Problem statement

- A thief robbing a store finds $n$ items; the $i$th item is worth $v_i$ dollars and weights $w_i$ pounds. He wants to take as valuable a load as possible, but he can carry at most $W$ pounds in his knapsack. What items should be taken?

- Formally, the problem can be stated as follows:
  - Input: $n$ items of values $v_1, v_2, \ldots, v_n$ and of the weight $w_1, w_2, \ldots, w_n$, and a total weight $W$, where $v_i$, $w_i$ and $W$ are positive integers.
  - Output: a subsect $S \subseteq \{1, 2, \ldots, n\}$ of the items such that
    \[
    \sum_{i \in S} w_i \leq W \quad \text{and} \quad \sum_{i \in S} v_i \text{ is maximized.}
    \]

- This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once. The knapsack problem is an abstraction of many real problems, from investing to telephone routing.

Dynamic Programming

1. The solution is based on the optimal-substructure observation. Let $i_k$ be the highest-numbered item in an optimal solution $S = \{i_1, \ldots, i_k\}$ of $W$ pounds and items $\{1, 2, \ldots, n\}$. Then $S' = S - \{i_k\}$ is an optimal solution for $W - w_{i_k}$ pounds and items $\{i_1, \ldots, i_{k-1}\}$, and the value of the solution $S$ is $v_{i_k}$ plus the value of the subproblem solution $S'$.

2. We can express the optimal-substructure observation in the following recursive formula:

Define $c[i, w]$ to be the value of an optimal solution for items $1 \ldots i$ and maximum weight $w$. Then for $i = 0, 1, \ldots, n$ and $w = 1, 2, \ldots, W$:

\[
c[i, w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
c[i - 1, w] & \text{if } i > 0 \text{ and } w_i > w \\
\max(v_i + c[i - 1, w - w_i], c[i - 1, w]) & \text{if } i > 0 \text{ and } w_i \leq w 
\end{cases}
\]

This says that the value of a solution for item $i$ is

- either includes item $i$, in which case it is $v_i$ plus a subproblem solution for $i - 1$ items and the weight excluding $w_i$,
- or does not include item $i$, in which case it is a subproblem solution of $i - 1$ items and the same weight.
This is, if the thief picks item \(i\), he takes \(v_i\) value, and he can choose from items \(1...i-1\) up to the weight limit \(w - w_i\), and get \(c[i-1, w-w_i]\) additional value. On the other hand, if he decides not to take item \(i\), he can choose from items \(1...i-1\) up to the weight limit \(w\), and get \(c[i-1, w]\) value. The better of these two choices should be made.

3. Although the 0-1 knapsack problem doesn’t seem analogous to the LCS problem, the above formula for \(c\) is similar to the LCS formula: initial values are 0, and other values are computed from the inputs and “earlier” values of \(c\). So the 0-1 knapsack problem is like the LCS-LENGTH algorithm for finding the LCS of two sequences.

4. The pseudocode is presented below. The algorithm takes as inputs the maximum weight \(W\), the number of items \(n\), and the two sequences \(v = \langle v_1, v_2, \ldots v_n \rangle\) and \(w = \langle w_1, w_2, \ldots w_n \rangle\). It stores the \(c[i, w]\) values in the table of \(c[0..n, 0..W]\) whose entries are computed row-major order (This is, the first row of \(c\) is filled from left to right, then the second row, and so on.) At the end of the computation, \(c[n, W]\) contains the maximum value the thief can take.

```
Dynamic-0-1-Knapsack(v,w,n,W)
  for w = 0 to W
    c[0,w] = 0
  for i = 1 to n
    c[i,0] = 0
    for w = 1 to W
      if w[i] <= w then
        if v[i] + c[i-1,w-w[i]] > c[i-1,w] then
          c[i,w] = v[i] + c[i-1,w-w[i]]
        else
          c[i,w] = c[i-1,w]
        end if
      else
        c[i,w] = c[i-1,w]
      end if
    end for
  end for
```

5. The set of items to take can be deduced from the \(c\)-table by starting at \(c[n, W]\) and tracing where the optimal values came from.

- If \(c[i, w] = c[i-1, w]\), item \(i\) is not part of the solution, and we continue tracing with \(c[i-1, w]\).
- Otherwise item \(i\) is part of the solution, and we continue tracing with \(c[i-1, w-w_i]\).
6. The above algorithm takes $\Theta(nW)$ time total:

- $\Theta(nW)$ to fill in the $c$ table – $(n + 1)(W + 1)$ entries each requiring $\Theta(1)$ time to compute.
- $O(n)$ time to trace the solution (since it starts in row $n$ of the table and moves up 1 row at each step.)

7. Example. Let

$$n = 9,$$
$$v = \langle 2, 3, 3, 4, 4, 5, 7, 8, 8 \rangle,$$
$$w = \langle 3, 5, 7, 4, 3, 9, 2, 11, 5 \rangle,$$
$$W = 15.$$  

By the program $\text{Dynamic-0-1-Knapsack}(v,w,n,W)$, the calculated $c$-table is

<table>
<thead>
<tr>
<th>i/w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</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>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>23</td>
</tr>
</tbody>
</table>


The set of items to take = \{9, 7, 5, 4\}