Dynamic Programming - Theory and Applications

K. Subramani[1]

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

March 17, 2015

Outline

Dynamic Programming

Dynamic Programming

Main ideas

Dynamic Programming

Main ideas

1. Characterize the structure of an optimal solution.

Dynamic Programming

Main ideas

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

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.

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.

The Rod Cutting problem

The Rod Cutting problem

The Problem

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

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	17	17

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

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	17	17

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

Optimal substructure property

Optimal substructure property

Recurrence

Optimal substructure property

Recurrence

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

Optimal substructure property

Recurrence

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

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.

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,

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,

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

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,

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

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,

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

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

Recurrence (1) can be expressed more succinctly as:

$$r_n \quad =$$

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,

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

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

Recurrence (1) can be expressed more succinctly as:

$$\begin{aligned} r_n &= \max_{1 \leq i \leq n} (p_i + r_{n-i}) \\ r_0 &= 0 \end{aligned} \tag{2}$$

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,

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

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

Recurrence (1) can be expressed more succinctly as:

$$\begin{aligned} r_n &= \max_{1 \le i \le n} (p_i + r_{n-i}) \\ r_0 &= 0 \end{aligned} \tag{2}$$

Why are Recurrence (1) and Recurrence (2) equivalent?

A recursive implementation

A recursive implementation

Recursive Algorithm

A recursive implementation

Recursive Algorithm

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

1: **if** $(n = 0)$ **then**

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

```
if (n = 0) then
    return(0).
```

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

```
if (n = 0) then
    return(0).
end if
```

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

1: **if** ($n = 0$) **then**

2: **return**(0).

3: **end if**

4: $q = -\infty$.

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**

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

if $(n = 0)$ **then**
    return(0).
end if
$q = -\infty$.
for $(i = 1$ **to** $n)$ **do**
    $q = \max(q,$

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

if $(n = 0)$ **then**
    return(0).
end if
$q = -\infty$.
for $(i = 1$ **to** $n)$ **do**
    $q = \max(q, p[i] +$ CUT-ROD$(p, n - i))$.

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 2.12: The recursive rod-cutting algorithm

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

```
if (n = 0) then
    return(0).
end if
q = −∞.
for (i = 1 to n) do
    q = max(q, p[i] + CUT-ROD(p, n − i)).
end for
```

Algorithm 2.13: The recursive rod-cutting algorithm

Analysis

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

$$T(n) \quad =$$

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

if ($n = 0$) **then**
  return(0).
end if
$q = -\infty$.
for ($i = 1$ **to** n) **do**
  $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.
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)
if (n = 0) then
    return(0).
end if
q = −∞.
for (i = 1 to n) do
    q = max(q, p[i] + CUT-ROD(p, n − i)).
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

Analysis of the recursive algorithm

Analysis (contd.)

Analysis of the recursive algorithm

Analysis (contd.)

$$T(n) =$$

Analysis of the recursive algorithm

Analysis (contd.)

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ \end{cases}$$

Analysis of the recursive algorithm

Analysis (contd.)

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

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 $T(n) =$

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 $T(n) = 2^n$.

The Bottom-up approach

The Bottom-up approach

The bottom-up algorithm

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ be a new array.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ be a new array.
2: $r[0] = 0$.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

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

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ be a new array.
2: $r[0] = 0$.
3: **for** ($j = 1$ **to** n) **do**
4: $q = -\infty$.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ 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**

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ be a new array.
2: $r[0] = 0$.
3: **for** ($j = 1$ **to** n) **do**
4: $\quad q = -\infty$.
5: $\quad$ **for** ($i = 1$ **to** j) **do**
6: $\quad\quad q = \max(q, p[i] + r[j - i])$.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ be a new array.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        $q = \max(q, p[i] + r[j-i])$.
    end for

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ be a new array.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        $q = \max(q, p[i] + r[j - i])$.
    end for
    $r[j] = q$.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] be a new array.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        q = max(q, p[i] + r[j − i]).
    end for
    r[j] = q.
end for
```

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ be a new array.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        $q = \max(q, p[i] + r[j - i])$.
    end for
    $r[j] = q$.
end for
return($r[n]$).

Algorithm 2.29: Bottom-up rod-cutting

Analyzing the bottom-up approach

Analyzing the bottom-up approach

Analysis

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.

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,

$$T(n) \quad =$$

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,

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ \end{cases}$$

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,

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

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 $T(n) =$

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 $T(n) = \Theta(n^2)$.

Reconstructing the Solution

Reconstructing the Solution

The bottom-up algorithm with solution

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Reconstructing the Solution

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.

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
2: $r[0] = 0$.

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
2: $r[0] = 0$.
3: **for** ($j = 1$ **to** n) **do**

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

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

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        if ($q < p[i] + r[j - i]$) **then**

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
  $q = -\infty$.
  for ($i = 1$ **to** j) **do**
    if ($q < p[i] + r[j - i]$) **then**
      $q = p[i] + r[j - i]$.

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] and s[0 · ·n] be new arrays.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        if (q < p[i] + r[j − i]) then
            q = p[i] + r[j − i].
            s[j] = i. {The unsplittable left side is recorded.}
```

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        if ($q < p[i] + r[j - i]$) **then**
            $q = p[i] + r[j - i]$.
            $s[j] = i$. {The unsplittable left side is recorded.}
        end if

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
  $q = -\infty$.
  for ($i = 1$ **to** j) **do**
    if ($q < p[i] + r[j - i]$) **then**
      $q = p[i] + r[j - i]$.
      $s[j] = i$. {The unsplittable left side is recorded.}
    end if
  end for

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        if ($q < p[i] + r[j - i]$) **then**
            $q = p[i] + r[j - i]$.
            $s[j] = i$. {The unsplittable left side is recorded.}
        end if
    end for
    $r[j] = q$.

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] and s[0 · ·n] be new arrays.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        if (q < p[i] + r[j − i]) then
            q = p[i] + r[j − i].
            s[j] = i. {The unsplittable left side is recorded.}
        end if
    end for
    r[j] = q.
end for
```

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
  $q = -\infty$.
  for ($i = 1$ **to** j) **do**
    if ($q < p[i] + r[j - i]$) **then**
      $q = p[i] + r[j - i]$.
      $s[j] = i$. {The unsplittable left side is recorded.}
    end if
  end for
  $r[j] = q$.
end for
return($r[n]$).

Algorithm 2.45: Bottom-up rod-cutting

Outputting the solution

Outputting the solution

Printing the Solution

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**

Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)

```
while (n > 0) do
    print s[n].
```

Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)

```
while (n > 0) do
    print s[n].
    n = n − s[n].
```

Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)

```
while (n > 0) do
    print s[n].
    n = n − s[n].
end while
```

Algorithm 2.52: Extracting the solution

The Matrix Chain Multiplication problem

The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$,

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 $d_{i-1} \times d_i$,

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 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,

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 $p \cdot q \cdot r$.

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.

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 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 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

$$T(n) =$$

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

$$T(n) = \begin{cases} 1, & \text{if } n = 2 \\ & \end{cases}$$

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 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

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

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

Optimality Substructure

Optimality Substructure

Substructure

Optimality Substructure

Substructure

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

Optimality Substructure

Substructure

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

Yes!

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.

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?)

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.

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.

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

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 $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

$$m[i,j] =$$

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.

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

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

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.

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

$$m[i, j] = \begin{cases} 0, & \text{if } j = i \\ & \end{cases}$$

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.

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

$$m[i,j] = \begin{cases} 0, & \text{if } j = i \\ \min_{i \le k < j}(m[i,k] + m[k+1,j] & \end{cases}$$

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.

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

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

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.

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

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

Resource analysis

Resource analysis

Analysis

Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space.

Resource analysis

Analysis

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

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.

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 out dynamic programming algorithm is $O(n^3)$.

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 out dynamic programming algorithm is $O(n^3)$.

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

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 out 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;

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 out 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.

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 out 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;

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 out 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 $m[i, j]$.

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 out 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

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 out 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 $\langle A_{7\times 10} \cdot B_{10\times 3} \cdot C_{3\times 8} \cdot D_{8\times 4}\rangle$.

Binary Knapsack

Binary Knapsack

Binary Knapsack

Binary Knapsack

Binary Knapsack

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

Binary Knapsack

Binary Knapsack

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.

Binary Knapsack

Binary Knapsack

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.

Binary Knapsack

Binary Knapsack

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.

Binary Knapsack

Binary Knapsack

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.

Binary Knapsack

Binary Knapsack

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:

Binary Knapsack

Binary Knapsack

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:

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

Binary Knapsack

Binary Knapsack

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:

$$\begin{array}{lll} \max & \sum_{i=1}^{n} p_i \cdot x_i & \\ & \sum_{i=1}^{n} w_i \cdot x_i & \leq W \end{array}$$

Binary Knapsack

Binary Knapsack

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:

$$\begin{array}{lll} \max & \sum_{i=1}^{n} p_i \cdot x_i & \\ & \sum_{i=1}^{n} w_i \cdot x_i & \leq W \\ & x_i = \{0, 1\} \ \ \forall i = 1, 2, \ldots, n & \end{array}$$

A DP-based algorithm for binary knapsack

A DP-based algorithm for binary knapsack

Principle of optimality

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.

A DP-based algorithm for binary knapsack

Principle of optimality

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

A DP-based algorithm for binary knapsack

Principle of optimality

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

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\textsc{Knap}(n, W)$ 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 $o_n \notin S$.
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 $\textsc{Knap}(n, W)$ 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 $\textsc{Knap}(n - 1, W - w_n)$.

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 $o_n \notin S$.
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

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let 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 KNAP$(n-1, W - w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for KNAP$(n-1,$

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let 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 KNAP$(n-1, W-w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for KNAP$(n-1, W)$.

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let 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 KNAP$(n-1, W-w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for KNAP$(n-1, W)$. (Why?)

Formulating the recurrence

Formulating the recurrence

The Recurrence

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.

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?

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, $V[n, W]$.

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,

$$V[i, w] \quad = \quad \max \left\{ \right.$$

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,

$$V[i, w] = \max \begin{cases} V[i-1, w-w_i] + p_i \\ \end{cases}$$

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,

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

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,

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

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,

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

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,

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

4. Initial conditions:

Formulating the recurrence

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,

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

4. Initial conditions:

$$V[0, w] \quad = \quad 0, \quad 0 \leq w \leq W$$

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,

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

4. Initial conditions:

$$\begin{aligned} V[0, w] &= 0, \quad 0 \leq w \leq W \\ V[i, w] &= -\infty, \quad w < 0 \end{aligned}$$

Example

Example

Exercise

Example

Exercise

Solve the following instance of Knapsack:

Example

Exercise

Solve the following instance of Knapsack:

1. $n = 4$,

Example

Exercise

Solve the following instance of Knapsack:

1. $n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$,

Example

Exercise

Solve the following instance of Knapsack:

1. $n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$,

Example

Exercise

Solve the following instance of Knapsack:

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

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

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$											

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1											

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0						

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10

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

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2											

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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							

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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											

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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							

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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											

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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	0	0	0								

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

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	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	0	0	0	50	50	50	50	90	90	90	90

A Portfolio optimization example

A Portfolio optimization example

Example

A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

A Portfolio optimization example

Example

Consider the following portfolio optimization problem:

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

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.

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.

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?

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

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.

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,

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,

$$\begin{aligned} \max \quad & 11 \cdot x_1 + 8 \cdot x_2 + 6 \cdot x_3 \\ & 7 \cdot x_1 + 5 \cdot x_2 + 4 \cdot x_3 \leq 14 \end{aligned}$$

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,

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