Implementing dynamic programming algorithms

Our third programming exercise at this course asks you to write a solver subroutine that computes the edit distance between two given strings. (Click here to fast-forward to the problem statement and the submission dialog.)

One possible solution to the exercise is to prepare an implementation that uses dynamic programming to compute edit distance.

Dynamic programming algorithms are a class of recursive algorithms that utilize memoization (or, *tabulation*) to avoid recomputing already-computed values of a recurrence. As such, the design of a dynamic programming algorithm consists of

> 1. a *recurrence* for the (cost of an optimum) solution, and
> 2. a *memoization* strategy for memory management.

Often it is useful to model the dependencies in a recurrence with a directed acyclic graph (a DAG), which then helps to design the memory management so that redundant recomputation is avoided and essentially the minimum amount of memory is used towards this end. That is, if possible, a value should not be stored in memory unless it will actually be needed in a yet-to-take-place evaluation of the recurrence.

Before we proceed with the formal problem statement, here are some general observations and guidance that you may find useful in implementing dynamic programming algorithms:

- **Start with the recurrence**. First it is important to get the recurrence right, both in terms of having a mathematically correct recurrence, and having a correct implementation of this recurrence, without worrying about redundant recomputation. This in many cases enables you to correctly solve small instances, and enables you to incrementally develop and test more intricate implementations with careful memory management.
- **Time-space tradeoffs**. Dynamic programming is an example of a *time-space tradeoff* in algorithm design—that is, we are trading increased storage space (memory) for decreased running time by avoiding recomputation. As such it is important to remember that one can always recompute if memory is tight, but usually this comes at considerable cost in running time. In this exercise, both the wall-clock running time and memory are limited, so your task is to make careful use of both resources.
- **Memory is a hard resource**. Comparing running time and memory as computational resources in general, often it is possible to get more time—indeed, one can perhaps wait a little bit longer (which of course translates to increased energy consumption)—but memory capacity is often a hard resource that is difficult to increase. In this exercise, we place you into this situation by placing a rather low memory limit that requires you to carefully think how to use the available memory.
- **Understand the dependencies in the recurrence**. Often dynamic programming can be viewed as the task of "completing the entries of a table, one by one". Except for the base cases—the "marginals of a table"—each entry in the table depends on "previous" entries in a systematic manner. More generally, one can capture the dependencies using a DAG, where each vertex is joined by arcs to its immediate dependent vertices.
- **The order of evaluation**. Any topological ordering of the vertices of the DAG ("the entries of the table") can be used to evaluate the recurrence ("complete the entries of the table"). Often there are several natural choices for such an ordering. For example, one may often proceed one row at a time, one column at a time, or diagonally one diagonal at a time.
- **Not all evaluations need to be stored**. The order of evaluation determines what needs to be stored in memory at each point in time in terms of the dependencies of the yet-to-be-evaluted vertices of the DAG ("entries of the table"). As such, it often pays off to carefully examine the different natural orders of evaluation. In this exercise, you need to carefully manage your memory if you want to succeed in all the scaling tests.
- **Make the evaluation of the recurrence fast**. Dynamic programming is all about fast evaluation of the recurrence, so it pays off to optimize the implementation of the recurrence, which will be evaluated for literally *billions* of vertices ("entries") in the scaling tests. That is to say, in this exercise it may be a good idea to try to optimize the implementation to perform essentially a linear scan through one or more arrays of integers, with a handful of variables to effect the required minimization steps as well as passing data from the previous evaluation(s) to the current one.
- **Parallelization**. Dynamic programming algorithms can typically be parallelized to the extent the dependencies permit *independent* evaluation of the vertices ("entries of the table"). Again the design choice for the order of evaluation is fundamental to enable parallelization through independent sets in the DAG. This exercise can be solved with full points without the use of parallelization, however.

Now we are ready for the problem statement and the submission dialog.