Lecture 19
Dynamic Programming I: Rod Cutting & Fibonacci

Admin
→ PS 5 due tonight; PS 6 due in 9 days (only 1 grace day)
→ Re-grade requests on Quiz 2 due by 5:00 pm tonight

Today
→ Start to talk about DP (dynamic programming)
→ Rod-cutting problem → overview in lecture; more detailed treatment in recitation.

Imagine a simple problem:

```
  start
  1   2   3   4   5   6   7   8   9
```

Rules
- Start at lower left, end at upper right
- Collect the maximum number of gold coins possible
- Choose path consisting only of right (→) and up (↑) moves

Simple solution → try all such paths, each of which
consists of 1 → i on the bottom row, moving up
to top row, and continuing i → n on top row
(i can be 1, n, or between 1 and n).

○ = gold coin
(all coins have same value)
Analysis each such path calculation requires \( n \) additions
- there are \( n \) such paths
- \( \Theta(n^2) \)

Can we do better? Dynamic Programming Solution
- We might suspect that we can do better because the two paths above recompute the same summations for long stretches.
- In some sense, there are \((n-1)\) additions across the bottom row (from \( s \)) and \((n-1)\) additions across the top row (from \( t \)), from which all solutions can be generated with \( n \) more additions.

\[ \Theta(3n-2) = \Theta(n) \]

Some elements of dynamic programming
- Typically solving an optimization problem (but not always)
- Solving full problem involves solving subproblems
- Subproblems overlap in that same subproblem referred to repeatedly in solving overall problem.
- Efficiencies possible by organizing/reusing subproblem solutions
Rod-Cutting Problem

Scenario: You manufacture steel rods of some fixed length and sell them in segments of integer length. Because of the laws of supply and demand, the price per length is not constant. Given a set of prices for each possible length segment, how should you cut a given rod of length $n$ into integer-length segments to maximize revenue? No cost or loss is incurred with each cut.

<table>
<thead>
<tr>
<th>length, $i$</th>
<th>1 2 3 4 5 6 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>price, $P_i$</td>
<td>3 4 10 11 7 15 15 ...</td>
</tr>
</tbody>
</table>

Given

<table>
<thead>
<tr>
<th>max revenue, $r_i$</th>
<th>3 6 10 13 16 20 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i$</td>
<td>$P_i + P_{i-1}$</td>
</tr>
<tr>
<td>$P_{i-1} + P_{i-2}$</td>
<td>$P_{i-1} + P_{i-3}$</td>
</tr>
<tr>
<td>$P_{i-2} + P_{i-3}$</td>
<td>$P_{i-2} + P_{i-3}$</td>
</tr>
<tr>
<td>$r_i$</td>
<td>$r_{i-1}$</td>
</tr>
</tbody>
</table>

This is the set of answers our algorithm should find.

Formulations of problem solution:

1. Recursion on possible cuts

   \[ r_n = \max \{ P_n, r_i + r_{n-i}, r_{i+1} + r_{n-i-1}, \ldots, r_{n-1} + r_1 \} \]

   Seems to do "too much" work.
   For example, $r_i + r_{n-i}$ is same as $r_{n-1} + r_1$.

   maximum revenue obtainable for a rod whose initial uncut length is $n$.

   no cuts

   cut into $i$ & $n-i$, and recurse on "both pieces."
2. Recursion on right side only of possible first cuts
\[ \Gamma_n = \max (\rho_n, \rho_1 + \Gamma_{n-1}, \rho_2 + \Gamma_{n-2}, \ldots, \rho_i + \Gamma_{n-i}) \ldots \rho_n + \Gamma_i. \]
(letting \( \Gamma_0 = 0 \))

\[ \Gamma_n = \max_{1 \leq i \leq n} \left( \rho_i + \Gamma_{n-i} \right) \]
left-most piece \( \uparrow \)
recurse on right side only

Note:
- many subproblems of same type \((\Gamma_1, \Gamma_2, \ldots, \Gamma_{n-1})\)
- computing optimal solution for one subproblem requires using optimal solutions to smaller subproblems ("subsubproblems")
- these smaller subproblems are independent of the larger problem [in that once we hypothesize an initial cut at \( i \), the \( \Gamma_{n-i} \) subproblem is exactly the same as if we were starting a new "cutting problem" with a new rod of size \((n-i)\)]
- the subproblems overlap/repeat—\( \Gamma_4 \) and \( \Gamma_5 \) make use of \( \Gamma_3 \)
- the optimal solution to the overall problem consists of optimal solutions to subproblems → "optimal substructure" (seen previously for shortest paths).
- has similarities to divide & conquer, but here also optimize and try to find best way to divide problem, rather than using fixed division.
Implementations of (2)

(2a) Naïve/Direct \(\neq\) Dynamic Programming

```python
def r(p, n):
    if n == 0: return 0 \(\leftarrow\) defines \(r_0 = 0\)
    ans = -1 \(\leftarrow\) all revenues are positive, so this signifies initialized state
    for i in range (1, n+1):
        ans = max (ans, p[i] + r(p, n-i))
    return ans
```

THIS RUNS EXTREMELY SLOWLY!!! Let's see why.

Examine the pattern of recursion, where (i) is a vertex in a graph representing \(r(p, i)\) was called from parent in graph.

Consider \(r(p, 5)\):

\[ T(n) = \text{The number of calls to } r(p, i) \text{ resulting from an initial call to } r(p, n) \]

\[ T(n) = 1 + \sum_{j=0}^{n-1} T(j) \]

\[ T(0) = \frac{1}{2} \rightarrow T(n) = 2^n \]
Inefficient - Repeatedly recomputing same quantities over and over - corresponds to resolving some subproblems many times.

In solving $r(p, 5)$, we call $r(p, 2)$ 4 separate times

**Idea:** But there are only $n$ distinct subproblems $(T_1, T_2, \ldots, T_n)$. Compute the solution to each subproblem only once and reuse it.

**Two Ways:**

(i) Top-down + "memoize"

(ii) Bottom-up + use array of computed values

This is DYNAMIC PROGRAMMING! (both are)

2b) Top-down with memoization ← "Take a memo" - keep track of previous subproblem solutions & reuse

```python
new → memo = {} ← empty dictionary
def r(p, n):
    if n in memo: return memo[n]
    if n == 0: return 0
    ans = -1
    for i in range(1, n+1):
        ans = max(ans, p[i] + r(p, n-i))
    memo[n] = ans
    return ans
```

Running Time Analysis
- must solve $r(p, i)$ once for $1 \leq i \leq n$
- the for loop executes $i$ times
  $\implies \Theta(n^2)$ to compute $T_n$
2a) Bottom-up with array of computed values

\[ r = [0] * (n+1) \]

for \( j \) in range \( 1, n+1 \):
   \[ \text{ans} = \inf \]
   for \( i \) in range \( 1, j+1 \)
      \[ \text{ans} = \max (\text{ans}, p[i] + r[j-i]) \]
   \[ r[j] = \text{ans} \]

Subproblem dependence graph for \( r[5] \)

Vertices = subproblem to solve
Edges = dependencies

\[ \Rightarrow \text{Solving A requires solving B first} \]

Thus, the bottom-up approach considers subproblems in reverse topological sort order of the subproblem graph, and the top-down approach can be viewed as carrying out a depth-first search of the subproblem graph.

The algorithms shown here compute \( r_n \) (the value of the optimal solution) but not the list of cut lengths (the solution itself). With small additions the solution can also be retained.
Fibonacci Problem

\[ F_1 = 1, \quad F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} : \quad F = [1, 1, 2, 3, 5, 8, 13, \ldots] \]

Goal: Compute \( F_n \)
(a) Naïve / Direct \( \text{(not DP)} \)

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

Exponential running time due to redoing same work with repeated calls to same fib(i).
\[ T(n) = T(n-1) + T(n-2) + O(1) \geq 2T(n-2) + O(1) \geq 2^{n/2} \]

(b) Top-Down & memoized \( \text{(DP)} \)

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

- \( \text{fib}(k) \) only recurses first time called \( V_k \)
- only \( n \) nonmemo \( \Rightarrow \)

\[ T(n) = \Theta(n) \]
```python
fib = {}
for k in range(1, n+1):
    if k <= 2: f = 1
    else: f = fib[k-1] + fib[k-2]
    fib[k] = f
```

**Generalization**

Bottom-up does same computation as top-down, but the recursion is "unrolled".

Asymptotic running time of both is same.

Bottom-up can be practically faster because the constants are generally smaller (avoid overhead of recursion).

In some situations top-down is able to avoid unnecessary subproblems and thus be faster.