Algorithms: Subset Sum (Dynamic Programming)

Model 1: Some sets

$$A = \{1, 2, 3, 5, 7\}$$
$$B = \{4, 16, 19, 23, 25, 72, 103\}$$
$$C = \{3, 34, 6, 17\}$$
$$D = \{\}$$

1 For each number below, say whether each set has some subset which adds up to the given number. For example, A and C have subsets which add up to 7 ($\{7\}$ and $\{5, 2\}$ respectively), but B and D do not.

(a) 9

(b) 16

(c) 0

In general, consider the following problem, called the Subset Sum problem:

- **Input**:
  - a set $\{x_1, \ldots, x_n\}$ of n positive integers, and
  - a positive integer S.
- **Output**: is there a subset of $\{x_1, \ldots, x_n\}$ whose sum is exactly S?

Yes, the first element is x_1, not x_0. This is a deliberate choice which will come in handy later.

2 Describe a brute-force algorithm for solving this problem.

3 What is the running time of your brute-force algorithm?

Make sure you consider the time to add up each subset, not just the time to list them all.

4 Use your brute-force algorithm to decide whether there is any subset of C which adds up to 54. What about 55?

Let's see how to attack this problem using dynamic programming.

Step 1: Break the problem into subproblems and make a recurrence.

- We can make the problem simpler by restricting ourselves to only using *some* of the x_i. For example, a subproblem might look like "Can we find a subset of only $\{x_1, \ldots, x_k\}$ that adds up to S?" for some $k \le n$.
- However, by itself this doesn't help: just knowing whether we can add up to S using only $x_1, \ldots, x_k$ doesn't tell us whether we can add up to S using $x_1, \ldots, x_n$. In particular, in order to add up to S we might need to use some of the elements from $x_1, \ldots, x_k$ in addition to some of the other elements. We can fix this by generalizing along another dimension as well: we need to know whether we can add up not just to S itself, but to *any* sum $0 \le s \le S$. That is, a subproblem now looks like "Can we find a subset of only $\{x_1, \ldots, x_k\}$ that adds up to s?" for some $k \le n$ and $s \le S$.

Define $\mathit{canAddTo}(k, s)$ to be a true or false value which is the answer to the question, "Is there a subset of only the first k elements $\{x_1, \ldots, x_k\}$ which adds up to exactly s?"

5 Consider set $A = \{1, 2, 3, 5, 7\}$ from Model 1 again. Number the elements starting from 1, that is, $x_1 = 1$, $x_2 = 2$, $\ldots$, $x_5 = 7$. Evaluate each expression below as true or false, and give a brief justification for each.

(a) $\mathit{canAddTo}(5, 16)$

(b) $\mathit{canAddTo}(4, 16)$

(c) $\mathit{canAddTo}(4, 9)$

(d) $\mathit{canAddTo}(3, 1)$

(e) $\mathit{canAddTo}(3, 0)$

(f) $\mathit{canAddTo}(0, 2)$

(g) $\mathit{canAddTo}(0, 0)$

Now let's come up with a recurrence for *canAddTo* in the general case of determining whether there is a subset of $X = \{x_1, \dots, x_n\}$ which adds up to S.

6 Fill in base cases for *canAddTo*:

- $\mathit{canAddTo}(k, 0) =$ ________________________ for all $k \geq 0$,

  because we can always ____________________________.

- $\mathit{canAddTo}(0, s) =$ ________________________ for all $s > 0$,

  because there's no way to __________________________.

7 Now consider $\mathit{canAddTo}(k, s)$ in the general case, when $k > 0$ and $s > 0$. That is, we are trying to find whether we can add up to exactly s if we're only allowed to use $x_1, \dots, x_k$. In order to break this problem down into subproblems, we would need to decrease k and/or s. Fill in the following steps.

- If ____________, then we definitely cannot use x_k as part of a

  subset adding to s, because it is too ____________________.
  In this case, we would get the same result if we only allowed ourselves to use $\{x_1, \dots, x_{k-1}\}$, that is, $\mathit{canAddTo}(k, s)$ is the

  same as ______________________________________.

- Otherwise, we have two choices: we can try to use x_k as part of our subset or not. If we don't use it, it is the same as the previous case. If we do use it, then in order to complete a subset

  adding to s we have to make a subset using only __________ which adds up to ____.

8 Use your reasoning above to write down a complete recursive definition of *canAddTo*. Don't forget the base cases!

Step 2: Memoize.

9 Explain why it would be extremely slow to directly evaluate *canAddTo*(n, S) as a recursive function.

10 *canAddTo* takes a *pair* of values as input: $0 \leq k \leq n$ and $0 \leq s \leq S$. How many possible such pairs are there?

Of course there are infinitely many pairs of numbers; this question is really asking about how many different inputs to recursive calls we might possibly see after calling *canAddTo*(n, S).

11 If we wanted to memoize the results of *canAddTo* by storing the output corresponding to each possible input, what data structure should we use? Draw a picture.

12 How big is this data structure?

13 In what order can we fill in the data structure, so that we never try to fill in a value before filling in other values it depends on?

14 How long does it take to fill in each value?

15 Therefore, what is the running time of this dynamic programming algorithm?

16 Is this faster than your answer to Question 3?

Hint: this is a trick question.

17 Write some code (using either pseudocode or a language of your choice) to compute *canAddTo*(n, S) using the approach outlined here.