# FM-Index

# Suffix Array vs. FM-Index

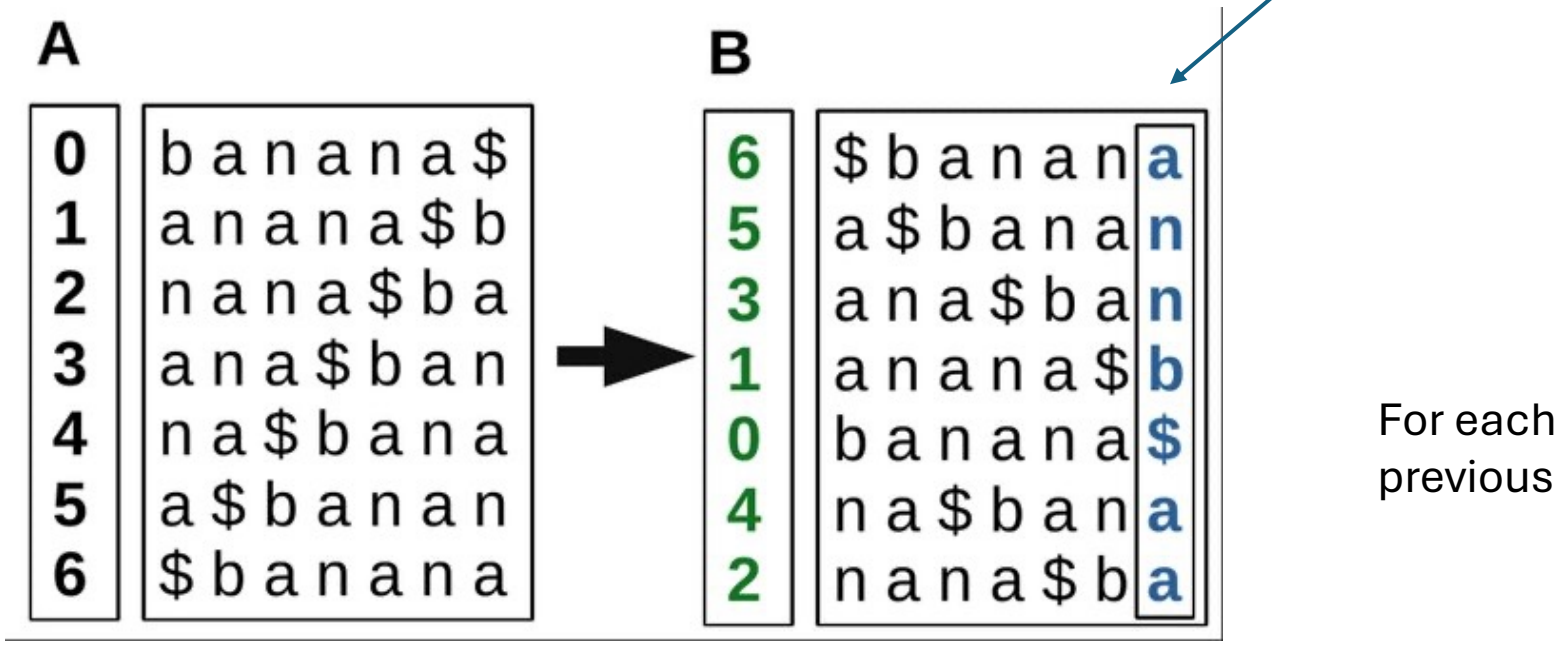
- FM-Index is a compressed full-text index.
- It space requirement is near entropy bound of the indexed text (often a few bits per character).
  - Repeats cause low entropy.
- For human genome about 3 billion bps,
  - Suffix array: 12-24GB
  - FM index: 1-3GB

# Burrows-Wheeler Transformation (BWT)

- A lossless string compress algorithm
  - Generate all rotations of a string $S$
  - Sort the generated rotations lexicographically
  - Keep only the last column as the output

BWT = annb$aa

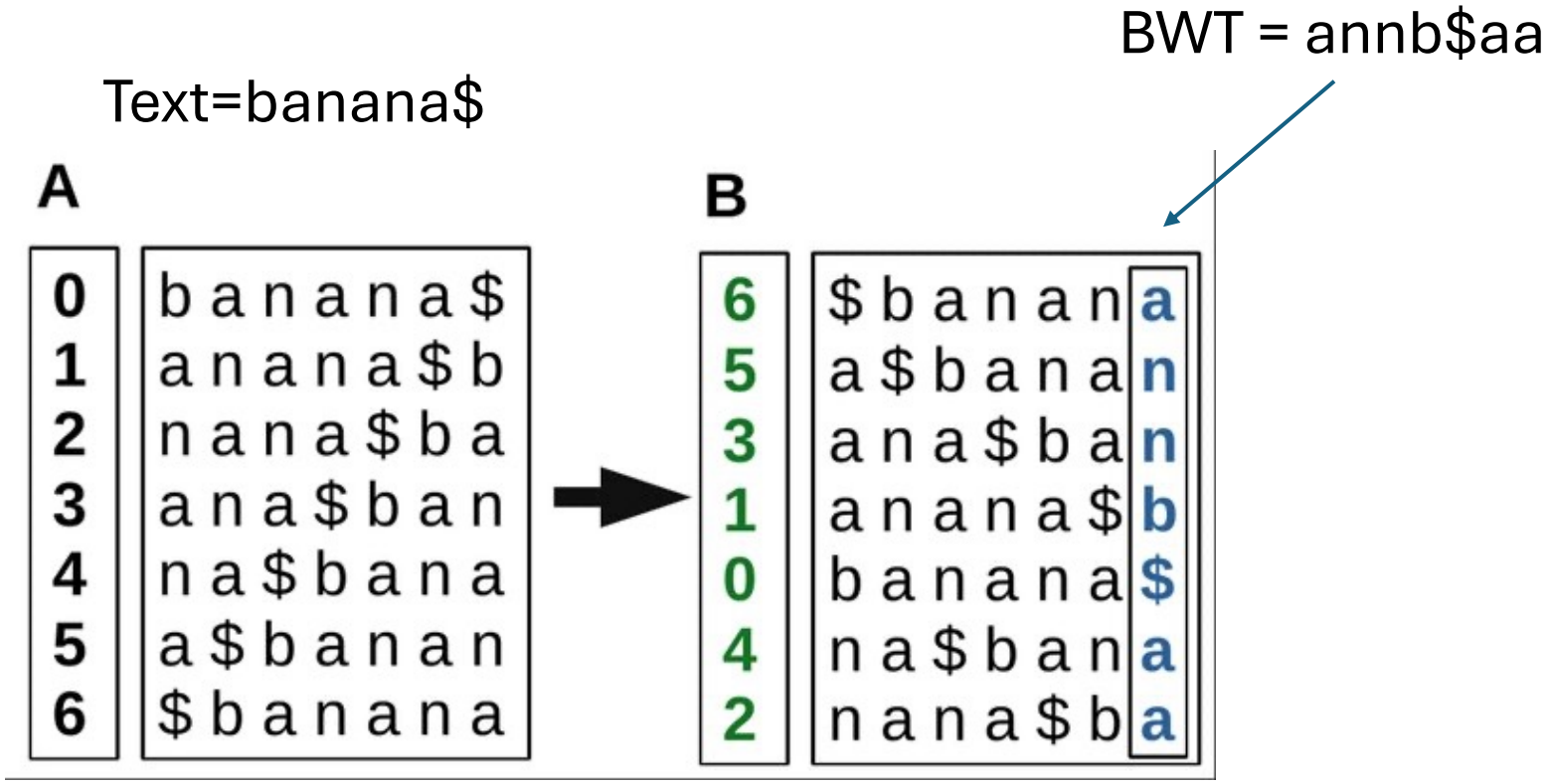
Text=banana$




For each row, the last column is the previous character of the first column.

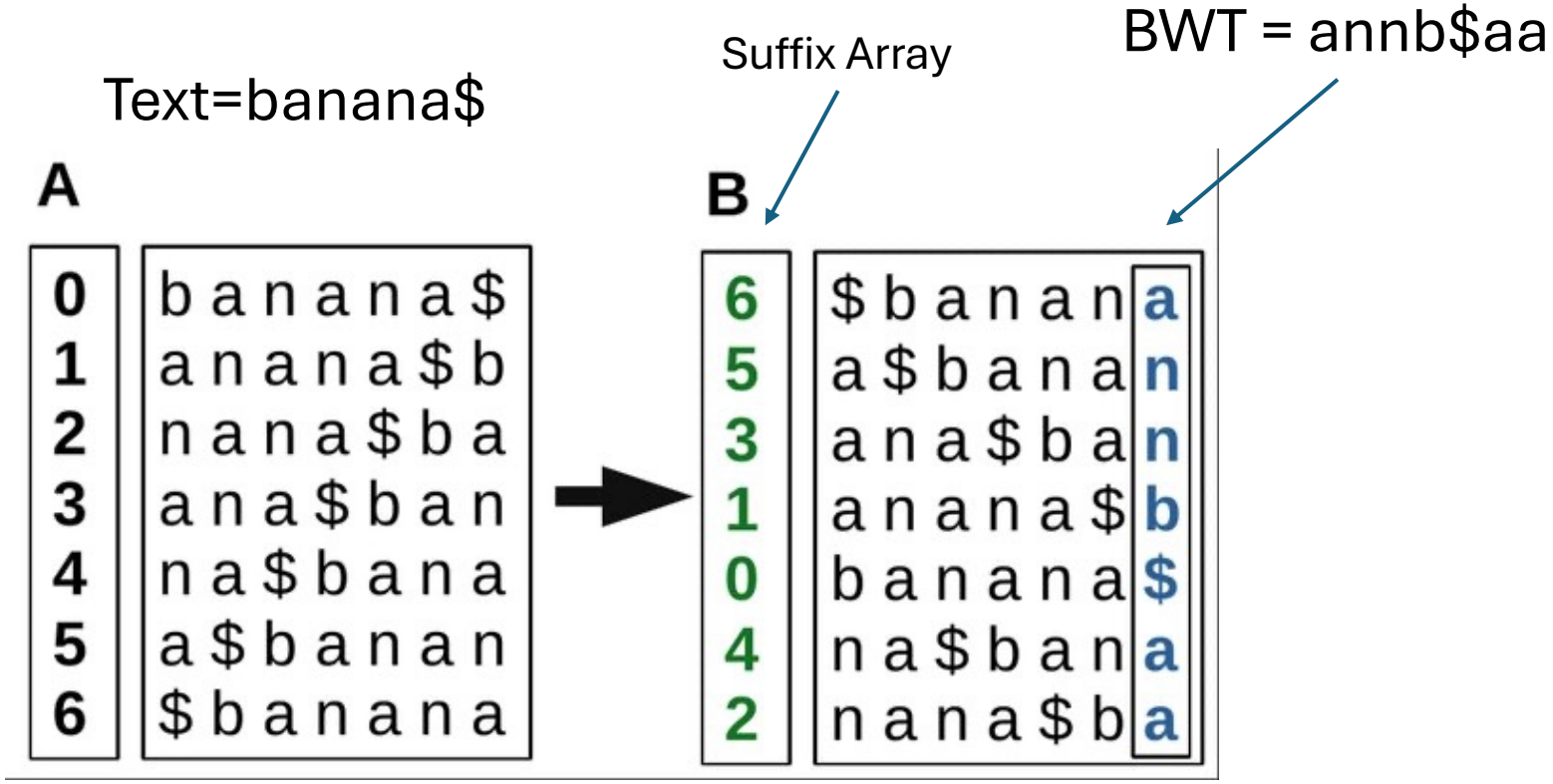
# Burrows-Wheeler Transformation (BWT)

- Last column is the previous character of the first column in the text.
- Since first column are grouped by similar substrings, their previous character are often grouped too. The last column tend to have long runs of identical letters.
- You can compress BWT by run-length encoding.

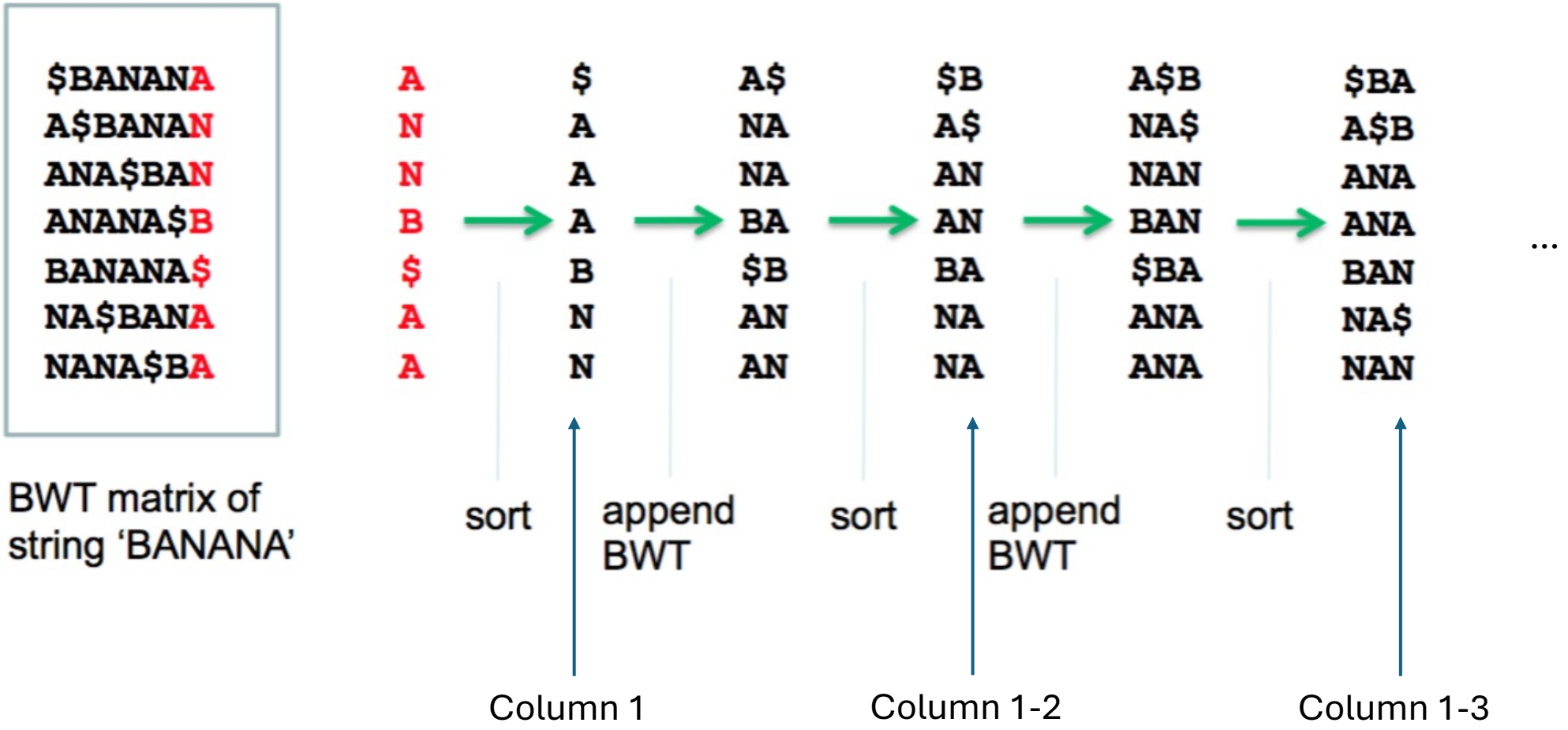


# Burrows-Wheeler Transformation (BWT)

- If indices are sorted together, you also get suffix array.
- Or, you can construct BWT efficiently by using a suffix array.



# Reconstruct Text from BWT (Naïve Approach)



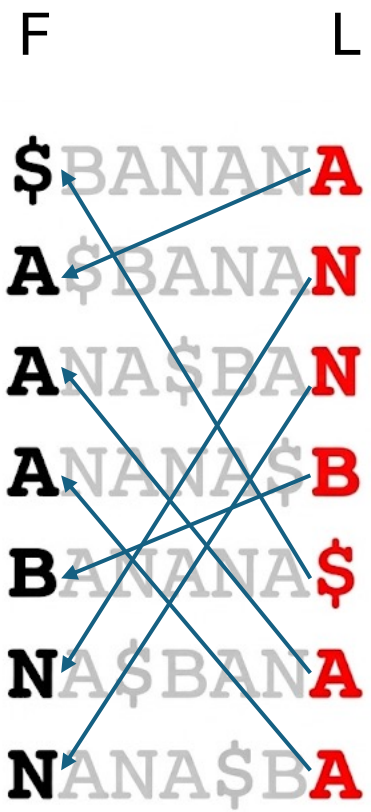
# BWT for Compression

- “With these techniques, a 45× coverage of real human genome sequence data compresses losslessly to under 0.5 bits per base, allowing the 135.3 Gb of sequence to fit into only 8.2 GB of space.”
- Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, Giovanna Rosone, Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform, *Bioinformatics*, Volume 28, Issue 11, June 2012, Pages 1415–1419, https://doi.org/10.1093/bioinformatics/bts173

# LF Mappipng

- Imagine occurrences of the same character in T have distinct identities.
- The mapping from the last column's character to its position in the first column is called LF mapping.
- In this example:
- LF[] = [1, 5, 6, 4, 0, 2, 3]

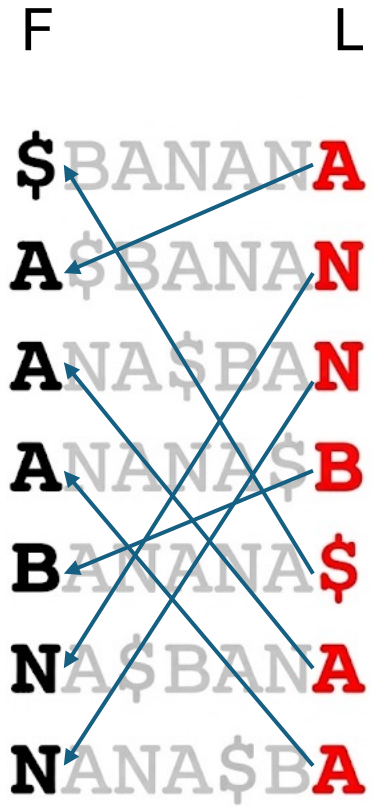
T= BANANA$



# Fast Reconstruction with LF

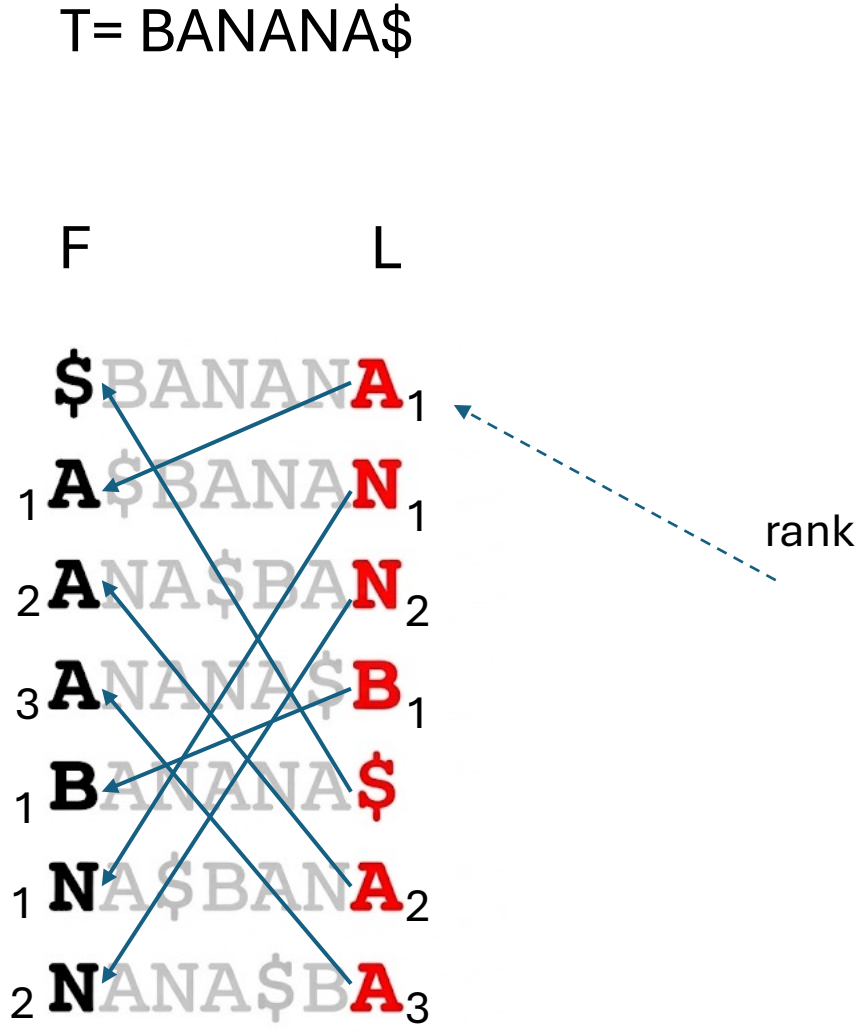
- If LF mapping is given, then reconstruction becomes easy.

```
T =''
r =1
while BWT[r] is not $
    T = BWT[r] + T
    r = LF[r]
```



# Rank

- **C(a)** is the number of occurrences of character smaller than a in T.
  - C(a) determines F.
- Rank(i) is the number of occurrences of L[i] up to row i.
- Question: Does LF mapping maintains the rank?

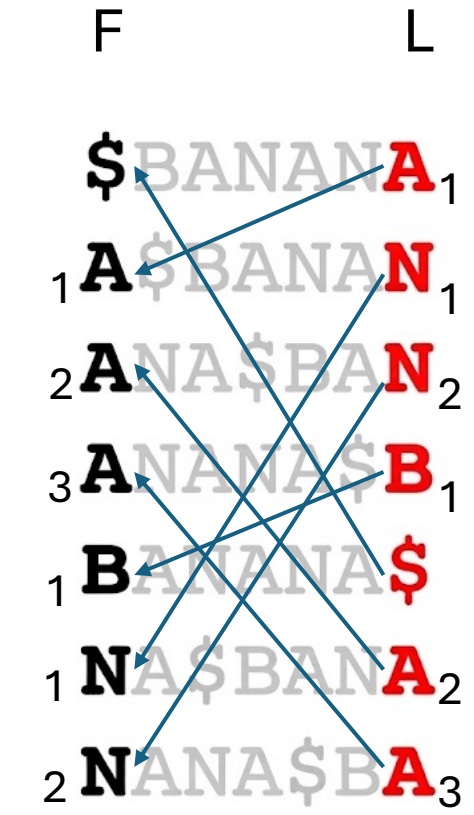


# Rank

- **Lemma**: the i-th occurrences of a character $a$ in L and F, respectively, are the same one.
  - **Proof**: Check all occurrences of $a$ in L, their rows start with the suffixes immediately after $a$. Therefore the orders are determined by the suffixes immediately after these $a$.
  - All occurrences of $a$ in F appear consecutively, therefore their orders are determined by the suffixes immediately after these $a$.
  - Therefore, they have the same order. **QED**.

- **Corollary**: If $a$=L[i] is the k-th occurrence of character $a$ in L, then F[LF[i]] is the k-th occurrence of $a$ in F.

T= BANANA$



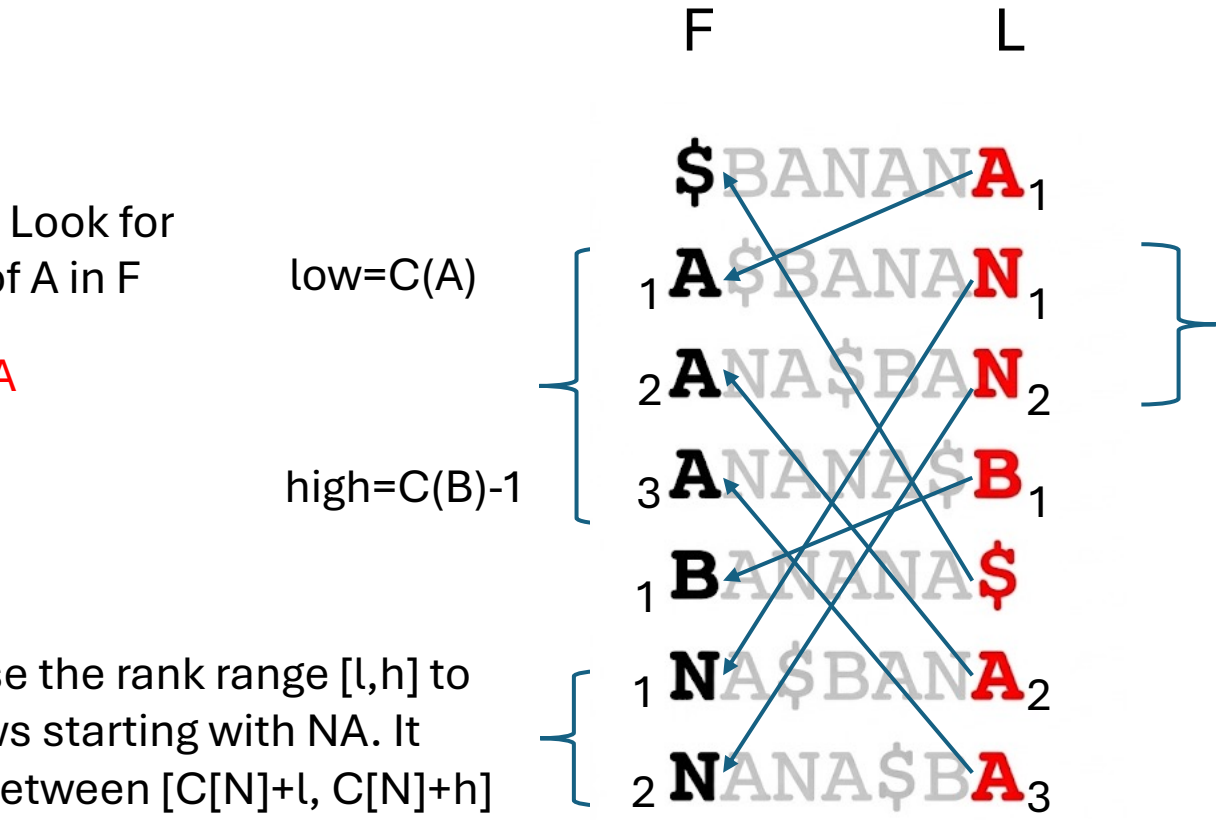
# Backward Query

Query = ANA

T= BANANA$

Step 1: Look for range of A in F

ANA




ANA

Step 2: All N in the current range in L precede A. Find their min and max ranks. Suppose it's [l, h].

Step 3: Use the rank range [l,h] to locate rows starting with NA. It must be between [C[N]+l, C[N]+h]

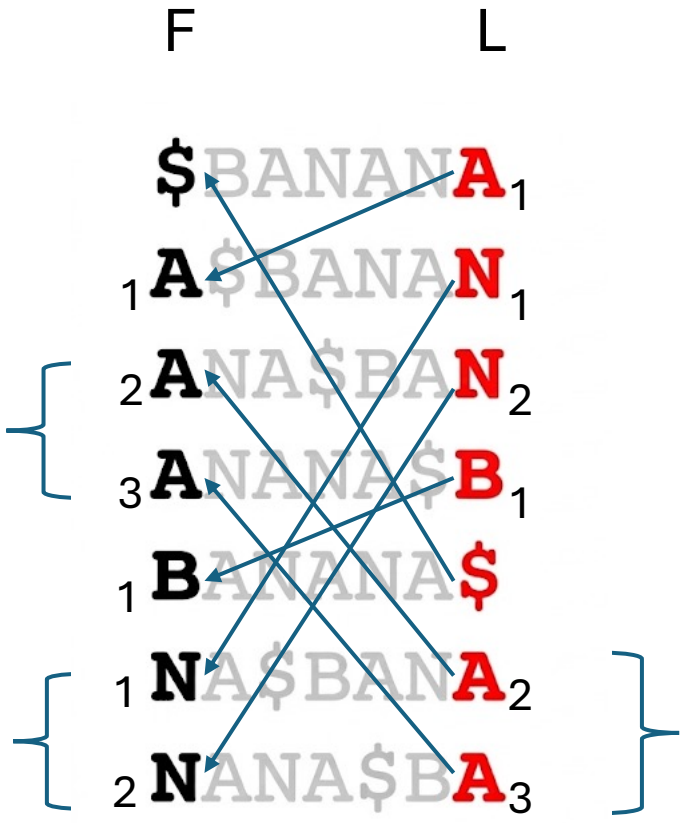
# Backward Query

Query = ANA

T= BANANA$

Step 5: Locate the rows start with ANA. They are [C(A)+l', C(A)+h']

Found!



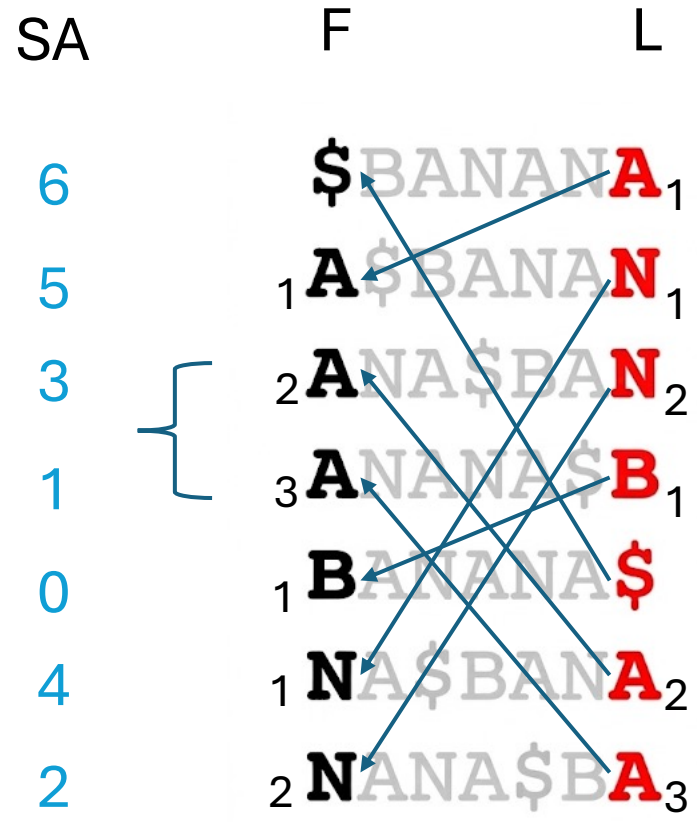

Step 4: All A in the current range in L precede NA. Find their ranks. Suppose they are between [l', h'].

# Backward Query

Query = ANA

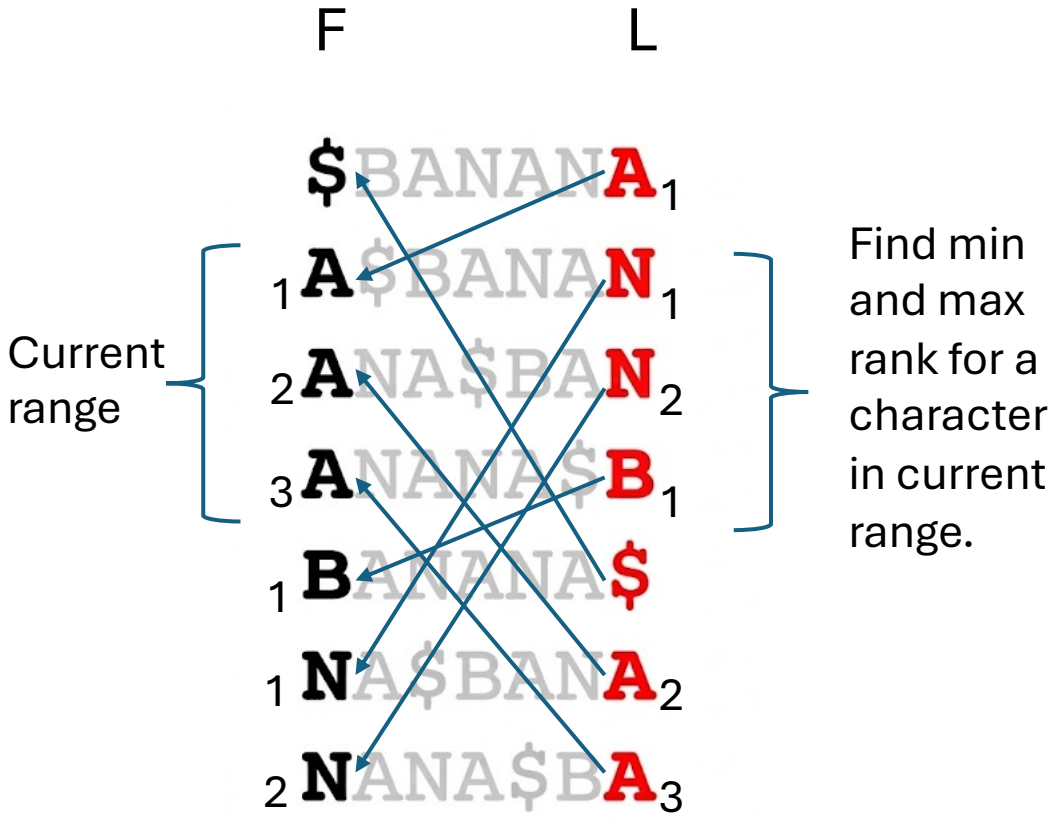
T= BANANA$

- If we only care about yes/no answer, we're done already.
- To find the actual location in T. One way is to also maintain a suffix array.
- The corresponding rows in SA will tell us the actual locations in T.



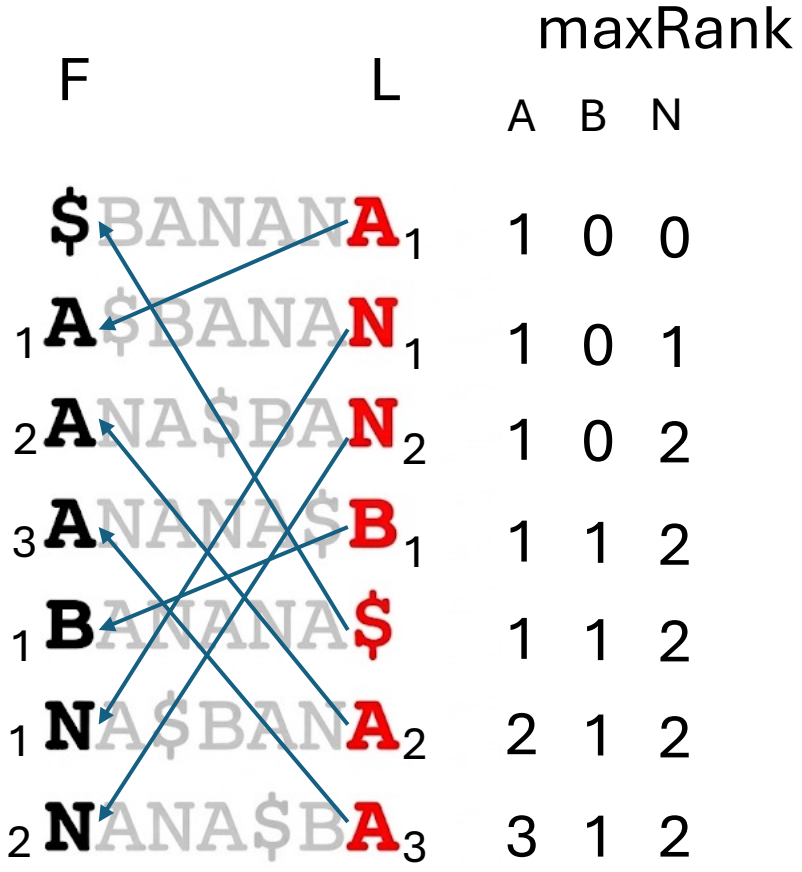
# Time Complexity

- O(m) iterations.
- But each iteration involves the query of the min and max rank of a character in the current range.
- This takes O(n) time in worse case.
- To reduce the time complexity, we can store **maxRank(a, i)** , which is the max rank of a up to row i.
- Then, finding min and max rank of a letter a in a range takes O(1) time.



# Fast Rank Query

- With the auxiliary structure maxRank,
- min rank of a within interval [l,h] is maxRank(a, l-1)+1
- max rank of a within interval [l,h] is maxRank(a, h)
- If min rank > max rank, then patern not found. Otherwise, continue the iteration.

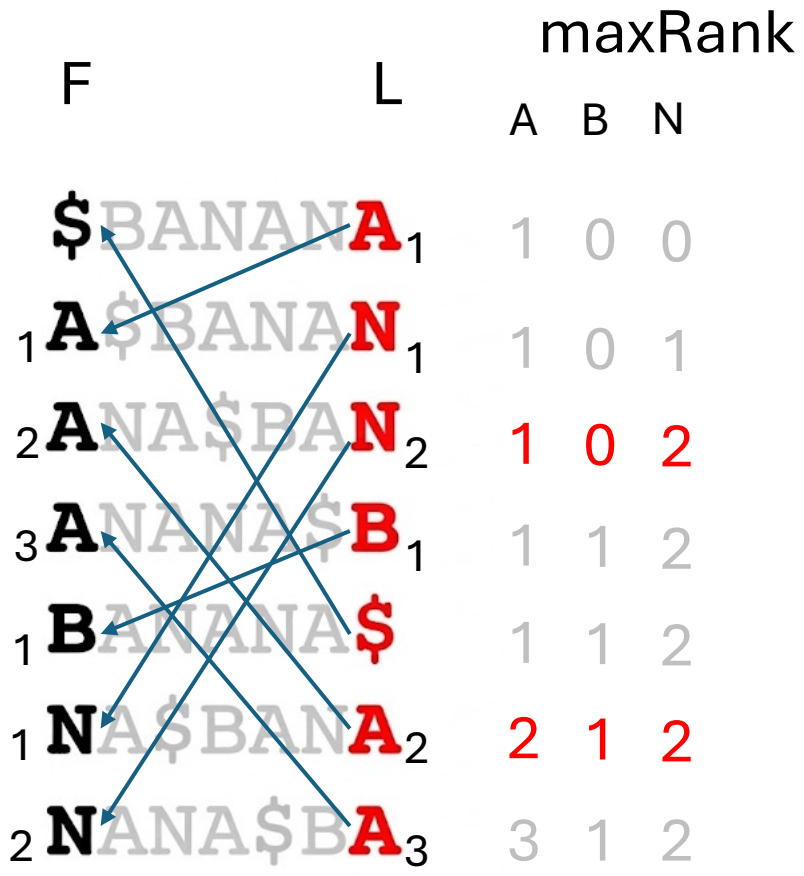


# Current Complexity

- Time complexity: O(m).
- Space complexity:
  - L: $n \log |\Sigma|$ bits
  - C[]: $4|\Sigma|$ bytes
  - maxRank[]: $4n|\Sigma|$ bytes
  - SA: 4n bytes
  - The total is not better than suffix array. We need to reduce it significantly.

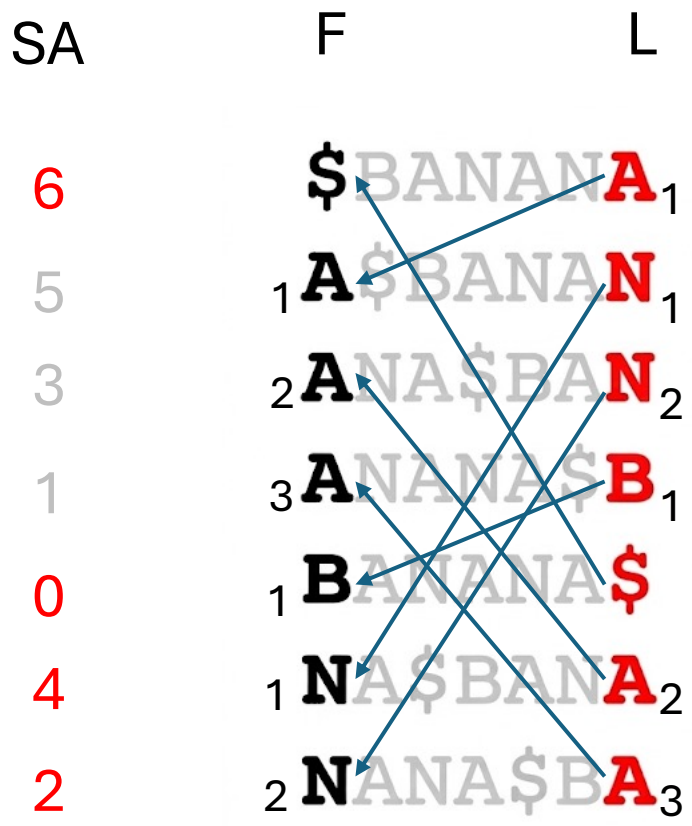
# Rank Checkpoints

- Store maxrank(a, i) every few other rows. E.g. every 3rd row (red) in the figure.
- The unstored values (gray) can be derived by
  - Retrieving the nearest checkpoint (red row)
  - Scanning through L to increase or decrease the value by 1 whenever encoutering the character.
- This takes constant time to scan, but reduce the memory footprint by a constant factor.
- The constant factor can be tuned. E.g. 32 or 64 or even c log(n).
- If you don't like the idea of scanning, this can be achieved by some careful bit operations and the POPCNT instruction on today's CPU.



# SA Checkpoint

- SA checkpoint uses the same idea. We only store the even values of the suffix indices.
- If the search lands on an even index, we are okay.
- If it lands on an odd indiex k, e.g. 3 in the example. We find its previous character $N_2$ by looking in L at the same row, and use LF mapping to find that $N_2$ row has value 2 in SA. Adding 1 to the value will get the correct suffix index.
- We also need to have a bit array indicating whether a position is checkpointed (otherwise, the grayed out space cannot be released)
- If we do SA checkpoint on every k-th index, then it takes O(k) steps for this process. But k is a constant.



# SA Checkpoint

| SA | isChecked | SAC |
|---|---|---|
| 6 | 1 | 6 |
| 5 | 0 | 0 |
| 3 | 0 | 4 |
| 1 | 0 | 2 |
| 0 | 1 | |
| 4 | 1 | |
| 2 | 1 | |

- Previous slide explained that one can compute the gray SA values if we can look up the SA values.
- SA is stored by a BitSet (isChecked) and the checked suffixes (SAC).
- How to map the indices in SA (and isChecked) to indices in SAC?

# SA Checkpoint

| SA | isChecked | total | checked |
|---|---|---|---|
| 6 | 1 | 1 | 6 |
| 5 | 0 | 1 | 0 |
| 3 | 0 | 1 | 4 |
| 1 | 0 | 1 | 2 |
| 0 | 1 | 2 | |
| 4 | 1 | 3 | |
| 2 | 1 | 4 | |

- Total[i] stores the accumulated checked positions up to i.
- With total array, we can do it.
- How to store the total array efficiently?

# Summary

- LF mapping can be used to construct the original string in linear time.
- LF is a virtual mapping.  LF[i] = rank[i]+C(L[i])
- maxRank allows backward search in O(m) time.
- Rank checkpoint allows to store takes less than n byte space for constant sized alphabet.
- FM index is a compressed data structure supporting O(m) text search.

# Comparison among indexing data structures

| | k-mer index | Suffix Tree | Suffix Array | BWT & FM Index |
|---|---|---|---|---|
| Space | $O(L)$ | $O(L)$ | $O(L)$ | $\ll O(L)$ |
| Search Time | $O(p)$ | $O(p)$ | $O(p \log L)$ | $O(p)$ |

- L: target (genome) length
- p: pattern (read) length

# Appendix: Assignment 2 Statistics

| Method | score | n |
|---|---|---|
| Fine-tuned ESM-2 | 0.7317 | 1 |
| BI-GRU | 0.7279 | 2 |
| BI-LSTM | 0.6647 | 6 |
| CNN | 0.6635 | 13 |
| 4mer | 0.6130 | 6 |
| MLP | 0.5922 | 2 |
| 5mer | 0.5823 | 1 |
| 3mer | 0.5800 | 3 |
| Transformer | 0.5100 | 1 |

# Appendix: LCP for Suffix Array

- Definitions:
  - SA[i] is the suffix ranked at i.
  - rank[i] is the rank of suffix at i. This is also called reverse SA.
  - Then SA[rank[i]] = i
  - LCP[i] = lcp(SA[i], SA[i-1])
- Key idea: If two suffixes share a prefix of length k, then their **next suffixes** (shifted by one) share at least k-1.
- Algorithm:

```
for i in 0..n-1:
   if rank[i] == 0:
      LCP[0] = 0
      continue
   j = SA[rank[i] - 1] // previous suffix in sorted order
   while i+k < n and j+k < n and T[i+k] == T[j+k]:
      k += 1
   LCP[rank[i]] = k
   if k > 0:
      k -= 1
```