Dynamic Programming

(CLRS 15)

Laura Toma, csci2200, Bowdoin College

Today we discuss a technique called "dynamic programming". The idea is to use a table to store partial solutions to sub-problems.

Using a table for storing and retrieving data was, at the time, reminiscent of "programming". Today this connotation is gone and the term "dynamic programming" can seem a little unintuitive. Name aside, *dynamic programming* is a standard, elegant and powerful technique.

Dynamic programming is generally used for optimization problems: these are problems that have many solutions, each solution has a value, and the goal is to compute the best solution—either the largest or smallest (we'll see examples).

To solve a problem by DP, we start by asking if the problem has *optimal substructure*: 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 our 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 one time? If yes, then we'll end up solving the same subproblem more than one time and incur an unnecessary cost. The next step is brilliant and simple: use a table to "cache" the solutions to subproblems, in order to avoid recomputing them.

This is DP in a nutshell: check for optimal substructure, formulate the problem recursively, use a table. Done.

The hardest part of using the dynamic programming technique is finding the recursive structure of the problem and coming up with a recursive solution.

Dynamic programming and Divide and conquer: DP and DC are similar in that both colve the problem recursively, but there are differences: with divide-and-conquer you partition the problem into *disjoint* subproblems. In DP solutions, 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.

We will discuss dynamic programming by looking at a few examples.

1 Warm-up: Winning a board game

A game-board consists of a row of n fields, each consisting of two numbers. The first number can be any positive integer, while the second is 1, 2, or 3. An example of a board with $n = 6$ could be the following:

Let the board be represented two arrays $Cost[1..n]$ and $Jump[1..n]$.

17	2	100	87	33	14
1	2	3	1	1	1

The object of the game is to jump from the first to the last field in the row. The top number of a field is the cost of visiting that field. The bottom number is the maximal number of fields one is allowed to jump to the right from the field. The cost of a game is the sum of the costs of the visited fields. The goal is to compute the **cheapest** game.

Optimal substructure. Assume we know the optimal solution, and assume the first move is to jump say 1. So now we are at position 1+1 on the board. What do we need to do next?

Notation. We make the following notation: Let $Cheapest(i)$ represent the cost of the cheapest game starting at position i. Put differently, when called with argument i, $Cheapest(i)$ computes the cost of the cheapest game starting at position i.

When called with argument $i = 1$, $Cheapest(1)$ computes the cost of the cheapest game starting at teh beginning and thus is the solution we are looking for.

Towards the solution. We are on the board at position i and depending on the value of $Jump[i]$ we have a couple of choices:

- if $Jump[i] = 1$ then we need to jump to the next position $i + 1$ and add $Cost[i]$ to the cost of this solution.
- if $Jump[i] = 2$ then we have two choices: we can either jump to position $i + 1$ or $i + 2$, and continue looking for the cheapest game from there. In either case we need to add $Cost[i]$ to the cost of this solution.
- if $Jump[i] = 3$ then we have three choices: we can either jump to position $i + 1$ or $i + 2$ or $i + 3$, and continue looking for the cheapest game from there. In either case we need to add $Cost[i]$ to the cost of this solution.

When we are in the situation to make a choice, we need to evaluate each option's cost, and pick the best one (in this case, the smallest cost).

A recursive solution. The following procedure implements this:

```
Cheapest(i)

     IF i>n THEN return 0
     x=Cost[i]+Cheapest(i+1)
     y=Cost[i]+Cheapest(i+2)
     z=Cost[i]+Cheapest(i+3)
     IF Jump[i]=1 THEN return x
     IF Jump[i]=2 THEN return min(x,y)
     IF Jump[i]=3 THEN return min(x,y,z)

END Cheapest
```

Analysis: What is the asymptotic running time of the procedure?

Let $T(n)$ be the worst-case time to find the cheapest game (starting at position 1) on a board of size n. This is a recursive procedure, so we'll use a recurrence relation.

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \Theta(1)$$

Solving this recurrence may be tricky, but by looking at it our intuition tells us it is too slow. We'll prove this by showing a lower bound for $T(n)$:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + T(n-3) + \Theta(1) \\ &\geq 3\,T(n-3) \\ &= 3^2\,T(n-6) \\ &= \dots \\ &= 3^k\,T(n-3k) \\ &\geq 3^{n/3} \\ &= \Omega(3^{n/3}). \end{aligned}$$

This is exponenitial. Not good.

A more efficient algorithm. Is it possible to find a better algorithm? Some algorithms are exponential and no-one has been able to find faster (polynomial) solutions —- these problems are believed to *not* have polynomial solutions (so called NP-complete problems, we'll talk about this later).

Let's try and understand why the running time of Cheapest(i) is exponential: How many different sub-problems can there be? That is, for a given n, how many different Cheapest(i) can there be? Only n.

To get some intuition, draw the recurrence tree for a board of size 3 with the second row values all equal to 3. You'll see that there are a lot of overlapping subproblems and a subproblem may be "solved" many times.

We create a table (an array) T of size n in which to store our results of prior runs. $T[i]$ stores the result of $Cheapest(i)$. We initialize the table with say 0.

The modified algorithm would be as follows:

```
Cheapest(i)

     IF T[i] != ∅ THEN return T[i] <——— if it's been calculated already, retrieve it
     IF i>n THEN return 0
     x=Cost[i]+Cheapest(i+1)
     y=Cost[i]+Cheapest(i+2)
     z=Cost[i]+Cheapest(i+3)
     IF Jump[i]=1 THEN answer = x
     IF Jump[i]=2 THEN answer = min(x,y)
     IF Jump[i]=3 THEN answer = min(x,y,z)
     T[i] = answer <—— store it
     return T[i]

END Cheapest
```

Analysis: The cost of a recursive call is $O(1)$ and we fill each entry in the table at most once, so the running time is $O(n)$.