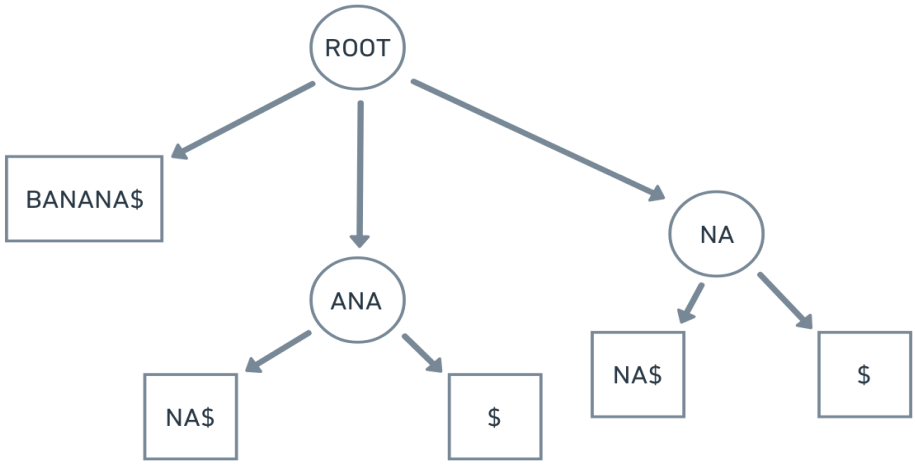
Step 5: Adding the Fourth Suffix "ANA$"

Add the fourth suffix of the string "BANANA$", which is "ANA$".



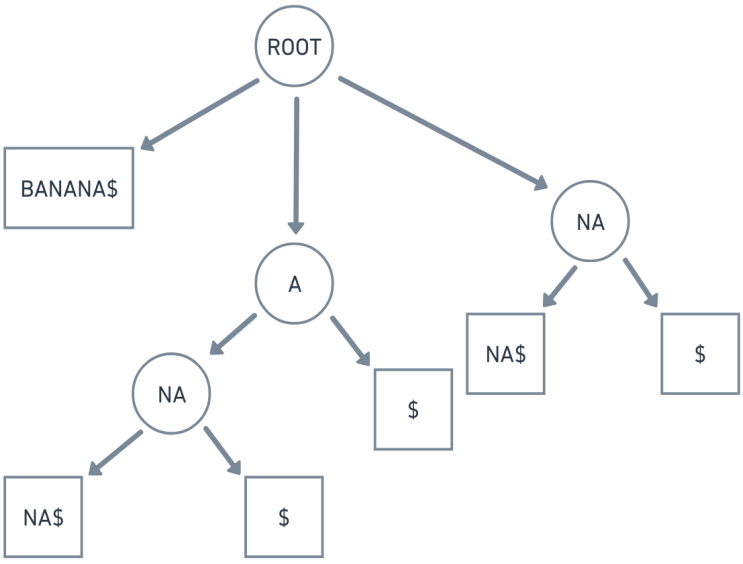
Step 6: Adding the Fifth Suffix “NA$”.

Add the fifth suffix of the string “BANANA$”, which is “NA$”.



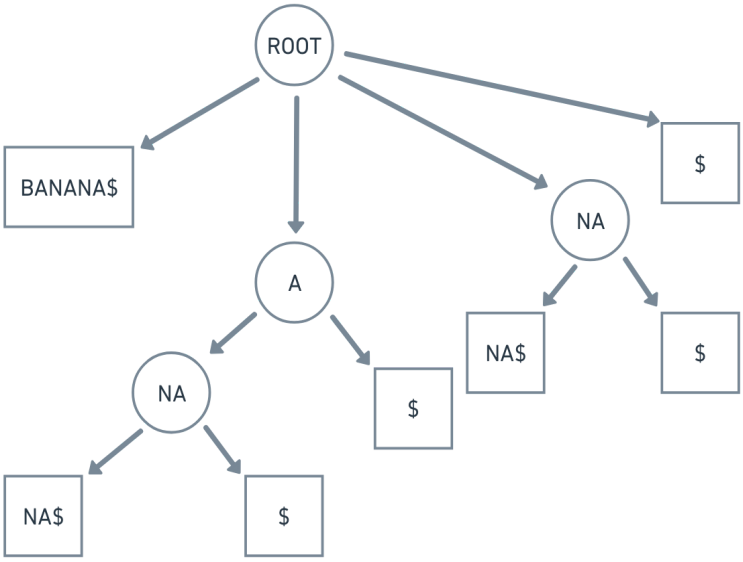
Step 7: Adding Sixth Suffix “A$”.

Add the sixth suffix of the string “BANANA$”, which is “A$”.

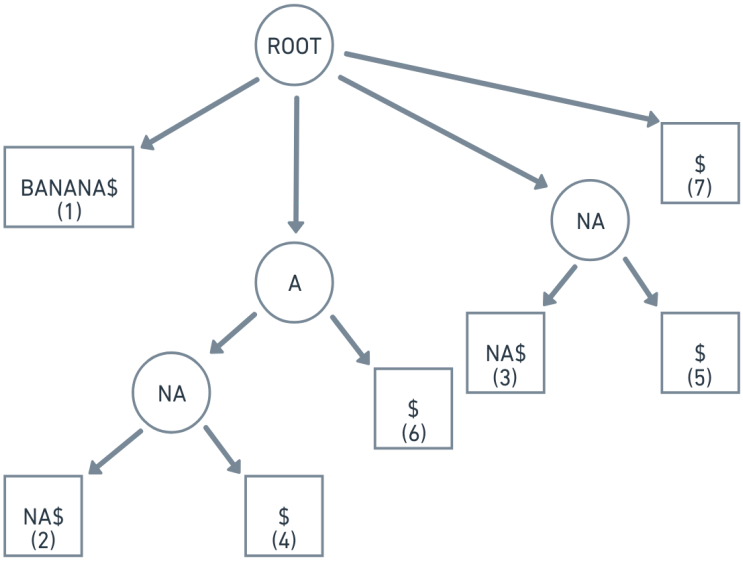


Step 8: Adding the Seventh Suffix "$".

Add the seventh suffix of string "BANANA$", which is "$".



Step 9: indexing all the leaf nodes.



3.1.2 Pattern matching using Suffix tree

Starting from the first character of the pattern and of the suffix Tree, do the following for every character

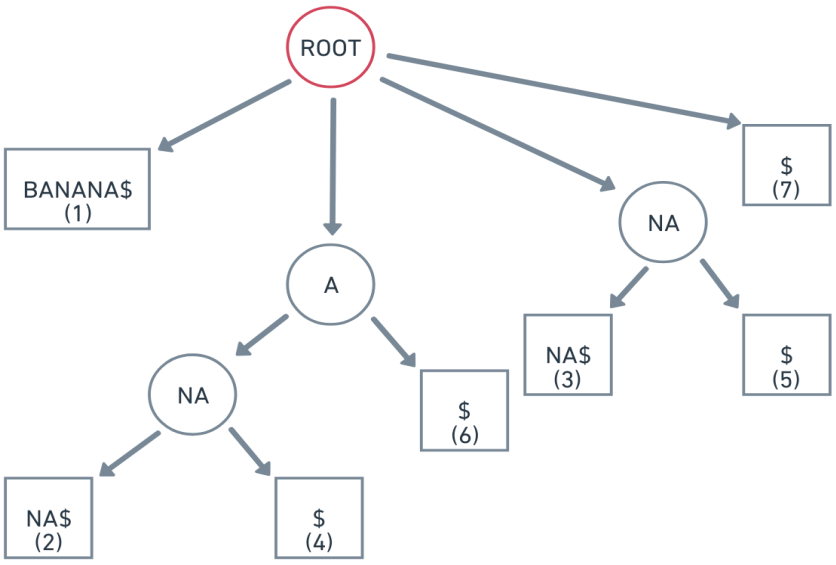
a) For the current character of the pattern, if there is an edge from the current node of the suffix tree, follow the edge.

b) If there is no edge, then the pattern doesn't exist and returns.

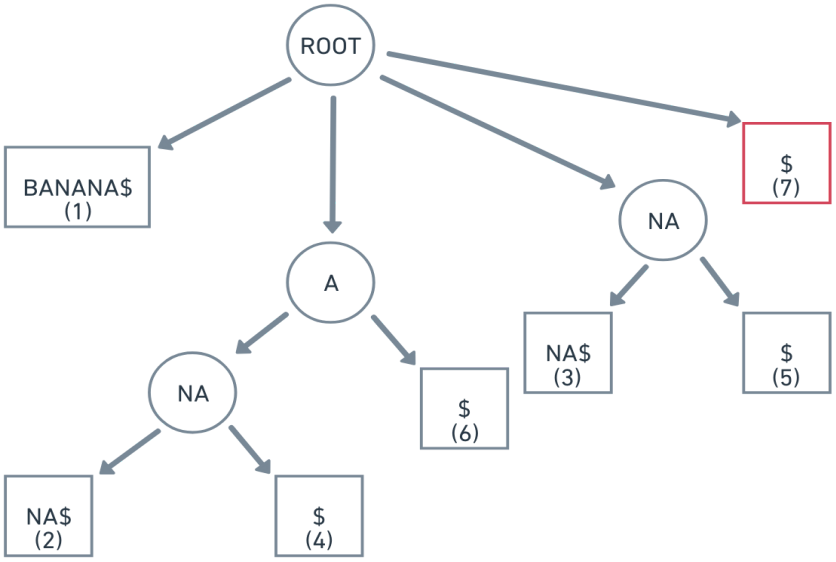
If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then pattern is found.

For example: let's us search "NAN"

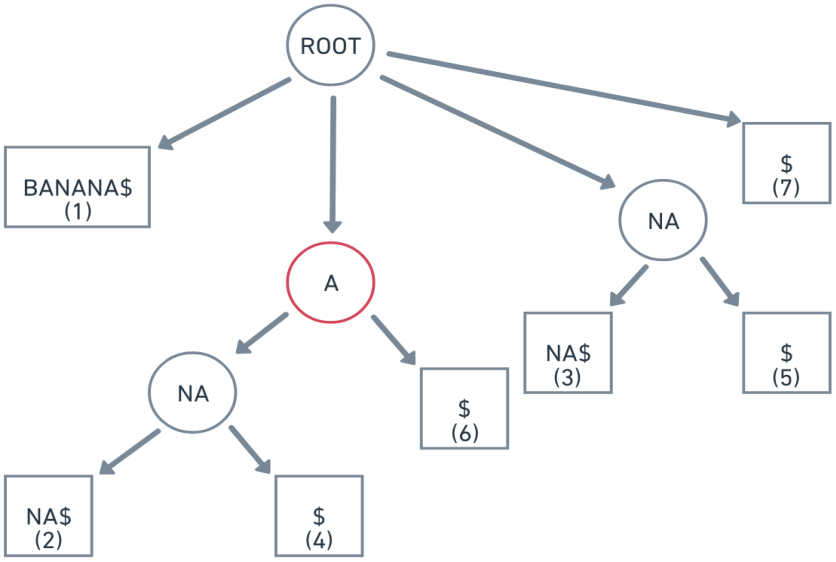
Consider the root node



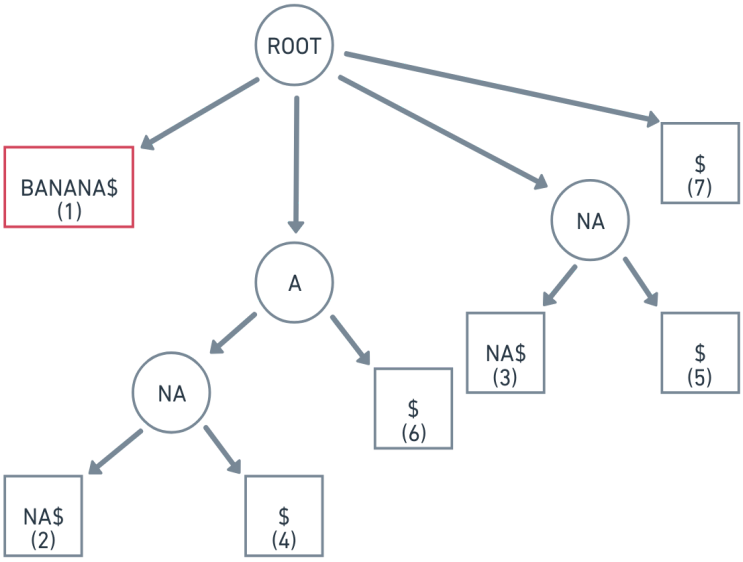
Path: $ No match found, moving back



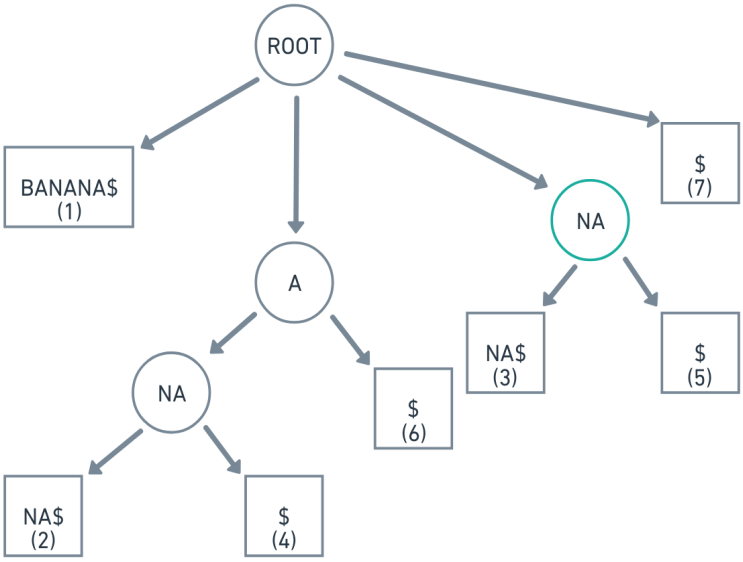
Path: A No match found, moving back



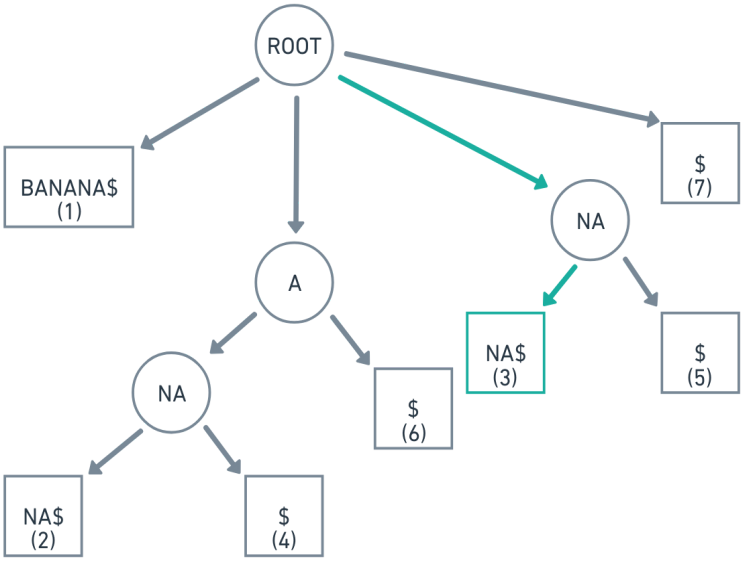
Path: BANANA$ No match found, moving back



Path: NA partial match found, searching deeper



Path: NANA$ pattern found at 3rd index



Suffix trees are essential for the performance of read alignment, a critical step in the bioinformatics process of determining the position of a short DNA sequence fragment (read) within a larger reference genome. During read alignment, the reference genome's suffix tree is constructed and each read is compared to this tree to determine its position within the genome.

The operation of read alignment utilizing a suffix tree entails the following:

1. **Construct the Suffix Tree:** The reference genome suffix tree is constructed using Ukkonen's algorithm or other powerful suffix tree construction algorithms.
2. **Match Reads to the Suffix Tree:** Each read is compared to the suffix tree. We start at the root and follow the read's sequence along the edges of the tree. If we find a match, the read will match a substring in our reference genome.
3. **Handle Mismatches and Gaps:** Read alignment typically involves mismatches (nucleotide differences between the read and reference) as well as gaps (insertion or deletion). To address these variations and enhance alignment accuracy, additional algorithms are employed, including the Smith–Waterman algorithm.

4. **Report Alignment Results:** Once a read has been compared to a reference genome, it is reported along with the alignment position as well as any discrepancies or gaps.

3.1.3 Time & Space complexity analysis

Let's analyze the time and space complexities of the read matching algorithm using the suffix tree construction described above.
assuming the average read length to be R, the number of reads to be N, and the size of the reference sequence to be M.

Time Complexity

1. **Suffix Tree Construction: O(M)** is the time complexity of building a suffix tree according to Ukkonen's algorithm which was used in the inbuilt library function also, for creating the suffix tree in our case, where M is the reference sequence length. This is due to the linear time complexity of building the suffix tree in relation to the input sequence length.
2. **Read Matching:** After constructing the suffix tree, the time complexity of matching each read to the suffix tree will be **O(R)** where R stands for the average length of read. The reason for this is that during the read matching process, we go through the nodes of the suffix until we have found a full match for our read or until we have reached the end of our read.

   Because there are only N reads to match against the reference series, the overall time complexity of matching all reads with the suffix tree can be defined as **O(M+ N*R)**.

Space Complexity

1. **Suffix Tree: O(M)** is the space complexity of constructing a suffix tree with Ukkonen's algorithm. A constructed suffix tree stores a compressed representation of all suffixes of a reference sequence.
2. **Additional Memory:** The memory usage of the algorithm during read matching is low and does not depend on reference size. The main use of memory during read matching is to store the current read position and traverse the suffix tree with pointers. Each read takes up **O(R)**,.

Given the number of reads to match, the overall memory complexity of the suffix tree-based read matching algorithm is **O(M + N * R)**.

To sum up, read matching algorithm based on suffix tree construction has a time complexity of **O(M + N * R)** and a space complexity of **O(M + N * R)**. The efficiency of the algorithm allows it to match large numbers of reads against the reference genome. This is especially useful in applications such as DNA sequence alignment where read alignment plays a critical role in detecting genetic variations and analyzing genomic data.

3.1.4 How many reads get aligned?

In Our case the
R ≃ 101 (101.58308420723118)
N = 3066720
M = 51974
And following were the results

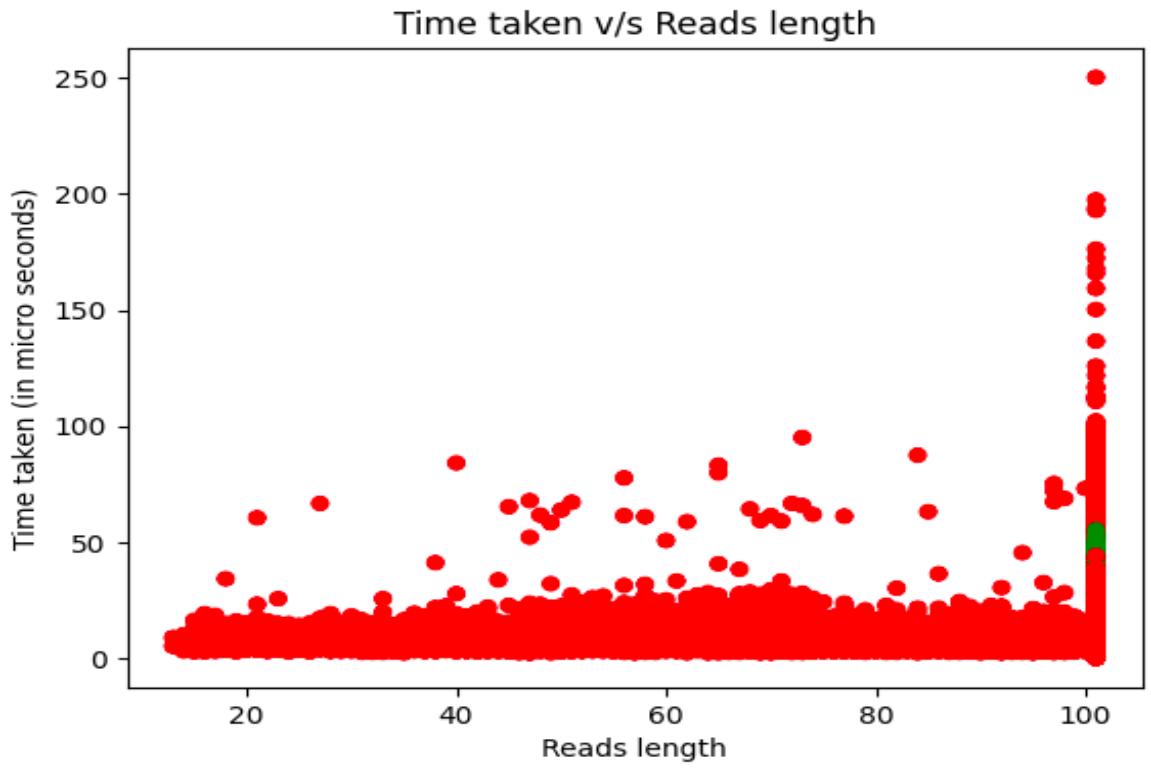


Fig2: *Showing the plot of reads length v/s the time taken for suffix tree to match the reads, the red points depicts the reads that doesn't matched while the green points represent the matched one*

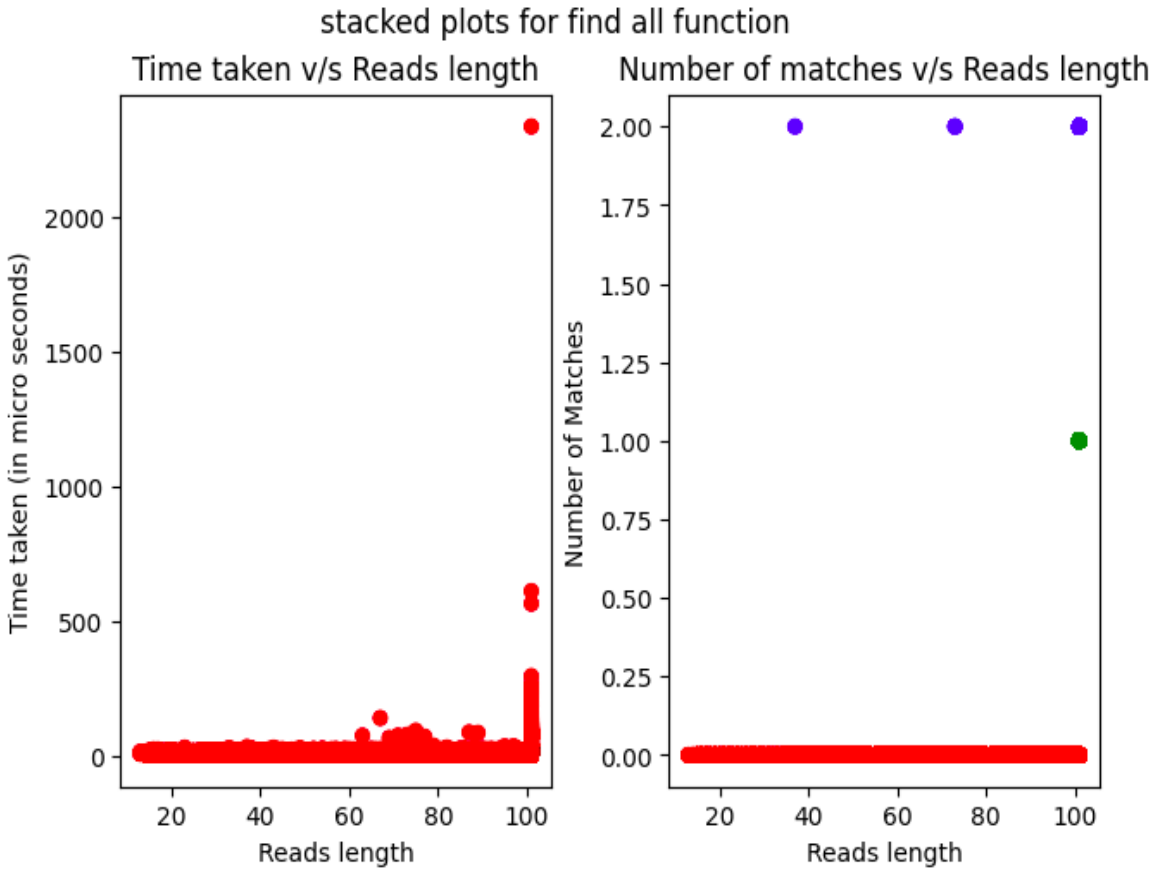


Fig3: *left plots shows the time taken by suffix tree to find all the occurrence of reads against the read length, the right plots shows the number of times a read matches v/s the read length where red, green and blue mean 0, 1 and 2 times matched respectively*

- Number of reads that matched matched 0 times: 3059853 (99.78%)
- Number of reads that matched 1 times: 5138 (0.17%)
- Number of reads that matched 2 times: 1729 (0.06%)

3.2 BWT and associated algorithm

Burrows-Wheeler transform (BWT) refers to a reversible data transformation method. It is mainly used for data compression and bioinformatics data indexing, including read alignment for DNA sequence analysis. The BWT was first introduced in 1994 by Michael Burrows in collaboration with David Wheeler.

The burrows-wheeler transform takes a string and rearranges it into a new string that has better compression properties. The burrows wheeler transform of a string is is constructed as follows:

1) Generate all possible circular rotations from the initial string.
2) Sort these rotations in a lexical manner.
3) Extract the last character of every sorted rotation to create a BWT string.

A string's BWT keeps some patterns and runs of the same characters, which makes it more compact. It also has properties like local grouping, which means similar characters tend to cluster together in a BWT string.

In bioinformatics, the BWT is commonly used in read alignment and indexation. When used in read alignment, the BWT helps in the generation of a data structure called a “Burrows-wheeler Aligner” index (BWA). The BWA index indexes the reference genome effectively, allowing for rapid alignment of short DNA sequence reads against the genome.

3.2.1 Construction of BWT string

Let's construct the Burrows-Wheeler Transform (BWT) for the string "BANANA"

Step 1: Appending the end character "$" at the end of the string “BANANA”. So the new string is “BANANA$”.

Step 2: Create all possible cyclic rotations of the new string "BANANA$".

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

Step 3: Sort these rotations lexicographically.

$	B	A	N	A	N	A
A	$	B	A	N	A	N
A	N	A	$	B	A	N
A	N	A	N	A	$	B
B	A	N	A	N	A	$
N	A	$	B	A	N	A
N	A	N	A	$	B	A

The Burrows-Wheeler Transform (BWT) for the string "BANANA" is: "ANNB$AA".

3.2.2 Pattern matching using BWT transform

In order to use the BWT of a string, we can create a data structure named the “index array” and perform rank and select operation on the index array. This index array, in combination with rank operation and select operation, allows us to perform fast and efficient BWT pattern matching.

As an example, let’s look at the BWT of the original “BANANA” string: BWT: “ANNB$AA”

Step 1: Building the Index Array An index array is a helper array that contains information about the position of certain characters(first character of each string in BWT transform) in a BWT string.

B	A	N	A	N	A	$
0	1	2	3	4	5	6

0	$	B	A	N	A	N	A
6	A	$	B	A	N	A	N
4	A	N	A	$	B	A	N
2	A	N	A	N	A	$	B
1	B	A	N	A	N	A	$
5	N	A	$	B	A	N	A
3	N	A	N	A	$	B	A

Step 2: Implementing Rank and Select Operations

- Rank(c, i): It returns the number of occurrences of character 'c' in the BWT string up to position i (exclusive).
- Select(c, k): It returns the position of the k-th occurrence of character 'c' in the BWT string.

Here are some rank and select operations for the BWT string “ANNB$AA” and its index array

Rank('A', 6) = 2
There is one occurrence of 'A' up to position 6 (exclusive).

Select('N', 1) = 2
The position of the first occurrence of 'N' is at position 2.

Select('$', 1) = 4
The position of the first occurrence of '$' is at position 4.

Step 3: In order to match a pattern in the initial string using BWT, we begin from the last character in the pattern and consider all the string positions of BWT in our initial band.
That is, the initial band is, [1, n] where n = length of original string + 1.
Step 4: Then we find the rank of the current character of the pattern at the starting and ending band position. Let's say the obtained ranks to be r1 and r2 respectively.
Step 5: Then we apply selection operations using the current character of the pattern, r1 and r2 let the obtained position after applying select operation by p1 and p2.
Step 6: So our new band is reduced to [p1, p2] and we move to the next character in our pattern.

And follow the same steps again for the next character(moving from the end to the beginning) until the pattern is exhausted or band size is reduced to zero(in case when the pattern doesn't match).

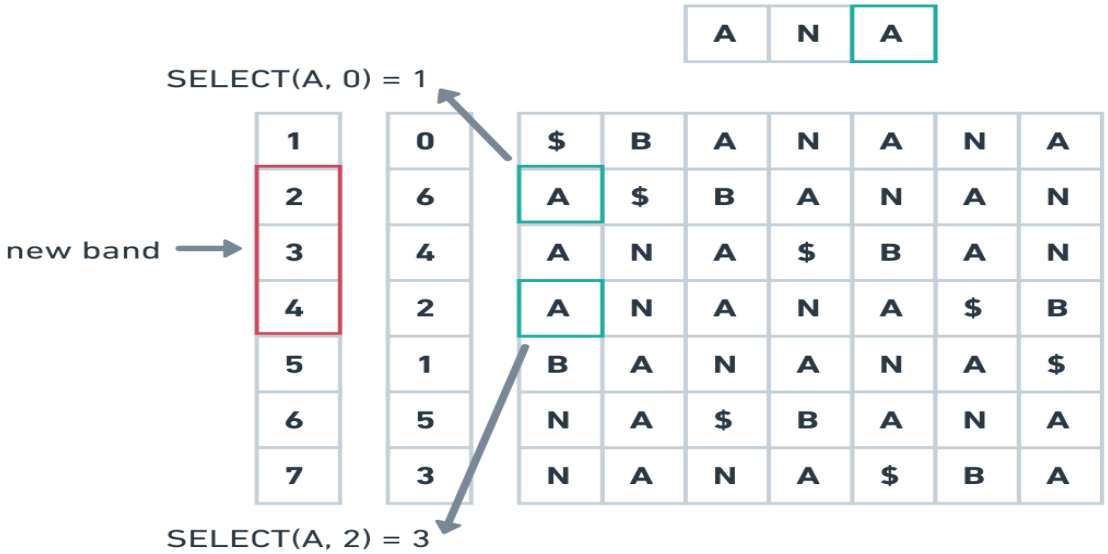
For instance, let's say we want to find the occurrence of the pattern "ANA" in the original string "BANANA" using its BWT "ANNB$AA".

1) Initial band: [1, 7]
Current char in pattern: A

A	N	A

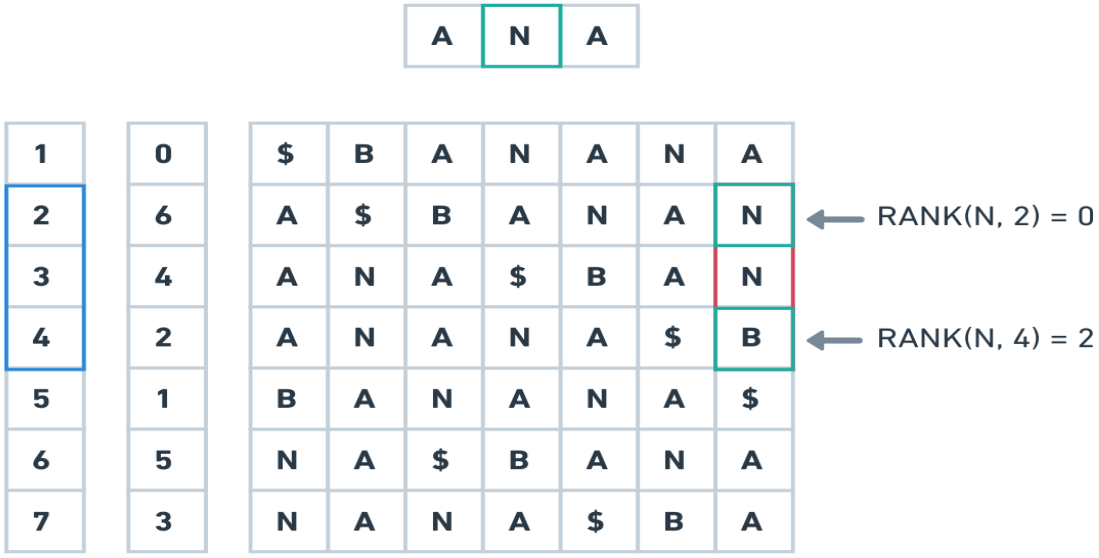
1	0	$	B	A	N	A	N	A ← RANK(A, 1) = 0
2	6	A	$	B	A	N	A	N
3	4	A	N	A	$	B	A	N
4	2	A	N	A	N	A	$	B
5	1	B	A	N	A	N	A	$
6	5	N	A	$	B	A	N	A
7	3	N	A	N	A	$	B	A ← RANK(A, 7) = 2

2) New band = [2,4]



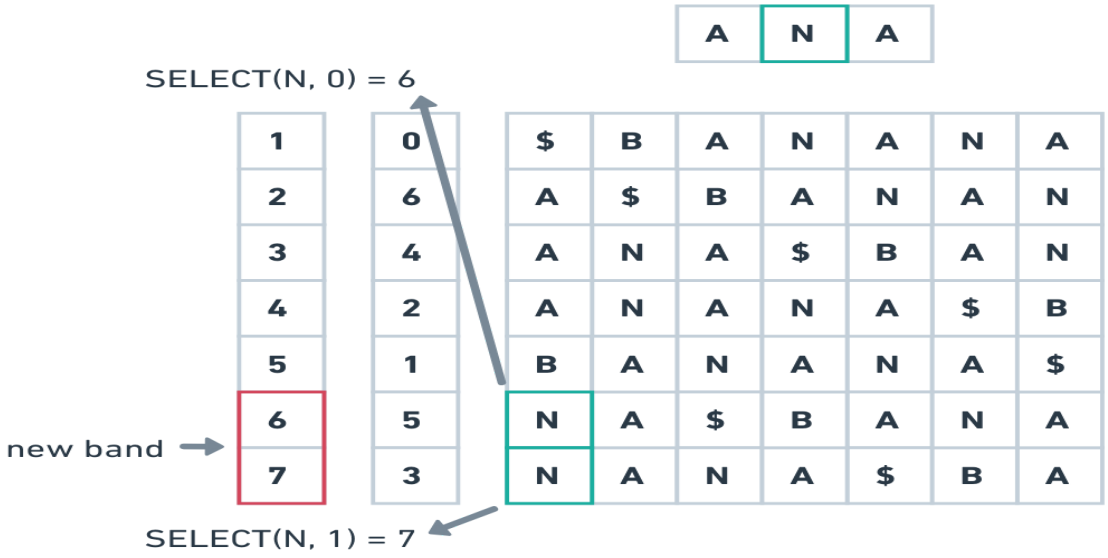
3) Current band: [2, 4]

Current character in pattern: N



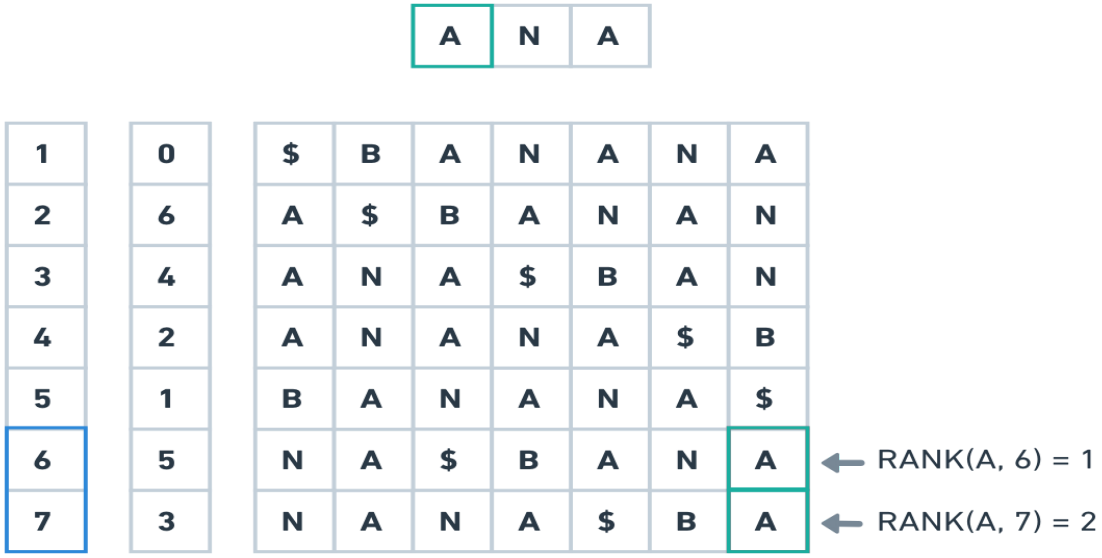
Note: *here since the last band character(B) is not equal to the current pattern character (N) so we need to subtract 1 from the last band rank(2) in order to get the correct new band.*

4) New band = [6, 7]

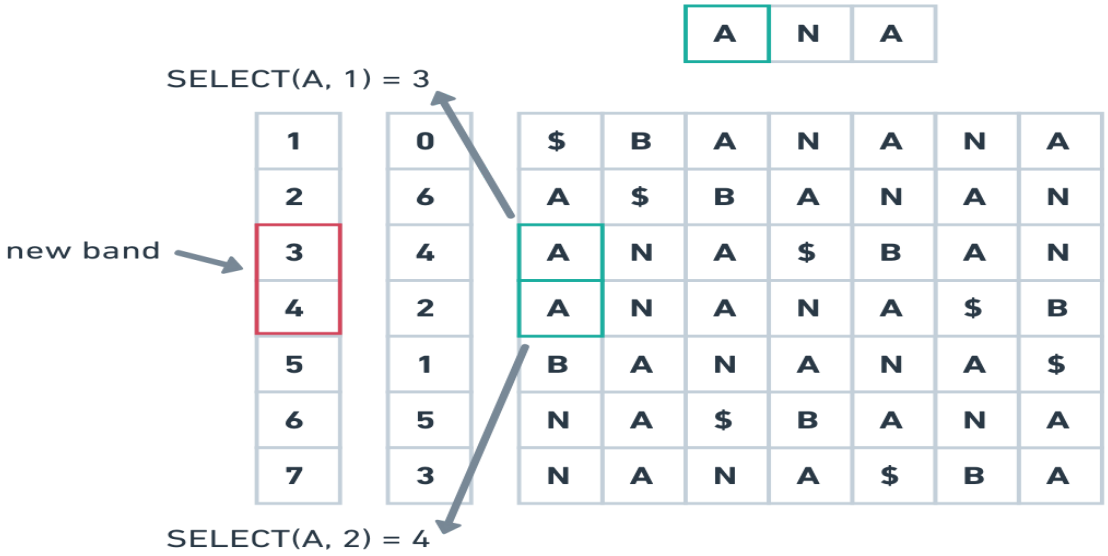


5) Current band: [6, 7]

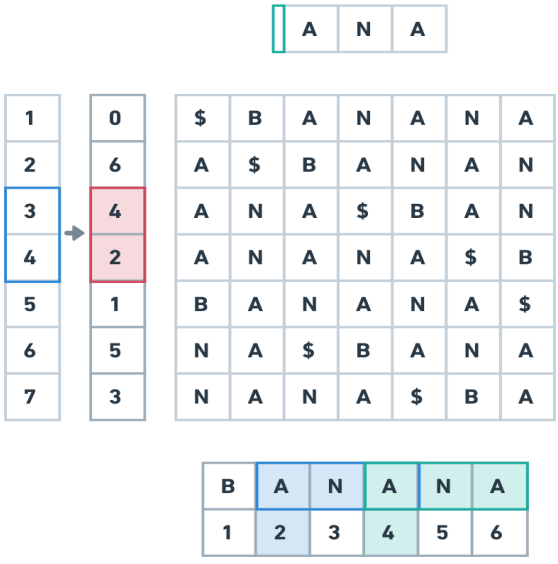
Current pattern character: A



6) New band: [3, 4]



7) Since the string has exhausted completely so the corresponding values in the index array corresponding to the current band will be the final position of at which the string “ANA” is matching in the original string “BANANA” That is [2, 4] and which is indeed the case.



3.2.3 Time & Space complexity analysis

The time and space complexity of the pattern matching algorithm using the Burrows-Wheeler Transform (BWT) depends on the length of the pattern and the size of the BWT string. And some other factors also play a crucial role in determining the time/space complexity of the pattern matching algorithm using BWT. Let's analyze each aspect separately.

Here the assumption is made that the string for which pattern matching algorithm is used is only a genomic string. That is, it contains a random sequence of only 4 characters(A, C, G, T). but in general the algorithm works fine for any other string as well, it's just that right now we are interested in the genomic strings and the algorithm also gives some space complexity advantage for genomic strings only. So that's why the complete analysis will be done considering that the original string is a genomic string for which we will be implementing the pattern matching algorithm.

Another assumption that is made is that, we are already provided with the BWT string and the index array for the BWT string, anyways this can be efficiently found using linear time algorithm hence not considering this initially and making the BWT transform and it's index array from scratch won't contribute much to the time complexity.

And the last assumption is that, the length of the original string is N, and length of the pattern is M, $delta_1$ is the time optimization milestone for the rank operation and $delta_2$ is memory optimization milestone for the find index operation.

Time Complexity

1. **Construction of the rank array:** we need to iterate in the whole BWT string for each distinct character of the original string, which is 4. Hence for this operation the time complexity will be O(4*N) or **O(N)**.
2. **Populating the Bit-array:** for this operation also, we need to iterate in the whole BWT string for each distinct character in the original string, hence time complexity for this operation will be **O(N)**.

3. **Compressing the Index/map array:** need to store only the index of beginning character at each delta_2 position in first column of BWT array, the index array/map array is already available to us so we just need to iterate in that and store the index, which would be **O(N)**, as size of map array is equal to N + 1.
4. **Iterating in the pattern:** while iterating in the pattern for each character we need to perform the 2 rank & 2 select operation, time complexity for rank operation is **O(delta$_1$)** and that for select is **O(1)** so the total time complexity will be **O(M * delta$_1$)**.
5. **Finding the index:** at last we need to find the original index of the character of the strings that are in the final band. So a pattern can match at (N - M + 1) position and for each position we need find the index in the space optimized index array having the time complexity **O(delta$_2$)** so the overall time complexity for this operation would be O((N - M + 1)*delta$_2$). Since size of pattern is really small in comparison to the size of the original string for all practical purpose so this can be further approximated to **O(N * delta$_2$)**.

So the overall time complexity will be **O(M * delta$_1$ + N * delta$_2$)**

Space Complexity

1. **Construction of the rank array:** for each character of the original string we need to store the rank of the every delta$_1$ position in BWT string, so the space complexity would be O($\frac{4*N}{delta1}$) or **O($\frac{N}{delta1}$)**.
2. **Populating the Bit-array:** for each character of the original string we need to create bit array of the length of BWT string, so the space complexity would be O(4*N) bits or **O($\frac{N}{2}$)** bytes.
3. **Compressing the Index/map array:** for this operation we are only storing the index corresponding to each delta$_2$ position of the BWT string, so space consumed will be **O($\frac{N}{delta2}$)**.
4. **Storing the count of each character:** Since there are only 4 characters so the space complexity will be **O(1)**.

Hence the overall space complexity will be **O(N * ($\frac{1}{delta1}$ + $\frac{1}{delta2}$ + $\frac{1}{2}$))**

Note: Here the factor of ½ is crucial, since the particle value of $delta_1$ and $delta_2$ are much larger than 2 so the fraction of ½ becomes important in deciding the overall space complexity.

3.2.4 variation of Time/memory consumed with Delta parameters

Since the time complexity was O(M * $delta_1$ + N * $delta_2$), the time consumed will be linearly proportional to both $delta_1$ and $delta_2$.

The space complexity was O(N * ($\frac{1}{delta1} + \frac{1}{delta2} + \frac{1}{2}$)), hence the memory consumed will be inversely proportional to both the $delta_1$ and $delta_2$.

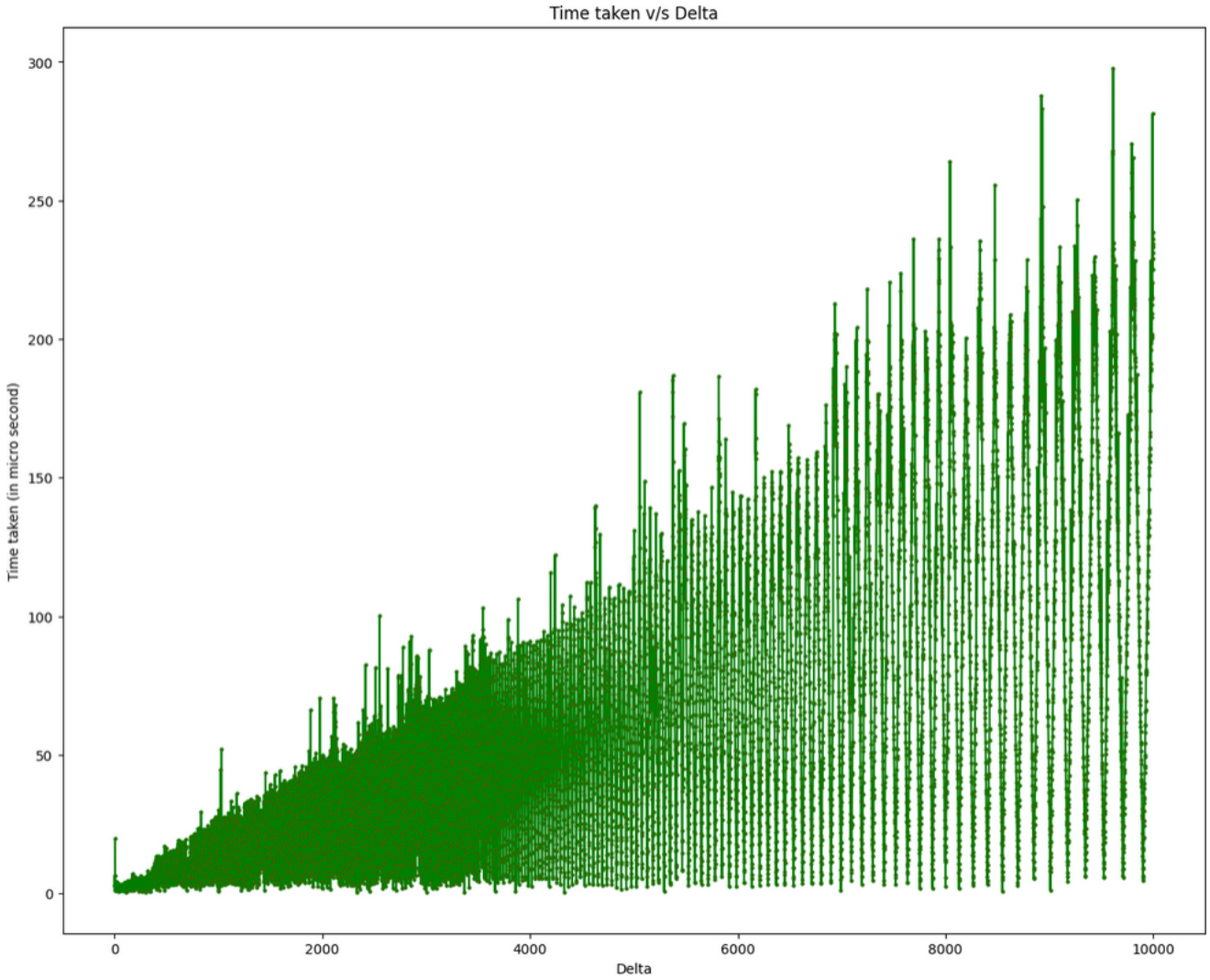


Fig3: *variation of time consumed by rank operation with* $delta_1$

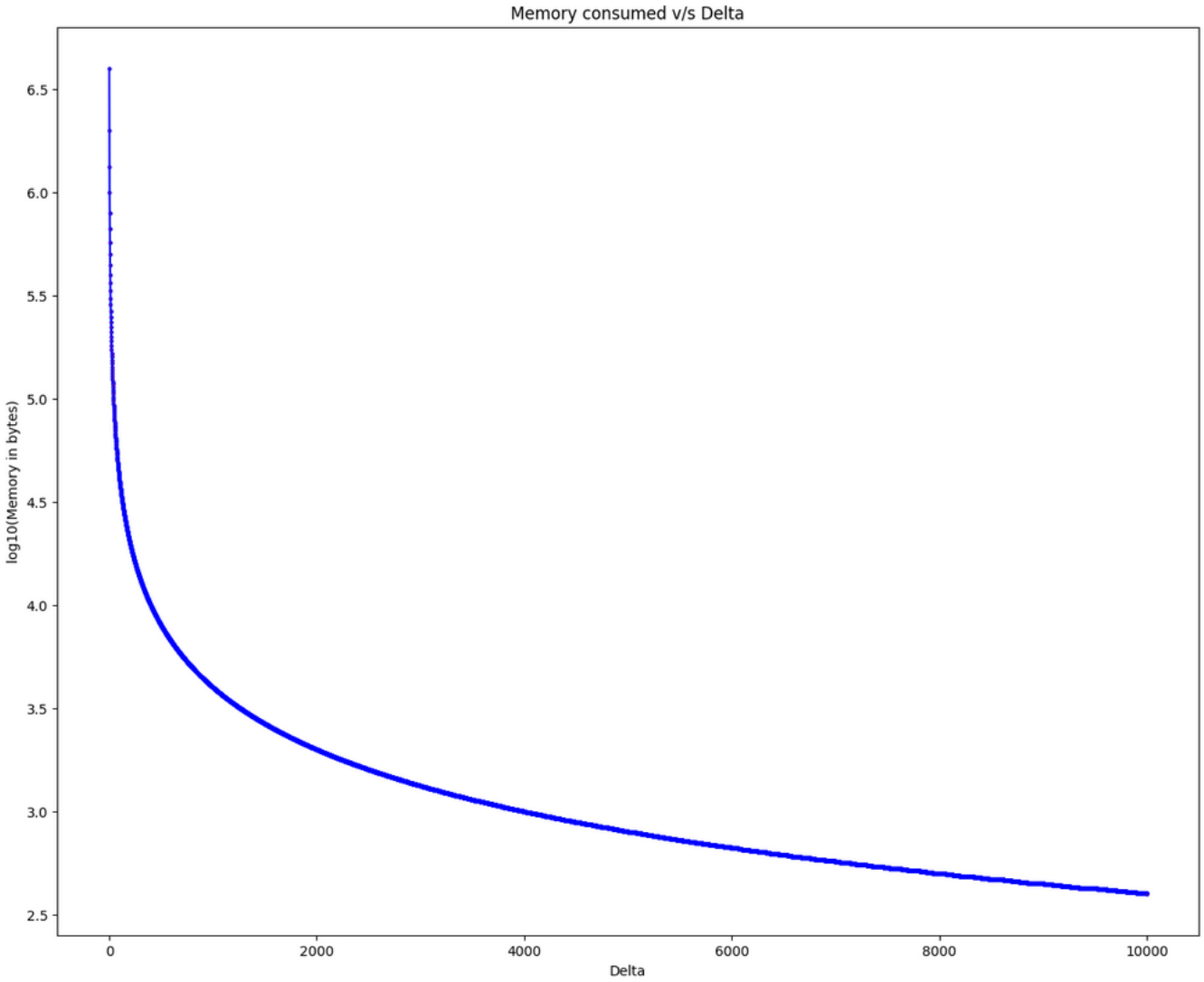


Fig4: *variation of memory consumed for finding rank (efficiently) with delta$_1$*

3.2.5 How many reads get aligned?

- Number of reads that matched matched 0 times: 3059853 (99.78%)
- Number of reads that matched 1 times: 5138 (0.17%)
- Number of reads that matched 2 times: 1729 (0.06%)

4. Solving the Color Blindness Problem

4.1 Theoretical calculation

For theoretical calculations various things were tried but the best results which were showing were coming when the following heuristic is adopted, but before going over the heuristics let's see some key concepts that were used in the heuristic in order to understand them.

4.1.1 Approx pattern matching

We created a function which will match a pattern to the string with at-most K miss-matches. For matching a pattern approximately the function uses the following algorithm:

Step 1: Divide the pattern in K + 1 parts, so that at least 1 part will exist which will match exactly.
Step 2: Now let's say that out of those K + 1 parts, m parts match exactly. And each of those m parts matches at p[i] number of positions.

So for each (part, p[i]) perform the linear search in the original string and compare the remaining part of the pattern other than the matched part, and account for at-most K mismatches

So following were the settings for heuristic that gives us the best result:
1) take the red green exon region with 100 padding at the start of the 1st red exon and 100 padding at the end of the 6th green exon. That is, the reference sequence that we are using is only limited to the red-green exon region with some padding and having length 51,974. we are not taking the whole reference genome as our original string as we are only interested in reads matching the red and green exon region only, so this approach both saves us on time and memory

2) after extracting the padded red green exon region, match all reads with following set of rules

- **2.1)** if the read doesn't contain N then match the read with approx miss-match function having the value of allowed miss-matches passed to be 1 only.
- **2.2)** if the read contains N then, let's x be the number of N's that read contain, then match the read with approx miss-match function having the value of allowed miss matches passed to be (x + 1).
- **2.3)** after checking all positions of approx miss match for a read, allow a read to match a particular region only if the percentage of read length that is matching in that region is greater than equal to the threshold percentage **(25%)**.
- **2.4)** if a read matches in both green and red don't consider that read.

3) bothe the delta value (1 & 2) were taken to be 100 for the BWT pattern matching algorithm, but the choice of deltas is immaterial here as we have seen earlier that both the deltas only

contribute to time/space complexity of the algorithm they don't have any impact on the correctness of the result. In general the higher the value of delta you take the more time it will take for the algorithm to produce results but on the other hand less will be the memory consumed by it. So considering these things in mind the choice of deltas should be made as per convenience.

So after following this heuristic the final result which we got was this:

Exon region	Percentage of reads matching in red to the green region
1	37.78
2	29.66
3	27.08
4	117.78
5	74.8
6	N/A

Table2: *percentage of reads matching in the red exon region to the same green exon region as per the heuristic described earlier*

4.2 What is the data telling us?

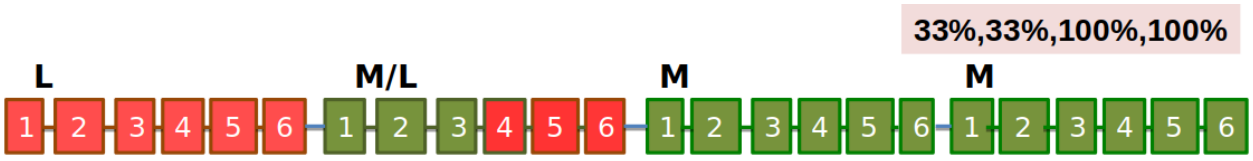


Fig5: *Final mosaic of genome formed after lopsided cut happened at the 3rd exon of red and green gene*

Out of all the possible way of chromosome mixing , this mosaic closely matches to the data that we have got, which tells the way in which the lopsided cut happened between the 2 x-chromosomes is depicted in the following figure

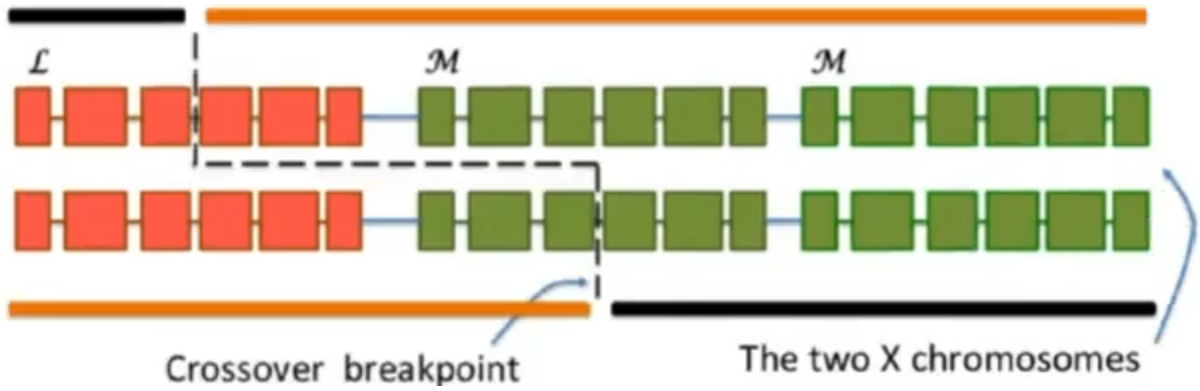


Fig6: *How the 3rd red and green exon region are cutted to form the new gene mosaic*

Since half is of the gene is green and half is of the gene is red in the new chromosome
And only the first two are getting executed. That is only the first two are getting converted to the sensors. Also this sensor will be a mix of both red and green genes, typically we have found a full red exon followed by a full green exon. So due to these reasons the peaks of the wavelength of red and green color will come closer and the person will not be able to differentiate between red and green and will require more contrast.

4.3 What do we Suspect is the cause?

Since the sequence-like, similarity between L and M is high and there are only 15 differences between both the genes so these two regions would be considered similar hence such a cut can happen.
There might be a match other places also but then it is completely random on how this cut is made

5. Link to the code