P1:

a)

bbacbbcb

	0	1	2	3	4	5	6	7
a	0	0	3	0	0	0	3	0
b	1	2	2	1	5	6	2	7/match
c	0	0	0	4	0	0	7	0

b)

bbacbbcb

	0	1	2	3	4	5	6	7
√	1	2	3	4	5	6	7	match
×	0	0	1	0	0	0	2	0

c)

use KMP-algorithm, search the pattern in the string, change the index pointer in pattern backward according to the failure array when it does not match, and move index pointer forward when there is a match. If index points to the last of the pattern, the whole pattern matches. If pointer to text reaches the end, there is no match of pattern.

```
match(T, P), to return the index of the the first match
T: String of length n, text; P: String of length m, pattern
F: KMP Failure Array
      i ← 0;
      j ← 0;
      while i < n do{
            if T[i] = P[j] then{
                  if j = m – 1 then {
                        //there is a match
                        return i – j;
                  } else {
                        i ← i + 1;
                        j ← j + 1;
                  }
            } else {
                  if j > 0 then {
                        j ← F[j – 1];
                  } else {
                        i ← i + 1;
                  }
            }
      }
//no match is found
```

```
return -1;
```

P2:

This can be down by using KMP-automaton method, searching pattern w in string that is x repeated once. For example, if inputs are “alloy” and “loyal”, the algorithm should return true if we can find “alloy” as a substring of “loy**alloy**al”.

```
bool shift(string w, string x){
      //KMP(T,P) is the same algorithm as discussed in class.
      if (KMP(w, x+x) == false){
            return false;
      }
      return true;
}
```

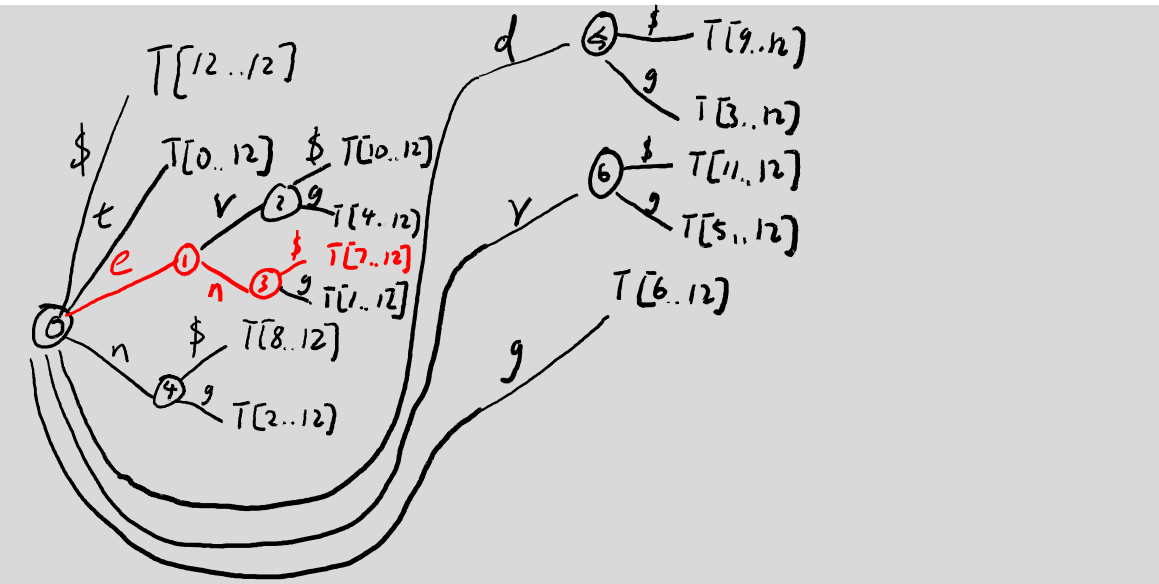
If we can find in the repeated string, it means the tails of x is the start of w, and the beginning of x is the tail of w. Since w and x are of the same length, length of matched head plus tail is the length of x. In the example at the beginning, “al” is the tail of x and head of w, “loy” is the tail of w and head of x. We can shift “loy” part, and we get x from w.

The running time of this algorithm is $\Theta(n + 2n) = \theta(n)$.

P3:

a)

tendergender



b)

the repeated pattern is "$ender$" at index 1 and index 7 with length 5.

d)

First build a suffix tree, which takes O(n) as discussed in class, then find the deepest leaf (“deepest” means in “T[n .. max]”, max – n is maximum.) which is also in a branch. Then its adjacent leaf, which shares the same parent and ends with a “$” sign, is the maximum repeated pattern.

Since the branch indicates difference, as it branches, it guarantees at least two substring share the same pattern. Then the “deepest” leaf in that branch is guaranteed to be repeated by another substring.

P4

a)

Assume towards contradiction that for every compression algorithm A and $n \in \mathbb{N}$ there is an input $w \in \Sigma^n$ for which $|A(w)| < |w|$. Then we can compress $A(w)$ and get $A(A(w))$ with $\left|A\left(A(w)\right)\right| < |A(w)|$. We should be able to infinitely repeat this process and get the compress length real small. However, since the length is integer, the length will keep decreasing and eventually become negative, which creates a contradiction. Hence the original statement is true.

b)

total number of words:

$$|\Sigma^{\leq n}| = |\Sigma|P0 + |\Sigma|P1 + |\Sigma|P2 + |\Sigma|P3 + \cdots + |\Sigma|P|\Sigma| + |\Sigma|P|\Sigma| \times (|\Sigma|) + |\Sigma|P|\Sigma| \times (|\Sigma|)^2 + \cdots + |\Sigma|P|\Sigma| \times (|\Sigma|)^{n-|\Sigma|}$$

we need a different representation for each word. Assume towards contradiction that more than half of the words can be compressed to smaller words, assume $w \in \Sigma^{\leq n}$, let $A^*(w)$ represents compressing w * times, then $\{A(w): w \in \Sigma^{\leq n}, |A(w)| < |w|\} \subseteq \Sigma^{\leq n-1}$, However there are not many words

P5:

a)

T=bananablues

letter	a	b	e	l	n	s	u
frequency	3	2	1	1	2	1	1

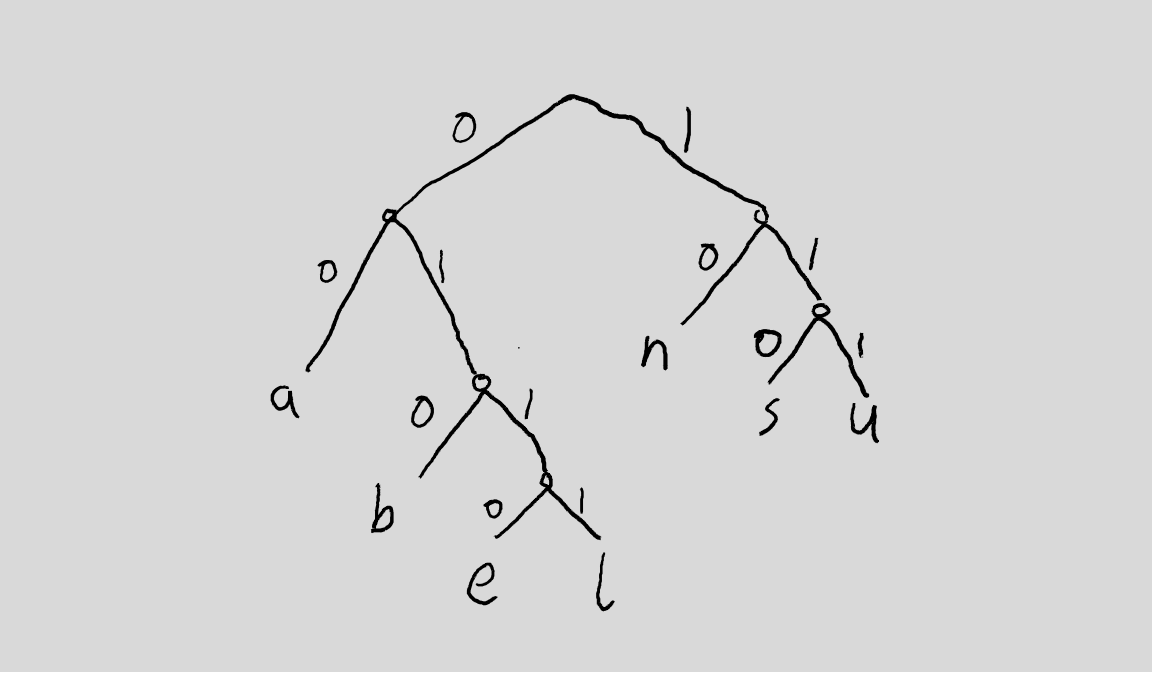


Table:

letter	a	b	e	l	n	s	u
code	00	010	0110	0111	10	110	111

Code: 010 00 10 00 10 00 010 0111 111 0110 110

0100010001000010011111110110110

b)

1000100010010110011110110111111110010

Decodes to:

10 00 10 00 10 010 110 0111 10 110 111 111 10 010

nananbslnsuunb

c)

letter	a	b	e	l	n	s	u
code	00	010	0110	0111	10	110	111

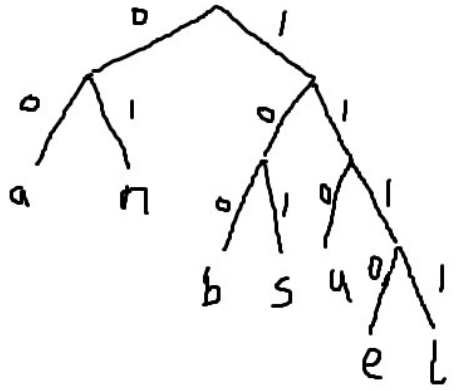
Sort:

letter	a	n	b	s	u	e	l
frequency	2	2	3	3	3	4	4
canonicalHuffmanCode	00	01	100	101	110	1110	1111

New code for “bananablues”:

100 00 01 00 01 00 100 1111 110 1110 101

100000100010010011111101110101



d)

We encode “bananablues” by Huffman’s algorithm, and we will get tri No.1. Then we encode “zananazlues”, replacing “b” with “z”, and use Huffman’s algorithm, and we will get tri No.2. Tri No.1 and No.2 have different shape; however, canonical Huffman algorithm on Tri No.1 and No.2 produce a tri of the same shape. Therefore the result of Huffman’s algorithm depends on the symbol used, violating the canonical Huffman algorithm’s property that it does not need the information of the encoded word.