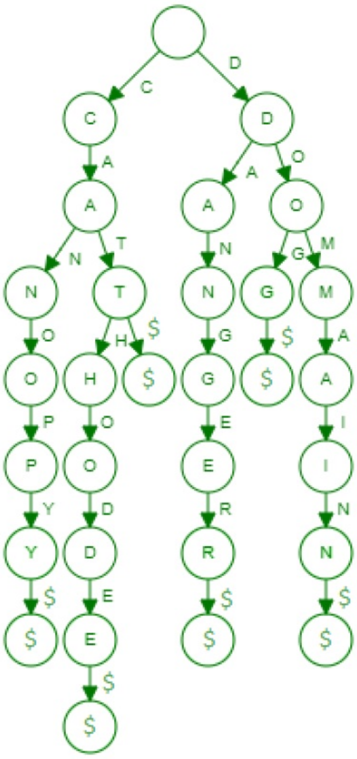
Week 8 Tutorial Solution Sheets

(to be completed during the week 8 tutorial)

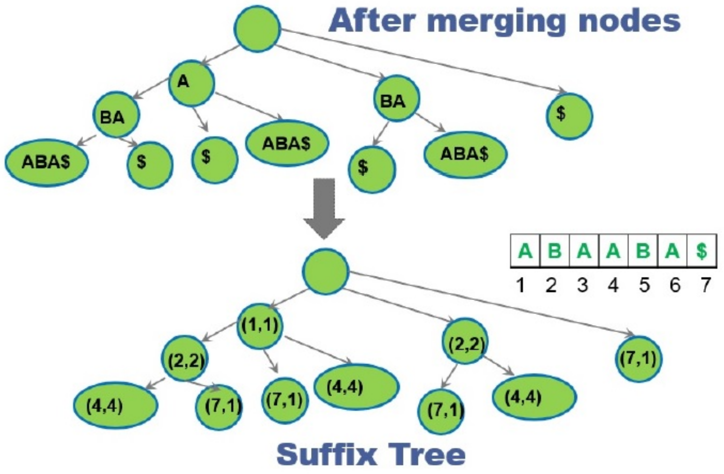
Assessed Preparation

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:



Problem 3. Write the suffix array for the string NRSOOCIMPSE$ and compute its Burrows-Wheeler transform. Also show Show the steps in the inverse Burrows-Wheeler transform for the string.

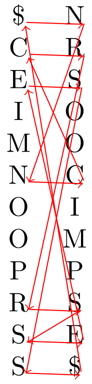
SOLUTION:

i	$SA[i]$	Suffix
0	11	$
1	5	CIMPSE$
2	10	E$
3	6	IMPSE$
4	7	MPSE$
5	0	NRSOOCIMPSE$
6	4	OCIMPSE$
7	3	OOCIMPSE$
8	8	PSE$
9	1	RSOOCIMPSE$
10	9	SE$
11	2	SOOCIMPSE$

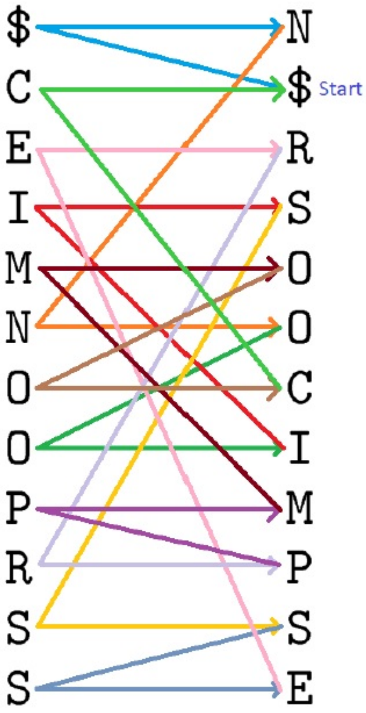
BWT:

$NRSOOCIMPSE
CIMPSE$NRSOO
E$NRSOOCIMPS
IMPSE$NRSOOC
MPSE$NRSOOCI
NRSOOCIMPSE$
PSE$NRSOOCIM
OCIMPSE$NRSO
OOCIMPSE$NRS
RSOOCIMPSE$N
SE$NRSOOCIMP
SOOCIMPSE$NR

Therefore, the BWT of the string is EOSCI$MOSNPR.
Inverse BWT using L-F mapping



The loop will continue and we will not find the solution.
NOTE: If the string is N$RSOOCIMPSE, The inverse BWT using L-F is:

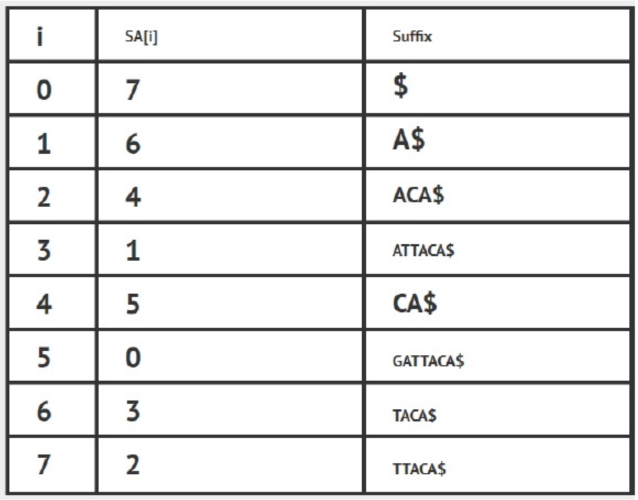


Which yields the string $NOISSERPMOC. Reversing this gives us the original string, COMPRESSION$. The choice of the position of $ is important.

Tutorial Problems

Problem 4. 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 “$” 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 5. Draw a suffix tree for the string GATTACA$ and compute its Burrows-Wheeler transform.
SOLUTION:



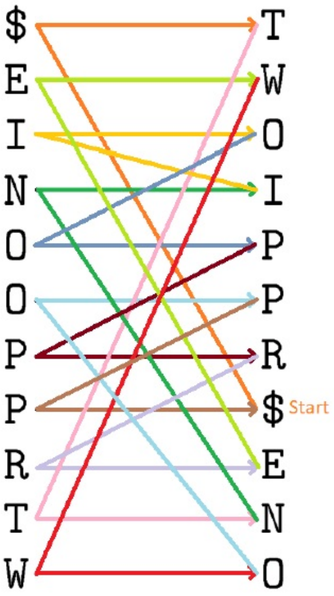
i	SA[i]	Suffix
0	7	$
1	6	A$
2	4	ACA$
3	1	ATTACA$
4	5	CA$
5	0	GATTACA$
6	3	TACA$
7	2	TTACA$

Source: `https://visualgo.net/bn/suffixarray`

The sorted rotations of ACACIA$ are shown below. The BWT is obtained by reading the characters down the right hand side, shown in red. The BWT of the string is therefore `ACTGA$TA`.

```
$GATTACA
A$GATTAC
ACA$GATT
ATTACA$G
CA$GATTA
GATTACA$
TACA$GAT
TTACA$GA
```

Problem 6. Show the steps in the inverse Burrows-Wheeler transform for the string TWOIPPR$ENO.
SOLUTION:
To invert BWT we use the L-F mapping.



Which yields the string $TNIOPREWOP. Reversing this gives us the original string, POWERPOINT$

Problem 7. Compute the Burrows-Wheeler transform of the string woolloomooloo$, and show the steps taken by the pattern matching algorithm for the following patterns:

(a) olo

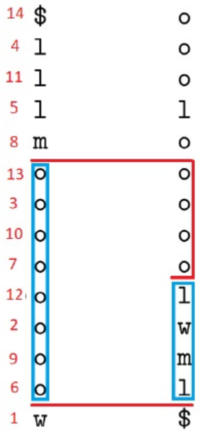
(b) oll

(c) oo

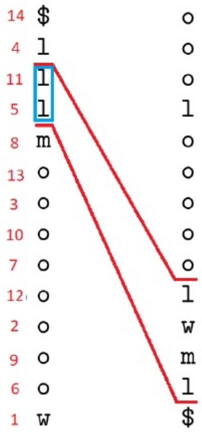
(d) wol

SOLUTION:
The BWT of woolloomooloo$ is ooolooooolwml$. If the location of a substring is also to be determined, during the pre-processing step, in addition to the first and last column, we also maintain the ID of cyclic rotation (i.e., suffix ID) with each row this is the same as a suffix array (see the figures below where the IDs are shown in red). During the query processing, these IDs are used to determine the location as explained later. When searching for patterns we start at the back of the pattern, and search in reverse, using the LF mapping to guide us to occurrences of the previous character each time. Diagrams for each search follow.

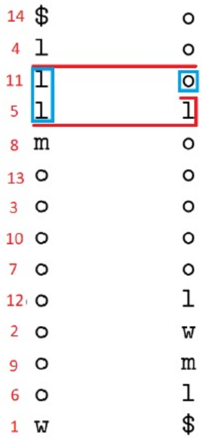
(a) **Searching for** `olo`



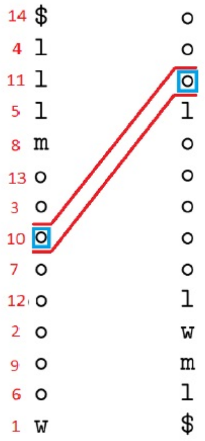
Starting at “o”, we see there are 2 “l”s within the range in the last column, so we contract the range



Using the L-F mapping, our range now contains the cyclic shifts which have “lo” as a prefix.

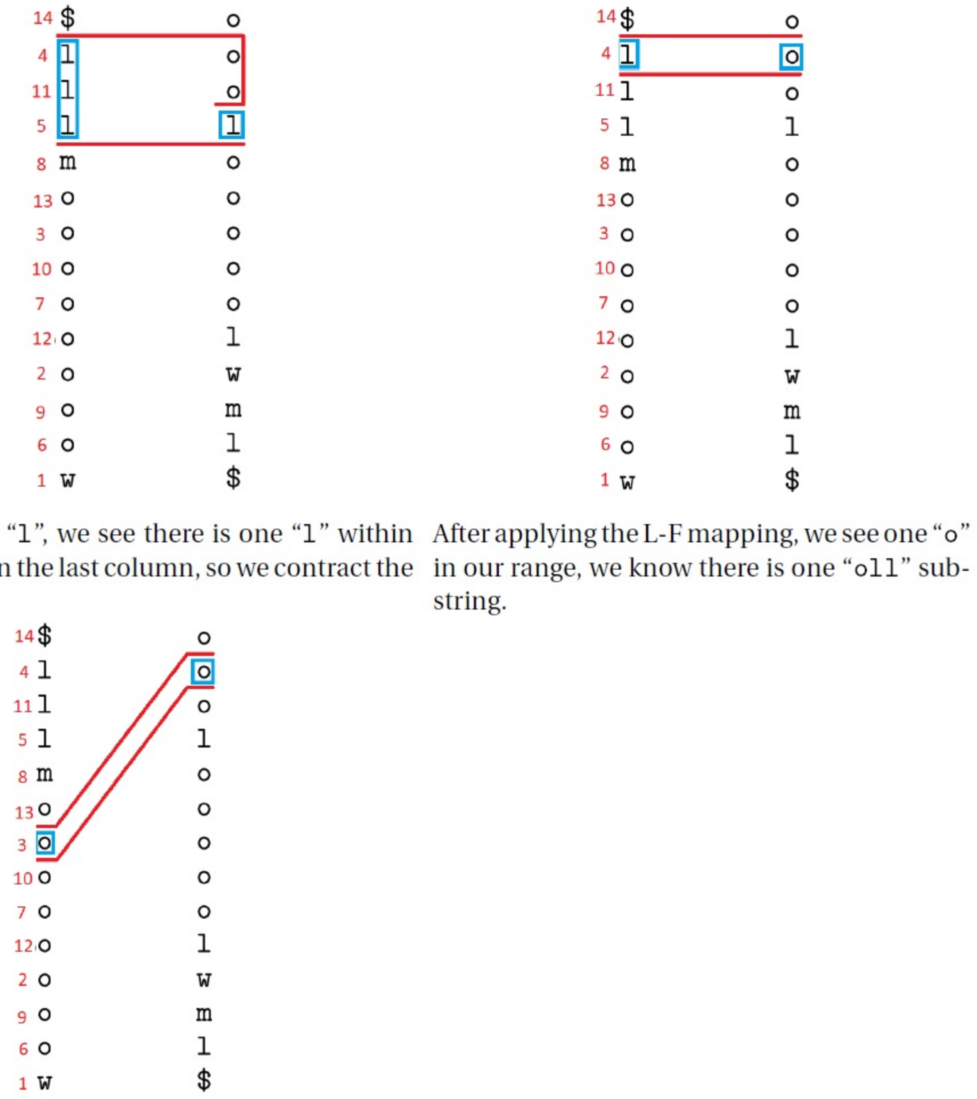


In the last column there is only one “o” in our range, so there is one instance of the substring `olo`



We use L-F mapping to get the location of “o” in the first column and determine its location using suffix ID, e.g., the location of `olo` in the string is 10.

(b) **Searching for** `oll`

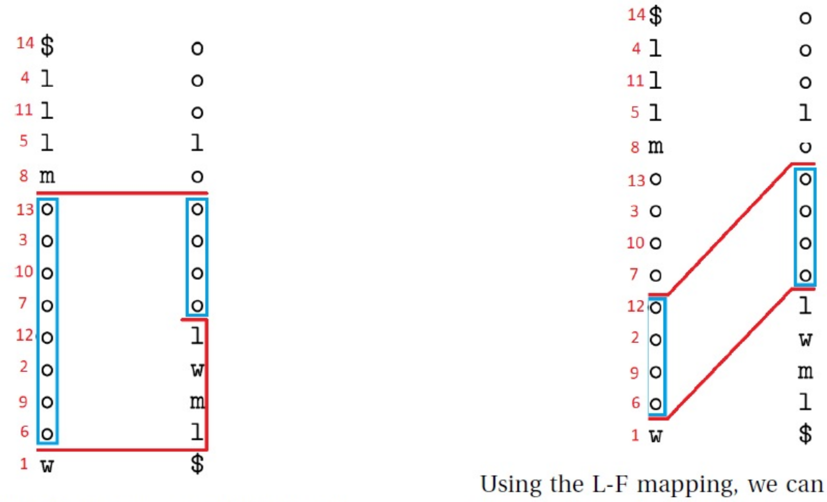


Starting at “l”, we see there is one “l” within the range in the last column, so we contract the range

After applying the L-F mapping, we see one “o” in our range, we know there is one “oll” substring.

Again using the L-F mapping, we can now look up the location of our substring which is 3.

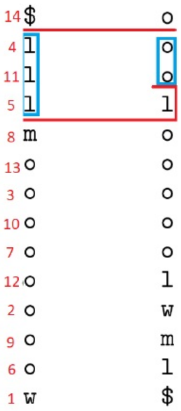
(c) **Searching for** oo



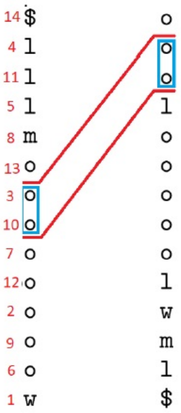
Starting with “o”, we can see 4 “o”s in the range, so there are 4 “oo” substrings.

Using the L-F mapping, we can now look up the locations of our substrings in the suffix array (12, 2, 9 and 6)

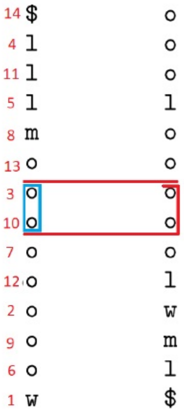
(c) **Searching for** wol



Starting at "l", we see there are 2 "o"s within the range in the last column, so we contract the range



Apply the L-F mapping. We are tracking ol substrings.



We observe that the range contracts to 0 (there are no "l"s in the range) so there are no wol substrings.

Problem 8. 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:

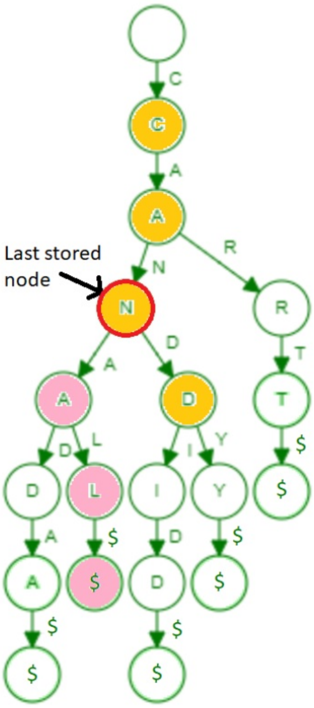
All words with a prefix p are leaves in the subtree rooted at the node containing the last character in p. So we have to count the leaves in the subtree of that node. We could traverse the trie to get to the last node of p in O(m) and then do a complete traversal of the subtree, but this would take much 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 9. Describe how to implement predecessor queries in a trie. The predecessor of a string S is the lexicographically greatest string in the trie that is lexicographically less than or equal to S. Your data structure should perform predecessor queries in O(n+m) time, where m is the length of the query string, and n is the 1 length of the answer string (the predecessor). It is possible that a string has no predecessor (if it is less than all of the strings in the trie), in which case you should return null.

SOLUTION:

Lets assume that the contents of our trie are terminated with a $\$ \in \Sigma$. To find the predecessor of a string in a trie, 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

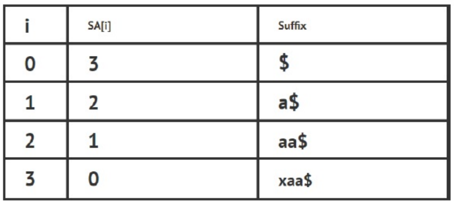
query. Otherwise we look at the node we were keeping track of, and if it is null, we return null since the word must have no predecessor, as none of its prefixes could be extended with a lexicographically lesser character. If such a node does exist, go back to that node, go along the edge with the greatest 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 candwe 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 ≠ 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
```

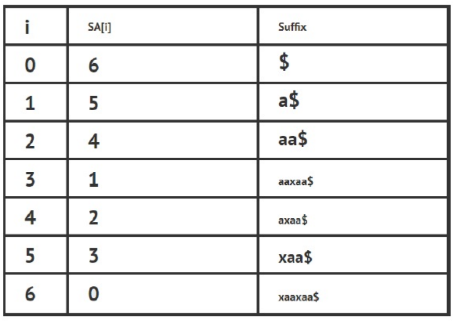
Problem 10. The minimum lexicographical rotation problem is the problem of finding, for a given string, its cyclic rotation that is lexicographically least. For example, given the string banana, its cyclic rotations are banana, ananab, nanaba, anaban, naban, abanan. The lexicographically (alphabetically) least one is abanan. Describe how to solve the minimum lexicographical rotation problem using a suffix array.
SOLUTION:
A cyclic rotation of a string is a suffix of that string followed by a prefix of that string. Since a suffix array stores the suffixes in sorted order, the lexicographically least suffix will be the first element of the array (ignoring the "\$" which will always be first.) However, we can run into problems such as the string `XAA$`:



i	SA[i]	Suffix
0	3	$
1	2	a$
2	1	aa$
3	0	xaa$

Source: `https://visualgo.net/bn/suffixarray`

Here, the suffix "A\$" would be followed by XA in its cyclic rotation, giving AXA, wheras the suffix AA\$ would be followed by X, giving AAX, which is lexicographically earlier than AXA. So it is not enough to just construct the suffix array, we need to account for the elements of the corresponding prefix.
We do this by first appending a copy of the string in question to itself, and then constructing the suffix array. This gaurantees that the corresponding prefix for each suffix is taken into account in the order of the suffix array. Note that we should only consider suffixes which are strictly longer than the original string, since only these suffixes contain characters from the appended copy.



i	SA[i]	Suffix
0	6	$
1	5	a$
2	4	aa$
3	1	aaxaa$
4	2	axaa$
5	3	xaa$
6	0	xaaxaa$

Source: `https://visualgo.net/bn/suffixarray`

Now AAXAA\$X is the first such string in the suffix array, and hence AAX is the lexicographically least rotation.

Supplementary Problems

Problem 11. 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

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

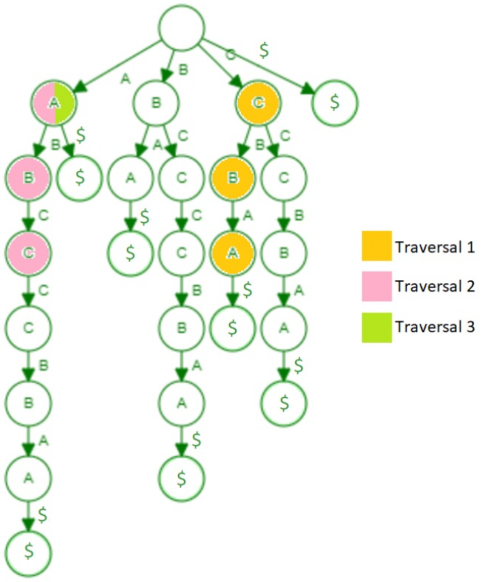
where $|p|$ denotes the length of the prefix p. Describe an algorithm that solves this problem in O(T) time, where T is the total length of the strings s_1, ..., s_n . Assume that we have three strings baby, bank, bit with weights 10, 20 and 40, respectively. The score of prefix ba is $2 \times 10 +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

Problem 12. 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.

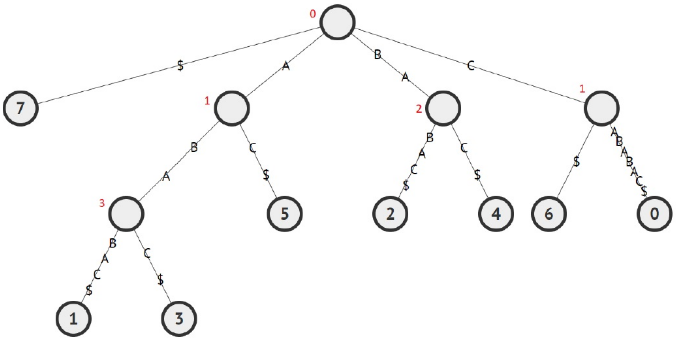


Source: `https://www.cs.usfca.edu/~galles/visualization/Trie.html`

Problem 13. 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:

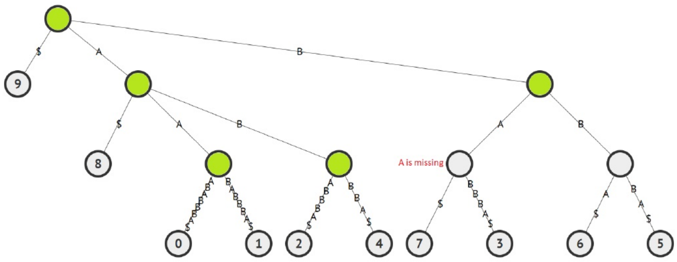
Consider a suffix trie of S. A unique substring corresponds to a node that has only one leaf below it. We want to find the highest such node, since this will be the shortest such substring. Define a candidate node as a node which has a leaf as a child, whose edge label contains more than just \$. In 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. Lets 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 14. 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 15. A string S of length n is called k-periodic if S consists of $n = k$ repeats of the string $S[1..k]$. For example, the string abababab is two-periodic since it is made up of four copies of ab. Of course, all strings are n-periodic (they are made of one copy of themselves!) The period of a string is the minimum value of k such that it is k-periodic. Describe an efficient algorithm for determining the period of a string using a suffix array.
SOLUTION:
To solve this problem, lets relate the period of the string to the concept of cyclic rotations. If a string is k-periodic, then this means that it is equal to its kth cyclic rotation. For example, if we take the string abababab, then its cyclic rotations are babababa, then abababab, which is the original string again. We can therefore use an idea similar to Problem 10. Lets append a copy of the string S to itself to obtain the string SS. Now we observe that if a string is k-periodic, then the string S will appear in position k +1 in SS. For example, take abababab, and append it to itself to obtain abababababababab. Then we notice that the substring at position 3 is precisely the original string. Thus, after suffix array on the string SS is constructed, we can do a substring search for S in the suffix array to find all suffixes that contain the substring S. Among these suffixes, the suffix with the smallest ID (ignoring the first suffix) can be used to determine the value of k. For example, we search abababab in the suffix array of abababababababab which matches with suffixes 1, 3, 5, 7, 9. The smallest suffix (except first suffix) that matches is 3. Therefore, the value of k is 2. Building the suffix array takes O($nlog^2n$) time, and substring search takes O($nlog(n)$).

Problem 16. Write psuedo code for an algorithm that converts the suffix tree of a given string into the suffix array in O(n) time.
SOLUTION:
A suffix array is just the suffixes in sorted order. Since the suffix tree contains all of the suffixes as leaves, we just need to traverse it in lexicographic order and note down the order in which we find the leaves, and this will give us the suffix array. Whenever we reach a leaf, we can figure out which suffix it belongs to by looking at its length. A suffix of length k must be the suffix from position $n - k + 1$

```
function TO_SUFFIX_ARRAY()
    SA[1..$n$] = null
    counter = 1    // Tracks which suffix we are up to
    CONVERT(root, 0)
    return SA[1..$n$]
end function
```

```
function CONVERT(node, length)
    if node is a leaf then
        SA[counter] = $n$ - length + 1
        counter = counter + 1
    else
        for each child of node, in lexicographic order do
            CONVERT(child, length + child.num_chars)
        end for
    end if
end function
```

Problem 17. (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 ai 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 18. (Advanced) Recall the pattern-matching algorithm that uses the Burrows-Wheeler transform of the text string. One downside of this algorithm is the large memory requirement if we decide to store the occurrences Occ(c, i) explicitly for every position i and every character c . In this problem we will explore some space-time tradeoffs using milestones. Suppose that instead of storing Occ(c, i) for all values of i , we decide to store it for every kth position only, i.e. we store Occ(c,0), Occ(c,k), Occ(c,2k), Occ(c,3k), ... for all characters c.

(a) What is the space complexity of storing the preprocessed statistics in this case?

(b) What is the time complexity of performing the preprocessing? To compute Occ(c,i), we will take the value of Occ(c,j) where j is the nearest multiple of k up to i and then manually count the rest of the occurrences in $S[j + 1..i]$

(c) What is the time complexity of performing a query for a pattern of lengthm?

(d) Describe how bitwise operations can be exploited to reduce the space complexity of the preprocessed statistics to O($\frac{\sum |n|}{w}$), where w is the number of bits in a machine word, while retaining the ability to perform pattern searches in O(m).

SOLUTION:

(a) Since we will store the value of Occ(c ,k) for all multiples of k up to n, the amount of space required will be

$$O(\frac{|\Sigma| n}{k})$$

(b) We perform a linear sweep over the text string, maintaining the counts for all characters. When we hit a multiple of k, we save the current state of the counts. In total, this takes

$$O(|\Sigma| + n + \frac{|\Sigma| n}{k})$$

time. (c) For a given position, we can compute the nearest multiple of k in constant time, hence the only work required is to count the number of occurrences between the milestone and the query position. In the worst case, we may have to count k-1 elements, hence this takes O(k) time per character. Therefore to search a pattern of length m takes O(mk) time in the worst case.
(d) In order to achieve O(m) time pattern searches, we need to be able to compute Occ(c,i) in constant time. To do so, we make milestones of size $k = w$, giving us the value of Occ at every wth position. This gives us the space complexity desired. Our goal is to speed up the computation of counting the remaining occurrences between the milestones.
For each milestone position j , for each character $c \in A$, lets store a w-bit integer whose b th bit is 1 if the character at position $j + b$ is c, or a 0 otherwise. To compute the number of occurences of c in the interval [j +1..i] then corresponds to counting the number of 1 bits in the first $i - j$ positions of this integer. This can be achieved by masking off the bottom bits by taking its bitwise AND with $2i - j - 1$ and then using a pop count operation, which counts the number of 1 bits in an integer. Assuming that our machine model supports popcount in O(1) time, we can now compute Occ(c,i) in O(1) time. Since we stored just one integer at each milestone for each character, the amount of extra space used is O($\frac{|\Sigma|n}{w}$), in addition to the O($\frac{|\Sigma|n}{w}$) space used for the milestones, making the total space complexity as required.

Problem 19. (Advanced) A given suffix array could have come from many strings. For example, the suffix array 7, 6, 4, 2, 1, 5, 3 could have come from banana\$, or from dbxbxa\$, or many other possible strings.

(a) Devise an algorithm that given a suffix array, determines any string that would produce that suffix array. You can assume that you have as large of an alphabet as you need.

(b) Devise an algorithm that given a suffix array, determines the lexicographically least string that would produce that suffix array. You can assume that you have as large of an alphabet as required. Your algorithm should run in O(n) time.

SOLUTION:
(a) To construct any string that has a given suffix array is quite simple. The first suffix will be the empty suffix sowe always have \$ as the final character. From there, the next suffix should begin with a, the next with b, the next with c, and so on. This means that we place a at position $SA[2]$, b at position $SA[3]$, c at position $SA[4]$ and so on. For example, given the suffix array above, this algorithm would produce the string dcfbea\$.
(b) To produce the lexicographically earliest string is a bit trickier. What we would like to do is reuse the same character as many times as possible, starting several suffixes with a, then several with b, and so on if we can. But we need to be careful to ensure that we keep the order of the suffixes correct. Suppose we place some letter c at the position $SA[i]$. How do we determine whether it is safe to also place c at position $SA[i + 1]$? Well, the suffix at position $SA[i + 1]$ must be greater since it comes later in the suffix array, so we need to ensure that $c+S[(SA[i]+1)..n] < c+S[(SA[i+1]+1)..n]$, which means that we need $S[(SA[i]+1)..n] < S[(SA[i+1]+1)..n]$. In other words, we just need to compare the two suffixes at positions $SA[i]+1$ and $SA[i+1]+1$. To do so quickly, lets compute the ranks of the suffixes. These are easy to compute since we have the full suffix array. Deciding whether we can place character c then comes down to checking whether $rank[SA[i] + 1] < rank[SA[i + 1] + 1]$, which can be done in constant time. If it is not safe to reuse character c , we move on to the next character in the alphabet. In total, this algorithm takes O(n) time since computing the ranks can be done in O(n), and each suffix comparison takes O(1) with the rank array. This algorithm will produce bacaca\$ as the lexicographically least string with the same suffix array as banana\$.