Lecture 19: Dynamic Programming I: Memoization and Guessing

Lecture Overview

- Memoization and subproblems
- Examples
  - Fibonacci
  - Coin Row Problem
  - Robotic Coin Collection

Dynamic Programming (DP)

Big idea, hard, yet simple

- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution (“only”) via DP
- Particularly for optimization problems (min / max) (e.g., shortest paths)

* DP \approx “controlled brute force”
* DP \approx recursion + re-use

History

Richard E. Bellman (1920-1984)
Richard Bellman received the IEEE Medal of Honor, 1979. “Bellman . . . explained that he invented the name ‘dynamic programming’ to hide the fact that he was doing mathematical research at Research ANd Development (RAND) corporation under a Secretary of Defense who ‘had a pathological fear and hatred of the term, research’. He settled on the term ‘dynamic programming’ because it would be difficult to give a ‘pejorative meaning’ and because ‘it was something not even a Congressman could object to’ ” [John Rust 2006]
Fibonacci Numbers

\[ F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \]

Goal: compute \( F_n \)

Naïve Algorithm

follow recursive definition

\[
\text{fib}(n):
\begin{align*}
\quad & \text{if } n \leq 2: \text{ return } f = 1 \\
\quad & \text{else: return } f = \text{fib}(n - 1) + \text{fib}(n - 2)
\end{align*}
\]

\[ T(n) = T(n - 1) + T(n - 2) + O(1) \geq F_n \approx \varphi^n \]
\[ \geq 2T(n - 2) + O(1) \geq 2^{n/2} \]

EXPONENTIAL — BAD!

Memoized DP Algorithm

Remember, remember

Figure 1: Naïve Fibonacci Algorithm.
**memo** = { }

```python
fib(n):
    if n in memo: return memo[n]
    else: if n ≤ 2 : f = 1
          else: f = fib(n - 1) + fib(n - 2)
          memo[n] = f
          return f
```

*⇒* fib(k) only recurses **first** time called, ∀k

*⇒* only n nonmemoized calls: k = n, n - 1, . . . , 1

*⇒* memoized calls free (Θ(1) time)

*⇒* Θ(1) time per call (ignoring recursion)

**POLYNOMIAL — GOOD!**

* DP ≈ recursion + memoization

  * memoize (remember) & re-use solutions to **subproblems** that help solve problem

    - in Fibonacci, subproblems are \( F_1, F_2, \ldots, F_n \)

*⇒* time = # of subproblems · time per subproblem

  * Fibonacci: # of subproblems is \( n \), and time/subproblem is \( \Theta(1) = \Theta(n) \) (ignore recursion!)

**Bottom-up DP Algorithm**

```python
fib = {}
for k in [1, 2, . . . , n]:
    if k ≤ 2: f = 1
    else: f = fib[k - 1] + fib[k - 2]
    fib[k] = f
return fib[n]
```

* exactly the same computation as memoized DP (recursion “unrolled”)
• in general: topological sort of subproblem dependency DAG

\[ \ldots \xrightarrow{} F_{n-2} \xrightarrow{} F_{n-1} \xrightarrow{} F_n \]

• practically faster: no recursion
• analysis more obvious
• can save space: just remember last 2 fibs \( \Rightarrow \Theta(1) \)

lect19.py has Python code for different versions of Fibonacci.
[Sidenote: There is also an \( O(\lg n) \)-time algorithm for Fibonacci, via different techniques]

**Coin Row Problem**

Row of \( n \) coins with positive values (not necessarily distinct)

\[ c_0, c_1, c_2, \ldots, c_{n-1} \]

Goal: Pick maximum amount of money subject to constraint that no two coins adjacent in row can be picked up.

\[ 5, 1, 2, 10, 6, 2 \]

Odd/even partitions do not result in maximum. \( 5 + 2 + 6 = 13, 1 + 10 + 2 = 13 \). Maximum is \( 5 + 10 + 2 = 17 \).

**DP Solution to Coin Row Problem**

Let \( M(n) \) be maximum amount that can be picked up from row of \( n \) coins.

**Guess**: maximum amount that can be picked up does not include last coin.
Those without last coin: maximum amount is \( M(n - 1) \).

**Guess**: maximum amount that can be picked up does include last coin.
Those with last coin: maximum amount is \( M(n - 2) + c_{n-1} \).
The optimal solution is the maximum of the two guesses and is given by:

\[ M(n) = \max(c_{n-1} + M(n-2), M(n-1)), \quad n \geq 2 \]

\[ M(0) = 0 \quad M(1) = c_0 \]

This has \( \Theta(n) \) complexity, each \( M(i) \) computation takes \( \Theta(1) \) time given that \( M(< i) \) have all been computed.

**Example Computation**

<table>
<thead>
<tr>
<th></th>
<th>( c_0 )</th>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>( c_3 )</th>
<th>( c_4 )</th>
<th>( c_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M(0) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>( M(1) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( M(2) )</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( M(3) )</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( M(4) )</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( M(5) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>( M(6) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>

We iterate from smaller indices to larger to compute the maximum value solution.

We need to trace back to find coins that are actually picked. Start with \( M(6) \).
Clearly, \( c_5 = 2 \) is picked because \( 17 > 15 \). Next, look at \( M(5) \). \( c_4 = 6 \) is not picked.
(This makes sense since we have the constraint that adjacent coins cannot be picked.)
Looking at \( M(4) \), \( c_3 = 10 \) is picked because \( 10 + 5 > 7 \), implying that \( c_2 \) cannot be picked.
Looking at \( M(2) \), we see that \( c_1 \) is not picked (it could have been but is not an optimal choice), and \( c_0 = 5 \) is picked.

**Coding**

There are two ways to code a DP: (1) Recursive implementation with memoization table or dictionary, (2) Bottom-up iterative computation (smaller to larger problems).

Note that recurrence or iterative version gives value of optimal solution, not the solution itself. Look at code that traces back to find indices of coins that were selected in `lect19.py`. 

Robotic Coin Collection

We have a grid of squares and coins on selected squares. A robot moves from square (0, 0) at the bottom left to (n, m) on the top right only moving one step up or to the right each time. If it sees a coin, it picks it up. (The number of columns is \( n + 1 \) and the number of rows is \( m + 1 \).)

**Goal:** Maximize number of coins picked by choosing an appropriate path.

![Diagram of robotic coin collection](http://demonstrations.wolfram.com/PickingUpCoinsInAGrid/)

[Sidenote: There are an exponential number of possible paths!]

**Algorithm using DP**

- \( c_{ij} = 1 \) if there is a coin at \((i, j)\) else 0.
- \( S(i, j) \): largest number of coins robot can pick up until and including \((i, j)\).
- **Goal:** Maximize \( S(n, m) \).

The insight is that the robot can reach \((i, j)\) in only 2 different ways: From the left \((i - 1, j)\) square or from the bottom \((i, j - 1)\) square.

Therefore, we can write:

\[
S(i, j) = \max(S(i - 1, j), S(i, j - 1)) + c_{ij}, \ i \geq 1, \ j \geq 1
\]

\[
S(0, 0) = 0
\]

\[
S(i, 0) = S(i - 1, 0) + c_{i0}, \ i \geq 1
\]

\[
S(0, j) = S(0, j - 1) + c_{0j}, \ j \geq 1
\]

Try coding a recursive, memoized solution and an iterative solution. Trace the path taken by the robot in your implementation. What is the algorithm complexity?