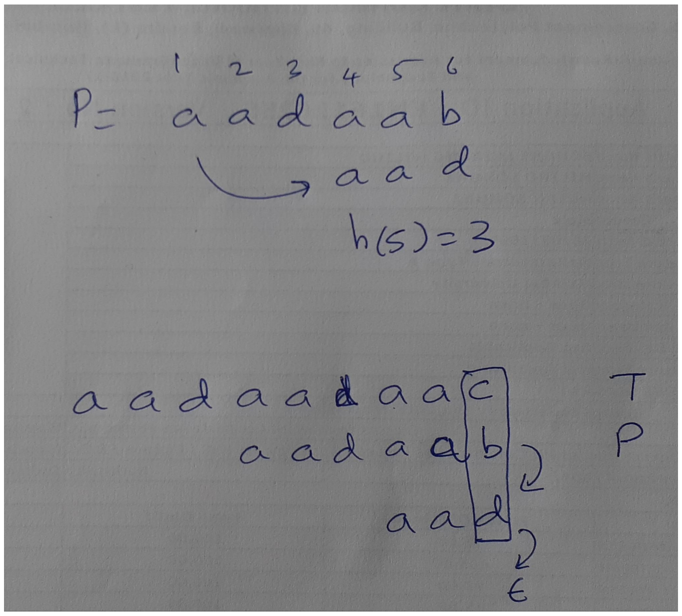
Tutorial 6. Solutions.

1. **This is concerning Prof. Garg's argument that the total number of unsuccessful comparisons = shifts in KMP algorithm is no more than |T|. Is this strictly true?**



Let T and P be as shown. We see that for i=5, h(i)=3, since aad satisfies all the conditions. However, when locating it in a text, consider the situation when P(6)!=T(9)!=P(3). All the three values are distinct. When this happens, the first shift of 3 will not work, and h(2) will need to be called. Thus, for a location in the main text, the number of unsuccessful comparisons may be the size of the alphabet.

2. **Please review the KMP algorithm to see how it detects two overlapping occurrences of the pattern.**

The best way to understand this is to put a $ at the end of the pattern. Whenever the $ is encountered, register a success and execute the shift as well.

T	a	b	b	a	b	a	b	a	a
P				a	b	a	$		
shift						a	b	a	$
again								a	b

3. **For a pattern P[1...n], h(i) is defined to be the smallest k>0 such that P[1]=P[k+1],..., P[i-k]=P[i], but P[i-k+1]!=P[i+1]. If there is no such match, define h(i) to be i. Fill in the table below.**

i	1	2	3	4	5	6	7	8	9
P	b	a	b	b	a	a	b	b	a
h(i)	1	2	2	4	3	6	6	8	7
i=1	b								
h(i)=1	No proper suffix that is also a prefix								
i=2	b	a							
h(i)=2	No proper suffix that is also a prefix								
i=3	b	a	b	**b**					
h(i)=2			b	**a**					
i=4	b	a	b	b	**a**				
h(i)=4				b	**a**				
	No matching prefix-suffix pair satisfying P[i-k+1]=P[i+1]								
i=5	b	a	b	b	a	**a**			
h(i)=3				b	a	**b**			
i=6	b	a	b	b	a	a			
h(i)=6	No proper suffix that is also a prefix								
i=7	b	a	b	b	a	a	b	**b**	
h(i)=6							b	**a**	
i=8	b	a	b	b	a	a	b	b	**a**
h(i)=8								b	**a**
	No matching prefix-suffix pair satisfying P[i-k+1]=P[i+1]								
i=9	b	a	b	b	a	a	b	b	a
h(i)=7								b	a

4. **Suppose that there is a letter z in P of length n such that it occurs in only one place, say k, which is given in advance. Can you optimize on the computation of h?**

   The computation of the 'h[]' array is done using the lps array.
   The code for computing lps(longest proper prefix which is also suffix.) array.

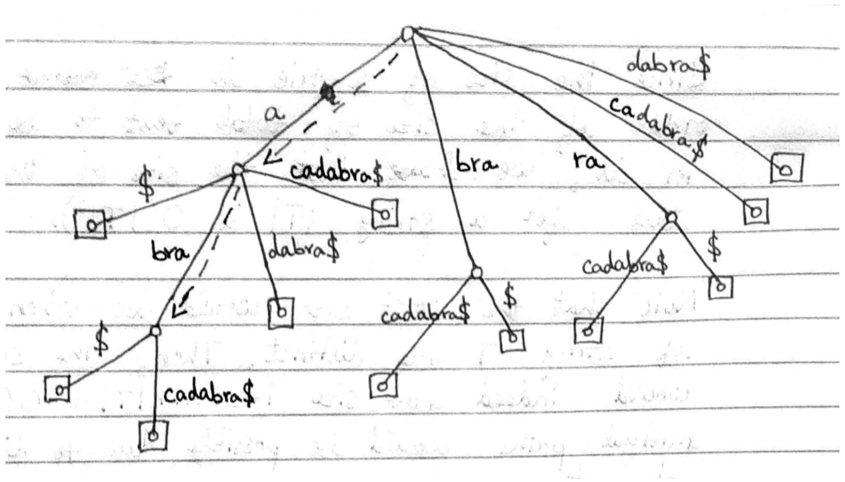
```
void compute_lps(char* pat, int M, int* lps) {

    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;                    Case1
            i++;
        }
        else // (pat[i] != pat[len])         Case2
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];
                                             Case2_1
                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {                                Case2_2
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

   If we know the element at index k(which is z here) appears only once, we already know lps[k]=0.

When computing the lps array for our array P containing only one z at index k.

The algorithm proceeds with iterating on i until it reaches M-1.
When i=k,(that means lps array is already being computed for 0 to i-1).
Obviously **Case1** will not be true as 'z' appears only once and hence it can't be equal to any other element of array P at any other index than 'k'. So, the algorithm goes to Case2. And it keeps recursing onto **Case2_1** until the len variable becomes 0 wherein it lands at **Case2_2**. Instead of doing this normal lps, we can construct a modified lps, where while loop first computes lps of i=2...k-1 and then as z(at index k) appears only once lps[k]=0 directly and then again a while loop for i=k+1..M-1.
Thus there are a reduced number of operations in this modified_lps, for index=k. And hence an optimised version of h. As 'h' basically constructs lps array first and uses this array to construct itself.

5. **Compute the Suffix Trie for *abracadabra$*. Compress degree 1 nodes. Use substrings as edge labels. Put a square around nodes where a word ends. Use it to locate the occurrences of *abr.***



The compressed suffix tree corresponding to the string "abracadabra$" is shown above. Note that there are 11 square nodes, corresponding to 11 suffixes of the string. (Although $ by itself would also be a suffix, we treat it as a terminating character here.)

Now, we use the suffix tree to locate occurrences of "abr" in the string. Suppose the pattern "abr" occurs at some position k within the string. Then, the substring of "abracadabra$" from position k to the end must be a suffix present in the above suffix tree. Thus, to locate all occurrences of "abr" in the string, we only need to find all suffixes that begin with "abr". Starting at the root, we traverse along the unique path that corresponds to "abr"; this path is shown in the figure by the broken arrows. From that path, the only two leaves that can be reached are those corresponding to suffixes "abra$" and "abracadabra$".

Hence, there are two occurrences of “abr” in the string “abracadabra$”, which, when extended to the end, form the two suffixes we found above. From these two suffixes, we see that “abr” occurs at positions 5 and 12 from the right end, and thus, positions 12-5+1=8 and 12-12+1=1 from the left end.

6. **Review the argument that for a given text T, consisting of k words, the ordinary Trie occupies space which is a constant multiple of |T|. How is it that the suffix tree for a text T is of size O(|T|^2)?**

   A suffix tree, for a given string T, is essentially a trie that stores all possible suffixes of the string T. Any string T would have |T| number of suffixes, corresponding to the last 1,2,...,|T| characters in the string. Thus, the total size of the text to be stored in the corresponding trie (i.e, the suffix tree) is

$$1 + 2 + ... + |T| = O(|T|^2)$$

   Since the size of the trie is of the order of the size of total text to be stored in it, we have that the size of the suffix tree for a string T is O(|T|^2).