Dynamic Programming

Reading: CLRS 15

Dynamic Programming is generally used for optimization problems. Optimization problems have many "candidate solutions," each of which has a value, and the goal is to compute the solution with optimal value (depending on the problem, optimal could mean either the largest or smallest).

We need three conditons to be met in order to apply dynamic programming:

1. The problem needs to have *optimal substructure*: *Does an optimal solution to a problem include inside it optimal solutions to sub-problems?* Or, another way to put it: *Can an optimal solution to a problem be built up from 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.

2. The recursive algorithm must have *subproblem overlap*. Two different subproblems make the same recursive call to solve a (sub)subproblem, so the same (sub)subproblem is being solved twice.

3. We must be able to parameterize subproblems, so that each subproblem has a unique index associate with it. This is often the most difficult part.

If all of these are satisfied we can use dynamic programming to make the algorithm hugely more efficient, transforming an exponential time problem to a linear time problem! How? We use the subproblem indices to store solutions to subproblems as they are calculated and use those cached solutions to avoid calculating the answer to a subproblem more than once. Simple, but brilliant,

Dynamic Programming and Divide-and-Conquer: Dynamic Programming and Divide-and-Conquer are similar in that both solve the problem recursively and then combine solutions to subproblems to get the overall solution. But, Divide-and-Conquer partitions the problem into *disjoint* subproblems. In Dynamic Programming, on the other hand, the subproblems are not disjoint; they overlap. In fact, this is what makes Dynamic Programming so powerful.

NOTE: Why is it called "Dynamic Programming?" 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" is not intuitive.

Let's look at some examples.

A Simple Board Game

The problem: A game-board is represented by two arrays `cost[1..n]` and `jump[1..n]`. (Note the 1-based indexing.) The value `cost[1..n]` represents the cost of visiting that position. The value `jump[1..n]` represents the maximal number of positions one is allowed to jump to the right. For example, for $n = 6$:

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 position in the row. The cost of a game is the sum of the costs of the visited fields. The goal is to compute the **cheapest** game.

Simplifying Assumption: The optimal solution has two parts to it: its cost, and the set of jumps. We will first focus on how to find the cheapest cost. We will show that we can easily extend our solution to actually computing the jumps corresponding to the optimal cost.

Optimal Substructure. Suppose someone tells us the sequence of moves M for the cheapest game. Suppose the first move is to jump 2 spaces. So now we are at position 2 on the board, and we incurred the cost 17 of being in position 1. Claim: It must be that the remaining part of O (in our case, $133 - 17 = 116$) must be the optimal game from 2.

Why is this true? We'll use a contradiction argument: Let's assume that the claim was not true, namely that: (A) the remaining part of O is not the optimal game from 2; in other words, the optimal game from 2 is smaller than 116. Then we could take the optimal game from 2 (which is smaller than 116), add to it $cost[1]$ (in other words: jump from 1 to 2), and we get overall a solution to jump from 1 to the end that's cheaper than $116 + 17$. That's not possible, because we said we started with an optimal solution. Therefore our assumption (A) lead to a logical imopssibiliy. Therefore (A) cannot be right.

How do we actually solve the problem? So now we know the problem has optimal substructure, so we know we can think about it recursively. But...how? In fact, optimal substructure gives us a constructive way to approach the problem, because it amounts to the following:

Given a first jump J, a solution to the remaining game is the optimal solution to the overall game after the initial jump J. So, all we need to do is look at each possible first jump and solve the remaining game recursively.

We'll need to come up with a notation that will allow for a recursive formulation.

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 and returns the cost of the cheapest game starting at position i.

To find best game from 1, call with $i = 1$: `cheapest(1)`.

A recursive solution for `cheapest(i)`: We explore all moves, and pick the one that gives the cheapest game.

- if `jump[i] = 1`, we don't have any choice. We jump to the next position $i+1$ and recurse from $i+1$.
- if `jump[i] = 2`, we have two choices: we either jump to $i+1$ and find the optimal solution from there, or jump to $i+2$ and find the optimal solution from there. Since we don't know which one is the best one, we try both, and choose the cheapest.
- if `jump[i] = 3`, we have three choices: we either jump to $i+1$ and find the optimal solution from there, or jump to $i+2$ and find the optimal solution from there, or jump to $i+3$ and find the optimal solution from there. Since we don't know which one is the best one, we try all three, and choose the cheapest.

The following procedure implements this.

```
cheapest(i)
    if (i > n)
        return 0
    if (jump[i] == 1)
        x=cost[i] + cheapest(i+1)
        answer = x
    else if (jump[i] == 2)
        x=cost[i] + cheapest(i+1)
        y=cost[i] + cheapest(i+2)
        answer = min(x,y)
    else if (jump[i] == 3)
        x=cost[i] + cheapest(i+1)
        y=cost[i] + cheapest(i+2)
        z=cost[i] + cheapest(i+3)
        answer = min(x,y,z)
    return answer
```

Analysis: What is the asymptotic running time of the procedure? Since this is a recursive algorithm, we express the running time with a recurrence. Let $T(n)$ be the worst-case time to find the cheapest game (starting at position 1) on a board of size n.

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

Solving this recurrence may be tricky, but this is one of the situations where we don't care. All we want a lower bound for $T(n)$:

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

We are not sure precisely of the order of growth of the running time, but we know it's at least exponential. Exponential is not good.

Why exponential? Is it possible to find a better algorithm? When looking at an exponential algorithm, don't expect it can always be improved. Sometimes it can, and it feels like magic. Let's try and understand why the running time of `cheapest(i)` is exponential:

Question: How many different sub-problems `cheapest(i)` can there be?
Answer: At most n, one for each value of i.

Question: If there are only $O(n)$ different sub-problems, then why is the algorithm exponential?
Answer: It's because of **overlapping calls**. To get some intuition, draw the recurrence tree for a board of size 3 with the jumps all equal to 3. You'll see that there are a lot of overlapping subproblems and a subproblem may be solved many times.

Caching solutions. To avoid redundant calls to the same problem, we create a table $T[1..n]$ and $T[i]$ will store the result of `cheapest(i)`. Note that a subproblem `cheapest(i)` is parametrized by an index i, so we can map between a subproblem and its location in the table.

We initialize the table with a value that can be distinguished from the value returned by `cheapest(i)`; in this case we can use 0.

Dynamic Programming with Memoization

The recursive algorithm can be modified to use the table. It is called Dynamic Programming with memoization:

```
Cheapest(i)
    if (i > n)
        return 0
    // if it's been already calculated, retrieve it
    if (T[i] != 0)
        return T[i]
    if (Jump[i] == 1)
        x=Cost[i]+Cheapest(i+1)
        answer = x
    else if (Jump[i] == 2)
        x=Cost[i]+Cheapest(i+1)
        y=Cost[i]+Cheapest(i+2)
        answer = min(x,y)
    else if (Jump[i] == 3)
        x=Cost[i]+Cheapest(i+1)
        y=Cost[i]+Cheapest(i+2)
        z=Cost[i]+Cheapest(i+3)
        answer = min(x,y,z)
    T[i] = answer // store the answer
    return answer
```

Analysis: Let's visualize the recursion: To find the solution to our problem we call Cheapest(1). This will recursively call Cheapest(2), Cheapest(3), which in turn will call Cheapest(i) for $i > 1$, until i will reach the end of the table. At that point recursion will hit base case and will return a value for Cheapest(n), which will be stored in $T[n]$. The first value stored in the table will be $T[n]$, followed by $T[n-1]$, and so on. We can see that the recursion moves left to right, and the table is filled right to left.

Calling Cheapest(1) triggers a cascade of recursive calls that will fill the entire table. The key of the analysis is to separate between recursion and the actual computation.

Cost of recursion: Each Cheapest(i) is solved exactly once, because afterwards it is stored in $T[i]$ and is retrieved when needed. There are n different subproblems, and each is called recursively exactly once. Thefore the cost of recursion is $O(n)$.

Cost of computation: We know that calling Cheapest(1) will fill the entire table. Let's ignore recursion, and imagine that when calling Cheapest(i), $T[i+1]$, $T[i+2]$ and $T[i+3]$ have been computed and are stored in the table. Cheapest(i) will retrieve them in $O(1)$ time and compute the miminum of x, y, z in $O(1)$ time. Ignoring the cost of recursion, Cheapest(i) can be solved in $O(1)$ time.

Therefore the cost of Cheapest(1) is: $O(n) + O(n \cdot O(1)$, for a total of $O(n)$.

Iterative DP solution

It is actually possible to eliminate recursion completely and compute the cheapest game iteratively, but iterating through the table in the order in which recursion fills it (in this case: right to left). From an asymptotic point of view the recursiove and non-recursive solutions are equivalent; in practice recursion adds an overheadd so an iterative solution is preferred.

```
create table T[1..n+3] and set T[n+1]=T[n+2]=T[n+3]= 0

for (i = n down to 1)

    if (Jump[i] == 1)
        T[i] = cost[i]+ T[i+1]
    esle if (Jump[i] == 2)
        T[i] = cost[i]+ min(T[i+1], T[i+2])
    else if (Jump[i] == 3)
        T[i] = cost[i]+ min(T[i+1],T[i+2], T[i+3])

// the value in T[1] is the cost of the cheapest game
```

From finding optimal cost to finding the optimal game

To be continued.

Summary

With DP we improved from exponential to linear. However, 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?