Dynamic programming example: subset sum

CSCI 382, Algorithms

October 28, 2019

As in the activity from class, given a set $X = \{x_1, x_2, \ldots, x_n\}$ and a target value S, we wish to determine whether there is a subset of X with sum exactly equal to S.

Step 1: A Recurrence

Consider different ways of splitting up or restricting the overall problem into subproblems or subcases, and come up with a recurrence.

The key to solving this problem is to generalize it along *two* dimensions: we consider both summing to any $s \leq S$, and we also consider trying to use only the x_i up to x_k, that is, $\{x_1, \ldots, x_k\}$, instead of the full set X. That is, $\mathit{canAddTo}(k, s)$ will be **True** if there is a subset of $\{x_1, \ldots, x_k\}$ which adds to exactly s. We have the following recurrence:

$$\mathit{canAddTo}(k, 0) = \textbf{True}$$

$$\mathit{canAddTo}(0, s) = \textbf{False} \qquad (\text{when } s > 0)$$

$$\mathit{canAddTo}(k, s) = \begin{cases} \mathit{canAddTo}(k-1, s) & \text{if } x_k > s \\ \mathit{canAddTo}(k-1, s) \vee \mathit{canAddTo}(k-1, s - x_k) & \text{otherwise} \end{cases}$$

That is,

- We can always add to the sum 0 by picking the empty subset.
- We can never add up to a nonzero sum if we aren't allowed to use any of the x_i.
- If $x_k > s$, then it can't be used in a subset that sums to s, so whether we can make the sum s using a subset of $x_1 \ldots x_k$ has the same answer as whether we can make s using a subset of $x_1 \ldots x_{k-1}$.
- Otherwise, we can try both omitting x_k (in which case we have to make s using elements up to x_{k-1}, as before), or using it (in which case we have to make the remaining $s - x_k$ using the elements up to x_{k-1}.

Step 2: Induction

An inductive proof of correctness follows the outlines of the above argument. Our induction hypothesis is to assume that *canAddTo* will give the correct answer for any $k' < k$ and/or $s' < s$, and then argue why we do the right things with the results of the recursive calls made.

Step 3: Memoize

If there are overlapping subproblems, memoize.

This most definitely has overlapping subproblems. One simple approach, as discussed on the in-class activity, is to use a 2D array c of size $(n+1) \times (S+1)$, so $c[k][s]$ will store the output of *canAddTo*(k, s). Each entry depends only on entries either above it, or above it and to the left, so we can fill it in row order or column order. In pseudocode:

```
for k from 0 to n do
    for s from 0 to S do
        if s = 0 then
            c[k][s] = True
        else if k = 0 then
            c[k][s] = False
        else if x_k > s then
            c[k][s] = c[k-1][s]
        else
            c[k][s] = c[k-1][s] || c[k-1][s-x_k]
```

Figure 1: SubsetSum

Alternatively, we could use the technique of having a recursive function which checks first to see whether the required output is already in the array.

Step 4: Remember Your Choices!

To compute the actual optimal solution instead of just the optimal value, save the choices made at each step.

What information does *canAddTo* discard? It is precisely the choice of whether to use x_k or not. The Boolean "or" operation will be **True** if either one of its inputs is; it does not care which. Therefore we will make another $(n+1) \times (S+1)$ array of booleans called *use*, where *use*$[k][s]$ is **True** if and only if we should use x_k in a subset to make s (Figure 2). Assume *use* gets initialized with all **False** values.

```
for k from 0 to n do
    for s from 0 to S do
        if s = 0 then
            c[k][s] = True
        else if k = 0 then
            c[k][s] = False
        else if x_k > s then
            c[k][s] = c[k − 1][s]
        else
            without ← c[k − 1][s]
            with ← c[k − 1][s − x_k]
            if with then
                use[k][s] = True
            c[k][s] = with || without
```

Figure 2: SubsetSum

If $c[n][S]$ is **True**, then there is some subset of X which adds to S. To reconstruct such an actual subset we can work our way backwards as follows:

```
k ← n
s ← S
Initialize T to the empty set
while k > 0 and s > 0 do
    if use[k][s] then
        Add x_k to T
        s ← s − x_k
    k ← k − 1
```