15-451/651: Design & Analysis of Algorithms November 29, 2018
Lecture #25: Burrows-Wheeler Transform last changed: November 29, 2018

We're going to see next a different kind of operation on strings, called the Burrows-Wheeler Transform (BWT). This transformation of a string has been useful in compression — that was its original motivation, and the `b` in `bzip2` stands for this — but it also has formed the basis of a number of modern string algorithms that can operate on text using a small amount of space. The BWT was invented in 1983, and published as a tech report in 1994 by Burrows and Wheeler[1].

We will see:

- How to compute the BWT of a string.
- Intuition about why the BWT is useful in compression.
- How to invert the BWT to recover the original string.
- How to use the BWT to search a compressed string.

1 The Burrows-Wheeler Transform

Let S be a string of length n over an alphabet Σ. We will assume all of our strings end with a special, unique character \$. We will compute $bwt(S)$, which is another length n string that is a permutation of the characters of S. Just as with suffix arrays, we will first define the BWT as the output of a slow algorithm, but then see how to compute it more quickly.

A string $y \cdot x$ is a *cyclic rotation* of a string S if $S = x \cdot y$, where x, y are strings and the $\cdot$ operator indicates concatenation. For example, if $S = banana\$$, then $ana\$ban$ is a cyclic rotation of S (with $y = ana\$$ and $x = ban$). To compute the BWT (slowly), list all of the cyclic rotations of S, and then sort them lexicographically. Again, supposing $S = banana\$$:

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

The matrix of characters on the right is the *BWT matrix*. Let M_S be this matrix for a string S. We define $bwt(S)$ to be the string of characters in the last column of M_S. In the above example, $bwt(S) = annb\$aa$. Formally,

$$bwt(S) = M_S[0, n-1] \cdot M_S[1, n-1] \cdot \ldots \cdot M_S[n-1, n-1]$$

where n is the length of S.

[1]`http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf`

1.1 Why is this useful for compression?

Notice in our example above that the transformation tended to group equal characters together. There is intuition behind this, given by Burrows and Wheeler in their original paper. The *right suffix* of character s_i is the substring $s_{i+1}s_{i+2}\dots s_{n-1}$. Consider the word "the". Every place "the" occurs, the right suffix of the "t" will start with *he*.... The rows starting with these right suffixes will be near each other in the BWT matrix because the rows are sorted. For each of these, "t" will be the last character of the row, because the last character of each row comes before the start of its row in S since we are dealing with cyclic rotations. So these "t" instances will be near each other in the BWT.

Intuitively, strings that have regions of low complexity, with lots of equal characters, are easier to compress (e.g. "aaaaabaaa" can be encoded as 5a1b3a (or similar), while "abababbba" is harder to encode that way).

1.2 Recovering the original string its BWT

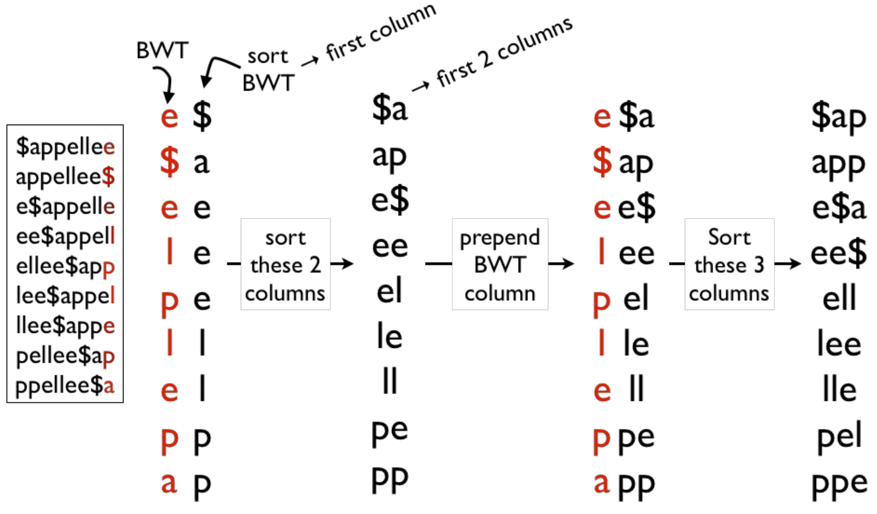
This nice compression property wouldn't be that useful if we couldn't recover the original string. Luckily, (and sort of amazingly) we can recover the string from just the BWT of the string. We'll first see a computationally slow way to do this, and later see a faster way.

We will recover the string by recovering the BWT matrix from just the BWT string. We can recover the string as the first row of this matrix (which will start with "$").

Let's see how to recover the matrix. First, note that it is easy to recover the first column of M_S. It is simply the letters of S sorted alphabetically. Let's call this first column F. Let's also call the last column (the BWT) L, with the entires of these strings denoted F_i, L_i. Because the matrix M_S consists of all the cyclic shifts, L_i *precedes* F_i in S for each i.

Denote by $L \cdot F$ the "vertical concatenation" of the L and F columns. That is, this creates a two-column matrix LF, with row i equal to $L_i \cdot F_i$. These two character strings all occur in S. In particular, they each occur at the start of some cyclic rotation. We therefore can sort LF alphabetically to recover now the first *two* columns of M_S. We now repeat, computing $sort(L \cdot LF)$ to recover the first 3 columns, and so on. After $O(n)$ iterations, we have recovered the entire matrix.

Here's an example:



This takes $O(n^3 \log n)$ time: $O(n)$ iterations, with $O(n \log n)$ comparisons to sort, each comparison taking $O(n)$ time. Obviously, this is inefficient. But it at least shows that the BWT of a string contains all of the information to reconstruct the string!

1.3 Faster computation of the BWT

Our naïve algorithm to compute the BWT created the entire BWT matrix takes time $O(n^2 \log n)$ to sort n strings (each of the cyclic rotations), each comparison taking $O(n)$ time. In fact, we can compute the BWT from the suffix array of a string in $O(n)$ time. This is because the suffix array and BWT string are very closely related.

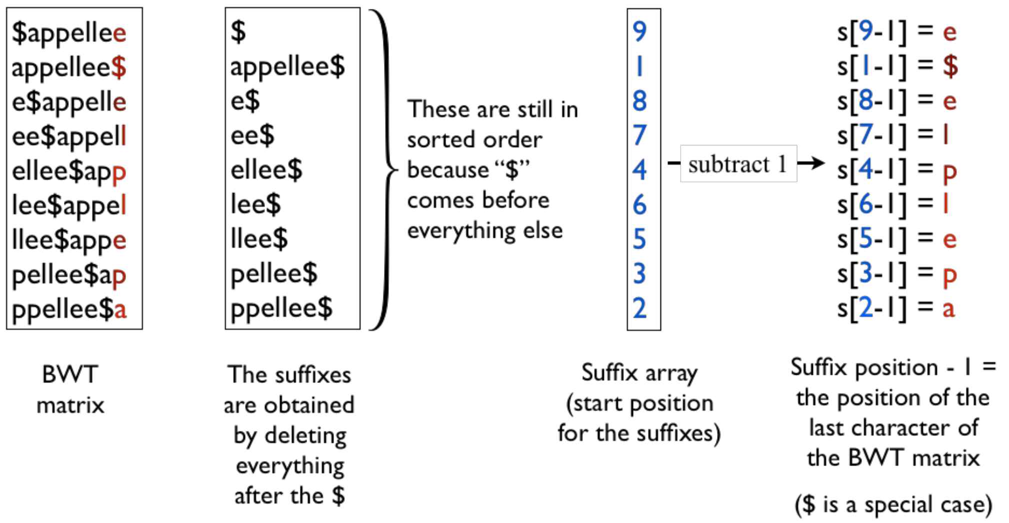
Recall that the suffix array was created (conceptually) by sorting all the suffixes of S. Consider the BWT matrix M_S but with, in each row, the characters after the "$" deleted. Each row in this new matrix represents a suffix of S, and they are still in sorted order because $ comes before every letter. So the suffix array of S is the indices of the suffixes in the order that they appear in M_S.

Consider row i of M_S. The suffix in this row is given by $SA[i]$, where SA is the suffix array of S. Now, the ith letter of the BWT is the one that comes just before this suffix: $S[SA[i] - 1]$. So $bwt(S)$ can be computed by (assuming 1-indexing for everything):

```
BWTfromSA(S, SA):
   bwt = ['x'] * n              # allocate space for the BWT
   for i = 1 ... n:
       bwt[i] = S[SA[i]-1]      # use BWT <-> SA correspondence
   return bwt
```

This is just $O(n)$ array lookups. Next lecture, we will see a fast $O(n \log n)$ algorithm to compute the suffix array of a string. So, combined, this gives a $O(n \log n)$ algorithm to create the BWT.

Here's an example (again assuming 1-indexing):



Handling the $ character is a minor special case. Since the $ character appears at the end of the first suffix, when we look at the character "before" this suffix, we really mean the last character of the string (aka $). This is a minor change to the pseudocode above.

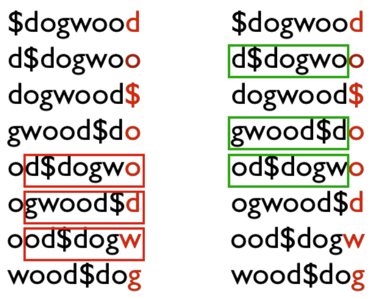
1.4 Faster inversion of the BWT

Next, let's see a way to invert the BWT that is faster. This faster algorithm uses a very important property of the BWT matrix called the LF mapping property. This property gives a relationship between the order of characters in the last and first columns of the BWT matrix.

Theorem 1 (LF mapping) *For every character $c \in \Sigma$, the ith occurrence of c in L corresponds to the same character in S as the ith occurrence of c in F, for every i.*

Proof: Define the *right context* of a character F_i to be the $n-1$ remaining columns of the BWT matrix. In F, all occurrences of c are together in a single interval, and they are ordered by their right contexts. In L, the occurrences of c can be scattered in L, but they are also ordered by their right contexts since M_S has cyclic rotations. Since they are sorted according to the same string, they are in the same order. ■

Here's a figure illustrating the above proof:



The red boxes at left show the right contexts of the "o" characters. The "o"s in this range are ordered by the strings in these boxes. At right, the green boxes show the right contexts of the "o" characters in L. Now, the "right contexts" are at the left because of the cyclic rotation. The order of the "o"s in L are determined by these right contexts. Since the "o"s are sorted by the same "key" in both L and F, they must be in the same order. Note that there are no ties in the ordering because of the $ character.

1.4.1 Inversion algorithm

The LF property lets us invert the BWT directly. We reconstruct S starting from *the end* of S. S ends with \$, which is the first character of F. Which character comes just before \$? The first character of the BWT string (aka L). So, now we know S ends with $L_0\$$. This is general: the pairs (L_i, F_i) give predecessor relationships. That is, F_i is the character immediately preceding L_i in S. This follows (again) from the fact that we are dealing with cyclic rotations.

Continuing our example: We know the string ends with L_0F_0, and now we want to find the predecessor to L_0. To do this, we need to find the correct occurrence of L_0 in F. Here's where we use the LF mapping property: If L_0 is the ith occurrence of the character L_0 in L, then the corresponding character is the ith occurrence of the character L_0 in F.

But the ith occurrence of a character c in F is easy to find if we keep an array C where $C[c]$ is the row at which the range of c characters starts in F. C can be precomputed by scanning L, counting the number of occurrences of each character. Then, the location of ith occurrence of any character c in F can be computed as $C[c] + i - 1$.

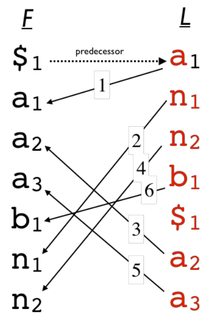
The last piece of the puzzle: when we are looking at L_j, how do we know how many instances of the character L_j occur before position j in L? In other words, define $rank(j)$ to be the number of occurrences of the character L_j that occur in positions $\leq j$. How do we know $rank(j)$? The answer is we can pre-compute these as well, creating a n-long array $rank$ where $rank[j]$ is the number of occurrences of character L_j before or at position j of L.

This leads to the following algorithm to recover S from $bwt(S)$:

```
S = ‘‘$’’
p = 0
for i = 1 .. n-1:
    predecessor = L[p]                  # char to prepend
    S = predecessor . S                 # prepend it to S
    p = C[predecessor] + rank[p] - 1    # find the right row in F
return S
```

This runs in $O(n)$ time. The arrays $rank$ and C can be computed in $O(n)$ time with a single scan of L. Notice that the algorithm only implicitly uses F — it doesn't construct the F string directly. This is possible because F has such a simple structure.

Here's an example:



The small numbers next to each character gives the $rank$ of the character. Following the arrows in the order of their numbers will spell out the original string. Each solid arrow connects the equal characters of equal ranks in L and F. For clarity, only the first predecessor arrow is show (all these arrows go horizontally left to right).

There are some tricks that can be used to speed this up even more. First, it is possible to operate on a *compressed* version of L rather than L directly. Second, it is possible to reduce the size of the rank array via subsampling. These ideas were worked out by Ferragina and Manzini in 2000. We won't have time to cover them, but they are explained well in Ben Langmead's slides[2].

2 Applications of the BWT

2.1 Search

The BWT can be directly searched to answer the question "does a string q occur in S?". To do this will require expanding our $rank$ array somewhat to make it efficient, but first let's see the main idea.

[2]`http://www.cs.jhu.edu/~langmea/resources/lecture_notes/10_bwt_and_fm_index_v2.pdf`

The first idea is to search $q = q_1, \ldots, q_m$ left-to-right. That is we will first match the last character of q, then q_{m-1}, and so on. The reason to do this is the same reason that we invert $bwt(S)$ from last character to first character: because (F_i, L_i) gives a predecessor relationship.

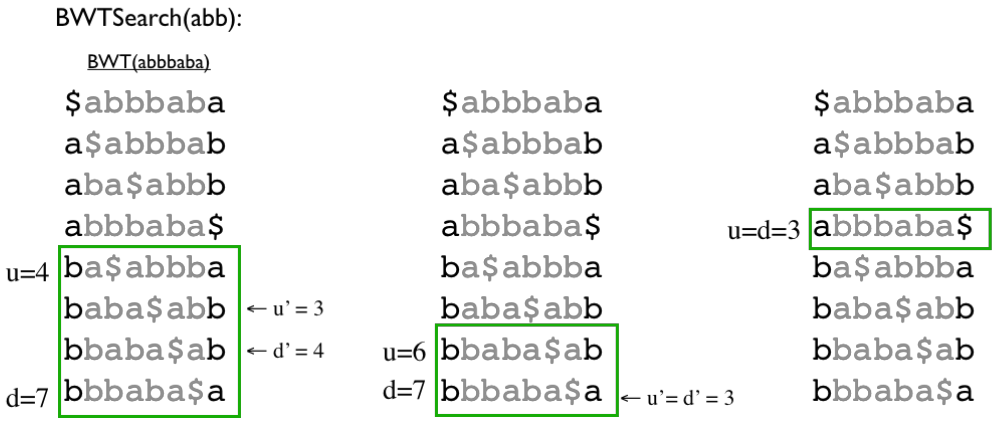
Also, as with the inversion algorithm, we are going to be conceptually alternating between looking at L and F.

To start then, we're looking for occurrences of q_m. We can find the range of these in F in the same way as before using the C array. Let $[u, d]$ be the range of rows starting with q_m. We then look at the same range in L. If q occurs in S, q_{m-1} at least once in this range.

Suppose the rank of the first occurrence of q_{m-1} in this range is u' and the rank of the last occurrence of q_{m-1} in the range is d'. Look now at the range for q_{m-1} in F. The only instances of the characters that we care about in this range are those with rank between u' and d'. Again by the LF-mapping property, since the instances of q_{m-1} in this range are all those within an interval, they are also exactly the instances within some range in F. And this range is easy to find: it's $(C[q_{m-1}] + u' - 1, C[q_{m-1}] + d' - 1)$.

We update u and d to be this range, and we just repeat, now looking for q_{m-2}, and so on. If we are able to continue this way through all of q without the range $[u, d]$ becoming empty, we know q occurs in S. If $[u, d]$ becomes empty, we report that q does not exist.

Here's an example:



Note that we do not have an easy way to find *where* q occurs. To do that, one must store extra data structures. We can however count the number of occurrences: it is the number of positions inside the final $[u, d]$ range.

One last detail: how do we efficiently find u' and d'? We could linearly scan the *rank* array, since u' and d' are just the ranks of the first and last occurrence of the character within the current range. This is too slow, however. Instead, we expand our *rank* array to have $n \times |\Sigma|$ entries, where

$$Rank(i, c) = \#\text{ occurrences of } c \text{ at positions} \leq i \text{ in } L$$

Now, at a step when we are looking for character c, we have

$$u' = Rank(u - 1, c) + 1$$
$$d' = Rank(d, c)$$

This new *Rank* array is somewhat larger than our old *rank*. Though if $|\Sigma| = O(1)$, it is still linear in the size of L. It turns out that again via subsampling, this array can be made smaller.

2.2 Compression

BWT was originally motivated by compression, and Burrows and Wheeler give in their paper a specific implementation of an algorithm to compress that takes advantage of the BWT's reordering properties. Their algorithm is based on a general "move-to-front" (MTF) compression scheme, which was introduced by Bentley, Sleator, Tarjan and Wei in 1986[3]. It turns out that the BWT is particularly suited to this kind of scheme, because MTF is adaptive: as the frequency of letters change, the code adapts, using fewer bits to encode more frequent characters.

Let's see this compression scheme. The best way to explain it is to think of the compressor C sending a string S to the decompressor D over a lossless, perfect channel. This channel might be a file on disk and the write and read steps might be separated in time indefinitely. In the MTF scheme, each of C and D maintain separate lists L_C and L_D of characters that they will update to keep in sync. To send a character x, C and D communicate in the following way:

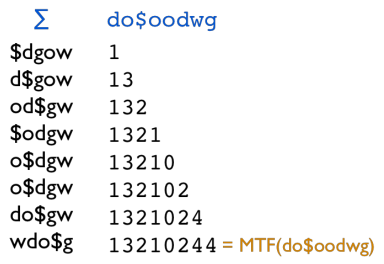
Move-to-front compression Send(x):

1. C finds x in the list L_C. If x is not in the list, append it to the end. Let i_x be the position of x in the list (1-based).
2. C sends i_x to D (using a variable number of bits — see below). If $i_x = |L_C|$, indicating that we just added x to the list, C also sends x.
3. D receives i_x. If $i_x > |L_D|$, D reads a character x from the channel and appends it to L_D.
4. D prints the character at position i_x of L_D.
5. Both C and D move x to the front of their lists.

Since C and D add characters to the end of their lists at the same time, and reshuffle their lists in the same way, L_C and L_D are the same list of characters throughout the communication. This means that D will decode the message correctly.

What does this have to do with compression? Frequently used characters will end up near the front of the lists since each time they are used they are moved to the front. We saw earlier that MTF was a 4-competitive algorithm for list update when charging for access time. That means that MTF keeps the lists accesses short for frequently accessed items. So, i_x in step 2 will generally be a small number for common characters. We take advantage of that by using a variable number of bits to send i_x: common characters will have small numbers that can be encoded in few bits. Rarer characters will have larger codes, but are rarer.

Here's an example:



Σ	do$oodwg
$dgow	1
d$gow	13
od$gw	132
$odgw	1321
o$dgw	13210
o$dgw	132102
do$gw	1321024
wdo$g	13210244 = MTF(do$oodwg)

[3] J. L. Bentley, D. D. Sleator, R. E. Tarjan, V. K. Wei, A Locally Adaptive Data Compression Scheme, Communications of the ACM-Vol. 29, No. 4, 1986

In the example above, the protocol is modified bit: when Σ is small, both C and D can start with the letters of Σ in some predetermined order.

A specific way to do this is with a "prefix code". To send $i \geq 1$, we send $\lfloor \log_2 i \rfloor$ 0's, followed by the binary encoding of i using $\lfloor \log_2 i \rfloor + 1$ bits. This gives a codeword for i of length $1 + 2\lfloor \log i \rfloor$.

Theorem 2 *If b occurs n_b times in string S of length n, then the average cost to transmit b using the MTF procedure above is:*

$$1 + 2\log(n/n_b)$$

Proof: Let n be the length of S and consider a character b that occurs n_a times in S. Let p_i be the position of b in L_C when b is transmitted for the ith time ($i = 1, \ldots, n_a$). The average cost of sending b is then (ignoring floors and ceilings for convenience):

$$avgcost(b) = \frac{1}{n_b}\sum_{i=1}^{n_b}(2\log p_i + 1) = 1 + 2\sum_{i=1}^{n_b}\frac{1}{n_b}\log p_i$$

Since log is a concave function, we have $\sum_{i=1}^{n_b}\frac{1}{n_b}\log p_i \leq \log\sum_{i=1}^{n_a}\frac{p_i}{n_b}$. In addition, $\sum_{i=1}^{n_b} p_i \leq n$ because to increment the position of b, some other character must be output and moved to the front. Combining with the above, we have

$$avgcost(b) \leq 1 + 2\log\sum_{i=1}^{n_b}\frac{p_i}{n_b} \leq 1 + 2\log(n/n_b).$$

■

The bound in the above theorem has a nice interpretation: the average position of b in the MTF list is $\leq n/n_b$. So, the average cost of sending b is less that the cost of the average. It also turns out that transmitting with the *optimum*, static prefix code must use at least $\log(n/n_b)$ bits for each character. This implies

Theorem 3 *Let $M(S)$ be the average number of bits per character used to compress S with MTF, and let $OPT(S)$ be the average number of bits per character used in an optimal, static prefix code. Then $M(S) \leq 2OPT(S) + 1$.*