Dynamic Programming

- We know that we can use the divide-and-conquer technique to obtain efficient algorithms
- Sometimes, the direct use of divide-and-conquer produces really bad and inefficient algorithms
Fibonacci Numbers

• Fibonacci numbers are defined by the following recurrence:

\[
F_n = \begin{cases} 
F_{n-1} + F_{n-2} & \text{if } n \geq 2 \\
1 & \text{if } n = 1 \\
0 & \text{if } n = 0 
\end{cases}
\]

| n  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(F_n)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>...</td>
</tr>
</tbody>
</table>

A Recursive Algorithm

Algorithm Fibonacci(n)
if n <= 1, then:
   return 1
else:
   return Fibonacci(n-1) + Fibonacci(n-2)

• What is the running time?
Finonacci

- Why is it so slow?
- Can we do better?
  - Recursion is not always best!

Dynamic Programming

- Not really dynamic
- Not really programming
- Name is used for historical reasons
- It comes from the term “mathematical programming”, which is a synonym for optimization.
Dynamic Programming

- Dynamic programming improves inefficient recursive algorithms
- How?
  - Solves each subsubproblem once and saves the answer in a table
- Used to solve optimization problems
  - Many possible solutions
  - Wish to find a solution with the optimal value

Four Steps for Dynamic Programming

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution, typically in a bottom-up fashion
- Construct an optimal solution from computed information
Rod Cutting

- A company buys long steel rods of length $n$ and cuts them into shorter rods of length $1 \leq i \leq n$, which it then sells
- Each cut is free
- The management wants to know the best way to cut up the rods to make the most money

<table>
<thead>
<tr>
<th>length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

Example

- Can cut up a rod in $2^{n-1}$ different ways
  - You can choose to cut or not cut after the first $n-1$ inches
- What are the possible ways of cutting a rod of length 4 ($n = 4$)?
- What is the best way?
Initial Optimal Revenues

- Optimal revenues $r_i$, by inspection: ($i =$ length)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$r_i$</th>
<th>optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 (no cuts)</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2 (no cuts)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3 (no cuts)</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2 + 2</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>2 + 3</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>6 (no cuts)</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>1 + 6 or 2 + 2 + 3</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>2 + 6</td>
</tr>
</tbody>
</table>

Optimal Revenues

- We can determine the optimal revenue $r_n$ of a rod of length $n$ by taking the maximum of:
  - $p_n$: price by not cutting
  - $r_1 + r_{n-1}$: maximum revenue for a rod of length 1 and a rod of length $n-1$
  - $r_2 + r_{n-2}$: maximum revenue for a rod of length 2 and a rod of length $n-2$
  - $\ldots$
  - $r_{n-1} + r_1$
  - $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$
Optimal Substructure

• To solve a problem of size n, solve problem of smaller sizes. After making a cut, we have two subproblems. The optimal solution to the original problem incorporates optimal solutions to the subproblems.

• Example:

Simplifying

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

• Every optimal solution has a leftmost cut. In other words, there’s some cut that gives a first piece of length i cut off the left end, and a remaining piece of length n - i on the right
  • Need to divide only the remainder, not the first piece.
  • Leaves only one subproblem to solve, rather than two subproblems.
  • Say that the solution with no cuts has first piece size i = n with revenue \( p_n \), and remainder size 0 with revenue \( r_0 = 0 \).
Recursive Top-Down Solution

\textbf{CUT-ROD}(p, n)\footnote{n: rod length \text{ p: array that holds prices } p_i \text{ for lengths } 1 \leq i \leq n}

\begin{verbatim}
if n == 0
    return 0
q = -\infty
for i = 1 to n
    q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))
return q
\end{verbatim}

- Is it correct?
- Is it efficient?

- For \( n = 4 \):

\begin{verbatim}
    /\    /
   / \   /  \
5 /   /   \
\end{verbatim}

- Note: Labels from root to leaf give cut points
Dynamic-Programming Solution

- Don’t solve same subproblems repeatedly
- “Store, don’t recompute”
  - Trade-off
- Can turn an exponential-time solution to a polynomial-time solution
- Two approaches:
  - Top-down with memoization
  - Bottom up

Top-Down with Memoization

- Solve recursively, but store each result in a table
- To find the solution to a subproblem, first look in the table.
  - If there, use it
  - Otherwise, compute it and store in table
Memoized Cut-Rod

\[
\text{MEMOIZED-CUT-ROD}(p, n) \\
\text{let } r[0..n] \text{ be a new array} \\
\text{for } i = 0 \text{ to } n \\
\quad r[i] = -\infty \\
\text{return } \text{MEMOIZED-CUT-ROD-AUX}(p, n, r)
\]

\[
\text{MEMOIZED-CUT-ROD-AUX}(p, n, r) \\
\text{if } r[n] \geq 0 \\
\quad \text{return } r[n] \\
\text{if } n == 0 \\
\quad q = 0 \\
\text{else } q = -\infty \\
\quad \text{for } i = 1 \text{ to } n \\
\quad \quad q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n-i, r)) \\
\quad r[n] = q \\
\text{return } q
\]

Bottom-Up

- Sort the subproblems by size and solve the smaller ones first

\[
\text{BOTTOM-UP-CUT-ROD}(p, n) \\
\text{let } r[0..n] \text{ be a new array} \\
\quad r[0] = 0 \\
\quad \text{for } j = 1 \text{ to } n \\
\quad \quad q = -\infty \\
\quad \quad \text{for } i = 1 \text{ to } j \\
\quad \quad \quad q = \max(q, p[i] + r[j - i]) \\
\quad \quad r[j] = q \\
\text{return } r[n]
\]
Running Time

- What is the running time of the previous two algorithms?

Subproblem graphs

- Directed Graph:
  - One vertex for each distinct subproblem
  - Has a directed edge \((x, y)\) if computing an optimal solution to subproblem \(x\) directly requires knowing an optimal solution to subproblem \(y\)
Subproblem Graph for Rod-Cutting

• When \( n = 4 \):

Note: Identical to previous graph

Reconstructing a Solution

• We have only computed the value of an optimal solution
  • i.e. When \( n = 4 \), \( r_n = 10 \)
  • We still don’t know how to cut up the rod!
Rod-Cutting

EXTENDED-BOTTOM-UP-CUT-ROD\((p, n)\)

let \(r[0 \ldots n]\) and \(s[0 \ldots n]\) be new arrays

\(r[0] = 0\)

for \(j = 1\) to \(n\)

\(q = -\infty\)

for \(i = 1\) to \(j\)

if \(q < p[i] + r[j - i]\)

\(q = p[i] + r[j - i]\)

\(s[j] = i\)

\(r[j] = q\)

return \(r\) and \(s\)

Saves the first cut made in an optimal solution for a problem of size \(i\) in \(s[i]\).

To print out the cuts made in an optimal solution:

PRINT-CUT-ROD-SOLUTION\((p, n)\)

\((r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}\((p, n)\)\)

while \(n > 0\)

print \(s[n]\)

\(n = n - s[n]\)

---

Example

- PRINT-CUT-ROD-SOLUTION

<table>
<thead>
<tr>
<th>(i)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r[i])</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>13</td>
<td>17</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>(s[i])</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- How to cut up rod of length \(n = 8\)?

- ...of length \(n = 6\)?

- ...of length \(n = 7\)?
Problem

- Do exercise 15.1-5 on page 370

Summary

- Divide and Conquer is best used when there are no overlapping subproblems
- Otherwise, use dynamic programming!