Dynamic Programming

Module: Algorithm Design Techniques

Overview

This week we introduce an algorithmic 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. We'll explore dynamic programming through examples.

What is dynamic programming (DP) and how does it work?

- In a nutshell, dynamic programming is recursion without repetition: we start from a recursive solution which does redundant computation, and we improve it by getting rid of that unnecessary computation.
- We are given a problem: for e.g. find the best way to cut a rod into pieces in order to maximize profit.
- The first part in a DP solution is establishing a (correct) *recursive formulation* to solve the given problem.
  - To be able to solve a 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.
  - Once we establish that the problem has optimal substructure, the next (and most important/difficult) step is to express computing a solution to our problem recursively, in terms of solutions to smaller sub-problems. That is, we solve some (carefully chosen) subproblems and combine their solutions in some way to get the optimal solution for the overall problem.
- Analysis (without DP). How long will it take to compute a solution to our original problem? Since the solution is recursive, we'll analyze it by writing a recurrence relation for its running time.
- Could we use DP to improve the running time? Is it possible that we solve the same subproblem more than once? If yes, then we'll incur an unnecessary cost which can add up to

be significant (actually in many of the problems we'll see they add up to exponential time). We can improve the solution to this problem by making the recursion "smart"/using DP.

- Recursive DP with a table: Extend the recursive solution to use a table to "cache" the solutions to subproblems, in order to avoid recomputing them. Before computing a subproblem, check if its answer is stored in the table, and if yes, return it from there. If not, compute it and store it. This makses sure that the problems are computed (at most) once and avoids re-computing.

- Analyze the recursive DP: How long does it take?

- Iterative DP: Eliminate recursion by computing the subproblems from smallest to largest, and storing solutions in a table. We refer to this as "iterative DP". It uses the table and fills it in the right order (which is the order recursion would have filled it), but without using recursion.

- Recursive DP or iterative DP? The running time is the same. Itierative DP may be better in practice because it avoids the recursion overhead. Also sometimes considered more elegant.

- 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

- The most difficult part of dynamic programming is finding a recursive formulation for the problem. It's not the table — adding the table is trivial.

- 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.

- 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)?