Math 182: Problem Set 4
Kenny Guo

Question 1:

Prove that if G is a connected undirected graph, there is a vertex v such that G remains connected when v, and all edges containing v, are removed.

Hint: Despite what it may sound like, this is a question about the algorithms we've discussed in class.

Proof. Since G is connected, we can obtain a BFS tree T that spans all vertices of G. T must contain some leaf, which we will choose to be v, such that removing it (and its edges to other nodes) does not disrupt the connectivity of T (and thus G), as there is a path between any other two nodes of G by nature of the BFS tree. □

Question 2:

Given a connected undirected graph $G = (V, E)$ and an edge $(x, y) \in E$, design an efficient algorithm to detect whether there is a cycle containing the edge (x, y). For full credit, this algorithm should run in $O(|V| + |E|)$ time.

Idea: If y is reachable from x (or vice versa) via a path that does not use (x, y), then including (x, y) would mean there is a cycle containing (x, y). We can test reachability by running BFS or DFS ($O(|V| + |E|)$ time) on the graph with (x, y) removed.

```
HasCycleContainingEdge(G=(V,E), x, y):
    Remove (x,y) from G
    T = BFS(G, root=x)
    if y in T:
        return True
    else:
        return False
```

Runtime is $O(|V| + |E|)$. Finding (x, y) to remove in an adjacency list is $O(|V|)$ time. BFS is linear time, and one can either test if y is added for each step during the process, or search for it in the BFS tree after, which is also linear time. □

This algorithm correctly detects whether there is a cycle containing (x, y).

Proof. Assume the algorithm returns True. Then by BFS, y is reachable from x via a path that does not include (x, y). Adding (x, y) to this path thus creates a cycle containing (x, y).

Assume G has a cycle containing (x, y). After removing (x, y), there will still be a path from x to y, and BFS will thus add y to the spanning tree it finds. The algorithm thus returns True. □

Question 3:

Scientists have collected n specimens and are attempting to sort them into two groups. They do this by examining pairs of specimens, and determining if they belong in the same groups or different groups. If it is unclear, they will simply not make a decision. They worry that they may have made inconsistent judgments: maybe it is impossible to sort the specimens into two groups.

Given n specimens and m judgments of "same" or "different" for pairs of specimens, design an efficient algorithm to determine whether the judgments are consistent. That is, determine whether or not the specimens can be labeled A or B, such that for each pair (i, j) labeled "same" i and j will be labeled the same, and for each pair (i, j) labeled "different", i and j will be labeled differently. For full points, the algorithm should run in $O(n + m)$ time.

Idea: Let the n specimens be nodes and the m judgments be edges labeled "same" or "different", and store these in an adjacency list. We then loop across all specimens, and perform a BFS-like algorithm starting from that node (and assigning it to an arbitrary group if unclassified), looking at its judgment-neighbors and grouping them accordingly, while appending newly visited nodes to a queue. If we ever reach a contradiction, we return False. If we make it through all specimens with no contradictions, we return True.

```
IsConsistent(specimens, judgments):
    n = len(specimens)
    m = len(judgments)
    group = new array length n
    for i = 1, 2, ..., n:
        group[i] = null
    A = adjacency list from judgments, storing "same"/"different"

    for root = 1, 2, ..., n:
        if group[root] = null:
            group[root] = 0
            Q = new queue containing root
            while Q is not empty:
                u = pop(Q)
                for (v, judgment) in A[u]:
                    if judgment = "same":
                        required = group[u]
                    else:
                        required = 1 - group[u]

                    if group[v] == null:
                        group[v] = required
                        Q.push(v)
                    else if group[v] != required:
                        return False
    return True
```

This algorithm runs in $O(n+m)$ time.

Proof. Instantiating `group` takes $O(n)$. Instantiating the adjacency list takes $O(n+m)$. BFS visits each node at most once, and each judgment is checked at most twice, for $O(n+m)$ time. Thus, total runtime is $O(n+m)$. □

This algorithm correctly checks consistency of judgments.

Proof. Assume the algorithm returns True. Since the algorithm looped across all specimens, it must have assigned them all a label. It also looped across all judgments, and when it examined an edge labeled "same", it made sure the endpoints had the same label, and when same thing for "different". Since the algorithm completed this loop, it never found a contradiction in the judgments, and thus, they were consistent.

Now suppose for contrapositive the algorithm returns False. This only happens when the algorithm finds an edge whose constraint contradicts labels that were already assigned: either when an edge labeled "same" connects two vertices that have already been forced to have different labels, or an edge labeled "different" connects two vertices that have already been forced to have the same label, within the same connected component. Thus, there is no way to satisfy all judgments at once, and they are inconsistent. □

Question 4:
Prove that if G has n vertices, all of which have degree at least $n/2$, then G is connected.
Hint: This question does not use algorithms, just counting.

Proof. WLOG, we assume G is simple. Suppose for contradiction that G is not connected, so there exist two vertices, x, y with no path between them. This also implies that we have at least two connected components. By the pigeonhole principle, one component, call it $G_1 = (V_1, E_1)$, has at most $floor(n/2)$ vertices. If n is odd, then clearly any vertex in V_1 cannot have an edge with $n/2$ other vertices, since all its neighbors must lie in G_1; a contradiction. But also, if n is even, any vertex can only share an edge with at most $|V_1|-1$, so its degree is strictly less than $n/2$. We reach contradiction. □

Question 5:
So far in this class, we have focused on algorithms with efficient runtime. One can also attempt to design *space-efficient* algorithms. Dynamic programming can be thought of as sacrificing space-efficiency in favor of runtime: instead of spending time recomputing solutions to subproblems, we spend space to store the solutions for re-use. In many cases it is possible to minimize the space used by dynamic programming algorithms by overwriting solutions that are no longer needed. This technique is referred to as using *rolling arrays*.

For this problem, you do not need to prove correctness or compute runtimes.

(a) Design a dynamic programming algorithm to compute the n-th Fibonacci number that only uses a 2-element array.

(b) Design a dynamic programming algorithm to solve the knapsack problem without replacement that uses a $2 \times (C+1)$ matrix (rather than the $k \times (C+1)$ matrix we used in class), where C is the total capacity of the knapsack. (Two arrays of length $C+1$ also work.)

(c) (Challenge) Design a dynamic programming algorithm to solve the knapsack problem without replacement using a 1-dimensional array of length $C+1$.

---

a) For Fibonacci numbers, we only need to store the last 2 computed Fibonacci numbers, and then we can overwrite as we go. We initialize with the base cases, and use modulo 2 to overwrite the earliest Fibonacci number in the array.

```
Fib(n):
    A = new array length 2
    A[0] = 0
    A[1] = 1

    if n <= 1:
        return A[n]

    for i = 2,3,...,n:
        A[i mod 2] = A[(i-1) mod 2] + A[(i-2) mod 2]

    return A[n mod 2]
```

b) The recurrence for Knapsack w/o replacement for item i is

$$K_{w,i} = \max\{K_{w-w_i,i-1} + v_i, K_{w,i-1}\},$$

which only requires the values from item $i-1$. Thus, we can overwrite $K_{\cdot,i-2}$ as we compute $K_{\cdot,i}$ in our rolling array, using modulo 2 once again.

```
KnapsackNoReplacement(weight, val, C, k):
    K = new 2 x (C+1) array
    for c = 0,1,...,C:
        K[c, 0] = 0
    for i = 1,2,...,k:
        cur = i mod 2
        prev = 1 - cur
        for c = 0,1,...,C:
            K[c, cur] = max(K[c, prev], K[c - weight[i], prev] + val[i])
    return K[C, k mod 2]
```

c) The idea is we keep track of one array, K, where $K[c]$ is the best value achieved with capacity c, using items processed up to that point. When looping across items, update the capacities where that item could fit, taking the maximum of the old best value at capacity c, or the best value at

remaining capacity $c - weight[i]$ plus the value of i. We update backwards from highest to lowest capacity so that the current item is not reused during the same round (i.e. $K[c - weight[i]]$ is not using a $K[c]$ value already updated under item i).

```
KnapsackOneArray(weight, val, C, k):
    K = new array length C+1
    for c = 0,1,...,C:
        K[c] = 0
    for i = 1,2,...,k:
        for c = C, C-1, ..., weight[i]:
            K[c] = max(K[c], K[c - weight[i]] + value[i])
    return K[C]
```