Homework 8

C-12.1 Binomial coefficients are a family of positive integers that have a number of useful properties and they can be defined in several ways. One way to define them is as an indexed recursive function, \( C(n, k) \), where the “C” stands for “choice” or “combinations.” In this case, the definition is as follows:

\[
C(n,0) = 1, \\
C(n,n) = 1, \\
\text{And, for } 0 < k < n, \\
C(n,k) = C(n-1, k-1) + C(n-1, k)
\]

a) Show that, if we don’t use memorization, and \( n \) is even, then the running time for computing \( C(n, n/2) \) is at least \( 2^{n/2} \)

b) Describe a scheme for computing \( C(n,k) \) using memoization. Give a big-oh characterization of the number of arithmetic operations needed for computing \( C(n, \text{ceil}(n/2)) \) in this case.

Sol

a. If we don’t use memoization, then every application of the recursive equation doubles the number of calls that we make, until we hit a boundary condition.

b. If we build Pascal’s triangle, then we can apply memoization to this problem. The total number of arithmetic operations needed to compute \( C(n,k) \) is \( O(n^2) \) in this case.

C-12.3 Show that, in the coins-in-a-line game, a greedy-denial strategy of having the first player, Alice, always choose the available coin that minimizes the maximum value of the coin available to Bob will not necessarily result in an optimal solution for her.

Sol Consider the problem instance \((1, 2, 1, 2, 1, 2, 1, 2, 3, 2)\). In order to deny Bob from getting the 3, Alice will choose 1, 1, 1, 1, and then get the 3, for a score of 7, whereas Bob gets 2, 2, 2, 2, 2, for a score of 10, and wins.

C-12.6 Given a sequence \( S = (x_0, x_1, x_2, ..., x_{n-1}) \) of numbers, describe an \( O(n^2) \) time algorithm for finding a longest subsequence \( T = (x_{i_0}, x_{i_1}, x_{i_2}, ..., x_{i_{k-1}}) \) of numbers, such that \( i_j < i_{j+1} \) and \( x_{i_j} > x_{i_{j+1}} \). That is, \( T \) is a longest decreasing subsequence of \( S \)

Sol There are two at least two ways to solve this problem. One is to sort \( S \) backward into \( T \), and then compute the common subsequence of \( S \) and \( T \). The other method is to compute at first for each position \( i \), the length of the longest decreasing subsequence ending at position \( i \), i.e., \( \text{temp}[i] \), and then find the longest decreasing subsequence using \( \text{temp}[i] \). The following code will do that.
A substring of some character string is a contiguous sequence of characters in that string (which is different than a subsequence, of course). Suppose you are given two character strings, X and Y, with respective lengths n and m. Describe an efficient algorithm for finding a longest common substring of X and Y.

Solution: We may modify the longest common subsequence algorithm for this problem as follows:

Algorithm LCS(X,Y):

Input: Strings X and Y with n and m elements, respectively
Output: For i = 1, ..., n, j = 1,...,m, the length L[i, j] of a longest common suffix of both the string X[1..i] = x1x2...xi and the string Y[1..j] = y1y2...yj
        for i=1 to n do L[i,0] = 0;
        for j=1 to m do L[0,j] = 0;
        for i=1 to n do
          for j=1 to m do
            if (xi = yj) then
              L[i, j] = L[i-1, j-1] + 1
            else
              L[i, j] = 0;
return array L

The max(L[i,j]) gives us the length of a longest common string, where i and j tells us where it’s located in X and Y.

C-12.13 Suppose we are given a collection $A = \{a_1, a_2, \ldots, a_n\}$ of n positive integers that add up to N. Design an $O(nN)$ time algorithm for determining whether there is a subset $B \subset A$ such that

$$\sum_{a_i \in B} a_i = \sum_{a_i \in A \setminus B} a_i.$$

Sol

Algorithm 3 Algorithm that $B \subset A$ such that $\sum_{a_i \in B} a_i = \sum_{a_i \in A \setminus B} a_i.$

```
1: procedure SUBSETSUM($A$)
2:     res ← new int[$n$][A.length/2]
3:     res[0][0] = True
4:     for $i$ from 1 to A.length do
5:         for $j$ from 1 to sum($A$)/2 do
6:             res[$i$][$j$] ← ans[$i-1$][$j$]
7:             if $A[i] \leq j$ then
8:                 res[$i$][$j$] ← res[$i-1$][$j$] or res[$i-1$][$j-A[i]$]
9:         end if
10:     end for
11: end for
12: return res[$n$][A.length/2]
13: end procedure
```

R-13.6 Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices be given by the below table:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2,3,4)</td>
</tr>
<tr>
<td>2</td>
<td>(1,3,4)</td>
</tr>
<tr>
<td>3</td>
<td>(1,2,4)</td>
</tr>
<tr>
<td>4</td>
<td>(1,2,3,6)</td>
</tr>
<tr>
<td>5</td>
<td>(6,7,8)</td>
</tr>
<tr>
<td>6</td>
<td>(4,5,7)</td>
</tr>
<tr>
<td>7</td>
<td>(5,6,8)</td>
</tr>
<tr>
<td>8</td>
<td>(5,7)</td>
</tr>
</tbody>
</table>

a) Draw G
b) Order the vertices as they are visited in a DFS traversal starting at vertex 1
c) Order the vertices as they are visited in a BFS traversal starting at vertex
C-13.3 Let $T$ be the spanning tree rooted at the start vertex produced by the depth-first search of a connected, undirected graph, $G$. Argue why every edge of $G$, not in $T$, goes from a vertex in $T$ to one of its ancestors, that is, it is a back edge.

**Sol** Let $(u, v)$ be an arbitrary edge of $G$, and without loss of generality, assume $u$.starttime $< v$.starttime. Then by the rule of DFS, the search must discover and finish $v$ before it finishes $u$. If the search direction is from $u$ to $v$ at the first time that search explores edge $(u, v)$, then $v$ is not discovered until that time, for otherwise the search would have already explored this edge in the direction from $v$ to $u$. In this case, $(u, v)$ is a tree edge. If the search explores $(u, v)$ first in the direction from $v$ to $u$, then it becomes a back edge. So for undirected graph, every edge of that graph is either a tree edge or a back edge, no cross edges.

C-13.10 Show that, if $T$ is a BFS tree produced for a connected graph $G$, then, for each vertex $v$ at level $i$, the path of $T$ between $s$ and $v$ has $i$ edges, and any other path of $G$ between $s$ and $v$ has at least $i$ edges.

**Sol** Suppose $T$ is a BFS tree produced from a connected graph $G$, then for each vertex $v$ at level $i$, the path between $s$ (the start point) and $v$ has $i$ edges, and any other path of $G$ between $s$ and $v$ has at least $i$ edges.

- The first claim is trivial. In a BFS, every node at a given level $i$ is directly accessible from some node at level $i - 1$ by definition. Thus to go from $s$ to any node at level
1, you must traverse a single edge. To get to any node in level 2, you must first traverse to level 1 via a single edge, then to the given node via another single edge. This proceeds until the ith level is reached, requiring i edges

- Breadth-first searches always search the nearest nodes for a given node v before exploring the neighboring nodes. Thus, if there existed a path \( p \in G \) such that \( |p| < i \), then that means at some point prior to v, a traversal was skipped, which is a contradiction. Thus for any other path \( p \in G \) connecting s with v, \( |p| \geq i \)

**C-13.11** The directed version of the BFS algorithm classifies nontree edges as being either back edges, but it does not distinguish between these two types. Given a BFS spanning tree T, for a directed graph, G, and a set of nontree edges, \( E' \), describe an algorithm that can correctly label each edge in \( E' \) as being either a back edge or cross edge. Your algorithm should run in \( O(n+m) \) time, where \( n \) is the number of vertices and \( m \) is the number of edges

**Sol** Construct an Euler tour, \( P \), of T, which visits each vertex first on the left, then possibly multiple times from below, and then finally on the right. Label each vertex, \( v \), with the index, \( L_v \), in \( P \) of the first visit for \( v \) (on the left) and the index, \( R_v \), of the last visit for \( v \) (on the right). Notice that a node, \( v \), is an ancestor of a node, \( w \), if and only if the interval \([L_w,R_w]\) is contained inside the interval \([L_v,R_v]\). Thus, given the labels of the vertices, we can perform a scan of the edges in \( E' \) and label each edge as being either a back edge or a cross edge in \( O(1) \) time each. Therefore, the running time of this labeling algorithm is \( O(n + m) \).

**C-13.13** In the pseudocode description of the directed DFS traversal algorithm we did not distinguish the labeling of cross edges and forward edges. Describe how to modify the directed DFS algorithm so that it correctly labels each nontree edge as a back edge, forward edge, or cross edge.

**Sol** The directed DFS algorithm given in the book correctly identifies discovery edges and back edges, but it labels some edges as forward/cross edges. To distinguish between these two, we can add another for-loop to the algorithm, which is executed after the first for-loop. This second for-loop considers each incoming edge, \( e = (w,v) \), for \( v \), and if \( w \) is active, then it labels \( e \) as a forward edge. Then, in the place where we were previously labeling an edge as forward/cross edge, we would now label it as a cross edge.