**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 follws:

**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 $\mathcal{S} \subseteq \{1, 2, \ldots, n\}$ of the items such that

$$\sum_{i \in \mathcal{S}} w_i \leq W \quad \text{and} \quad \sum_{i \in \mathcal{S}} v_i \quad \text{is maximized.}$$

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 $k$ be the highest-numberd item in an optimal solution $\mathcal{S}$ of $W$ pounds and items $\{1, 2, \ldots, n\}$. Then $\mathcal{S}' = \mathcal{S} - \{k\}$ is an optimal solution for $W - w_k$ pounds and items $1, 2, \ldots, k-1$, and the value of the solution $\mathcal{S}$ is $v_k$ *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

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

   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 $1 \ldots i-1$ 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 $1 \ldots i-1$ 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 analogus 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 $v = \langle v_1, v_2, \dots v_n \rangle$ and $w = \langle w_1, w_2, \dots w_n \rangle$. 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 – $(n+1) \cdot (W+1)$ 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.)

**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 `Dynamic-0-1-Knapsack(v,w,n,W)`, the calculated c-table is

| $w \rightarrow$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i = 0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $i = 1$ | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $i = 2$ | 0 | 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| $i = 3$ | 0 | 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 8 |
| $i = 4$ | 0 | 0 | 0 | 2 | 4 | 4 | 4 | 6 | 6 | 7 | 7 | 7 | 9 | 9 | 9 | 9 |
| $i = 5$ | 0 | 0 | 0 | 4 | 4 | 4 | 6 | 8 | 8 | 8 | 10 | 10 | 11 | 11 | 11 | 13 |
| $i = 6$ | 0 | 0 | 0 | 4 | 4 | 4 | 6 | 8 | 8 | 8 | 10 | 10 | 11 | 11 | 11 | 13 |
| $i = 7$ | 0 | 0 | 7 | 7 | 7 | 11 | 11 | 11 | 13 | 15 | 15 | 15 | 17 | 17 | 18 | 18 |
| $i = 8$ | 0 | 0 | 7 | 7 | 7 | 11 | 11 | 11 | 13 | 15 | 15 | 15 | 17 | 17 | 18 | 18 |
| $i = 9$ | 0 | 0 | 7 | 7 | 7 | 11 | 11 | 15 | 15 | 15 | 19 | 19 | 19 | 21 | 23 | 23 |

Optimal value $= c[n, W] = c[9, 15] = 23$.
The set of items to take $= \{9,\ 7,\ 5,\ 4\}$.