1. Suppose you are given an array of numbers, some of which are marked as *icky*, and you want to compute the length of the longest increasing subsequence of A that includes at most k icky numbers. Your input consists of the integer k, the number array $A[1..n]$, and another boolean array $Icky[1..n]$.

   (a) Describe an algorithm for this problem using dynamic programming.

Solution: For any indices $i < j$ and any integer x, let $LILIS(i, j, x)$ denote the length of the longest increasing subsequence of $A[j..n]$ containing only numbers larger than $A[i]$ and containing at most x icky numbers. (*LILIS* is an abbreviation for *Longest Increasing Limited-Ickiness Subsequence*.)

If we add a sentinel value $A[0] = -\infty$ (with $Icky[0] = \text{FALSE}$), we need to compute $LILIS(0, 1, k)$.

This function obeys the following recurrence (with implicit "else"s between cases):

$$LILIS(i,j,x) = \begin{cases} -\infty & \text{if } x < 0 \\ 0 & \text{if } j > n \\ LILIS(i, j+1, x) & \text{if } A[i] \geq A[j] \\ \max\left\{ \begin{array}{c} LILIS(i, j+1, x) \\ 1 + LILIS(j, j+1, x-1) \end{array} \right\} & \text{if } Icky[j] \\ \max\left\{ \begin{array}{c} LILIS(i, j+1, x) \\ 1 + LILIS(j, j+1, x) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a 3-dimensional array $LILIS[0..n, 1..n, 0..k]$.

Each entry $LILIS[i, j, x]$ depends only on entries of the form $LILIS[\cdot, j+1, \cdot]$, so we can fill the array by decreasing j in the outermost loop, and considering the other two indices in any order in the inner loops.

The resulting algorithm runs in $\boldsymbol{O(kn^2)}$ ***time***. ■

Solution: To simplify our algorithm, let's add a sentinel values to both ends of the input arrays:

- $A[0] = -\infty$ and $Icky[0] = \textsc{False}$
- $A[n+1] = +\infty$ and $Icky[n+1] = \textsc{False}$

Now for any index i and any integer x, let $LILIS(i,x)$ denote the length of the longest increasing subsequence of $A[i..n+1]$ that starts with $A[i]$ and contains at most x icky numbers. (*LILIS* is an abbreviation for *Longest Increasing Limited-Ickiness Subsequence*.) We need to compute $LILIS(0,k)-2$.

This function obeys the following recurrence (with implicit "else"s between cases and the implicit base case $\max\varnothing = 0$):

$$LILIS(i,x) = \begin{cases} -\infty & \text{if } x < 0 \text{ or } (x = 0 \text{ and } Icky[i]) \\ 1 & \text{if } i = n+1 \\ 1 + \max\left\{ LILIS(j, x-1) \;\middle|\; \begin{array}{c} i < j \le n+1 \text{ and} \\ A[i] < A[j] \end{array} \right\} & \text{if } Icky[i] \\ 1 + \max\left\{ LILIS(j, x) \;\middle|\; \begin{array}{c} i < j \le n+1 \text{ and} \\ A[i] < A[j] \end{array} \right\} & \text{if } \neg Icky[i] \end{cases}$$

We can memoize this function into a two-dimensional array $LILIS[0..n, 0..k]$.

Each entry $LILIS[i,x]$ depends only on entries of the form $LILIS[j,\cdot]$ with $j > i$, so we can fill the array by decreasing i in the outer loop, and considering x in any order in the inner loop. Evaluating each entry $LILIS[i,x]$ requires $O(n)$ time to loop over all indices $j > i$.

The resulting algorithm runs in $\boldsymbol{O(kn^2)}$ ***time***. ■

(b) Describe an algorithm for this problem by reducing it to a standard graph problem.

Solution: First we add a sentinel value $A[0] = -\infty$ with $Icky[0] = \text{FALSE}$.

Then construct a directed acyclic graph $G = (V, E)$ with the following vertices and edges:

- $V = \{0, 1, 2 \ldots, n\} \times \{0, 1, 2, \ldots, k\}$. The number of vertices is $O(nk)$.
- E contains every edge $((i, x), (j, y))$ satisfying the following conditions:
  - $i < j$
  - $A[i] < A[j]$
  - $y = x + \big[Icky[j]\big]$

  The number of edges is $O(n^2 k)$.

Because $i < j$ for every edge $((i, x), (j, y))$, this graph has no cycles.

Every path from $(0, 0)$ to (j, y) in G corresponds to an increasing subsequence of $A[0 .. j]$ that starts with $A[0]$, ends with $A[j]$, and contains exactly $y \le k$ icky elements. Thus, we need to compute the longest path in G' starting at $(0, 0)$.

We can compute the longest path in G in $O(|V| + |E|) = \boldsymbol{O(kn^2)}$ ***time*** using the algorithm described in the textbook (or by negating all edge weights and using the shortest paths algorithm for dags described in lecture). ■

Think about later:

2. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices. Recall that a *palindrome* is a string that is equal to its reversal. How would you find the length of the longest palindrome that is the label of a path in G?

   (a) Describe an algorithm for this problem using dynamic programming.

Solution: Let $G = (V, E)$ be the input dag, and let $v.label$ denote the label of any node v.

For any two vertices u and v, let $LPPL(u, v)$ denote the length of the longest palindrome that is a label of a path from u to v in G. (LPPL is an abbreviation for *Longest Palindrome Path Length*.) This function obeys the following recurrence:

$$LPPL(u,v) = \begin{cases} 1 & \text{if } u = v \\ -\infty & \text{if } u.label \neq v.label \\ \max\left\{ \max\limits_{(u,u')} \max\limits_{(v',v)} \left\{2 + LPPL(u',v')\right\},\ 2 \right\} & \text{if } u.label = v.label \text{ and } (u,v) \in E \\ \max\limits_{(u,u')} \max\limits_{(v',v)} \left\{2 + LPPL(u',v')\right\} & \text{otherwise} \end{cases}$$

The first two base cases are straightforward. The third case includes a base case for the single edge (u, v), which is necessary for even-length palindromes, but also considers other possible paths from u to v.

We can memoize this function into a two-dimensional array $LPPL[1..|V|, 1..|V|]$, indexed by the vertices ***in topological order***. Each subproblem $LPPL[u, v]$ depends only on other subproblems $LPPL[u', v']$ where u' follows u and v follows v' in topological order. So we can fill the memoization array by decreasing i in the outer loop and increasing j in the inner loop.

We need to compute $O(|V|^2)$ array entries $LPPL[u, v]$. Evaluating each entry $LPPL[u, v]$ requires considering every edge (u, u') out of u together with every edge (v', v) into v. Without further optimization, our algorithm considers *every* pair of edges in the graph. So our overall algorithm runs in $\boldsymbol{O(|V|^2 + |E|^2)}$ ***time***.

Alternatively, instead of using an explicit topological sort, we could use two nested depth-first searches. The outer loop uses a depth-first search of G; at each vertex of G, we perform an independent depth-first search of the reversed graph $rev(G)$.

```
LongestPalindromePath(G):
  for each node u in postorder
      for each node v in reverse postorder
          ⟨⟨Evaluate LPPL[u, v]⟩⟩
```

For each of the $O(|V| + |E|)$ steps of the outer depth-first search, we spend $O(|V| + |E|)$ time performing an inner depth-first search of $rev(G)$, so the overall algorithm again runs in $O((|V| + |E|)^2) = \boldsymbol{O(|V|^2 + |E|^2)}$ ***time***. ■

(b) Describe an algorithm for this problem by reducing it to a standard graph problem.

Solution: Let $G = (V, E)$ be the input dag, and let $v.label$ denote the label of any node v. We construct a new edge-weighted directed graph $G' = (V', E')$ as follows:

- $V' = \{(u, v) \mid u, v \in V \text{ and } u.label = v.label\} \cup \{s\}$, where s is an artificial start node.
- E' contains three types of edges:
  - $(s, (v, v))$ with weight 1, for every vertex $v \in V$
  - $(s, (u, v))$ with weight 2, for every vertex $(u, v) \in V'$ such that $(u, v) \in E$
  - $((u, v), (u', v'))$ with weight 2 for every pair of vertices $(u, v), (u', v') \in V'$ such that $(u', u) \in E$ and $(v, v') \in E$

Altogether, G' has $O(|V|^2)$ vertices and $O(|E|^2)$ edges. G' is a dag, because every edge out of (u, v) leads to a vertex (u', v') where v' appears after v in any topological order for G, and there are no edges into s.

There is a one-to-one correspondence between palindrome paths in the input dag G and paths starting at s in our new dag G'. Specifically:

- Every even-length path $u_\ell \to \cdots u_2 \to u_1 \to v_1 \to v_2 \to \cdots \to v_\ell$ in the input graph G whose label is a palindrome (so $u_i.label = v_i.label$ for every index i) corresponds to a path $s \to (u_1, v_1) \to (u_2, v_2) \to \cdots \to (u_\ell, v_\ell)$ in our new dag G'.
- Similarly, every odd-length path $u_\ell \to \cdots u_2 \to u_1 \to v \to w_1 \to w_2 \to \cdots \to w_\ell$ in G whose label is a palindrome (so $u_i.label = w_i.label$ for every index i) corresponds to a path $s \to (v, v) \to (u_1, w_1) \to (u_2, w_2) \to \cdots \to (u_\ell, w_\ell)$ in G'.

In both cases, the total weight of the path in G' is equal to the number of vertices in the corresponding palindrome path in G. Thus, finding the longest palindrome path in G is equivalent to finding the ***longest path*** in G' starting at s.

We can compute the longest path in G' in $O(|V'| + |E'|) = \boldsymbol{O(|V|^2 + |E|^2)}$ ***time*** using the algorithm described in the textbook (or by negating all edge weights and using the shortest paths algorithm for dags described in lecture). ■