Searching Genomic Sequences

- Searching for a string of length $m$ in a text of length $n$
- Indexing strings with trees
  - Keyword tree $O(n)$ construction, $O(m)$ search
  - Suffix tree $O(m)$
- Suffix Arrays as a practical alternative to Suffix tree
  - $O(\log n)$
- Burrows-Wheeler transform, back to $O(m)$
Keyword Tree

- A tree for representing a “dictionary” of terms
- Merges common prefixes into a single path
- Example:
  - miss
  - mississippi
  - mist
  - mister
  - sister
  - sippy
Keyword Tree

• Queries supported:
  Does keyword, k, appear in my text?
  – missstep
  – sip
• Searching via “Threading”
• Useful for spell checking, but hashing is preferred
• Not good for how many words contain “sis”
• **A compressed** keyword tree of suffixes from a single string

• Compressed by collapsing all nodes with out-degree of one

• Leaf nodes are labeled by the starting location of the suffix that terminates there

• Note that we now add an end-of-string character ‘$’
Suffix Tree Features

• How many leaves in a text of length $n$?

• Given a suffix tree for a text.
  How long to determine if a substring of length $n$ occurs in the text?

• Can I find how many occurrences of a substring, and where?
**Suffix Tree Features**

- **How much storage?**
  - Just for the edge strings \(O(n^2)\)
  - Trick: Rather than storing an actual string at each edge, we can instead store 2 integer offsets into the original text

- In practice the storage overhead of Suffix Trees is too high, \(O(n)\) vertices with data and \(O(n)\) edges with associated data
There exists a depth-first traversal that corresponds to lexicographical ordering (alphabetizing) all suffixes:

11. $  
10. i$  
7. ippi$  
4. issippi$  
1. ississippi$  
0. mississippi$  
9. pi$  
8. ppi$  
6. sippi$  
3. sissippi$  
5. ssippi$  
2. ssissippi$
One could exploit this property to construct a Suffix Tree:
  - Make a list of all suffixes: $O(n)$
  - Sort them: $O(n^2 \log n)$
  - Traverse the list from beginning to end while threading each suffix into the tree created so far, when the suffix deviates from a known path in the tree, add a new node with a path to a leaf.

There is a faster $O(m)$ algorithm by Ukkonen.
• Sorting however did capture important aspects of the suffix trees structure
• A sorted list of tree-path traversals, our sorted list, can be considered a “compressed” version of a suffix tree.
• Save only the index to the beginning of each suffix
  11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2

• Key: Argsort(text): returns the indices of the sorted elements of a text
Argsort

• One of the smallest Python functions yet:

```python
def argsort(text):
    return sorted(range(len(text)), cmp=lambda i,j: -1 if text[i:] < text[j:] else 1)

print argsort("mississippi")
```

$ python suffixarray.py
[11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]

• What types of queries can be made from this “compressed” form of a suffix tree
• We call this a “Suffix Array”
Suffix Array Queries

• Has similar capabilities to a Suffix Tree
• Does ‘sip’ occur in “mississippi”?  
• How many times does ‘is’ occur?  
• How many ‘i’’s?  
• What is the longest repeated subsequence?
• Given a suffix array for a sequence. How long to determine if a pattern of length \( m \) occurs in the sequence?
Searching Suffix Arrays

- Functions to find the first and last occurrence of a pattern

```python
def findFirst(pattern, text, sfa):
    # Finds the index of the first occurrence of pattern in the suffix array
    hi = len(text)
    lo = 0
    while (lo < hi):
        mid = (lo+hi)//2
        if (pattern > text[sfa[mid]:]):
            lo = mid + 1
        else:
            hi = mid
    return lo

def findLast(pattern, text, sfa):
    # Finds the index of the last occurrence of pattern in the suffix array
    hi = len(text)
    lo = 0
    m = len(pattern)
    while (lo < hi):
        mid = (lo+hi)//2
        i = sfa[mid]
        if (pattern >= text[i:i+m]):
            lo = mid + 1
        else:
            hi = mid
    return lo-1
```
Augmenting Suffix Arrays

- It is possible to augment a suffix array to facilitate converting it into a suffix tree

- Longest Common Prefix, (lcp)
  - Note that branches, and, hence, interior nodes if needed are introduced immediately following a shared prefix of two adjacent suffix array entries
    
    $\begin{array}{ll}
    \$ & lcp = 0 \\
    i & lcp = 1 \\
    ippi & lcp = 1 \\
    issi & lcp = 4 \\
    ississippi & lcp = 0 \\
    mississippi & lcp = 0 \\
    \end{array}$

- If we store the lcp along with the suffix array it is a simple matter to reconstruct and traverse the corresponding Suffix Tree
Other Data Structures

• There is another trick for finding patterns in a text string, it comes from a rather odd remapping of the original text called a “Burrows-Wheeler Transform” or BWT.

• BWTs have a long history. They were invented back in the 1980s as a technique for improving lossless compression. BWTs have recently been rediscovered and used for DNA sequence alignments. Most notably by the Bowtie and BWA programs for sequence alignments.
String Rotation

• Before describing the BWT, we need to define the notion of Rotating a string. The idea is simple, a rotation of $i$ moves the prefix$_i$, to the string’s end making it a suffix.

  Rotate(“tarheel$”, 3) $\rightarrow$ “heel$tar”
  Rotate(“tarheel$”, 7) $\rightarrow$ “$starheel”
  Rotate(“tarheel$”, 1) $\rightarrow$ “arheel$t$”
BWT Algorithm

BWT (string text)

\[
\text{table}_i = \text{Rotate}(text, i) \text{ for } i = 0..\text{len}(text)-1
\]

sort table alphabetically

return (last column of the table)

\[
\begin{array}{ll}
tarheel$ & $tarheel \\
arheel$t & arheel$t \\
rheel$ta & eel$tarh \\
heel$tar & el$tarhe \\
eel$tarh & heel$tar \\
el$tarhe & l$tarhee \\
l$tarhee & rheel$ta \\
$tarheel & tarheel$
\end{array}
\]

BTW(“tarheels$”) = “ltherea$”
BWT in Python

• Once again, this is one of the simpler algorithms that we’ve seen

```python
def BWT(s):
    # create a table, with rows of all possible rotations of s
    rotation = [s[i:] + s[:i] for i in xrange(len(s))]
    # sort rows alphabetically
    rotation.sort()
    # return (last column of the table)
    return ''.join([r[-1] for r in rotation])
```

• Input string of length $n$, output a messed up string of length $n$
Inverse of BWT

- A property of any transform is that there is no information loss and they are invertible.

InverseBWT(string s)

  add s as the first column of a table strings

  repeat length(s)-1 times:
    sort rows of the table alphabetically
    add s as the first column of the table

  return (row that ends with the 'EOF' character)

```plaintext
l  l$  l$t  l$ta  l$tar  l$tarh  l$tarhe  l$tarhee
a  an  anr  anrh  anrhe  anrhee  anrhehe  anrheehe
r  r$  r$t  r$ta  r$tar  r$tarh  r$tarhe  r$tarhee
```

2/12/14
def inverseBWT(s):
    # initialize table from s
    table = [c for c in s]
    # repeat length(s) - 1 times
    for j in xrange(len(s)-1):
        # sort rows of the table alphabetically
        table.sort()
        # prepend s as the first column
        table = [s[i]+table[i] for i in xrange(len(s))]
    # return (row that ends with the 'EOS' character)
    return table[[r[-1] for r in table].index('$$')]
How to use a BWT?

- A BWT is a "last-first" mapping meaning the $i^{th}$ occurrence of a character in the first column corresponds to the $i^{th}$ occurrence in the last.
- Also, recall the first column is sorted
- $\text{BWT("mississippi$") } \rightarrow \text{ “ipssm$pissii”}$
- Compute from $\text{BWT(s)}$ a sorted dictionary of the number of occurrences of each letter
  
  $$C[*][i+1] = \{ ‘$’:1, ‘i’:4, ‘m’:1, ‘p’:2, ‘s’:4 \}$$

- Using the last entry it is a simple matter to find indices of the first occurrence of a character on the "left" sorted side
  
  $$O = \{ ‘$’:0, ‘i’:1, ‘m’:5, ‘p’:6, ‘s’:8 \}$$
Searching for a Pattern

- Find “iss” in “mississippi”
- Searches for patterns take place in reverse order (last character to first)
- Use the O index to find the range of entries starting with the last character

\[ I = \{ 's':0, 'i':1, 'm':5, 'p':6, 's':8 \} \]

\[
\begin{align*}
C[\text{letter}][i] &= $\text{imps} \\
0 &\text{ $mississippi} 00000 \\
1 &i\text{mississip} 01000 \\
2 &ippi\text{mississ} 01010 \\
3 &issippi$miss 01011 \\
4 &ississippi$m 01012 \\
5 &\text{mississippi}$ 01112 \\
6 &pi$\text{mississip} 11112 \\
7 &ippi$\text{mississi} 11122 \\
8 &sippi$\text{missis} 12122 \\
9 &si\text{issippi$mis} 12123 \\
10 &sippi$\text{missi} 12124 \\
11 &ssissippi$\text{mi} 13124 \\
& 14124 \\
O[\text{letter}] &= 01568
\end{align*}
\]

2/12/14
Searching for a Pattern

• This is done using the FMIndex as follows:

```python
def find(pattern, FMindex):
    lo = 0
    hi = len(FMindex)
    for l in reversed(pattern):
        lo = O[l] + C[lo][l]
        hi = O[l] + C[hi][l]
    return lo, hi
```

```python
find("iss", FMindex)
```

```python
lo0, hi0 = 0, 12
lo1 = 0[‘s’] + C[0][‘s’] = 8 + 0 = 8
hi1 = 0[‘s’] + C[12][‘s’] = 8 + 4 = 12
lo2 = 0[‘s’] + C[8][‘s’] = 8 + 2 = 10
hi2 = 0[‘s’] + C[12][‘s’] = 8 + 4 = 12
lo3 = 0[‘i’] + C[10][‘i’] = 1 + 2 = 3
hi3 = 0[‘i’] + C[12][‘i’] = 1 + 4 = 5
```

```plaintext
C[letter][i] = $impso
0 $mississippi 00000
1 i$mississippi 01000
2 ippi$mississ 01010
3 issippi$miss 01011
4 ississippi$m 01012
5 mississippi$ 01112
6 pi$mississip 11112
7 ppi$mississi 11122
8 sippi$missis 12122
9 sissippi$mis 12123
10 ssippi$missi 12124
11 ssissippi$mi 13124
14124
```

0[letter] = 01568
Recovering the $i^{th}$ Suffix

- The Search algorithm returns the indices of matches within a suffix array that is implicitly represented by the BWT
- We can recover any suffix array entry by also using the FM-index
- Recall at this point we only have access to the BWT (shown in black) and the FM-index (shown in red and green)

<table>
<thead>
<tr>
<th>C[letter][i]</th>
<th>$imps$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 $mississippi$</td>
<td>00000</td>
</tr>
<tr>
<td>1 i$mississippi$</td>
<td>01000</td>
</tr>
<tr>
<td>2ippi$mississ$</td>
<td>01010</td>
</tr>
<tr>
<td>3 issippi$miss$</td>
<td>01011</td>
</tr>
<tr>
<td>4 ississippi$m$</td>
<td>01012</td>
</tr>
<tr>
<td>5 mississippi$</td>
<td>$</td>
</tr>
<tr>
<td>6 pi$mississi$p</td>
<td>11112</td>
</tr>
<tr>
<td>7 ppi$mississi$</td>
<td>11122</td>
</tr>
<tr>
<td>8 sippi$missis$</td>
<td>12122</td>
</tr>
<tr>
<td>9 sissippi$mis$</td>
<td>12123</td>
</tr>
<tr>
<td>10 ssippi$missi$</td>
<td>12124</td>
</tr>
<tr>
<td>11 ssissippi$mi$</td>
<td>13124</td>
</tr>
<tr>
<td></td>
<td>14124</td>
</tr>
</tbody>
</table>

$[letter] = 01568$
Recovering the $i^{th}$ Suffix

- The $i^{th}$ entry of the “hidden” Suffix Array can be found as follows:

```
def suffix(i, Fmindex, bwt):
    result = ‘’
    j = i
    while True:
        j = O[bwt[j]] + C[j][bwt[j]]
        result = bwt[j] + result
        if (i == j):
            break
    return result

suffix(3, Fmindex, bwt)
```

```
j = 0[‘s’] + C[3][‘s’] = 8 + 1; result = ‘s’
j = 0[‘s’] + C[9][‘s’] = 8 + 3; result = ‘ss’
j = 0[‘i’] + C[11][‘i’] = 1 + 3; result = ‘iss’
j = 0[‘m’] + C[4][‘m’] = 5 + 0; result = ‘miss’
j = 0[‘$’] + C[5][‘$’] = 0 + 0; result = ‘$miss’
j = 0[‘i’] + C[0][‘i’] = 1 + 0; result = ‘i$miss’
j = 0[‘p’] + C[1][‘p’] = 6 + 0; result = ‘pi$miss’
```

```
C[letter][i] = $imps
  0 $mississippi 00000
  1 i$mississippi 01000
  2 ippi$mississippi 01010
  3 issippi$mississippi 01011
  4 isissippi$m 01012
  5 mississippi$ 01112
  6 pi$mississipp 11112
  7 ppi$mississippi 11122
  8 sippi$mississippi 12122
  9 ssissippi$miss 12123
 10 ssippi$missi 12124
 11 ssissippi$m 13124
 14124
```

$[letter] = 01568$
Recovering the $i$th Suffix

- The $i$th entry of the “hidden” Suffix Array can be found as follows:

```python
def suffix(i, Fmindex, bwt):
    result = ''
    j = i
    while True:
        j = 0[bwt[j]] + C[j][bwt[j]]
        result = bwt[j] + result
        if (i == j):
            break
    return result
```

(suffix(3, Fmindex, bwt)
(continued)

<table>
<thead>
<tr>
<th>C[letter][i]</th>
<th>$imps$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$mississippi$ 00000</td>
</tr>
<tr>
<td>1</td>
<td>i$mississippi$ 01000</td>
</tr>
<tr>
<td>2</td>
<td>ippi$mississ$ 01010</td>
</tr>
<tr>
<td>3</td>
<td>issippi$miss$ 01011</td>
</tr>
<tr>
<td>4</td>
<td>ississippi$mis$ 01012</td>
</tr>
<tr>
<td>5</td>
<td>mississippi$ p$ 01112</td>
</tr>
<tr>
<td>6</td>
<td>pismississippi$ 11112</td>
</tr>
<tr>
<td>7</td>
<td>ppi$mississi$ 11122</td>
</tr>
<tr>
<td>8</td>
<td>sip$mississi$ 12122</td>
</tr>
<tr>
<td>9</td>
<td>sippi$mississi$ 12123</td>
</tr>
<tr>
<td>10</td>
<td>ssippi$mississi$ 12124</td>
</tr>
<tr>
<td>11</td>
<td>ssissippi$missi$ 13124</td>
</tr>
</tbody>
</table>

j = 0[‘p’] + C[1][‘p’] = 6 + 0; result = ‘pip$miss’
j = 0[‘p’] + C[6][‘p’] = 6 + 1; result = ‘ppi$miss’
j = 0[‘i’] + C[7][‘i’] = 1 + 1; result = ‘ippi$miss’
j = 0[‘s’] + C[2][‘s’] = 8 + 0; result = ‘sippi$miss’
j = 0[‘s’] + C[8][‘s’] = 8 + 2; result = ‘ssippi$miss’
j = 0[‘i’] + C[10][‘i’] = 1 + 2; result = ‘issippi$miss’
Searching for a pattern, $p$, in a BWT requires $O(|p|)$ steps (same as Suffix Tree!)

- Recovering any suffix from the implicit suffix tree requires $O(n)$ steps, where $n$ is the length of the BWT encoded string.

- There is actually yet another index that allows one to find prefixes, $r$, of suffixes in $O(|r|)$.

- The largest cost associated with the BWT is constructing and storing the FM-index. It can be built in $O(|n|)$ steps, and stored in $O(|\Sigma||n|)$ memory, where $\Sigma$ is the alphabet size.

- The FM-index can be sampled (not every entry needs to be computed), with the missing entries filled in on the fly.
Summary

• Query Power (Big is good)
  – BWTs support the fewest query types of these data structs
  – Suffix Trees perform a variety of queries in $O(m)$
Summary

• Memory Footprint (Small is good)
  – BWTs compress very well on real data
  – Difficult to store a full suffix tree for an entire genome