Dynamic Programming - Theory and Applications

K. Subramani

1Lane Department of Computer Science and Electrical Engineering
West Virginia University

March 17, 2015
Outline

1. Dynamic Programming
Dynamic Programming

Main ideas
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.
Dynamic Programming

Main ideas
Dynamic Programming

Main ideas

1. Characterize the structure of an optimal solution.
Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.
The Rod Cutting problem

Given a rod of \( n \) inches, and a table of prices \( p_i \), \( i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces.

How many possibilities?

Example

<table>
<thead>
<tr>
<th>Length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

Compute \( r_i, i = 1, 2, \ldots, 6 \).
The Rod Cutting problem

The Problem

<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>
</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>
</tr>
</tbody>
</table>

Compute $r_i$, $i = 1, 2, \ldots, 6$. 

Dynamic Programming

Optimization Methods in Finance
The Rod Cutting problem

The Problem

Given a rod of \( n \) inches, and a table of prices \( p_i, i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces.
The Rod Cutting problem

The Problem

Given a rod of $n$ inches, and a table of prices $p_i$, $i = 1, 2, \ldots, n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling it into pieces. How many possibilities?
The Rod Cutting problem

The Problem

Given a rod of $n$ inches, and a table of prices $p_i$, $i = 1, 2, \ldots, n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling it into pieces. How many possibilities?

Example
The Rod Cutting problem

The Problem

Given a rod of \( n \) inches, and a table of prices \( p_i, i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces. How many possibilities?

Example

<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>
</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>
</tr>
</tbody>
</table>
The Rod Cutting problem

The Problem

Given a rod of \( n \) inches, and a table of prices \( p_i, i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces. How many possibilities?

Example

<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>
</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>
</tr>
</tbody>
</table>

Compute \( r_i, i = 1, 2, \ldots 6 \).
Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. This is called the optimal substructure property. Hence, we can write,

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

(1)

Unlike Divide-and-Conquer, the subproblems could overlap. Recurrence (1) can be expressed more succinctly as:

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

(2)

Why are Recurrence (1) and Recurrence (2) equivalent?
Optimal substructure property

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. This is called the optimal substructure property. Hence, we can write,

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

Unlike Divide-and-Conquer, the subproblems could overlap. Recurrence (1) can be expressed more succinctly as:

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

Recurrence (2)

Why are Recurrence (1) and Recurrence (2) equivalent?
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally.
Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)
Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property.
Observe that once the first cut is made, you get two independent subproblems which
must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

\[
\begin{align*}
\text{Recurrence} \\
\text{Observe that once the first cut is made, you get two independent subproblems which} \\
\text{must be solved optimally. (Why?)} \\
\text{This is called the optimal substructure property. Hence, we can write,}
\end{align*}
\]
Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

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

(1)
Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

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

Unlike Divide-and-Conquer, the subproblems could overlap.
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

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

(1)

Unlike Divide-and-Conquer, the subproblems could overlap.

Recurrence (1) can be expressed more succinctly as:

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

(2)
Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

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

Unlike Divide-and-Conquer, the subproblems could overlap.

Recurrence (1) can be expressed more succinctly as:

\[ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \] \hspace{1cm} (2)

\[ r_0 = 0 \]
Dynamic Programming

Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

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

(1)

Unlike Divide-and-Conquer, the subproblems could overlap.

Recurrence (1) can be expressed more succinctly as:

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

(2)

\[ r_0 = 0 \]

Why are Recurrence (1) and Recurrence (2) equivalent?
A recursive implementation

Algorithm 2.1: The recursive rod-cutting algorithm

Function $\text{CUT-ROD}(p, n)$

1: if ($n = 0$) then
2:     return (0).
3: end if
4: $q = -\infty$.
5: for ($i = 1$ to $n$) do
6:     $q = \max(q, p[i] + \text{CUT-ROD}(p, n-i))$.
7: end for

Analysis

$T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise}
\end{cases}$
A recursive implementation

Recursive Algorithm

<table>
<thead>
<tr>
<th>Function</th>
<th>( C_{UT-ROD}(p, n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>if ( n = 0 ) then</td>
</tr>
<tr>
<td>2:</td>
<td>return 0.</td>
</tr>
<tr>
<td>3:</td>
<td>end if</td>
</tr>
<tr>
<td>4:</td>
<td>( q = -\infty ).</td>
</tr>
<tr>
<td>5:</td>
<td>for ( i = 1 ) to ( n ) do</td>
</tr>
<tr>
<td>6:</td>
<td>( q = \max(q, p[i] + C_{UT-ROD}(p, n - i)) ).</td>
</tr>
<tr>
<td>7:</td>
<td>end for</td>
</tr>
</tbody>
</table>

Algorithm 2.2: The recursive rod-cutting algorithm

Analysis

\[ T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise} \end{cases} \]
## Recursive Algorithm

<table>
<thead>
<tr>
<th>Recursive Algorithm</th>
</tr>
</thead>
</table>

A recursive implementation

### Function

\[
\text{Recursive Algorithm}(p, n) \]

1. **if** \( n = 0 \) **then**
2. \( \text{return} (0). \)
3. **end if**
4. \( q = -\infty. \)
5. **for** \( i = 1 \) **to** \( n \) **do**
6. \( q = \max(q, p[i] + \text{Recursive Algorithm}(p, n-i)). \)
7. **end for**

### Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise}
\end{cases}
\]
Recursive Algorithm

Function $\text{CUT-ROD}(p, n)$

```plaintext
1: if ($n = 0$) then
2: return (0).
3: end if
4: $q = -\infty$.
5: for ($i = 1$ to $n$) do
6: $q = \max(q, p[i] + \text{CUT-ROD}(p, n-i))$.
7: end for

Algorithm 2.4: The recursive rod-cutting algorithm

Analysis

$T(n) =$

$\begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise} \end{cases}$
```
A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)
1: if (n = 0) then
Recursive Algorithm

**Function** \( \text{CUT-ROD}(p, n) \)

1. if \( (n = 0) \) then
2. return(0).
A recursive implementation

Recursive Algorithm

**Function** \( \text{CUT-ROD}(p, n) \)

1. **if** \((n = 0) \) **then**
2. \hspace{1em} **return** \((0)\).
3. **end if**


**Recursive Algorithm**

<table>
<thead>
<tr>
<th>Function</th>
<th><code>CUT-ROD(p, n)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td><em>if (n = 0) then</em></td>
</tr>
<tr>
<td>2:</td>
<td><em>return(0).</em></td>
</tr>
<tr>
<td>3:</td>
<td><em>end if</em></td>
</tr>
<tr>
<td>4:</td>
<td><em>q = −∞.</em></td>
</tr>
</tbody>
</table>

---

Dynamic Programming

A recursive implementation

---

Optimization Methods in Finance
A recursive implementation

Recursive Algorithm

Function \textsc{Cut-Rod}(p, n)

1: if \((n = 0)\) then
2: \hspace{1em} return(0).
3: end if
4: \(q = -\infty\).
5: for \((i = 1 \text{ to } n)\) do
A recursive implementation

Recursive Algorithm

Function `CUT-ROD(p, n)`

1: if \( n = 0 \) then
2:    return(0).
3: end if
4: \( q = -\infty \).
5: for \( i = 1 \) to \( n \) do
6:    \( q = \max(q, p[i] + \text{CUT-ROD}(p, n-i)) \).
7: end for

Algorithm 2.10: The recursive rod-cutting algorithm

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise}
\end{cases}
\]
A recursive implementation

Recursive Algorithm

Function \texttt{CUT-ROD}(p, n)

1: \textbf{if} \ (n = 0) \ \textbf{then}
2: \quad \textbf{return}(0).
3: \textbf{end if}
4: \quad q = -\infty.
5: \textbf{for} \ (i = 1 \ \textbf{to} \ n) \ \textbf{do}
6: \quad q = \text{max}(q, p[i] + \texttt{CUT-ROD}(p, n - i)).
A recursive implementation

**Recursive Algorithm**

```plaintext
Function CUT-ROD(p, n)
1: if (n = 0) then
2: return(0).
3: end if
4: q = −∞.
5: for (i = 1 to n) do
6: q = max(q, p[i] + CUT-ROD(p, n − i)).
7: end for
```

**Algorithm 2.12:** The recursive rod-cutting algorithm
A recursive implementation

Recursive Algorithm

Function \texttt{CUT-ROD}(p, n)

1: \textbf{if} \hspace{0.5em} (n = 0) \textbf{then}
2: \hspace{1em} return(0).
3: \textbf{end if}
4: \hspace{1em} q \leftarrow -\infty.
5: \hspace{1em} \textbf{for} \hspace{0.5em} (i = 1 \textbf{ to } n) \textbf{ do}
6: \hspace{1.5em} q \leftarrow \max(q, p[i] + \texttt{CUT-ROD}(p, n - i)).
7: \hspace{1em} \textbf{end for}

\textbf{Algorithm 2.13:} The recursive rod-cutting algorithm

Analysis
Dynamic Programming

A recursive implementation

Recursive Algorithm

Function `CUT-ROD(p, n)`

1: if \( n = 0 \) then
2:    return 0.
3: end if
4: \( q = -\infty \).
5: for \( i = 1 \) to \( n \) do
6:    \( q = \max(q, p[i] + CUT-ROD(p, n - i)) \).
7: end for

Algorithm 2.14: The recursive rod-cutting algorithm

Analysis

\[ T(n) = \]
A recursive implementation

Recursive Algorithm

Function `CUT-ROD(p, n)`

1: if \( n = 0 \) then
2:     return 0.
3: end if
4: \( q = -\infty \).  
5: for \( i = 1 \) to \( n \) do
6:     \( q = \max(q, p[i] + \text{CUT-ROD}(p, n - i)) \).
7: end for

Algorithm 2.15: The recursive rod-cutting algorithm

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 
\end{cases}
\]
A recursive implementation

Recursive Algorithm

Function CUT-ROD\((p, n)\)

1: if \((n = 0)\) then
2: return \((0)\).
3: end if
4: \(q = -\infty\).
5: for \((i = 1 \text{ to } n)\) do
6: \(q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))\).
7: end for

Algorithm 2.16: The recursive rod-cutting algorithm

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{j=1}^{n} T(n - j), & \text{otherwise}
\end{cases}
\]
Analysis of the recursive algorithm

Let

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise} 
\end{cases} \]

It is not hard to see that

\[ T(n) = 2^n. \]
Dynamic Programming

Analysis of the recursive algorithm

Analysis (contd.)

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise} 
\end{cases} \]

It is not hard to see that \( T(n) = 2^n \).
Analysis of the recursive algorithm

It is not hard to see that $T(n) = 2^n$. 

$T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise} 
\end{cases}$
Analysis of the recursive algorithm

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
T(k) + 1, & \text{otherwise}
\end{cases} \]

It is not hard to see that \( T(n) = 2^n \).
Analysis of the recursive algorithm

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases}
\]

It is not hard to see that \( T(n) = 2^n \).
Analysis (contd.)

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases} \]

It is not hard to see that \( T(n) = \)
Analysis (contd.)

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases}
\]

It is not hard to see that \( T(n) = 2^n \).
The Bottom-up approach

Algorithm 2.17: Bottom-up rod-cutting

1: Let \( r[0 \ldots n] \) be a new array.
2: \( r[0] = 0 \).
3: for \( j = 1 \) to \( n \) do
4:   \( q = -\infty \).
5:   for \( i = 1 \) to \( j \) do
6:     \( q = \max(q, p[i] + r[j-i]) \).
7:   end for
8:   \( r[j] = q \).
9: end for
10: return \( r[n] \).
The Bottom-up approach

The bottom-up algorithm

```
1: Let r[0···n] be a new array.
2: r[0] = 0.
3: for (j = 1 to n) do
4:     q = −∞.
5:     for (i = 1 to j) do
6:         q = max(q, p[i] + r[j−i])
7:     end for
8:     r[j] = q.
9: end for
10: return (r[n])
```

Algorithm 2.18: Bottom-up rod-cutting
The Bottom-up approach

The bottom-up algorithm

Function $\text{BOTTOM-ROD-CUT}(p, n)$

1: Let $r[0 \cdots n]$ be a new array.
2: $r[0] = 0.$
3: for $(j = 1$ to $n)$ do
4:   $q = -\infty.$
5:   for $(i = 1$ to $j)$ do
6:     $q = \max(q, p[i] + r[j-i]).$
7:   end for
8:   $r[j] = q.$
9: end for
10: return $(r[n]).$
The Bottom-up approach

The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let $r[0 \cdot n]$ be a new array.
The Bottom-up approach

The bottom-up algorithm

**Function** \( \text{BOTTOM-ROD-CUT}(p, n) \)

1. Let \( r[0 \ldots n] \) be a new array.
2. \( r[0] = 0. \)
The Bottom-up approach

The bottom-up algorithm

Function $\text{BOTTOM-ROD-CUT}(p, n)$

1: Let $r[0 \cdot n]$ be a new array.
2: $r[0] = 0.$
3: for $(j = 1 \text{ to } n)$ do

   4: $q = -\infty$

   5: for $(i = 1 \text{ to } j)$ do

      6: $q = \max(q, p[i] + r[j - i])$

   7: end for

   8: $r[j] = q$

9: end for

10: return $(r[n])$. 

Algorithm 2.22: Bottom-up rod-cutting
The bottom-up algorithm

Function `BOTTOM-ROD-CUT(p, n)`

1: Let \( r[0 \cdot n] \) be a new array.
2: \( r[0] = 0 \).
3: for \( (j = 1 \text{ to } n) \) do
4: \( q = -\infty \).
The Bottom-up approach

The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let `r[0 · · n]` be a new array.
2. `r[0] = 0`.
3. **for** `(j = 1 to n) do`
4. `q = −∞`.
5. **for** `(i = 1 to j) do`
The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let `r[0 · · · n]` be a new array.
2: `r[0] = 0`.
3: `for (j = 1 to n) do`
4: \( q = -\infty \).
5: `for (i = 1 to j) do`
6: \( q = \max(q, p[i] + r[j - i]) \).
The Bottom-up approach

The bottom-up algorithm

Function \textsc{Bottom-Rod-Cut}(p, n)

1: Let \( r[0 \cdot n] \) be a new array.
2: \( r[0] = 0. \)
3: for (\( j = 1 \) to \( n \)) do
4: \( q = -\infty. \)
5: for (\( i = 1 \) to \( j \)) do
6: \( q = \max(q, p[i] + r[j - i]). \)
7: end for
8: \( r[j] = q. \)
9: end for
10: return (\( r[n] \)).
The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let `r[0 · · · n]` be a new array.
2: `r[0] = 0`.
3: **for** `(j = 1 to n) do`
4: `q = −∞`.
5: **for** `(i = 1 to j) do`
6: `q = max(q, p[i] + r[j − i])`.
7: **end for**
8: `r[j] = q`.

**Algorithm 2.27:** Bottom-up rod-cutting
The bottom-up algorithm

Function `BOTTOM-ROD-CUT(p, n)`

1. Let $r[0 \cdot n]$ be a new array.
2. $r[0] = 0$.
3. for $(j = 1$ to $n)$ do
4.     $q = -\infty$.
5.     for $(i = 1$ to $j)$ do
6.         $q = \max(q, p[i] + r[j - i])$.
7.     end for
8.     $r[j] = q$.
9. end for
The Bottom-up approach

The bottom-up algorithm

```plaintext
Function BOTTOM-ROD-CUT(p, n)
1: Let r[0 · · n] be a new array.
2: r[0] = 0.
3: for (j = 1 to n) do
4:   q = −∞.
5:   for (i = 1 to j) do
6:     q = max(q, p[i] + r[j − i]).
7:   end for
8:   r[j] = q.
9: end for
10: return(r[n]).
```

Algorithm 2.29: Bottom-up rod-cutting
Analyzing the bottom-up approach

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly, 

\[ T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise} 
\end{cases} \]

It is not hard to see that \( T(n) = \Theta(n^2) \).
Analyzing the bottom-up approach

Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed. Accordingly, 

\[ T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise} 
\end{cases} \]

It is not hard to see that \( T(n) = \Theta(n^2) \).
Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.
The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

\[ T(n) = \sum_{j=1}^{n} \sum_{i=1}^{j} \]
The running time of the algorithm can be approximated by the number of times that Line (6) is executed. Accordingly,

\[ T(n) = \begin{cases} 
0, & \text{if } n = 0 
\end{cases} \]
The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

$$ T(n) = \begin{cases} 0, & \text{if } n = 0 \\ \sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise} \end{cases} $$
Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

\[ T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise}
\end{cases} \]

It is not hard to see that \( T(n) = \)
The running time of the algorithm can be approximated by the number of times that Line (6) is executed. Accordingly,

\[
T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise}
\end{cases}
\]

It is not hard to see that \( T(n) = \Theta(n^2) \).
Reconstructing the Solution

Dynamic Programming

Algorithm 2.30: Bottom-up rod-cutting

```
1: Let \( r[0 \cdots n] \) and \( s[0 \cdots n] \) be new arrays.
2: \( r[0] = 0. \)
3: for \( (j = 1 \text{ to } n) \)
   4: \( q = -\infty. \)
   5: for \( (i = 1 \text{ to } j) \)
      6: if \( (q < p[i] + r[j-i]) \)
         7: \( q = p[i] + r[j-i]. \)
         8: \( s[j] = i. \) \{The unsplittable left side is recorded.\}
      9: end if
   10: end for
   11: \( r[j] = q. \)
12: end for
13: return \( r[n] \).
```

Optimization Methods in Finance
Function \( B_{\text{OTTOM-ROD-CUT}}(p, n) \)

1: Let \( r[0 \ldots n] \) and \( s[0 \ldots n] \) be new arrays.

2: \( r[0] = 0. \)

3: for \( (j = 1 \ldots n) \) do

4: \( q = -\infty. \)

5: for \( (i = 1 \ldots j) \) do

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

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

8: \( s[j] = i. \) \{The unsplittable left side is recorded.\}

9: end if

10: end for

11: \( r[j] = q. \)

12: end for

13: return \( (r[n]). \)

Algorithm 2.31: Bottom-up rod-cutting
The bottom-up algorithm with solution

**Function** BOTTOM-ROD-CUT$(p, n)$

1: Let $r[0 \cdots n]$ and $s[0 \cdots n]$ be new arrays.
2: $r[0] = 0$.
3: for $(j = 1$ to $n)$ do
4:   $q = -\infty$.
5:   for $(i = 1$ to $j)$ do
6:     if $(q < p[i] + r[j - i])$ then
7:       $q = p[i] + r[j - i]$.
8:       $s[j] = i$. \{The unsplittable left side is recorded.\}
9:     end if
10:   end for
11:   $r[j] = q$.
12: end for
13: return $(r[n])$.

Algorithm 2.32: Bottom-up rod-cutting
Reconstructing the Solution

The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let \( r[0 \cdot n] \) and \( s[0 \cdot n] \) be new arrays.
Dynamic Programming

Reconstructing the Solution

The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let \( r[0 \ldots n] \) and \( s[0 \ldots n] \) be new arrays.
2. \( r[0] = 0 \).
The bottom-up algorithm with solution

**Function** \( \text{BOTTOM-ROD-CUT}(p, n) \)

1. Let \( r[0 \cdot n] \) and \( s[0 \cdot n] \) be new arrays.
2. \( r[0] = 0. \)
3. \( \textbf{for } (j = 1 \textbf{ to } n) \textbf{ do} \)
   
   4. \( q = -\infty. \)
   5. \( \textbf{for } (i = 1 \textbf{ to } j) \textbf{ do} \)
      
      6. \( \textbf{if } (q < p[i] + r[j - i]) \textbf{ then} \)
         
         7. \( q = p[i] + r[j - i]. \)
         
         8. \( s[j] = i. \) \{The unsplittable left side is recorded.\}
   9. \( \textbf{end if} \)
   10. \( \textbf{end for} \)
   11. \( r[j] = q. \)
   12. \( \textbf{end for} \)
13. \( \textbf{return } (r[n]). \)
Reconstructing the Solution

The bottom-up algorithm with solution

**Function** Bottom-Rod-Cut\((p, n)\)

1. Let \(r[0 \cdot n]\) and \(s[0 \cdot n]\) be new arrays.
2. \(r[0] = 0\).
3. **for** \((j = 1\) to \(n)\) **do**
   4. \(q = -\infty\).
Reconstructing the Solution

The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let `r[0 \cdot n]` and `s[0 \cdot n]` be new arrays.
2: `r[0] = 0`.
3: **for** `j = 1` **to** `n` **do**
4: \( q = -\infty \).
5: **for** `i = 1` **to** `j` **do**
Dynamic Programming

Reconstructing the Solution

The bottom-up algorithm with solution

Function \textsc{Bottom-Rod-Cut}(p, n)

1: Let $r[0 \cdots n]$ and $s[0 \cdots n]$ be new arrays.
2: $r[0] = 0$.
3: for $(j = 1$ to $n)$ do
4: \hspace{1em} $q = -\infty$.
5: \hspace{1em} for $(i = 1$ to $j)$ do
6: \hspace{2em} if ($q < p[i] + r[j - i]$) then
7: \hspace{3em} $q = p[i] + r[j - i]$.
8: \hspace{2em} $s[j] = i$.
9: end if
10: end for
11: $r[j] = q$.
12: end for
13: return $(r[n])$. 

Algorithm 2.38: Bottom-up rod-cutting
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let `r[0 · · · n]` and `s[0 · · · n]` be new arrays.
2. `r[0] = 0`.
3. **for** `(j = 1 to n)` **do**
4. `q = −∞`.
5. **for** `(i = 1 to j)` **do**
6. `if` `(q < p[i] + r[j − i])` **then**
7. `q = p[i] + r[j − i]`.
8. `s[j] = i`.
9. **end if**
10. **end for**
11. `r[j] = q`.
12. **end for**
13. **return** `r[n]`.

Algorithm 2.39: Bottom-up rod-cutting
The bottom-up algorithm with solution

**Function** \textsc{Bottom-Rod-Cut}(p, n)

1: Let \(r[0 \cdot n]\) and \(s[0 \cdot n]\) be new arrays.
2: \(r[0] = 0\).
3: \textbf{for} \ (j = 1 \textbf{to} n) \textbf{do}
4: \(q = -\infty\).
5: \textbf{for} \ (i = 1 \textbf{to} j) \textbf{do}
6: \textbf{if} \ (q < p[i] + r[j - i]) \textbf{then}
7: \(q = p[i] + r[j - i]\).
8: \(s[j] = i\). \{\text{The unsplittable left side is recorded.}\}
The bottom-up algorithm with solution

Function \textsc{Bottom-Rod-Cut}(p, n)

1: Let $r[0 \cdots n]$ and $s[0 \cdots n]$ be new arrays.
2: $r[0] = 0$.
3: \textbf{for} ($j = 1$ \textbf{to} $n$) \textbf{do}
4: \hspace{1em} $q = -\infty$.
5: \hspace{1em} \textbf{for} ($i = 1$ \textbf{to} $j$) \textbf{do}
6: \hspace{2em} \textbf{if} ($q < p[i] + r[j - i]$) \textbf{then}
7: \hspace{3em} $q = p[i] + r[j - i]$.
8: \hspace{3em} $s[j] = i$. \{The unsplittable left side is recorded.\}
9: \hspace{2em} \textbf{end if}
10: \hspace{1em} \textbf{end for}
11: $r[j] = q$.
12: \textbf{end for}
13: \textbf{return} $r[n]$.
The bottom-up algorithm with solution

Function $\text{BOTTOM-ROD-CUT}(p, n)$

1: Let $r[0 \cdots n]$ and $s[0 \cdots n]$ be new arrays.
2: $r[0] = 0$.
3: for $(j = 1$ to $n)$ do
4:   $q = -\infty$.
5:   for $(i = 1$ to $j)$ do
6:     if $(q < p[i] + r[j - i])$ then
7:       $q = p[i] + r[j - i]$.
8:       $s[j] = i$. {The unsplittable left side is recorded.}
9:     end if
10:   end for
11: $r[j] = q$.
12: end for
13: return $(r[n])$. 

Algorithm 2.42: Bottom-up rod-cutting
Reconstructing the Solution

The bottom-up algorithm with solution

```
Function BOTTOM-ROD-CUT(p, n)
1: Let r[0 · · · n] and s[0 · · · n] be new arrays.
2: r[0] = 0.
3: for (j = 1 to n) do
4:     q = −∞.
5:     for (i = 1 to j) do
6:         if (q < p[i] + r[j − i]) then
7:             q = p[i] + r[j − i].
8:             s[j] = i. {The unsplittable left side is recorded.}
9:       end if
10:     end for
11: r[j] = q.
```

The bottom-up algorithm with solution

Function \textsc{Bottom-Rod-Cut}(p, n)

1: Let \( r[0 \cdot n] \) and \( s[0 \cdot n] \) be new arrays.
2: \( r[0] = 0. \)
3: for (\( j = 1 \) to \( n \)) do
4: \( q = -\infty. \)
5: for (\( i = 1 \) to \( j \)) do
6: if (\( q < p[i] + r[j - i] \)) then
7: \( q = p[i] + r[j - i]. \)
8: \( s[j] = i. \) \{The unsplittable left side is recorded.\}
9: end if
10: end for
11: \( r[j] = q. \)
12: end for

Return \( r[n] \).
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let `r[0 · n]` and `s[0 · n]` be new arrays.
2. `r[0] = 0`.
3. **for** `(j = 1` **to** `n)` **do**
4.   `q = −∞`.
5.   **for** `(i = 1` **to** `j)` **do**
6.     **if** `(q < p[i] + r[j − i])` **then**
7.       `q = p[i] + r[j − i]`.
8.       `s[j] = i`. **{The unsplittable left side is recorded.}**
9.     **end if**
10.   **end for**
11. `r[j] = q`.
12. **end for**

**Algorithm 2.45:** Bottom-up rod-cutting
Outputting the solution

Algorithm 2.46: Extracting the solution

```plaintext
FUNCTION PRINT-SOLUTION(p, n)
1: while (n > 0) do
2:   print s[n].
3:   n = n - s[n].
4: end while
```

Dynamic Programming
Optimization Methods in Finance
Dynamic Programming

Outputting the solution

Algorithm 2.47: Extracting the solution

Printing the Solution

Function \( \text{PRINT}\)-\text{SOLUTION}(p, n)

1: while \( n > 0 \) do
2:    print \( s[n] \).
3:    \( n = n - s[n] \).
4: end while
Outputting the solution

Printing the Solution

**Function** PRINT-SOLUTION(p, n)
Outputting the solution

Printing the Solution

Function PRINT-SOLUTION($p, n$)
1: while ($n > 0$) do
**Function** PRINT-SOLUTION\((p, n)\)

1. while \((n > 0)\) do
2. print \(s[n]\).
Printing the Solution

Function PRINT-SOLUTION\((p, n)\)
1: while \((n > 0)\) do
2: \quad print \(s[n]\).
3: \quad n = n - s[n].
Algorithm 2.52: Extracting the solution

Function PRINT-SOLUTION(p, n)
1: while (n > 0) do
2:   print s[n].
3:   n = n − s[n].
4: end while
The Matrix Chain Multiplication problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_i-1 \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 
1, & \text{if } n = 2 \\
\sum_{k=1}^{n-1} T(k) \cdot T(n-k), & \text{otherwise} 
\end{cases}$$

Solving the recurrence gives the $n$th Catalan number whose growth is $\Omega(4^n/n^{3/2})$. 
The Matrix Chain Multiplication problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.

2. The entries in the matrices do not affect the optimum solution.

Solving the recurrence gives the $n$th Catalan number whose growth is $\Omega(4^n / n^{3/2})$. 
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, where

1. The total number of scalar multiplications when multiplying two matrices of $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders $T(n) = \begin{cases} 1, & \text{if } n = 2 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n-k), & \text{otherwise} \end{cases}$

Solving the recurrence gives the $n$th Catalan number whose growth is $\Omega(4^n n^{3/2})$. 
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,
The Problem

You are required to compute the matrix product \( A_1 \cdot A_2 \cdots A_n \), where matrix \( A_i \) has dimensions \( d_{i-1} \times d_i \), while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions \( p \times q \) and \( q \times r \) is \( p \cdot q \cdot r \).
The Matrix Chain Multiplication problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.
The Matrix Chain Multiplication problem

The Problem
You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) =$$
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 1, & \text{if } n = 2 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n-k), & \text{otherwise} \end{cases}$$
The Problem
You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 1, & \text{if } n = 2 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n-k), & \text{otherwise} \end{cases}$$
The Matrix Chain Multiplication problem

The Problem
You are required to compute the matrix product $A_1 \cdot A_2 \cdot \cdots \cdot A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 1, & \text{if } n = 2 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n - k), & \text{otherwise} \end{cases}$$

Solving the recurrence gives the $n^{th}$ Catalan number whose growth is $\Omega\left(\frac{4^n}{n^2}\right)$. 
Optimality Substructure

If somebody gave you the first grouping, can the problem be simplified? Yes!
The two subproblems that result must be solved optimally.

Therefore, the optimality substructure applies.

Let \( m[i, j] \) denote the optimal number of scalar multiplications to multiply the matrices \( \langle A_i, A_{i+1}, \ldots, A_j \rangle \).

\[
\begin{align*}
m[i, j] &= 0, \quad \text{if } j = i \\
&= \min_{i \leq k < j} (m[i, k] + m[k+1, j] + d_{i-1} \cdot d_k \cdot d_{j-1}), \quad \text{if } j > i.
\end{align*}
\]
If somebody gave you the first grouping, can the problem be simplified? Yes! The two subproblems that result must be solved optimally. Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.
Optimality Substructure

If somebody gave you the first grouping, can the problem be simplified?
Optimality Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes!
Substructure

If somebody gave you the **first** grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally.
If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)
Optimality Substructure

**Substructure**

If somebody gave you the **first** grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$. 
If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let \( m[i, j] \) denote the optimal number of scalar multiplications to multiply the matrices \( \langle A_i, A_{i+1}, \ldots A_j \rangle \).

\[
m[i, j] = \]

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

$$m[i, j] = \begin{cases} 0, \\ \end{cases}$$
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

$$m[i, j] = \begin{cases} 0, & \text{if } j = i \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + d_{i-1} \cdot d_k \cdot d_{j-1}), & \text{if } j > i \end{cases}$$
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

\[
m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j]), & \text{if } j > i
\end{cases}
\]
Optimality Substructure

If somebody gave you the first grouping, can the problem be simplified?
Yes! The two subproblems that result must be solved optimally. (Why?)
Therefore, the optimality substructure applies.

Let \( m[i, j] \) denote the optimal number of scalar multiplications to multiply the matrices \( \langle A_i, A_{i+1}, \ldots A_j \rangle \).

\[
m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j] + d_{i-1} \cdot d_k \cdot d_j), & \text{if } j > i.
\end{cases}
\]
Optimality Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

$$m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j] + d_{i-1} \cdot d_k \cdot d_j), & \text{if } j > i.
\end{cases}$$
For space usage, observe that we need an array $m[i,j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i,j]$.

Example: Find the optimal parenthesization for the chain $\langle A_7 \times 10 \cdot B_{10} \times 3 \cdot C_3 \times 8 \cdot D_8 \times 4 \rangle$. 
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$.

Example: Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. 

Dynamic Programming
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$. Can you show that the time required is $\Theta(n^3)$?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$. Example: Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. 
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$. 

Can you show that the time required is $\Theta(n^3)$?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$. 

Example: Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. 
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time.
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.
Dynamic Programming

Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

We have left out some details in the algorithm;
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note

*We have left out some details in the algorithm; such as extracting the optimal solution.*
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$. Can you show that the time required is $\Theta(n^3)$?

We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem;
Resource analysis

**Analysis**

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

**Note**

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$. 
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$. 

Example
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$.

Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. 
You are given $n$ objects $O = \{o_1, o_2, \ldots, o_n\}$.

Object $o_i$ has weight $w_i$ and profit $p_i$.

You are also given a knapsack of weight capacity $W$.

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.

The integer programming formulation is:

\[
\begin{align*}
\text{max} & \quad \sum_{i=1}^{n} p_i \cdot x_i \\
\text{subject to} & \quad \sum_{i=1}^{n} w_i \cdot x_i \leq W \\
& \quad x_i \in \{0, 1\} \quad \forall i = 1, 2, \ldots, n
\end{align*}
\]
You are given \( n \) objects \( O = \{o_1, o_2, \ldots, o_n\} \).

Object \( o_i \) has weight \( w_i \) and profit \( p_i \).

You are also given a knapsack of weight capacity \( W \).

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.

The integer programming formulation is:

\[
\begin{align*}
\text{max} & \quad \sum_{i=1}^{n} p_i \cdot x_i \\
\text{subject to} & \quad \sum_{i=1}^{n} w_i \cdot x_i \leq W \\
& \quad x_i = \{0, 1\} \quad \forall i = 1, 2, \ldots, n
\end{align*}
\]
You are given \( n \) objects \( O = \{ o_1, o_2, \ldots, o_n \} \).
You are given $n$ objects $O = \{o_1, o_2, \ldots, o_n\}$.

Object $o_i$ has weight $w_i$ and profit $p_i$. 
You are given \( n \) objects \( O = \{ o_1, o_2, \ldots, o_n \} \).

Object \( o_i \) has weight \( w_i \) and profit \( p_i \).

You are also given a knapsack of weight capacity \( W \).
You are given \( n \) objects \( O = \{ o_1, o_2, \ldots, o_n \} \). Object \( o_i \) has weight \( w_i \) and profit \( p_i \). You are also given a knapsack of weight capacity \( W \). The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
You are given \( n \) objects \( O = \{ o_1, o_2, \ldots, o_n \} \).

Object \( o_i \) has weight \( w_i \) and profit \( p_i \).

You are also given a knapsack of weight capacity \( W \).

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.
You are given \( n \) objects \( O = \{ o_1, o_2, \ldots, o_n \} \).

Object \( o_i \) has weight \( w_i \) and profit \( p_i \).

You are also given a knapsack of weight capacity \( W \).

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.

The integer programming formulation is:
You are given $n$ objects $O = \{o_1, o_2, \ldots, o_n\}$.

Object $o_i$ has weight $w_i$ and profit $p_i$.

You are also given a knapsack of weight capacity $W$.

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.

The integer programming formulation is:

$$\max \sum_{i=1}^{n} p_i \cdot x_i$$
You are given $n$ objects $O = \{o_1, o_2, \ldots, o_n\}$.

Object $o_i$ has weight $w_i$ and profit $p_i$.

You are also given a knapsack of weight capacity $W$.

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.

The integer programming formulation is:

$$\max \sum_{i=1}^{n} p_i \cdot x_i$$

subject to

$$\sum_{i=1}^{n} w_i \cdot x_i \leq W$$

and $x_i \in \{0, 1\}$ for all $i = 1, 2, \ldots, n$. 

Rewritten in a more readable format:

1. You are given $n$ objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object $o_i$ has weight $w_i$ and profit $p_i$.
3. You are also given a knapsack of weight capacity $W$.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
5. Profits are additive.
6. The integer programming formulation is:

$$\max \sum_{i=1}^{n} p_i \cdot x_i$$

subject to

$$\sum_{i=1}^{n} w_i \cdot x_i \leq W$$

and $x_i \in \{0, 1\}$ for all $i = 1, 2, \ldots, n$. 

You are given \( n \) objects \( O = \{ o_1, o_2, \ldots, o_n \} \).

Object \( o_i \) has weight \( w_i \) and profit \( p_i \).

You are also given a knapsack of weight capacity \( W \).

The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Profits are additive.

The integer programming formulation is:

\[
\text{max} \quad \sum_{i=1}^{n} p_i \cdot x_i \\
\sum_{i=1}^{n} w_i \cdot x_i \leq W \\
x_i = \{0, 1\} \quad \forall i = 1, 2, \ldots, n
\]
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{K}_\text{NAP}(n, W)$ denote the given instance of the problem.

2. Let $S \subseteq O$ denote the optimal solution.

3. Focus on object $o_n$.

4. Either $o_n \in S$ or $o_n \notin S$.

5. If $o_n \in S$, then $S \setminus \{o_n\}$ must constitute an optimal solution for $\text{K}_\text{NAP}(n-1, W-w_n)$.

6. If $o_n \notin S$, then $S$ must be an optimal solution for $\text{K}_\text{NAP}(n-1, W)$.

(Why?)
Let $K_{\text{NAP}}(n, W)$ denote the given instance of the problem.

Let $S \subseteq O$ denote the optimal solution.

Focus on object $o_n$.

Either $o_n \in S$ or $o_n \notin S$.

If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $K_{\text{NAP}}(n-1, W - w_n)$.

If $o_n \notin S$, then $S$ must be an optimal solution for $K_{\text{NAP}}(n-1, W)$.

Principle of optimality
Let $K\text{nap}(n, W)$ denote the given instance of the problem.
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{Knap}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
Dynamic Programming

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$. 

(Why?)
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \notin S$. 
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \not\in S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \not\in S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $\text{KNAP}(n - 1, W - w_n)$.

Dynamic Programming
Optimization Methods in Finance
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $\text{KNAP}(n - 1, W - w_n)$. (Why?)
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \not\in S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $\text{KNAP}(n - 1, W - w_n)$. (Why?)
6. If $o_n \not\in S$, then $S$ must be an optimal solution for
A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \not\in S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $\text{KNAP}(n - 1, W - w_n)$. (Why?)
6. If $o_n \not\in S$, then $S$ must be an optimal solution for $\text{KNAP}(n - 1,$
A DP-based algorithm for binary knapsack

**Principle of optimality**

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $\text{KNAP}(n - 1, W - w_n)$. (Why?)
6. If $o_n \notin S$, then $S$ must be an optimal solution for $\text{KNAP}(n - 1, W)$. 
A DP-based algorithm for binary knapsack

**Principle of optimality**

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object $o_n$.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ must constitute an optimal solution for $\text{KNAP}(n - 1, W - w_n)$. (Why?)
6. If $o_n \notin S$, then $S$ must be an optimal solution for $\text{KNAP}(n - 1, W)$. (Why?)
Formulating the recurrence

Let $V[i, w]$ denote the optimal solution for the subset \{o_1, o_2, \ldots, o_i\}, assuming that the Knapsack has a capacity $w$.

Which entry of the table are we interested in? Clearly, $V[n, W]$.

As per the discussion above, $V[i, w] = \max\{V[i-1, w-w_i] + p_i (o_i is included), V[i-1, w] (o_i is excluded)\}$.

Initial conditions: $V[0, w] = 0$, $0 \leq w \leq W$.

$V[i, w] = -\infty$, $w < 0$. 

Optimization Methods in Finance
The Recurrence

Let $V[i, w]$ denote the optimal solution for the subset \{o_1, o_2, \ldots, o_i\}, assuming that the Knapsack has a capacity $w$.

Which entry of the table are we interested in? Clearly, $V[n, W]$.

As per the discussion above, $V[i, w] = \max\{V[i-1, w-w_i] + p_i (o_i is included), V[i-1, w] (o_i is excluded)\}$.

Initial conditions: $V[0, w] = 0, 0 \leq w \leq W$.

$V[i, w] = -\infty, w < 0$. 
Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity $w$. 
The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in?
The Recurrence

1. Let \( V[i, w] \) denote the optimal solution for the subset \( \{o_1, o_2, \ldots, o_i\} \), assuming that the Knapsack has a capacity \( w \).

2. Which entry of the table are we interested in? Clearly, \( V[n, W] \).
Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset \{\(o_1, o_2, \ldots, o_i\}\), assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

3. As per the discussion above,
Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

3. As per the discussion above,

$$V[i, w] = \max \begin{cases} 
V[i-1, w-v_i] + p_i (o_i \text{ is included}) \\
V[i-1, w] (o_i \text{ is excluded}) 
\end{cases}$$
The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

3. As per the discussion above,

$$V[i, w] = \max \left\{ V[i-1, w-w_i] + p_i \right\}$$
Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset \{o_1, o_2, \ldots, o_i\}, assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

3. As per the discussion above,

$$V[i, w] = \max \left\{ V[i-1, w-w_i] + p_i \quad (o_i \text{ is included}) \right\}$$
Formulating the recurrence

The Recurrence

1. Let \( V[i, w] \) denote the optimal solution for the subset \( \{o_1, o_2, \ldots, o_i\} \), assuming that the Knapsack has a capacity \( w \).

2. Which entry of the table are we interested in? Clearly, \( V[n, W] \).

3. As per the discussion above,

\[
V[i, w] = \max \begin{cases} V[i - 1, w - w_i] + p_i & (o_i \text{ is included}) \\ V[i - 1, w] & \end{cases}
\]
The Recurrence

1. Let \( V[i, w] \) denote the optimal solution for the subset \( \{o_1, o_2, \ldots, o_i\} \), assuming that the Knapsack has a capacity \( w \).

2. Which entry of the table are we interested in? Clearly, \( V[n, W] \).

3. As per the discussion above,

\[
V[i, w] = \max \begin{cases} 
V[i - 1, w - w_i] + p_i \quad (o_i \text{ is included}) \\
V[i - 1, w] \quad (o_i \text{ is excluded})
\end{cases}
\]
Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

3. As per the discussion above,

$$
V[i, w] = \max \begin{cases} 
V[i - 1, w - w_i] + p_i & (o_i \text{ is included}) \\
V[i - 1, w] & (o_i \text{ is excluded})
\end{cases}
$$

4. Initial conditions:
Formulating the recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset \( \{o_1, o_2, \ldots, o_i\} \), assuming that the Knapsack has a capacity $w$.

2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

3. As per the discussion above,

\[
V[i, w] = \max \left\{ V[i-1, w-w_i] + p_i \quad (o_i \text{ is included}) \right. \\
\left. V[i-1, w] \quad (o_i \text{ is excluded}) \right\}
\]

4. Initial conditions:

\[
V[0, w] = 0, \quad 0 \leq w \leq W
\]
The Recurrence

1. Let \( V[i, w] \) denote the optimal solution for the subset \( \{o_1, o_2, \ldots, o_i\} \), assuming that the Knapsack has a capacity \( w \).

2. Which entry of the table are we interested in? Clearly, \( V[n, W] \).

3. As per the discussion above,

\[
V[i, w] = \max \begin{cases} 
V[i - 1, w - w_i] + p_i & (o_i \text{ is included}) \\
V[i - 1, w] & (o_i \text{ is excluded})
\end{cases}
\]

4. Initial conditions:

\[
\begin{align*}
V[0, w] &= 0, & 0 \leq w \leq W \\
V[i, w] &= -\infty, & w < 0
\end{align*}
\]
Dynamic Programming

Example

Exercise

Solve the following instance of Knapsack:

\[ n = 4, \quad w = \langle 5, 4, 6, 3 \rangle, \quad W = 10, \quad p = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th></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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i=1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i=2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>i=3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>i=4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

Dynamic Programming

Optimization Methods in Finance
**Example**

**Exercise**

Solve the following instance of Knapsack:

- \( n = 4 \)
- \( w = \langle 5, 4, 6, 3 \rangle \)
- \( W = 10 \)
- \( p = \langle 10, 40, 30, 50 \rangle \)

**Solution**

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( i = 1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( i = 2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>( i = 3 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>( i = 4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

<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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>
Solve the following instance of Knapsack:

\( n = 4 \),
Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \]
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \; \mathbf{w} = \langle 5, 4, 6, 3 \rangle, \; W = 10, \]
Exercise

Solve the following instance of Knapsack:

\( n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \)
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \]
Exercise

Solve the following instance of Knapsack:

$n = 4$, \( \mathbf{w} = \langle 5, 4, 6, 3 \rangle \), \( W = 10 \), \( \mathbf{p} = \langle 10, 40, 30, 50 \rangle \).

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \mathbf{w} = \langle 5, 4, 6, 3 \rangle, W = 10, \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \]

Solution

\[
V[i, w] \quad | \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \\
\hline
i = 0 \quad | \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \\
\]

Dynamic Programming

Optimization Methods in Finance
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \mathbf{w} = \langle 5, 4, 6, 3 \rangle, \ W = 10, \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \]

Solution

\[
\begin{array}{cccccccccccc}
V[i, w] & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
i = 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
i = 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \; w = \langle 5, 4, 6, 3 \rangle, \; W = 10, \; p = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \]

Solution

\[
\begin{array}{|c|cccccccccc|}
\hline
V[i, w] & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
i = 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
i = 1 & 0 & 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 10 & 10 \\
i = 2 & 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
\hline
\end{array}
\]
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \mathbf{w} = \langle 5, 4, 6, 3 \rangle, W = 10, \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \]

Solution

\[
\begin{array}{c|cccccccccc}
V[i, w] & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
i = 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 10 & 10 \\
2 & 0 & 0 & 0 & 0 & 40 & 40 & 40 & 40 & 40 & 50 & 50 \\
\end{array}
\]
Example

Exercise

Solve the following instance of Knapsack:

1 \( n = 4, \; \mathbf{w} = \langle 5, 4, 6, 3 \rangle, \; W = 10, \; \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \)

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \mathbf{w} = \langle 5, 4, 6, 3 \rangle, W = 10, \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Exercise

Solve the following instance of Knapsack:

\( n = 4, \quad \mathbf{w} = \langle 5, 4, 6, 3 \rangle, \quad W = 10, \quad \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \)

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>70</td>
</tr>
</tbody>
</table>
Solve the following instance of Knapsack:

\[ n = 4, \mathbf{w} = \langle 5, 4, 6, 3 \rangle, W = 10, \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \]

\[ \]

**Solution**

\[
\begin{array}{c|ccccccccccc}
V[i, w] & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
i = 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 10 & 10 \\
2 & 0 & 0 & 0 & 0 & 40 & 40 & 40 & 40 & 40 & 50 & 50 \\
3 & 0 & 0 & 0 & 0 & 40 & 40 & 40 & 40 & 40 & 50 & 70 \\
4 & & & & & & & & & & & \\
\end{array}
\]
**Exercise**

Solve the following instance of Knapsack:

\[ n = 4, \mathbf{w} = \langle 5, 4, 6, 3 \rangle, W = 10, \mathbf{p} = \langle 10, 40, 30, 50 \rangle. \]

**Solution**

\[
\begin{array}{|c|cccccccccc|}
\hline
i \quad | \quad V[i, w] & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
0 \quad | \quad 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 \quad | \quad 0 & 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
2 \quad | \quad 0 & 0 & 0 & 0 & 40 & 40 & 40 & 40 & 40 & 50 & 50 & 50 \\
3 \quad | \quad 0 & 0 & 0 & 0 & 40 & 40 & 40 & 40 & 40 & 50 & 70 & \\
4 \quad | \quad 0 & 0 & 0 & 0 & \hline
\end{array}
\]
Exercise

Solve the following instance of Knapsack:

\[ n = 4, \ w = \langle 5, 4, 6, 3 \rangle, \ W = 10, \ p = \langle 10, 40, 30, 50 \rangle. \]

Solution

<table>
<thead>
<tr>
<th>( V[i, w] )</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

You have 14K to invest in three possible investments.

Investment I₁ requires an investment of 7K and a profit of 11K.

Investment I₂ requires an investment of 5K and a profit of 8K.

Investment I₃ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation

Let $x_i$, ($i=1, 2, 3$) be 1 if Investment $I_i$ is selected and 0 otherwise.

Accordingly, we have,

$$\text{max } 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3$$

$$7 \cdot x_1 + 5 \cdot x_2 + 4 \cdot x_3 \leq 14$$

$$x_i = \{0, 1\} \quad \forall i = 1, 2, 3$$
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

You have 14K to invest in three possible investments.

Investment $I_1$ requires an investment of 7K and a profit of 11K.

Investment $I_2$ requires an investment of 5K and a profit of 8K.

Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

**Knapsack formulation**

Let $x_i$, ($i = 1, 2, 3$) be 1 if Investment $I_i$ is selected and 0 otherwise.

Accordingly, we have:

$$\text{max } 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3 - 7 \cdot x_1 - 5 \cdot x_2 - 4 \cdot x_3 \leq 14$$

$x_i = \{0, 1\} \ \forall \ i = 1, 2, 3$
Example

Consider the following portfolio optimization problem:

You have 14K to invest in three possible investments.
Investment \text{I}_1 \text{ requires an investment of } 7K \text{ and a profit of } 11K.
Investment \text{I}_2 \text{ requires an investment of } 5K \text{ and a profit of } 8K.
Investment \text{I}_3 \text{ requires an investment of } 4K \text{ and a profit of } 6K.
How do you distribute your money among the three investments to maximize profits?

Knapsack formulation
Let \( x_i \), (\( i = 1, 2, 3 \)) be 1 if Investment \text{I}_i \text{ is selected and 0 otherwise. Accordingly, we have,}

\[
\text{max } 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3 \\
7 \cdot x_1 + 5 \cdot x_2 + 4 \cdot x_3 \leq 14
\]
\[
x_i \in \{0, 1\} \quad \forall \quad i = 1, 2, 3
\]
Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $l_1$ requires an investment of 7K and a profit of 11K.
3. Investment $l_2$ requires an investment of 5K and a profit of 8K.
4. Investment $l_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?
Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
4. Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
4. Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation

Let $x_i$, $(i = 1, 2, 3)$ be 1 if Investment $I_i$ is selected and 0 otherwise.
Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
4. Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation

Let $x_i$, ($i = 1, 2, 3$) be 1 if Investment $I_i$ is selected and 0 otherwise.

Accordingly, we have,
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
4. Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation

Let $x_i$, ($i = 1, 2, 3$) be 1 if Investment $I_i$ is selected and 0 otherwise.

Accordingly, we have,

$$\max \quad 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3$$
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
4. Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation

Let $x_i$, $(i = 1, 2, 3)$ be 1 if Investment $I_i$ is selected and 0 otherwise.

Accordingly, we have,

$$
\max \quad 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3 \\
7 \cdot x_1 + 5 \cdot x_2 + 4 \cdot x_3 \leq 14
$$
A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

1. You have 14K to invest in three possible investments.
2. Investment $I_1$ requires an investment of 7K and a profit of 11K.
3. Investment $I_2$ requires an investment of 5K and a profit of 8K.
4. Investment $I_3$ requires an investment of 4K and a profit of 6K.

How do you distribute your money among the three investments to maximize profits?

Knapsack formulation

Let $x_i$, $(i = 1, 2, 3)$ be 1 if Investment $I_i$ is selected and 0 otherwise.

Accordingly, we have,

$$\max \quad 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3$$
$$7 \cdot x_1 + 5 \cdot x_2 + 4 \cdot x_3 \leq 14$$
$$x_i = \{0, 1\} \quad \forall i = 1, 2, 3$$