A ***subsequence*** of a sequence (for example, an array, a linked list, or a string), obtained by removing zero or more elements and keeping the rest in the same sequence order. A subsequence is called a ***substring*** if its elements are contiguous in the original sequence. For example:

- **SUBSEQUENCE**, **UBSEQU**, and the empty string ε are all substrings of the string **SUBSEQUENCE**;
- **SBSQNC**, **UEQUE**, and **EEE** are all subsequences of **SUBSEQUENCE** but not substrings;
- **QUEUE**, **SSS**, and **FOOBAR** are not subsequences of **SUBSEQUENCE**.

Describe and analyze ***dynamic programming*** algorithms for the following problems. For the first three, use the backtracking algorithms you developed on Wednesday.

1. Given an array $A[1\,..\,n]$ of integers, compute the length of a longest ***increasing*** subsequence of A. A sequence $B[1\,..\,\ell]$ is *increasing* if $B[i] > B[i-1]$ for every index $i \geq 2$.

2. In lecture we defined the recurrence of the above problem as:

$$LIS_{LEC}(i,j) = \begin{cases} 0 & i = 0 \\ LIS_{LEC}(i-1,j) & A[i] \geq A[j] \\ \max\begin{cases} LIS_{LEC}(i-1,j) \\ 1 + LIS_{LEC}(i-1,i) \end{cases} & A[i] < A[j] \end{cases} \tag{1}$$

   But when we worked out the problem in lab just now the recurrence (LIS_{LAB}) looks very different. Is one of them wrong? If not, what's the difference?

3. Given an array $A[1\,..\,n]$ of integers, compute the length of a longest ***decreasing*** subsequence of A. A sequence $B[1\,..\,\ell]$ is *decreasing* if $B[i] < B[i-1]$ for every index $i \geq 2$.

4. Given an array $A[1\,..\,n]$ of integers, compute the length of a longest ***alternating*** subsequence of A. A sequence $B[1\,..\,\ell]$ is *alternating* if $B[i] < B[i-1]$ for every even index $i \geq 2$, and $B[i] > B[i-1]$ for every odd index $i \geq 3$.

5. Given an array $A[1\,..\,n]$ of integers, compute the length of a longest ***convex*** subsequence of A. A sequence $B[1\,..\,\ell]$ is *convex* if $B[i] - B[i-1] > B[i-1] - B[i-2]$ for every index $i \geq 3$.

6. Given an array $A[1\,..\,n]$ of integers, compute the length of a longest ***weakly-increasing*** subsequence of A. A sequence $B[1\,..\,\ell]$ is *weakly-increasing* if each element is larger than the average of its two previous elements (i.e., $2 \cdot B[i] > B[i-1] + B[i-2]$ for all $i > 2$).

7. Given an array $A[1\,..\,n]$, compute the length of a longest ***palindrome*** subsequence of A. Recall that a sequence $B[1\,..\,\ell]$ is a *palindrome* if $B[i] = B[\ell - i + 1]$ for every index i.

Basic steps in developing a dynamic programming algorithm

1. **Formulate the problem recursively.** This is the hard part. There are two distinct but equally important things to include in your formulation.

   (a) **Specification.** First, give a clear and precise English description of the problem you are claiming to solve. Not *how* to solve the problem, but *what* the problem actually is. Omitting this step in homeworks or exams is an automatic zero.

   (b) **Solution.** Second, give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem. It generally helps to think in terms of a recursive definition of your inputs and outputs. If you discover that you need a solution to a *similar* problem, or a slightly *related* problem, you're attacking the wrong problem; go back to step 1.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input?

   (b) **Analyze running time.** Add up the running times of all possible subproblems, *ignoring the recursive calls*.

   (c) **Choose a memoization data structure.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. But some problems need a more complicated data structure.

   (d) **Identify dependencies.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

   (e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, you should consider the base cases first, then the subproblems that depends only on base cases, and so on. ***Be careful!***

   (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.