Problem Set 7

1. Let G be a directed graph with vertices numbered 1 through n. Suppose that all edges of G are of the form (i, j) with $i < j$ (so edges always point from lower numbered vertices to higher vertices).

   (a) Show how to find the longest path in G in $O(m + n)$ time. The path can start and end at any vertex.

   (b) Show how to turn the Longest Increasing Subsequence problem into an instance of part (a).

   Solution. Let $L[i]$ be the length of the longest path ending at vertex i. Let S_i be the set of vertices pointing to vertex i (note that S_i can be computed for all i in $O(m)$ time). If S_i is empty then $L[i] = 0$. Otherwise we have
   $$L[i] = \max_{j \in S_i} L[j] + 1.$$
   We iteratively compute $L[1], \ldots, L[n]$ using this recurrence. Computing i runs in time proportional to its indegree, so computing all values takes $O(m + n)$ time.

   To find the path, find the i that maximizes $L[i]$. This is the end of the path. Then traverse backward to a node j with $L[j] = L[i] - 1$, and so on until you reach a node with no incoming edges.

   To solve the longest increasing subsequence problem, suppose that our array is $A[1], \ldots, A[n]$. Draw a directed edge from $A[i]$ to $A[j]$ if $i < j$ and $A[i] \leq A[j]$. The longest path in this graph corresponds to the LIS.

2. At each of n equally spaced points along a highway you're considering purchasing a billboard. For each position i, you've projected that putting up a billboard in position i will net you a revenue of $R[i]$. However, due to local regulations, billboards must be spaced at least k positions apart.

(a) Give an $O(n)$ algorithm to find the maximum revenue that you can make.

(b) Suppose that in addition to the rule above, you're only able to put up M billboards total. Give an $O(Mn)$ algorithm to find the maximum revenue that you can make.

Solution. For part (a), let $OPT[i]$ be the maximum revenue you can make by buying billboards up to (and possibly including) position i. Consider the optimal revenue we can make among the first i billboards. If we don't buy billboard i then our optimal revenue is the best we can do among the first $i-1$, so $OPT[i-1]$. If we do buy billboard i then we make revenue $R[i]$, and our remaining billboards can be any valid set from the first $i-k$. So the most revenue we can make from the remaining billboards is $OPT[i-k]$. So we have

$$OPT[i] = \max(OPT[i-1], OPT[i-k] + R[i]).$$

To complete our recurrence we need a base case. We'll define OPT[i] to be 0 if $i \leq 0$ (after all we can't make any revenue using only the billboards numbered at most -3). Then we can compute OPT in linear time using the recurrence.

For part (b) let $OPT[i, m]$ be the optimal revenue using only the first i billboards and at most m billboards total. As before, we can either buy or not buy billboard i. This give the recurrence

$$OPT[i, m] = \max(OPT[i-1, m], R[i] + OPT[i-k, m-1]).$$

Again we need base cases. Let's say $OPT[i, 0] = 0$ for all i and $OPT[i, m] = 0$ whenever $i \leq 0$. We could implement this iteratively, but it's easy with memoization. See the implementation.

3. Say that a sequence $a_1, a_2, \dots, a_k$ is "zigagging" if either

$$a_1 \leq a_2 \geq a_3 \leq \dots$$

or

$$a_1 \geq a_2 \leq a_3 \geq \dots$$

Given an array A of length n, give an $O(n^2)$ algorithm to find the length of the longest zigzagging subsequence.

```
Initialize OPT to be an $n \times M$ array filled with dummy values $\perp$
Function computeOPT($i, m$):
    if $i \leq 0$ or $m = 0$ then
        return 0
    end
    if OPT[i,m] is not $\perp$ then
        return OPT[i,m]
    end
    OPT[i,m] = max(computeOPT[i-1,m], R[i] +
    computeOPT[i-k, m-1])
    return OPT[i,m]
```

Solution. Let `OPT[up, i]` be the longest zigzagging sequence ending at position i, where $A[i]$ is larger than or equal to the second-to-last term (i.e. where the sequence is "zagging up" to position i). Let `OPT[down, i]` be the longest zigzagging sequence ending at position i, where $A[i]$ is smaller than or equal to the second-to-last term.

Similar to the LIS problem, we have the following recurrence
`OPT[up, i]` = $\max_{j \leq i, A[j] \leq A[i]}$ `OPT[down, j] + 1`
`OPT[down, i]` = $\max_{j \leq i, A[j] \geq A[i]}$ `OPT[up, j] + 1`

In each case, if the max is empty we interpret it as 0. This recurrence can be computed in $O(n^2)$ time as with LIS.

4. Let G be an undirected graph. The Hamiltonian Path Problem asks whether there is a path in G that visits every vertex exactly once. Such a path is called a Hamiltonian Path. We will soon see that there (probably) no polynomial time algorithm for this problem. But we can still do a little better than brute force.

   (a) Consider the following brute force algorithm. Iterate over all permutations of the vertices. For each permutation, check if the permutation corresponds to a Hamiltonian path. What is the runtime of this algorithm?

   (b) Give a dynamic programming algorithm that runs in $O(n^2 2^n)$ time.

   (c) Show that this is an asymptotic improvement over the algorithm in part (a).

   Solution. Let S be a subset of vertices, and let v be a vertex in S.

Then define $H(S, v)$ to be true if there's a path contained in S that starts at v and visits every vertex of S exactly once.

Let S and v be arbitrary. Let $w_1, \ldots, w_k$ be the neighbors of v. Then by considering the possible second nodes along the path from v we get that

$$H(S, v) = H(S \setminus v, w_1) \vee \ldots H(S \setminus v, w_k).$$

We need a base case so we say $H(\{v\}, v)$ is true for all v. If all our base cases are true, how can we ever get an answer of False? We can define the "or" over an empty set of truth values to be false. (So if there are no adjacent vertices to v, then $H(S, v)$ will be false unless S only contains v itself.)

This recursion can be implemented in $O(n^2 2^n)$ time. There are at most $n2^n$ terms in the recurrence. Each takes at most $O(n)$ time to compute. (There's perhaps a little subtlety here with how we're representing subsets, and looking up terms indexed by subsets, but it's not important.)

1 Optional

1. In a Candy-Crush-like game you're given a linear array consisting of gems of various colors. You can blow up any consecutive sequence of k gems with the same color earning k^2 points. These gems are then removed from the array. You can keep blowing up sequences of gems until there are no gems left. Give a polynomial time algorithm to find the maximum score that you can achieve.