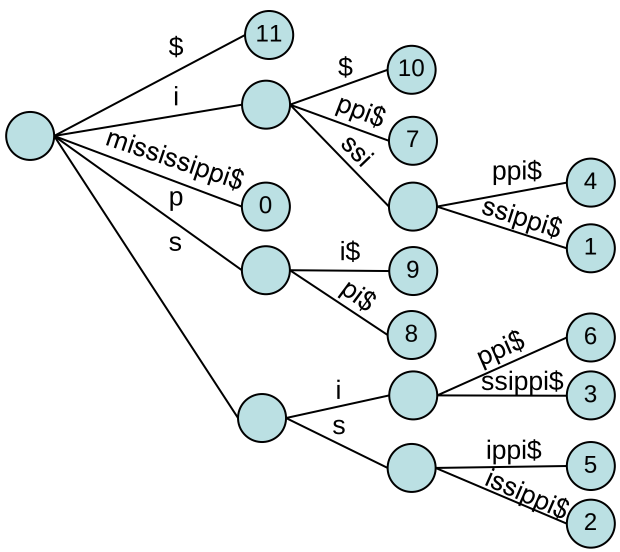
Lecture 17: Suffix Arrays and the Burrows-Wheeler Transform

Not in Book

Recall Suffix Trees

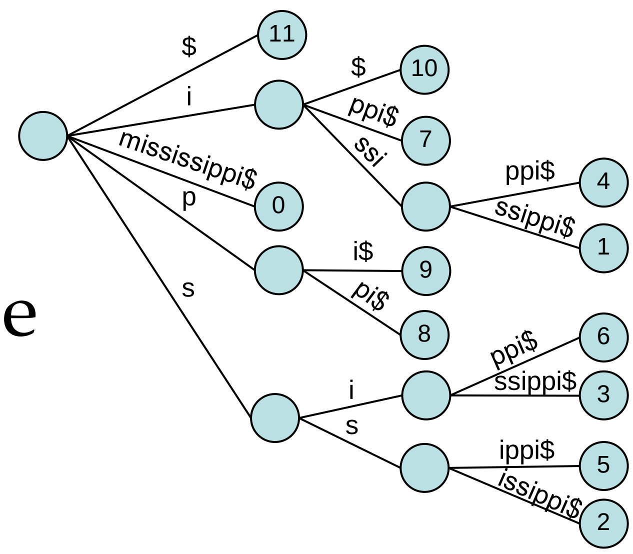
- A compressed keyword tree of suffixes from a given sequence
- Leaf nodes are labeled by the starting location of the suffix that terminates there
- Note that we now add an end-of-string character ‘$’



0. mississippi$
1. ississippi$
2. ssissippi$
3. sissippi$
4. issippi$
5. ssippi$
6. sippi$
7. ippi$
8. ppi$
9. pi$
10. i$
11. $

Suffix Tree Features

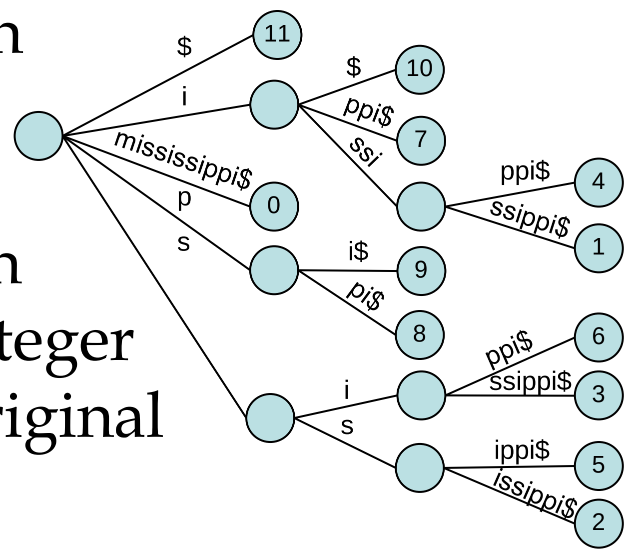
- How many leaves in a sequence of length m? O(m)
- How many nodes? (assume an alphabet of k characters) O(m)
- Given a suffix tree for a sequence. How long to determine if a pattern of length n occurs in the sequence? O(n)



0. mississippi$
1. ississippi$
2. ssissippi$
3. sissippi$
4. issippi$
5. ssippi$
6. sippi$
7. ippi$
8. ppi$
9. pi$
10. i$
11. $

Suffix Tree Features

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



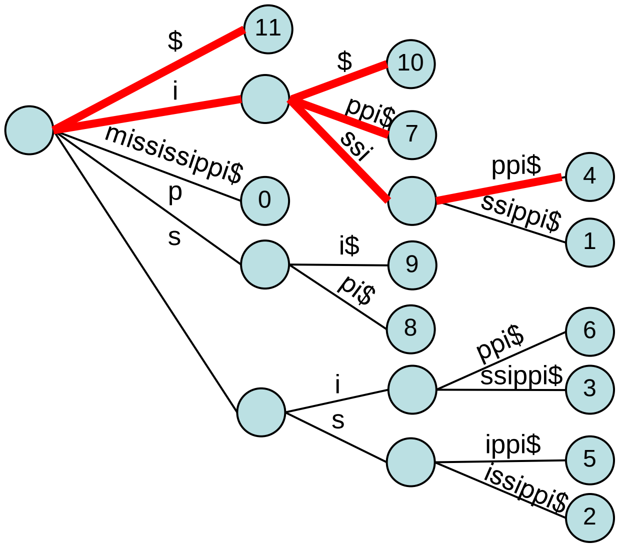
0. mississippi$
1. ississippi$
2. ssissippi$
3. sissippi$
4. issippi$
5. ssippi$
6. sippi$
7. ippi$
8. ppi$
9. pi$
10. i$
11. $

- In practice the storage overhead of Suffix Trees is too high, O(m) vertices with data and O(m) edges with associated data

Suffix Tree Properties

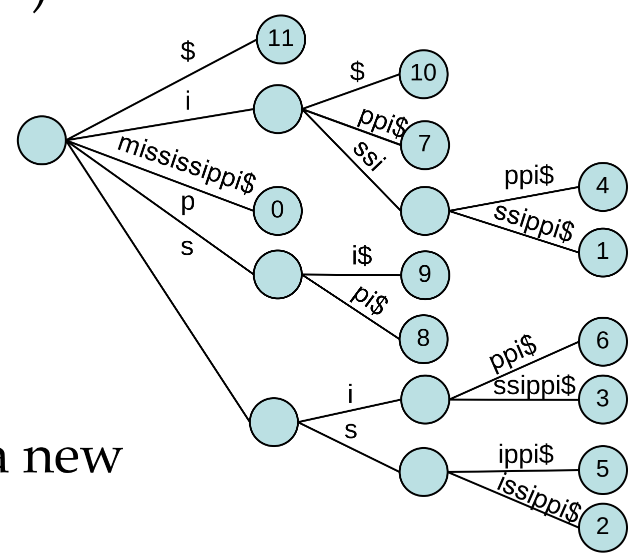
- There exists a depth-first traversal that corresponds to lexigraphical ordering (alphabetizing) all suffixes

11. $
10. i$
7. ippi$
4. issippi$
1. ississippi$
0. mississippi$
9. pi$
8. ppi$
6. sippi$
3. sissippi$
5. ssippi$
2. ssissippi$



Suffix Tree Construction

- One could exploit this property to construct a Suffix Tree
  - Make a list of all suffixes: O(m)
  - Sort them: O(m log m)
  - 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.



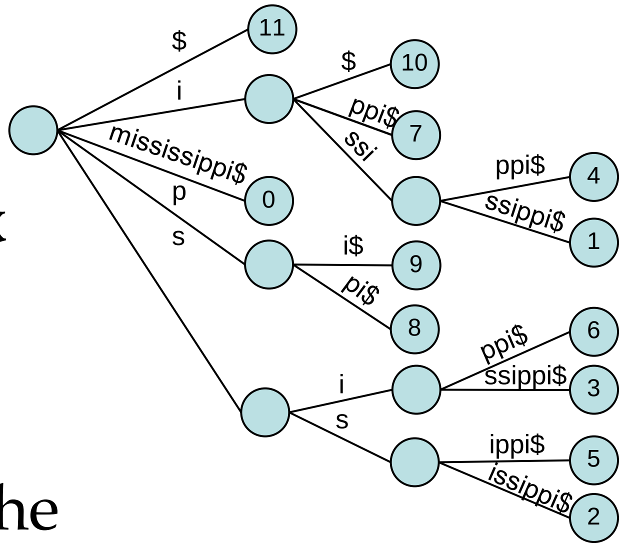
11. $
10. i$
7. ippi$
4. issippi$
1. ississippi$
0. mississippi$
9. pi$
8. ppi$
6. sippi$
3. sissippi$
5. ssippi$
2. ssissippi$

- ☹ Slower than the O(m) Ukkonen algorithm given last time

Saving space

- 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:

```
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 *n* occurs in the sequence? O(n log m)

11. $
10. i$
7. ippi$
4. issippi$
1. ississippi$
0. mississippi$
9. pi$
8. ppi$
6. sippi$
3. sissippi$
5. ssippi$
2. ssissippi$

Searching Suffix Arrays

- Separate functions for finding the first and last occurrence of a pattern via binary search

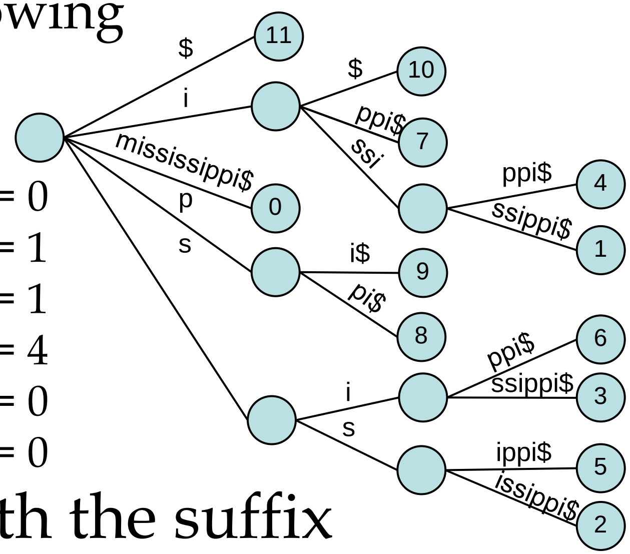
```
def findFirst(pattern, text, sfa):
    """ Finds the index of the first occurrence of pattern in the suffix array """
    hi = len(text)
    lo = 0
    while (lo < hi):
        mid = (lo+hi)//2
        if (pattern > text[sfa[mid]:]):
            lo = mid + 1
        else:
            hi = mid
    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 than branches, and, hence, interior nodes if needed are introduced immediately following a shared prefix of two adjacent suffix array entries

$	lcp = 0
i$	lcp = 1
ippi$	lcp = 1
issipi$	lcp = 4
ississippi$	lcp = 0
mississippi$	lcp = 0



11. $
10. i$
7. ippi$
4. issippi$
1. ississippi$
0. mississippi$
9. pi$
8. ppi$
6. sippi$
3. sissippi$
5. ssippi$
2. ssissippi$

- If we store the lcp along with the suffix array it becomes a trivial matter to reconstruct and traverse the corresponding Suffix Array

Construction of Suffix Array

- Strategy thus far
  - build suffix tree, and enumerate elements of SA via depth first traversal
  - O(m) time to construct suffix tree via Ukonnen's alg
  - O(m) space, but too large in practice
- Linear time direct construction of SA
  - Kärkkäinen, Sanders, Burkhart algorithm (2005)
  - O(m) time by relatively simple asymmetric divide and conquer algorithm
  - space efficient

Faster search in Suffix Array

- Reduce complexity to O(n + lg m)
  - Idea: augment SA with longest common prefix (LCP)
  - Interleave binary search and matching of query
  - LCP insures no need to recheck prefix of query

Even faster search

- Use hashtable for length h prefixes
  - $h = \log_k m$ (where k is the size of the alphabet)
  - reduces *expected* length of search interval to O(1)
- Time complexity
  - O(1) expected if not present
  - $O(n)$ expected if present
  - $O(n + \lg m)$ worst case
- Space complexity $O(m)$
  - Accelerator: m words
  - Suffix array: m words

Another Approach

- There is another trick for finding patterns in a text, 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 circular rotation of a string. A rotation of *i* moves the prefix$_i$, to the string's end making it a suffix.

Rotate("tarheel\$", 3) → "heel\$tar"

Rotate("tarheel\$", 7) → "\$tarheel"

Rotate("tarheel\$", 1) → "arheel\$t"

BWT Algorithm

BWT (string text)

  table$_i$ = Rotate(text, i) for i = 0..len(text)-1

  sort table alphabetically

  return (last column of the table)

tarheel$	$tarhee	l
arheel$t	arheel$	t
rheel$ta	eel$tar	h
heel$tar	el$tarh	e
eel$tarh	heel$ta	r
el$tarhe	l$tarhe	e
l$tarhee	rheel$t	a
$tarheel	tarheel	$

BTW(“tarheels$”) = “ltherea$”

BWT in Python

- Once again, this is one of the simpler algorithms that we've seen

```
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 *m*, output a messed up string of length *m*

Inverse of BWT

- A property of a 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 '$' character)
```

l	l$	l$t	l$ta	l$tar	l$tarh	l$tarhe	l$tarhee
t	ta	tar	tarh	tarhe	tarhee	tarheel	$\boxed{\text{tarheel\$}}$
h	he	hee	heel	heel$	heel$t	heel$ta	heel$tar
e	ee	eel	eel$	eel$t	eel$ta	eel$tar	eel$tarh
r	rh	rhe	rhee	rheel	rheel$	rheel$t	rheel$ta
e	el	el$	el$t	el$ta	el$tar	el$tarh	el$tarhe
a	ar	arh	arhe	arhee	arheel	arheel$	arheel$t
$	$t	$ta	$tar	$tarh	$tarhe	$tarhee	$tarheel

Inverse BTW in Python

- A slightly more complicated routine

```
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()
        # insert 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.
- Recall the first column is sorted
- BWT("mississippi$") → "ipssm$pissii"
- Compute from BWT a sorted dictionary of the number of occurrences of each letter

  C[*][m] = { '$':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 }

BWT FM-index

```
C[letter][i] = $imps
 0 $mississippi 00000
 1 i$mississipp 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
   O[letter] = 01568
```

Searching for a Pattern

- Find "iss" in "mississippi"
- Search 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 = { '$':0, 'i':1, 'm':5, 'p':6, 's':8 }

```
C[letter][i] = $imps
 0 $mississippi 00000
 1 i$mississipp 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
   O[letter] = 01568
```

Searching for a Pattern

- Find "sis" in "mississippi"
- Of these, how many BTW entries match the second-to-last character? If none string does not appear
- Use the C-index to find all offsets of occurrences of these second to last characters, which will be contiguous

```
C[letter][i] = $imps
 0 $mississippi 00000
 1 i$mississipp 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
   O[letter] = 01568
```

Searching for a Pattern

- This is done using the FMindex as follows:

```
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
```

```
C[letter][i] = $imps
 0 $mississippi 00000
 1 i$mississipp 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
     O[letter] = 01568
```

find("iss", Fmindex)

lo0, hi0 = 0, 12

lo1 = O['s'] + C[0]['s'] = 8 + 0 = 8
hi1 = O['s'] + C[12]['s'] = 8 + 4 = 12

lo2 = O['s'] + C[8]['s'] = 8 + 2 = 10
hi2 = O['s'] + C[12]['s'] = 8 + 4 = 12

lo3 = O['i'] + C[10]['i'] = 1 + 2 = 3
hi3 = O['i'] + C[12]['i'] = 1 + 4 = 5

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 again using the FM-index
- Recall at this point we only have access to the BWT (shown in black) and the FMindex (shown in red and green)

```
C[letter][i] = $imps
0 $mississippi 00000
1 i$mississipp 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
   O[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
```

```
C[letter][i] = $imps
 0 $mississippi 00000
 1 i$mississipp 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
  O[letter] = 01568
```

suffix(3, Fmindex, bwt)

j = O[‘s’] + C[3][‘s’] = 8 + 1; result = ‘s’

j = O[‘s’] + C[9][‘s’] = 8 + 3; result = ‘ss’

j = O[‘i’] + C[11][‘i’] = 1 + 3; result = ‘iss’

j = O[‘m’] + C[4][‘m’] = 5 + 0; result = ‘miss’

j = O[‘$’] + C[5][‘$’] = 0 + 0; result = ‘$miss’

j = O[‘i’] + C[0][‘i’] = 1 + 0; result = ‘i$miss’

j = O[‘p’] + C[1][‘p’] = 6 + 0; result = ‘pi$miss’

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
```

```
C[letter][i] = $imps
0 $mississippi 00000
1 i$mississipp 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
   O[letter] = 01568
```

suffix(3, Fmindex, bwt)
(continued)

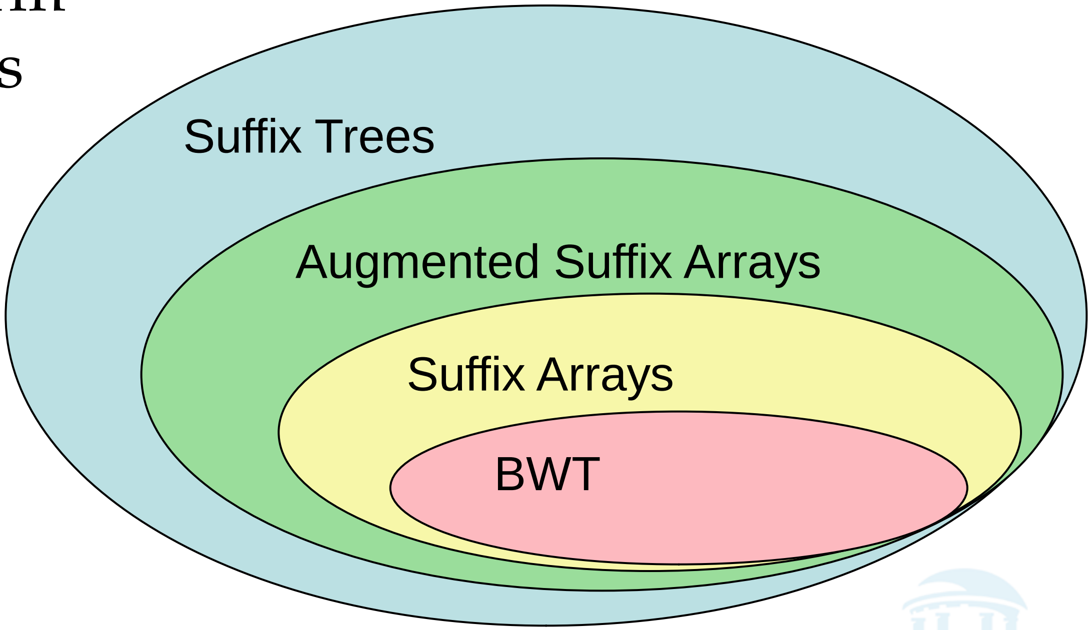
j = O[‘p’] + C[1][‘p’] = 6 + 0; result = ‘pi$miss’
j = O[‘p’] + C[6][‘p’] = 6 + 1; result = ‘ppi$miss’
j = O[‘i’] + C[7][‘i’] = 1 + 1; result = ‘ippi$miss’
j = O[‘s’] + C[2][‘s’] = 8 + 0; result = ‘sippi$miss’
j = O[‘s’] + C[8][‘s’] = 8 + 2; result = ‘ssippi$miss’
j = O[‘i’] + C[10][‘i’] = 1 + 2; result = ‘issippi$miss’

BWT Search Details

- Searching for a pattern, p, in a BWT requires $O(|p|)$ steps (same as Suffix Tree!)
- Recovering any entry 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 FMindex. It can be built in $O(|n|)$ steps, and stored in $O(|\Sigma|\,|n|)$ memory, where Σ is the alphabet size

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 the full suffix tree for an entire genome