Lecture 16
Dynamic Programming

CS 161 Design and Analysis of Algorithms
Ioannis Panageas
Dynamic Programming

▶ Avoids brute force search
▶ Somewhat similar to D&C, but there are major differences
▶ Takes some practice to get used to

Note: This is difficult material. Readings:
▶ [GT]: Chapter 12
▶ [CLRS]: Chapter 15
▶ [Kleinberg and Tardos]: Chapter 6
Dynamic Programming

Dynamic programming

Note: This is difficult material.

Readings:

▶ [GT]: Chapter 12
▶ [CLRS] Chapter 15
▶ [Kleinberg and Tardos], Chapter 6
Dynamic Programming

Dynamic programming

- Avoids brute force search

Note: This is difficult material.

Readings:

- [GT]: Chapter 12
- [CLRS] Chapter 15
- [Kleinberg and Tardos], Chapter 6
Dynamic Programming

Dynamic programming

▶ Avoids brute force search
▶ Somewhat Similar to D&C, but there are major differences

Note: This is difficult material.

Readings:
▶ [GT]: Chapter 12
▶ [CLRS] Chapter 15
▶ [Kleinberg and Tardos], Chapter 6
Dynamic Programming

Dynamic programming

- Avoids brute force search
- Somewhat Similar to D&C, but there are major differences
- Takes some practice to get used to

Note: This is difficult material.

Readings:
- [GT]: Chapter 12
- [CLRS] Chapter 15
- [Kleinberg and Tardos], Chapter 6
Dynamic Programming

Dynamic programming
  ▶ Avoids brute force search
  ▶ Somewhat Similar to D&C, but there are major differences
  ▶ Takes some practice to get used to

Note: This is difficult material.
Dynamic Programming

Dynamic programming

- Avoids brute force search
- Somewhat Similar to D&C, but there are major differences
- Takes some practice to get used to

**Note:** This is difficult material. Readings:

- [GT]: Chapter 12
- [CLRS] Chapter 15
- [Kleinberg and Tardos], Chapter 6
Dynamic Programming

Dynamic programming

- Avoids brute force search
- Somewhat Similar to D&C, but there are major differences
- Takes some practice to get used to

Note: This is difficult material. Readings:
- [GT]: Chapter 12
Dynamic Programming

Dynamic programming

▶ Avoids brute force search
▶ Somewhat Similar to D&C, but there are major differences
▶ Takes some practice to get used to

**Note:** This is difficult material. Readings:

▶ [GT]: Chapter 12
▶ [CLRS] Chapter 15
Dynamic Programming

Dynamic programming

- Avoids brute force search
- Somewhat Similar to D&C, but there are major differences
- Takes some practice to get used to

Note: This is difficult material. Readings:

- [GT]: Chapter 12
- [CLRS] Chapter 15
- [Kleinberg and Tardos], Chapter 6
Dynamic Programming vs. Recursion

▶ Dynamic programming be thought of as being the reverse of recursion
▶ Similar to D&C:
  ▶ Is based on a recurrence
  ▶ Obtains problem solution by using subproblem solutions
▶ Opposite of D&C:
  ▶ Works from small problems to large problems
  ▶ Motivated by recursion but does not actually use recursion
  ▶ Avoids redundantly solving the same subproblem multiple times by storing subproblem solutions
  ▶ This requires careful indexing of subproblems
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions
- Opposite of D&C:
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion

- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions

- Opposite of D&C:
  - Works from small problems to large problems
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions
- Opposite of D&C:
  - Works from small problems to large problems
  - Motivated by recursion but does not actually use recursion
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions
- Opposite of D&C:
  - Works from small problems to large problems
  - Motivated by recursion but does not actually use recursion
- Avoids redundantly solving the same subproblem multiple times by storing subproblem solutions
Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions
- Opposite of D&C:
  - Works from small problems to large problems
  - Motivated by recursion but does not actually use recursion
- Avoids redundantly solving the same subproblem multiple times by storing subproblem solutions
  - This requires careful indexing of subproblems
Dynamic Programming vs. Recursion

Recursion is top-down
Dynamic Programming (DP) is bottom-up

Recursion solves all relevant subproblems
DP may also solve some irrelevant subproblems

Recursion may solve some subproblems many times
DP solves each subproblem only once

D&C / Memoized Dynamic
Recursion
Recursion
Programming

Basic approach
recursion
recursion
iteration

Use of recurrence
top-down
top-down
bottom-up

Store subproblem solutions
No
Yes
Yes

Space needed for stack
Yes
Yes
No

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Dynamic Programming vs. Recursion

- Recursion is top-down
  Dynamic Programming (DP) is bottom-up
Dynamic Programming vs. Recursion

- Recursion is top-down
  Dynamic Programming (DP) is bottom-up
- Recursion solves all relevant subproblems
  DP may also solve some irrelevant subproblems
Dynamic Programming vs. Recursion

- Recursion is top-down
- Dynamic Programming (DP) is bottom-up
- Recursion solves all relevant subproblems
- DP may also solve some irrelevant subproblems
- Recursion may solve some subproblems many times
- DP solves each subproblem only once
Dynamic Programming vs. Recursion

- Recursion is top-down
  Dynamic Programming (DP) is bottom-up
- Recursion solves all relevant subproblems
  DP may also solve some irrelevant subproblems
- Recursion may solve some subproblems many times
  DP solves each subproblem only once

<table>
<thead>
<tr>
<th></th>
<th>D&amp;C / Recursion</th>
<th>Memoized Recursion</th>
<th>Dynamic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic approach</td>
<td>recursion</td>
<td>recursion</td>
<td>iteration</td>
</tr>
<tr>
<td>Use of recurrence</td>
<td>top-down</td>
<td>top-down</td>
<td>bottom-up</td>
</tr>
<tr>
<td>Store subproblem solutions</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Space needed for stack</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Problem: Weighted interval scheduling

Input: Collection of \( n \) Intervals represented by Start Time, Finish Time, and Value: \((s(j), f(j), v(j))\).

Problem: Find a non-overlapping set of intervals that maximizes the total value.

Example:

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
1 & 2 & 3 & 4 \\
2 & 3 & 5 & 7 \\
4 & 2 & 5 & 1 \\
6 & 9 & 12 & 1 \\
\end{array}
\]
Problem: Weighted interval scheduling

▶ **Input:** Collection of \( n \) Intervals represented by Start Time, Finish Time, and Value: \((s(j), f(j), v(j))\).
Problem: Weighted interval scheduling

- **Input:** Collection of \( n \) Intervals represented by Start Time, Finish Time, and Value: \((s(j), f(j), v(j))\).
- **Problem:** Find a non-overlapping set of intervals that maximizes the total value.
Problem: Weighted interval scheduling

▶ **Input:** Collection of \( n \) Intervals represented by Start Time, Finish Time, and Value: \((s(j), f(j), v(j))\).

▶ **Problem:** Find a non-overlapping set of intervals that maximizes the total value.

▶ **Example:**

<table>
<thead>
<tr>
<th>( j )</th>
<th>( s(j) )</th>
<th>( f(j) )</th>
<th>( v(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram:

```
0 1 2 3 4 5 6 7 8 9 10 11 12
1
2
3
4
5
6
```

0 1 2 3 4 5 6 7 8 9 10 11 12
2
4
4
4
7
2
1

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Weighted interval scheduling problem: Preprocessing
Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time.

<table>
<thead>
<tr>
<th>( j )</th>
<th>( s(j) )</th>
<th>( f(j) )</th>
<th>( v(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & & & & & & & & & 2 & & & \\
2 & & & & & & 4 & & & & & & \\
3 & & & & & & & & 4 & & & \\
4 & & & & & & & & & 7 & & & \\
5 & & & & & & & & & & & 2 & \\
6 & & & & & & & & & & & 1 & \\
\end{array} \]
Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time. (Here they are already sorted).

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>
Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time. (Here they are already sorted).
2. For each interval $j$, define $p(j)$ to be:

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>
Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time. (Here they are already sorted).

2. For each interval $j$, define $p(j)$ to be:
   - The highest-numbered interval $i < j$ that does not overlap interval $j$ (if such an interval exists)

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>
Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time. (Here they are already sorted).

2. For each interval $j$, define $p(j)$ to be:
   - The highest-numbered interval $i < j$ that does not overlap interval $j$ (if such an interval exists)
   - 0 (if no such an interval exists)

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

![Diagram showing intervals and their scheduling order]
Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time. (Here they are already sorted).

2. For each interval $j$, define $p(j)$ to be:
   - The highest-numbered interval $i < j$ that does not overlap interval $j$ (if such an interval exists)
   - 0 (if no such an interval exists)

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
<th>$p(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

![Diagram of intervals and their $p(j)$ values]

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Simple recursive algorithm

For the problem on intervals 1 through j:

▶ Either the optimal solution contains the last interval or it doesn’t
▶ If it does:
    ▶ The optimal value is $v(j)$ plus the value of the optimal collection from 1, . . . , $p(j)$
▶ If it does not:
    ▶ Optimal value is the value of the optimal collection from 1, . . . , $j - 1$
▶ So the optimal value is the maximum of these two possible values:

def OPT(j):
if j = 0: return 0
else: return max($v(j)$ + OPT($p(j)$), OPT($j - 1$))

Correct, but very inefficient because . . .
Simple recursive algorithm

For the problem on intervals 1 through \( j \):

Either the optimal solution contains the last interval or it doesn’t:

- If it does:
  - The optimal value is \( v(j) \) plus the value of the optimal collection from 1, \ldots, \( p(j) \)
- If it does not:
  - Optimal value is the value of the optimal collection from 1, \ldots, \( j-1 \)

So the optimal value is the maximum of these two possible values:

\[
\text{def OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(v(j) + \text{OPT}(p(j)), \text{OPT}(j-1)) & \text{otherwise}
\end{cases}
\]

Correct, but very inefficient because \( \text{OPT}(j) \) is recomputed multiple times.
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t

```python
def OPT(j):
    if j = 0: return 0
    else: return max(v(j)+OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because . . .
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t
- If it does:

```python
def OPT(j):
    if j = 0: return 0
    else: return max(v(j)+OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because . . .
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t
- If it does:
  - The optimal value is $v(j)$ plus the value of the optimal collection from 1, . . . , $p(j)$

```python
def OPT(j):
    if j = 0: return 0
    else: return max(v(j)+OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because . . .
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t
- If it does:
  - The optimal value is $v(j)$ plus the value of the optimal collection from 1, \ldots, $p(j)$
- If it does not:

```python
def OPT(j):
    if j = 0: return 0
    else: return max(v(j)+OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because the same value of OPT() is recomputed multiple times.
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t
  - If it does:
    - The optimal value is $v(j)$ plus the value of the optimal collection from 1, \ldots, $p(j)$
  - If it does not:
    - optimal value is the value of the optimal collection from 1, \ldots, $j-1$

```python
def OPT(j):
    if j == 0: return 0
    else: return max(v(j)+OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because the same value of OPT() is recomputed multiple times.
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t
- If it does:
  - The optimal value is $v(j)$ plus the value of the optimal collection from 1, $\ldots$, $p(j)$
- If it does not:
  - optimal value is the value of the optimal collection from 1, $\ldots$, $j-1$
- So the optimal value is the maximum of these two possible values:
Simple recursive algorithm

For the problem on intervals 1 through \( j \):

- Either the optimal solution contains the last interval or it doesn’t
- If it does:
  - The optimal value is \( \nu(j) \) plus the value of the optimal collection from 1, \ldots, \( p(j) \)
- If it does not:
  - optimal value is the value of the optimal collection from 1, \ldots, \( j - 1 \)
- So the optimal value is the maximum of these two possible values:

```python
def OPT(j):
    if j = 0: return 0
    else: return max(\nu(j) + OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because . . .
Simple recursive algorithm

For the problem on intervals 1 through \( j \):

- Either the optimal solution contains the last interval or it doesn’t
- If it does:
  - The optimal value is \( v(j) \) plus the value of the optimal collection from 1, \ldots, \( p(j) \)
- If it does not:
  - Optimal value is the value of the optimal collection from 1, \ldots, \( j - 1 \)
- So the optimal value is the maximum of these two possible values:

```python
def OPT(j):
    if j == 0: return 0
    else: return max(v(j) + OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because ...
Simple recursive algorithm

For the problem on intervals 1 through $j$:

- Either the optimal solution contains the last interval or it doesn’t
- If it does:
  - The optimal value is $v(j)$ plus the value of the optimal collection from 1, \ldots, $p(j)$
- If it does not:
  - optimal value is the value of the optimal collection from 1, \ldots, $j-1$
- So the optimal value is the maximum of these two possible values:

  ```python
  def OPT(j):
      if j = 0: return 0
      else: return max(v(j)+OPT(p(j)), OPT(j-1))
  ```

Correct, but very inefficient because ... 

- the same value of OPT() is recomputed multiple times.
Memoizing the recursion

We can avoid recomputing OPT() values by storing them. So we just look up a previously computed value rather than recomputing it.

Declare an array $M[1..n]$, where each entry can contain an integer or "undefined".

Initialize all entries to "undefined".

```python
def MemoizedOPT(j):
    if j == 0: return(0);
    else:
        if M[j] = "undefined" :
            M[j] = max(v(j)+MemoizedOPT(p(j)), MemoizedOPT(j-1))
        return (M[j])
```
Memoizing the recursion

- We can avoid recomputing OPT() values by storing them

```python
def Memoized(j):
    if j == 0:
        return 0;
    elif M[j] == "undefined":
        M[j] = max(v(j)+Memoized(p(j)), Memoized(j-1))
    return M[j]
```
Memoizing the recursion

► We can avoid recomputing $\text{OPT}()$ values by storing them
  ► So we just look up a previously computed value rather than recomputing it

```python
def Memoized_OPT(j):
    if j == 0:
        return(0);
    else:
        if M[j] == "undefined"
            M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
        return (M[j])
```
Memoizing the recursion

- We can avoid recomputing OPT() values by storing them
  - So we just look up a previously computed value rather than recomputing it
- Declare an array $M[1..n]$, where each entry can contain an integer or "undefined"
Memoizing the recursion

- We can avoid recomputing OPT() values by storing them
  - So we just look up a previously computed value rather than recomputing it
- Declare an array $M[1..n]$, where each entry can contain an integer or "undefined"
- Initialize all entries to "undefined"
Memoizing the recursion

- We can avoid recomputing OPT() values by storing them
  - So we just look up a previously computed value rather than recomputing it
- Declare an array $M[1..n]$, where each entry can contain an integer or ”undefined”
- Initialize all entries to “undefined”

```python
def Memoized_OPT(j):
    if j = 0:    return(0);
else:
    if M[j] = "undefined" :
        M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
    return (M[j])
```
Analysis of Memoized Algorithm

```python
def Memoized_OPT(j):
    if j == 0:
        return 0;
    else:
        if M[j] = "undefined" :
            M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
        return M[j]

Memoized_OPT(n)
```

Run Memoized_OPT on a collection of n intervals:

▶ For every pair of recursive calls, an entry of M gets filled in.
▶ Hence, O(n) calls.
Analysis of Memoized Algorithm

```python
def Memoized_OPT(j):
    if j == 0:  return(0);
    else:
        if M[j] = "undefined" :
            M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
        return (M[j])
Memoized_OPT(n)
```
Analysis of Memoized Algorithm

```python
def Memoized_OPT(j):
    if j = 0: return(0);
    else:
        if M[j] = "undefined" :
            M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
        return (M[j])

Memoized_OPT(n)
```

Run Memoized_OPT on a collection of \( n \) intervals:
Analysis of Memoized Algorithm

```python
def Memoized_OPT(j):
    if j = 0:  return(0);
    else:
        if M[j] = "undefined" :
            M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
        return (M[j])
Memoized_OPT(n)
```

Run Memoized_OPT on a collection of $n$ intervals:
  
  - For every pair of recursive calls, an entry of $M$ gets filled in.
Analysis of Memoized Algorithm

```python
def Memoized_OPT(j):
    if j = 0: return(0);
    else:
      if M[j] = "undefined" :
         M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
      return (M[j])
Memoized_OPT(n)
```

Run Memoized_OPT on a collection of $n$ intervals:
- For every pair of recursive calls, an entry of $M$ gets filled in.
- Hence, $O(n)$ calls.
Dynamic Programming Solution

In memoized recursion, we entered a value in the M array based on values that appear earlier in that array. Instead of computing the entries in array M recursively, we can:

- Get rid of the recursion entirely
- Compute the array entries iteratively

This is the dynamic programming solution.

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

Simple, efficient code.

Runs in $O(n)$ time.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the M array based on values that appear earlier in that array

```python
def Iterative(OPT):
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

Simple, efficient code.

Runs in $O(n)$ time.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the M array based on values that appear earlier in that array.
- Instead of computing the entries in array M recursively, we can:

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)], M[j-1])
```

- Simple, efficient code.
- Runs in $O(n)$ time.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the $M$ array based on values that appear earlier in that array.
- Instead of computing the entries in array $M$ recursively, we can:
  - Get rid of the recursion entirely.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the M array based on values that appear earlier in that array.
- Instead of computing the entries in array M recursively, we can:
  - Get rid of the recursion entirely
  - Compute the array entries iteratively

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

Simple, efficient code.

Runs in $O(n)$ time.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the $M$ array based on values that appear earlier in that array.
- Instead of computing the entries in array $M$ recursively, we can:
  - Get rid of the recursion entirely
  - Compute the array entries iteratively
- This is the **dynamic programming** solution.
Dynamic Programming Solution

▪ In memoized recursion, we entered a value in the M array based on values that appear earlier in that array.

▪ Instead of computing the entries in array M recursively, we can:
  ▪ Get rid of the recursion entirely
  ▪ Compute the array entries iteratively

▪ This is the **dynamic programming** solution.

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

Simple, efficient code.
Runs in \( O(n) \) time.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the M array based on values that appear earlier in that array.
- Instead of computing the entries in array M recursively, we can:
  - Get rid of the recursion entirely
  - Compute the array entries iteratively
- This is the dynamic programming solution.

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

- Simple, efficient code.
Dynamic Programming Solution

- In memoized recursion, we entered a value in the M array based on values that appear earlier in that array.
- Instead of computing the entries in array M recursively, we can:
  - Get rid of the recursion entirely
  - Compute the array entries iteratively
- This is the **dynamic programming** solution.

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

- Simple, efficient code.
- Runs in $O(n)$ time.
Computing the Optimal Set of Intervals

```python
def Iterative OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

There is one issue here:

The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.

This is a standard with dynamic programming problems. We usually proceed in two steps.

1. We first consider how to compute the optimum cost or value
2. Once we know how to compute the optimum cost or value, we then consider how to compute a configuration that has the optimum cost or value.

Compute additional information (usually an additional array) as we compute the optimum cost or value.

Run a post-processing step that uses this additional information.
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

There is one issue here:
The algorithm given above computes the value of an optimal interval set, but not the intervals.
This is a standard with dynamic programming problems. We usually proceed in two steps.
1. We first consider how to compute the optimum cost or value
2. Once we know how to compute the optimum cost or value, we then consider how to compute a configuration that has the optimum cost or value.

Compute additional information (usually an additional array) as we compute the optimum cost or value.
Run a post-processing step that uses this additional information.
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

▶ There is one issue here:
Computing the Optimal Set of Intervals

def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])

▶ There is one issue here:
    ▶ The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j) + M[p(j)], M[j-1])
```

- There is one issue here:
  - The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.

- This is a standard with dynamic programming problems. We usually proceed in two steps.
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

- There is one issue here:
  - The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.

- This is a standard with dynamic programming problems. We usually proceed in two steps.
  1. We first consider how to compute the optimum cost or value
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

▶ There is one issue here:

▶ The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.

▶ This is a standard with dynamic programming problems. We usually proceed in two steps.

1. We first consider how to compute the optimum cost or value
2. Once we know how to compute the optimum cost or value, we then consider how to compute a configuration that has the optimum cost or value.
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

- There is one issue here:
  - The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.

- This is a standard with dynamic programming problems. We usually proceed in two steps.
  1. We first consider how to compute the optimum cost or value
  2. Once we know how to compute the optimum cost or value, we then consider how to compute a configuration that has the optimum cost or value.

- Compute additional information (usually an additional array) as we compute the optimum cost or value.
Computing the Optimal Set of Intervals

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v(j)+M[p(j)],M[j-1])
```

- There is one issue here:
  - The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.

- This is a standard with dynamic programming problems. We usually proceed in two steps.
  1. We first consider how to compute the optimum cost or value
  2. Once we know how to compute the optimum cost or value, we then consider how to compute a configuration that has the optimum cost or value.

- Compute additional information (usually an additional array) as we compute the optimum cost or value.

- Run a post-processing step that uses this additional information
Computing the Optimal Set of Intervals

For each $j$, we remember whether the optimal set for the first $j$ intervals contains interval $j$.

We compute two arrays:

- $M[j]$ stores the best value we can get from the first $j$ intervals (as before)
- $keep[j]$ stores whether the best choice for the first $j$ intervals includes interval $j$

**Iterative OPT:**

```plaintext
M[0] = 0
for j = 1 to n:
    if $v(j) + M[p(j)] > M[j-1]$
        $M[j] = v(j) + M[p(j)]$
        keep[j] = True
    else:
        $M[j] = M[j-1]$
        keep[j] = False
```
Computing the Optimal Set of Intervals

- For each $j$, we remember whether the optimal set for the first $j$ intervals contains interval $j$. 

Iterative OPT:

- $M[0] = 0$
- for $j = 1$ to $n$:
  - if $v(j) + M[p(j)] > M[j-1]$:
    - $M[j] = v(j) + M[p(j)]$
    - keep$[j] = \text{True}$
  - else:
    - $M[j] = M[j-1]$
    - keep$[j] = \text{False}$
Computing the Optimal Set of Intervals

- For each $j$, we remember whether the optimal set for the first $j$ intervals contains interval $j$.
- We compute two arrays:

```plaintext
Iterative OPT:

$M[0] = 0$

for $j = 1$ to $n$:

if $v(j) + M[p(j)] > M[j-1]$

$M[j] = v(j) + M[p(j)]$

keep$[j] = True$

else:

$M[j] = M[j-1]$

keep$[j] = False$
```
Computing the Optimal Set of Intervals

- For each $j$, we remember whether the optimal set for the first $j$ intervals contains interval $j$.
- We compute two arrays:
  - $M[j]$ stores the best value we can get from the first $j$ intervals (as before)
Computing the Optimal Set of Intervals

- For each $j$, we remember whether the optimal set for the first $j$ intervals contains interval $j$.
- We compute two arrays:
  - $M[j]$ stores the best value we can get from the first $j$ intervals (as before)
  - $\text{keep}[j]$ stores whether the best choice for the first $j$ intervals includes interval $j$

```python
def IterativeOPT:
    M[0] = 0
    for j = 1 to n:
        if $v(j) + M[p(j)] > M[j-1]$
            $M[j] = v(j) + M[p(j)]$
            $\text{keep}[j] = True$
        else:
            $M[j] = M[j-1]$
            $\text{keep}[j] = False$
```
Computing the Optimal Set of Intervals

- For each $j$, we remember whether the optimal set for the first $j$ intervals contains interval $j$.
- We compute two arrays:
  - $M[j]$ stores the best value we can get from the first $j$ intervals (as before)
  - $\text{keep}[j]$ stores whether the best choice for the first $j$ intervals includes interval $j$

```python
def Iterative_OPT:
    M[0] = 0
    for j = 1 to n:
        if v(j)+M[p(j)] > M[j-1]:
            M[j] = v(j)+M[p(j)]
            keep[j] = True
        else:
            M[j] = M[j-1]
            keep[j] = False
```
Computing the Optimal Set of Intervals, continued
Computing the Optimal Set of Intervals, continued

Once we have computed the two arrays \( M[ ] \) and \( \text{keep}[ ] \):
Computing the Optimal Set of Intervals, continued

Once we have computed the two arrays \(M[]\) and \(\text{keep}[]\):

```python
def PrintSolution(j):
    if j = 0:
        return;
    if keep[j]:
        PrintSolution(p(j))
        print(j)
    else:
        PrintSolution(j-1)

PrintSolution(n)
```
Our example
Our example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
<th>$p(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The array $M$ contains the solutions of the subproblems. We will refer to this as the memoization table.
Our example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
<th>$p(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The array $M$ contains the solutions of the subproblems. We will refer to this as the memoization table.

Selected intervals: $\{1, 4\}$.
Our example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
<th>$p(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The array $M$ contains the solutions of the subproblems. We will refer to this as the memoization table.

Selected intervals: \{1, 4\}. 

M:  0 2 4 6 9 9 9
keep: T T T T F F
Our example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$s(j)$</th>
<th>$f(j)$</th>
<th>$v(j)$</th>
<th>$p(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Selected intervals: \(\{1, 4\}\).

The array $M$ contains the solutions of the subproblems. We will refer to this as the **memoization table**.

\[
\begin{array}{ccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
2 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
3 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
4 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
5 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
6 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
\end{array}
\]
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

▶ There are only a polynomial number of subproblems
▶ The solution to the original problem can be easily computed from the solution to the subproblems.
▶ For example, when the original problem is one of the subproblems
▶ There is an ordering on the subproblems, together with
▶ A recurrence on subproblem solution that enable the solution to any subproblem \( P \) to be computed from the solutions to some of the subproblems that precede \( P \) in the ordering.

We saw this in the case of the weighted interval scheduling problem.
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

▶ There are only a polynomial number of subproblems
▶ The solution to the original problem can be easily computed from the solution to the subproblems.
▶ For example, when the original problem is one of the subproblems
▶ There is an ordering on the subproblems, together with
▶ A recurrence on subproblem solution that enable the solution to any subproblem $P$ to be computed from the solutions to some of the subproblems that precede $P$ in the ordering

We saw this in the case of the weighted interval scheduling problem.
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

▶ There are only a polynomial number of subproblems

For example, when the original problem is one of the subproblems

▶ There is an ordering on the subproblems, together with

▶ A recurrence on subproblem solution that enables the solution to any subproblem \( P \) to be computed from the solutions to some of the subproblems that precede \( P \) in the ordering

We saw this in the case of the weighted interval scheduling problem.
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solution to the subproblems.

For example, when the original problem is one of the subproblems, there is an ordering on the subproblems, together with a recurrence on subproblem solution that enable the solution to any subproblem \( P \) to be computed from the solutions to some of the subproblems that precede \( P \) in the ordering.

We saw this in the case of the weighted interval scheduling problem.
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

▶ There are only a polynomial number of subproblems
▶ The solution to the original problem can be easily computed from the solution to the subproblems.
  ▶ For example, when the original problem is one of the subproblems
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solution to the subproblems.
  - For example, when the original problem is one of the subproblems
- There is
  - An ordering on the subproblems, together with
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solution to the subproblems.
  - For example, when the original problem is one of the subproblems
- There is
  - An ordering on the subproblems, together with
  - A recurrence on subproblem solution that enable the solution to any subproblem $P$ to be computed from the solutions to some of the subproblems that precede $P$ in the ordering
Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solution to the subproblems.
  - For example, when the original problem is one of the subproblems
- There is an ordering on the subproblems, together with
  - A recurrence on subproblem solution that enable the solution to any subproblem $P$ to be computed from the solutions to some of the subproblems that precede $P$ in the ordering

We saw this in the case of the weighted interval scheduling problem.
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
2. A precise definition of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
3. The goal: the solution to the original problem, expressed in terms of certain values of the function from item #2.
4. The initial value(s)/condition(s): values of the function from item #2 for small subproblems that do not need to be decomposed further.
5. The recurrence: a formula describing how to compute the solution of a subproblem from the solutions to smaller subproblems.

Here, “smaller” means “earlier in the ordering.”
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
2. A precise definition of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
3. The goal: the solution to the original problem, expressed in terms of certain values of the function from item #2.
4. The initial value(s) / condition(s): values of the function from item #2 for small subproblems that do not need to be decomposed further.
5. The recurrence: a formula describing how to compute the solution of a subproblem from the solutions to smaller subproblems.

Here, "smaller" means "earlier in the ordering"
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.

2. A precise definition of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
2. A precise definition of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
3. The goal: the solution to the original problem, expressed in terms of certain values of the function from item #2.

4. The initial value(s) / condition(s): values of the function from item #2 for small subproblems that do not need to be decomposed further.
5. The recurrence: a formula describing how to compute the solution of a subproblem from the solutions to smaller subproblems.

Here, "smaller" means "earlier in the ordering"
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
2. A precise definition of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
3. The goal: the solution to the original problem, expressed in terms of certain values of the function from item #2.
4. The initial value(s) / condition(s): values of the function from item #2 for small subproblems that do not need to be decomposed further.
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
2. A precise definition of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
3. The goal: the solution to the original problem, expressed in terms of certain values of the function from item #2.
4. The initial value(s) / condition(s): values of the function from item #2 for small subproblems that do not need to be decomposed further.
5. The recurrence: a formula describing how to compute the solution of a subproblem from the solutions to smaller subproblems.
Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The **subproblem domain**: the set of indices of the subproblems.
2. A **precise definition** of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the **memoization table** represents.)
3. The **goal**: the solution to the original problem, expressed in terms of certain values of the function from item #2.
4. The **initial value(s) / condition(s)**: values of the function from item #2 for small subproblems that do not need to be decomposed further.
5. The **recurrence**: a formula describing how to compute the solution of a subproblem from the solutions to smaller subproblems.

Here, ”smaller” means ”earlier in the ordering”
Solution to Weighted-Interval Scheduling

1. Subproblem domain: $\{0, \ldots, n\}$

2. Function / Memoization table definition: $M(j)$ is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range $\{1, \ldots, j\}$.

3. Goal: $M(n)$

4. Initial value: $M(0) = 0$

5. Recurrence:
   
   $$M(j) = \max(v(j) + M(p(j)), M(j-1)) \text{ for } j \geq 1.$$  

Here, $p(j)$ is a precomputed function defined by:

$$p(j) = \begin{cases} 
    \text{The highest-numbered interval } i < j \text{ that does not overlap interval } j \text{ if such an interval exists} \\
    0 \text{ otherwise}
  \end{cases}$$
Solution to Weighted-Interval Scheduling

1. Subproblem domain: \( \{0, \ldots, n\} \)
Solution to Weighted-Interval Scheduling

1. Subproblem domain: \( \{0, \ldots, n\} \)

2. Function / Memoization table definition: \( M(j) \) is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range \( \{1, \ldots, j\} \)
Solution to Weighted-Interval Scheduling

1. Subproblem domain: \{0, \ldots, n\}
2. Function / Memoization table definition: \(M(j)\) is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range \{1, \ldots, j\}
3. Goal: \(M(n)\)
Solution to Weighted-Interval Scheduling

1. **Subproblem domain:** \( \{0, \ldots, n\} \)
2. **Function / Memoization table definition:** \( M(j) \) is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range \( \{1, \ldots, j\} \)
3. **Goal:** \( M(n) \)
4. **Initial value:** \( M(0) = 0 \)
Solution to Weighted-Interval Scheduling

1. Subproblem domain: \{0, \ldots, n\}
2. Function / Memoization table definition: \( M(j) \) is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range \{1, \ldots, j\}
3. Goal: \( M(n) \)
4. Initial value: \( M(0) = 0 \)
5. Recurrence: \( M(j) = \max(v(j) + M(p(j)), M(j - 1)) \) for \( j \geq 1 \).
Solution to Weighted-Interval Scheduling

1. Subproblem domain: \( \{0, \ldots, n\} \)

2. Function / Memoization table definition: \( M(j) \) is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range \( \{1, \ldots, j\} \)

3. Goal: \( M(n) \)

4. Initial value: \( M(0) = 0 \)

5. Recurrence: \( M(j) = \max(v(j) + M(p(j)), M(j - 1)) \) for \( j \geq 1 \). Here, \( p(j) \) is a precomputed function defined by

\[
p(j) = \begin{cases} 
\text{The highest-numbered interval } i < j \text{ that does not overlap interval } j \text{ if such an interval exists} \\
0 \text{ otherwise}
\end{cases}
\]
Truck loading problem
Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)
Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)

Problem definition:
Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)

Problem definition:

- Truck has weight limit of $W$. 
Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)

Problem definition:

- Truck has weight limit of $W$.
- $n$ boxes. Box $i$ has weight $w_i$.
Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)

Problem definition:

- Truck has weight limit of $W$.
- $n$ boxes. Box $i$ has weight $w_i$.
- We want to load the truck to carry the maximum weight possible, subject to the weight restriction.
Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)

Problem definition:

- Truck has weight limit of $W$.
- $n$ boxes. Box $i$ has weight $w_i$.
- We want to load the truck to carry the maximum weight possible, subject to the weight restriction.
Greedy heuristics don’t work
Greedy heuristics don’t work

1. Heaviest boxes first:
Greedy heuristics don’t work

1. Heaviest boxes first:
   \[ W = 100, \ w_1 = 51, \ w_2 = 50, \ w_3 = 50 \]
Greedy heuristics don’t work

1. Heaviest boxes first:
   \[ W = 100, \ w_1 = 51, \ w_2 = 50, \ w_3 = 50 \]

2. Lightest boxes first:
Greedy heuristics don’t work

1. Heaviest boxes first:
   \[ W = 100, \ w_1 = 51, \ w_2 = 50, \ w_3 = 50 \]

2. Lightest boxes first:
   \[ W = 100, \ w_1 = 1, \ w_2 = 50, \ w_3 = 50 \]
Dynamic Programming Solution: Basic Idea

Suppose we have $i$ boxes and a truck with weight capacity $j$.

- Either the optimum solution contains the last box or it doesn't.
- If the optimum solution contains the last box:
  - The optimum value is $w_i$ plus the optimum value we can get by fitting the first $i-1$ boxes on the truck, after accounting for the weight taken up by box $i$.
- If the optimum solution does not contain the last box:
  - The optimum value is the optimum value we can get from the first $i-1$ boxes.

We will express this more formally on the next slide.
Dynamic Programming Solution: Basic Idea

Suppose we have $i$ boxes and a truck with weight capacity $j$. 
Dynamic Programming Solution: Basic Idea

Suppose we have $i$ boxes and a truck with weight capacity $j$.

- Either the optimum solution contains the last box or it doesn’t.
Dynamic Programming Solution: Basic Idea

Suppose we have $i$ boxes and a truck with weight capacity $j$.

- Either the optimum solution contains the last box or it doesn’t.
- If the optimum solution contains the last box:
Dynamic Programming Solution: Basic Idea

Suppose we have $i$ boxes and a truck with weight capacity $j$.

- Either the optimum solution contains the last box or it doesn’t.
- If the optimum solution contains the last box:
  - The optimum value is $w_i$ plus the optimum value we can get by fitting the first $i - 1$ boxes on the truck, \textit{after accounting for the weight taken up by box $i$}.
Dynamic Programming Solution: Basic Idea

Suppose we have \( i \) boxes and a truck with weight capacity \( j \).

- Either the optimum solution contains the last box or it doesn’t.
- If the optimum solution contains the last box:
  - The optimum value is \( w_i \) plus the optimum value we can get by fitting the first \( i - 1 \) boxes on the truck, after accounting for the weight taken up by box \( i \).
- If the optimum solution does not contain the last box:
Dynamic Programming Solution: Basic Idea

Suppose we have $i$ boxes and a truck with weight capacity $j$.

- Either the optimum solution contains the last box or it doesn’t.
- If the optimum solution contains the last box:
  - The optimum value is $w_i$ plus the optimum value we can get by fitting the first $i - 1$ boxes on the truck, after accounting for the weight taken up by box $i$.
- If the optimum solution does not contain the last box:
  - The optimum value is the optimum value we can get from the first $i - 1$ boxes.
Dynamic Programming Solution: Basic Idea

Suppose we have \( i \) boxes and a truck with weight capacity \( j \).

- Either the optimum solution contains the last box or it doesn’t.
- If the optimum solution contains the last box:
  - The optimum value is \( w_i \) plus the optimum value we can get by fitting the first \( i - 1 \) boxes on the truck, after accounting for the weight taken up by box \( i \).
- If the optimum solution does not contain the last box:
  - The optimum value is the optimum value we can get from the first \( i - 1 \) boxes.

We will express this more formally on the next slide.
Solution: Expressed as recurrence equation

Let $OPT(i, j)$ be the maximum weight we can get by loading from boxes 1 through $i$, up to the weight limit $j$.

Applying what we said on the previous slide:

$$OPT(i, j) = \max (w_i + OPT(i-1, j-w_i), OPT(i-1, j))$$

Note that if $w_i > j$, we can't use box $i$, so only the second choice is available.

This recurrence equation gives us the dynamic programming solution (specified on next slide).
Solution: Expressed as recurrence equation

- Let \( \text{OPT}(i, j) \) be the maximum weight we can get by loading from boxes 1 through \( i \), up to the weight limit \( j \).
Solution: Expressed as recurrence equation

- Let $OPT(i, j)$ be the maximum weight we can get by loading from boxes 1 through $i$, up to the weight limit $j$.
- Applying what we said on the previous slide:

$$OPT(i, j) = \max (w_i + OPT(i-1, j-w_i), OPT(i-1, j))$$

- Note that if $w_i > j$, we can't use box $i$, so only the second choice is available.
- This recurrence equation gives us the dynamic programming solution (specified on next slide).
Solution: Expressed as recurrence equation

Let $OPT(i, j)$ be the maximum weight we can get by loading from boxes 1 through $i$, up to the weight limit $j$.

Applying what we said on the previous slide:

$$OPT(i, j) = \max (w_i + OPT(i - 1, j - w_i), OPT(i - 1, j))$$

Note that if $w_i > j$, we can’t use box $i$, so only the second choice is available.

This recurrence equation gives us the dynamic programming solution (specified on next slide).
Solution: Expressed as recurrence equation

- Let $\text{OPT}(i, j)$ be the maximum weight we can get by loading from boxes 1 through $i$, up to the weight limit $j$.
- Applying what we said on the previous slide:
  \[
  \text{OPT}(i, j) = \max \left( w_i + \text{OPT}(i - 1, j - w_i), \text{OPT}(i - 1, j) \right)
  \]
- Note that if $w_i > j$, we can’t use box $i$, so only the second choice is available.
Solution: Expressed as recurrence equation

- Let $OPT(i, j)$ be the maximum weight we can get by loading from boxes 1 through $i$, up to the weight limit $j$.

- Applying what we said on the previous slide:

  $$OPT(i, j) = \max \left( w_i + OPT(i - 1, j - w_i), OPT(i - 1, j) \right)$$

- Note that if $w_i > j$, we can’t use box $i$, so only the second choice is available.

- This recurrence equation gives us the dynamic programming solution (specified on next slide)
Specifying the Solution
Specifying the Solution

1. Subproblem domain: \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)
Specifying the Solution

1. **Subproblem domain:** \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. **Function / Memoization table definition:** \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) boxes into a truck with maximum capacity \( j \).
Specifying the Solution

1. Subproblem domain: \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. Function / Memoization table definition: \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) boxes into a truck with maximum capacity \( j \).

3. Goal: \( \text{OPT}(n, W) \)
Specifying the Solution

1. **Subproblem domain:** \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. **Function / Memoization table definition:** \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) boxes into a truck with maximum capacity \( j \).

3. **Goal:** \( \text{OPT}(n, W) \)

4. **Initial values:**

   \[
   \begin{align*}
   \text{OPT}(i, 0) & = 0 \quad \text{for all } i \geq 0 \\
   \text{OPT}(0, j) & = 0 \quad \text{for all } j \geq 0
   \end{align*}
   \]
Specifying the Solution

1. **Subproblem domain:** \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. **Function / Memoization table definition:** \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) boxes into a truck with maximum capacity \( j \).

3. **Goal:** \( \text{OPT}(n, W) \)

4. **Initial values:**

   \[
   \begin{align*}
   \text{OPT}(i, 0) &= 0 \text{ for all } i \geq 0 \\
   \text{OPT}(0, j) &= 0 \text{ for all } j \geq 0
   \end{align*}
   \]

5. **Recurrence:**

   \[
   \text{OPT}(i, j) = \begin{cases} 
   \text{max} \left( w_i + \text{OPT}(i - 1, j - w_i), \text{OPT}(i - 1, j) \right) & \text{if } w_i \leq j \\
   \text{OPT}(i - 1, j) & \text{if } w_i > j
   \end{cases}
   \]
Truck Loading Problem DP Pseudocode: compute OPT Matrix

```python
def compute_opt_matrix(w):
    for i = 0 to n: OPT[i,0] = 0
    for j = 0 to W: OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if w[i] > j:
                OPT[i,j] = OPT[i-1,j]
            else:
                OPT[i,j] = max(w[i] + OPT[i-1,j-w[i]], OPT[i-1,j])
    return OPT
```

This tells us the maximum possible weight, but we need to also compute which boxes to load to achieve this maximum weight . . .
def compute_opt_matrix(w):
    for i = 0 to n: OPT[i,0] = 0
    for j = 0 to W: OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if w[i] > j:
                OPT[i,j] = OPT[i-1,j]
            else:
                OPT[i,j] = max(w[i] + OPT[i-1,j-w[i]], OPT[i-1,j])
    return OPT
def compute_opt_matrix(w):
    for i = 0 to n:   OPT[i,0] = 0
    for j = 0 to W:  OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if w[i] > j:
                OPT[i,j] = OPT[i-1,j]
            else:
                OPT[i,j] = max(w[i] + OPT[i-1,j-w[i]], OPT[i-1,j])
    return OPT

This tells us the maximum possible weight, but we need to also compute which boxes to load to achieve this maximum weight . . .
Truck Loading Problem DP Pseudocode: compute choice of boxes

def compute_opt(w):
    for i = 0 to n: OPT[i,0] = 0
    for j = 0 to W: OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if (w[i] > j) or (w[i] + OPT[i-1,j-w[i]] <= OPT[i-1,j]):
                OPT[i,j] = OPT[i-1,j]
                keep[i,j] = False
            else:
                OPT[i,j] = w[i] + OPT[i-1,j-w[i]]
                keep[i,j] = True
    return (OPT,keep)

Running time: $O(n \cdot W)$
Truck Loading Problem DP Pseudocode: compute choice of boxes

Introduce an new array \( \text{keep}[i, j] \), which tells us whether we keep box \( i \) when we solve the subproblem with \( i \) boxes and capacity \( j \).
Truck Loading Problem DP Pseudocode: compute choice of boxes

Introduce an new array `keep[i,j]`, which tells us whether we keep box `i` when we solve the subproblem with `i` boxes and capacity `j`.

```python
def compute_opt_strategy(w):
    for i = 0 to n:  OPT[i,0] = 0
    for j = 0 to W:  OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if (w[i] > j) or (w[i] + OPT[i-1,j-w[i]] <= OPT[i-1,j])
                OPT[i,j] = OPT[i-1,j]
                keep[i,j] = False
            else:
                OPT[i,j] = w[i] + OPT[i-1,j-w[i]]
                keep[i,j] = True
    return (OPT,keep)
```
Truck Loading Problem DP Pseudocode: compute choice of boxes

Introduce an new array \texttt{keep}[i,j], which tells us whether we keep box \( i \) when we solve the subproblem with \( i \) boxes and capacity \( j \).

\begin{verbatim}
def compute_opt_strategy(w):
    for i = 0 to n:
        OPT[i,0] = 0
    for j = 0 to W:
        OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if (w[i] > j) or (w[i] + OPT[i-1,j-w[i]] <= OPT[i-1,j]):
                OPT[i,j] = OPT[i-1,j]
                keep[i,j] = False
            else:
                OPT[i,j] = w[i] + OPT[i-1,j-w[i]]
                keep[i,j] = True
    return (OPT,keep)
\end{verbatim}

Running time: \( O(n \cdot W) \)
Truck Loading Problem DP Pseudocode: compute choice of boxes [continued]

def print_solution(OPT, keep, i, j):
    if i == 0:
        return
    if keep[i, j]:
        print_solution(OPT, keep, i-1, j-w[i])
        print(i)
    else:
        print_solution(OPT, keep, i-1, j)

// Main program starts here
(OPT, keep) = compute_opt_strategy(w)
print_solution(OPT, keep, n, W)
def print_solution(OPT, keep, i, j):
    if i == 0:
        return
    if keep[i, j]:
        print_solution(OPT, keep, i-1, j-w[i])
        print (i)
    else:
        print_solution(OPT, keep, i-1, j)

// Main program starts here
(OPT, keep) = compute_opt_strategy(w)
print_solution(OPT, keep, n, W)
Truck Loading Problem Example

Solution:

Maximum weight = 11
Keep box 3.
⇒ $i = 2$, $j = 12 - 7 = 5$
Keep box 2.
⇒ $i = 1$, $j = 5 - 4 = 1$
Do not keep box 1.
⇒ $i = 0$, $j = 1$
⇒ Done

Boxes 2 and 3
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<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>
<td></td>
<td></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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11

Keep box 3.

⇒ i = 2, j = 12 − 7 = 5

Keep box 2.

⇒ i = 1, j = 5 − 4 = 1

Do not keep box 1.

⇒ i = 0, j = 1

⇒ Done
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<td>0</td>
<td>0</td>
<td></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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11
Keep box 3.
⇒ i = 2, j = 12 - 7 = 5
Keep box 2.
⇒ i = 1, j = 5 - 4 = 1
Do not keep box 1.
⇒ i = 0, j = 1
⇒ Done

Boxes 2 and 3
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<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>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</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>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</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>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11
Keep box 3.
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Solution:
Maximum weight = 11
Keep box 3. \(\Rightarrow i = 2, j = 12 - 7 = 5\)
## Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<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>
<td>-</td>
<td>-</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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

### Solution:

**Maximum weight = 11**

Keep box 3. \( \Rightarrow i = 2, j = 12 - 7 = 5 \)

Keep box 2.
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11

Keep box 3. ⇒ $i = 2, j = 12 - 7 = 5$

Keep box 2. ⇒ $i = 1, j = 5 - 4 = 1$
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<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>
<td>-</td>
<td>-</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>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11

Keep box 3. ⇒ $i = 2, j = 12 - 7 = 5$

Keep box 2. ⇒ $i = 1, j = 5 - 4 = 1$

Do not keep box 1.
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11

Keep box 3. ⇒ \(i = 2, j = 12 - 7 = 5\)

Keep box 2. ⇒ \(i = 1, j = 5 - 4 = 1\)

Do not keep box 1. ⇒ \(i = 0, j = 1\)
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11

Keep box 3. ⇒ \( i = 2, j = 12 - 7 = 5 \)

Keep box 2. ⇒ \( i = 1, j = 5 - 4 = 1 \)

Do not keep box 1. ⇒ \( i = 0, j = 1 \) ⇒ Done
Truck Loading Problem Example

3 boxes with weights 9, 4, and 7. Truck capacity = 12.

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<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>
<td>-</td>
<td>-</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>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Solution:

Maximum weight = 11

Keep box 3. \( \Rightarrow i = 2, j = 12 - 7 = 5 \)
Keep box 2. \( \Rightarrow i = 1, j = 5 - 4 = 1 \)
Do not keep box 1. \( \Rightarrow i = 0, j = 1 \Rightarrow \text{Done} \)

Boxes 2 and 3
0/1 Knapsack Problem

We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.

There are \( n \) items: item \( i \) has weight \( w_i \), value \( v_i \).

Knapsack can handle a total weight of at most \( W \).

We want to put in items with maximum total value, subject to the weight restriction.

We can put all of an item in the knapsack, or none of it (fractional items have no value.).

Recall: If fractional items can be taken, greedy heuristic works:

Order items according to value per unit weight.

This does not work if we can only take whole items.

Example:

\[ W = 100 \]

Item 1:
\[ w_1 = 20, \quad v_1 = 80 \]

Item 2:
\[ w_2 = 90, \quad v_2 = 90. \]
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$. 
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$. 
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- **Recall:** If fractional items can be taken, greedy heuristic works:
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- Recall: If fractional items can be taken, greedy heuristic works:
  - Order items according to value per unit weight.
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- Recall: If fractional items can be taken, greedy heuristic works:
  - Order items according to value per unit weight.
  - This does not work if we can only take whole items.
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- Recall: If fractional items can be taken, greedy heuristic works:
  - Order items according to value per unit weight.
  - This does not work if we can only take whole items.
- Example:
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- **Recall**: If fractional items can be taken, greedy heuristic works:
  - Order items according to value per unit weight.
  - This does not work if we can only take whole items.
- **Example**:
  - $W = 100$
0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are $n$ items: item $i$ has weight $w_i$, value $v_i$.
- Knapsack can handle a total weight of at most $W$.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- **Recall:** If fractional items can be taken, greedy heuristic works:
  - Order items according to value per unit weight.
  - This does not work if we can only take whole items.
- **Example:**
  - $W = 100$
  - Item 1: $w_1 = 20$, $v_1 = 80$
  - Item 2: $w_2 = 90$, $v_2 = 90$. 
Dynamic Programming Solution

- Very similar to truck loading problem.
- Let $OPT(i, j)$ be the value of the best way to load the first $i$ items, using a knapsack with maximum capacity $j$.
- If we optimally load $i$ items using maximum capacity $j$, either we include item $i$ or we don't.
  - So: $OPT(i, j) = \max \left( v_i + OPT(i-1, j-w_i), OPT(i-1, j) \right)$;
- If $w_i > j$, we can't use item $i$, so only the second choice is available.
Dynamic Programming Solution

- Very similar to truck loading problem.

\[ \text{OPT}(i, j) = \max \left( v_i + \text{OPT}(i-1, j-w_i), \text{OPT}(i-1, j) \right) ; \]

If \( w_i > j \), we can't use item \( i \), so only the second choice is available.
Dynamic Programming Solution

- Very similar to truck loading problem.
- Let $\text{OPT}(i, j)$ be the value of the best way to load the first $i$ items, using a knapsack with maximum capacity $j$. 
Dynamic Programming Solution

- Very similar to truck loading problem.
- Let $OPT(i, j)$ be the value of the best way to load the first $i$ items, using a knapsack with maximum capacity $j$.
- If we optimally load $i$ items using maximum capacity $j$ either we include item $i$ or we don’t.
Dynamic Programming Solution

- Very similar to truck loading problem.
- Let $\text{OPT}(i, j)$ be the value of the best way to load the first $i$ items, using a knapsack with maximum capacity $j$.
- If we optimally load $i$ items using maximum capacity $j$ either we include item $i$ or we don’t. So:

$$\text{OPT}(i, j) = \max (v_i + \text{OPT}(i - 1, j - w_i), \text{OPT}(i - 1, j));$$
Dynamic Programming Solution

- Very similar to truck loading problem.
- Let $\text{OPT}(i, j)$ be the value of the best way to load the first $i$ items, using a knapsack with maximum capacity $j$.
- If we optimally load $i$ items using maximum capacity $j$ either we include item $i$ or we don’t. So:

$$\text{OPT}(i, j) = \max (v_i + \text{OPT}(i - 1, j - w_i), \text{OPT}(i - 1, j));$$

- If $w_i > j$, we can’t use item $i$, so only the second choice is available.
Dynamic Programming Solution

- Very similar to truck loading problem.
- Let \( \text{OPT}(i, j) \) be the value of the best way to load the first \( i \) items, using a knapsack with maximum capacity \( j \).
- If we optimally load \( i \) items using maximum capacity \( j \) either we include item \( i \) or we don’t. So:

\[
\text{OPT}(i, j) = \max (v_i + \text{OPT}(i - 1, j - w_i), \text{OPT}(i - 1, j))
\]

- If \( w_i > j \), we can’t use item \( i \), so only the second choice is available.
Specifying the Solution

1. Subproblem domain \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. Function /Memoization table definition: \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) items into a knapsack with maximum capacity \( j \).

3. Goal: \( \text{OPT}(n, W) \)

4. Initial values:
   \( \text{OPT}(i, 0) = 0 \) for all \( i \geq 0 \)
   \( \text{OPT}(0, j) = 0 \) for all \( j \geq 0 \)

5. Recurrence:
   \[
   \text{OPT}(i, j) = \begin{cases} 
   v_i + \text{OPT}(i-1, j-w_i) & \text{if } w_i \leq j \\
   \text{OPT}(i-1, j) & \text{if } w_i > j
   \end{cases}
   \]
Specifying the Solution

1. Subproblem domain \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)
Specifying the Solution

1. **Subproblem domain** \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. **Function /Memoization table definition:** \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) items into a knapsack with maximum capacity \( j \).
Specifying the Solution

1. Subproblem domain \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. Function /Memoization table definition: \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) items into a knapsack with maximum capacity \( j \).

3. Goal: \( \text{OPT}(n, W) \)
Specifying the Solution

1. **Subproblem domain** \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. **Function /Memoization table definition**: \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) items into a knapsack with maximum capacity \( j \).

3. **Goal**: \( \text{OPT}(n, W) \)

4. **Initial values**:

   \[
   \begin{align*}
   \text{OPT}(i, 0) &= 0 \quad \text{for all } i \geq 0 \\
   \text{OPT}(0, j) &= 0 \quad \text{for all } j \geq 0
   \end{align*}
   \]
Specifying the Solution

1. Subproblem domain \( \{0, \ldots, n\} \times \{0, \ldots, W\} \)

2. Function /Memoization table definition: \( \text{OPT}(i, j) \) is the value of the best way of loading a subset of the first \( i \) items into a knapsack with maximum capacity \( j \).

3. Goal: \( \text{OPT}(n, W) \)

4. Initial values:

\[
\begin{align*}
\text{OPT}(i, 0) &= 0 \quad \text{for all } i \geq 0 \\
\text{OPT}(0, j) &= 0 \quad \text{for all } j \geq 0
\end{align*}
\]

5. Recurrence:

\[
\text{OPT}(i, j) = \begin{cases} 
\max (v_i + \text{OPT}(i - 1, j - w_i), \text{OPT}(i - 1, j)) & \text{if } w_i \leq j \\
\text{OPT}(i - 1, j) & \text{if } w_i > j
\end{cases}
\]
Pseudocode for DP Solution to 0/1 Knapsack Problem

def compute_opt_strategy(w,v):
    for i = 0 to n:  OPT[i,0] = 0
    for j = 0 to W:  OPT[0,j] = 0
    for i = 1 to n:
        for j = 1 to W:
            if (w[i] > j) or (v[i] + OPT[i-1,j-w[i]] <= OPT[i-1,j])
                OPT[i,j] = OPT[i-1,j]
                keep[i,j] = False
            else:
                OPT[i,j] = v[i] + OPT[i-1,j-w[i]]
                keep[i,j] = True
    return (OPT,keep)
Pseudocode for DP Solution to 0/1 Knapsack Problem
[continued]

def print_solution(OPT, keep, i, j):
    if i == 0: return
    if keep[i, j]:
        print_solution(OPT, keep, i-1, j-w[i])
        print (i)
    else:
        print_solution(OPT, keep, i-1, j)

// Main program starts here
(OPT, keep) = compute_opt_strategy(w, v)
print_solution(OPT, keep, n, W)
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative: $(A \times B) \times C = A \times (B \times C)$

3. The parenthesizing may effect the efficiency.

$A$: $p \times q$

$B$: $q \times r$

$C$: $r \times s$

$(A \times B) \times C$: Number of scalar multiplications is: $p \cdot q \cdot r + p \cdot r \cdot s$

$A \times (B \times C)$: Number of scalar multiplications is: $q \cdot r \cdot s + p \cdot q \cdot s$
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative:

   $$(A \times B) \times C = A \times (B \times C)$$
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative:

\[
(A \times B) \times C = A \times (B \times C)
\]

3. The parenthesizing may effect the efficiency.
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative:

   $$(A \times B) \times C = A \times (B \times C)$$

3. The parenthesizing may effect the efficiency.

   $A$: $p \times q$
   $B$: $q \times r$
   $C$: $r \times s$
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative:

$$ (A \times B) \times C = A \times (B \times C) $$

3. The parenthesizing may effect the efficiency.

- $A$: $p \times q$
- $A \times B$: $p \times r$
- $B$: $q \times r$
- $B \times C$: $q \times s$
- $C$: $r \times s$
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative:

$$(A \times B) \times C = A \times (B \times C)$$

3. The parenthesizing may affect the efficiency.

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$p \times q$</td>
<td>$p \times r$</td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>$q \times r$</td>
<td>$q \times s$</td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>$r \times s$</td>
<td>$r \times s$</td>
<td></td>
</tr>
</tbody>
</table>

$(A \times B) \times C$: Number of scalar multiplications is:

$$p \cdot q \cdot r + p \cdot r \cdot s$$
Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires $q$ scalar multiplications).

2. Matrix multiplication is associative:

\[(A \times B) \times C = A \times (B \times C)\]

3. The parenthesizing may effect the efficiency.

\[A: \ p \times q \quad A \times B: \ p \times r\]
\[B: \ q \times r \quad B \times C: \ q \times s\]
\[C: \ r \times s\]

\[(A \times B) \times C: \text{Number of scalar multiplications is:}\]
\[p \cdot q \cdot r + p \cdot r \cdot s\]

\[A \times (B \times C): \text{Number of scalar multiplications is:}\]
\[q \cdot r \cdot s + p \cdot q \cdot s\]
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

$$(A \times B) \times C:$$ Cost is $40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$

$A \times (B \times C):$ Cost is $2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000 = 14,000$

$A \times (B \times C)$ is considerably more efficient.

Parenthesization Matters
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$. 
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: 
  
  Cost is $40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 80000 + 200000 = 280000$.

$(A \times (B \times C))$ is considerably more efficient.
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is

$$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50$$
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is

$$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000$$
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is

\[
40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000
\]
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is
  
  $$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$$

- $A \times (B \times C)$:
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is
  \[40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000\]

- $A \times (B \times C)$: Cost is
Example

Suppose \( A \) is \( 40 \times 2 \), \( B \) is \( 2 \times 100 \), and \( C \) is \( 100 \times 50 \).

- \((A \times B) \times C\): Cost is

\[
40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000
\]

- \( A \times (B \times C)\): Cost is

\[
2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50
\]
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is

$$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$$

- $A \times (B \times C)$: Cost is

$$2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000$$
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is

  $$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$$

- $A \times (B \times C)$: Cost is

  $$2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000 = 14,000$$
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

- $(A \times B) \times C$: Cost is
  
  \[
  40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000
  \]

- $A \times (B \times C)$: Cost is
  
  \[
  2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000 = 14,000
  \]

$A \times (B \times C)$ is considerably more efficient
Example

Suppose $A$ is $40 \times 2$, $B$ is $2 \times 100$, and $C$ is $100 \times 50$.

1. $(A \times B) \times C$: Cost is

   $$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$$

2. $A \times (B \times C)$: Cost is

   $$2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000 = 14,000$$

$A \times (B \times C)$ is considerably more efficient

Parenthesization Matters
Optimal Matrix Chain Multiplication problem

Given \( n \) matrices: \( A_1, \ldots, A_n \).

Matrix \( A_i \) is \( d_i-1 \times d_i \).

What is the most efficient way of grouping (i.e., parenthesizing) to compute \( A_1 \times \cdots \times A_n \)?

Most efficient means fewest scalar multiplications.

Example:

\[
\begin{align*}
A_1 &: 10 \times 15 \\
A_2 &: 15 \times 5 \\
A_3 &: 5 \times 60 \\
A_4 &: 60 \times 100 \\
A_5 &: 100 \times 20 \\
A_6 &: 20 \times 40 \\
A_7 &: 40 \times 47
\end{align*}
\]

\[
d_0 = 10 \\
d_1 = 15 \\
d_2 = 5 \\
d_3 = 60 \\
d_4 = 100 \\
d_5 = 20 \\
d_6 = 40 \\
d_7 = 47
\]

As we will see, the optimal cost is 56,500 scalar multiplications.

The optimal grouping is:

\[
((A_1 \times A_2) \times ((A_3 \times A_4) \times ((A_5 \times A_6) \times A_7)))
\]
Optimal Matrix Chain Multiplication problem

- Given $n$ matrices: $A_1, \ldots, A_n$. 
Optimal Matrix Chain Multiplication problem

- Given $n$ matrices: $A_1, \ldots, A_n$.
- Matrix $A_i$ is $d_{i-1} \times d_i$. 

As we will see, the optimal cost is 56,500 scalar multiplications.

The optimal grouping is:

$$\left((A_1 \times A_2) \times \left(\left((A_3 \times A_4) \times (A_5 \times A_6)\right) \times A_7\right)\right)$$
Optimal Matrix Chain Multiplication problem

- Given $n$ matrices: $A_1, \ldots, A_n$.
- Matrix $A_i$ is $d_{i-1} \times d_i$.
- What is the most efficient way of grouping (i.e., parenthesizing) to compute $A_1 \times \cdots \times A_n$?
Optimal Matrix Chain Multiplication problem

- Given \( n \) matrices: \( A_1, \ldots, A_n \).
- Matrix \( A_i \) is \( d_{i-1} \times d_i \).
- What is the most efficient way of grouping (i.e., parenthesizing) to compute \( A_1 \times \cdots \times A_n \)?
  - Most efficient means fewest scalar multiplications
Optimal Matrix Chain Multiplication problem

- Given $n$ matrices: $A_1, \ldots, A_n$.
- Matrix $A_i$ is $d_{i-1} \times d_i$.
- What is the most efficient way of grouping (i.e., parenthesizing) to compute $A_1 \times \cdots \times A_n$?
  
  - Most efficient means fewest scalar multiplications

Example:
Optimal Matrix Chain Multiplication problem

- Given $n$ matrices: $A_1, \ldots, A_n$.
- Matrix $A_i$ is $d_{i-1} \times d_i$.
- What is the most efficient way of grouping (i.e., parenthesizing) to compute $A_1 \times \cdots \times A_n$?

  - Most efficient means fewest scalar multiplications

Example:

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$10 \times 15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2$</td>
<td>$15 \times 5$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>$5 \times 60$</td>
</tr>
<tr>
<td>$A_4$</td>
<td>$60 \times 100$</td>
</tr>
<tr>
<td>$A_5$</td>
<td>$100 \times 20$</td>
</tr>
<tr>
<td>$A_6$</td>
<td>$20 \times 40$</td>
</tr>
<tr>
<td>$A_7$</td>
<td>$40 \times 47$</td>
</tr>
</tbody>
</table>
Optimal Matrix Chain Multiplication problem

- Given \( n \) matrices: \( A_1, \ldots, A_n \).
- Matrix \( A_i \) is \( d_{i-1} \times d_i \).
- What is the most efficient way of grouping (i.e., parenthesizing) to compute \( A_1 \times \cdots \times A_n \)?

  - Most efficient means fewest scalar multiplications

**Example:**

\[
\begin{array}{l}
A_1 : 10 \times 15 \\
A_2 : 15 \times 5 \\
A_3 : 5 \times 60 \\
A_4 : 60 \times 100 \\
A_5 : 100 \times 20 \\
A_6 : 20 \times 40 \\
A_7 : 40 \times 47
\end{array}
\]

\[
\begin{array}{l}
d_0 = 10 \\
d_1 = 15 \\
d_2 = 5 \\
d_3 = 60 \\
d_4 = 100 \\
d_5 = 20 \\
d_6 = 40 \\
d_7 = 47
\end{array}
\]

- As we will see, the optimal cost is 56,500 scalar multiplications
Optimal Matrix Chain Multiplication problem

- Given \( n \) matrices: \( A_1, \ldots, A_n \).
- Matrix \( A_i \) is \( d_{i-1} \times d_i \).
- What is the most efficient way of grouping (i.e., parenthesizing) to compute \( A_1 \times \cdots \times A_n \)?

- Most efficient means fewest scalar multiplications

**Example:**

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( d_0 )</th>
<th>( d_1 )</th>
<th>( d_2 )</th>
<th>( d_3 )</th>
<th>( d_4 )</th>
<th>( d_5 )</th>
<th>( d_6 )</th>
<th>( d_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10 \times 15 )</td>
<td>10</td>
<td>15</td>
<td>5</td>
<td>60</td>
<td>100</td>
<td>20</td>
<td>40</td>
<td>47</td>
</tr>
<tr>
<td>( 15 \times 5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 5 \times 60 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 60 \times 100 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 100 \times 20 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 20 \times 40 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 40 \times 47 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- As we will see, the optimal cost is 56,500 scalar multiplications
Optimal Matrix Chain Multiplication problem

- Given \( n \) matrices: \( A_1, \ldots, A_n \).
- Matrix \( A_i \) is \( d_{i-1} \times d_i \).
- What is the most efficient way of grouping (i.e., parenthesizing) to compute \( A_1 \times \cdots \times A_n \)?

  - Most efficient means fewest scalar multiplications

**Example:**

\[
\begin{align*}
A_1 &: 10 \times 15 \\
A_2 &: 15 \times 5 \\
A_3 &: 5 \times 60 \\
A_4 &: 60 \times 100 \\
A_5 &: 100 \times 20 \\
A_6 &: 20 \times 40 \\
A_7 &: 40 \times 47
\end{align*}
\]

\[
\begin{align*}
d_0 &= 10 \\
d_1 &= 15 \\
d_2 &= 5 \\
d_3 &= 60 \\
d_4 &= 100 \\
d_5 &= 20 \\
d_6 &= 40 \\
d_7 &= 47
\end{align*}
\]

- As we will see, the optimal cost is 56,500 scalar multiplications
- The optimal grouping is:
Optimal Matrix Chain Multiplication problem

- Given $n$ matrices: $A_1, \ldots, A_n$.
- Matrix $A_i$ is $d_{i-1} \times d_i$.
- What is the most efficient way of grouping (i.e., parenthesizing) to compute $A_1 \times \cdots \times A_n$?
  - Most efficient means fewest scalar multiplications

**Example:**

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 \times 15$</td>
<td>$15 \times 5$</td>
<td>$5 \times 60$</td>
<td>$60 \times 100$</td>
<td>$100 \times 20$</td>
<td>$20 \times 40$</td>
<td>$40 \times 47$</td>
</tr>
</tbody>
</table>

- Most efficient means fewest scalar multiplications

- As we will see, the optimal cost is 56,500 scalar multiplications
- The optimal grouping is:

$$(A_1 \times A_2) \times (((A_3 \times A_4) \times A_5) \times A_6) \times A_7)$$
Dynamic Programming Solution

A_i \times \cdots \times A_j

Define $M(i,j) = \text{the number of multiplications required to compute the product } A_i \times \cdots \times A_j \text{ using the best possible grouping}$

The final multiplication will consist of a left subchain and a right subchain.

Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$

The cost of computing the left subchain is $M(i,k)$

The cost of computing the right subchain is $M(k+1,j)$

Cost of final multiplication:

- $(A_i \times \cdots \times A_k)$ is $d_i - 1 \times d_k$
- $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$

Cost of multiplication is $d_i - 1 \times d_k \times d_j$

Total cost is $M(i,k) + M(k+1,j) + d_i - 1 \times d_k \times d_j$.
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$

Define $M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$

The final multiplication will consist of a left subchain and a right subchain.

Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$

The cost of computing the left subchain is $M(i, k)$

The cost of computing the right subchain is $M(k+1, j)$

Cost of final multiplication:

$(A_i \times \cdots \times A_k)$ is $d_i - 1 \times d_k$

$(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$

Cost of multiplication is $d_i - 1 \times d_k \times d_j$

Total cost is $M(i, k) + M(k+1, j) + d_i - 1 \times d_k \times d_j$. 
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
Dynamic Programming Solution

- Subproblems: optimally multiplying chains \( A_i \times \cdots \times A_j \)
- Define \( M(i, j) \) = the number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at \( A_k \): \( (A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j) \)
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) = \text{the number of multiplications required to compute the product } A_i \times \cdots \times A_j \text{ using the best possible grouping}$
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k + 1, j)$
Dynamic Programming Solution

- Subproblems: optimally multiplying chains \( A_i \times \cdots \times A_j \)

- Define \( M(i,j) = \) the number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) using the best possible grouping

- The final multiplication will consist of a left subchain and a right subchain.

- Suppose the left subchain stops at \( A_k \): \((A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)\)
  - The cost of computing the left subchain is \( M(i,k) \)
  - The cost of computing the right subchain is \( M(k+1,j) \)
  - Cost of final multiplication:

\[
\begin{align*}
\text{Cost} &= M(i,k) + M(k+1,j) + d_i - 1 \times d_k \times d_j \\
\text{Choose the best index } k: \quad M(i,j) &= \min_{i \leq k \leq j-1} (M(i,k) + M(k+1,j) + d_i - 1 \times d_k \times d_j)
\end{align*}
\]
Dynamic Programming Solution

- Subproblems: optimally multiplying chains \( A_i \times \cdots \times A_j \)
- Define \( M(i, j) = \) the number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at \( A_k \): \((A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)\)
  - The cost of computing the left subchain is \( M(i, k) \)
  - The cost of computing the right subchain is \( M(k+1, j) \)
  - Cost of final multiplication:
    - \((A_i \times \cdots \times A_k)\) is \( d_{i-1} \times d_k \)
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k+1, j)$
  - Cost of final multiplication:
    - $(A_i \times \cdots \times A_k)$ is $d_{i-1} \times d_k$
    - $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j)$ = the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k + 1, j)$
  - Cost of final multiplication:
    - $(A_i \times \cdots \times A_k)$ is $d_{i-1} \times d_k$
    - $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$
    - Cost of multiplication is $d_{i-1}d_kd_j$
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k+1, j)$
  - Cost of final multiplication:
    - $(A_i \times \cdots \times A_k)$ is $d_{i-1} \times d_k$
    - $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$
    - Cost of multiplication is $d_{i-1}d_kd_j$
  - Total cost is $M(i, k) + M(k+1, j) + d_{i-1}d_kd_j$. 
Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i, j) = \text{the number of multiplications required to compute the product } A_i \times \cdots \times A_j \text{ using the best possible grouping}$
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k+1, j)$
  - Cost of final multiplication:
    - $(A_i \times \cdots \times A_k)$ is $d_{i-1} \times d_k$
    - $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$
    - Cost of multiplication is $d_{i-1}d_kd_j$
  - Total cost is $M(i, k) + M(k+1, j) + d_{i-1}d_kd_j$.
- Choose the best index $k$: 
Dynamic Programming Solution

- **Subproblems:** optimally multiplying chains $A_i \times \cdots \times A_j$

- Define $M(i, j) = \text{the number of multiplications required to compute the product } A_i \times \cdots \times A_j \text{ using the best possible grouping}$

- The final multiplication will consist of a left subchain and a right subchain.

- Suppose the left subchain stops at $A_k$: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k + 1, j)$
  - Cost of final multiplication:
    - $(A_i \times \cdots \times A_k)$ is $d_{i-1} \times d_k$
    - $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$
    - Cost of multiplication is $d_{i-1}d kd_j$
    - Total cost is $M(i, k) + M(k + 1, j) + d_{i-1}d kd_j$.

- Choose the best index $k$:
  \[
  M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k + 1, j) + d_{i-1}d kd_j) 
  \]
Specifying the Solution

1. Subproblem domain
   \{(i, j) : 1 \leq i \leq j \leq n\}

2. Function / Memoization table definition:
   \(M(i, j)\) is the minimum number of multiplications required to compute the product
   \(A^i \times \cdots \times A^j\) (using the best possible grouping).

3. Goal:
   \(M(1, n)\)

4. Initial values:
   \(M(i, i) = 0\)

5. Recurrence:
   \(M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k+1, j) + d_{i-1}d_kd_{j+1})\)

Note:
▶ The fact that \(M(i, i+1) = d_{i-1}d_id_{i+1}\) does not need to be stated
  as an initial condition.
▶ It follows from the recurrence equation that
  \(M(i, i+1) = M(i, i) + M(i+1, i+1) + d_{i-1}d_id_{i+1} = 0 + 0 + d_{i-1}d_id_{i+1} = d_{i-1}d_id_{i+1}\). 
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)
Specifying the Solution

1. **Subproblem domain** \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. **Function / Memoization table definition:** \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)
Specifying the Solution

1. **Subproblem domain** \{ (i, j) : 1 \leq i \leq j \leq n \} 

2. **Function / Memoization table definition**: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. **Goal**: \( M(1, n) \)

4. **Initial values**: \( M(i, i) = 0 \)
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)

4. Initial values: \( M(i, i) = 0 \)

5. Recurrence:
\[
M(i, j) = \min_{i \leq k \leq j - 1} (M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j)
\]
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)

4. Initial values: \( M(i, i) = 0 \)

5. Recurrence:

\[
M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j)
\]

Note:
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)

4. Initial values: \( M(i, i) = 0 \)

5. Recurrence:

\[
M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k+1, j) + d_{i-1}d_kd_j)
\]

Note:

- The fact that \( M(i, i+1) = d_{i-1}d_id_{i+1} \) does not need to be stated as an initial condition.
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)

4. Initial values: \( M(i, i) = 0 \)

5. Recurrence:

\[
M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j)
\]

Note:

- The fact that \( M(i, i + 1) = d_{i-1}d_id_{i+1} \) does not need to be stated as an initial condition.

- It follows from the recurrence equation that
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)

4. Initial values: \( M(i, i) = 0 \)

5. Recurrence:

\[
M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k+1, j) + d_{i-1}d_kd_j)
\]

Note:

- The fact that \( M(i, i + 1) = d_{i-1}d_id_{i+1} \) does not need to be stated as an initial condition.

- It follows from the recurrence equation that

\[
M(i, i + 1) = M(i, i) + M(i + 1, i + 1) + d_{i-1}d_id_{i+1}
\]
Specifying the Solution

1. **Subproblem domain** \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. **Function / Memoization table definition:** \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. **Goal:** \( M(1, n) \)

4. **Initial values:** \( M(i, i) = 0 \)

5. **Recurrence:**

\[
M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j)
\]

**Note:**

- The fact that \( M(i, i + 1) = d_{i-1}d_id_{i+1} \) does not need to be stated as an initial condition.
- It follows from the recurrence equation that

\[
M(i, i + 1) = M(i, i) + M(i + 1, i + 1) + d_{i-1}d_id_{i+1} = 0 + 0 + d_{i-1}d_id_{i+1}
\]
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq j \leq n\} \)

2. Function / Memoization table definition: \( M(i, j) \) is the minimum number of multiplications required to compute the product \( A_i \times \cdots \times A_j \) (using the best possible grouping).

3. Goal: \( M(1, n) \)

4. Initial values: \( M(i, i) = 0 \)

5. Recurrence:

\[
M(i, j) = \min_{i \leq k \leq j-1} (M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j)
\]

Note:

- The fact that \( M(i, i + 1) = d_{i-1}d_id_{i+1} \) does not need to be stated as an initial condition.

- It follows from the recurrence equation that

\[
M(i, i + 1) = M(i, i) + M(i + 1, i + 1) + d_{i-1}d_id_{i+1}
\]
\[
= 0 + 0 + d_{i-1}d_id_{i+1}
\]
\[
= d_{i-1}d_id_{i+1}.
\]
The input is just the array of dimensions: \( d_0, \ldots, d_n \).

We need to compute the chain costs in increasing order of the chain lengths. (The length of the chain \( A_i \times \cdots \times A_j \) is \( j - i + 1 \).)

```python
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            M[i,j] = +\infty
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
    return M
```
Pseudocode

- The input is just the array of dimensions: $d_0, \ldots, d_n$. 

```python
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            M[i,j] = ∞
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
    return M
```
Pseudocode

- The input is just the array of dimensions: $d_0, \ldots, d_n$.
- We need to compute the chain costs in increasing order of the chain lengths. (The length of the chain $A_i \times \cdots \times A_j$ is $j - i + 1$.)

```python
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
    return M
```
Pseudocode

- The input is just the array of dimensions: $d_0, \ldots, d_n$.
- We need to compute the chain costs in increasing order of the chain lengths. (The length of the chain $A_i \times \cdots \times A_j$ is $j - i + 1$.)

```python
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            M[i,j] = +\infty
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
    return M
```
Computing the chains
Computing the chains

- Augment the preceding pseudocode by storing the best split for each \((i, j)\) in an array \(S\).
Computing the chains

- Augment the preceding pseudocode by storing the best split for each \((i, j)\) in an array \(S\).
- \(S[i, j] = k\) when the best split for \(A_i \times \cdots \times A_j\) is
  \[
  (A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)
  \]
Computing the chains

- Augment the preceding pseudocode by storing the best split for each \((i, j)\) in an array \(S\).
- \(S[i, j] = k\) when the best split for \(A_i \times \cdots \times A_j\) is
  \[(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)\]

```python
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            M[i,j] = +\infty
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
                    S[i,j] = k
    return M, S
```
Solution to our example

Optimal value is 56500

Optimal grouping is:

\[ A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7 \]
Solution to our example

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$10 \times 15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2$</td>
<td>$15 \times 5$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>$5 \times 60$</td>
</tr>
<tr>
<td>$A_4$</td>
<td>$60 \times 100$</td>
</tr>
<tr>
<td>$A_5$</td>
<td>$100 \times 20$</td>
</tr>
<tr>
<td>$A_6$</td>
<td>$20 \times 40$</td>
</tr>
<tr>
<td>$A_7$</td>
<td>$40 \times 47$</td>
</tr>
</tbody>
</table>
Solution to our example

A_1 : 10 \times 15
A_2 : 15 \times 5
A_3 : 5 \times 60
A_4 : 60 \times 100
A_5 : 100 \times 20
A_6 : 20 \times 40
A_7 : 40 \times 47

d_0 = 10
d_1 = 15
d_2 = 5
d_3 = 60
d_4 = 100
d_5 = 20
d_6 = 40
d_7 = 47

Optimal value is 56500
Optimal grouping is:
A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7
### Solution to our example

**A**

- **A**<sub>1</sub>: 10 × 15
- **A**<sub>2</sub>: 15 × 5
- **A**<sub>3</sub>: 5 × 60
- **A**<sub>4</sub>: 60 × 100
- **A**<sub>5</sub>: 100 × 20
- **A**<sub>6</sub>: 20 × 40
- **A**<sub>7</sub>: 40 × 47

**d**

- **d**<sub>0</sub> = 10
- **d**<sub>1</sub> = 15
- **d**<sub>2</sub> = 5
- **d**<sub>3</sub> = 60
- **d**<sub>4</sub> = 100
- **d**<sub>5</sub> = 20
- **d**<sub>6</sub> = 40
- **d**<sub>7</sub> = 47

**Optimal value is 56500**

**Optimal grouping is:**

\[
A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7
\]
### Solution to our example

| \( A_j \) | \( 10 \times 15 \) |
| \( A_2 \) | \( 15 \times 5 \) |
| \( A_3 \) | \( 5 \times 60 \) |
| \( A_4 \) | \( 60 \times 100 \) |
| \( A_5 \) | \( 100 \times 20 \) |
| \( A_6 \) | \( 20 \times 40 \) |
| \( A_7 \) | \( 40 \times 47 \) |

| \( d_0 \) | 10 |
| \( d_1 \) | 15 |
| \( d_2 \) | 5 |
| \( d_3 \) | 60 |
| \( d_4 \) | 100 |
| \( d_5 \) | 20 |
| \( d_6 \) | 40 |
| \( d_7 \) | 47 |

---

\[
\begin{array}{cccccccc}
  & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 1 & 0 & 750 & 3750 & 35750 & 41750 & 46750 & 56500 \\
 2 & 0 & 4500 & 37500 & 41500 & 47000 & 56925 \\
 3 & 0 & 30000 & 40000 & 44000 & 53400 \\
 4 & 0 & 120000 & 168000 & 214000 \\
 5 & 0 & 80000 & 131600 \\
 6 & 0 & 37600 \\
 7 & 0 & & & & & & \\
\end{array}
\]

Optimal value is 56500
## Solution to our example

### Task:

Given the dimensions of the objects and their respective costs, determine the optimal grouping to maximize the total cost.

### Dimensions:

- $A_1: 10 \times 15$
- $A_2: 15 \times 5$
- $A_3: 5 \times 60$
- $A_4: 60 \times 100$
- $A_5: 100 \times 20$
- $A_6: 20 \times 40$
- $A_7: 40 \times 47$

### Costs:

- $d_0 = 10$
- $d_1 = 15$
- $d_2 = 5$
- $d_3 = 60$
- $d_4 = 100$
- $d_5 = 20$
- $d_6 = 40$
- $d_7 = 47$

### Solution:

<table>
<thead>
<tr>
<th></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>0</td>
<td>0</td>
<td>750</td>
<td>3750</td>
<td>35750</td>
<td>41750</td>
<td>46750</td>
<td>56500</td>
</tr>
<tr>
<td>1</td>
<td>---</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4500</td>
<td>37500</td>
<td>41500</td>
<td>47000</td>
<td>56925</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>30000</td>
<td>40000</td>
<td>44000</td>
<td>53400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>120000</td>
<td>168000</td>
<td>214000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>80000</td>
<td>131600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>37600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Optimal value is 56500

### Optimal grouping is:

Optimal grouping is:

- $A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7$
Solution to our example

<table>
<thead>
<tr>
<th>Item</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$10 \times 15$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$15 \times 5$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>$5 \times 60$</td>
</tr>
<tr>
<td>$A_4$</td>
<td>$60 \times 100$</td>
</tr>
<tr>
<td>$A_5$</td>
<td>$100 \times 20$</td>
</tr>
<tr>
<td>$A_6$</td>
<td>$20 \times 40$</td>
</tr>
<tr>
<td>$A_7$</td>
<td>$40 \times 47$</td>
</tr>
</tbody>
</table>

$d_0 = 10$
$d_1 = 15$
$d_2 = 5$
$d_3 = 60$
$d_4 = 100$
$d_5 = 20$
$d_6 = 40$
$d_7 = 47$

Optimal value is 56500

Optimal grouping is:

$$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7$$
Solution to our example

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 \times 15$</td>
<td>$15 \times 5$</td>
<td>$5 \times 60$</td>
<td>$60 \times 100$</td>
<td>$100 \times 20$</td>
<td>$20 \times 40$</td>
<td>$40 \times 47$</td>
</tr>
</tbody>
</table>

$d_0 = 10$
$d_1 = 15$
$d_2 = 5$
$d_3 = 60$
$d_4 = 100$
$d_5 = 20$
$d_6 = 40$
$d_7 = 47$

Optimal value is 56500

Optimal grouping is:

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7$

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Solution to our example

<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>A_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15</td>
<td>5</td>
<td>60</td>
<td>100</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>5</td>
<td>60</td>
<td>100</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>60</td>
<td>100</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>60</td>
<td>100</td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ A_1 : 10 \times 15 \]
\[ A_2 : 15 \times 5 \]
\[ A_3 : 5 \times 60 \]
\[ A_4 : 60 \times 100 \]
\[ A_5 : 100 \times 20 \]
\[ A_6 : 20 \times 40 \]
\[ A_7 : 40 \times 47 \]

\[ d_0 = 10 \]
\[ d_1 = 15 \]
\[ d_2 = 5 \]
\[ d_3 = 60 \]
\[ d_4 = 100 \]
\[ d_5 = 20 \]
\[ d_6 = 40 \]
\[ d_7 = 47 \]

Optimal value is 56500

Optimal grouping is:
\[(A_1 \times A_2) \times (A_3 \times A_4 \times A_5 \times A_6 \times A_7)\]
Solution to our example

\[ A_1 : 10 \times 15 \]
\[ A_2 : 15 \times 5 \]
\[ A_3 : 5 \times 60 \]
\[ A_4 : 60 \times 100 \]
\[ A_5 : 100 \times 20 \]
\[ A_6 : 20 \times 40 \]
\[ A_7 : 40 \times 47 \]

\[ d_0 = 10 \]
\[ d_1 = 15 \]
\[ d_2 = 5 \]
\[ d_3 = 60 \]
\[ d_4 = 100 \]
\[ d_5 = 20 \]
\[ d_6 = 40 \]
\[ d_7 = 47 \]

Optimal value is 56500

Optimal grouping is:

\[ (A_1 \times A_2) \times (A_3 \times A_4 \times A_5 \times A_6 \times A_7) \]
Solution to our example

Optimal value is 56500

Optimal grouping is:

\[(A_1 \times A_2) \times ((A_3 \times A_4 \times A_5 \times A_6) \times A_7)\]
Solution to our example

Optimal value is 56500

Optimal grouping is:

\[(A_1 \times A_2) \times ((A_3 \times A_4 \times A_5 \times A_6) \times A_7)\]
### Solution to our example

#### 1. \( A_1 \) : 10 \( \times \) 15

#### 2. \( A_2 \) : 15 \( \times \) 5

#### 3. \( A_3 \) : 5 \( \times \) 60

#### 4. \( A_4 \) : 60 \( \times \) 100

#### 5. \( A_5 \) : 100 \( \times \) 20

#### 6. \( A_6 \) : 20 \( \times \) 40

#### 7. \( A_7 \) : 40 \( \times \) 47

#### \( d_0 = 10 \)

#### \( d_1 = 15 \)

#### \( d_2 = 5 \)

#### \( d_3 = 60 \)

#### \( d_4 = 100 \)

#### \( d_5 = 20 \)

#### \( d_6 = 40 \)

#### \( d_7 = 47 \)

### Table:

<table>
<thead>
<tr>
<th>( 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>1</td>
<td>0</td>
<td>750</td>
<td>3750</td>
<td>35750</td>
<td>41750</td>
<td>46750</td>
<td>56500</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4500</td>
<td>37500</td>
<td>41500</td>
<td>47000</td>
<td>56925</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>30000</td>
<td>40000</td>
<td>44000</td>
<td>53400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>120000</td>
<td>168000</td>
<td>214000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>80000</td>
<td>131600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>37600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Optimal value is 56500

### Optimal grouping is:

\[
(A_1 \times A_2) \times (((A_3 \times A_4 \times A_5) \times A_6) \times A_7)
\]
### Solution to our example

- $A_1 : 10 \times 15$
- $A_2 : 15 \times 5$
- $A_3 : 5 \times 60$
- $A_4 : 60 \times 100$
- $A_5 : 100 \times 20$
- $A_6 : 20 \times 40$
- $A_7 : 40 \times 47$

\[ d_0 = 10 \]
\[ d_1 = 15 \]
\[ d_2 = 5 \]
\[ d_3 = 60 \]
\[ d_4 = 100 \]
\[ d_5 = 20 \]
\[ d_6 = 40 \]
\[ d_7 = 47 \]

### Optimal value

Optimal value is 56500

Optimal grouping is:

\[
(A_1 \times A_2) \times (((A_3 \times A_4 \times A_5) \times A_6) \times A_7)
\]
Solution to our example

\[ \begin{align*}
A_1 & : 10 \times 15 \\
A_2 & : 15 \times 5 \\
A_3 & : 5 \times 60 \\
A_4 & : 60 \times 100 \\
A_5 & : 100 \times 20 \\
A_6 & : 20 \times 40 \\
A_7 & : 40 \times 47 \\
\end{align*} \]

\[ d_0 = 10 \\
 d_1 = 15 \\
 d_2 = 5 \\
 d_3 = 60 \\
 d_4 = 100 \\
 d_5 = 20 \\
 d_6 = 40 \\
 d_7 = 47 \]

Optimal value is 56500

Optimal grouping is:

\[(A_1 \times A_2) \times (((A_3 \times A_4) \times A_5) \times A_6) \times A_7)\]
Solution to our example

<table>
<thead>
<tr>
<th></th>
<th>$A_1$: $10 \times 15$</th>
<th>$A_2$: $15 \times 5$</th>
<th>$A_3$: $5 \times 60$</th>
<th>$A_4$: $60 \times 100$</th>
<th>$A_5$: $100 \times 20$</th>
<th>$A_6$: $20 \times 40$</th>
<th>$A_7$: $40 \times 47$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>$= 10$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_1$</td>
<td>$= 15$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_2$</td>
<td>$= 5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_3$</td>
<td>$= 60$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_4$</td>
<td>$= 100$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_5$</td>
<td>$= 20$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_6$</td>
<td>$= 40$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d_7$</td>
<td>$= 47$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></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>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>750</td>
<td>4500</td>
<td>30000</td>
<td>120000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3750</td>
<td>37500</td>
<td>40000</td>
<td>168000</td>
<td>80000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>35750</td>
<td>41500</td>
<td>44000</td>
<td>214000</td>
<td>131600</td>
<td>37600</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>41750</td>
<td>47000</td>
<td>53400</td>
<td>214000</td>
<td>131600</td>
<td>37600</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>46750</td>
<td>56925</td>
<td>56925</td>
<td>56925</td>
<td>56925</td>
<td>56925</td>
<td>56925</td>
</tr>
<tr>
<td>6</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
</tr>
<tr>
<td>7</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
<td>56500</td>
</tr>
</tbody>
</table>

Optimal value is $56500$

Optimal grouping is:

$$(A_1 \times A_2) \times (((A_3 \times A_4) \times A_5) \times A_6) \times A_7$$
Optimal Binary Search Trees

Given (as input):
▶ A set of values to be stored as keys in a binary search tree
▶ The frequency of access of each value.
▶ The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

Problem:
▶ Compute a binary search tree that minimizes the weighted lookup cost.

The weighted lookup cost in a binary tree with $n$ nodes is:

$$n \sum_{i=1}^{n} p_i c_i$$

▶ $p_i$ = probability (frequency) of accessing node $i$
▶ $c_i$ = cost of accessing node $i = 1 + \text{depth(node} i\text{)}$
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
- The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

Problem:
- Compute a binary search tree that minimizes the weighted lookup cost.

The weighted lookup cost in a binary tree with $n$ nodes is:

$$n \sum_{i=1}^{n} p_i c_i$$

- $p_i$ = probability (frequency) of accessing node $i$
- $c_i$ = cost of accessing node $i$ = 1 + depth(node $i$)
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
  - The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

The weighted lookup cost in a binary tree with \( n \) nodes is:

\[
\sum_{i=1}^{n} p_i c_i
\]

- \( p_i \) = probability (frequency) of accessing node \( i \)
- \( c_i \) = cost of accessing node \( i \) = 1 + depth(node \( i \))
Optimal Binary Search Trees

Given (as input):
- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
  - The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

Problem:
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
  - The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

Problem:

- Compute a binary search tree that minimizes the weighted lookup cost.
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
  - The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

Problem:

- Compute a binary search tree that minimizes the weighted lookup cost.

The weighted lookup cost in a binary tree with \( n \) nodes is:

\[
\sum_{i=1}^{n} p_i c_i
\]
Optimal Binary Search Trees

**Given (as input):**

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
  - The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

**Problem:**

- Compute a binary search tree that minimizes the **weighted lookup cost**.

The **weighted lookup cost** in a binary tree with \( n \) nodes is:

\[
\sum_{i=1}^{n} p_i c_i
\]

- \( p_i \) = probability (frequency) of accessing node \( i \)
Optimal Binary Search Trees

Given (as input):

- A set of values to be stored as keys in a binary search tree
- The frequency of access of each value.
  - The frequencies might add up to 1, in which case they are probabilities, but the algorithm works whether or not this is the case.

Problem:

- Compute a binary search tree that minimizes the weighted lookup cost.

The weighted lookup cost in a binary tree with $n$ nodes is:

$$\sum_{i=1}^{n} p_i c_i$$

- $p_i$ = probability (frequency) of accessing node $i$
- $c_i$ = cost of accessing node $i = 1 + \text{depth(node } i\text{)}$
Example

Suppose we have the following data values and frequency values:

<table>
<thead>
<tr>
<th>Data</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

One possible binary search tree:

```
A .26
  C .24
    E .16
    G .14
  B .06
    F .10
  D .04
```

Weighted lookup cost is 2.76:

<table>
<thead>
<tr>
<th>Node</th>
<th>Frequency</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.26</td>
<td>.78</td>
</tr>
<tr>
<td>B</td>
<td>.06</td>
<td>.12</td>
</tr>
<tr>
<td>C</td>
<td>.24</td>
<td>.72</td>
</tr>
<tr>
<td>D</td>
<td>.04</td>
<td>.04</td>
</tr>
<tr>
<td>E</td>
<td>.16</td>
<td>.48</td>
</tr>
<tr>
<td>F</td>
<td>.10</td>
<td>.20</td>
</tr>
<tr>
<td>G</td>
<td>.14</td>
<td>.42</td>
</tr>
</tbody>
</table>

2.76
Example

Suppose we have the following data values and frequency values:

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>( p_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>
Example

Suppose we have the following data values and frequency values:

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

One possible binary search tree:

![Tree Diagram]

Weighted lookup cost is 2.76:

<table>
<thead>
<tr>
<th>Node</th>
<th>$p_i$</th>
<th>$c_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.26</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>.06</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>.24</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>.04</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>.16</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>.10</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>.14</td>
<td>3</td>
</tr>
</tbody>
</table>

$2.76$
Example

Suppose we have the following data values and frequency values:

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

One possible binary search tree:
Example

Suppose we have the following data values and frequency values:

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

One possible binary search tree:

Weighted lookup cost is 2.76:

<table>
<thead>
<tr>
<th>$i$</th>
<th>Node</th>
<th>$p_i$</th>
<th>$c_i$</th>
<th>$p_i c_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
<td>3</td>
<td>.78</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
<td>2</td>
<td>.12</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
<td>3</td>
<td>.72</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
<td>1</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
<td>3</td>
<td>.48</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
<td>2</td>
<td>.20</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
<td>3</td>
<td>.42</td>
</tr>
</tbody>
</table>
Example

Suppose we have the following data values and frequency values:

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

One possible binary search tree: (non-optimal)

Weighted lookup cost is 2.76:

<table>
<thead>
<tr>
<th>$i$</th>
<th>Node</th>
<th>$p_i$</th>
<th>$c_i$</th>
<th>$p_i c_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
<td>3</td>
<td>.78</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
<td>2</td>
<td>.12</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
<td>3</td>
<td>.72</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
<td>1</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
<td>3</td>
<td>.48</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
<td>2</td>
<td>.20</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
<td>3</td>
<td>.42</td>
</tr>
</tbody>
</table>

2.76
Problem Statement

Input:
A list of data values and their frequency values

▶ K₁, . . . , Kₙ are the keys
▶ We assume that the keys are distinct and in sorted order:
  K₁ < K₂ < · · · < Kₙ.

▶ p₁, . . . , pₙ are the corresponding frequency values

Output:
a binary search tree of smallest weighted lookup cost.

Note:
▶ We assume all searches are successful (i.e., every search request is for one of the n keys K₁, . . . , Kₙ).
▶ The generalization to allowing unsuccessful searches is discussed in [CLRS].
Problem Statement

**Input:** A list of data values and their frequency values
Problem Statement

**Input:** A list of data values and their frequency values

- $K_1, \ldots, K_n$ are the keys
Problem Statement

Input: A list of data values and their frequency values

- $K_1, \ldots, K_n$ are the keys
  - We assume that the keys are distinct and in sorted order:
    $$K_1 < K_2 < \cdots < K_n.$$
Problem Statement

**Input:** A list of data values and their frequency values

- $K_1, \ldots, K_n$ are the keys
  - We assume that the keys are distinct and in sorted order:
    \[ K_1 < K_2 < \cdots < K_n. \]
- $p_1, \ldots, p_n$ are the corresponding frequency values
Problem Statement

**Input:** A list of data values and their frequency values

- $K_1, \ldots, K_n$ are the keys
  - We assume that the keys are distinct and in sorted order:
    
    \[ K_1 < K_2 < \cdots < K_n. \]

- $p_1, \ldots, p_n$ are the corresponding frequency values

**Output:** a binary search tree of smallest weighted lookup cost.
Problem Statement

**Input:** A list of data values and their frequency values
  
  ▶ $K_1, \ldots, K_n$ are the keys
    ▶ We assume that the keys are distinct and in sorted order:
      
      $$K_1 < K_2 < \cdots < K_n.$$ 
  
  ▶ $p_1, \ldots p_n$ are the corresponding frequency values

**Output:** a binary search tree of smallest weighted lookup cost.

**Note:**
Problem Statement

**Input:** A list of data values and their frequency values

- $K_1, \ldots, K_n$ are the keys
  - We assume that the keys are distinct and in sorted order:
    \[ K_1 < K_2 < \cdots < K_n. \]

- $p_1, \ldots, p_n$ are the corresponding frequency values

**Output:** a binary search tree of smallest weighted lookup cost.

**Note:**

- We assume all searches are successful (i.e., every search request is for one of the $n$ keys $K_1, \ldots, K_n$.)
Problem Statement

**Input:** A list of data values and their frequency values

- $K_1, \ldots, K_n$ are the keys
  - We assume that the keys are distinct and in sorted order:
    \[ K_1 < K_2 < \cdots < K_n. \]
- $p_1, \ldots, p_n$ are the corresponding frequency values

**Output:** a binary search tree of smallest weighted lookup cost.

**Note:**

- We assume all searches are successful (i.e., every search request is for one of the $n$ keys $K_1, \ldots, K_n$.)
- The generalization to allowing unsuccessful searches is discussed in [CLRS].
Finding Optimal Binary Tree

Define $E(i, j) = \text{the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys } K_i, \ldots, K_j$.

**Goal:** $E(1, n)$

**Base cases:**
- Tree with 1 node: $E(i, i) = p_i K_i p_i$
- Tree with 0 nodes: $E(i, i-1) = 0$

We need to develop a recurrence equation...
Finding Optimal Binary Tree

Define $E(i,j) =$ the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys $K_i, \ldots, K_j$. 
Finding Optimal Binary Tree

Define $E(i, j) =$ the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys $K_i, \ldots, K_j$.

Goal: $E(1, n)$
Finding Optimal Binary Tree

Define $E(i, j) =$ the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys $K_i, \ldots, K_j$.

Goal: $E(1, n)$

Base cases:
Finding Optimal Binary Tree

Define $E(i, j) =$ the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys $K_i, \ldots, K_j$.

Goal: $E(1, n)$

Base cases:
- Tree with 1 node: $E(i, i) = p_i$
Finding Optimal Binary Tree

Define $E(i, j) =$ the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys $K_i, \ldots, K_j$.

Goal: $E(1, n)$

Base cases:

- Tree with 1 node: $E(i, i) = p_i$
- Tree with 0 nodes: $E(i, i - 1) = 0$
Finding Optimal Binary Tree

Define $E(i, j) = \text{the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys } K_i, \ldots, K_j$.

Goal: $E(1, n)$

Base cases:

- **Tree with 1 node:** $E(i, i) = p_i$
- **Tree with 0 nodes:** $E(i, i - 1) = 0$

We need to develop a recurrence equation . . .
Finding Optimal Binary Tree: Develop recurrence equation

Suppose the root is $K_r$, where $i \leq r \leq j$.

- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$.
- Note that if $r = i$, this is an empty tree.

- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$.
- Note that if $r = j$, this is an empty tree.
Finding Optimal Binary Tree: Develop recurrence equation

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Suppose the root is $K_r$, where $i \leq r \leq j$.
- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$.
- Note that if $r = i$, this is an empty tree.
- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$.
- Note that if $r = j$, this is an empty tree.
Finding Optimal Binary Tree: Develop recurrence equation

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Suppose the root is $K_r$, where $i \leq r \leq j$

\[
\begin{align*}
\text{Optimal tree for } & K_i, \ldots, K_{r-1} \\
\text{Optimal tree for } & K_{r+1}, \ldots, K_j
\end{align*}
\]
Finding Optimal Binary Tree: Develop recurrence equation

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Suppose the root is $K_r$, where $i \leq r \leq j$
- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$
- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$
Finding Optimal Binary Tree: Develop recurrence equation

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Suppose the root is $K_r$, where $i \leq r \leq j$
- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$
  - Note that if $r = i$, this is an empty tree
- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$
  - Note that if $r = j$, this is an empty tree
Finding Optimal Binary Tree: Develop recurrence equation

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Suppose the root is $K_r$, where $i \leq r \leq j$
- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$
  - Note that if $r = i$, this is an empty tree
- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$
Finding Optimal Binary Tree: Develop recurrence equation

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Suppose the root is $K_r$, where $i \leq r \leq j$
- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$
  - Note that if $r = i$, this is an empty tree
- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$
  - Note that if $r = j$, this is an empty tree
Develop recurrence equation [continued]

Optimal tree for $K_i, \ldots, K_{r-1}$

Optimal tree for $K_{r+1}, \ldots, K_j$

Observation:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r-1)$.
- When we make this tree a subtree of the tree rooted at $K_r$, we push each node in the subtree down one level, increasing the cost of each node by 1.
- So the total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ in the tree rooted at $K_r$ is $E(i, r-1) + p_i + p_{i+1} + \ldots + p_{r-1}$.
- Similarly, the total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is $E(r+1, j) + p_{r+1} + \ldots + p_j$. 

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Develop recurrence equation [continued]

Optimal tree for $K_i, \ldots, K_{r-1}$

Optimal tree for $K_r$

Optimal tree for $K_i, \ldots, K_{r-1}$

Optimal tree for $K_{r+1}, \ldots, K_j$

Observation:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r-1)$.
- When we make this tree a subtree of the tree rooted at $K_r$, we push each node in the subtree down one level, increasing the cost of each node by 1.
- So the total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ in the tree rooted at $K_r$ is $E(i, r-1) + p_i + p_{i+1} + \ldots + p_{r-1}$.
- Similarly, the total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is $E(r+1, j) + p_{r+1} + \ldots + p_j$. 

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Observation:
Develop recurrence equation [continued]

Optimal tree for $K_i, \ldots, K_{r-1}$

Observation:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r - 1)$. 

Optimal tree for $K_r$

Optimal tree for $K_r+1, \ldots, K_j$
Develop recurrence equation [continued]

Observation:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r-1)$.
- When we make this tree a subtree of the tree rooted at $K_r$, we push each node in the subtree down one level, increasing the cost of each node by 1.
Develop recurrence equation [continued]

Observation:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r - 1)$.
- When we make this tree a subtree of the tree rooted at $K_r$, we push each node in the subtree down one level, increasing the cost of each node by 1.
- So the total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ in the tree rooted at $K_r$ is

$$E(i, r - 1) + p_i + p_{i+1} + \ldots + p_{r-1}.$$
Develop recurrence equation [continued]

Optimal tree for $K_i, \ldots, K_{r-1}$

Optimal tree for $K_i, \ldots, K_{r-1}$

Optimal tree for $K_{r+1}, \ldots, K_j$

### Observation:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r-1)$.
- When we make this tree a subtree of the tree rooted at $K_r$, we push each node in the subtree down one level, increasing the cost of each node by 1.
- So the total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ in the tree rooted at $K_r$ is
  
  $$E(i, r-1) + p_i + p_{i+1} + \ldots + p_{r-1}.$$ 

- Similarly, the total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is
  
  $$E(r+1, j) + p_{r+1} + \ldots + p_j.$$
Develop recurrence equation [continued]

As we just observed:

▶ The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is $E(i, r-1) + p_i + \ldots + p_{r-1}$.

▶ The total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is $E(r+1, j) + p_{r+1} + \ldots + p_j$.

The weighted cost of the root node is $1 \cdot p_r = p_r$.

Hence the weighted cost of the tree is:

$$E(i, r-1) + E(r+1, j) + p_i + \ldots + p_{r-1} + p_{r+1} + \ldots + p_j.$$
Develop recurrence equation [continued]

As we just observed:
▶ The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is $E(i, r-1) + p_i + \ldots + p_{r-1}$.
▶ The total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is $E(r+1, j) + p_{r+1} + \ldots + p_j$.

The weighted cost of the root node is $1 \cdot p_r = p_r$.

Hence the weighted cost of the tree is:
$$E(i, r-1) + E(r+1, j) + p_i + \ldots + p_{r-1} + p_{r+1} + \ldots + p_j.$$
As we just observed:
As we just observed:

- The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is

$$E(i, r - 1) + p_i + \ldots + p_{r-1}.$$
Develop recurrence equation [continued]

As we just observed:

- The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is
  \[ E(i, r-1) + p_i + \ldots + p_{r-1}. \]
- The total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is
  \[ E(r+1, j) + p_{r+1} + \ldots + p_j. \]
Develop recurrence equation [continued]

As we just observed:

- The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is
  \[ E(i, r-1) + p_i + \ldots + p_{r-1}. \]
- The total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is
  \[ E(r + 1, j) + p_{r+1} + \ldots + p_j. \]

The weighted cost of the root node is $1 \cdot p_r = p_r$. 
Develop recurrence equation [continued]

As we just observed:

- The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is
  \[ E(i, r-1) + p_i + \ldots + p_{r-1}. \]
- The total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is
  \[ E(r+1, j) + p_{r+1} + \ldots + p_j. \]

The weighted cost of the root node is $1 \cdot p_r = p_r$.

Hence the weighted cost of the tree is:
As we just observed:

- The total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is
  \[ E(i, r - 1) + p_i + \ldots + p_{r-1}. \]
- The total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is
  \[ E(r + 1, j) + p_{r+1} + \ldots + p_j. \]

The weighted cost of the root node is $1 \cdot p_r = p_r$.

Hence the weighted cost of the tree is:

\[ E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_{r-1} + p_r + p_{r+1} + \ldots + p_j. \]
Develop recurrence equation [continued]

We have just seen that weighted cost of the tree is:

$$E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_j$$

We can simplify this by defining $W(i, j)$ to be the sum of the frequencies of the keys $K_i, \ldots, K_j$:

$$W(i, j) = p_i + \ldots + p_j$$

With this simplification, the cost of the best tree we can build on keys $K_i, \ldots, K_j$ with $K_r$ as the root is:

$$E(i, r - 1) + E(r + 1, j) + W(i, j)$$
Develop recurrence equation [continued]

We have just seen that weighted cost of the tree is:

\[
E(i, r-1) + E(r+1, j) + p_i + \ldots + p_j
\]

We can simplify this by defining \( W(i, j) \) to be the sum of the frequencies of the keys \( K_i, \ldots, K_j \):

\[
W(i, j) = p_i + \ldots + p_j
\]

With this simplification, the cost of the best tree we can build on keys \( K_i, \ldots, K_j \) with \( K_r \) as the root is:

\[
E(i, r-1) + E(r+1, j) + W(i, j)
\]
We have just seen that weighted cost of the tree is:

\[ E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_j \]
We have just seen that weighted cost of the tree is:

$$E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_j$$

We can simplify this by defining $W(i, j)$ to be the sum of the frequencies of the keys $K_i, \ldots, K_j$. 
Develop recurrence equation [continued]

\[ K_r \]

Optimal tree for \( K_i, \ldots, K_{r-1} \)

Optimal tree for \( K_{r+1}, \ldots, K_j \)

- We have just seen that weighted cost of the tree is:
  \[
  E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_j
  \]

- We can simplify this by defining \( W(i, j) \) to be the sum of the frequencies of the keys \( K_i, \ldots, K_j \):
  \[
  W(i, j) = p_i + \ldots + p_j
  \]
Develop recurrence equation [continued]

- We have just seen that weighted cost of the tree is:
  \[ E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_j \]
- We can simplify this by defining \( W(i, j) \) to be the sum of the frequencies of the keys \( K_i, \ldots, K_j \):
  \[ W(i, j) = p_i + \ldots + p_j \]
- With this simplification, the cost of the best tree we can build on keys \( K_i, \ldots, K_j \) with \( K_r \) as the root is:
Develop recurrence equation [continued]

We have just seen that weighted cost of the tree is:

\[ E(i, r - 1) + E(r + 1, j) + p_i + \ldots + p_j \]

We can simplify this by defining \( W(i, j) \) to be the sum of the frequencies of the keys \( K_i, \ldots, K_j \):

\[ W(i, j) = p_i + \ldots + p_j \]

With this simplification, the cost of the best tree we can build on keys \( K_i, \ldots, K_j \) with \( K_r \) as the root is:

\[ E(i, r - 1) + E(r + 1, j) + W(i, j) \]
Develop recurrence equation [finally!]

We have just seen that the cost of the best tree we can build on keys $K_i, \ldots, K_j$ with $K_r$ as the root is:

$$E(i, r-1) + E(r+1, j) + W(i, j).$$

To get the best tree, we need to pick the best root. So our recurrence equation is

$$E(i, j) = \min_{i \leq r \leq j} \{E(i, r-1) + E(r+1, j) + W(i, j)\}.$$
Develop recurrence equation [finally!]

- We have just seen that the cost of the best tree we can build on keys $K_i, \ldots, K_j$ with $K_r$ as the root is:

$$E(i, r - 1) + E(r + 1, j) + W(i, j)$$
Develop recurrence equation [finally!]

- We have just seen that the cost of the best tree we can build on keys $K_i, \ldots, K_j$ with $K_r$ as the root is:

$$E(i, r - 1) + E(r + 1, j) + W(i, j)$$

- To get the best tree, we need to pick the best root.
Develop recurrence equation [finally!]

- We have just seen that the cost of the best tree we can build on keys $K_i, \ldots, K_j$ with $K_r$ as the root is:

\[
E(i, r - 1) + E(r + 1, j) + W(i, j)
\]

- To get the best tree, we need to pick the best root.
- So our recurrence equation is
We have just seen that the cost of the best tree we can build on keys $K_i, \ldots, K_j$ with $K_r$ as the root is:

$$E(i, r - 1) + E(r + 1, j) + W(i, j)$$

To get the best tree, we need to pick the best root.

So our recurrence equation is

$$E(i, j) = \min_{i \leq r \leq j} (E(i, r - 1) + E(r + 1, j) + W(i, j)).$$
Develop recurrence equation [finally!]

- We have just seen that the cost of the best tree we can build on keys \( K_i, \ldots, K_j \) with \( K_r \) as the root is:

\[
E(i, r-1) + E(r+1, j) + W(i, j)
\]

- To get the best tree, we need to pick the best root.

- So our recurrence equation is

\[
E(i, j) = \min_{i \leq r \leq j} (E(i, r-1) + E(r+1, j) + W(i, j)).
\]
Specifying the Solution

1. Subproblem domain
\[(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n\]

Note: The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. Function / Memoization table definition:
\(E(i, j)\) is the minimum weighted lookup cost for a binary search tree on the keys \(K_i, \ldots, K_j\), where \(p_i\) is the frequency of key \(K_i\).

3. Goal:
\(E(1, n)\)

4. Initial values:
\(E(i, i - 1) = 0\)

Note: Earlier we stated an additional set of initial values:
\(E(i, i) = p_i\). Since these values follow from the stated initial values and the recurrence equation, they do not need to be given as initial values.

5. Recurrence:
\[E(i, j) = \min_{i \leq r \leq j} \left( E(i, r - 1) + E(r + 1, j) + W(i, j) \right),\]
where \(W(i, j) = p_i + \cdots + p_j\).
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n\} \)
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n\} \)
   - **Note:** The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n \} \)
   
   - **Note:** The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. Function / Memoization table definition: \( E(i, j) \) is the minimum weighted lookup cost for a binary search tree on the keys \( K_i, \ldots, K_j \), where \( p_i \) is the frequency of key \( K_i \).
Specifying the Solution

1. **Subproblem domain** \( \{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n\} \)

   ▶ **Note:** The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. **Function / Memoization table definition:** \( E(i, j) \) is the minimum weighted lookup cost for a binary search tree on the keys \( K_i, \ldots, K_j \), where \( p_i \) is the frequency of key \( K_i \).

3. **Goal:** \( E(1, n) \)
Specifying the Solution

1. Subproblem domain \( \{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n\} \)
   
   ▶ Note: The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. Function / Memoization table definition: \(E(i, j)\) is the minimum weighted lookup cost for a binary search tree on the keys \(K_i, \ldots, K_j\), where \(p_i\) is the frequency of key \(K_i\).

3. Goal: \(E(1, n)\)

4. Initial values: \(E(i, i - 1) = 0\)
Specifying the Solution

1. **Subproblem domain** \{ (i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n \}
   - **Note:** The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. **Function / Memoization table definition:** \(E(i, j)\) is the minimum weighted lookup cost for a binary search tree on the keys \(K_i, \ldots, K_j\), where \(p_i\) is the frequency of key \(K_i\).

3. **Goal:** \(E(1, n)\)

4. **Initial values:** \(E(i, i - 1) = 0\)
   - **Note:** Earlier we stated an additional set of initial values: \(E(i, i) = p_i\). Since these values follow from the stated initial values and the recurrence equation, they do not need to be given as initial values.
Specifying the Solution

1. **Subproblem domain**: \(\{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n\}\)
   
   - **Note**: The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. **Function / Memoization table definition**: \(E(i, j)\) is the minimum weighted lookup cost for a binary search tree on the keys \(K_i, \ldots, K_j\), where \(p_i\) is the frequency of key \(K_i\).

3. **Goal**: \(E(1, n)\)

4. **Initial values**: \(E(i, i - 1) = 0\)
   
   - **Note**: Earlier we stated an additional set of initial values: \(E(i, i) = p_i\). Since these values follow from the stated initial values and the recurrence equation, they do not need to be given as initial values.

5. **Recurrence**:
   
   \[
   E(i, j) = \min_{i \leq r \leq j} \left( E(i, r - 1) + E(r + 1, j) + W(i, j) \right),
   \]
Specifying the Solution

1. **Subproblem domain** \( \{(i, j) : 1 \leq i \leq n + 1 \text{ and } i - 1 \leq j \leq n \} \)
   
   ▶ **Note:** The pair \((i, j) = (n + 1, n)\) needs to be handled, which is why the upper limit on the range of \(i\) is \(n + 1\) rather than \(n\).

2. **Function / Memoization table definition:** \(E(i, j)\) is the minimum weighted lookup cost for a binary search tree on the keys \(K_i, \ldots, K_j\), where \(p_i\) is the frequency of key \(K_i\).

3. **Goal:** \(E(1, n)\)

4. **Initial values:** \(E(i, i - 1) = 0\)
   
   ▶ **Note:** Earlier we stated an additional set of initial values: \(E(i, i) = p_i\). Since these values follow from the stated initial values and the recurrence equation, they do not need to be given as initial values.

5. **Recurrence:**

\[
E(i, j) = \min_{i \leq r \leq j} (E(i, r - 1) + E(r + 1, j) + W(i, j)),
\]

where \(W(i, j) = p_i + \cdots + p_j\).
Computation of $W(i, j)$

The values of $W(i, j) = p_i + \cdots + p_j$ can be precomputed in $O(n^2)$ time:

for $i = 1$ to $n+1$:
    $W[i,i-1] = 0$
for $j = i$ to $n$
Computation of $W(i, j)$

- The values of
  
  $$W(i, j) = p_i + \cdots + p_j$$

  can be precomputed in $O(n^2)$ time:

  ```plaintext
  for i = 1 to n+1:
      W[i, i-1] = 0
  for j = i to n:
  ```
Computation of $W(i, j)$

- The values of $W(i, j) = p_i + \cdots + p_j$

  can be precomputed in $O(n^2)$ time:

  for $i = 1$ to $n+1$:
  
  $W[i,i-1] = 0$

  for $j = i$ to $n$
  
Code to compute the cost of the optimal tree

```python
def OptimalTreeCost(p):
    for i = 1 to n+1:
        E[i,i-1] = 0
        W[i,i-1] = 0
    for j = i to n:
    for size = 1 to n:
        for i = 1 to n - size + 1 do
            j = i + size - 1
            E[i,j] = \infty;
            for r = i to j:
                x = E[i,r-1] + E[r+1,j] + W[i,j]
                if x < E[i,j]:
                    E[i,j] = x
    return(E)
```
Code to compute the cost of the optimal tree

def OptimalTreeCost(p):
    for i = 1 to n+1:
        E[i,i-1] = 0
        W[i,i-1] = 0
    for j = i to n:
    for size = 1 to n:
        for i = 1 to n - size + 1 do
            j = i + size - 1
            E[i,j] = +\infty;
            for r = i to j:
                x = E[i,r-1] + E[r+1,j] + W[i,j]
                if x < E[i,j]:
                    E[i,j] = x
    return(E)
Modified code to compute the root of each optimal subtree

To compute the optimal tree, we need to store the root of each optimal subtree. We compute a second array $\text{root}$ which tells us the best root of the tree for the keys $K_i, \ldots, K_j$:

```python
def OptimalTreeCost(p):
    for i = 1 to n+1:
        E[i,i-1] = 0
        W[i,i-1] = 0
    for j = i to n:
    for size = 1 to n:
        for i = 1 to n - size + 1 do
            j = i + size - 1
            E[i,j] = +\infty;
            for r = i to j:
                x = E[i,r-1] + E[r+1,j] + W[i,j]
                if x < E[i,j]:
                    E[i,j] = x
                    root[i,j] = r
    return(E,root)
```
Modified code to compute the root of each optimal subtree

To compute the optimal tree, we need to store the root of each optimal subtree. We compute a second array \texttt{root} which tells us the best root of the tree for the keys \(K_i, \ldots, K_j\):

```python
def OptimalTreeCost(p):
    E = [[0] * (n+1) for _ in range(n+1)]
    W = [[0] * (n+1) for _ in range(n+1)]
    for i in range(1, n+1):
        E[i][i-1] = 0
        W[i][i-1] = 0
    for j in range(i, n+1):
        W[i][j] = W[i][j-1] + p[j]
    for size in range(1, n+1):
        for i in range(1, n - size + 1):
            j = i + size - 1
            E[i][j] = float('inf');
            for r in range(i, j+1):
                x = E[i][r-1] + E[r+1][j] + W[i][j]
                if x < E[i][j]:
                    E[i][j] = x
                    root[i][j] = r
    return (E, root)
```
Modified code to compute the root of each optimal subtree

To compute the optimal tree, we need to store the root of each optimal subtree. We compute a second array `root` which tells us the best root of the tree for the keys $K_i, \ldots, K_j$:

```python
def OptimalTreeCost(p):
    for i = 1 to n+1:
        E[i,i-1] = 0
        W[i,i-1] = 0
    for j = i to n
    for size = 1 to n:
        for i = 1 to n - size + 1 do
            j = i + size - 1
            E[i,j] = +\infty;
            for r = i to j:
                x = E[i,r-1] + E[r+1,j] + W[i,j]
                if x < E[i,j]:
                    E[i,j] = x
                    root[i,j] = r
    return(E,root)
```
Code generate the optimal tree

Once we have computed the arrays $E$ and $root$, the following pseudocode computes the optimal binary tree:

```python
def OptimalTree(root, keys):
    # keys is the array of key values, indexed from 1 to n
    def buildTree(i, j):
        if j < i: return null
        r = root[i,j]
        node = new binary tree node
        node.key = keys[r]
        node.leftchild = buildTree(i, r-1)
        node.rightchild = buildTree(r+1, j)
        return node
    return buildTree(1, n)
```
Code generate the optimal tree

Once we have computed the arrays $E$ and $\text{root}$, the following pseudocode computes the optimal binary tree:

```python
def OptimalTree(root, keys):
    # keys is the array of key values, indexed from 1 to n
    def buildTree(i, j):
        if j < i: return null
        r = root[i,j]
        node = new binary tree node
        node.key = keys[r]
        node.leftchild = buildTree(i, r-1)
        node.rightchild = buildTree(r+1, j)
        return node
    return buildTree(1, n)
```

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Solution to our example
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>0.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>0.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>0.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>0.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20

![Diagram of the example solution](image)
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
## Solution to our example

### Data

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

### Weighted lookup cost

```
<table>
<thead>
<tr>
<th>j</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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>.26</td>
<td>.38</td>
<td>0.92</td>
<td>1.02</td>
<td>1.38</td>
<td>1.68</td>
<td>2.20</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>.06</td>
<td>.36</td>
<td>0.44</td>
<td>0.80</td>
<td>1.10</td>
<td>1.52</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>.24</td>
<td>.32</td>
<td>0.68</td>
<td>0.96</td>
<td>1.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>.04</td>
<td>.24</td>
<td>0.44</td>
<td>0.82</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>.16</td>
<td>.36</td>
<td>0.70</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>.10</td>
<td>.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Tree Diagram

```
  C
 / \   .24
A   .26
   /   [4, 7]
[1, 0] [2, 2]
```
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>
# Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

The weighted lookup cost is 2.20.
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>( p_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
- & 0.26 & 0.38 & 0.92 & 1.02 & 1.38 & 1.68 & 2.20 \\
1 & 1 & 3 & 3 & 3 & 3 & 3 & 3 \\
2 & 0.06 & 0.36 & 0.44 & 0.80 & 1.10 & 1.52 & - \\
3 & 2 & 3 & 3 & 3 & 5 & 5 & - \\
4 & 0.24 & 0.32 & 0.68 & 0.96 & 1.34 & - & - \\
5 & 3 & 3 & 5 & 5 & - & - & - \\
6 & 0.04 & 0.24 & 0.44 & 0.82 & - & - & - \\
7 & 4 & 5 & 5 & 5 & - & - & - \\
8 & 0 & 0.16 & 0.36 & 0.70 & - & - & - \\
9 & 5 & 5 & 6 & - & - & - & - \\
10 & 0 & 0.10 & 0.34 & - & - & - & - \\
11 & 6 & 7 & 7 & - & - & - & - \\
12 & 0 & 0.14 & - & - & - & - & - \\
13 & 7 & - & - & - & - & - & - \\
14 & - & - & - & - & - & - & - \\
\end{array}
\]

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>$i$</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>$E$</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>$F$</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>$G$</td>
<td>.14</td>
</tr>
</tbody>
</table>
Solution to our example

<table>
<thead>
<tr>
<th>i</th>
<th>Data</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>.26</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>.06</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>.24</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>.04</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>.16</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>.10</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>.14</td>
</tr>
</tbody>
</table>

Weighted lookup cost = 2.20