Frequency Counting

- Many problems can be solved by counting the number of times each character appears in a string—the order does not matter.
- e.g. Anagram recognition

Example: GNU = GNU'sNotUnix (10625)

- Given a number of rules $x \rightarrow S$ (x a letter, S a string) and a starting string s, how many times does a specific letter appear after all rules are applied n times?
- The result of rule application depends only on the frequency of each letter.
- Can represent the frequency count as a vector of 128 elements.
- Can represent the rule application as a matrix.
- Use fast matrix exponentation.

Input Parsing

- Usually, a grammar is given for the language.
- Each grammar rule contains a variable and a number of “forms”—they may contain other variables.
- Typically: write a function for each variable, and recursively call the functions for other variables.
- Sometimes you may have to try each rule, or multiple ways to apply a rule.
- Recursive approach may not be the most efficient, but for short strings it is usually sufficient.

Example: Slurpys (384)

- You are given a three "variables"—slurpy, slump, slimp.
- Write a function to check each kind. They may call each other recursively.
- A slurpy is a slimp followed by a slump: try all possible ways of partitioning the input string into two parts and check.

Example: Number of Paths (10854)

- Given the source code of a program with (possibly nested) IF-THEN-ELSE statements, how many different execution paths are there?
- Read all the keywords into a vector of strings.
- Look for the "outer" IF-THEN-ELSE blocks. For each block, multiply the number of paths together (they are independent).
- Keep track of "nesting level": increment for "IF" and decrement for "END_IF".
- Recursively find the number of paths in each branch, add the results.

String Matching

- Given strings s and t (lengths n and m), does t appear as a substring of s? If so, where is the first occurrence?
- Standard `string::find()`: $O(nm)$.
- KMP algorithm: $O(m)$ preprocessing time, $O(n)$ time per search (`kmp.cc`).
- Especially useful if we are searching for the same t in multiple strings.

Longest Common Substring

- Given two strings s and t of lengths m and n, what is the longest common substring? (Note: not subsequence)
- This can be solved by dynamic programming.
- Let $f(i,j)$ be the length of the longest substring ending at $s[i]$ and $t[j]$.
- Base case: $f(i,j) = 0$ if $i < 0$ or $j < 0$.
- Recurrence:

$$f(i,j) = \begin{cases} 1 + f(i-1, j-1) & \text{if } s[i] = t[j] \\ 0 & \text{otherwise.} \end{cases}$$

- Look for the maximum value of $f(i,j)$.
- Complexity: $O(mn)$. We will see a better way later.

Edit Distance

- Given two strings s and t of lengths m and n, what is the minimum number of operations to modify s to t:
  - Change a character
  - Insert a character
  - Delete a character
- This can be solved by dynamic programming.
- Example: String Distance and Transform Process (526).

Edit Distance

- Let $f(i,j)$ be the edit distance of $s[0,\ldots,i-1]$ and $t[0,\ldots,j-1]$. We are interested in $f(m,n)$.
- Base cases: $f(i,0) = i$ (delete), $f(0,j) = j$ (insert).
- Recurrence:

$$f(i,j) = \min(f(i-1,j-1)+(s[i-1] \neq t[j-1]), f(i,j-1)+1, f(i-1,j)+1)$$

corresponding to change, insert, and delete a character.

- To recover the operations, remember which of the three options led to the minimum at each step.

Repeated Searches

- Sometimes we have very long strings but we want to do repeated searches within a string.
- e.g. s has n characters, and we want to know if each of $t_1, \ldots, t_m$ (lengths $n_1, \ldots, n_m$) appears as a substring of s.
- Running KMP m times would result in a complexity of $O((n_1 + \ldots + n_m) + nm)$.
- We can pre-process the string s into a different data structure to facilitate with searches.

Suffix Arrays

- Given a string s, we want to consider all non-empty suffixes.
- e.g. $s =$ `"banana"`. The suffixes are: `"banana"`, `"anana"`, `"nana"`, `"ana"`, `"na"`, `"a"`.
- Notice that a substring of s is simply a prefix of some suffix.
- To search for a string t in s, we can ask instead:
  "is t a prefix of some suffix in s?"
- Why is this any better?

Suffix Arrays

- Suppose we sort all of the n suffixes:
  - `"a"`
  - `"ana"`
  - `"anana"`
  - `"banana"`
  - `"na"`
  - `"nana"`
- To search for a prefix, we can use binary search. Complexity: $O(|t| \log n)$.
- Example: $t =$ `"ana"`
- To search for strings $t_1, \ldots, t_m$ in s, we only need $O((n_1 + \ldots + n_m) \log n)$, after suffix array is constructed.

Constructing Suffix Arrays

- Each suffix can be identified by the index of the first character in the original string.
- The array can be represented as an array of integers of size n.
- Simply sorting the suffixes: $O(n^2 \log n)$ because each comparison in a sorting algorithm is $O(n)$.
- We need a better way.

Constructing Suffix Arrays

- First, we sort each suffix based on first 2 characters in $O(n)$ operations with radix sort.
- Next we sort each suffix based on first 4 characters—equivalent to first 2 pairs.
- Note that from the first sort, we have a "rank" for each pair so we can apply radix sort again.
- Double the number of characters examined each time.
- Overall complexity: $O(n \log n)$.
- See code in textbook. Note that the code assumes `'.'` is not in the string.
- `suffixarray.cc` in library: $O(n)$ construction.

Longest Common Prefix

- The longest common prefix (LCP) array is useful for many applications.
- LCP(i) is the length of the longest common prefix between the suffixes at positions i and $i-1$ in the suffix array.

i	Suffix	SA[i]	LCP[i]
0	a	5	0
1	ana	3	1
2	anana	1	3
3	banana	0	0
4	na	4	0
5	nana	2	2

Longest Common Prefix

- The LCP array can be computed in $O(n)$ time once the suffix array is constructed (see `suffixarray.cc`).
- The nonzero LCP values indicate repeated occurrences of a substring.
- A contiguous sequence of k nonzero LCP values means that there is a substring that occurs $k+1$ times.
- The length of that substring is the minimum of those LCP values.

Example: Glass Beads (719)

- Given a string s of length n, find the lexicographically smallest rotation.
- Brute force: generate all n rotations, sort them. Too slow for this problem.
- Trick: look at the string ss. A rotation is just a substring of length n.
- Compute the suffix array for ss, and look for the first suffix that has length at least n. The first n characters give the answer.
- Complexity: $O(n)$.

Example: GATTACA (11512)

- Given a long string, find the longest substring that occurs at least twice.
- Compute the suffix array and the LCP array, and look for the maximum value in the LCP array.
- If there is a tie, choose the first one (lexicographical order).

Longest Common Substring

- Given two strings s and t of lengths m and n, what is the longest common substring?
- We know it can be done in $O(mn)$ operations.
- Trick: Form the string `s#t` where `#` is a character not found in either s or t.
- Now look for the longest repeated substring.
- How do we know that we don't choose two occurrences that both occur in s (or t)?
- We consider `LCP[i]` if and only if `SA[i-1]` and `SA[i]` belong to different parts of the string.