Lecture 19: Burrow-Wheeler Transform

Sam McCauley

November 19, 2024

Williams College

Admin

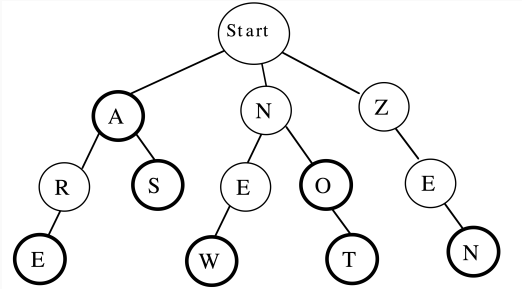
- Jeremy Fineman visiting Friday!
  - Talking about: his improvement to the classic Bellman-Ford Dynamic Programming algorithm
  - First improvement since 1958!
  - I've been very excited about this talk all semester
- Final project proposals by email on Thursday.
- Schedule updated on website. Probably no Van Emde Boas trees :(
- Any questions?

Reflections on course so far

- Part 1: Time vs space
- Part 2: Randomization
- Part 3: LP, ILP, MIP

Part 4: Strings

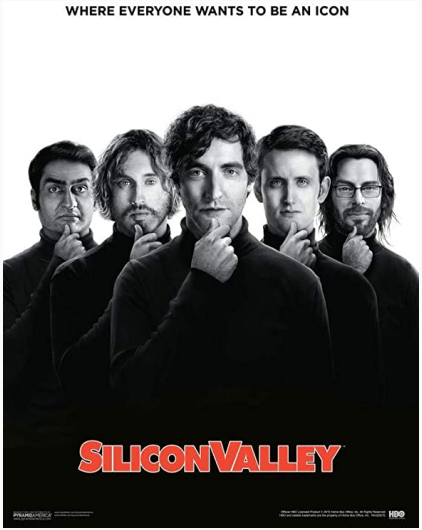
What is this part of the course?



The “Lexicon lab” in 136

- Previously in this course we’ve looked at how to *solve problems*
- This section: more about how to *handle data*
- Focus on strings—tons of applications; lots of really cool algorithms research

Focus: compression



- Take data and make it smaller
- Important! (Though sometimes overstated. . .)
- Nice self-contained topic to start before Thanksgiving

Compression: Our goals

- Take a string s, map it to a string $m = c(s)$ (using a compression function c)
- There exists a decryption function d, such that for all s, $d(c(s)) = s$
  - *Lossless* compression: we want to be able to recover the exact string
- Goal: make the string smaller. Want $|m| \leq |s|$.

Bad news: lossless compression is not possible

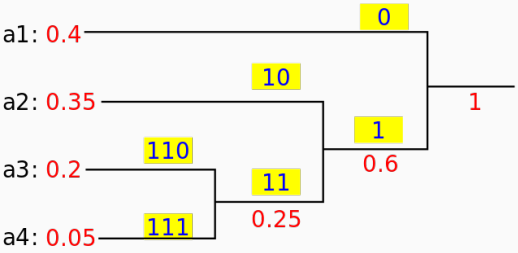
Proof sketch:

- Let's say we want to compress all binary strings of length $\leq \ell$ (in other words: map all to a string of length $\leq \ell - 1$).
- Each of the $2^{\ell+1} - 1$ strings of length $\leq \ell$ must be mapped to some string of length $< \ell$
  - A given compressed string *m* can only be mapped to by one string (otherwise we don't know which of the original strings to recover)
- But there are only $2^{\ell} - 1$ strings of length $\leq \ell - 1$. So $\approx$ half the strings can't be compressed!

What does this mean?

- Can't *guarantee* that strings get smaller
- What we'll do instead:
  - Methods that *often* help on strings we care about!
- We probably don't want to compress an arbitrary string like `afoiewjfiowefjweoifjawepgvheufgahegieg`
- Instead, we want to compress strings that look like English text, or DNA, or something like that.
- Goal: compression methods that work well on real-world data

First try: Huffman Coding



- Assign a sequence of bits to each character
- More frequent characters get longer sequences of bits
- *Prefix-free:* allows us to greedily decode
  - No code is a prefix of another!
- There is a simple, $O(n \log n)$ time method to calculate the optimal Huffman code

Huffman Coding

Symbol	Huffman Code
[space]	111
e	010
t	1101
a	1011
...	...
j	1100001011
q	11000010101
z	11000010100

- Let's encode `zeta` and decode `10111101010`
- Useful when some characters are much more common than others
- English text? Yes! Other languages? Also yes. DNA? ...kind of.
- What compression opportunities in (say) language text is this missing out on?

Key observation for compressing much of text

- Characters are NOT independent!
- *u* after *q* is *extremely* frequent in English. But Huffman codes alone can't capture this.
- DNA (and some kinds of text) may have long sequences of the same letter.
- What can we do about this?
  - Could look at encoding *pairs* of characters. (Treat every pair of consecutive characters as a single character.)
  - Or, could use a fancier method. (Run length encoding? Keep track of common substrings? Some adaptive combination of both?)

Two methods for lossless compression

Tailor-made methods (like Lempel-Ziv and variants):

- Interesting methods, used frequently in practice
- Tons of research into making these methods efficient, effective

Other option: adjust Huffman coding to try to make it work

- How well can this do?

Some Intuition

- Let's say we have the string AAAAAAAA.... (1 million characters long)
- How can we encode this with Huffman??
- Map A to 0. Then can write 00000000.... Can we do better?
- Better encoding: something like "Write A 1 million times"
- Huffman falls short!
- Even worse if we have something like 1 million As, followed by 1 million Bs, then 1 million Cs, 1 million Ds...
- What about ABABABAB... ?

Move-to-front transform

Move-to-front: a cool first step to address some of these issues.

- Goal: preprocess the string so that long runs (and long close-to-runs) of the same character can be encoded more efficiently.
- Must be invertible (so that we can decode later)
- Does:
  - Improve performance when the same character is close to other occurrences of the same character
  - Perform well when one character is repeated a lot
- Does *NOT*:
  - Take advantage of relationships between different successive characters
  - Example: u always coming after q is no advantage at all

Move-to-front transform

Transform a string s into a string $MTF(s)$:

- Keep an list L of all possible characters. Start with L just keeping the characters in some arbitrary order.
  - For these examples: $L = \{a, b, c, \ldots, y, z\}$
  - In general, L encodes all 255 possible `char` values. Start with $L[i] = i$.
- Start with empty s'. For each $i = 1$ to $|s|$:
  - If $s[i]$ is the jth character in L, append j to s
  - Move j to the front of L.
- Return s' as $MTF(s)$ when done
- Let's do a couple examples on the board: `banana`, `abracadabra`

Move-to-front transform: decode

Transform a string $s' = MTF(s)$ into the original string s:

- Goal: recover L at each time step used when encoding
- Start with same L
- Start with empty s. For each $i = 1$ to $|s'|$:
  - If $s'[i] = j$, then write $L[j]$ to s
  - Move j to the front of L.
- Let's decode the board examples.

Move-to-front discussion

- Move to front transforms sequences of nearby characters into common characters
- Plan: to encode a string s, we first calculate $MTF(s)$, and do Huffman coding on that
- To decode, first Huffman decode the string. This gives us $MTF(s)$. Use the above method to recover s
- Can greatly improve Huffman coding performance if characters are close together
- In the *worst case* might not improve anything. (Could even make performance a good amount worse—when?)

Burrows-Wheeler Transform

Where we are

- Lempel-Ziv: Fairly technical method to take advantage of common substrings/correlations between sequences characters
  - `zip` uses a combination of Lempel-Ziv and Huffman coding
- What we've seen: move-to-front and Huffman to take advantage of consecutive characters
- What we'd like: a simple, reversible preprocessing method that makes common *subsequences* into common *characters*.
- We have MTF: so turning common subsequences into *nearby* repetitions of the same character is enough

Burrows-Wheeler Transform (BWT)

- Invented around 1995
- Turns common subsequences into sequences of nearby characters
- (This is a super weird thing to be able to do. We'll look at a few examples to try to get some intuition about it.)
- Reversible!

BWT Game Plan

To compress a string s:

1. Use BWT to obtain a string $s_b = BWT(s)$. s_b has the property that common subsequences of s correspond to nearby characters of s_b
2. Use MTF to obtain a string $s_m = MTF(s_b)$. s_m has the property that nearby characters of s_b (and therefore common subsequences of s) correspond to common characters in s_m
3. Use Huffman coding on s_m to obtain a final compressed string s_h. Common characters in s_m require few bits to output.

All of the above is reversible, so this is a method for lossless compression.

- Believe it or not: this method *outperforms* fancier state of the art compression methods in some circumstances
- This is exactly what `bzip2` does.

What does Burrows-Wheeler Transform do?

Let's talk about performing BWT on a string s of length n. Let's assume that s ends with a special character \$ (this will be helpful for us)

- Goal: take the *context* of each character into account
- How many other characters should we look at? 1? 2?
- Silly point: we'll do best if we consider the entire $n - 1$ characters surrounding each character
- What does it even mean to take the $n - 1$-character context of a string into account?

BWT: Looking at the context of a character

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

- Take all n *circular suffixes* of the string (wrap around from beginning)
- The “context” of each character is the $n - 1$ characters following it

What does this give us?

a $ b a n a n

- For each character of the string: we look at all characters that follow it
- What can we glean from the characters after a given character?
- If a substring appears a lot, it will result in a lot of similar (how?) sequences of n characters
- Example: in English text, almost every q will be followed by a u.
- In banana, almost every a is followed by an n; every n is followed by an a
- Recall: group characters with similar contexts together. So let's sort the characters using the $n-1$ characters that follow them

First, an observation

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

- The context of a character (the $n-1$ characters following it) are the contents of the row that the character *ends*
- So: let's look at the *last* column of this table

The Burrows-Wheeler Transform

$	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

- First, sort the rotated strings lexicographically
- Take the *last character* of each rotation
- This is the BWT of the string
- BWT(`banana`) = `annb$aa`

OK What's going on here?

- This is efficient!?
- This is reversible!?

What we *do* have:

- Characters will wind up next to each other if they are followed by lexicographically similar ($n - 1$-character) strings
- So: if all qs are followed by a u, then EVERY q will wind up in the portion of the BWT corresponding to suffixes beginning with u. Unclear how good this is...

One board example

- What is the BWT of dogwood?
- Hopefully we got: do$oodwg
- More interesting example: what if we take the BWT of the first line of `chromosome1.txt` (human DNA)

Still to show: reversible and efficient

Let's start with reversible

- On the board: let's say we have a BWT transformed string; the result is `e$elplepa`
- What do we know based on how the BWT works? Can I recover ANYTHING about the original string? Can I recover anything about the original table?
- Result:
- `appellee$`

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table
- This gives us *all pairs* of characters. If we sort THOSE, the second character of the result gives the second column of the table
- So on until the table is recovered
- Our string: row ending with $

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table

a
n
n
b
$
a
a

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table

$	a
a	n
a	n
a	b
b	$
n	a
n	a

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table
- This gives us *all pairs* of characters. If we sort THOSE, the second character of the result gives the second column of the table

$	b	a
a	$	n
a	n	b
a	n	n
b	a	$
n	a	a
n	a	a

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table
- This gives us *all pairs* of characters. If we sort THOSE, the second character of the result gives the second column of the table
- So on until the table is recovered
- Our string: row ending with $

$	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

Efficiency

How much time and space does *encoding* take for a string of length n? First, encoding:

- Filling out the table: $O(n^2)$ time and space.
- Sorting the table?
  - $O(n)$ time to compare two items
  - $O(n \log n)$ comparisons
  - *Total:* $O(n^2 \log n)$ time.

Efficiency

How much time and space does this method take now for a string of length n?
Now, decoding:

- Recover one column at a time
- To recover a column: sort (last column) appended to current columns we have
  - $O(n)$ time to compare two items
  - $O(n \log n)$ comparisons
  - This means $O(n^2 \log n)$ time *per column*
  - $O(n^3 \log n)$ time overall

This is terrible! But there's a ton of redundancy here. Can we do better?

Efficient BWT Encoding

- Using a clever method, can get much faster time BWT encoding
- Need another data structure: suffix array

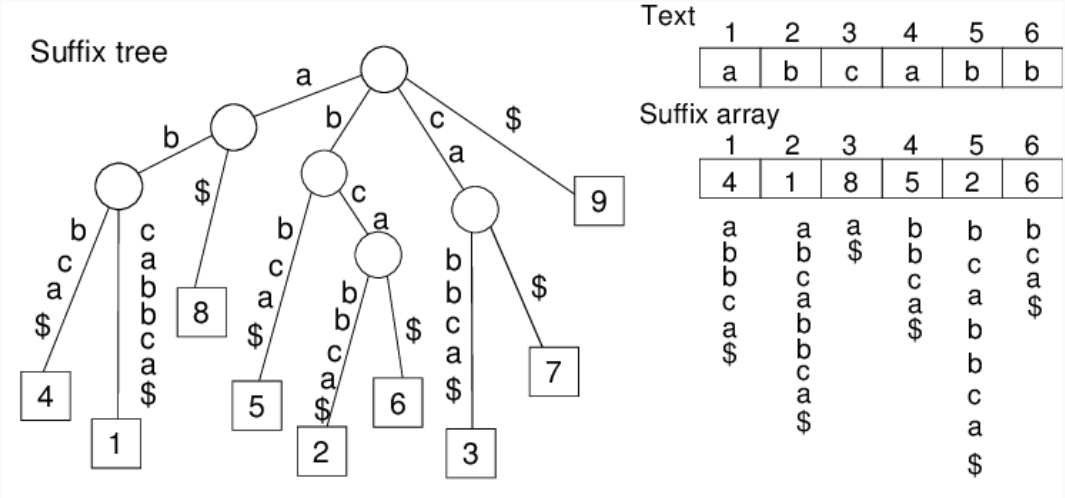
Suffix Array

Any string of length n has a suffix array A of n indices:

- $A[i]$ contains the index of the ith suffix of s in sorted order.
- Example: suffix array for `banana$` is:
- `6 5 3 1 0 4 2`
- Very similar to what we want for BWT. (We'll talk about that in a second). How fast do you think one can compute this?
- Answer: can do this in $O(n)$ time for constant-size alphabet. (*Faster* than sorting.)
- We'll talk about how to get $O(n \log n)$ time for this next Tuesday

Suffix Array Applications

- *Far* wider use than just creating BWT
- In short: a suffix array is compressed, but allows trie-like operations



Suffix Array to BWT

- Let's say you are given the suffix array for the string.
- How can you get the BWT?
- Let's do it one character at a time.
- The ith character of the string is the *last column character* corresponding to the ith suffix in sorted order
- So: BWT$[i] = s[j]$, where $j = SA[i] - 1$. (Watch out for negative indices)
- Linear time method to calculate the BWT!
- Any practical problems with this methdology?
  - Very cache-inefficient if our string is large enough for that to be an issue

Where we are

- If you're given a suffix array, can calculate the BWT in a simple linear scan. No extra space (beyond the original string and the suffix array)
- Now: can we reverse the BWT quickly as well?
- Let's fill out the BWT in *reverse order*. What characters can we fill in?
  - Last character must be $ (easy)
  - Previous character is the first character in the row containing $
- Key observation: let's say we just wrote a character in the last column. We want to find the character before that in the original string. If we can find that character in the first column, we know the next character to write (as it's the corresponding last-column-character).

Quickly inverting the BWT

Lemma

Consider a character c in the last column of the BWT table. The order of all occurrences of c in the last column is the same as the order of all occurrences of c in the first column of the table.

Proof: Let's say the ith instance of c is followed by a (circular) suffix s_i in the original string s. Then row i of the BWT table consists of s_i concatenated with c. Therefore, the order of all instances of c in the *last column* of the table is exactly the same as the lexicographic order of the s_i.

Quickly inverting the BWT

Lemma

Consider a character c in the last column of the BWT table. The order of all occurrences of c in the last column is the same as the order of all occurrences of c in the first column of the table.

Proof (contd):

Now let's look at the *first column*. Since the first row is sorted lexicographically, all instances of c in the first column are adjacent rows in the BWT table. Furthermore, the row beginning with the ith instance of c consists of c concatenated with s_i. But then, the order of the instances is the same as the lexicographic order of the s_i.

So the orders are the same!

Diagram of proof

Diagram of proof

Quickly inverting the BWT

Let's start deducing the original string from back to front. Let's use the example do$oodwg.

- What's the last character? What's the second to last character?
- Idea: keep a pointer to the index in the BWT we just wrote. How can we use that to deduce the next index we're writing?
- Rephrase: how can we deduce the next character we're writing? As: how can we deduce what row it is in in the first column ?
  - Let's say we just wrote the *i*th character in the *first* column of the BWT table
  - Its previous character is the *i*th character in the *last* column—in other words, the *i*th character in the BWT

Quickly Inverting the BWT

- When we write a character, goal is: find out where it was in the first column. If we get that we're done
- Idea:
- We just wrote the *j*th character in BWT; let's say it's character *c*
- Let's say there are ℓ occurrences of *c* earlier in the BWT
- Then we're looking for the ℓth *c* in the first column
- Example: invert e$elplepa using this method.

Data structures for inversion

How can we quickly answer: “I’m at index *j* of the BWT; I see character *c*. How many instances of *c* are there at indices *j* and earlier in the BWT?”

- Precompute with a linear scan!
- Keep track of how many of each character seen so far. Write the value for each character of the BWT. Call this array *rank*.

Data structures for inversion (contd.)

How can we quickly answer: “In the first column of the BWT table, in what row does the *i*th occurrence of character *c* occur?”

- The *c*s are all clustered together in the first row. Enough to tell where the grouping begins.
- For each character *c*, keep track of how many characters before *c* (in lexicographic order) occur in the entire string *s*
- Linear time preprocessing: first, get counts of each character in the string. Then, sum successive entries to get the count of all entries before the character. Call this array *C*

Algorithm for Inversion

First, write $ in the last slot. Find the index of $ in the BWT; call this `index`. Then, do the following for $n - 1$ iterations:

- Find the row r in the BWT whose first column contains the character c we just wrote:
  - Calculate how many instances of this character occur earlier in the BWT; this is $rank[index]$
  - Find the location where we begin writing character c in the first column: this is $C[c]$
  - Therefore, we are looking for $r = rank[index] + C[c]$.
- Prepend BWT[r] to s.
- Update index = r

Our final compression approach

- First, BWT the string using the above (causes characters that appear in similar contexts to be grouped together)
- Then use MTF on the result (causes characters that are grouped together to result in a large number of low-value characters)
- Then use Huffman coding on the result of that (characters that appear often can be written with a very small number of bits).

Practical considerations

- This is still pretty slow: why?
- Cache-inefficient! Each encode/decode step requires random array access. On large enough strings this is an L3 miss for EACH characterwe encode/decode
- How can we avoid this?
- In practice: break into decent-sized blocks that fit into L3 cache! BWT each individually