Dynamic Programming: The Rod Cutting Problem

Version of October 26, 2016
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C).
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.

Main idea of DP

1. Analyze the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution (usually bottom-up)
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
- In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
- In D, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
  - In DP, many large subproblems reuse solution to same smaller problem.
- DP often used for optimization problems
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
- In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
- In D, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
  - In DP, many large subproblems reuse solution to same smaller problem.
- DP often used for optimization problems
- Problems have many solutions; we want the best one
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
- In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
- In D, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
  - In DP, many large subproblems reuse solution to same smaller problem.
- DP often used for optimization problems
- Problems have many solutions; we want the best one

Main idea of DP
Introduction

- Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)
  - Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
  - In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
  - In D, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
    - In DP, many large subproblems reuse solution to same smaller problem.
  - DP often used for optimization problems
  - Problems have many solutions; we want the best one

Main idea of DP

1. Analyze the structure of an optimal solution
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
- In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
- In D, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
  - In DP, many large subproblems reuse solution to same smaller problem.
- DP often used for optimization problems
- Problems have many solutions; we want the best one

Main idea of DP

1. Analyze the structure of an optimal solution
2. Recursively define the value of an optimal solution
Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
- In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
- In D, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
  - In DP, many large subproblems reuse solution to same smaller problem.
- DP often used for optimization problems
- Problems have many solutions; we want the best one

Main idea of DP

1. Analyze the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution (usually bottom-up)
Input: We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.  

Example: $n = 4$ and $p_1 = 1$, $p_2 = 5$, $p_3 = 8$, $p_4 = 9$

If we do not cut the rod, we can earn $p_4 = 9$

If we cut it into 4 pieces of length 1, we earn $4 \cdot p_1 = 4$

If we cut it into 2 pieces of length 1 & a piece of length 2, we earn $2 \cdot p_1 + p_2 = 9$

If we cut it into 2 pieces of length 2, we can earn $2 \cdot p_2 = 10$

There are more options, but the maximum revenue is 10

In general, rod of length $n$ can be cut in $2^{n-1}$ different ways, since we can choose cutting, or not cutting, at all distances $i$ ($1 \leq i \leq n-1$) from the left end.
**Input:** We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.

**Goal:** To determine the maximum revenue $r_n$, obtainable by cutting up the rod and selling the pieces.
Input: We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.

Goal: to determine the maximum revenue $r_n$, obtainable by cutting up the rod and selling the pieces.

Example: $n = 4$ and $p_1 = 1$, $p_2 = 5$, $p_3 = 8$, $p_4 = 9$
Rod Cutting

- **Input:** We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.

- **Goal:** to determine the maximum revenue $r_n$, obtainable by cutting up the rod and selling the pieces.

- **Example:** $n = 4$ and $p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9$
  - If we do not cut the rod, we can earn $p_4 = 9$
Input: We are given a rod of length \( n \) and a table of prices \( p_i \) for \( i = 1, \ldots, n \); \( p_i \) is the price of a rod of length \( i \).

Goal: to determine the maximum revenue \( r_n \), obtainable by cutting up the rod and selling the pieces.

Example: \( n = 4 \) and \( p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9 \)

- If we do not cut the rod, we can earn \( p_4 = 9 \)
- If we cut it into 4 pieces of length 1, we earn \( 4 \cdot p_1 = 4 \)
Rod Cutting

- **Input**: We are given a rod of length \( n \) and a table of prices \( p_i \) for \( i = 1, \ldots, n \); \( p_i \) is the price of a rod of length \( i \).

- **Goal**: to determine the maximum revenue \( r_n \), obtainable by cutting up the rod and selling the pieces

- **Example**: \( n = 4 \) and \( p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9 \)
  - If we do not cut the rod, we can earn \( p_4 = 9 \)
  - If we cut it into 4 pieces of length 1, we earn \( 4 \cdot p_1 = 4 \)
  - If we cut it into 2 pieces of length 1 & a piece of length 2, we earn \( 2 \cdot p_1 + p_2 = 9 \)
Input: We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.

Goal: to determine the maximum revenue $r_n$, obtainable by cutting up the rod and selling the pieces

Example: $n = 4$ and $p_1 = 1$, $p_2 = 5$, $p_3 = 8$, $p_4 = 9$

- If we do not cut the rod, we can earn $p_4 = 9$
- If we cut it into 4 pieces of length 1, we earn $4 \cdot p_1 = 4$
- If we cut it into 2 pieces of length 1 & a piece of length 2, we earn $2 \cdot p_1 + p_2 = 9$
- If we cut it into 2 pieces of length 2, we can earn $2 \cdot p_2 = 10$
Input: We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.

Goal: to determine the maximum revenue $r_n$, obtainable by cutting up the rod and selling the pieces

Example: $n = 4$ and $p_1 = 1$, $p_2 = 5$, $p_3 = 8$, $p_4 = 9$

- If we do not cut the rod, we can earn $p_4 = 9$
- If we cut it into 4 pieces of length 1, we earn $4 \cdot p_1 = 4$
- If we cut it into 2 pieces of length 1 & a piece of length 2, we earn $2 \cdot p_1 + p_2 = 9$
- If we cut it into 2 pieces of length 2, we can earn $2 \cdot p_2 = 10$
- There are more options, but the maximum revenue is 10
**Rod Cutting**

- **Input**: We are given a rod of length $n$ and a table of prices $p_i$ for $i = 1, \ldots, n$; $p_i$ is the price of a rod of length $i$.

- **Goal**: to determine the maximum revenue $r_n$, obtainable by cutting up the rod and selling the pieces

- **Example**: $n = 4$ and $p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9$
  - If we do not cut the rod, we can earn $p_4 = 9$
  - If we cut it into 4 pieces of length 1, we earn $4 \cdot p_1 = 4$
  - If we cut it into 2 pieces of length 1 & a piece of length 2, we earn $2 \cdot p_1 + p_2 = 9$
  - If we cut it into 2 pieces of length 2, we can earn $2 \cdot p_2 = 10$
  - There are more options, but the maximum revenue is 10

- In general, rod of length $n$ can be cut in $2^{n-1}$ different ways, since we can choose cutting, or not cutting, at all distances $i$ $(1 \leq i \leq n - 1)$ from the left end
We can calculate the maximum revenue $r_n$ in terms of optimal revenues for shorter rods

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

- $p_n$ if we do not cut at all
- $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and $n-1$
- $r_2 + r_{n-2}$ if we take the sum of optimal revenues for 2 and $n-2$
- $\ldots$
Optimal Solution

- We can calculate the maximum revenue \( r_n \) in terms of optimal revenues for shorter rods

\[
r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)
\]

- \( p_n \) if we do not cut at all
- \( r_1 + r_{n-1} \) if we take the sum of optimal revenues for 1 and \( n-1 \)
- \( r_2 + r_{n-2} \) if we take the sum of optimal revenues for 2 and \( n-2 \)
- \( \ldots \)

- Another approach. Set \( r_0 = 0 \) and

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

- Cut a piece of length \( i \), with remainder of length \( n-i \)
- Only the remainder, and not the first piece, may be further divided
Recursive Top-down Implementation

Cut-Rod($p, n$)

\[
\text{if } n = 0 \text{ then}
\]
\[
\quad \text{return } 0;
\]
end
\[
q = -\infty;
\]
for $i = 1$ to $n$ do
\[
\quad q = \max(q, p[i] + \text{Cut-Rod}(p, n - i));
\]
end
return $q$;
Recursive Top-down Implementation

Cut-Rod\( (p, n) \)

\[
\text{if } n = 0 \text{ then} \\
\quad \text{return } 0; \\
\text{end} \\
q = -\infty; \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad q = \max(q, p[i] + \text{Cut-Rod}(p, n - i)); \\
\text{end} \\
\text{return } q;
\]

Algorithm Time

\[T(n) = \begin{cases} 
1 + \sum_{0 \leq j \leq n - 1} T(j), & \text{if } n > 0, \\
1, & \text{if } n = 0.
\end{cases}\]
Recursive Top-down Implementation

\textbf{Cut-Rod}(p, n)

\begin{verbatim}
if \( n = 0 \)
  return 0;
end

q = -\infty;

for \( i = 1 \) to \( n \) do
  q = max(q, \( p[i] + \text{Cut-Rod}(p, n - i) \));
end

return q;
\end{verbatim}

\textbf{Algorithm Time}

- \( T(n) \): the total number of calls made to \text{Cut-Rod} when called with rod length \( n \)
Recursive Top-down Implementation

Cut-Rod($p$, $n$)

\[
\text{if } n = 0 \text{ then} \\
\quad \text{return } 0; \\
\text{end} \\
q = -\infty; \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad q = \max(q, p[i] + \text{Cut-Rod}(p, n - i)); \\
\text{end} \\
\text{return } q;
\]

Algorithm Time

- $T(n)$: the total number of calls made to Cut-Rod when called with rod length $n$

\[
T(n) = \begin{cases} 
1 + \sum_{0 \leq j \leq n-1} T(j), & \text{if } n > 0, \\
1, & \text{if } n = 0.
\end{cases}
\]
Recursive Top-down Implementation

\textbf{Cut-Rod}(p, n)

\begin{verbatim}
if \( n = 0 \) then
    return 0;
end
q = -\infty;
for i = 1 to n do
    q = max(q, p[i] + \text{Cut-Rod}(p, n - i));
end
return q;
\end{verbatim}

\textbf{Algorithm Time}

- \( T(n) \): the total number of calls made to \text{Cut-Rod} when called with rod length \( n \)

\[
T(n) = \begin{cases} 
1 + \sum_{0 \leq j \leq n-1} T(j), & \text{if } n > 0, \\
1, & \text{if } n = 0.
\end{cases}
\]

- Induction \( \Rightarrow T(n) = 2^n \)
Explanation of Exponential Cost

- Algorithm calls same subproblem many times

\[ p_1 + r_3 \]
\[ p_2 + r_2 \]
\[ p_3 + r_1 \]
\[ p_4 \]
After solving a subproblem, store the solution
After solving a *subproblem*, store the solution.

Next time you encounter the same subproblem, look up the solution, instead of solving it again.
Concept of DP

- After solving a *subproblem*, store the solution
  - Next time you encounter same subproblem, lookup the solution, instead of solving it again
  - Uses *space* to save *time*
After solving a *subproblem*, store the solution

- Next time you encounter same subproblem, lookup the solution, instead of solving it again
- Uses *space* to save *time*

Two main methodologies: top-down and bottom-up
After solving a subproblem, store the solution

- Next time you encounter same subproblem, lookup the solution, instead of solving it again
- Uses space to save time

Two main methodologies: top-down and bottom-up

- Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice
After solving a *subproblem*, store the solution
- Next time you encounter same subproblem, lookup the solution, instead of solving it again
- Uses *space* to save *time*

Two main methodologies: top-down and bottom-up
- Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice

Main idea of bottom-up DP
After solving a subproblem, store the solution
   Next time you encounter same subproblem, lookup the solution, instead of solving it again
   Uses space to save time

Two main methodologies: top-down and bottom-up
   Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice

Main idea of bottom-up DP
   Don’t wait until subproblem is encountered.
   Sort the subproblems by size; solve smallest subproblems first
After solving a *subproblem*, store the solution
- Next time you encounter same subproblem, lookup the solution, instead of solving it again
- Uses space to save time

Two main methodologies: top-down and bottom-up
- Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice

Main idea of bottom-up DP
- Don’t wait until subproblem is encountered.
- Sort the subproblems by size; solve smallest subproblems first
- Combine solutions of small subproblems to solve larger ones
\( p_i \) are the problem inputs.
\( r_i \) is max profit from cutting rod of length \( i \).
Goal is to calculate \( r_n \).
\begin{itemize}
  \item \( p_i \) are the problem inputs.
  \item \( r_i \) is max profit from cutting rod of length \( i \).
  \item Goal is to calculate \( r_n \)
  \item \( r_i \) defined by
    \begin{itemize}
      \item \( r_1 = 1 \) and \( r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i}) \)
    \end{itemize}
\end{itemize}
$p_i$ are the problem inputs.

$r_i$ is max profit from cutting rod of length $i$.

Goal is to calculate $r_n$

$r_i$ defined by

- $r_1 = 1$ and $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$

Iteratively fill in $r_i$ table by calculating $r_1, r_2, r_3, \ldots$

$r_n$ is final solution
$p_i$ are the problem inputs.

$r_i$ is max profit from cutting rod of length $i$.

Goal is to calculate $r_n$

$r_i$ defined by

- $r_1 = 1$ and $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$

Iteratively fill in $r_i$ table by calculating $r_1, r_2, r_3, \ldots$

$r_n$ is final solution

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>\ldots</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_i$</td>
<td>$p_1$</td>
<td></td>
<td></td>
<td></td>
<td>\ldots</td>
<td></td>
</tr>
</tbody>
</table>
Bottom-Up-Cut-Rod($p, n$)

\[
\begin{align*}
  r[0] &= 0; & \text{// Array } r[0...n] \text{ stores the computed optimal values} \\
  \text{for } j = 1 \text{ to } n \text{ do} & \\
  & \quad \text{// Consider problems in increasing order of size} \\
  & \quad q = -\infty; \\
  & \quad \text{for } i = 1 \text{ to } j \text{ do} \\
  & \quad \quad \text{// To solve a problem of size } j, \text{ we need to consider all} \\
  & \quad \quad \text{decompositions into } i \text{ and } j - i \\
  & \quad \quad q = \max(q, p[i] + r[j - i]); \\
  & \quad \text{end} \\
  & \quad r[j] = q; \\
  \text{end} \\
  \text{return } r[n];
\end{align*}
\]
Bottom-Up-Cut-Rod(p, n)

\[
\begin{align*}
    r[0] &= 0; & \text{// Array } r[0...n] \text{ stores the computed optimal values} \\
    \text{for } j = 1 \text{ to } n \text{ do} & \\
        & \text{// Consider problems in increasing order of size} \\
        q &= -\infty; \\
        \text{for } i = 1 \text{ to } j \text{ do} & \\
            & \text{// To solve a problem of size } j, \text{ we need to consider all} \\
            & \text{decompositions into } i \text{ and } j - i \\
            q &= \max(q, p[i] + r[j - i]); \\
        \text{end} & \\
        r[j] &= q; \\
    \text{end} & \\
    \text{return } r[n]; & \\
\end{align*}
\]

- Cost: \(O(n^2)\)
DP Bottom-up Implementation

Bottom-Up-Cut-Rod($p$, $n$)

```plaintext
r[0] = 0; // Array r[0...n] stores the computed optimal values
for $j = 1$ to $n$ do
    // Consider problems in increasing order of size
    $q = -\infty$;
    for $i = 1$ to $j$ do
        // To solve a problem of size $j$, we need to consider all decompositions into $i$ and $j-i$
        $q = \max(q, p[i] + r[j-i])$;
    end
    $r[j] = q$;
end
return $r[n]$;
```

- **Cost:** $O(n^2)$
- The outer loop computes $r[1], r[2], \ldots, r[n]$ in this order
Bottom-Up-Cut-Rod(\(p, n\))

\[
\begin{align*}
  r[0] &= 0; & \text{Array } r[0...n] \text{ stores the computed optimal values} \\
  \text{for } j = 1 \text{ to } n \text{ do} & \\
    & \quad \text{// Consider problems in increasing order of size} \\
    & \quad q = -\infty; \\
    & \quad \text{for } i = 1 \text{ to } j \text{ do} \\
    & \quad \quad \text{// To solve a problem of size } j, \text{ we need to consider all} \\
    & \quad \quad \text{decompositions into } i \text{ and } j - i \\
    & \quad \quad q = \max(q, p[i] + r[j - i]); \\
    & \quad \text{end} \\
    & \quad r[j] = q; \\
  \text{end} & \\
  \text{return } r[n];
\end{align*}
\]

- **Cost:** \(O(n^2)\)
  - The outer loop computes \(r[1], r[2], \ldots, r[n]\) in this order
  - To compute \(r[j]\), the inner loop uses all values \(r[0], r[1], \ldots, r[j - 1]\) (i.e., \(r[j - i]\) for \(1 \leq i \leq j\))
Algorithm only computes \( r_i \). It does not output the cutting.

Easy fix

- When calculating \( r_j = \max_{1 \leq i \leq j}(p_i + r_{j-i}) \)
  - store value of \( i \) that achieved this max in new array \( s[j] \).
  - This \( j \) is the size of last piece in the optimal cutting.

After algorithm is finished, can reconstruct optimal cutting by unrolling the \( s_j \).
Extended-Bottom-Up-Cut-Rod($p, n$)

// Array $s[0...n]$ stores the optimal size of the first piece to cut off
$r[0] = 0$; // Array $r[0...n]$ stores the computed optimal values

for $j = 1$ to $n$ do
  $q = -\infty$;
  for $i = 1$ to $j$ do
    // Solve problem of size $j$
    if $q < p[i] + r[j - i]$ then
      $q = p[i] + r[j - i]$;
      $s[j] = i$; // Store the size of the first piece
    end
  end
  $r[j] = q$;
end

while $n > 0$ do
  // Print sizes of pieces
  Print $s[n]$;
  $n = n - s[n]$;
end