Burrows-Wheeler Transform (BWT)

Agenda

- Compression
- Building the BWT string
- BWT Inversion
  - Naive
  - LF-Mapping
- Substring search with BWT
  - Search with trie : Space of trie O(N^2)
  - Search with compressed string BWT : Space compressed < O(N)
  - Awesome to use if we're dealing with compressed large strings and wanna search for a pattern in the original string (but remember the string is now in compressed form).

To achieve lossless compression

Run-length encoding

- Simple lossless data compression.

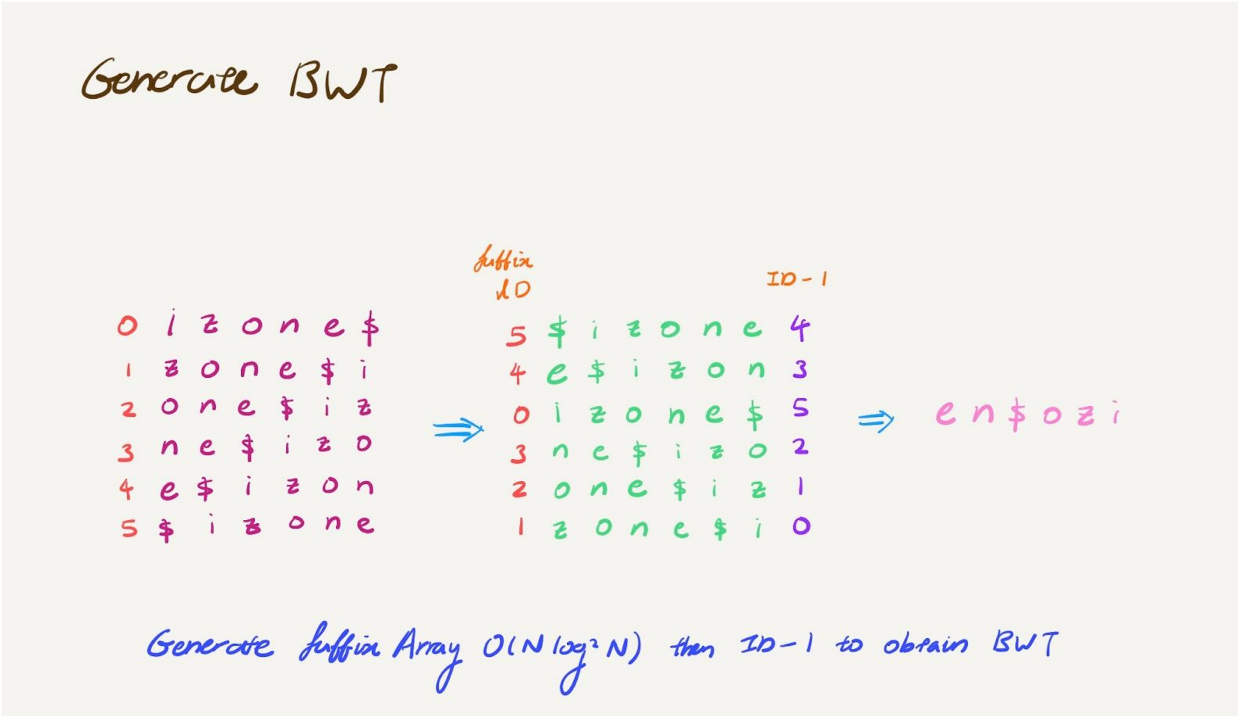
So the compression that we want

- Make the original text smaller.
- We can recover the original string when we decompress!

What is BWT?

- BWT is gonna move / group similar alphabets closer.
  - That is why we can use it for compression.
- We can rearrange back to the original string easily.
  - Without needing a key!

How to make the BWT string?



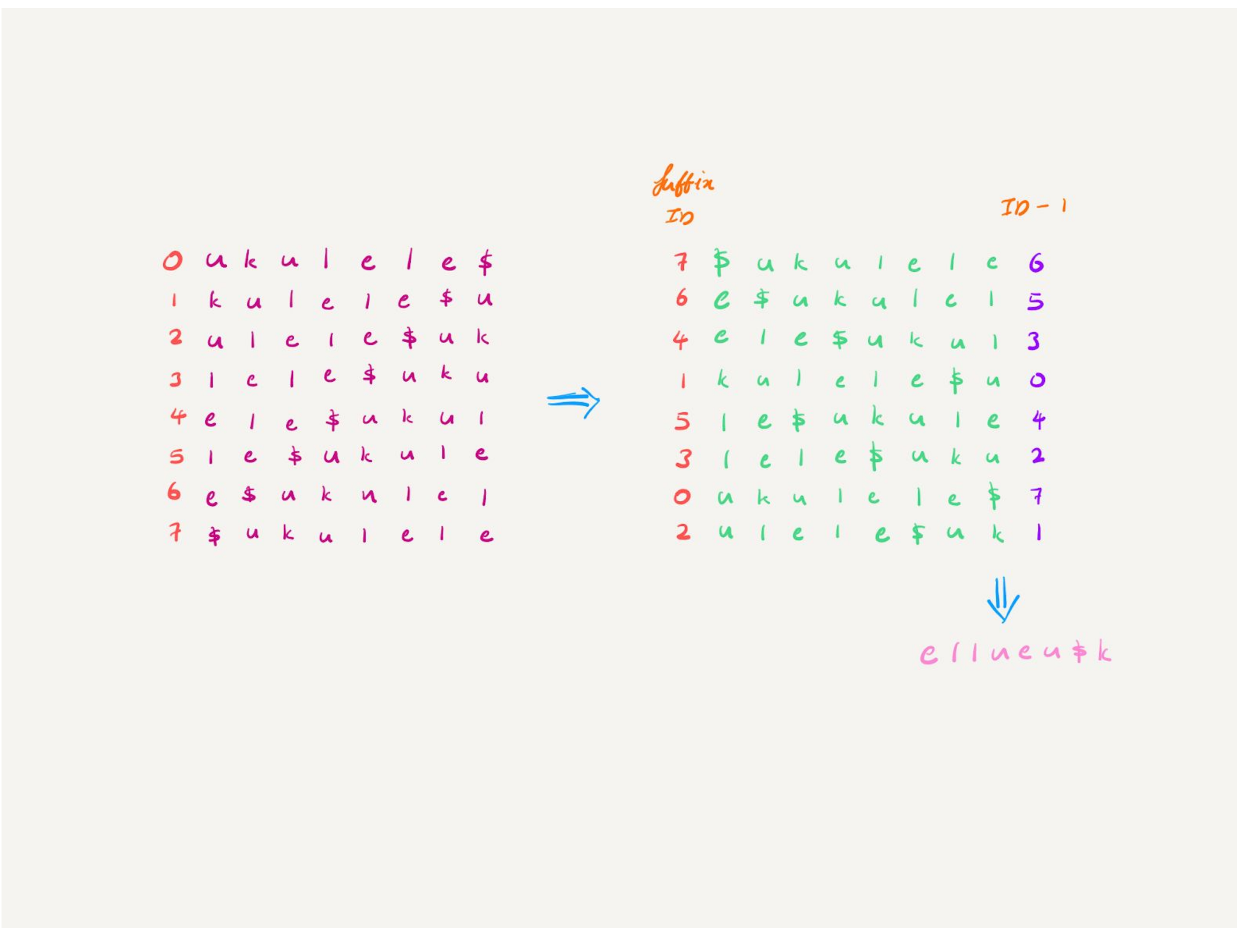
- Take a string, add ‘$’ behind.
- Generate all cyclic representations of a string.
    - text = “izone$”
        - izone$
        - zone$i
        - one$iz
        - ne$izo
        - e$izon
        - $izone
    - It’s clear that ‘i’ is followed by ‘z’, ‘z’ is followed by ‘o’ due the cyclic property.
    - Then when you sort it, similar alphabets will be grouped together (this is a not so good example).
- Time
    - O(N^2) time to generate N strings, each of N length.
- Space
    - O(N^2) space to store them all.

    - Use suffix arrays to reduce to O(N).
- Generate all cyclic representations of a string (Suffix ID).
- Sort the strings by using the suffix array, prefix doubling method O(N(log^2(N)). O(N) if using Ukkonen.
- Last column is the BWT string :)
    - bwt(text) = “en$ozi” (suffix ID - 1)

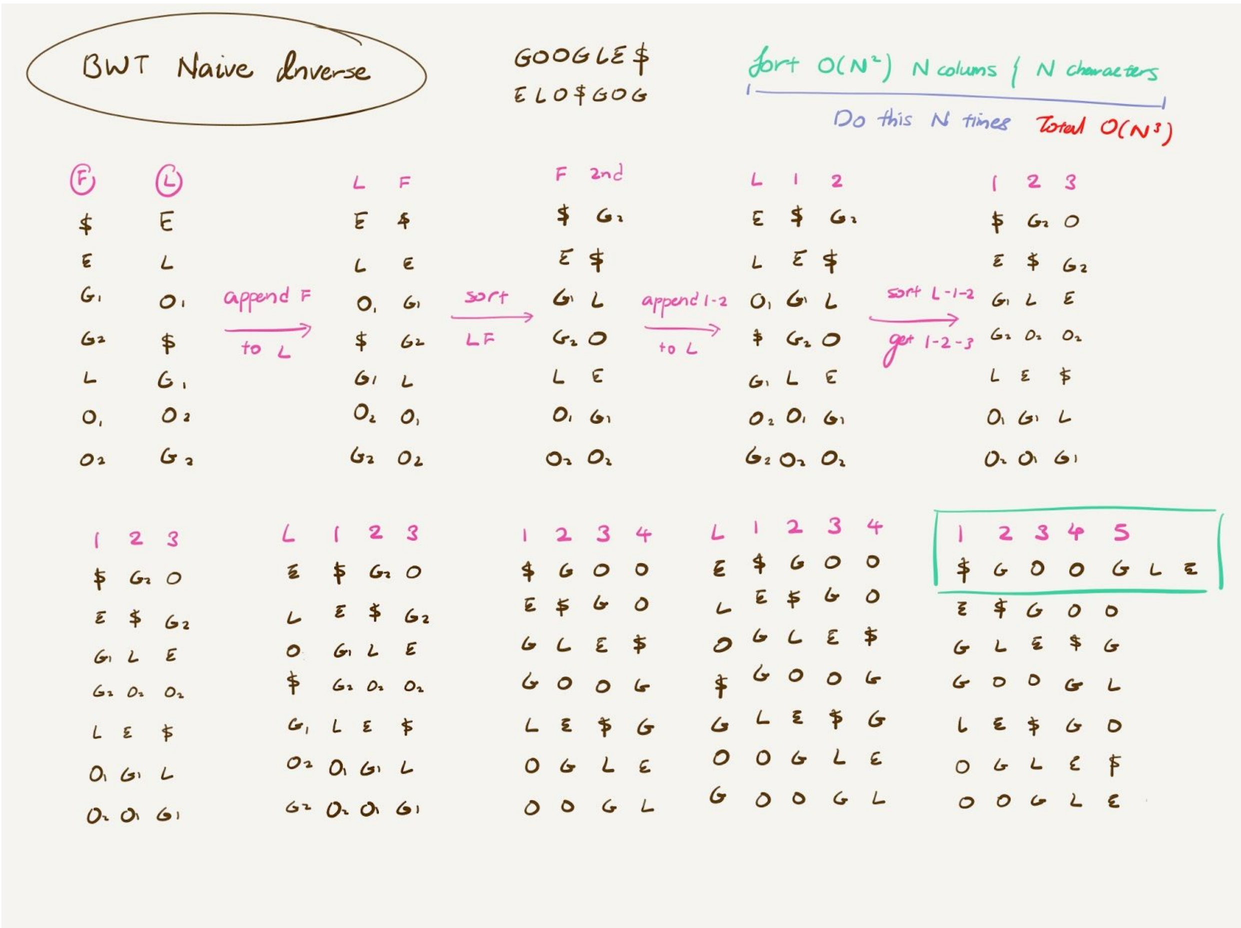
So why does this work?

- Due to the last and first relationship.
- Mainly due to the last-first property from the cyclic rotations.
- Once we sort it, the last column will group similar letters together (so in a large english text, common words will stick together).

Example #2



Invert BWT



- bwt(text) -> bwt_text ->bwt^-1(bwt_text) = text
- Given “ELO$GOG”
- I know this is the Last column.
- How do I produce the First column?
    - Just sort the alphabets in your Last column.
- L :- ELO$GOG
- F :- $EGGLOO (sort Last column using counting sort O(N+M), M = 27 Characters)
- So how to obtain the rest of the columns?
    - Since it’s a cyclic relationship, E is to be followed by $, L and E so on and so forth.
- Concatenate the Last and First column. (Append the First column to the Last column). Then sort it.

- From Last-First, we get First-Second.
    - Think of us appending the rest of the columns to the last column, then everytime we sort the last column we are gonna get the first.
- Append First-Second to Last, then sort it to obtain First-Second-Third.
- ...
- Then the first row will be the original string.

So what happened?

- So once I sort the last column, I get the first "sorted" column (First column is always in order).
- So once I sort the Last and the First, my Last column will be sorted to be the First column.
- Then the First column will be the Second column.
- Concatenate Last-First-Second, sort Last-First-Second, First-Second-Third.
- So we are making use of the K-mers of GOOGLE$' is GO, OO, OG, GL, LE, E$, $G.
- So what is the complexity? (Sort N columns, each column has a maximum of N characters. Repeat N times.) So the entire operation is O(N^3).
    - Radix sort is O(N^2), we do this N times and concatenate all of them.
- O(N^2) space to store all of the string.
- We can't use the suffix array because we don't know the actual word.

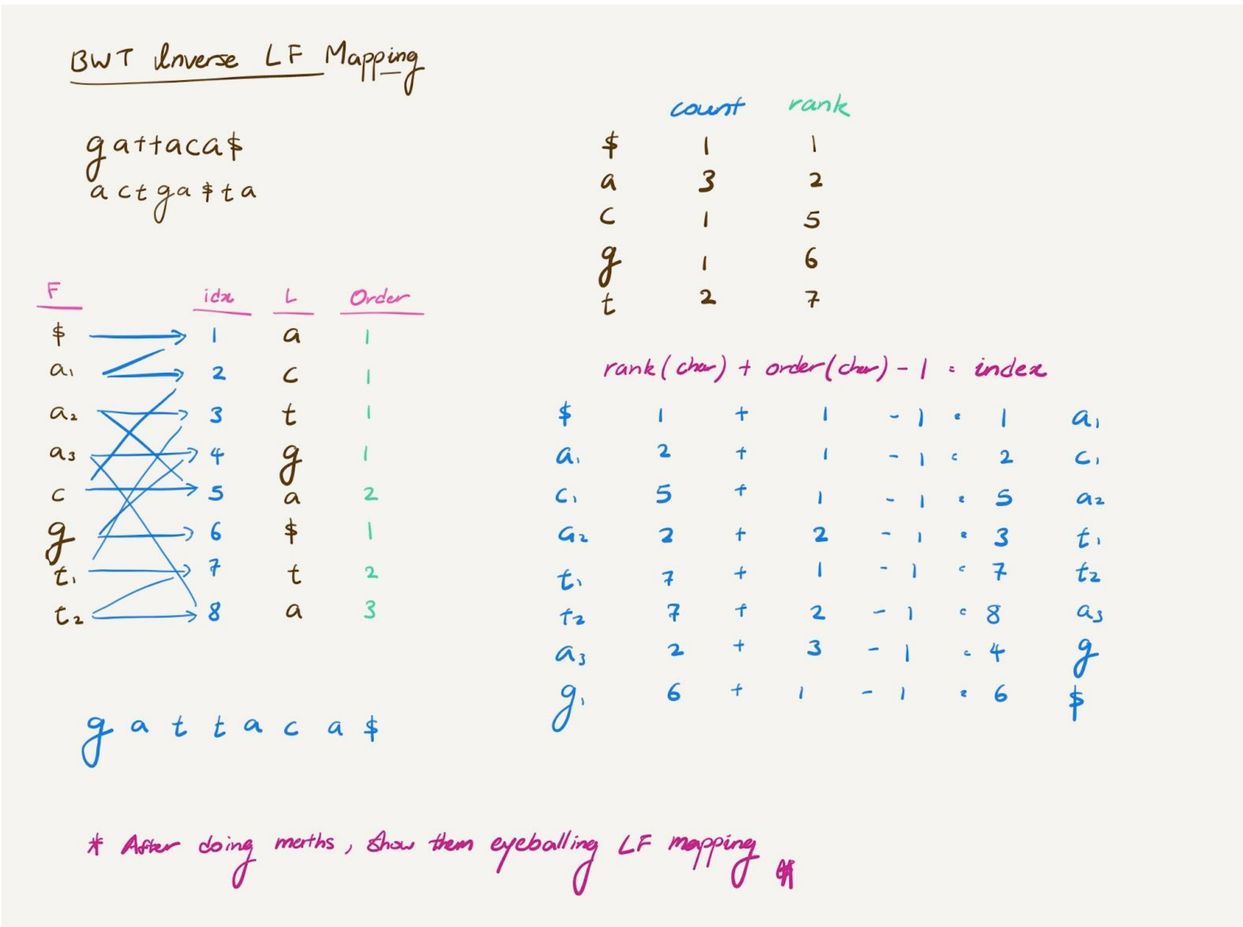
But it sucks O(N^3) time and O(N^2) space, so we shall reduce it to O(N) time and space.

LF Mapping

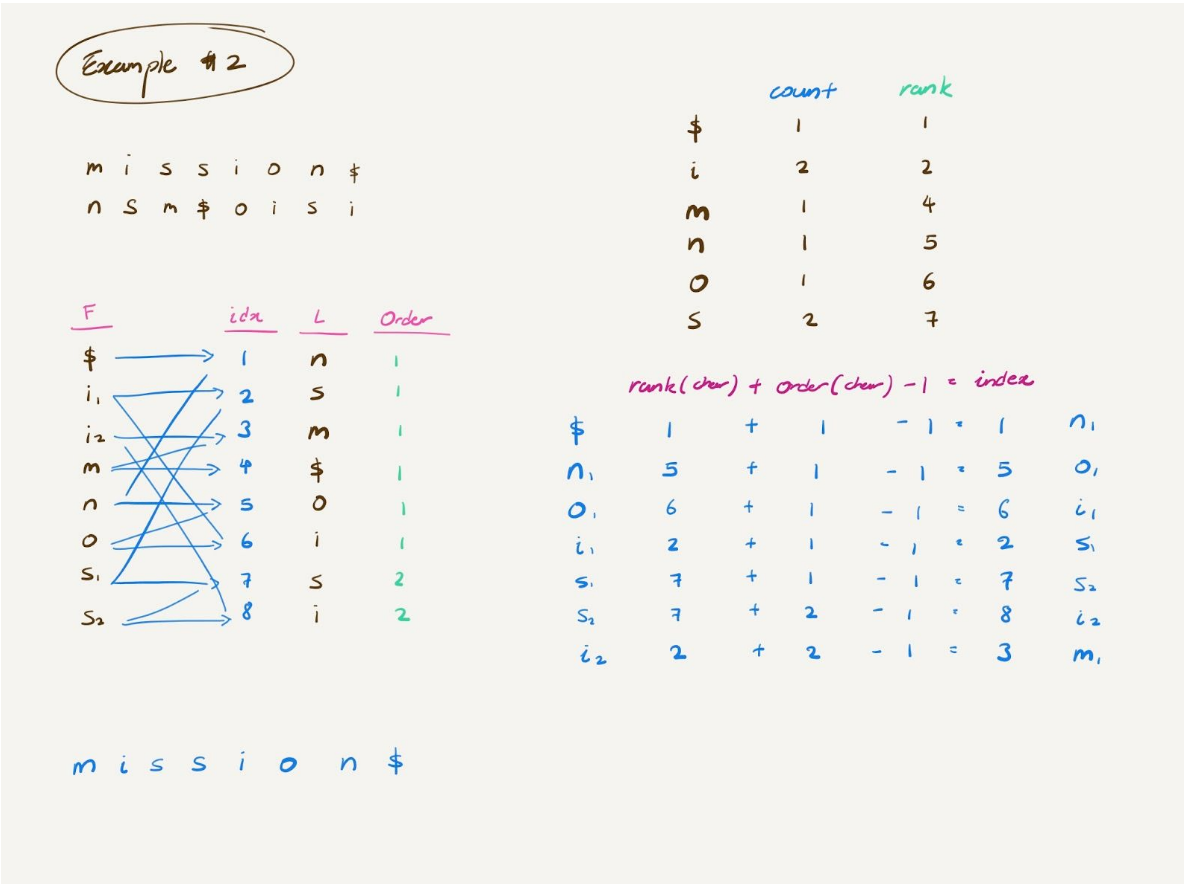
- We know that the first row is the original string.
- We know that the first column is sorted.
- The first occurrence of the letter in the first column is the same as the first occurrence of the letter in the last column and vice versa.
- Which means the order of the character in the first column is the same as the order in the last column. Because of the cyclic property.
- So this observation allows us to do LF-mapping, map the last with the first.
- I go through each position 1 by 1 O(N), so I do the math and prepend O(1 + 1) (O(N) * O(1)) is still O(N).

- Math
    - Rank(“char”) + Order(“char”) - 1 = Index
- Space complexity O(N)
    - Rank array and last column.

Example #1



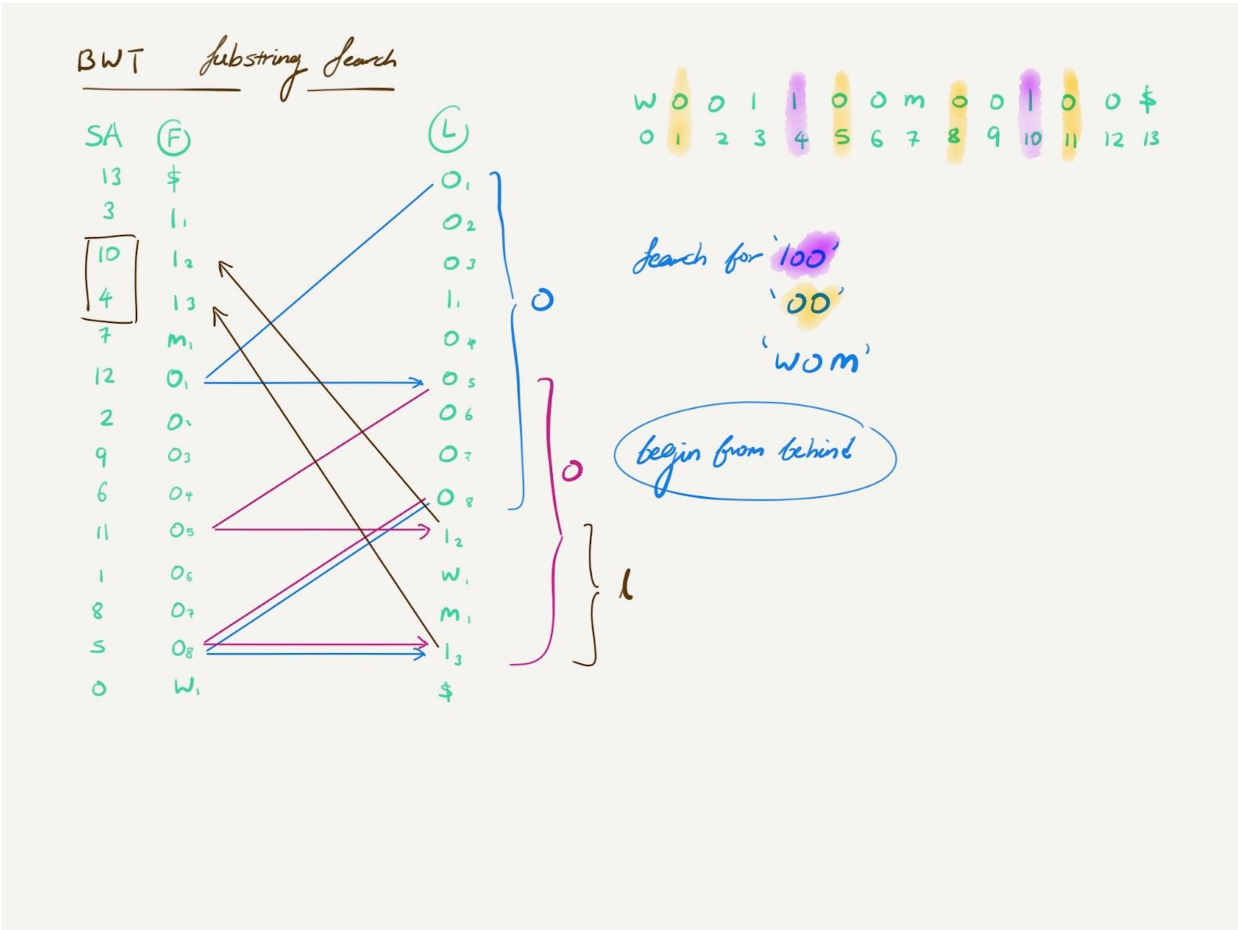
Example #2



BWT Substring Search

- If N string is millions or M pattern is billions, so you stored in compressed format BWT
- Search it quickly with BWT.
- Amazing thing about BWT, you can search a substring from the original string on a BWT string.
- We only use the last column.
- Goal is to search for the range within the space.
- Keep reducing the search space.
- Then search for the next alphabet.
- Complexity O(M) M is length of pattern
- I also know the position from the suffix array and number of occurrences.

Example #1



Conclusion

BWT

- Compress O(N)
- Reconvert O(N)
- Search O(M)