Burrows-Wheeler
Transform

Recap

- HW due next week
  final implementation project
- In today & tomorrow

Today: The Burrows-Wheeler Transform

Idea: Data Compression

First, recall the steps:

- Construct all circular permutations of the input
- Sort them
- Store last column after sorting, plus the index of original string in sorted list

Classic banana example:

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

EOM char is smallest

Output:

annb$aa

First - compression:

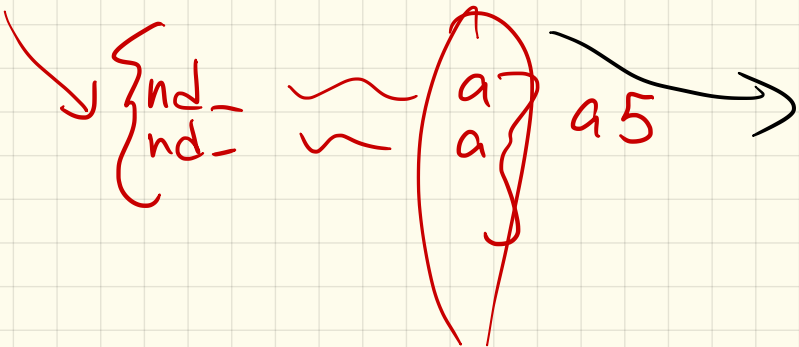
Why does this make compression easier?

Well - groups common things together!

Simple example: if "and_" is common but randomly distributed through the text, harder to find.

After the BWT, all of these will be a row ending with "a" and starting with "nd_".

Simple compression algorithms can then be used



Example: Run length encoding

Idea: Replace "aaaaa" by "a5".

Given long runs, improvement is obvious.

simpler & faster

Another: Huffman codes:

Do frequency analysis, & make more common characters have a smaller encoding string.

(Uses prefix free codes & Huffman trees - often taught in data structures.)

End product: Tools like bzip - highly effective for text compression.

Back to BWT:

Again, though - if you're paying attention, <u>any</u> compression could do this.

Also - any sort would make compression faster.

Key aspect of BWT:

It is invertible!

(+ fast - later)

How?

Well, last column contains all of the characters, just in the wrong order:

ANNB$AA

Sort, and you recover 1st column:

$AAABNN

Between these, now have all pairs:

last ANNB$AA

1st $AAABNN

⇒

A$ ✓
NA ✓
NA ✓
BA ✓
$B ✓
AN ✓
AN ✓

BANANA$

sort again

	last
$B	A
A$	N
AN	N
AN	B
BA	$
NA	A
NA	A

1st & 2nd columns!

Since we know the last column, can get all triples

Continue this, + output row ending with $.

	last
$B	A
A$	N
AN	N
AN	B
BA	$
NA	A
NA	A

Triples

A$B ✓
NA$ ✓
NAN ✓
BAN ✓
$BA ✓
ANA ✓
ANA ✓

sort

4-tuples

1	2	3	last
$	B	A	A
A	$	B	N
A	N	A	N
A	N	A	B
B	A	N	$
N	A	$	A
N	A	N	A

Code is fairly simple, although complexity analysis can get interesting

bwt.py

```python
#! /usr/bin/env python
"""
A simple Burrows-Wheeler transform function in python.

Algorithm presented in:
Burrows M, Wheeler DJ: A Block Sorting Lossless Data Compression Algorithm.
    Technical Report 124. Palo Alto, CA: Digital Equipment Corporation; 1994.

USAGE: bwt.py [-h] [-i INDEX] STRING
"""

import argparse

def bw_transform(s):
    n = len(s)
    m = sorted([s[i:n]+s[0:i] for i in range(n)])
    I = m.index(s)
    L = ''.join([q[-1] for q in m])
    return (I, L)

from operator import itemgetter

def bw_restore(I, L):
    n = len(L)
    X = sorted([(i, x) for i, x in enumerate(L)], key=itemgetter(1))

    T = [None for i in range(n)]
    for i, y in enumerate(X):
        j, _ = y
        T[j] = i

    Tx = [I]
    for i in range(1, n):
        Tx.append(T[Tx[i-1]])

    S = [L[i] for i in Tx]
    S.reverse()
    return ''.join(S)
```

$n^2 \log n^2$

input: $n \times n$ (if n is length of the string)

Runtime: $O(n^2 \log n)$, space $O(n^2)$

BWT + suffix tree connection

Let
s = appellee$

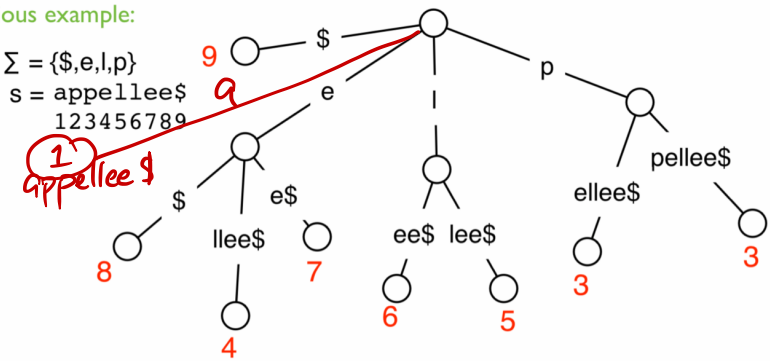
BWT order is same as suffix tree order!

BWT matrix	The suffixes are obtained by deleting everything after the $
$appellee	$
appellee$	appellee$
e$appelle	e$
ee$appell	ee$
ellee$app	ellee$
lee$appel	lee$
llee$appe	llee$
pellee$ap	pellee$
ppellee$a	ppellee$

BWT = e$elplepa

Ordered suffix tree for previous example:

Σ = {$,e,l,p}
s = appellee$
123456789



↳ build another rep: suffix array

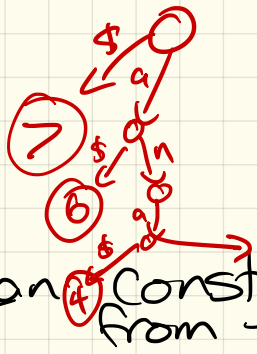
Suffix array:

i	1	2	3	4	5	6	7
$S[i]$	b	a	n	a	n	a	$

Suffix	i
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

Suffix	i
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

Then:
Suffix array is the starting position of sorted suffixes:



i	1	2	3	4	5	6	7
$A[i]$	7	6	4	2	1	5	3

Can construct Suffix array from the suffix tree...

This gives a new, faster way to compute BWT:

use suffix tree!

Example: s= appellee$

BWT matrix
$appellee
appellee$
e$appelle
ee$appell
ellee$app
lee$appel
llee$appe
pellee$ap
ppellee$a

BWT

e
$
e
l
p
l
e
p
a

Relable s to get indices

appellee$

1 2 3 4 5 6 7 8 9

Suffixes

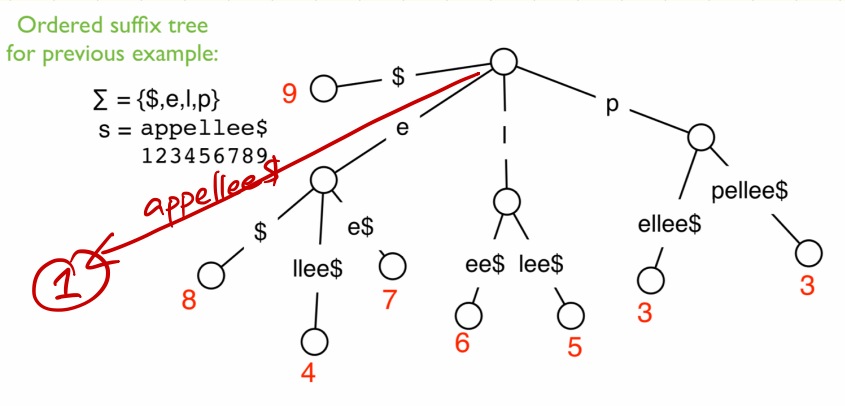
$
appellee$
e$
ee$
ellee$
lee$
llee$
pellee$
ppellee$

Suffix array: 9 1 8 7 4 6 5 3 2

Subtract 1 from each: 8 7 6 3 5 4 2 1

Get position of entry i in BWT

Suffix arrays + suffix trees:



appellee$

1 2 3 4 5 6 7 8 9

BWT matrix	The suffixes are obtained by deleting everything after the $	Suffix array (start position for the suffixes)	Suffix position - 1 = the position of the last character of the BWT matrix
$appellee	$	9	s[9-1] = e
appellee$	appellee$	1	s[1-1] = $
e$appelle	e$	8	s[8-1] = e
ee$appell	ee$	7	s[7-1] = l
ellee$app	ellee$	4	s[4-1] = p
lee$appel	lee$	6	s[6-1] = l
llee$appe	llee$	5	s[5-1] = e
pellee$ap	pellee$	3	s[3-1] = p
ppellee$a	ppellee$	2	s[2-1] = a

These are still in sorted order because "$" comes before everything else

− subtract 1 →

($ is a special case)

So, new way to compute:

Compute SA (suffix tree – $O(n)$)

$$BWT[i] = \begin{cases} T[SA[i]-1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases}$$

suffix array

original string

(depending on indexing)

Ex:

$ a b a a b a	
a $ a b a a b	
a a b a $ a b	
a b a $ a b a	
a b a a b a $	
b a $ a b a a	
b a a b a $ a	

BWM(T)

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

SA(T)

Runtime:

- Easy $O(n^2 \log n)$ algorithm:
  compute circular permutations
  sort
  read last row

- Some direct + space efficient $O(n)$-algorithms!
  (won't cover)

  built from:

- Use Suffix array to get BWT.
  $O(n)$ time + space
  (but constants can be large)

Optimality:

BWT is in fact very good at compression.

Empirical entropy: defined in terms of the # of occurences of each symbol or group of them.

k^{th} order empirical entropy gives a lower bound on achievable compression, depending on k symbols before it

[Manzini 2001] showed BWT is optimal (up to constant factor) for any k

Uses is bioinformatics:

Speeding up alignment!
(reduces memory requirement)

Bowtie

Software

Highly accessed | Open access

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome

Ben Langmead*, **Cole Trapnell**, **Mihai Pop** and **Steven L Salzberg**

* Corresponding author: Ben Langmead langmead@cs.umd.edu

Author Affiliations

Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA
For all author emails, please log on.

Genome Biology 2009, **10**:R25 doi:10.1186/gb-2009-10-3-r25

The electronic version of this article is the complete one and can be found online at:
http://genomebiology.com/2009/10/3/R25

BWA

Fast and accurate short read alignment with Burrows–Wheeler transform

Heng Li and Richard Durbin*

Author Affiliations

* To whom correspondence should be addressed.

Received February 20, 2009.
Revision received May 6, 2009.
Accepted May 12, 2009.

Bioinformatics (2009) 25(14):1754-1760.

Bowtie Performance

Varying read length using Bowtie, Maq and SOAP

Length	Program	CPU time	Wall clock time	Peak virtual memory footprint (megabytes)	Bowtie speed-up	Reads aligned (%)
36 bp	Bowtie	6 m 15 s	6 m 21 s	1,305	-	62.2
	Maq	3 h 52 m 26 s	3 h 52 m 54 s	804	36.7×	65.0
	Bowtie -v 2	4 m 55 s	5 m 00 s	1,138	-	55.0
	SOAP	16 h 44 m 3 s	18 h 1 m 38 s	13,619	216×	55.1
50 bp	Bowtie	7 m 11 s	7 m 20 s	1,310	-	67.5
	Maq	2 h 39 m 56 s	2 h 40 m 9 s	804	21.8×	67.9
	Bowtie -v 2	5 m 32 s	5 m 46 s	1,138	-	56.2
	SOAP	48 h 42 m 4 s	66 h 26 m 53 s	13,619	691×	56.2
76 bp	Bowtie	18 m 58 s	19 m 6 s	1,323	-	44.5
	Maq 0.7.1	4 h 45 m 7 s	4 h 45 m 17 s	1,155	14.9×	44.9
	Bowtie -v 2	7 m 35 s	7 m 40 s	1,138	-	31.7

The performance of Bowtie v0.9.6, SOAP v1.10, and Maq versions v0.6.6 and v0.7.1 on the server platform when aligning 2 M untrimmed reads from the 1,000 Genome project (National Center for Biotechnology Information Short Read Archive: SRR003084 for 36 base pairs [bp], SRR003092 for 50 bp, and SRR003196 for 76 bp). For each read length, the 2 M reads were randomly sampled from the FASTQ file downloaded from the Archive such that the average per-base error rate as measured by quality values was uniform across the three sets. All reads pass through Maq's "catfilter". Maq v0.7.1 was used for the 76-bp reads because v0.6.6 does not support reads longer than 63 bp. SOAP is excluded from the 76-bp experiment because it does not support reads longer than 60 bp. Other experimental parameters are identical to those of the experiments in Table 1. CPU, central processing unit.

Maq & SOAP build hash table of locations of k-mers

Langmead et al. (2008)