What is a transform?

- Multiply two near-zero numbers, what happens?
  - Add their logarithms: log(a)+log(b) = log(ab), invertible
  - What is log of 10^{-13}? Benefits of transform?
- What is FFT: Fast Fourier Transform?
  - O(n log n) method for computing a Fourier Transform
  - Better than $O(n^2)$, huge difference for lots of data points
  - Shazam? how shazam might work
- Feature extraction from images: faces, edges, lines, …
  - Hough transform
- Wavelet transforms do something too, but …

Burrows Wheeler Transform

- Michael Burrows and David Wheeler in 1994, BWT
- By itself it is NOT a compression scheme
  - It's used to preprocess data, or *transform* data, to make it more amenable to compression like Huffman Coding
  - Huff depends on redundancy/repetition, as do many compression schemes
- http://en.wikipedia.org/wiki/Burrows-Wheeler_transform
- http://marknelson.us/1996/09/01/bwt
- Main idea in BWT: transform the data into something more compressible and make the transform fast, though it will be slower than no transform
  - TANSTAAFL (what does this mean?)

David Wheeler (1927-2004)

- Invented subroutine
- "Wheeler was an inspiring teacher who helped to develop computer science teaching at Cambridge from its inception in 1953, when the Diploma in Computer Science was launched as the world's first taught course in computing."

Mike Burrows

He's one of the pioneers of the information age. His invention of Alta Vista helped open up an entire new route for the information highway that is still far from fully explored. His work history, intertwined with the development of the high-tech industry over the past two decades, is distinctly a tale of scientific genius.

http://www.stanford.edu/group/gpj/cgi-bin/drupal/?q=node/60

BWT efficiency

- BWT is a block transform – requires storing n copies of the file with time `O(n log n)` to sort copy (file has length n)
  - We can't really do this in practice in terms of storage
  - Instead of storing n copies of the file, store one copy and an integer index (break file into blocks of size n)
- But sorting is still `O(n log n)` and it's actually worse
  - Each comparison in the sort looks at the entire file
  - In normal sort analysis the comparison is `O(1)`, strings are small
  - Now we have key comparison of `O(n)`, so sort is actually…
  - $O(n^2 \log n)$, why?

BWT at 10,000 ft: big picture

- Remember, goal is to exploit/create repetition (redundancy)
  - Create repetition as follows
  - Consider original text: ***duke blue devils.***
  - Create n copies by shifting/rotating by one character

```
0: duke blue devils.       9:  devils.duke blue
1: uke blue devils.d       10: devils.duke blue
2: ke blue devils.du       11: evils.duke blue d
3: e blue devils.duk       12: vils.duke blue de
4:  blue devils.duke       13: ils.duke blue dev
5: blue devils.duke        14: ls.duke blue devi
6: lue devils.duke b       15: s.duke blue devil
7: ue devils.duke bl       16: .duke blue devils
8: e devils.duke blu
```


BWT at 10,000 ft: big picture

- Once we have n copies (but not really n copies!)
  - Sort the copies
  - Remember the comparison will be O(n)
  - We'll look at the last column, see next slide
    - What's true about first column?

```
4:   blue devils.duke      13: ils.duke blue dev
9:   devils.duke blue      2:  ke blue devils.du
16: .duke blue devils      14: ls.duke blue devi
5:  blue devils.duke       6:  lue devils.duke b
10: devils.duke blue       15: s.duke blue devil
0:  duke blue devils.      7:  ue devils.duke bl
3:  e blue devils.duk      1:  uke blue devils.d
8:  e devils.duke blu      12: vils.duke blue de
11: evils.duke blue d
```


|ees .kudvuiblllde| | .bddeeeikllsuuv|

```
4:   blue devils.duke
9:   devils.duke blue
16: .duke blue devils
5:  blue devils.duke
10: devils.duke blue
0:  duke blue devils.
3:  e blue devils.duk
8:  e devils.duke blu
11: evils.duke blue d
13: ils.duke blue dev
2:  ke blue devils.du
14: ls.duke blue devi
6:  lue devils.duke b
15: s.duke blue devil
7:  ue devils.duke bl
1:  uke blue devils.d
12: vils.duke blue de
```

- Properties of first column
  - Lexicographical order
  - Maximally 'clumped' why?
  - From it, can we create last?
- Properties of last column
  - Some clumps (real files)
  - Can we create first? Why?
- See row labeled 8:
  - Last char precedes first in original! True for all rows!
- Can recreate everything:
  - Simple (code) but hard (idea)

What do we know about last column?

- Contains every character of original file
  - Why is there repetition in the last column?
  - Is there repetition in the first column?
- Keep the last column because we can recreate the first
  - What's in every column of the sorted list?
  - If we have the last column we can create the first
    - Sorting the last column yields first
  - We can create every column which means if we know what row the original text is in we're done!
    - Look back at sorted rows, what row has index 0?

BWT from a 5,000 ft view

- How do we avoid storing n copies of the input file?
  - Store once with *index* of what the first character is
  - 0 and "duke blue devils." is the original string
  - 3 and "duke blue devils." is "e blue devils. du"
  - What is 7 and "duke blue devils."
- You'll be given a class Rotatable that can be sorted
  - Construct object from original text and *index*
  - When compared, use the *index* as a place to start
  - Rotatable can report the last char of any "row"
  - Rotatable can report its *index* (stored on construction)

BWT 2,000 feet

- To transform all we need is the last column and the row at which the original string is in the list of sorted strings
  - We take these two pieces of information and either compress them or transform them further
  - After the transform we run Huff on the result
- We can't store/sort a huge file, what do we do?
  - Process big files in chunks/blocks
    - Read block, transform block, Huff block
    - Read block, transform block, Huff block…
    - Block size may impact performance

Toward BWT from zero feet

- First look at code for HuffProcessor.compress
  - Tree already made, `preprocessCompress`
  - How `writeHeader,writeCompressedData` work?

```
public int compress(InputStream in, OutputStream out) {
   BitOutputStream bout = new BitOutputStream(out);
   BitInputStream bin = new BitInputStream(in);
   int bitCount = 0;

   myRoot = makeTree();
   makeMapEncodings(myRoot,"");
   bitCount += writeHeader(bout);
   bitCount += writeCompressedData(bin,bout);
   bout.flush();
   return bitCount;
}
```

BWT from zero feet, part I

- Read a block of data, transform it, then huff it
  - To huff we write a magic number, write header/tree, and write compressed bits based on Huffman encodings
  - We already have huff code, need to use on a transformed bunch of characters rather than on the input file
- So process input stream by passing it to BW transform which reads a chunk and returns `char[]`, the last column
  - A char is a 16-bit, unsigned value, we only need 8-bit value, but use char because we can't use byte
    - In Java byte is signed, -128,.. 127
    - What does all that mean?

Use what we have, need new stream

- We want to use existing compression code we wrote before
  - Read a block of 8-bit/chunks, store in char[] array
  - Repeat until no more blocks, *last block not full*?
  - Block as `char[]`, treat as stream and feed it to Huff
    - Count characters, make tree, compress
- We need an Adapter, something that takes char[] array and turns it into an InputStream which we feed to Huff compressor
  - ByteArrayInputStream, turns byte[] to stream
  - We can store 8-bit chunks as bytes for stream purposes

ByteArrayInputStream and blocks

```
public int compress(InputStream in, OutputStream out) {
        BitOutputStream boout = new BitOutputStream(out);
        BitInputStream bin = new BitInputStream(in);
         int bitCount = 0;
    BurrowsWheeler bwt = new BurrowsWheeler();
    while (true){
        char[] chunk = bw.transform(bin);
        if (chunk.length < 1) break;
        chunk = btw.mtf(chunk);
        byte[] array = new byte[chunk.length];
        for(int k=0; k < array.length; k++){
            array[k] = (byte) chunk[k];
        }
        ByteArrayInputStream bas =
             new ByteArrayInputStream(array);
        preprocessInitialize(bas);
       myRoot = makeTree();
         makeMapEncodings(myRoot,"");
         BitInputStream blockBis = new BitInputStream(new ByteArrayInputStream(array));
         bitCount += writeHeader(bout);
         bitCount += writeCompressedData(blockBis,bout);
    }
    bout.flush(); return bitCount;
}
```


How do we untransform?

- Untransforming is very slick
  - Basically sort the last column in O(n) time
  - Run an O(n) algorithm to get back original block
- We sort the last column in O(n) time using a *counting sort*, which is sometimes one phase of radix sort
  - Call sort: easier to code and a good first step
  - The counting sort leverages that we're sorting "characters" --- whatever we read when doing compression which is an 8-bit chunk
  - How many different 8-bit chunks are there?

Counting sort

- If we have an array of integers all of whose values are between 0 and 255, how can we sort by counting number of occurrences of each integer?
  - Suppose we have 4 occurrences of one, 1 occurrence of two, 3 occurrences of five and 2 occurrences of seven, what's the sorted array? (we don't know the original, just the counts)

1	1	1	1	2	5	5	5	7	7

  - What's the answer? How do we write code to do this?
- More than one way, as long as O(n) doesn't matter really

Another transform: Move To Front

- In practice we can introduce more repetition and redundancy using a Move-to-front transform (MTF)
  - We're going to compress a sequence of numbers (the 8-bit chunks we read, might be the last column from BWT)
  - Instead of just writing the numbers, use MTF to write
- Introduce more redundancy/repetition if there are runs of characters. For example: consider "AAADDDFFFF"
  - As numbers this is 97 97 97 100 100 100 102 102 102
  - Using MTF, start with `index[k] = k`
    - `0,1,2,3,4,…,96,97,98,99,…,255`
  - Search for 97, initially it's at index[97], then MTF
    - `97,0,1,2,3,4,5,…, 96,98,99,…,255`

More on why MTF works

- As numbers this is 97 97 97 100 100 100 102 102 102
  - Using MTF, start with `index[k] = k`
  - Search for 97, initially it's at index[97], then MTF
    - `97,0,1,2,3,4,5,…,96,98,99,100,101,…`
  - Next time we search for 97 where is it? At 0!
- So, to write out 97 97 97 we actually write 97 0 0, then we write out 100, where is it? Still at 100, why? Then MTF:
  - `100,97,0,1,2,3,…96,98,99,101,102,…`
- So, to write out 97 97 97 100 100 100 102 102 102 we write:
  - `97, 0, 0, 100, 0, 0, 102, 0, 0`
  - Lots of zeros, ones, etc. Thus more Huffable, why?

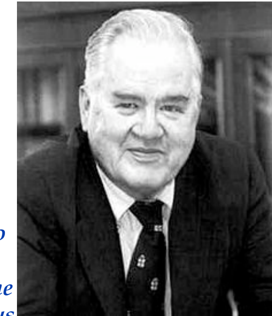
Complexity of MTF and UMTF

- Given n characters, we have to look through 256 indexes (worst case)
  - So, `256*n`, this is …. `O(n)`
  - Average case is much better, the whole point of MTF is to find repeats near the beginning (what about MTF complexity?)
- How to untransform, undo MTF, e.g., given
  - `97, 0, 0, 100, 0, 0, 102, 0, 0`
- How do we recover AAADDDFFF (97,97,97,100,100,…102)
  - Initially `index[k] = k`, so where is 97? `O(1)` look up, then MTF

Burrows Wheeler Summary

- Transform data: make it more "compressable"
  - Introduce redundancy
  - First do BWT, then do MTF (latter provided)
  - Do this in chunks
  - For each chunk array (after BWT and MTF) huff it
- To uncompress data
  - Read block of huffed data, uncompress it, untransform
  - Undo MTF, undo BWT: this code is given to you
  - Don't forget magic numbers

John Tukey: 1915-2000



- Cooley-Tukey FFT
- Bit: Binary Digit
- Box-plot
- "software" used in print

Far better an approximate answer to the **right** *question, which is often vague, than an* **exact** *answer to the wrong question, which can always be made precise.*

The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.