Week 7 Studio Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, psuedocode may be provided where it illustrates a particular useful concept.

Implementation checklist

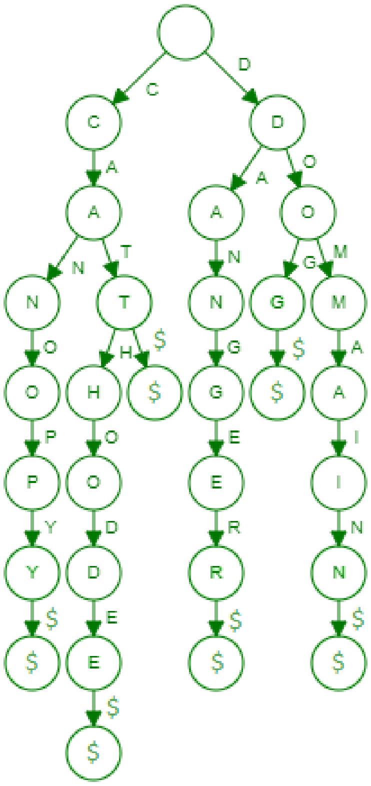
It will be most beneficial for your learning if you have completed this checklist **before** the studio.

By the end of week 7, write Python code for a Trie structure (recommended to use a class). It should have an insert method and a search method.

Assessed Preparation

Problem 1. Draw a prefix trie containing the strings `cat`, `cathode`, `canopy`, `dog`, `danger`, `domain`. Terminate each string with a $ character to indicate the ends of words.

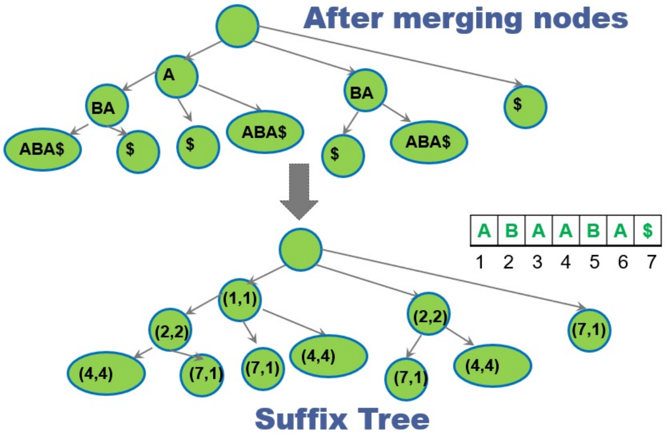
Solution



Source: `https://www.cs.usfca.edu/~galles/visualization/Trie.html`

Problem 2. Draw a suffix tree for the string `ABAABA$`.

Solution



Studio Problems

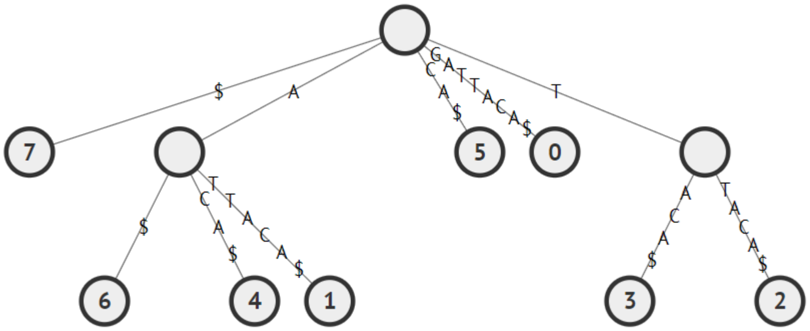
Problem 3. Describe an algorithm that given a sequence of strings over a constant-size alphabet, counts how many different ones there are (i.e. how many there are ignoring duplicates). Your algorithm should run in $O(T)$ time where T is the total length of all of the strings.

Solution

Initialise `count` = 0. Append "\$" to each string. Insert each string into a trie. After all strings have been inserted, traverse the tree and count the number of "\$" nodes. If two strings are the same, they will terminate in the same node, so the number of "\$"s is the number of distinct strings. It will take $O(T)$ to build the trie, and $O(T)$ to traverse the trie, so the whole algorithm runs in $O(T)$.

Problem 4. Draw a suffix tree for the string `GATTACA$`.

Solution



Source: `https://visualgo.net/en/suffixtree`

The numbers at the leaf nodes are the suffix IDs (assuming first suffix has ID 0). This visualisation merges the nodes that form a chain but do not show tuple representation of the branch. Write the tuple representation yourself. In exams, you must write the nodes in tuple (x,y) representation.

Problem 5. Describe an algorithm that given a string S of length n, counts the number of distinct substrings of S in $O(n)$ time.

Solution

First let's consider how to solve this with a suffix trie, and then convert our solution to work in a suffix tree. A substring of S is a prefix of a suffix of S. This means that each substring of S can be found by following some path from the root of a suffix trie of S to some node (possibly leaf, possibly internal). If two substrings of S are the same, they will end at the same node. So to count distinct substrings, we can just count the nodes of the suffix trie, which takes $O(n^2)$ time.

Now we want to use a suffix tree instead. If we think about converting a trie to a tree, each node in the tree is made from one or more nodes in the trie, and the number of characters stored at each node in the tree is exactly the number of nodes which were compressed to create that tree node. So to do the equivalent of counting nodes in the trie, we need to count total characters in the tree. We can do this with a single traversal taking $O(n)$ time, because a suffix tree has $O(n)$ nodes, and at each node, we look up the indices of its parent edge label to determine its length, so counting the characters in a node takes $O(1)$.

Problem 6. Earlier, in Studio 4, we learned about the longest common **subsequence** problem. Now we consider a similar, related problem, the *longest common substring* problem. Given two strings s_1 and s_2, your goal is to find a string that is a substring of both s_1 and s_2 and is as long as possible. Describe how a suffix tree can be used to solve this problem. Your algorithm should run in $O(n+m)$ time, where n, m are the lengths of s_1, s_2 respectively. You may assume that you can construct a suffix tree in linear time.

Hint: It is often easier to think about how to solve problems in a trie context, and then convert your solution to work in a tree.

Solution

Finding the longest common substring of s_1 and s_2 is very similar to finding the longest repeated substring of $s_1 s_2$, which we know can be found by creating a suffix tree and then finding the deepest node with more than one child. The issue with naively applying this to $s_1 s_2$ is that, in the concatenated string, the longest repeated substring might lie totally in s_1 (or s_2), or partially in s_1 and partially in s_2. We can solve the second problem easily by adding a seperator character (which is not present in either string) in our concatenation, so that we have $s_1 \# s_2 \$$.

An important observation about the placement of "#" and "$" in the tree is that no internal node can contain either of them. That is, all instances of these terminator characters must appear in leaves. This is clearly true for "$", since it terminates the whole string. For "#", since it only appears once in the string, and since we know that all internal nodes in a tree have at least two children, i.e. they correspond to repeating substrings, we know that "#" cannot be in any internal node, as it does not repeat.

Bearing this observation in mind, we need to address the problem of ensuring that one of our repetitions is in s_1, and the other is in s_2. Consider an interal node of the tree. If it has a leaf descendant node with a "#", then it must appear in s_1. If it has a leaf descendant without a "#", then it must appear in s_2. So we need to find the deepest such node.

We traverse the tree, and at each node we store two flags, one for being a substring of s_1 and the other for being a substring of s_2. If a node is a substring of s_1, all its ancestors are substrings of s_1, and similarly for s_2. Once we have finished we return the depth of the node with the longest substring that was a substring of both s_1 and s_2, i.e. had both flags set.

It is worth having a think about how to set the flags for every node in the tree in linear time, and also how to check whether a leaf node contains a "#" in constant time.

Problem 7. Describe how to modify a prefix trie so that it can performs queries of the form "count the number of strings in the trie with prefix p". Your modification should not affect the time or space complexity of building the trie, and should support queries in $O(m)$ time, where m is the length of the prefix.

Solution

All words with a prefix p are leaves in the subtree rooted at the node containing the last character in p. So we have to count the leaves in the subtree of that node. We could traverse the trie to get to the last node of p in $O(m)$ and then do a complete traversal of the subtree, but this would take much longer than $O(m)$, possibly even taking $O(T)$ where T is the number of characters in the tree.

Instead, we modify the insertion algorithm. We will store a counter at each node of our trie, initialised to 0. When inserting a word, whenever we move to or create a node, we increment the counter at that node by 1. This counter tracks the number of words which have the prefix corresponding to that node. Then, to count the number of strings with prefix p, we just traverse the tree to the last node of p which takes $O(m)$ time, and return the count of that node.

Problem 8. Given a trie and a query string, we want to find the lexicographically greatest string in the trie which is lexicogrphically less than or equal to the query string.

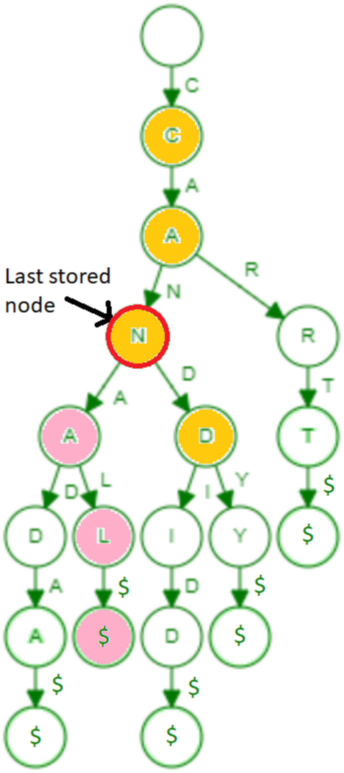
Your data structure should perform these queries in $O(n+m)$ time, where m is the length of the query string, and n is the length of the answer string (the predecessor). It is possible that some queries have no string which satisfies the criterion above (if it is less than all of the strings in the trie), in which case you should return **null**.

Example: Suppose the trie contains "aaa", "aba", "baa". If the query is "ba", the result is "aba". If the query is "baa", the result is "baa".

Solution

Let's assume that the contents of our trie are terminated with a $\$ \notin \Sigma$. We traverse the query string, keeping track of the deepest node with a child lexicographically less than the child we are about to move to (initially **null**). This node is the longest prefix that the query string shares with the answer. We either end up failing at some point (the current node has no child with the character we want), or we reach the end of the query string. If we reach the end of the query string (including the "$") we return the query. Otherwise we look at the node we were keeping track of, and if it is **null**, we return **null** since the word must have no predecessor, as none of its prefixes could be extended with a lexicographically lesser character. If such a node does exist, go back to that node, go along the edge with the greatest character such that it is lexicographically less than the character we followed in our initial traversal. Then follow the lexicographically greatest path from that node to a leaf. The string corresponding to that path is the solution. This takes $O(m)$ to traverse the query and $O(n)$ to traverse the answer string.

In the below example, the query is `candelabra`. After the traversal, `n` is the deepest node with a child lexicographically less than the node we traverse to (`d` does not satisfy this condition since its children are both greater than `e`). So after traversing `cand` we go back to `n` and then traverse until we find a leaf, always choosing the lexicographically greatest option, which yields `canal`.



Source: `https://www.cs.usfca.edu/~galles/visualization/Trie.html`

```
function PREDECESSOR($S[1..n]$)
```

```
    node = root
    ancestor = null    // Lowest ancestor with an extension lexicographically less than $S$
    ancestor_child = null
    answer = “”
    for each character c in $S[1..n]$ do
        if node has an edge with label < c then
            ancestor = node
            ancestor_child = the greatest child of ancestor lexicographically less than c
        end if
        if node has an edge for character c then node = node.get_child(c)
        else break
        if c = \$ then return answer    // $S$ is in the trie
        else answer += c
    end for
    if ancestor = null then return null    // Nothing is lexicographically less than $S$
    while node $\neq$ ancestor do
        node = node.parent
        answer.pop_back()    // Remove last character from answer
    end while
    answer += ancestor_child.character
    node = ancestor_child
    while node is not a leaf do
        c = lexicographically greatest edge of node
        answer += c
        node = node.get_child(c)
    end while
    return answer.pop_back()    // Remove the \$ from the answer string
end function
```

Problem 9. You are given a sequence of n strings $s_1, s_2, ..., s_n$ each of which has an associated weight $w_1, w_2, ..., w_n$. You wish to find the “most powerful prefix” of these words. The most powerful prefix is a prefix that maximises the following function

$$\text{score}(p) = \sum_{\substack{s_i \text{ such that } p \\ \text{is a prefix of } s_i}} w_i \times |p|,$$

where $|p|$ denotes the length of the prefix p. Describe an algorithm that solves this problem in $O(T)$ time, where T is the total length of the strings $s_1, ..., s_n$. Assume that we have three strings `baby`, `bank`, `bit` with weights 10, 20 and 40, respectively. The score of prefix `ba` is $2 \times 10 + 2 \times 20 = 60$, the score of prefix `b` is $1 \times 10 + 1 \times 20 + 1 \times 40 = 70$ and the score of prefix `bit` is $3 \times 40 = 120$.

Solution

For each prefix, we need to keep track of its score. We modify our prefix trie to store a score at each node, and each time we insert a word, we update the score by adding to it the weight of the current word multiplied by the length of the prefix represented by this node.

Once all insertions are complete, we traverse the tree to find the node with the maximum score. The prefix corresponding to this node is the answer.

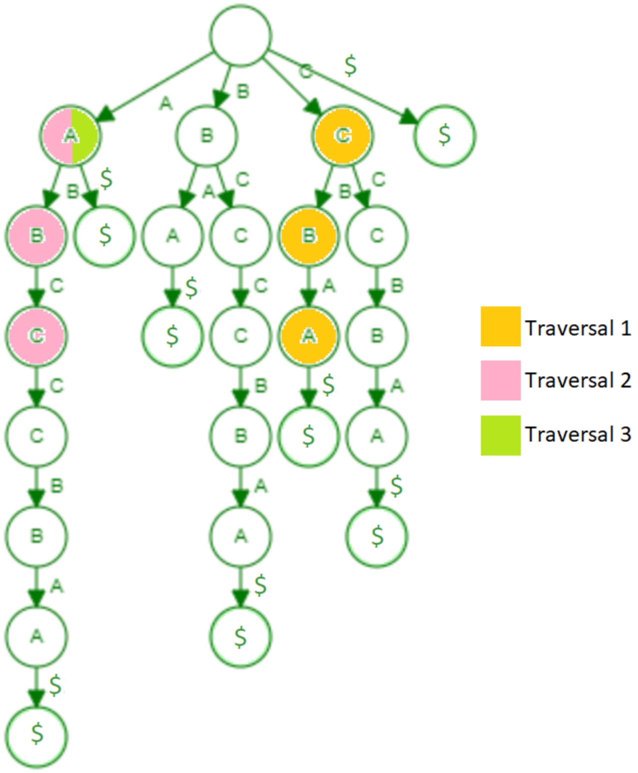
Supplementary Problems

Problem 10. Given a string S and a string T, we wish to form T by concatenating substrings of S. Describe an algorithm for determining the fewest concatenations of substrings of S to form T. Using the same substring multiple times counts as multiple concatenations. For example, given the strings $S =$ `ABCCBA` and $T =$ `CBAABCA`, it takes at least three substrings, e.g. `CBA + ABC + A`. Your algorithm should run in $O(n+m)$ time, where n and m are the lengths of S and T. You can assume that you have an algorithm to construct suffix tree in linear time.

Solution

We first make the observation that we can be greedy. If we are up to a certain point $T[j...]$, then it is optimal to take the longest possible substring from S that matches $T[j..k]$ for the largest possible k. First build a suffix tree from S, taking $O(n)$ time, then search for the string T in the tree. This search is going to find the longest substring of S which corresponds to the start of T. Once we reach a point where the search stops, this means that the current substring cannot be extended, so we go back to the root and search again from the next character of T. We repeat this process until we reach the last character in T.

Below is an illustration of the example case given in this problem.



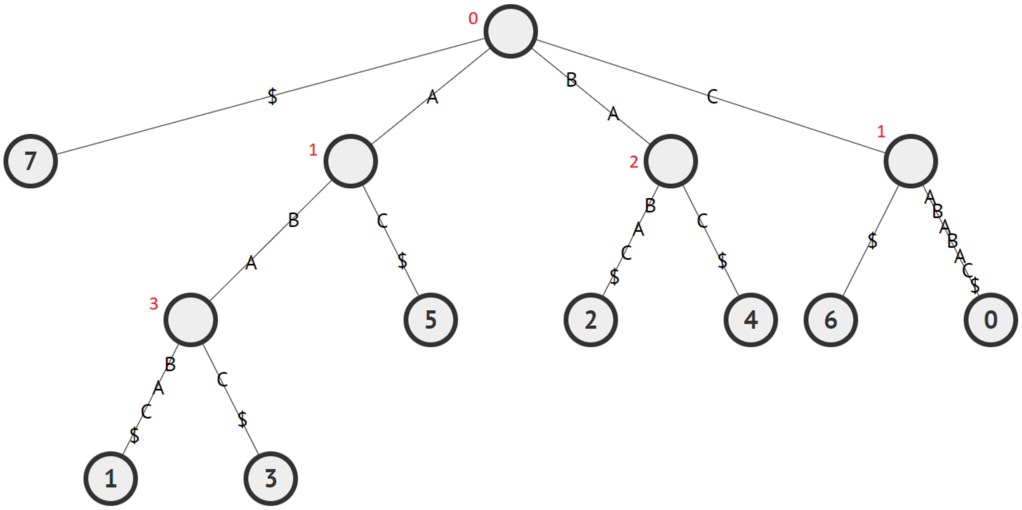
Source: `https://www.cs.usfca.edu/~galles/visualization/Trie.html`

Problem 11. Describe an algorithm for finding a shortest unique substring of a given string S. That is, finding a shortest substring of S that only occurs once in S. For example, given the string `cababac`, the shortest unique substrings are `ca` and `ac` of length two. Your algorithm should run in $O(n)$ time, where n is the length of S. You can assume that you have an algorithm to construct suffix tree in linear time.

Solution

Consider a suffix trie of S. A unique substring corresponds to a node that has only one leaf below it. We want to find the highest such node, since this will be the shortest such substring.

Define a candidate node as a node which has a leaf as a child, whose edge label contains more than just \$. In a suffix tree, all internal nodes have two or more children. So we want to find the shallowest candidate node, i.e. the candidate node which has the fewest characters on the path from the root. Let's traverse the tree, tracking the shallowest candidate. When we finish the traversal, the correct length is the length stored in the shallowest candidate + 1. This is because we need to include the first character in one of the leaf edges, since internal nodes are non-unique substrings. The substring stored on the path from the root to the shallowest candidate, including the first character in any of its descendants is a shortest unique substring.



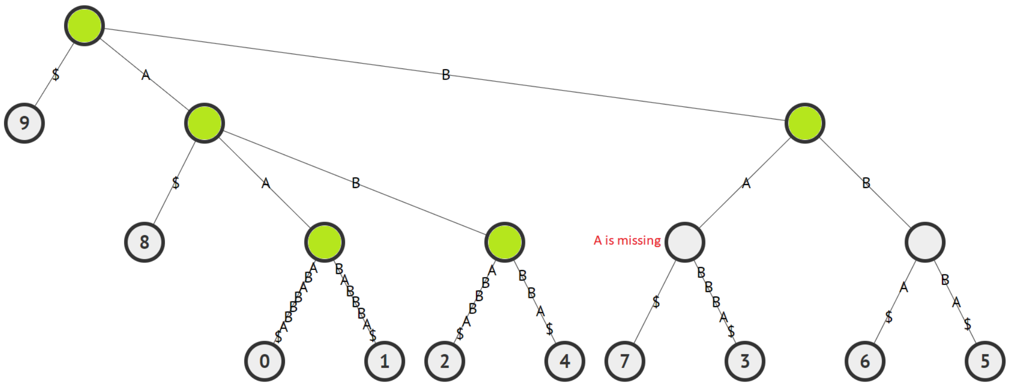
Source: `https://visualgo.net/en/suffixtree`

Red numbers indicate the length we are tracking as we traverse the trie. When we finish, there are two candidate nodes with depth 1, so the length of the shortest non-repeating substring is 2, and the solutions is either `AC` or `CA`

Problem 12. Describe an algorithm for finding a shortest string that is not a substring of a given string S over some fixed alphabet. For example, over the alphabet {`A`,`B`} the shortest string that is not a substring of `AAABABBBA` is `BAA` of length three. Your algorithm should run in $O(n)$ time, where n is the length of S.

Solution

If we traverse the tree, the shallowest node we find that does not have every character in the alphabet as a child will give us the shortest missing substring. This is because, if a node has every alphabet character as a child, then we can extend the substring at that node to every possible substring with 1 more character. If one or more alphabet characters are missing, we cannot do this, and so we have found a missing substring. The missing substring is given by the string at that node + the missing alphabet character. Note that the number of nodes in the suffix tree is $O(n)$. So, the worst-case cost is $O(n)$.



Source: `https://visualgo.net/en/suffixtree`

In the example above, the shortest missing string is BAA

Problem 13. (Advanced) Prefix tries also have applications in processing integers. We can store integers in their binary representation in a prefix trie over the alphabet $\{0,1\}$. Use this idea to solve the following problem. We are given a list of w-bit integers $a_1, a_2, ..., a_n$. We want to answer queries of the form "given a w-bit integer x, select one of the integers a_i to maximise $x \oplus a_i$, where $\oplus$ denotes the XOR operation. Your queries should run in $O(w)$ time.

Solution

To maximise a binary number, we want to set 1 bits in the highest slots possible. This means we should select the a_i which has the opposite bit to x in the highest possible positions as we read from left to right.

If we have a prefix trie containing $a_1, a_2, ..a_n$, using x we can traverse the trie as follows. Each time we come to a node, see if the node has a child which is NOT(x[i]) (i.e. 0 if $x[i] = 1$, or 1 if $x[i] = 0$). If so, choose that child. If not, the node has only one child, so go to that node. One way to view this is like the ordinary trie traversal but opposite. We always chose to go to a child with a **different** label than the current character, instead of the one with the same character if possible.

When we reach a leaf, the choice of a_i which maximises a_i XOR x is that leaf. Since we traversed a single path from root to leaf, our query has taken $O(w)$ time.

Problem 14. (Advanced) Given a list of w-bit integers $a_1, a_2, ..., a_n$, find the maximum possible cumulative XOR of a subarray $a_i, ..., a_j$, i.e. maximise

$$a_i \oplus a_{i+1} \oplus ... \oplus a_{j-1} \oplus a_j,$$

where $\oplus$ denotes the XOR operation. Your algorithm should run in $O(nw)$ time. Hint: Use the fact that $x \oplus x = 0$, and the ideas from Problem 13.

Solution

First, some properties of XOR. It is associative, meaning that $((a \oplus b) \oplus c) = (a \oplus (b \oplus c))$, and, as given in the question, $x \oplus x = 0$. This means that if we have a list of numbers all being XOR'ed together, and some of these number are identical, we can reorder the list using associativity until the pairs of identical numbers are next to each other, and then cancel them using the second property.

Define a function

$$F(L,R) = a_L \oplus a_{L+1} \oplus ... \oplus a_R$$

The problem we are trying to solve is now to find an L and R which maximise F, where $1 \leq L \leq R \leq n$. Note that because of the facts above, we have

$$F(L,R) = F(1,L-1) \oplus F(1,R)$$

So if we had all the values $F(1,1), F(1,2) ... F(1, j-1)$ in a trie, we could compute the maximum value for $F(i,j), i \leq j$ in $O(w)$ with a query to the trie described in Problem 13. Then we simply iterate over $1 \leq i \leq n$, keeping track of the maximum, which takes $O(nw)$.

```
function MAX_SUBARRAY_XOR($A[1..n]$)
    prefix_xor = 0
    trie = Trie()    // The trie from Problem 13
    maximum_xor = 0
    for i=1 to n do
        prefix_xor = prefix_xor $\oplus A[i]$    // Extend the current prefix
        maximum_xor = max(maximum_xor, trie.query(prefix_xor))
        trie.insert(prefix_xor)
    end for
    return maximum_xor
end function
```

Problem 15. **(Advanced)** In o 5, we wrote a dynamic programming solution to the following problem. Given a target string S of length n and a list of m strings of length at most n, find the minimum number of strings from L to concatenate to form S. A word from L may be used multiple times. We devised an $O(n^2 m)$ algorithm for this problem. Describe how you can use a prefix trie to speed up this solution to $O(n^2 + nm)$ time.

Solution

In Studio 5, we wrote the following dynamic programming solution:

$$\mathtt{DP}[i] = \{\text{The minimum number of words needed to form the suffix } S[i..n]\},$$

for $0 \leq i \leq n$.

$$\mathtt{DP}[i] = \begin{cases} 0 & \text{if } i = n, \\ \infty & \text{if no word } w \in L \text{ is a prefix of } S[i..n] \\ 1 + \min\limits_{\substack{w \in L \\ w \text{ prefix of } S[i..n]}} \mathtt{DP}[i + |w|] & \text{otherwise.} \end{cases}$$

A part of this should immediately jump out at us. Note that we check over all strings $w \in L$ that are prefixes of $S[i..n]$. We have a very good data structure for handling this sort of thing, a prefix trie! So, instead of simply looping over all of the words in L and checking whether they are prefixes, we will first insert all of the contents of L into a trie, terminated by $\$ \notin \Sigma$. Since there are m strings of length at most n this takes $O(nm)$ time. Now to solve the subproblem $\mathtt{DP[i]}$, we simply search the trie for $S[i..n]$, and at each character, we check whether the current prefix p is in L (by seeing whether it has a \$ child), and if so, we consider the subproblem $\mathtt{DP[i+|p|]}$. This means that solving each subproblem takes just $O(n)$ time, and since there are n subproblems, this yields a total time of $O(n^2 + nm)$.

Problem 16. **(Advanced)** Recall that we deal with suffix trees because suffix tries are very space inefficient and slow to produce. It is easy to see that $O(n^2)$ is an upper bound on the size of a suffix trie of a string of length n, since there are n suffixes of length at most n. Prove that it is also a lower bound, i.e. that there exists a string of length n such that its suffix trie takes $\Omega(n^2)$ space.

Solution

First, we recall that each node of a suffix trie represents a substring of the string. Therefore a worst-case suffix trie with the most nodes is one for a string that has the most possible distinct substrings. There are many strings that fit this bill, but a famous and interesting set of them are the *De Brujin sequences* (These were previously covered as a FIT2004 assignment). A De Brujin sequence of order m over an alphabet Σ of size k is a sequence of length k^m that cyclically (substrings may wrap around) contains all possible length m strings over Σ exactly once. This means that every substring of a De Brujin sequence of length n or greater is distinct. Take $m = \log_k(n)$, then we get a De Brujin sequence of length n such that all substrings of length at least $\log_k(n)$ are distinct. There are

$$\binom{n+1}{2} - O(n\log(n))$$

such substrings, since there are $\binom{n+1}{2}$ substrings in total, and for each position in the string, there are $\log_k(n)$ positions to which we could form a substring of length $\log_k(n)$, and hence there are at most $O(n\log(n))$ substrings that are not distinct. Since $\binom{n+1}{2} = \Omega(n^2)$, this sequence has at least $\Omega(n^2)$ distinct substrings, and hence its suffix trie will have $\Omega(n^2)$ nodes.