Now let's look at an application of dynamic programming to a problem that has haunted us throughout the course:

**Subset-Sum Problem**

Let $S = \{ s_1, s_2, \ldots, s_n \}$ and let the target value be $k$.

I'm going to present the dynamic programming solution very briefly - you should be able to fill in any gaps and complete the steps outlined above.

We can look at the process of finding a solution as a sequence of decisions: first we decide whether or not to include $s_n$, then $s_{n-1}$, etc. After we have made decisions about $s_n$, $s_{n-1}$, ..., $s_j$ we are left decisions to make about the elements of $\{s_1 \ldots s_j\}$ - that is part of the identification of each subproblem. But every time we decide to include an element of the set, the target value is reduced by that much. This suggests that we need two parameters to identify each subproblem:

- a parameter $i$ that identifies the highest index in the set of elements we still have to decide about: $\{s_1, s_2, \ldots, s_i\}$
- a parameter $x$ that identifies the target value we are trying to reach.

Now we can define the notation for specific subproblems:

$$S_S(i, x) = \text{T iff there is a subset of } \{s_1 \ldots s_i\} \text{ that sums to } x.$$  

Having two parameters indicates that we should use a 2-dimensional table to store the results for subproblems. I'll use a table $S_S_T$ in which we list the elements of $S$ on one axis, and possible target values on the other. For the possible target values, we will use 1 .. $k$ because we don't know which subproblems are going to be needed.

We will store $S_S(i, x)$ in $S_S_T[i][x]$
We can establish the necessary recurrence by observing that

\[ S_S(1, x) = T \text{ if } s_1 = x \]
\[ = F \text{ if } s_1 \neq x \]

\[ S_S(i, x) = T \quad \text{if} \quad s_i = x \]
\[ \text{or} \quad S_S(i - 1, x) \quad - \text{there is a subset of } \{s_1 \ldots s_{i-1}\} \quad \text{that sums to } x \]
\[ \text{or} \quad S_S(i - 1, x - s_i) \quad - \text{there is a subset of } \{s_1 \ldots s_{i-1}\} \quad \text{that sums to } x - s_i \]
\[ = F \quad \text{otherwise} \]

The three cases in which \( S_S(i, x) \) is true are:
- \( s_i = x \) … if this is true, the target value \( x \) is achieved simply by using the subset \( \{s_i\} \)
- \( S_S(i - 1, x) \) ... if this is true, we can achieve \( x \) without using \( s_i \)
- \( S_S(i - 1, x - s_i) \) ... if this is true, we can achieve \( x \) by combining \( s_i \) with some elements of \( \{s_1, \ldots s_{i-1}\} \)

Thus each value \( S_S_T[i][x] \) in the table is based on checking three things:

\[ s_i \]
\[ S_S_T[i-1][x] \]
\[ S_S_T[i-1][x-s_i] \quad - \text{note that if } x-s_i \leq 0, \text{ we can ignore this option} \]

Thus each element of the table can be computed in constant time. It is easy to see that since each table element refers to elements in the previous row, we can fill in the table from top to bottom, one row at a time.

If the table has a \( T \) in the bottom right corner then the answer to \( \text{Subset}_\text{Sum}(S,k) \) is Yes, and if not then the answer is No.

This gives us a concise and simple-seeming algorithm that will always correctly solve Subset Sum … and yet earlier we learned that Subset Sum is NP-Complete. Does our new algorithm show that \( P = NP \)?
Not surprisingly, the answer is no … this is not a proof that \( P = NP \).

It is true that this algorithm runs in \( O(n^k) \) time …. which looks like a polynomial. But we have to remember that we need to determine running time as a function of the size of the input … and in this case the input consists of \( n+1 \) values: the \( n \) elements of \( S \), and the target value \( k \).

But there’s no upper limit on \( k \)! Suppose \( k = 2^n \) … then our complexity is \( O(n \times 2^n) \) which is clearly not polynomial. So in the worst case this algorithm will require exponential time.

But if \( k \) is limited by a polynomial function of \( n \) … for example, if we can guarantee that \( k \leq n^2 \) … then the dynamic programming solution does run in polynomial time.