FIT2004: Supplementary notes for lecture 7

Advanced String Processing: The Burrows-Wheeler Transform

DANIEL ANDERSON [1]

Two of the most common problems that arise in string processing are text compression and pattern matching. We've already seen several algorithms for performing pattern matching, each with different advantages and disadvantages. The Burrows-Wheeler transform is a string transformation with powerful applications to text compression and pattern matching on very large strings.

Summary: Burrows-Wheeler Transform

In this lecture, we cover:

- The Burrows-Wheeler transform of a string
- The inverse Burrows-Wheeler transform
- The application of BWT to pattern matching

Recommended Resources: Burrows-Wheeler Transform and Pattern Matching

- Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994
- `http://users.monash.edu/~lloyd/tildeAlgDS/Strings/`

The Burrows-Wheeler Transform

The Burrows-Wheeler transform (BWT) of a string is a permutation of the characters in that string that often improves the compressibility of the resulting string and facilitates fast pattern matching.

Definition: The Burrows-Wheeler transform

The Burrows-Wheeler transform of a string `S[1..N]` is a string formed by taking the final character of each cyclic permutation of `S` arranged in sorted order.

Let's make this clearer with an example. Consider our favourite string "banana$", remembering that we use the special $ symbol to denote the end of the string, and that it is considered lexicographically less than any other character in the alphabet. The *cyclic permutations* of a string are the permutations of that string that can be obtained by rotating the elements in a cyclic order. For example, the cyclic permutations of "banana$" are

```
banana$
anana$b
nana$ba
ana$ban
na$bana
a$banan
$banana
```

For a string of length `N`, there are exactly `N` cyclic permutations, given by

`S[i..N] + S[1..i-1]`, for `i` = 1 to N,

where `S[...] + S[...]` denotes a string concatenation. To obtain the BWT of a string, we sort these cyclic permutation in lexicographical order. For "banana$", this gives us

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

The BWT of “banana$” is the string formed by taking the final character of each of these sorted cyclic permutations, so we have

```
BWT("banana$") = "annb$aa"
```

Useful properties of the BWT

Property 1: BWT groups runs of similar characters

We can see in the BWT of “banana$”, that the transformed text often consists of groups of repeated characters. This makes BWT very useful for **text compression** methods that take advantage of repeated characters.

Property 2: The BWT matrix contains permutations of all the k-mers of `S[1..N]`

The BWT matrix is the matrix of all sorted cyclic permutations of `S[1..N]`. For example, the BWT matrix for “banana$” is

$$M = \begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}$$

Notice that the BWT of `S[1..N]` is the string formed by the final column of M. Also, we can see from that above matrix that

1. Every column of M is a permutation of `S[1..N]`
2. Every pair of adjacent columns of M contains a permutation of all of the length 2 substrings of `S[1..N]` (also called 2-mers of `S`)
3. Every sequence of k consecutive columns of M contains a permutation of all of the length k substrings of `S[1..N]` (called the k-mers of `S`)

This observation may not seem obviously useful yet, but it underpins the **pattern matching** algorithm that utilises BWT later.

Property 3: The last column of the BWT matrix precedes the first column

Since M contains the sorted cyclic permutations of `S[1..N]`, the contents of the last column of M are precisely the characters that cyclically proceed the characters in the first column. **This might seem unspectacular, but this is the most important property of the BWT which underpins many of the following observations and subsequent algorithms that we will derive. Make sure this sinks in before proceeding.**

Property 4: The BWT is invertible

Given the BWT of a string `BWT(S[1..N])`, it is possible to transform back to the original text `S[1..N]` without any extra information. In particular, this means that compression methods can take advantage of BWT without any extra overhead, ie. it is a “free” improvement to the compressibility of a string.

Computation of the BWT

Clearly we can compute the BTW of a string in $O(N^2 \log(N))$ time by simply storing all of the cyclic permutations in a list, sorting them using a fast sorting algorithm and then taking the last character of each. This is far too slow in practice, so better methods for computing the BTW have been developed.

The key observation to make is that because the special end-of-string character \$ is lexicographically less than all other characters, the order of the cyclic permutations is precisely the same order as the sorted suffixes of the string.

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

Notice that the characters in black are exactly the sorted suffixes. This means that we can use the suffix array to compute the BWT efficiently.

Algorithm: Burrows-Wheeler Transform by Suffix Array

```
function BWT(S[1..N])
    Set SA[1..N] = suffix_array(S)
    Set result[1..N] = {}
    for i = 1 to N do
        result[i] = S[SA[i]-1] or '$' if SA[i] = 1
    end for
    return result
end function
```

Since suffix arrays can be computed in linear time, this implies that the Burrows-Wheeler transform can also be computed in just linear time in the length of the text.

The Inverse Burrows-Wheeler Transform

It is not obvious, but one of the most remarkable properties of the BWT is that it is invertible. Let's denote by $\texttt{BWT}^{-1}$ the inverse BWT, ie. the transformation such that

$$\texttt{BWT}(\texttt{BWT}^{-1}(\texttt{S})) = \texttt{BWT}^{-1}(\texttt{BWT(S)}) = \texttt{S}.$$

To prove that the BWT is invertible, let's look at a very inefficient way of performing the inversion first. Then, we will study a more efficient algorithm for the inverse BWT.

The naive inversion method

The naive inversion algorithm reconstructs the entire BWT matrix M starting from just the transformed string `BWT(S)`. For example, we will use our favourite string "banana\$", whose BWT is given by "annb\$aa". Initially, we know only the contents of the last column of M, since this contained the BWT string.

$$M = \begin{bmatrix} ? & ? & ? & ? & ? & ? & a \\ ? & ? & ? & ? & ? & ? & n \\ ? & ? & ? & ? & ? & ? & n \\ ? & ? & ? & ? & ? & ? & b \\ ? & ? & ? & ? & ? & ? & \$ \\ ? & ? & ? & ? & ? & ? & a \\ ? & ? & ? & ? & ? & ? & a \end{bmatrix}$$

Since the BWT matrix M contains all of the cyclic permutations in sorted order, we know that in particular, the first column is sorted. Therefore, we can easily obtain the first column of M by sorting the characters of `BWT(S)`. Sorting the characters of “annb\$aa”gives “\$aaabnn”.

$$M = \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & b \\ b & ? & ? & ? & ? & ? & \$ \\ n & ? & ? & ? & ? & ? & a \\ n & ? & ? & ? & ? & ? & a \end{bmatrix}$$

Next, we use the fact that we know that the last column of M contains precisely the characters that proceed the characters in the first column (Property 3). Therefore, if we append the first column of M to the last column of M, we will obtain the set of all 2-mers of `S` (length 2 substrings.) These 2-mers can then be sorted to obtain the first two columns the BWT matrix M.

$$\begin{bmatrix} a \\ n \\ n \\ b \\ \$ \\ a \\ a \end{bmatrix} + \begin{bmatrix} \$ \\ a \\ a \\ a \\ b \\ n \\ n \end{bmatrix} = \begin{bmatrix} a & \$ \\ n & a \\ n & a \\ b & a \\ \$ & b \\ a & n \\ a & n \end{bmatrix} \xrightarrow{\text{sort}} \begin{bmatrix} \$ & b \\ a & \$ \\ a & n \\ a & n \\ b & a \\ n & a \\ n & a \end{bmatrix}$$

So our reconstructed BWT matrix M now looks like this.

$$M = \begin{bmatrix} \$ & b & ? & ? & ? & ? & a \\ a & \$ & ? & ? & ? & ? & n \\ a & n & ? & ? & ? & ? & n \\ a & n & ? & ? & ? & ? & b \\ b & a & ? & ? & ? & ? & \$ \\ n & a & ? & ? & ? & ? & a \\ n & a & ? & ? & ? & ? & a \end{bmatrix}$$

Again, we know that the final column of M contains the characters that proceed the characters in the first column (Property 3), so we can obtain the 3-mers (substrings of length 3) of `S` by appending the first two columns of M to the final column. Sorting all of the 3-mers will then give us the first three columns of M.

$$\begin{bmatrix} a \\ n \\ n \\ b \\ \$ \\ a \\ a \end{bmatrix} + \begin{bmatrix} \$ & b \\ a & \$ \\ a & n \\ a & n \\ b & a \\ n & a \\ n & a \end{bmatrix} = \begin{bmatrix} a & \$ & b \\ n & a & \$ \\ n & a & n \\ b & a & n \\ \$ & b & a \\ a & n & a \\ a & n & a \end{bmatrix} \xrightarrow{\text{sort}} \begin{bmatrix} \$ & b & a \\ a & \$ & b \\ a & n & a \\ a & n & a \\ b & a & n \\ n & a & \$ \\ n & a & n \end{bmatrix}$$

We now just repeat this process until we have the entire matrix. Append the first k columns to the final column and sort the results to obtain the first $k+1$ columns and repeat until we have reconstructed the entire matrix M. Once we have reconstructed the entire matrix, the original string `S` is simply the row that ends with the \$ character, or equivalently, it is the first row cyclically shifted one character to the left.

$$M = \begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ \mathbf{b} & \mathbf{a} & \mathbf{n} & \mathbf{a} & \mathbf{n} & \mathbf{a} & \boldsymbol{\$} \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}$$

While this construction demonstrates that indeed the BWT can be inverted, it is not an efficient way of doing so. Some more clever observations about the properties of the BWT and the matrix M will help us to derive a more efficient inversion algorithm.

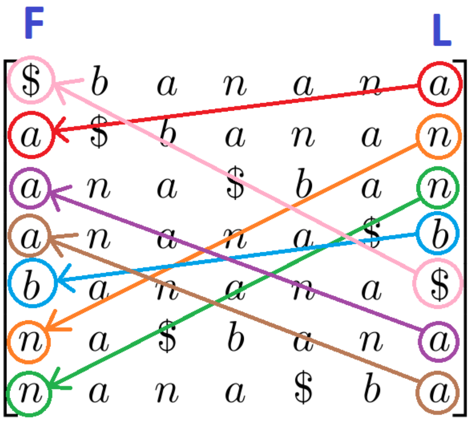
Efficient inversion of the BWT

The key to performing the inversion of the BWT fast is a single powerful observation called the **LF-mapping**.

Property 5: The LF-mapping of the BWT matrix

In the BWT matrix M, the i^{th} occurrence of a character C in the **last column (L)** corresponds to the i^{th} occurrence of C in the **first column (F)**.

In other words, the first occurrence of 'a' in the last column corresponds to the first occurrence of 'a' in the first column, the third occurrence of 'r' in the last column corresponds to the third occurrence of 'r' in the first column and so on.



$$\begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}$$

The LF-mapping for the BWT matrix M of the string "banana\$"

Proof: Why the LF-mapping works

Consider all of the occurrences of some character C in the last column (L) of the BWT matrix. Since the characters of the last column proceed the characters of the first column (Property 3), the relative sorted order of the occurrences of C in the last column is decided by the lexicographical order of the rows containing the aforementioned occurrences of C. Since the rows of the matrix A are sorted lexicographically, the contents of the first column (F) of M are also sorted by the lexicographical ordering of the rows, hence the occurrences of the character C in the first column are sorted the same as those in the last column.

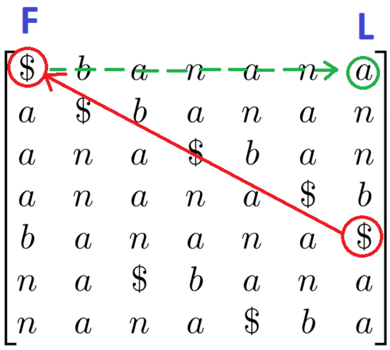
Utilising the LF-mapping, we can quickly and efficiently produce the inverse BWT of a string. Given the BWT of a string `BWT(S[1..N])`, we will produce the original string one character at a time by starting from the end and following the LF-mapping to each previous character until we reach the end again. We know that the final character is always the special terminating character \$ which occurs only once in the string, so we know our initial location in the last column is in the position at which '\$' occurs.

Using the string "banana\$" to demonstrate:

```
current_string = "$",    current_position = 5
```

Since we are at the first occurrence of '\$' in the string, using the LF-mapping, we know that the corresponding location in the first column would be the first occurrence of '\$', which would occur at position 1. Invoking Property 3 yet again, we know that the previous character in the string must be the one at position 1 in the last column (the BWT string), which is an 'a'.

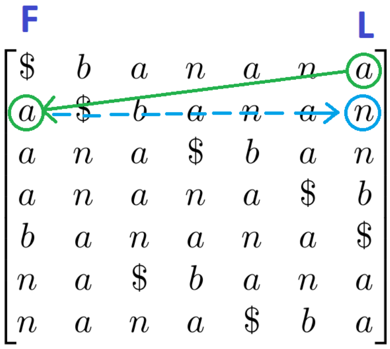
```
current_string = "$a",    current_position = 1
```



Moving from the last character in the string '$' and deducing the second last character 'a' via the LF-mapping

Now we are at position 1, and since this is the first occurrence of the character 'a', we know that the corresponding position in the first column of M will be the first occurrence of 'a' there, which is at position 2. Again, using Property 3, we can deduce that the next previous character in our reconstructed string is the one in position 2 of the BWT string, which is an 'n'.

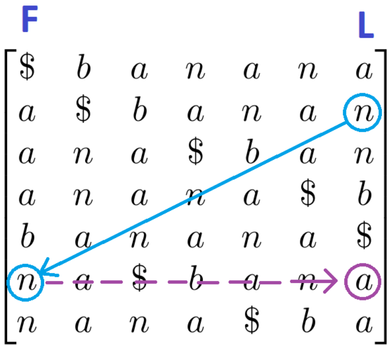
`current_string = "$an",    current_position = 2`



Moving from the current character in the string 'a' and deducing the previous character 'n' via the LF-mapping

At position 2 we are looking at the first occurrence of 'n', so we calculate that the first occurrence of 'n' in the first column would be at position 6 (this can be computed by the fact that there are 5 characters in the string that are lexicographically less than 'n'). Using Property 3, we know that the previous character in the string is therefore 'a' (the 6th character in the BWT string.)

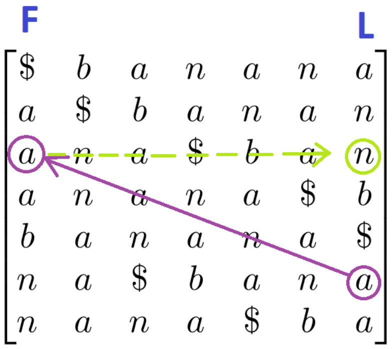
`current_string = "$ana",    current_position = 6`



Moving from the current character in the string 'n' and deducing the previous character 'a' via the LF-mapping

We are at position 6 looking at the second occurrence of 'a', so we calculate that the second occurrence of 'a' occurs at position 3 in the first column, and hence deduce that the previous character in the string is an 'n'.

```
current_string = "$anan",     current_position = 3
```

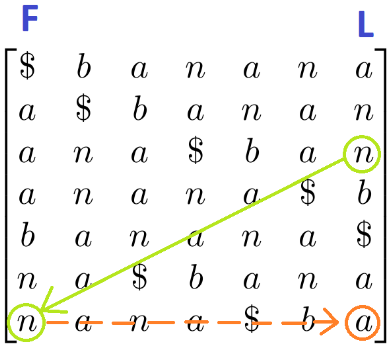


$$\begin{array}{ccccccc} F & & & & & & L \\ \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{array}$$

Moving from the current character in the string 'a' and deducing the previous character 'n' via the LF-mapping

From position 3 at the second occurrence of 'n' in the last column, we find that the second occurrence of 'n' in the first column appears at position 7, so the previous character of the string is the one at position 7 in the transformed string, which is the third 'a'.

```
current_string = "$anana",     current_position = 7
```

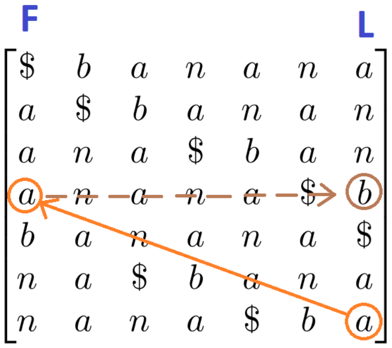


$$\begin{array}{ccccccc} F & & & & & & L \\ \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{array}$$

Moving from the current character in the string 'n' and deducing the previous character 'a' via the LF-mapping

Finally, from the location of the third 'a', we calculate that the previous character is at position 4, which corresponds to a 'b' in the transformed string.

```
current_string = "$ananab",     current_position = 4
```



$$\begin{array}{ccccccc} F & & & & & & L \\ \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{array}$$

Moving from the current character in the string 'a' and deducing the previous character 'b' via the LF-mapping

If we followed the LF-mapping one more time, we would arrive back at the '$' character where we began. Now that we are finished, we can simply reverse the reconstructed string to obtain the inverted text "banana$". Notice how this entire process never uses any of the columns of the matrix M except for the first and the last.

Next, we should figure out how to actually do this inversion quickly. Clearly if we just iterate through the transformed string at every step to find out which occurrence we are looking at, the entire process will take $O(N^2)$, which is still not very good. To avoid this, we should pre-compute the number of occurrences of character in the string with a single linear pass through the transformed text, and then use this information later to quickly determine the index of a particular occurrence.

1. First, perform a linear pass over the transformed text and count the number of occurrences of each character. Since the alphabet is assumed to be known and fixed, we can use a simple array to store the counts and hence this step takes just $O(N)$ time.

2. We can then compute for each character c in the alphabet, the location where c first occurs in the left column of M. We refer to this location as the rank of the character c. The location of the first occurrence of the character c simply corresponds to the total number of occurrences of characters that are alphabetically less than c, so we can compute this for all of the characters by iterating through the alphabet and accumulating the prefix sums of the counts that we computed in the first step. This step takes $O(|\mathcal{A}|)$ where $|\mathcal{A}|$ is the size of the alphabet, assumed to be fixed and small.

3. We must also compute for each location in the string, the number of times that the character at that position occurred earlier in the string. This computation can be combined with step 1 by storing an additional array of length N and using the character counter as it is updated.

With these statistics precomputed, we can compute for any character c occurring at position i, the corresponding position in the LF-mapping in just $O(1)$ time by observing that

$$\texttt{pos(i)} = \texttt{rank(c)} + \texttt{prior\_occurrences(i)}$$

In other words, the corresponding position is simply the index of the first occurrence of c plus 1 for each occurrence of c that came before position i in the transformed text.

Since we precomputed the necessary statistics to compute the LF-mapping in $O(1)$, the total running time of the inverse BWT algorithm is $O(N)$.

Pattern Matching with the Burrows-Wheeler Transform

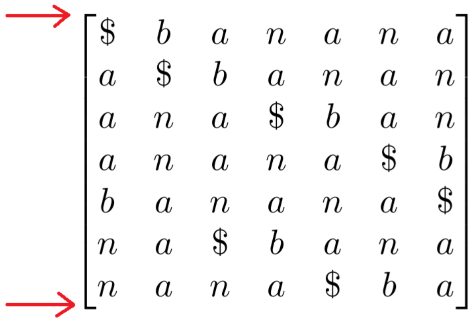
In its original inception, the Burrows-Wheeler transform was designed and implemented for the purposes of data compression. Many years later, it was discovered that transform's close similarities to the suffix array combined with the LF-mapping could actually be used to perform fast pattern matching on the original text.

We have already seen how to perform pattern matching in $O(N+M)$ with the Rabin-Karp algorithm, or in just $O(M\log(N))$ or $O(M)$ with a suffix array or suffix tree respectively. BWT, like the later two is highly applicable to situations where we have a single extremely large text string `T[1..N]` and a large number of relatively small patterns `P[1..M]`.

Since the BWT matrix consists of the sorted cyclic shifts of the text `T`, as was the case for suffix arrays, all occurrences of a particular pattern will occur at indices that form a contiguous section of the sorted cyclic shifts. The BWT pattern matching algorithm works similar to the suffix array pattern matching algorithm by narrowing down the this range of indices, which can then be translated into a sequence of actual indices into the text using the suffix array.

The algorithm searches the pattern string `P` from the back, finding at each step, the range of locations of the cyclic shifts of `T` that have the current suffix `P[i..M]` as a prefix. The suffix is expanded, adding an additional character at each step and finding the new matching locations fast by exploiting Property 3 and the LF-mapping.

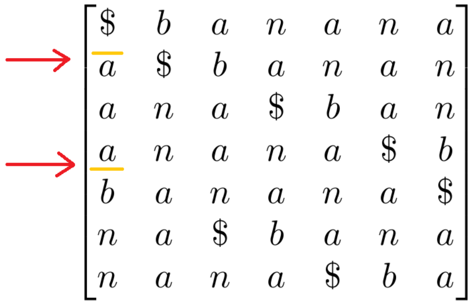
Let's do an example before seeing the details. Consider our favourite text string "banana$" and suppose that we are searching for the pattern "ana". Initially, we consider the empty suffix of "ana", the empty string, so the entire matrix is a match (everything contains the empty string as a prefix).



$$\begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}$$

The initial location of the matches for the empty suffix. Every string is initially a match before we begin to shrink the size of the matched ranged.

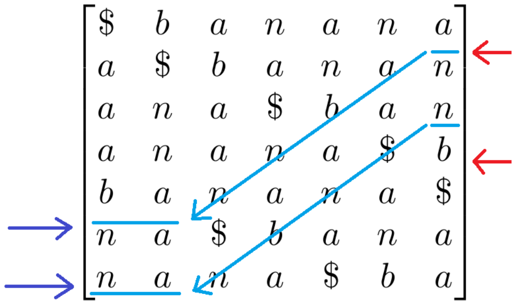
We begin searching from the back of "ana", so we first locate the cyclic shifts that contain "a" as a prefix, which corresponds to positions 2 - 4 in the BWT matrix.



$$\begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}$$

The matching prefixes for the suffix 'a'.

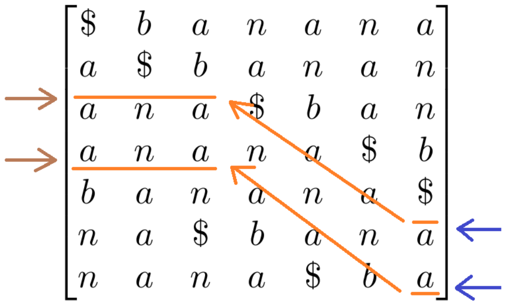
Now for the interesting part, we wish to expand the suffix 'a' to the suffix "na" and find all corresponding matches in the prefixes of the cyclic shifts of `T`. Using Property 3, we know that any 'n's that proceed the 'a's that we are looking at can be found by looking at the final column. Indeed, inspecting the final column shows us that there are two 'n's within the current range, which are the first and second occurrence of 'n'. Using the LF-mapping, we can move the current matching range to the locations of the 'n's.



$$\begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}$$

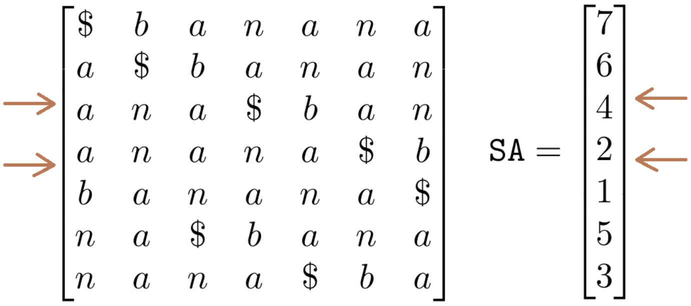
The matching prefixes for the suffix "na".

The current matching range now correctly points to the locations of the cyclic shifts that have "na" as a prefix. Finally, we want to expand the suffix "na" to become "ana" and find the new corresponding locations of the cyclic shifts that contain "ana" as a prefix. Again, we use Property 3 and look in the last column for the occurrences of 'a' that lie within the current matching range. We see that the current matching range contains the second and third 'a', so using the LF-mapping, we move the current matching range to the section containing the second and third 'a' in the first column.



The matching prefixes for the suffix “ana”.

The matching range now points to all cyclic shifts that contain “ana” as a prefix, which means we have found all occurrences of “ana” as a substring. Remember that we do not actually have the entire matrix in memory (it would take $O(N^2)$ memory to store), so to actually obtain the indices at which the matches occur, we would use the suffix array.



The locations of the matches in the suffix array of “banana$”.

Looking in the suffix array at the range we just found, we see that the substring “ana” should occur at positions 4 and 2, which is correct.

Implementing this pattern search algorithm efficiently follows most of the same ideas as the inverse transform algorithm. In order to exploit the LF-mapping, we need to precompute the counts and rank of each character in the alphabet. Additionally, we need to compute for each character c in the alphabet and each position i in the transformed text, how many occurrences of c are there from the beginning up to and including position i.

Notice that this collection of statistics is enough to compute the inverse transform as well, in fact it is a strict superset of the statistics that we computed for the inverse transform, which consisted of the same information but without each character having its own separate prefix-occurrences counter.

If we maintain the current matching range with two indices `start` and `end`, then we can observe that updating these pointers to account for the next character `P[i]` in the pattern can be done in $O(1)$ using the precomputed statistics.

```
start = rank[P[i]] + num_occurences[P[i]][start-1]
end   = rank[P[i]] + num_occurences[P[i]][end] - 1
```

The algorithm will return the range of matches [`start, end`], which can then be looked up in the suffix array if desired. If only the number of occurrences of the pattern is required and not their positions, then the suffix array is not necessary. If the range [`start, end`] is empty (ie. `start` $>$ `end`) then no matches were found.

Since the precomputed statistics allow us to update the `start` and `end` pointers in $O(1)$ for each character of the pattern `P[1..M]`, the worst case time complexity of the pattern matching algorithm is $O(M)$.

Disclaimer: These notes are intended to supplement and enrich the content provided by the lectures. They are not intended to be a complete and thorough reference and should not be used as a replacement for attending lectures.