How do I love thee? Let me count the ways. Suppose there are $n$ ways of loving someone and I can love you in any $k$ of them. Assuming order doesn't matter, there are simply \( \binom{n}{k} = \frac{n!}{k!(n-k)!} \) ways. If order does matter — eg, if buying you flowers on Monday and taking you to a show on Tuesday differs from taking you to a show on Monday and buying you flowers on Tuesday, then we have \( \frac{n!}{(n-k)!} \), or \( \binom{n}{k} \) — but what if I can love you in $k$ ways, then $m$ ways? This scenario requires the multichoose operation,

\[
\binom{n}{k, m} = \frac{n!}{k!(n-k)!m!(n-k-m)!}
\]
A Recurring Problem

- Finding patterns within sequences
- Variants on this idea
  - Finding repeated motifs amongst a set of strings
  - What are the most frequent k-mers
  - How many times does a specific k-mer appear
- Fundamental problem: *Pattern Matching*
  - Find all positions of a particular substring in a given sequence?
Pattern Matching

- **Goal:** Find all occurrences of a pattern in a text
- **Input:** Pattern \( p = p_1, p_2, \ldots, p_n \) and text \( t = t_1, t_2, \ldots, t_m \)
- **Output:** All positions \( 1 < i < (m - n + 1) \) such that the \( n \)-letter substring of \( t \) starting at \( i \) matches \( p \)

```python
def bruteForcePatternMatching(p, t):
    locations = []
    for i in xrange(0, len(t)-len(p)+1):
        if t[i:i+len(p)] == p:
            locations.append(i)
    return locations

print bruteForcePatternMatching("ssi", "imissmissmississippi")

[11, 14]
```
Pattern Matching Performance

- Performance:
  - $m$ - length of the text $t$
  - $n$ - the length of the pattern $p$
  - Search Loop - executed $O(m)$ times
  - Comparison - $O(n)$ symbols compared
  - Total cost - $O(mn)$ per pattern
- In practice, most comparisons terminate early
- Worst-case:
  - $p = "AAAT"
  - $t = "AAAAAAAAAAAAAAAAAAAAAT"$
We can do better!

If we preprocess our pattern we can search more efficiently \(O(n)\)

Example:

```
imissmissmississippi
1. s
2. s
3. s
4. SSi
5. s
6. SSi
7. s
8. SSI - match at 11
9. SSI - match at 14
10. s
11. s
12. s
```

- At steps 4 and 6 after finding the mismatch \(i \neq m\) we can skip over all positions tested because we know that the suffix "sm" is not a prefix of our pattern "ssi"
- Even works for our worst-case example "AAAAAT" in "AAAAAAAAAAAAAAAAAT" by recognizing the shared prefixes ("AAA" in "AAAA")
- How about finding multiple patterns \([p_1, p_2, \ldots, p_3]\) in \(t\)
Keyword Trees

- We can preprocess the set of strings we are seeking to minimize the number of comparisons
- **Idea:** Combine patterns that share prefixes, to *share* those comparisons
  - Stores a set of keywords in a rooted labeled tree
  - Each edge labeled with a letter from an alphabet
  - All edges leaving a given vertex have distinct labels
  - Leaf vertices are indicated
  - Every keyword stored can be spelled on a path from root to some leaf vertex
  - Searches are performed by “threading” the target pattern through the tree
- A tree is a special graph as discussed previously
  - one connected component
  - $N$ nodes
  - $N-1$ edges
  - No loops
  - Exactly one path from any.
- A **Trie** is a tree that is related to a sequence.
  - Generally, there is a 1-to-1 correspondence between either nodes or edges of the trie and a symbol of the sequence
Prefix Trie Match

- **Input**: A text \( t \) and a trie \( P \) of patterns
- **Output**: True if \( t \) leads to a leaf in \( P \); False otherwise

What is output for:

- *apple*
- *band*
- *april*

Performance:

- \( O(m) \) - the length of the text, \( t \)
- Independent of how many strings are in the Keyword Trie
Multiple Pattern Matching

- $t$ - the text to search through
- $P$ - the trie of patterns to search for

```python
def multiplePatternMatching(t, P):
    locations = []
    for i in xrange(0, len(t)):
        if PrefixTrieMatch(t[i:], P):
            locations.append(i)
    return locations
```
Multiple Pattern Matching Example

```
multiplePatternMatching("bananapple", P):
0: PrefixTrieMatching("bananapple", P) = True
1: PrefixTrieMatching("ananapple", P) = False
2: PrefixTrieMatching("nanapple", P) = False
3: PrefixTrieMatching("anapple", P) = False
4: PrefixTrieMatching("napple", P) = False
5: PrefixTrieMatching("apple", P) = True
6: PrefixTrieMatching("pple", P) = False
7: PrefixTrieMatching("ple", P) = False
8: PrefixTrieMatching("le", P) = False
9: PrefixTrieMatching("e", P) = False

locations = [0, 5]
```
Improvements

- Based on our previous speed-up
- We can add failure edges to our Trie
- *Aho-Corasick Algorithm*
Multiple Pattern Matching Performance

- $m \cdot \text{len}(t)$
- $d$ - max depth of $P$ (longest pattern in $P$)
- $O(md)$ to find all patterns
- Can be decreased further to $O(m)$ using Aho-Corasick Algorithm (see pg 353)
- Memory issues
  - Tries require a lot of memory
  - Practical implementation is challenging
  - Genomic reads - millions to billions of
- Patterns typically of length > 100
Another Twist

- What if our list of keywords were simply all suffixes of a given string

Example: ACATG
  CATG
  ATG
  TG
  G

- The resulting keyword tree:
- A **Suffix Trie**
Suffix Tree

A compressed Suffix Trie

- Combines nodes with in and out degree 1
- Edges are text substrings
- All internal nodes have at least 3 edges
- All leaf nodes are labeled with an index
Uses for Suffix Trees

- Suffix trees hold all suffixes of a text, T
  - i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
- Can be built in $O(m)$ time for text of length $m$
- To find any pattern $P$ in a text:
  - Build suffix tree for text, $O(m)$, $m = |T|$
  - Thread the pattern through the suffix tree
  - Can find pattern in $O(n)$ time! ($n = |P|$)
- $O(|T| + |P|)$ time for "Pattern Matching Problem"
  (better than Naïve $O(|P||T|)$)
- Build suffix tree and lookup pattern
- Multiple Pattern Matching in $O(|T| + k|P|)$
Suffix Tree Overhead

- **Input:** text of length m
- **Computation**
  - $O(m)$ to compute a suffix tree
  - Does not require building the suffix trie first
- **Memory**
  - $O(m)$ - nodes are stored as offsets and lengths
- Huge hidden constant, best implementations
- Requires about $20m$ bytes
- 3 GB human genome = 60 GB RAM
Suffix Tree Examples

- What is the string represented in the suffix tree?
- What letter occurs most frequently?
- How many times does "ATG" appear, and where?
- How long is the longest repeated k-mer?
In theory, suffix trees are extremely powerful for making a variety of queries concerning a sequence:
- What is the shortest unique substring?
- How many times does a given string appear in a text?

Despite the existence of linear-time construction algorithms, and $O(m)$ search times, suffix trees are still rarely used for genome-scale searching.

Large storage overhead
Is there some other data structure to gain efficient access to all of the suffixes of a given string with less overhead than a suffix tree?

Some things we know

- Searching an unordered list of items with length $n$ generally requires $O(n)$ steps
- However, if we sort our items first, then we can search using $O(\log(n))$ steps
- Thus, if we plan to do frequent searches there is some advantage to performing a sort first and amortizing its cost over many searches

For strings suffixes are interesting items. Why?

<table>
<thead>
<tr>
<th>Suffixes: panamabananas</th>
<th>Sorted Suffixes: abananas</th>
</tr>
</thead>
<tbody>
<tr>
<td>anamabananas</td>
<td>amabananas</td>
</tr>
<tr>
<td>namabananas</td>
<td>anamabananas</td>
</tr>
<tr>
<td>amabananas</td>
<td>ananas</td>
</tr>
<tr>
<td>mabananas</td>
<td>ananas</td>
</tr>
<tr>
<td>abananas</td>
<td>as</td>
</tr>
<tr>
<td>bananas</td>
<td>bananas</td>
</tr>
<tr>
<td>ananas</td>
<td>mabananas</td>
</tr>
<tr>
<td>nanas</td>
<td>namabananas</td>
</tr>
<tr>
<td>anas</td>
<td>nanas</td>
</tr>
<tr>
<td>nas</td>
<td>nas</td>
</tr>
<tr>
<td>as</td>
<td>panamabananas</td>
</tr>
<tr>
<td>s</td>
<td>s</td>
</tr>
</tbody>
</table>
Questions you can ask

Is there any use for a list of sorted suffixes?

Sometimes the questions are complicated and the answers are simple.

Sorted Suffixes:
- abananas
- amabananas
- anamabananas
- ananas
- anas
- as
- bananas
- mabananas
- namabananas
- nanas
- nas
- panamabananas
- s

- Does the substring "nana" appear in the original string? How?
- How many times does "ana" appear in the string?
- What is the most/least frequent letter in the original string?
- What is the most frequent two-letter substring in the original string?
Properties of a Naive sorted suffix implementation

- Size of the sorted list if the given text has a length of \( m \)? \( O(m^2) \)
- Cost of the sort? \( O(m^2 \log(m)) \)
- Not practical for big \( m \)
- There are many ways to sort
  - What is an \textit{in place} sort?
  - What is a \textit{stable} sort?
  - What is an \textit{arg sort}?
Arg Sorting

Consider the list:

\[ [7, 2, 4, 3, 1, 5, 0, 6] \]

When sorted it is simply:

\[ [0, 1, 2, 3, 4, 5, 6, 7] \]

Its arg sort is:

\[ [6, 4, 1, 3, 2, 5, 7, 0] \]

- The \( i \)th element in the arg sort is the index of the \( i \)th element from the original list when sorted.
- Thus, \([A[i] \text{ for } i \text{ in argsort}(A)] == \text{sorted}[A]\)
def argsort(input):
    return sorted(range(len(input)), cmp=lambda i,j: 1 if input[i] >= input[j] else -1)

A = [7,2,4,3,1,5,6,8]
print argsort(A)
print [A[i] for i in argsort(A)]

print
B = ['TAGACAT', 'AGACAT', 'GACAT', 'ACAT', 'CAT', 'AT', 'T']
print argsort(B)
print [B[i] for i in argsort(B)]

[6, 4, 1, 3, 2, 5, 7, 0]
[0, 1, 2, 3, 4, 5, 6, 7]

[3, 1, 5, 4, 2, 6, 0]
['ACAT', 'AGACAT', 'AT', 'CAT', 'GACAT', 'T', 'TAGACAT']
Next Time

- We'll see how arg sorting can be used to simplify representing our sorted list of suffixes
- Suffix arrays
- Burrows-Wheeler Transforms
- Applications in sequence alignment