20191018

Subset Sum – Again?

We have looked at Subset Sum in some depth. Here's the problem definition again:

Given a set of n positive integers $S = \{s_1, s_2, \ldots, s_n\}$ and a target integer k, is there a subset of S that sums to exactly k?

Clearly for any set S and any target k, the answer is either Yes or No. I have previously claimed that this is one of the most difficult problems there is, in terms of the computational complexity of algorithms that we can use to solve it. We have seen that the BFI algorithm runs in O(2^n), while the Horowitz-Sahni algorithm runs in O($n * 2^{\frac{n}{2}}$)

Now let's solve it using Dynamic Programming.

Suppose the answer for set S and target k is "Yes" (to be consistent with the way we developed this in class, I will use "True" or just "T" for situations where the answer is "Yes")

Then the last element of S (s_n) is either in the desired subset ... or it isn't.

If s_n is in the solution, then the rest of the solution must be selected from $\{s_1, \ldots s_{n-1}\}$ with the target value reduced to $k - s_n$

If s_n is not in the solution, then the solution must be selected from $\{s_1, \ldots s_{n-1}\}$ with the target value remaining at k

If the answer is True, one of these two situations must be True.

Following the now familiar dynamic programming reasoning, we don't know which of these two subproblems applies so we evaluate both of them.

We can write $(S[1..n], k)$ to represent the original problem: we are looking for a subset of $\{s_1, \ldots s_n\}$ that sums to k.

This lets us write

$(S[1..n], k)$ = True if and only if $(S[1..n-1], k - s_n)$ = True

or

$(S[1..n-1), k)$ = True

Consider the first of these reduced problems. Let $x = k - s_n$. We can write

$(S[1..n-1], x)$ = True if and only if $(S[1..n-2], x - s_{n-1})$ = True

or

$(S[1..n-2], x)$ = True

and we can see that this relationship just cascades down through the subsets. When we notice that the subsets we are working with all start with s_1, we can simplify the notation even further: we can use SS(i,x) to represent the problem instance (S[1..i],x) ... ie, "Does $\{s_1 \ldots s_i\}$ have a subset that sums to x?

It may seem that by focusing on subsets that all start with s_1 we are severely limiting the number of subsets we will consider – but remember we are considering subsets of these subsets. If there is a solution we will find it.

At this point we have done most of the work of creating our recurrence relation:

$SS(i, x)$ = True if and only if $SS(i-1, x - s_i)$ = True or $SS(i-1, x)$ = True

We just need some base cases. Consider the problems of the form SS(1,x). Basically this problem is asking "Does $\{s_1\}$ contain a subset that sums to x? Clearly the answer is Yes (True) only if x == 0 or x = s_1 ... so these problems are base cases (ie. we can determine the answer without further recursion).

There are some other situations where we can immediately determine whether the answer is True or False:

SS(i,0) = True $\forall$ i ... because the empty set sums to 0 and it is a subset of everything.

If x < 0, SS(i,x) = False (we assume all integers in S are positive)

If s_i = x, then SS(i,x) = True

We can use these as base cases as well, if needed.

Now we can decide how to store the results to the reduced problems we will solve. Fortunately the recurrence relation basically tells us what to do. Since each subproblem is identified by two parameters i and x, we can use a 2-dimensional table with the possible values of i on one axis (I'll use the vertical) and the possible values of x on the other (horizontal).

The possible values of i are 1, 2, ..., n. I'll show the value of s_i on the line for i.

The possible values of x ... well, we don't know which subproblems are going to be important. Hey, it's DP, we will do them all! We can work out SS(i,x) for all values of x from 1 to k.

Let's do an example: $S = \{1, 4, 7, 12, 13, 23\}, k = 18$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4																		
7																		
12																		
13																		
23																		

I have filled in the first row – these are some of the base cases we identified above.

Consider the next row. The first cell represents the problem SS(2,1), ie "Does $\{s_1, s_2\}$ contain a subset that sums to 1. Our recurrence tells us the answer is True iff either SS(1,1) or SS(1,-3) is True. SS(1,1) IS True, so we get T for SS(2,1). In general we will see that T values always propagate vertically through the table.

You can work out that SS(2,2), AND SS(2,3) are F and SS(2,4) is T. But when we get to SS(2,5), things get a bit more interesting. SS(2,5) is True iff either of SS(1,5) is True (it's not) OR SS(1,1) is True (it is!) Make sure you see what's going on here: we subtracted s_2 from 5 to get the reduced target value.

That's it for exciting stuff on the second row. You can work out that all the remaining values are F. Now we have:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4	T	F	F	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F
7																		
12																		
13																		
23																		

Now I'll fill in the rest of the table. You should make sure you see why T's start to appear.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4	T	F	F	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F
7	T	F	F	T	T	F	T	T	F	F	T	T	F	F	F	F	F	F
12	T	F	F	T	T	F	T	T	F	F	T	T	T	F	F	T	T	F
13	T	F	F	T	T	F	T	T	F	F	T	T	T	T	F	T	T	T
23	T	F	F	T	T	F	T	T	F	F	T	T	T	T	F	T	T	T

So the table tells us the answer to SS(6,18) is Yes ... but what is the subset that adds to 18? We start in the bottom right corner and trace our way back through the T values. Any time the T we are on has another T right above it, we don't need the s_i that corresponds to this row. If the T we are on has F right above it, we jump back to the "reduced target" (ie $x - s_i$) in the previous row. This tells us the current s_i is part of the solution.

Repeating this step until we have identified the elements in the solution, we find that the subset that sums to 18 is {1,4,13}

Note that each value in the table is either a base case that we can fill in immediately, or it is the result of looking at one or two values in the previous row. This means each value in the table is computed in constant time.

This gives us a concise and efficient-looking algorithm that will always correctly solve Subset Sum ... which seems to contradict our earlier assertion that there are no fast algorithms for this problem. What's going on?

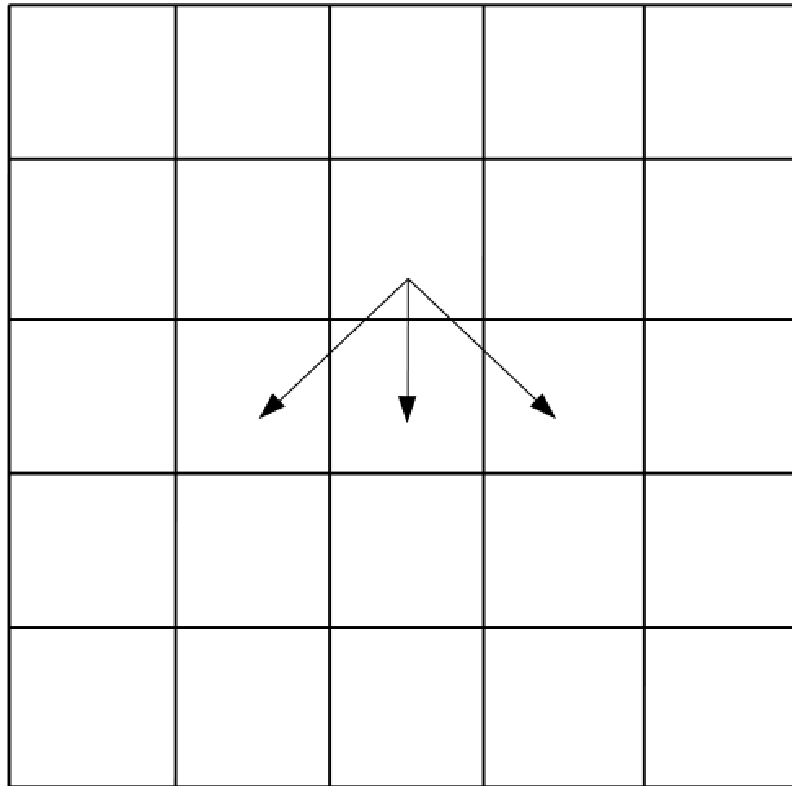
The table we build in this algorithm has size n*k, so the algorithm runs in O(n*k) time. The problem is the target figure k. There's no upper limit on it.

Remember that our measure of complexity expresses the running time of the algorithm in terms of the size of the input. For Subset Sum, the size of the input is n+1 (the set, and k). But if k happens to be 2^n then the running time of the algorithm is in O($n * 2^n$)

So the worst-case running time of this algorithm is exponential and it's consistent with the claim that there are no fast algorithms for solving Subset Sum. But the good news is that if we know that k is "not too big" (for example k $\leq n^2$) then this algorithm is an excellent solution to the Subset Sum problem.

We concluded our introduction to Dynamic Programming by looking at a very easy problem:

Given an n*m grid of squares (like a chess-board that isn't required to be square) in which each square contains an integer representing a cost, find the least-cost path from the top to the bottom of the grid, subject to the constraint that from square (i,j) we can only move down and to the left to square (i+1, j-1), straight down to square (i+1, j), or down and to the right to square (i+1, j+1). This figure shows the moves available from one of the squares.



It's not hard to see that a Greedy Algorithm won't work here. The next figure shows an instance of the problem where the optimal solution (shown with stars) doesn't start on the least cost square in the top row, and at each step except the last one, doesn't use the least cost successor of the current square..

7	4	12	1	4
2	6	15	26	19
2	14	7	3	8
19	18	8	4	9
21	5	18	14	16

A student suggested using Dijkstra's algorithm to solve this problem. We could do that by attaching an extra square at the top, connected to all the squares in the top row, and similarly attaching an extra square at the bottom. This would certainly work. The DP solution might do less work because it just iterates through all the squares, solving each subproblem in constant time, as opposed to solving fewer subproblems but taking O(log n) to choose which one to solve on each iteration.

As an exercise you should work out the recurrence relation for this algorithm, expressing the cost of reaching square (i,j) as a function of reaching the squares that have (i,j) as a successor. Make sure you consider the special cases for the squares on the left and right sides of the board, which have only 2 predecessors each. The total (or aggregate) cost to reach each square is computed in constant time.

If the board has dimension n*m, and we need only constant time to compute the aggregate cost of reaching each square, the algorithm runs in O(n*m) time (compare this to Dijkstra's algorithm on a graph with n*m vertices).

The reason I introduced this simple problem is that it turns out to be immensely practical and useful.   In class I showed this video which shows a remarkable application of this DP algorithm:

https://www.youtube.com/watch?v=qadw0BRKeMk

This functionality has now been incorporated into Photoshop.