Hannah Kwon
[redacted]
Professor Hsieh

Final

Computer Science 180 - Introduction to Algorithms and Complexity

Question 1

(a) False
(b) False
(c) False
(d) Θ(nlogn)
(e) Θ(n^2logn)

Question 2

```
#dp = an array of size n
int fillArray (n, k) :
    dp[0] = k
    dp[1] = k
    dp[2] = dp[1] * k - k
    for idx in range(3, n) :
        dp[i] = dp[i-1] * k - dp[i-3] * (k-1)

return dp[-1] #return the last element
```

The above algorithm is a dynamic programming algorithm and takes a bottom-up approach. We use an array in which each index, i, represents the possible combinations for a subarray of size i. We know that the array of a size 1 have k possible combinations, and for size 2 we have k * k possible combinations. To find the combinations of the size 3 array, we can't do k * k * k since three consecutive elements are not supposed to be adjacent. We have to subtract k cases in order for this particular possibility. This could be computed by dp[i-3] * (k-1), which is ignoring the cases where the two previous numbers are repetitive. Then for each combinations it could be shown by dp[i] = dp[i-1] * k - dp[i-3] * (k-1).

This is a polynomial O(n) time complexity algorithm. The first three indexes can be computed by O(1) operations. Then, iterating from the 4th index to the end of the array takes O(n) time with an O(1) computation for each. Thus, the overall time complexity is O(n).

Question 3

```
solution(n, m, prereqs, values):
    next_classes = construct a reverse prerequisite array
    #(see reverse_prereqs function for the alg)

    no_prereq_array = array of courses with no prereqs
    k = size of no_prereq_array

    // Perform knapsack variation:
    memo = array of size k * m
    ks(memo, next_classes, k, m, prereqs)

ks(memo, next_classes, k, m, values):
    values = []

    append ks(memo, next_classes, k-1, m) to values

    queue = queue with: (k, 0)
    while queue is not empty:
        (course, prev_value) = pop item from queue

        if not memoized append to values:
        ks(memo, next_classes, k-1, m - values[course] - prev_value, values)
        for each subsequent_course in next_classes[course]:
            add to queue: (subsequent_course, values[course])

    return max(values) and add to memo

reverse_prereqs(prereqs):
    result = array of size n where each index contains an empty array

    for each index i in prereqs:
        prereq = prereqs[i]
        result[prereq].append(i)

    return result
```

In this solution we perform a modified knapsack. As one input, we are given an array of prerequisites. We first reverse this by constructing an array of arrays, in which each index i represents a course i+1, and the value at index i is an array of courses that can be taken immediately after completing the current course. For example, if the input prerequisite array is [0, 1, 1, 2], then the reversed array is [[2,3],[4],[],[]]

We then create a memoized array. The dimensions of the array are m x the number of courses with no prereqs. The m dimension represents the "cost" in the knapsack problem. It is the number
of courses that can still be taken. The other dimension represents each course with no prereq.

The variation of the knapsack goes like this: each knapsack iteration is called with a value k representing a course with no prereqs. We enumerate all possible classes that can be taken as a result of taking
this course k. For each possible followup class, we call the knapsack variation, recalculating the inputs for each call of knapsack. This is essentially the knapsack problem except instead of taking the max of two options in each call, we are calling knapsack multiple times for each possibility of taking a course and its followup courses.

This algorithm is O(mn^2). Let's say some fraction of courses have no prereqs. We will use n / t to represent the number of courses with no prereqs. Our memoized array has dimensions m x n/t. Since n/t courses have no prereqs, that means n - n/t courses are follow up courses. For each m x n/t calls to knapsack, we compute n - n/tm = (t-1)n/tm followup courses.

Question 4

Let us claim that this is NP-complete.
We can prove this by removing these nodes from the graph G to build a new Graph G''. Then we run DFS from each node to test for cycles, which is a polynomial time complexity.
As the hint gives, an instance of vertex cover is an undirected graph G and a positive integer k. In our reduction, we build a new graph G'' with the same vertices as G but with edge {u, v} of G replaced by a pair of oppositely directed edges (u,v) and (v, u). Let us call this new directed graph G''. Our instance of our new vertex set is G'', k which is the same k as in the instance of vertex cover. This reduction is a polynomial time complexity since it simply copies G but replaces each edge with two edges.
Now, let us show that there is a vertex cover of size k in G if and only if there is a subset of k vertices in G'' that can remove the cycle. First, assume that there is a vertex cover of size k in G. Now, remove these k vertices from G'' along with the edges that are associated with that node. Here for any directed edge (u,v) that is in G'', at least one of u or v must have been removed, because one of u or v must have been in our vertex cover. Thus, after removing these k nodes and their incident edges from G''. No two vertices have an edge between them, and consequently there can be no cycles.
Contrarily, assume that there exists a set S of k nodes in G'' whose removal breaks all cycles. By construction every pair of nodes u, v that have an edge between them in G have a cycle (u, v), (v, u) in G''. Since the removal of set S broke all cycles in G'', for each edge {u,v} that is associated in G, the set S must contain either u or v or could be both. Thus, S is a vertex cover of G. By this, we can prove that this problem is NP-complete.