EECS3101 notes::An introduction to dynamic programming

Jerry Wu

2023-02-16

Dynamic programming

You've probably heard of the term "dynamic programming" at least once or twice if you're a computer science student or a prospective software engineer, but what exactly does it mean? Let us start with a *motivating* example.

The fibonacci sequence

Take the recurrence relation for calculating the nth term of the fibonacci sequence::

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

What is the runtime of the code on slide 4? The naive implementation involves directly converting the recurrence relation to code. After drawing the recursion tree, we can see that the runtime of the code is $O(2^n)$, which is very inefficient. Why is this the case? This is because some sub problems overlap (i.e. F_2 being calculated twice, F_5 being calculate 3 times, etc). We want to eliminate these unnecessary calculations in order to optomize our solution.

Memoization & bottom up

We can use two techniques to implement dynamic programming.

> *"If you can solve it using a loop, you should avoid using recursion.-Larry YL Zhang 2023"*

- Memoization entails using an external data structure to store the solution to subproblems in order to avoid redundant calculations. We're still using the original recursive solution for this technique. (*slide 8*)
- Bottom up entails solving smaller subproblems first so their answers are ready when the larger subproblems will be calculated. This technique is iterative and uses an array/list to index subproblem solutions. (*slide 9*)

Dynamic programming is most frequently used to solve problems involving optomization, i.e. finding the max or min of something, and problems that involve overlapping subproblems (fibonacci). Of course, we can't use dynamic programming to solve every problem.

Example 1:: Maximum sum with no adjacent items

Given an array of n coins, with various positive values, pick a subset of the coins with the constraing that you're not allowed to pick two adjacent coins. What is the maximum ammount that you can collect?

$$\xi_1 = \{3, 2, 7, 10\} | answer = 13(3 + 10)$$
$$\xi_2 = \{3, 2, 5, 10, 7\} | answer = 15(3 + 5 + 7)$$

The greedy approach for ξ_2 isn't optimal (we didn't pick 10).

The general idea

To devise a dynamic programming solution, we want to follow this formula::

- Define your subproblems.
- Find a recursive relationship between the solutions to subproblems, i.e. verify the optimal substructure (i.e. the optimal solution of a problem only depends on the optimal solution of a smaller subproblem) and base case.
- Implement the solution either using memoization or bottom up (the latter is usually preferred for simplicity's sake).

So we can apply these steps to the question. What describes the size of the problem? Well that would be the total number of coins in the set.

Define a subproblem as::

$$\rho(i) = \max\{1 < |coins| < i\}$$

Create a recursive array definition for our answer such that::

$$ans[i] = \max\{ans[i-1], A[i] + ans[i-2]\}$$

The full code is on **slide 15**.

Dynamic programming will only work if the problem has an optimal substructure!

A good example of a problem with no optimal substructure is the longest simple path problem (slide 17)