Please be concise: each of the problems should not take more than 1 page. I prefer shorter answers that are right to the point.

**Problem 1**

Read the posted explanation by Gusfield on the LCP array construction. When it was argued that \( LCP(Suff_{k+1}; Suff_{Pred(k+1)}) \geq h - 1 \), it says “if the path to leaf \( Pred(k+1) \) does not extend below \( v \), that will be impossible”. Now, give a more expanded argument why this will be impossible.

**Problem 2**

In class, we briefly introduced a data compression method. We defined \( Prior_i \) as the longest prefix of \( S[i..n] \) that also occurs in \( S[1..i-1] \) for a given string \( S \). We define its length of this prefix as \( l_i \) and its starting position as \( s_i \). We also mentioned the way of data compression as follows: scan \( S \) from left to right; for each position \( i \), if \( l_i > 0 \), then we output \( [l_i, s_i] \); Otherwise, we output \( S[i] \).

Now, show me a suffix tree-based compression algorithm that runs in \( O(m) \) time, where \( m \) is the length of the string.

**Problem 3**

In class, we described the problem of searching pattern \( P \) in a text using a suffix array. That method needs the LCE value. Longest common extension (LCE) for two positions \( i \) and \( j \) in a string \( S \) (whose length is \( n \)) is the length of the longest common prefix of suffix \( i \) and suffix \( j \) of \( S \). For example, for \( S=abaabac \), \( LCE(1,4)=3 \), \( LCE(3,4)=1 \).

(a) Now, you are given the suffix array \( POS \) and its LCP array of \( S \). Prove that \( LCE(i,j) \) is equal to the smallest LCP value between suffix \( i \) and suffix \( j \) in the LCP array. That is, suppose \( POS[a]=i \) and \( POS[b]=j \) (where \( a < b \)), then \( LCE(i,j) = \min_{k=a+1}^{b} LCP[k] \).

(b) How many LCE values are needed in the binary search? How long does it take to compute these values using the lemma in Part (a)? Your method should run in time \( O(n) \), where \( n \) is the length of the text.

**Problem 4**

Recall that MUM (maximal unique match) for two strings \( S_1 \) and \( S_2 \) is a substring that appears in both \( S_1 \) and \( S_2 \) exactly once, and are longest (i.e. can not be extended in both directions).

Now find a linear time algorithm for finding all the MUMs in two strings \( T \) and \( T' \) using a suffix array \( POS \) and LCP array. You can not use suffix tree in this case (that is, do not construct a suffix tree from suffix array).