Exercises

Modify this algorithm to deal with the situations in which the supply of coins in some denominations is limited.

Let \( d[1..n] \) be another input array, and require at most \( d[i] \) coins of type \( i \) be used. Note \( 0 \leq d[i] \leq \infty \) for \( 1 \leq i \leq n \).

Once the table \( C[1..n, 0..n] \) has been filled, the second problem can be solved, i.e. exactly which coins are to be discarded.

If \( C[i, j] = C[i-1,j] \), then no coins of type \( i \) are needed to pay \( j \) units when restricted to types \( \{1, \ldots, i-1\} \). We move up one row to \( C[i-1,j] \) to see what to do next.

If \( C[i, j] = 1 + C[i, j-d[i]] \), we pay out one coin of type \( i \) then move left to \( C[i, i-d[i]] \) to see what to do next.

If \( C[i, j] \) equals both \( C[i-1,j] \) and \( 1 + C[i, j-d[i]] \), then either action is acceptable.
Exercise:
Write a recursive algorithm which given the filled table \( C[1...n, 0...N] \), prints a sequence of \( C[n, N] \) coins types whose value adds to \( N \), i.e., \( C[n, N] = 0 \). Print an appropriate message.

The 0-1 Knapsack Problem

A thief wishes to steal \( n \) objects indexed \( i = 1 \) to \( n \). Let

\[
\begin{align*}
V_i &= \text{value of object } i \\
W_i &= \text{weight of object } i
\end{align*}
\]

The thief has a knapsack which can carry a maximum weight of \( W \). His goal is to fill the knapsack in a way which maximizes the total value of the goods stolen, while respecting its capacity constraint.

Let

\[
x_i = \begin{cases} 
0 & \text{if object } i \text{ is NOT taken} \\
1 & \text{if object } i \text{ is TAKEN}
\end{cases}
\]
Thus the problem is to choose $x_i \in \{0, 1\}$ (1 ≤ i ≤ n) to

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

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

where $v_i > 0$, $w_i > 0$, $W > 0$.

To solve this problem we create a table $V[0 \ldots n; 0 \ldots W]$ where $V[i, i]$ is the maximum value of the objects in the set $\{1, \ldots, i\}$ whose total weight does not exceed $i$ (1 ≤ i ≤ n, 0 ≤ i ≤ W).

To determine $V[i, i]$ we have in general two alternatives:

- Do not include object $i$. In this case at most value $V[i-1, i]$ can be stolen.
- Include object $i$. This increases the value of the load by $v_i$, and reduces the remaining capacity by $w_i$. Thus in this case at most value $v_i + V[i-1, i-w_i]$ can be stolen.
Choosing the best alternative yields:

\[ V[i,i] = \max(V[i-1,i], v_i + V[i-1, i-w_i]) \]

Including boundary and out of bound entries we have:

\[ V[i,i] = \begin{cases} 
0 & i = 0, i > n \\
\max(V[i-1,i], v_i + V[i-1, i-w_i]) & i > 0, i \leq n \\
-\infty & i < 0 
\end{cases} \]

**Example:**

\( n = 5, \ W = 10 \)

<table>
<thead>
<tr>
<th>( w_1 = 1 )</th>
<th>( v_1 = 1 )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>17</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>25</td>
<td>26</td>
<td>26</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>25</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

**Exercise:**

Write an algorithm which given the input arrays \( V[1...n] \) and \( W[1...n] \), and the weight limit \( W \), fills in the table \( V[1...n; 0...W] \) and returns \( V[n,W] \). (Or if you prefer, return the whole table.)
Exercise
Write an algorithm which given the filled table $V[i...n; 0...w]$ prints out a list of exactly which objects to include.

The Principle of Optimality

An optimization problem satisfies the principle of optimality if the optimal solution to any non-trivial instance is a combination of some of its subinstances.

I.e., an optimal solution contains within it optimal solutions to certain subproblems.

I.e., in an optimal sequence of choices, each subsequence is also optimal.

We also say that such a problem exhibits optimal substructure.

E.g. Coin Change:
$C[i, w] = \min (C[i-1, w], 1 + C[i, w-d[i]])$

E.g. 0-1 Knapsack:
$V[i, w] = \max (V[i-1, w], v_i + V[i-1, w-w_i])$
DEFINITION: A \textit{U-V Path} is a sequence of alternative vertices and incident edges starting at \textit{U} and ending at \textit{V}, in which no vertex is repeated (except possibly \textit{U}=\textit{V}).

The \textbf{length of a path} is the number of edges in the sequence.

\textbf{Problem}: Determine a \textit{shortest U-V Path}.

Let \(d(u,v)\) denote the length of such a path.

Observe: any segment of a shortest path is also a shortest path.
Proof: Let $P$ be a shortest $u-v$ path, $s$ and $t$ two intermediate vertices on $P$, and $P'$ the segment of $P$ from $s$ to $t$. If $P'$ were not a shortest $s-t$ path we could splice $P'$ out of $P$ and replace it with a shorter $s-t$ path $P''$. The resulting $u-v$ path would then be shorter than $P$, contradicting the fact that $P$ was shortest.

The shortest path problem satisfies the principle of optimality.

One might easily come to believe that all optimality problems exhibit optimal substructure. But this is false. If a problem fails to satisfy the optimality principle, it is probably impossible to solve it using dynamic programming.

Ex. Longest Path in Graph

Problem: Determine a longest $u-v$ path.

Let $L(u,v)$ determine the length of such a path.
Observe that a segment of a longest U-V path may not be a longest path. In the previous example, one checks that \( c(x, y) = 8 \), but a longest U-V path travels from \( x \) to \( y \) in 7 steps.

This problem therefore violates the principle of optimality.

**General Procedure**

1. Characterize the structure of an optimal solution. (i.e. show that your problem involves some choice(s) which lead one or more subproblems.)

2. Recursively define an optimal solution (i.e. in terms of optimal subproblem solutions.)

3. Compute the value of an optimal solution in a bottom-up fashion (i.e. construct table)

4. Construct an optimal solution (using the table generated in (3).)