COMP3121 Assignment 3

Alex Nguyen z3379933

09/05/16

Question 1)

Design an algorithm which for every two sequences A and B gives the number of different occurrences of A in B, i.e., the number of ways one can delete some of the symbols of B to get A. For example, the sequence ba has three occurrences in the sequence baba: baba, baba, baba.

To determine a recursion formula to cycle through sub-problem solutions, we draw a table of substrings using the question example: $A =$ "ba", $B =$ "$baba$". Solving subproblems in both A and B will involve splitting both up into:

$A(substring) : (empty)$,"a","ba"

$B(mainstring) : (empty)$,"a","ba","bab","$baba$"

We can then see all the sub-problems by tallying up how many substrings in gradual components of B. See below:

Subsequence Table	(empty)	a	b
(empty)	1	0	0
a	1	1	0
ba	1	1	1
bab	1	1	2
baba	1	1	3

We tabulate the substring component (A) and count whether substring matches (from the start) of current 'component' string of B. If so, add 1 to count of subsequences*.

Also, each i (row in the table) will progress to a longer component of ($mainstring$) to consider. As a longer ($mainstring$) component will carry over all the subsequences from the shorter iteration (e.g. $i = 2$ to $i = 3$: "ba" to "bab" will have "ba" subsequences as well as any new ones found in bab for any ($substring$) component j).

Hence, the +1 count mentioned above should also be added to the previous $i - 1'th(mainstring)$ and $j - 1'th(substring)$. Otherwise if there is no new subsequence found, simply take the previous row's string - the number should be the same between ($mainstring$) components.

We can see that this leads to the 3 count of subsequences A in B. This can be summarised and implemented by the below recursion: Define the number of subsequences at i, j in the calculation table as $N[i][j]$,

$$N[i,j] = \begin{cases} N[i-1, j-1] + 1 & substring(j) \in mainstring(i) \\ 0 & i = 0, j \neq 0 \\ 1 & j = 0 \\ N[i-1, j] & otherwise \end{cases}$$

To get the number of subsequence occurence of B in A, simply look for the $N[i, j]$ value at indices indicating end of A and B say, m and n respectively.

See pseudocode below for rough illustrative algorithm $O(mn)$ to demonstrate the memoization efficiency.

```
memo[][] //cache array
for (i 1 ... n ) { //n is length of B
  for (j =1 ... m ) { //m is length of A
    if (mainstring(i).contains(substring(j)) memo[i][j] = memo[i-1][j-1] + 1;
    else if (i = 0 && j != 0) memo[i][j] = 0;
    else if (j = 0) memo[i][j] = 1;
    else memo[i][j] = memo[i-1]pj]

    return memo[n][m]
```

Question 2)

Design an algorithm which given the weights and IQs of n elephants, will find a longest sequence of elephants such that their weights are increasing but IQs are decreasing.

First perform modified merge sort so that all elephants are sorted on decreasing weight. This is achievable in $O(nlogn)$ time. As the first factor is ordered, we can now find the largest sequence of increasing IQ of elephants by solving subproblems for largest increasing subsequence in this sequence of (now weight decreasing) elephants.

Subproblems: For every $i \leq n$ find a subsequence of σ_i of decreasing elephant weight subsequence $DE_i =< e_1, e_2, ..e_i >$ such that:

1) $\sigma_j =< a_1 \dots a_j >$ is the longest increasing subsequence (comparing IQ values, represented by a_i) in DE_n for $1 \leq i < n$; (i.e. $a_i > a_{i-1} > a_{i-2} \dots$)

2) For convenience, σ_j ends in a_j

We add condition 2) as it allows us to use a given a_k to increment a current σ_{k-1} if it satisfies condition 1) (by comparing $a_k > a_{k-1}$) with confidence that this will result in the longest increasing subsequence at k. This is legitimate as all longest increasing subsequences have to end with an item a_l - we just specify which. Since we are finding the optimal solution for all possible subsequences ending in a_j, all ending subsequences will eventually be explored despite this restriction.

Hence, the recurrence can be defined using the above logic to recursively construct a set and take the set with maximum length:

$$\sigma_j = \begin{cases} a_j + \max\limits_{i=1\dots j-1}\{\sigma_i | LE(\sigma_i) & a_j > a_i\} \\ a_j & j = 1 \\ a_j & otherwise \end{cases}$$

As above, the base case subsequence will consist of the first encountered term. In this case $LE(\sigma_i)$ calculates the length of the input σ_i. Hence, the recursion builds by considering all previous subsequences $1 \leq j < n$ and picking the subsequence with maximum length ($LE(\sigma_i)$) such that the current term a_j is greater than the maximum (last) term inside σ_i, a_i.

If the current last term in subsequence σ_j is greater than the last of the previous max increasing subsequence (last picked σ_i), add a_j to form σ_j. Otherwise, look for a smaller (lower LE) subsequence such that $a_j > a_i$ and do the same - if this does not exist, then start a new subsequence: a_j alone.

Note: $LE(0) = 0, LE(1) = 1$ as base cases for length.

Following this recursion for $1 \leq j < n$ will produce the longest increasing subsequence at σ_n in terms of IQ. As elephants in DE_n were already pre-sorted on decreasing weight, this will produce the longest subsequence where both IQ is increasing and weight is decreasing.

Question 3)

Give an algorithm that makes a guest list for the party that maximises the sum of the fun ratings of the guests.

The recursion will revolve around fun in each node (parent p at level i) and compared to the sum of fun ratings for every direct sub-employee Σc_l (child nodes at level i - 1).

In each sub-problem choice, we choose the maximum between the total fun score: Let g be the grand-child node level and $F(n)$ be the total fun sum for all levels up to level n

a) The fun rating at the party including parent (boss at current level) including p: $F(g) + p$

b) Total fun rating at the party excluding p and instead keeping all p's k employees: $F(g) + \sum_{l=0}^{k} c_l$

At each node stage starting from 0, take the maximum between the two cases a) and b), memoize the max value and update the $F(i)$ and repeat for $0 \leq i \leq n$ levels . As there are $l = 0 \dots j$ parents at each level, this comparison has to be made for all parent nodes from $0 \dots j$ and the maximum fun value between each parent-$\sum$ child relationship. Continue this until level n is reached in the tree - the president, who is invited based on his fun compared to his executives.

Assume anyone on the same level as a p_i will not be able to come if any p_i is unable to come (fun rating of employees are more).Assume Child and Grandchild of leaf nodes (very bottom level) have fun score of 0.

For each level, the max of every parent node p_l and the sum of employee e_l's fun will be compared with other parent nodes. The maximum in these considered will be takent to give the maximum total fun level $F(i)$ at every given level. At the end of the recursion the $F(n)$ value will give the maximum fun value at the party. The recurrence is as follows:

$$F(i) = \begin{cases} \max\limits_{l=1\dots j} \{F(i-2) + p_l, F(i-2) + \sum_{l=0}^{k} c_l\} & i > 0\} \\ \sum_{l=0}^{k} p_l & i = 0 \end{cases}$$

To save the guest list (whether parent nodes are included or child nodes), simply build into the memoization code a line to save which expression was taken - a) or b). If a) expression was taken over b), add all a subset of $p_l | l = 0 \dots j$ nodes to the cahce at level i (remove children if they are existing) . Else, add $e_l | l = 0 \dots k$ children. The guest list can then be read once level n is reached and $F(n)$ has been generated.

Once all sub-problems have been memoized, the solution can be done by recursing the equation above. This is demonstrated in rough pseudocode below:

```
Guests MaxFun(n) {
  F[ ] = array length n for each level of tree
  G[ ] = maps subsequences, length n for each level of tree
  for i = n ... 1 //iterate backwards
  parent = funRating(i) + sum[funRating(i->grandchildren)]
  children = sum[funRating(i->children)]
  F[i] = max(funRating(i) + sum[funRating(i->grandchildren)],
           sum[funRating(i->children)]
           //implement recursion above - children = employees
  if (F[i] = parent)
    G[i] =<all nodes on same level as i (i. colleagues)>
    else
  G[i] =<all children (i.children)>

  return G
```

Question 4)

You have to cut a wood stick into several pieces at the marks on the stick. The most affordable company, Analog Cutting Machinery (ACM), charges money according the length of the stick being cut. Your boss demands that you design an algorithm to find the minimum possible cutting cost for any given stick.

Assume the stick can be divided into even 'indices' along the stick from $i = 0 \dots n$ that reflected where it can be cut. The recursion is set up as follows:

Let $C(i, j)$ be the minimal cost of cutting a rod between marks i and j on the rod. However, the initial cut of the rod is dependent on its length, given by $cLength(i, j)$. As the question defines the cost of each rod cut directly dependent on how long it is at time of cut, it will be the max length - beginning section (relative to overall $0 \dots n$).

$$cLength(i, j) = j - i$$

Also, the minimum base subproblem would be a rod of 1 length which is assumed to be impossible to cut further (1 sig fig cuts for easy price calculation) This occurs when the chop indices are 1 index apart:

$$C(i, i + 1) = 0$$

With the first chop at point i, the wood stick becomes split into 2 extra sections of length $l = i - 0$ and $l = n - i$, which will require further corresponding min costs of $C(0, i)$ and $C(i, n)$ for each subsequent cut. As the rod had initial length n, so $cLength(0, n) = n$. We can see that the cost of the cuts of this rod will be:

$$C(0, i) + C(i, n) + cLength(0, n)$$

We repeat this strategy on the first two terms (subproblems of the initial) until all required cuts are performed. At each stage, (if known) the optimal solution for $C(i, j)$ will give the best index to cut such that cost of that cut is minimised. This will build up to give minimal cost of cutting the original rod.

This can be generalised for subsequent costs for a sub-stick section between marks i and j. Assume that the optimal $C(i, j)$ are all known/computed (memoized) - the minimum cost for any stick with length indices between i and j then becomes:

$$C(i, j) = \begin{cases} 0 & j = i + 1 \\ \min\limits_{k=i+1 \dots j-1} \{C(i, k) + C(k, j)\} + (j - i) & otherwise \end{cases}$$

Note: $cLength(i, j) = j - i$ is included in recursion as there is a cost to cut the stick at each step equal to its current length: $j - i$.

Following the recursion for as many steps as the desired number of cuts permits will provide the optimal solution for cutting a rod between interval $C(i, j)$. We cache any optimal $C(i, j)$ values along the recursion in case of reuse - by symmetry, $C(i, j) = C(j, i)$ for efficiencies in calculating optimal solutions. This way, normally $O(2^n)$ algorithm is reduced to $O(n^2)$ in looking up n values and evaluating n sub-problems in recursion (where n is number of cuts/recursion steps).

As illustrated above, this recursion can be implemented to find the cost of rod length n by finding $C(0, n)$. This is illustrated in pseudocode below:

```
woodCost (n) {
  memo[][] //cache array
  for (length = 1 ... n ) {
    for (i = 1; i + length - 1 < n; i++) {
      j = i + length - 1;
      if (i == j) memo[i][j] = cost(i, j);
      else
        memo[i][j] = MAX_INT;
        for (k ... j)
        memo[i][j] = min(memo[i][j], memo[i][k-1] + memo[k+1][j] + cost(i, j));
        //compare current value with previous (as loop progresses will find
        //min of all possible)
        //above step is recursion above, with memo array to save recurring
        //sub-problems
    }
  }
}
```

Question 5)

Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Similarly to counting subsequences in Question 1) above, we incrementally check for interleavings of components of X and Y in Z.

Say, $X = 101$ and $Y = 01$, $Z = 10011$. Assuming that bit strings also operate like normal strings and contain an 'empty' trailing component (to indicate conclusion), we compare $(empty), x_1 \dots x_3$ with $z_1 \dots z_3$. If x_i or $(empty)$ match z_i and the preceding component (x_{i-1}) also matched, then the bit string component of X so far remains interleaved - we place a True boolean. However, if a non-matching (with z_i) is encountered, a False is placed, showing the interleaving is broken for the first $z_1 \dots z_3$ component.

This forms the first row of a tabular progression checking string all interleaving possibilities. We fill out the table, form a recursion and use the memoization technique to use the efficiencies of dynamic programming to reduce runtime.

Next stage is consider $x_1 \dots x_3, y_1$ and compare these to $z_1 \dots z_4$ instead - extending the sub-bitstring to compare to. We fill out in a similar manner, with the final boolean result determining whether the sub-bit string interleaves the current subset of X and Y.

This is completed for the example X, Y and Z below:

Subsequence Table	(empty)	1	0	1
(empty)	T	T	T	F
0	T	T	T	F
1	F	F	T	T

Note: the intersection of (empty) and (empty) above return True as Z also contains an (empty) value to compare to.

The boolean at row m, column n will reveal whether Z is an interleaving of X and Y. In this case, it returns true (as shown in the question example).

This strategy (checking all possible sub-bitstrings of X and Y with sub-bitstrings of Z) can be summarised from top-down in the below recurrence for $I[i, j]$ - the boolean for whether Z is interleaving of X and Y, where i and j are lengths of X and Y respectively:

$$I[i,j] = \begin{cases} I[i-1,j] & Z[i+j] = X[i] \\ I[i,j-i] & Z[i+j] = Y[j] \\ T & i = 0, j = 0 \\ F & otherwise \end{cases}$$

The tabulation above visualises the memoisation technique as each check for interleaving checks boolean value stored the previous step (left/previous column for X, up/previous row for Y).

Hence, throughout the recursion values can be easily stored in a 2D array/other cache as they are encountered and the $I[m, n]$ quickly accessed in $O(1)$ time to give whether Z is interleaving of X and Y.

However, as the table computes $(m+n)(n+1)$ boolean values depending on X and Y, the algorithm runs on $O(mn)$ time - $O(n^2)$ if X and Y are equal length.

```
boolean interleaving(X, Y, Z, i, j) {
  // i, j, k are indices inside X, Y, Z to refer to for recursive calls
    bool = F;
    if (i = length(X) \& j = length(Y) return true; // end of X and Y strings
                                                    //to check - return
    if (X[i] = Z[i+j]) value = interleaving(X, Y, Z, i-1, j)
    if (Y[j] = Z[i+j]) value = interleaving(X, Y, Z, i-1, j)
    return bool;
}
```