Dynamic Programming

Module 4: 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.

1 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.
- For many problems using DP (ie storing the table) brings the running time from exponential to polynomial. That's significant!
- Time to look at some examples!

2 DP example: Winning a board game

The problem: A game board is represented by two arrays $cost[1..n]$ and $jump[1..n]$. The value $cost[i]$ represents the cost of visiting position i. The value $jump[i]$ 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.

Separating the optimal value from the rest of the solution: 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. Then, once we have an algorithm for that, we will think how we can extend it so that it computes not only the optimal cost, but also the jumps corresponding to the optimal cost. This will be our general strategy for all DP problems.

Optimal substructure: Does the problem have optimal substructure? Let's see. Consider someone tells us what is the cheapest game from 1 to the end; let's call this optimal game O; let's say for e.g. that the cost of O is 133. Let's imagine that the first move in O is to jump to location 2. 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 what's called a *contradiction argument*: Let's assume that the claim was not true, namely let's assume that: (A) the remaining part of O is not the optimal game from position 2. This means that the optimal game from position 2 is cheaper than 116. Then we could take the optimal game from 2 (which is cheaper than 116), add to it $cost[1]$, and we get overall a solution to jump from 1 to the end that wiill be cheaper than $116 + 17$. So what, you might ask? Well, that's not possible, because we said that O is the optimal game. So it;s not possible to get a game better than O. Therefore our assumption (A) lead to a logical imopssibiliy. Therefore (A) cannot be right.

So consider the optimal game: wherever the first jump is, the remaining part of the game must be the optimal game from that position. This shows that the optimal solution O contains within it optimal solutions to subproblems.

How do we actually solve the problem? So now we know the problem has optimal substructure, so we know we can approach it recursively. But...how?

In fact, optimal substructure gives us a constructive way to approach the problem, because it amounts to the following:

If someone told us the first jump, all we need to do is (jump and) recurse from there.

We'll need to come up with a notation that will allow us to solve the problem recursively.

Our notation. We need to decide how to parametrize the function we want to compute, that is, what parameters it should have and what they represent. We know we want to compute a function that returns the cheapest game, but what is the parameter in the game? It's the position we are at on the board! This gives us the idea to have the position on the board as a parameter in the recursive call.

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, we will call it with $i = 1$, $Cheapest(1)$.

A recursive solution for $Cheapest(i)$: Clearly, if we know the first move from i, we could just take it and recurse from there. But, we don't. So, what do we do? We explore all moves, and pick the one that gives 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

    ELSE

        IF Jump[i]==1 THEN
            x=Cost[i]+Cheapest(i+1)
            answer = x
        IF Jump[i]==2 THEN
            x=Cost[i]+Cheapest(i+1)
            y=Cost[i]+Cheapest(i+2)
            answer = min(x,y)
        IF Jump[i]==2 THEN
            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

END Cheapest
```

Correct: Why is this correct? At every step, the algorithm is exploring *all* possibilities and chooses the best one.

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. Then,

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

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. When running times are exponential, we won't care (in this class) about precise exponents and so on.

Why? Can we do better? 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 though, 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**.
Exercise: To get some intuition, draw the recurrence tree for a board of size 3 with the jumps all equal to 3. You can also try to draw the recursion tree for the board in this section.

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.

Recursive DP solution with memo-ization

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

```
Cheapest(i)

    IF i>n:  return 0

    //if it's been already calculated, retrieve it

    ELSE IF T[i] != 0:  return T[i]

    ELSE

        IF Jump[i]==1 THEN
            x=Cost[i]+Cheapest(i+1)
            answer = x
        IF Jump[i]==2 THEN
            x=Cost[i]+Cheapest(i+1)
            y=Cost[i]+Cheapest(i+2)
            answer = min(x,y)
        IF Jump[i]==2 THEN
            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

END Cheapest
```

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 THEN T[i] = cost[i]+ T[i+1]
     IF Jump[i]==2 THEN T[i] = cost[i]+ min(T[i+1], T[i+2])
     IF Jump[i]==3 THEN 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

Exercise: Now we have an algorithm to compute the cost of the cheapest game. Think of how you can extend it to compute the set of jumps that correspond to this optimal game.

Hint: there are at least two ways to do it. One is to extend Cheapest(i) so that when it makes a choice, it stores it in an additional table (so we'll keep a jump[] table in additional to the cost[] table; jump[i] will store the optimal jump at index i). The other way is to traverse the cost[] table and piece together the optimal jumps based on cost[i].

3 Comments, discussion, random notes, etc

- With DP we often improve from exponential to linear. Huge impact in practice!!
- 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)?
- 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.