Rubric (for all dynamic programming problems): For a problem worth 10 points:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - 1 point for a clear English description of the function. (Otherwise, we don’t even know what you’re trying to do.)
  - 1 point for base case(s). —½ for one minor bug, like a typo or an off-by-one error.
  - 3 points for recursive case(s). (4 points on exams.) —1 for each minor bug, like a typo or an off-by-one error.
  - 1 point for a brief justification of the recurrence. One or two sentences per case is usually enough. (This is not required on exams.)

★ No credit for the rest of the problem if the recursive case(s) are incorrect.

- 1 point for describing the memoization data structure
- 2 points for describing a correct evaluation order; a clear picture is sufficient. If you use nested loops, be sure to specify the nesting order.
- 1 point for analyzing the running time
- It is not necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal structure—such as a subset, partition, subsequence, or tree—an algorithm that computes only the value or cost of the optimal structure is sufficient for full credit.

Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, but this is not required for full credit. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, data structure, or evaluation order. (But you still need to describe the underlying recursive function in English, and you still need to justify implicit recurrence.)

Official solutions provide target time bounds. Algorithms that are faster than this target are worth more points, and slower algorithms are worth fewer points, typically 2 or 3 points for each factor of \( n \). Partial credit is scaled to the new maximum score, and all points above 10 are extra credit. Any correct algorithm, no matter how slow, is worth at least 3 points, assuming the analysis is also correct.
1. Suppose you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are not necessarily integers. A standard interview question is to describe an algorithm to find the the largest sum of elements in a contiguous subarray $A[i..j]$. Unfortunately, this question is so standard that it has its own Wikipedia page: http://en.wikipedia.org/wiki/Maximum_subarray_problem.

So instead we’ll ask you to solve a slightly different problem. **Describe and analyze an algorithm that finds the largest product of elements in a contiguous subarray $A[i..j]$.** (Assume for purposes of analysis that you can multiply any two numbers in $O(1)$ time.) There are lots of solutions for this problem on the web, too, but they’re all either incorrect, slow, badly written, or all of the above. Really, you’re much better off figuring it out yourself.

For example, the maximum subarray sum in the following array is 19, and the maximum subarray product is 504.

$$\begin{array}{cccccc}
0 & 14 & -7 & 5 & & \\
-6 & 12 & -7 & 0 & 14 & -7 & 5
\end{array}$$

2. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it’s his turn, Elmo always takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely hates it when grown-ups let him win.)

(a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do not follow the same greedy strategy as Elmo.

(b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

(c) Five years later, Elmo has become a significantly stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a perfect opponent. [Hint: What is a perfect opponent?]
3. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or bases of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The sequence of an RNA molecule is a string $B[1..n]$, where each character $B[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the secondary structure of the RNA molecule.

We say that two base pairs $(i, j)$ and $(i', j')$ with $i < j$ and $i' < j'$ overlap if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most bonded base pairs are non-overlapping. Bonds between overlapping base pairs create so-called pseudoknots in the secondary structure, which are essential for some RNA functions, but are more difficult to predict. For this problem, we will simply assume that bonds between overlapping base pairs are impossible.

Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can be paired with at most one other base.
- Only A-U pairs and C-G pairs can bond.
- Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
- Overlapping base pairs cannot bond.

The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.

(a) Describe and analyze an algorithm that computes the maximum possible number of bonded base pairs in a secondary structure for a given RNA sequence.

(b) A gap in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the score of a secondary structure to be the sum of the squares of the gap lengths.\(^1\) Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

---

\(^1\)This score function is utterly fictional; real RNA structure prediction requires much more complicated scoring functions.
4. **Flappy Bird** is a popular mobile game written by Nguyễn Hà Đông, released in May 2013 and removed from the market in February 2014, reportedly because Nguyễn felt guilty over the game’s addictiveness. The game features a flying bird, which moves to the right at constant speed. Whenever the player taps the screen, the bird is given a fixed upward velocity; between taps, the bird falls due to gravity. The bird flies through a landscape of pipes until it touches either a pipe or the ground, at which point the game is over. Your task, should you choose to accept it, is to develop an algorithm to play Flappy Bird automatically.

Well, okay, not Flappy Bird exactly, but the following drastically simplified variant, which I will call **Flappy Pixel**. Instead of a bird, the main character Flappy is a single point, specified by three integers: horizontal position \( x \) (in pixels), vertical position \( y \) (in pixels), and vertical speed \( y' \) (in pixels per frame). Flappy’s environment is described by two arrays \( Hi[1..n] \) and \( Lo[1..n] \), where for each index \( i \), we have \( 0 < Lo[i] < Hi[i] < h \) for some fixed height value \( h \). The game is described by the following piece of pseudocode:

```plaintext
FLAPPYPixel(Hi[1..n], Lo[1..n]):
   y ← \left\lceil \frac{h}{2} \right\rceil
   y' ← 0
   for x ← 1 to n
       if the player taps the screen
           y' ← 10
       else
           y' ← y' - 1
       y ← y + y'
       if y < Lo[x] or y > Hi[x]
           return FALSE \( \langle \text{player loses} \rangle \)
   return TRUE \( \langle \text{player wins} \rangle \)
```

Notice that in each iteration of the main loop, the player has the option of tapping the screen.

Describe and analyze an algorithm to determine the minimum number of times that the player must tap the screen to win Flappy Pixel, given the integer \( h \) and the arrays \( Hi[1..n] \) and \( Lo[1..n] \) as input. If the game cannot be won at all, your algorithm should return \( \infty \). Describe the running time of your algorithm as a function of \( n \) and \( h \).