Recitation 6 - Dynamic Programming

Sebastian Laudenschlager

sebastian.laudenschlager@colorado.edu

February 23, 2018

Dynamic Programming

- **Dynamic:** Multi-stage, time varying.
- **Programming:** Planning, decision-making (not writing code).
- Already seen lots of algorithms that split a problem into smaller subproblems.
- However, what if the subproblems have some sort of overlap, i.e. subproblems sharing subsubproblems?
- Don't want to solve a particular subproblem more than once, that would be wasteful.
- $\rightarrow$ Dynamic Programming: solve each subproblem just once, and save its result in a table.
- Whenever a solution to a particular subproblem is needed, just perform a look-up in the table.

Strategy

- Just like with greedy algorithms, dynamic programming algorithms usually used for optimization problems.
- As before, solutions not necessarily unique.
- Four general steps for dynamic programming algorithm:
  - Figure out the structure of the problem (overlapping subproblems?)
  - Recurse through subproblems, storing local solutions as necessary.
  - Compute value of optimal solution, in bottom-up fashion.
  - Construct the actual optimal solution, not just the optimal value.

Sample problem

- Consider the problem of a company buying steel rods, cutting them into shorter rods, and then selling the shorter rods.
- So for rods of length $i = 1, 2, \ldots, n$ we know the selling price p_i.
- Goal: Given a rod of length n inches, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- Note that not cutting the rod at all is a possibility.

Rod-cutting

- We can cut a rod of length n in 2^{n-1} different ways.
- If we find an optimal way of cutting the rod into, say k pieces, then the corresponding optimal decomposition is $n = i_1 + i_2 + \ldots + i_k$.
- Then the maximum revenue simply becomes $r_n = p_{i_1} + p_{i_2} + \ldots + p_{i_k}$.
- This is where the distinction between optimal solution value and the optimal solution itself comes into play.
  - Optimal value: The maximum revenue r_n.
  - Optimal solution: The set of cuts leading to the maximum revenue.

How to solve?

- Frame the maximum revenue as solutions to smaller subproblems, i.e. $r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$.
- First argument, i.e. p_n corresponds to making no cuts.
- Second argument, i.e. $r_1 + r_{n-1}$ corresponds to making an initial cut into two pieces of size 1 and $n - 1$.
- More generally, all arguments after p_n correspond to making a cut into two pieces of size i and $n - i$.
- We don't know what will be optimal, so we need to explore all possible values of i and pick the one that maximizes revenue.
- Once we make an initial cut, we have two smaller separate subproblems.
- Global optimum achieved by combining optima of the two smaller subproblems.

Towards pseudocode

- So rod-cutting exhibits optimal substructure.
- Can simplify the expression for r_n, i.e. express a decomposition as a left cut of length i, leaving a right side remainder of length $n - i$.
- Only allow the right rod to be further divided.
- So every rod piece of length n can be viewed this way: as some first piece and some remainder.
- Now we can rewrite the maximum revenue as

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

- This formulation only uses the solution to one subproblem, instead of two.

Pseudocode

```
def cutRod(p, n):
    if n == 0:
        return 0
    q = -Inf
    for i = 1:n
        q = max(q, p[i] + cutRod(p, n - i))
    return q
```

Running time

- What's the running time of `cutRod`?
- Turns out, it is $\mathcal{O}(2^n)$. Ouch.
- Why so inefficient?
- Lots of repeated calls to solve same subproblem.
- Wasteful. Can we do better?

Towards Dynamic Programming

- Top-down with memoization.
  - Write procedure as before, except we now check if the subproblem has been solved before.
  - If yes, just do a look-up of the solution to that subproblem.
  - If not, solve the subproblem as usual, with the exception that we save the result in a table of some sort.
- Bottom-up.
  - Sort the subproblems by size, and solve them in order.
  - Save all these results, so that we easily look up solutions to a particular subproblem as needed.
  - Usually better than top-down (better constant factors for asymptotic complexity).
- In either case, we need additional memory to store solutions of subproblems.
- $\rightarrow$ Time-memory trade-off

Rod cutting (bottom up)

```
def bottomUpRodCut(p, n):
    declare r[0, 1, ... , n] to be a new array
    r[0] = 0
    for j = 1:n
        q = -Inf
        for i = 1:j
            q = max(q, p[i] + r[j - i])
        r[j] = q
    return r[n]
```

How it works

- r will be an array to store the results of the subproblems.
- Initialize $r[0] = 0$, since a rod of length 0 earns no revenue.
- Solve each subproblem of size j in order.
- Note how we are directly referencing r in line 6 instead of using a recursive call.
- Store the result for subproblem j in $r[j]$.
- Finally, return $r[n]$ the optimal value we are seeking.

Example

- Suppose we have the following list of rod lengths and corresponding prices:

length i	1	2	3	4
price p_i	1	5	8	9

- For a rod of length $n = 4$, what is the optimal way of cutting the rod?
- $2^{4-1} = 8$ ways to cut it.
- Optimal revenue $r = 10$.
  - Two pieces of length $i = 2$.

Sanity Check

- Let's check to make sure our algorithm works:
- Initially, $r[0] = 0$.

$$
\begin{aligned}
j = 1, q = -\infty, & \; i = 1 : p[1] + r[0] = 1 \Rightarrow q = 1 \Rightarrow r[1] = 1 \\
j = 2, q = -\infty, & \; i = 1 : p[1] + r[1] = 2 \Rightarrow q = 2 \\
& \; i = 2 : p[2] + r[0] = 5 \Rightarrow q = 5 \Rightarrow r[2] = 5 \\
j = 3, q = -\infty, & \; i = 1 : p[1] + r[2] = 6 \Rightarrow q = 6 \\
& \; i = 2 : p[2] + r[1] = 6 \\
& \; i = 3 : p[3] + r[0] = 8 \Rightarrow q = 8 \Rightarrow r[3] = 8 \\
j = 4, q = -\infty, & \; i = 1 : p[1] + r[3] = 9 \Rightarrow q = 9 \\
& \; i = 2 : p[2] + r[2] = 10 \Rightarrow q = 10 \\
& \; i = 3 : p[3] + r[1] = 9 \\
& \; i = 4 : p[4] + r[0] = 9 \Rightarrow r[4] = 10
\end{aligned}
$$

Running time

- Had exponential running time before.
- Was the bottom-up dynamic programming approach an improvement?
- Certainly, now we have $\Theta(n^2)$ running time.

Optimal solution vs. optimal value

- So far, our rod-cutting algorithm simply returned the maximum revenue amount for a given rod of length n.
- What we really want, however, is the actual list of piece sizes so that we know *how* to cut the rod to achieve the maximum revenue.
- Can make a simple modification to our existing bottom-up algorithm.

Extended rod-cutting

```
def extendedRodCut(p, n):
    declare r[0, 1, ... , n], s[0, 1, ... , n]
    r[0] = 0
    for j = 1:n
        q = -Inf
        for i = 1:j
            if q < p[i] + r[j - i]:
                q = p[i] + r[j - i]
                s[j] = i
        r[j] = q
    return r, s
```

- Only difference is that we have an array s that gets updated with the optimal size i of the first piece to be cut off in a given subproblem j.

Writing the solution

```
def printSolution(p, n):
    (r, s) = extendedRodCut(p, n)
    while n > 0:
        print s[n]
        n = n - s[n]
```

When to apply Dynamic Programming

- Would be nice if we could split problem into subproblems and then solve those.
- Good indicator for possibly using a Dynamic Programming algorithm is thus optimal substructure.
    - Some (local) choice needs to made.
    - Rod cutting choice?
    - Given a way to find a locally optimal solution, can we build a set of solutions to subproblems?
    - Show that this set represents an optimal solution.

What else do we need?

- Need to have a polynomial number of subproblems, given some input size.
- If we try to recursively solve our given problem and we end up solving some subproblems multiple times, we say the problem has overlapping subproblems.
- Dynamic Programming takes advantage of the overlapping structure, at the expense of additional storage space.
- This is done via *memoization*, i.e. keep track of solutions to subproblems to avoid recomputing them.