Dynamic Programming – the Knapsack problem

(simplified version – this does not take values of items into consideration)

Given a set of `n` items with sizes `S[1], S[2], ..., S[n]` and a capacity `k`, find a subset of items so that the total size of the chosen items does not exceed the capacity and is a maximum possible (least amount of space left over).

`n` = the number of items
`k` = total capacity of the knapsack
`S` = array of sizes of items

Remember: Each item is represented once and only once in the array of sizes. If you have multiple items of a particular size/type, the items would be repeated in the array. Each individual item will either be selected or not selected to be in the knapsack. (Example: if you have **three** items of size 4 and **two** items of size 5 along with some other items of different sizes, your array might be something like S = [7, 5, 4, 2, 4, 5, 4].)

To use dynamic programming, we will build a solution to this problem by solving subproblems of various sizes:

the (i, j)-subproblem: for `0 ≤ i ≤ n`, and `0 ≤ j ≤ k`
determine whether or not there exists a subset of the first `i` items whose total size adds up to exactly `j`

`exist[i][j]` = `T` if there is **some** subset, from among the first **`i`** items, whose sizes add up to **exactly** **`j`**

`belong[i][j]` = `T` if, when constructing the subsolution for **`exist[i][j] = T`**, item **`i`** belongs to the set of items corresponding to the `(i,j)`-subsolution

Example:

```
n = 5
k = 16
S = [2, 3, 7, 5, 6]

exist[2][5] =

belong[2][5] =

exist[3][6] =

exist[5][6] =

belong[5][6] =

exist[3][5] =

belong[3][5] =
```

For the (i, j)-subproblem, we must decide whether or not to include item `i` of size `S[i]` in the subset of items whose total size adds up to exactly `j`.

First, if there is already a subset of the first `i-1` items that adds up to exactly `j`, then there is also a subset of the first `i` items that adds up to exactly `j`. But item `i` is not part of that solution.

```
if exist[i-1][j] = true,
      exist[i][j] = true
      belong[i][j] = false
```

Otherwise, we need to determine whether or not to **use** the item of size `S[i]` to make partial capacity `j`:

- First, make sure that `j - S[i] ≥ 0` (is it even possible to fit the item?)
- Remaining partial capacity is: `j - S[i]`
- Is there a subset of the previous `i-1` items whose sizes add up to `j - S[i]`? If so, then item `i` will belong to the solution.

```
if exist[i-1][j-S[i]]
      exist[i][j] = true
      belong[i][j] = true
```

The arrays exist and belong are filled from the top left corner. This is still a bottom-up approach for dynamic programming because we are working from the smallest subproblems (subsets of 0 items, then subsets of 1 item, then subsets of 2 items, …).

Once the array `exist` has been filled, where is the solution to the problem for all `n` items and capacity `k`?

exist[____][_____]

```
dynamicKnapsack(S, k)
{
  exist[0][0] = true
  for j = 1 to k
  {
    exist[0][j] = false
  }
  for i = 1 to n
  {
    for j = 0 to k
    {
      exist[i][j] = false
      if (exist[i-1][j])          // previous subset of  i-1  elements
      {                           //  adds up to exactly j
        exist[i][j] = true        // So subset of  i  elements sums to j
        belong[i][j] = false      //  but new ith element doesn't belong
      }
      else if (j - S[i] ≥ 0)      //  if the ith element fits
      {                           //    in remaining space
        if (exist[i-1][j-S[i]]) // use a partial solution if it exists
        {                         //  to build the next solution
          exist[i][j] = true
          belong[i][j] = true
        }
      }
    }
  }
}
```

n = 5
k = 16
S = [2, 3, 7, 5, 6]

exist	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0																	
1																	
2																	
3																	
4																	
5																	

belong	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0																	
1																	
2																	
3																	
4																	
5																	

How do you find the solution at the end of the algorithm?
Which items are chosen for the knapsack?

Think: If **`exist[n][k]`** is true, then there is some subset of all **`n`** items whose sizes add up to exactly **`k`**. In other words, the knapsack would be completely filled. If that is not the case, you need to find how full the knapsack is for the most optimal solution…

Starting in row **`n`** and column **`k`** (i.e., considering among all **`n`** items, move backwards in the row to find the largest possible total size for which a solution **`exists`**). The column number gives the total size of the items in the solution.

If the current item (indicated by the row number) does not **`belong`** to the solution, move up in the column (the current total size) until you find an item that does **`belong`**.

Subtract the **`size`** of that item from the current total size.

Move up a row (the current item has been removed so it is no longer in consideration).

Repeat until all items have been determined (the current total size is 0).

Use this sequence of steps to help you write an algorithm to determine the set of items for the knapsack.