Suffix trees are used in a variety of contexts

E.g., in the analysis of sequences (of different kinds).

ALL PAIRS SUFFIX-PREFIX PROBLEM

PROBLEM:

INPUT: $S_1, S_2, \ldots, S_n$.

OUTPUT: $\forall i,j$: the longest suffix of S_i that is a prefix of S_j

An important application is in *de novo* sequence assembly. The input to this problem will be subsequences (called reads) from a genomic sequence. The goal is to construct the genomic sequence from the reads. In order to do this reconstruction we need overlap information among the reads. We can represent the overlaps in a graph, where each node corresponds to a read. Two nodes will be connected by an edge if the two reads have a sufficient overlap. Once this graph is constructed we can traverse through the graph looking for long paths. Ideally, we would like to see a path going through every node (but this may not happen).

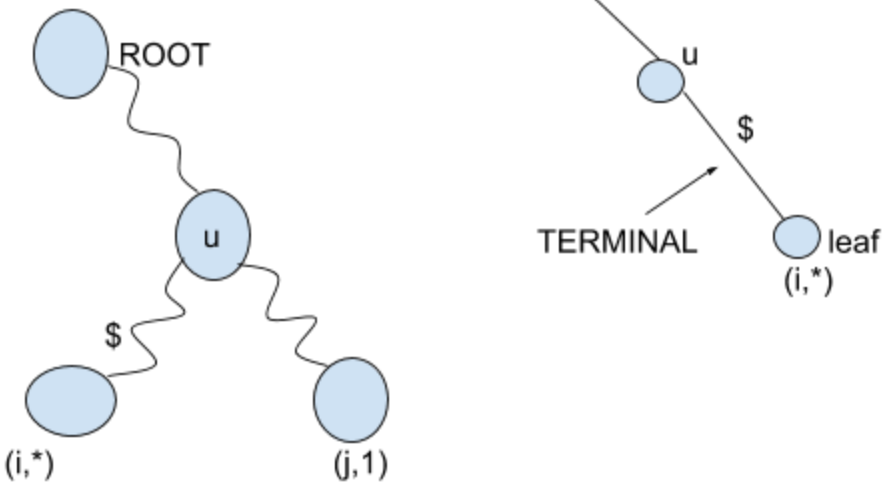
The size of the output is $O(n^2)$.

FACT: We can solve this problem in $O(M+n^2)$ time, where $M = \sum_{i=1}^{n} |S_i|$.

PROOF: An algorithm - construct a generalized suffix tree Q on $S_1, S_2, \ldots, S_n$.

This takes O(M) time. For any j, there is a path in Q for S_j.

We call an edge a TERMINAL if it is labeled with $.



The path label of u is a suffix of S_i. Also, the path label of u is a prefix of S_j.
If from among all such nodes u has the largest string depth, then the path label of u is the largest suffix of S_i that is a prefix of S_j.

For every $u \in Q$, define L(u) as:

$i \in L(u)$ if u has a terminal edge corresponding to a suffix of S_i.

For every u we can compute L(u) in one traversal through the tree. The maximum time spent at each node is proportional to the degree of the node. The sum is therefore all the edges, and we have a runtime of O(n).

Do one more depth-first search (DFS) through the tree. Keep *n* STACKS, one for each S_i where $1 \leq i \leq n$.

When node u is visited in the FORWARD DIRECTION, push u into stack i for $\forall i \in L(u)$.

If we visit a leaf labelled (j,1), the path from the root to this node corresponds to the entire string S_j. When we reach a leaf labeled (j,1) for any j, the top of stack i has the information about the longest suffix of S_i that is a prefix of S_j, $1 \leq i \leq n$. Specifically, if u is on top of stack i, then the path label of u is the longest suffix of S_i, which is a PREFIX of S_j.

$\forall j \ \exists$ label (j,1) $\rightarrow$ only one traversal

If a stack k is empty, it means that there is no suffix of S_k that is a prefix of S_j.

When we visit the node u in the reverse direction, we pop the top of stack i for $\forall i \in L(u)$.

Outputting all such pairs will take $O(n^2+M)$ time.

SUFFIX ARRAYS

While suffix trees are easier for visualizing algorithms, there is an issue if the alphabet is large. In a pattern search, we match characters in the desired pattern with the corresponding characters in a tree.

Consider a large alphabet Σ. How will we look for an edge in the suffix tree that starts with the specific character *x*? This can be done in unit time if we have a bit array of size |Σ| at each node, where Σ is the alphabet. There will be an index in this array corresponding to every character in the alphabet. If this node has a child and the edge to this child has a label starting with a character c, then the corresponding entry in the bit array will be 1. For example, let Σ ={a, b, c, d, e, f, g}. If a node has only two children and if the corresponding edge labels start with the characters c and e, respectively, then the bit array for this node will be:

a	b	c	d	e	f	g
0	0	1	0	1	0	0

...

This approach calls for a memory of $\ominus(M|\Sigma|)$, where M=|T|, T being the input text. Alternatively, if we can use O(M) memory if we are willing to spend more time at each node. For example, if the children of any node are ordered based on the first characters of the edge labels, then we can do a binary search at each node to locate a character of interest. In this case, the time needed to perform a string match will

be O(n log (|Σ|)). In summary, either we require more memory or more time for larger alphabets.

Suffix arrays circumvent this problem entirely. Suffix arrays are arrays of integers. Let $S = a_1\ a_2\ \ldots\ a_n$ be any string from an alphabet Σ.

We can define a lexical ordering among the suffixes of S. The suffix array of S is an array of SA[1:n] such that SA[i] is the starting position in S of the i^{th} smallest suffix of S.

Example: S = a a g c c g t t a g a c (where a<c<g<t)

1 2 3 4 5 6 7 8 9 10 11 12

i	SA[i]	Suffix
1	1	aagccgttagac
2	11	ac
3	9	agac
4	2	agccgttagac
5	12	c
6	4	ccgttagac
7	5	cgttagac
8	10	gac
9	3	gccgttagac
10	6	gttagac
11	8	tagac
12	7	ttagac

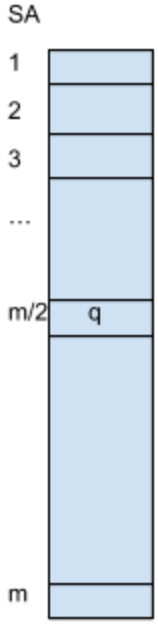
FACT: We can construct a suffix array in O(n) TIME on any string of length n.
PROOF: Construct a suffix tree on S in O(n) TIME. Followed by this, do a lexical DFS on Q. At any node, always use the next edge with the least starting character. In this case, the leaves will be visited in lexical order.

STRING MATCHING

INPUT: a text $T = t_1\ t_2 \ldots t_m$ and a pattern $P = p_1\ p_2 \ldots p_m$.
OUTPUT: all the occurrences of P in T.

CLAIM: We can use a suffix array to solve this problem.

IDEA: Do a BINARY SEARCH using the array. (The principle of a binary search is basically the same as a normal binary search except that here a comparison involves the comparison between two strings.)



Go to position m/2. Try to match the pattern character by character starting from position q in T. If the pattern cannot be fully matched, in the first position where the match fails, let the character in T be x and the character in P be y.

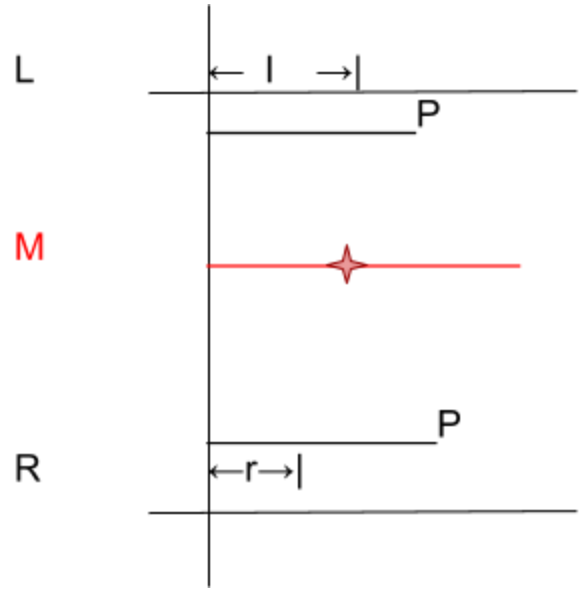
$t_1\ t_2 \ldots t_q \ldots$ x.... t_m
$p_1\ p_2 \ldots$ y $\ldots\ p_n$

Two cases:
x<y: pattern>suffix, so confine the binary search to bottom half of the array.
x>y: do a SEARCH in the top half.

TIME needed for the binary search = O(n log m) where O(log m) is the number of comparisons, and each comparison involves up to n character comparisons.

CLAIM: We can do pattern matching in O(n+log m) TIME.

PROOF:



where M = ⌊(L+R)/2⌋.

In the binary search we always work with an interval [L, R], where L and R are two suffixes of the text T. L and R are the starting positions in T of two suffixes. To begin with L=1 and R=m. The pattern P falls is between L and R. As the binary search dictates, P will be compared next with the suffix M where M=⌊(L+R)/2⌋. As the binary search progresses, the range [L, R] shrinks further and further.

Let l be the length of the longest common prefix of L and P.
Let r be the length of the longest common prefix between P and R.
Let mlr = Min{l, r} and let MLR = Max{l, r}.

If l>r, then, the first *r* characters will be the same for all the suffixes L through R. So, when we compare P with M, we can start the comparison from position mlr+1. This observation already will improve the binary search time nicely in practice. If we can ensure that we always compare P with M starting from position MLR+1, then we can show that the time needed is O(n+ log m). We'll show this in the next lecture.