Dynamic Programming

Module: Algorithm Design Techniques

Overview

This week we introduce a technique called *dynamic programming*. Dynamic programming is used for *optimization problems*: problems that have many solutions, where each solution has associated with it some sort of value, and the goal is to compute the solution with the best, or optimal, value. Depending on the problem, optimal could mean either largest or smallest. Any problem that asks to find the best, or the optimal, or the longest, or shortest [something], is an optimization problem. Some examples: finding the shortest path, finding a minimum spanning tree, finding the best way to invest money, finding the best way to pack dishes in the dishwasher, finding the optimal way to assign classes to classrooms so that for e.g. the classroom unused time is minimized, finding the best way to load a backpack with a set of items so that the value is maximized, finding the maximum sub-array, finding the longest common subsequence of two genes. We'll explore dynamic programming through examples.

Introduction

- DP in a nutshell: express the problem recursively in terms of smaller sub-problems, parametrize each sub-problem using an index, and use this index to store solutions to sub-problems in a table.
- The first part of a DP solution is recursion. DP solutions are **recursive**. To be able to solve an optimization problem recursively, the problem has to have what's called *optimal substructure*
- To see whether a problem has optimal substructure, we ask: *Does an optimal solution to a problem include inside it optimal solutions to sub-problems?* If yes, we can express the solution to the problem recursively. That is, we solve some (carefully chosen) subproblems and combine their solutions in some way to get the optimal solution for the overall problem.
- Then we analyze the recursive calls: how many different recursive calls can there be? Is it possible that we solve the same problem more than once? If yes, then we'll end up solving the same subproblem more than one time and incur an unnecessary cost. We'll see that these overlapping calls can add up to be significant, and actually in many of the problems we'll see they add up to exponential time.
- The next step is simple yet brilliant: use a table to "cache" the solutions to subproblems, in order to avoid recomputing them. This avoids re-computing solutions to sub-problems.

- This is DP in a nutshell: (1) check for optimal substructure and formulate the problem recursively, and (2) use a table to "cache" the solutions to subproblems, in order to avoid recomputing them.
- With DP we often improve from exponential to linear. Huge impact in practice!!
- We'll introduce DP by looking at examples!

Comments

- Not all exponential algorithms can be improved. There exist problems for which only exponential solutions are known, and no-one has been able to find a faster (polynomial) solution. Does the fact that no-one has been able to find one mean there isn't one? This is the essence of the most well-known open question in theoretical computer science (Is P = NP)?
- Why is this called "dynamic programming"? In 1970s, using a table for storing and retrieving data was, at the time, reminiscent of "programming". Today this connection is gone and the term "dynamic programming" can seem a little unintuitive, because it has nothing to do with programming, nor it's particularly dynamic. Name aside, dynamic programming is a standard, elegant and powerful technique.
- **Dynamic programming and divide-and-conquer:** DP and DC are similar in that both solve the problem recursively, but there are differences: with divide-and-conquer we partition the problem into *disjoint* subproblems. With DP, the recursive subproblems are not disjoint, they overlap. Without storing partial solutions in a table, one will end up solving the same subproblem more than one time and incur an unnecessary cost. DP uses a table to "cache" the solutions to subproblems, in order to avoid recomputing them.