Problem 1. Maximum Independent Set. Let $G = (V, E)$ be an undirected graph with $n$ nodes. Recall that a subset of the nodes is called an independent set if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here well see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a path if its nodes can be written as $v_1, v_2, \ldots, v_n$ with an edge between $v_i$ and $v_j$ if and only if the numbers $i$ and $j$ differ by exactly $1$. With each node $v_i$, we associate a positive integer weight $w_i$.

Consider, for example, the following five-node path. The weights are the numbers drawn inside the nodes.

The goal in this question is to solve the following problem: Find an independent set in a path $G$ whose total weight is as large as possible.

(a) What is the maximum-weight independent set in the example?

(b) Give an example to show that the following algorithm does not always find an independent set of maximum total weight.

Start with $S$ equal to the empty set

while some node remains in $G$ do

Pick a node $v_i$ of maximum weight
Add $v_i$ to $S$
Delete $v_i$ and its neighbors from $G$

return $S$

(c) Give an example to show that the following algorithm also does not always find an independent set of maximum total weight.

Let $S_1$ be the set of all $v_i$ where $i$ is an odd number
Let $S_2$ be the set of all $v_i$ where $i$ is an even number
(Note that $S_1$ and $S_2$ are both independent sets)
Determine which of $S_1$ or $S_2$ has greater total weight, and return this one
(d) Give an algorithm with $O(n)$ running time that takes an $n$-node path $G$ with weights and returns the weight of the largest independent set.

(e) Now give an $O(n)$ algorithm to compute the maximum-weight independent set itself. Your algorithm should use values stored in the memoization array from your previous algorithm to trace back through the array and construct the optimal solution. (Hint: given the memoization array, how can you tell whether node $v_n$ is in the optimal solution?)
Problem 2. Longest Increasing Subsequence. In the `longest increasing subsequence problem`, you are given as input an unsorted array $A$ of length $n$, e.g.,

$$A = 5, 2, 10, 3, -1, 6, 8, 9, 3$$

The goal is to find the longest strictly increasing subsequence of $A$. The subsequence need not be contiguous. For example, the boxed numbers below indicate the longest increasing subsequence in our example:

$$A = 5, 2, 10, 3, -1, 6, 8, 9, 3$$

To approach this problem, it is useful to define a “helper” function $\text{LIS}(j)$ to compute the length of the longest increasing subsequence that `ends at` index $j$ (i.e., it must `include` item $A[j]$ in the subsequence). Here are examples for $j = 3$ and $j = 5$:

$$5, 2, 10$$

Therefore $\text{LIS}(3)$ should return 2, and $\text{LIS}(5)$ should return 1.

Follow these steps to design a dynamic programming algorithm to find a longest increasing subsequence:

1. Write a recursive algorithm for $\text{LIS}(j)$
2. Translate this recursive algorithm into a recurrence. Define $\text{OPT}(j)$ to be the length of the longest increasing subsequence ending at index $j$, and write a recurrence for $\text{OPT}(j)$.
3. Use this recurrence to write an iterative algorithm to compute the value of $\text{OPT}(j)$ and store it in the array entry $M[j]$ for all $j$.
4. Use the computed optimal values to find the value of the overall longest increasing subsequence (ending at any $j$).