COMP3121 - Assignment 3

Mark Pollock - z3372890 - mpol525

Due: 21st May 2015

1 Question 1

I solved this question through the following logic:

1. Consider strings $A = a_0a_1a_2a_3...a_{n-1}$ and $B = b_0b_1b_2b_3...b_{m-1}$.
2. For each letter l in B: If l is equal to the first letter in A, add the number of occurences found by repeating this procedure with strings C and D where $C = a_1a_2a_3...a_{n-1}$ and $D = l...b_{m-1}$

Note: The above is a recursive algorithm and the base case is when a is of length 1, in which we just add 1 to the number of occurences (when l == a0) instead of repeating the procedure.

The following c++ code also defines the algorithm:

```
int getOccurenceCount(string a, string b){
   char bchar = b[0];
   int occurence = 0;
   for(int i = 0; i < b.size(); ++i){
      if(a[0] == b[i]){
         if(a.size() == 1)
            occurence++;
         else
            occurence += getOccurenceCount(a.substr(1, a.size() - 1),
                                           b.substr(i + 1, b.size() - 1));
      }
   }
   return occurence;
}


int main(int argc, char *argv[]){
   string a;
   string b;
   a = argv[1];
   b = argv[2];
   cout << getOccurenceCount(a, b) << endl;
   return 0;
}
```

2 Question 2

I solved this question by the following logic:

1. Create an array A of pairs<weight, IQ> and sort by descending IQ.
2. Then we only look at the weights part of each array element. We need to find the largest subsequence that is ascending. Since that will mean that IQ is descending and weight is ascending which is what the question asks for.
3. To do this we:
   Create an array of arrays of pairs<weight, IQ> L.
   L[0][0] ← A[0];
   for i = 1 .. A.size() we set L[i] to equal the longest increasing subsequence that finishes at element i. (Remember that we are only looking at the weights part of the pair).
   We then iterate through L and find k that has L[k] as the longest in L.
   This is then (one of) the longest increasing subset(s).

Pseudocode:
Imagine that we are given the elephants in an array of struct elephant{double weight, double IQ}.

```
elephant A[n];
A.sort   (by elephant.IQ);
A.reverse;
elephant L[][];
L[0].push_back(A[0]);
for i = 1 .. A.size(){
        for j = 0 .. i - 1{
                if((A[j].weight < A[i].weight) && (L[i].size() < L[j].size() + 1)){
                        L[i] = L[j];
                }
        }
        L[i].push_back(A[i]);
}
/* L is now such that L[k] holds the largest increasing subsequence
        that ends with elephant A[k] */

elephant seq[];

int longestIndex = 0;
for i = 0 .. L.size(){
        if(L[i].size() > L[longestIndex].size()){
                longestIndex = i;
        }
}

seq = L[longestIndex];
/* seq now holds the required subsequence of elephants */
```

3 Question 3

I solved this question by the following algorithm:

1. Start at the root. Now find the maximal sum with this root allowed to be included.
2. To do this, we find the maximal sum of each of the children for two cases (we also keep a list for each case):
   (a) The child is allowed to be included.
   (b) The child is NOT allowed to be included.

   We then decide whether we should use the root node or the children of the root for a maximal sum.
   i.e.

   $$if \quad (\Sigma Children_{using\ their\ root\ node} - \Sigma Children_{not\ using\ their\ root\ node} > root):$$
   $$Don't\ use\ the\ root\ node$$
   $$else:$$
   $$Do\ use\ the\ root\ node$$
3. We continue recursively until we reach the leaves and hence reach the base case. We can then unravel the recursion to determine the solution to the original problem.

More detailed description and pseudocode:
Notes:

- Assume we are given a tree T with the president as the root node.
- In the below code, 0 represents a function or list in which the root node is allowed to be included whilst 1 represents a function or list in which the root node is NOT allowed to be included.
- getFunLevel(List l) is a simple function that adds the fun level of each of the elements in list l and returns the result. This function has not been coded for sake of succinctness.

```
/*
 * Function that returns a list of the maximal sum of the tree below
 * the node given in the input parameter with this root node allowed
 * to be included in this list.
 */
List getMaximal0(node n){

        /* Base case - when we have reached a leaf we just need to return
           a list containing just this leaf */
        if(n has no children){
                return List(n);
        }

        /* List that contains the nodes in the tree that are a
```

```
            maximal sum if the root node is not used.
            i.e. the root nodes of the children are allowed. */
        List sumChildren0;
        for each child c of n:
                sumChildren0.add(getMaximal0(c));


        /* List that contains the nodes in the tree that are a
            maximal sum if the root node is used.
            i.e. the root nodes of the children are NOT allowed. */
        List sumChildren1;
        for each child c of n:
                sumChildren1.add(getMaximal1(c));

        /* See pt2 of the logic above */
        if(getFunLevel(sumChildren0) - getFunLevel(sumChildren1) > getVal(n)){
                return sumChildren0;
        }else{
                sumChildren1.add(n);
                return sumChildren1;
        }
}

/*
 * Function that returns a list of the maximal sum of the tree below
 * the node given in the input parameter with this root node NOT allowed
 * to be included in this list.
 */
List getMaximal1(node n){
        /* Base case - if n is a leaf and we can't use it then we
            need to just return an empty list */
        if(n has no children){
                return List();
        }

        List sumChildren1;
        for each child c of n:
                sumChildren1.add(getMaximal0(c));

        return sumChildren1;
}


/*
 * The main function that gets the party list for the given tree T.
 */
List getPartyList(Tree T){
        /* We simply need to return the party list found by getting the
            maximal sum of the tree of the root node with the root node
            allowed to be included */
        return(getMaximal0(T->root));
}
```

4 Question 4

This problem can be defined by the following recurrence:
The minimum cost of cutting the stick is equal to the minimum of (the minimum cost of cutting the left part of the stick plus the right part of the stick for every possible cut) plus the cost of the initial cut.
More simply, if we define the cuts required to be at positions $c_1, c_2, ..., c_n$ and the end points of the stick to be c_0 and c_{n+1} respectively and $C(c_0, c_{n+1})$ to be the problem. We can split the original problem into two sub-problems:

$$C(c_0, c_{n+1}) = \min_{k \in (1,...,n)} [C(c_0, c_k) + C(c_k, c_{n+1})] + cost(c_0, c_{n+1})$$

Where $cost(a, b)$ is simply the cost of making a cut on the stick with endpoints a, b.
So $cost(a, b) = b - a$
i.e. the length of the stick.

We can generalise this recurrence to be valid for any sub-stick:

$$C(c_i, c_j) = \min_{k \in (i+1,...,j-1)} [C(c_i, c_k) + C(c_k, c_j)] + cost(c_i, c_j)$$

Thus, for a DP solution we simply recurse according to the above whilst keeping the value for any sub-problem we solve.

Pseudo-code:
Notes:

- Let C be an array that holds the location of all cuts, IN ORDER. Including the end points.

```
global variables:
        int dp[numCuts + 2][numCuts + 2]
        int C[numCuts];

int minCost(int i, int j){

        /* If there are no cuts in between point i and point j
           then the cost is 0 */
        if(j == i + 1){
                dp[i][j] = 0;
                return 0;
        }

        /* If we have already computed this we do not need to
           recompute */
        if(dp[i][j] != -1)
                return dp[i][j];

        /* Apply the recursion */
        dp[i][j] = minCost(i, i+1) + minCost(i+1, j);
        for(int k = i + 1; k < j; ++k){
                int tempMin = minCost(i, k) + minCost(k, j);
```

```
                if(tempMin < dp[i][j])
                        dp[i][j] = tempMin;
        }
        dp[i][j] += C[j] - C[i];


        /* return the answer */
        return dp[i][j];
}

int getMinCostOfStick(Stick s){
        return(minCost(0, s.numCuts + 1));
}
```

5 Question 5

A naive way to solve this is to simply:

1. If the first character of X matches the first character of Z we move one character ahead in X and Z and then recursively check.
2. If the first character of Y matches the first character of Z we move one character ahead in Y and Z and then recursively check.
3. If either 1 or 2 returns true then we have an interleaving.

However, this has time complexity of $O(2^n)$. To make the algorithm more efficient I have used a DP solution in which we store the solutions of sub-problems in a table.

Let us store solutions of the sub problems in a 2D table:
bool interLeaved[m][n]
Now interLeaved[i][j] == True iff (Z[0..i+j-1] is an interleaving of X[0..i-1] and Y[0..j-1])

Now for the sub-problems to be useful we need to be able to apply them to the original problem. There are 5 cases to 'expand' the sub problems and 1 base case:

1. Case 1 - base case when X and Y are empty they interleave an empty string Z. Thus interLeaved[0][0] = TRUE
2. Case 2 - X is empty:
   interLeaved[i][j] = True iff((Y[j - 1] == Z[j - 1] && interLeaved[0][j - 1]
3. Case 3 - Y is empty:
   interLeaved[i][j] = True iff((X[i - 1] == Z[i - 1]) && interLeaved[i - 1][0])
4. Case 4 - Current character of Z matches with current character of X but does NOT match current character of Y:
   interLeaved[i][j] = True iff((interLeaved[i - 1][j]) && Z[i + j - 1] == X[i - 1])
5. Case 5 - Current character of Z matches with current character of Y but does NOT match current character of X:
   interLeaved[i][j] = True iff((interLeaved[i][j - 1]) && Z[i + j - 1] == Y[j - 1])
6. Case 6 - Current character of Z matches both current character of X and Y:
   interLeaved[i][j] = True iff((interLeaved[i - 1][j] || interLeaved[i][j - 1]) && Z[i + j - 1] == X[i - 1], Y[j - 1])

Pseudocode:

```
bool isInterLeaved(string X, string Y, string Z, int m, int n){
        \* Create DP table and set all to false originally */
        bool interLeaved[m+1][n+1] = {FALSE};
```

```
        /* Note that i represents how many letters of X
           we are considering and j represents how many
           letters of Y we are considering */
        for(int i = 0; i <= m; ++i){
                for(int j = 0; j <= n; ++j){

                        \* Case 1 - The base Case:
                           two empty strings have an
                           empty string as an interleaving */
                        if(i == 0 && j == 0)
                                interLeaved[0][0] = TRUE;

                        /* Case 2 - X is empty */
                        else if(i == 0 && Y[j - 1] == Z[j - 1])
                                interLeaved[0][j] == interLeaved[0][j - 1];

                        /* Case 3 - Y is empty */
                        else if(j == 0 && X[i - 1] == Z[i - 1])
                                interLeaved[i][0] == interLeaved[i - 1][0];

                        /* Case 4 - Z[i + j - 1] matches X[i - 1]
                           BUT does NOT match Y[j - 1] */
                        else if(X[i - 1] == Z[i + j - 1] && Y[j - 1] != Z[i + j - 1])
                                interLeaved[i][j] = interLeaved[i - 1][j];

                        /* Case 5 - Z[i + j - 1] matches Y[j - 1]
                           BUT does NOT match X[i - 1] */
                        else if(        Y[j - 1] == Z[i + j - 1] &&
                                        X[i - 1] != Z[i + j - 1])
                                interLeaved[i][j] = interLeaved[i][j - 1];

                        /* Case 6 - Z[i + j - 1] matches X[i - 1]
                           AND matches Y[j - 1] */
                        else if(X[i - 1] == Z[i + j - 1] &&
                                                Y[j - 1] == Z[i + j - 1])
                                interLeaved[i][j] = (interLeaved[i - 1][j]
                                                || interLeaved[i][j - 1]);

                        else
                                /* Nothing matches so leave as FALSE */
                }
        }

        return interLeaved[m][n];
}
```

6 Question 6

I solved this question by the following logic:

1. Sort turtles in increasing order of (weight + strength).
2. For each i from 1..n find the **lightest** legitimate tower using turtles $T_0, ..., T_i$ for each height k = i..1. By using the solution for the lightest legitimate tower of height k - 1 using turtles $T_0, ..., T_j$ for j = i - 1.
   Note: The base case is when i = 0, the highest legitimate tower is simply T_0.

Explanation and more detailed description of algorithm:

- Let us number the turtles in a tower from **Top to bottom**.
- By ordering the turtles in increasing order of (weight + strength) we restrict the solution to only containing towers that increase in (weight + strength) from the top of the tower to the bottom of the tower. However we can show that this does not *miss* any optimal solutions:
  - Consider a tower that is optimal such that there exists two turtles T_i and T_{i+1} such that T_i is directly above T_{i+1} in the tower **and** $w_i + s_i > w_{i+1} + s_{i+1}$. i.e. An optimal solution that our dynamic programming solution would miss.
  - Consider swapping T_i and T_{i+1}:
  - T_{i+1} will definitely not crack since it has less weight on it than before.
  - The turtles above the pair are clearly not affected and the turtles below the pair are not affected since the same total weight is above them.
  - Thus we only need to consider T_i. Let w_{above} denote the total weight of the turtles above the pair. Thus, T_i must withstand $w_{above} + w_{i+1}$ i.e. $s_i > w_{above} + w_{i+1}$ must hold.

$$w_i + s_i > w_{i+1} + s_{i+1} \tag{1}$$
$$\therefore s_i > w_{i+1} + s_{i+1} - w_i \tag{2}$$

    Also, since T_{i+1} was in a legitimate tower before the swap:

$$s_{i+1} \geq w_{above} + w_i \tag{3}$$

    Then, by subbing (3) into (2):

$$s_i > w_{i+1} + (w_{above} + w_i) - w_i \tag{4}$$
$$\geq w_{i+1} + w_{above} \tag{5}$$
$$\therefore s_i > w_{above} + w_{i+1} \quad as\ required \tag{6}$$

  - Thus, we have shown that if there exists an optimal solution that does not follow the restriction that we place on the solution (namely that $w_i + s_i < w_{i+1} + s_{i+1}$ for all i), we can swap adjacent turtles without affecting the legitimacy of the tower. We can then continue swapping adjacent turtles for which $w_i + s_i < w_{i+1} + s_{i+1}$ is broken until we obtain a legitimate tower such that $w_i + s_i < w_{i+1} + s_{i+1}$ holds for all i. Thus, we have a different optimal solution to the given one, but of the same height.

Algorithm run through:
Consider we have 5 turtles with the following weight, strength values:
{1, 6}, {2, 2}, {3, 8}, {5, 4}, {2, 4}.

We then sort them in increasing order of weight + strength:

$$T_1 = \{2, 2\}$$
$$T_2 = \{2, 4\}$$
$$T_3 = \{1, 6\}$$
$$T_4 = \{5, 4\}$$
$$T_5 = \{3, 8\}$$

We then run through the algorithm:

```
i = 1:
        Tower[1] = 2                        \\contains only T1
i = 2:
        k = 2:
                Can Tower[1] be extended by adding T2 to the bottom? YES
                Tower[2] = 4
                \\contains T1, T2
        k = 1:
                Do we have a new lightest tower of size 1? YES
                Tower[1] = 2
                \\contains T1 since T2 is not LIGHTER than T1
i = 3:
        k = 3:
                Can Tower[2] be extended by adding T3 to the bottom? YES
                Tower[3] = 5
                \\contains T1, T2, T3 since T3.strength >= Tower[2]
        k = 2:
                Do we have a new lightest tower of size 2? YES.
                Tower[2] = 3
                \\contains T1, T3 since T3.weight < T2.weight
        k = 1:
                Tower[1] = 1
                \\contains T3
i = 4:
        k = 4:
                Can Tower[3] be extended by adding T4 to the bottom? NO
                // Since T4.strength < Tower[3].
        k = 3:
                Do we have a new lightest tower of size 3? NO.
                // Since if we extend Tower[2] by adding T4 to the bottom, it is heavier th
        k = 2:
                Do we have a new lightest tower of size 2? NO.
                // Since if we extend Tower[1] by adding T4 to the bottom, it is heavier th
        k = 1:
                Do we have a new lightest tower of size 1? NO.
```

```
                // Since T4.weight > Tower[1]
i = 5:
        k = 4:
                Can Tower[3] be extended by adding T5 to the bottom? YES.
                // Since T5.strength = 8 < Tower[3] = 5.

At this point we have finished since we have gone through all i <= numTurtles.
Thus an optimal solution is:
Tower[4] = {Tower[3], T5} = {T1, T2, T3, T5}
```