HW 3

Kelly Joyce, Benjamin Haedt

October 2023

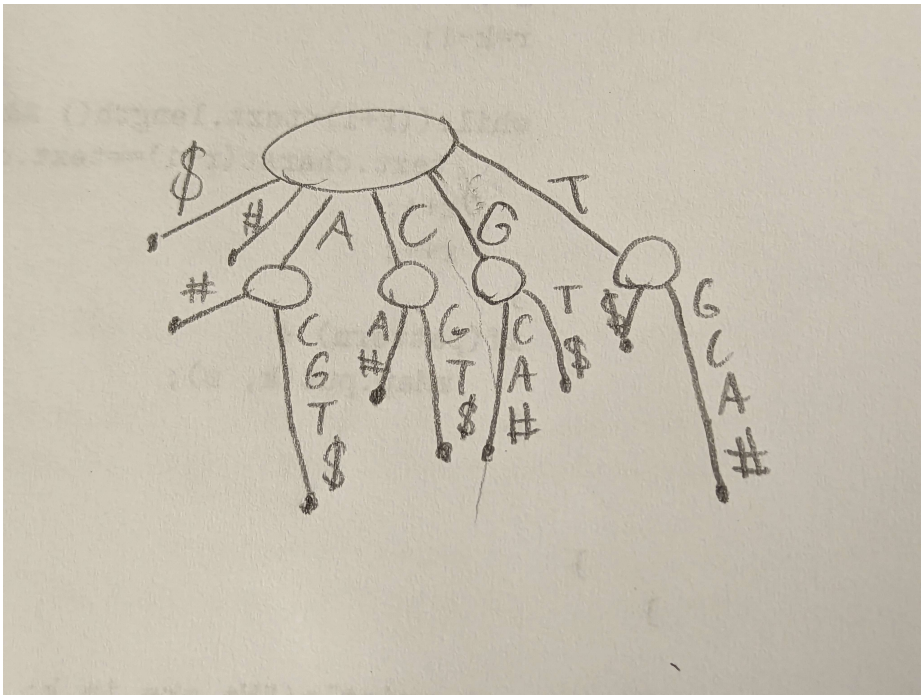
1 Problem 1

1. "gat" is the longest common substring in S1, S2, and S3.

2. Looking at S_1 = acgatca\$, S_2 = gattact#, S_3 = aagatgt, we want to find the longest common substring of S1, S2, and S3. To find the solution, construct a generalized suffix tree that incorporates all 3 sequences. Iterate through each internal node and find all of the internal nodes' depths that ends with at least one \$, #, and , showing that all of the sequences share that internal node (LCA). Then, pick the internal node that has the largest depth. The longest common substring is the substring from the root to the internal node found. With constructing the suffix tree in linear time, the overall process of constructing the tree and finding the longest common substring will take $O(|S_1| + |S_2| + |S_3|)$ time.

3. If you were to have a data set that contains all internal nodes with largest depth with at least one \$, #, and , rewriting it as you iterate through all of the internal nodes (as in, if it's an internal node for all 3 sequences and it's depth is greater than the current max's depth, rewrite it. If it's depth is equal, add the node to the data set). This would take $O(|S_1| + |S_2| + |S_3|)$ time.

2 Problem 2



3 Problem 3

To find all instances of pattern P within the DNA sequence S while allowing a Hamming distance of at most 1, do the following:
Iterate through sequence S, examining segments of size —P—.
For each segment, compute the Hamming distance between the segment and pattern P.
Whenever the Hamming distance is less than or equal to 1, make a note of the position of the segment as a potential match.
Move forward one position and repeat steps 2-3 until you have covered all possible segments in the sequence.
The time complexity of this algorithm remains $O(|P|^2 + |S|)$. This is because, for each segment, you perform a Hamming distance calculation in $O(|P|^2)$ time, and you traverse the entire sequence S in O(—S—) time.

4 Problem 4

ACGTACGT$

a.

i	S[i]	Suffix
1	9	$
2	5	ACGT$
3	1	ACGTACGT$
4	6	CGT$
5	2	CGTACGT$
6	7	GT$
7	3	GTACGT$
8	8	T$
9	4	TACGT$

b.

LCP(k,k+1)	k	Answer
	1	null
	2	null
	3	null
	4	null
	5	null
	6	null
	7	null
	8	null

LCP(k,k+4)	k	Answer
	1	ACGT
	2	CGT
	3	GT
	4	T

5 Problem 5

```
Sequence:  ACGTACGT$
SA[ 1 ]= 9  Suffix:  $
SA[ 2 ]= 5  Suffix:  ACGT$
SA[ 3 ]= 1  Suffix:  ACGTACGT$
SA[ 4 ]= 6  Suffix:  CGT$
SA[ 5 ]= 2  Suffix:  CGTACGT$
SA[ 6 ]= 7  Suffix:  GT$
SA[ 7 ]= 3  Suffix:  GTACGT$
SA[ 8 ]= 8  Suffix:  T$
SA[ 9 ]= 4  Suffix:  TACGT$
```

```
Sequence:  ATTACCG$
SA[ 1 ]= 8  Suffix:  $
SA[ 2 ]= 4  Suffix:  ACCG$
SA[ 3 ]= 1  Suffix:  ATTACCG$
SA[ 4 ]= 5  Suffix:  CCG$
SA[ 5 ]= 6  Suffix:  CG$
SA[ 6 ]= 7  Suffix:  G$
SA[ 7 ]= 3  Suffix:  TACCG$
SA[ 8 ]= 2  Suffix:  TTACCG$
```

```
def buildSuffixArray(sequence):
    global n
    n = len(sequence)
    suffixes = [(sequence[i:], i+1) for i in range(n)]
    suffixes.sort()
    suffixArray = [suffix[1] for suffix in suffixes]
    return suffixArray

def printArray(suffixArray,sequence):
    j=1
    for i in suffixArray:
        print("SA[",j,"]=",i," Suffix: ",sequence[i-1::])
        j=j+1

with open('input.fasta', 'r') as file:
    header = file.readline()
    sequence = file.readline().strip() + '$'
    print("Sequence: ",sequence)


suffixArray = buildSuffixArray(sequence)
printArray(suffixArray,sequence)
```

6 Extra Credit

Input Sequence is set to ACGTACGT$.

```
Enter pattern:
AC
Pattern found at SA[2]. String: ACGTACGT$
Pattern found at SA[1]. String: ACGT$
Enter pattern:
CGT
Pattern found at SA[4]. String: CGTACGT$
Pattern found at SA[3]. String: CGT$
```

```
Enter pattern:
AG
Pattern does not occur in this sequence.
```

```
import java.util.Scanner;

public class Main {
   public static int[] SA;
   public static String text; //from fasta file


   public static void main(String[] args) {
      // TODO Auto-generated method stub

      text="ACGTACGT$";
      SA=new int[text.length()];
      SA[0]=9; SA[1]=5; SA[2]=1; SA[3]=6; SA[4]=2; SA[5]=7; SA[6]=3;
          SA[7]=8; SA[8]=4;

      Scanner s=new Scanner(System.in);
      System.out.println("Enter pattern: ");
      String p=s.nextLine().toUpperCase();
      s.close();
      int L=0;
      int R=SA.length-1;
      binary_search(p,L,R);
```

```
}

public static void binary_search(String pattern, int L, int R)
    {//pattern needs $ at the end
   int l=lcp(pattern,getString(SA[L]),0);
   int r=lcp(pattern,getString(SA[R]),0);
   boolean found=false;

   while(!found) {
      int M=(L+R)/2;
      int mlr=Math.min(l, r);
      int m=lcp(pattern,getString(SA[M]),mlr);
      if(m==pattern.length()) {
         found=true;
         System.out.println("Pattern found at SA["+M+"]. String:
             "+getString(SA[M]));

         //below checks for other patterns as well by checking those
             next to M
         int N=M-1;
         if(N>0) {
            int n=lcp(pattern,getString(SA[N]),0);
            while(N>0 && n==pattern.length()) {
               System.out.println("Pattern found at SA["+N+"]. String:
                   "+getString(SA[N]));
               N--;
               n=lcp(pattern,getString(SA[N]),0);
            }
         }
         int B=M+1;
         if(B<pattern.length()) {
            int b=lcp(pattern,getString(SA[B]),0);
            while(B<pattern.length() && b==pattern.length()) {
               System.out.println("Pattern found at SA["+B+"]. String:
                   "+getString(SA[B]));
               B++;
               b=lcp(pattern,getString(SA[B]),0);
            }
         }

      } else if(getString(SA[M]).compareTo(pattern)>0) { //pattern in
          top half
         if(L>=R) { //If this is not here, the else if statement will
             sometimes be looped forever if no match
            System.out.println("Pattern does not occur in this
                sequence.");
            found=true;
         }
         R=M;
         r=m;
```

```
        } else { //pattern in bottom half
           if(L>=R) {
              System.out.println("Pattern does not occur in this
                  sequence.");
              found=true;
           }
           else {
              if(L+1 != R) {
                 L=M;
                 l=m;
              } else {
                 L=R;
                 l=r;
              }
           }

     }

  }

  private static String getString(int i) {
     // TODO Auto-generated method stub
     String current=text.substring(i-1);
     return current;
  }

  private static int lcp(String pattern, String suffix, int start) {
     // TODO Auto-generated method stub
     int i=start;
     while(pattern.length()>i && suffix.length()>i &&
         pattern.charAt(i)==suffix.charAt(i)) {
        i++;
     }
     return i;
  }

}
```