If bond \( i = 1 \) and \( j \leq d_i \), then we set
\[
C[i, j] = +\infty
\]
indicating that it is impossible to pay amount \( j \) using only coins of type 1.

**Ex.** \( n = 4, \quad N = 8 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
d_1 = 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
d_2 = 2 & 0 & 1 & 1 & 2 & 3 & 2 & 3 & 4 & \\
d_3 = 5 & 0 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 2 \\
d_4 = 6 & 0 & 1 & 2 & 1 & 1 & 1 & 2 & 2 &
\end{array}
\]

Note that if we have an unlimited supply of coins of value 1 (e.g., \( d_1 = 1 \)) then it is possible to discharge any amount. Otherwise, it may be impossible to pay certain amounts. We indicate this with \( C[i, j] = \infty \).

**Ex.** \( n = 3, \quad N = 8 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
d_1 = 2 & 0 & \infty & 1 & \infty & 2 & \infty & 3 & \infty & 4 \\
d_2 = 4 & 0 & \infty & 1 & \infty & 1 & \infty & 2 & \infty & 2 \\
d_3 = 5 & 0 & \infty & 1 & \infty & 1 & 1 & 2 & 2 & 2
\end{array}
\]
The following algorithm takes inputs $d[1..n], N,$ and uses a local array $C[1..n; 0..n]$

\textbf{CoinChange} \((d, N)\)

1. \(n \leftarrow \text{length}[d]\)
2. \(\text{for } i \leftarrow 1 \text{ to } n\)
3. \(C[i, 0] \leftarrow 0\)
4. \(\text{for } i \leftarrow 1 \text{ to } n\)
5. \(\text{for } j \leftarrow 1 \text{ to } N\)
6. \(\text{if } i = 1 \text{ and } j < d[i]\)
7. \(C[i, j] \leftarrow \infty\)
8. \(\text{else if } i = 1\)
9. \(C[i, j] \leftarrow 1 + C[i, j - d[i]]\)
10. \(\text{else if } j < d[i]\)
11. \(C[i, j] \leftarrow C[i - 1, j]\)
12. \(\text{else}\)
13. \(C[i, j] \leftarrow \min(C[i - 1, j], 1 + C[i, j - d[i]]\))
14. \(\text{return } C[n, N]\)

The algorithm can be easily altered to return the entire table $C[1..n; 0..n]$ of intermediate results.

The run time is obviously $O(nN)$ since each of $n \cdot (N + 1)$ table entries must be filled.
Exercise

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

Let $C[i \ldots n]$ be another input array, and require at most $C[i]_{\leq i}$ coins of type $i$ to be used. Note $0 \leq C[i] \leq 10$ for $1 \leq i \leq n$.

Once the table $C[1 \ldots n; 0 \ldots n]$ 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 $\{1, \ldots, i\}$. We move up one row to $C[i-1,j]$ to see what to do next.

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

If $C[i,j]$ equals both $C[i-1,j]$ and $1 + C[i,j-di]$ then either action is acceptable.
Excercise

Write a recursive algorithm which
Given the filled table \( C[1...n, 0...n] \),
print a sequence of \( C[n, n] \) coins
types whose value adds to \( n \). i.e.
\( C[n, n] = \infty \) 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 \( \leq \) i \( \leq \) n) to

\[
\text{maximize } \sum_{i=1}^{n} x_i v_i \\
\text{subject to } \sum_{i=1}^{n} x_i w_i \leq W
\]

where \( v_i > 0, w_i > 0, W > 0 \).

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

To determine \( V[i,j] \) we have in general two alternatives:

- Do not include object \( i \). In this case at most value \( V[i-1,j] \) 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 \( v_i + V[i-1,j-w_i] \) 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 Bounds Entries We Have

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

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w_1 = 1, v_1 = 1 )</td>
<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>
</tr>
<tr>
<td>( w_2 = 3, v_2 = 5 )</td>
<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>
</tr>
<tr>
<td>( w_3 = 5, v_3 = 12 )</td>
<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>
</tr>
<tr>
<td>( w_4 = 6, v_4 = 25 )</td>
<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>
</tr>
<tr>
<td>( w_5 = 7, v_5 = 30 )</td>
<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>35</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 $V[i \ldots n, 0 \ldots w]$ print 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 or its subinstances.

i.e., an optimal solution contains within its 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**

\[ C[i, j] = \min (C[i-1, j], 1 + C[i, j-d_i]) \]

**E.g. 0-1 Knapsack**

\[ V[i, j] = \max (V[i-1, j], v_i + V[i-1, j-w_i]) \]
Exercise: Shortest Paths in Graph

*Definition:* A **U-V Path** is a sequence of alternating vertices and incident edges starting at U and ending at V, in which no vertex is repeated (except possibly U=V).

The length of a path is the number of edges in the sequence.

**Problem:** Determine a 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.