Exercises: subset sum and knapsack

Questions

1. Why is knapsack a more general problem than subset sum. (Give a formal answer.)

2. Consider an instance of subset sum in which w1 = 1, w2 = 4, w3 = 3, w4=6 and W = 8. Draw the table of opt(i, w) values computed by dynamic programming.

3. Consider the instance of subset sum in the previous question, but now let W=7. The opt(i, w) table is

```
1 1 1 1 1 1 1
1 1 1 4 5 5 5
1 1 3 4 5 5 7
1 1 3 4 5 6 7
```

   Show, by backtracking through the table from opt(4, 7) that there is more than one solution.

4. For the subset sum problem, if we iteratively build a table for Opt(i, w), it obviously takes time and space O(N W), namely the size of the table. What if we used recursion instead? What would be the space and time required ?

5. Consider the "fractional knapsack" problem in which the weights are no longer integers and indeed the quantities you can take of a given item can be fractional, namely any weight between 0 and wi. For example, the quantities might be powder or liquid form. Suppose the values also scale proportionately e.g. if you take half of item 1, then it has half the weight and it has half the value. Again, assume you are limited by the total weight W.

   For this problem, there is a good greedy solution. What is it?

The next two examples are from the book "The Design and Analysis of Algorithms" by A. Levitin. They are meant to give you experience formulating recurrences for dynamic programming problems.

6. The Coin Row Problem: Suppose you have a row of coins with values that are positive integers c1, …, cn. These values might not be distinct. Your task is to pick up coins have as much total value as possible, subject to the constraint that you don't ever pick up two coins that lie beside

each other. How would you solve this using dynamic programming? Solve the problem for coins with values c1 to c6 as follows: (5, 1, 2, 10, 6, 2).

7. The Coin Change Problem: Suppose we have m types of coins with values c1 < c2 < .. cm. e.g. in the case of pennies, nickels, dimes, … we would have c1=1, c2=5, c3=10, … Let f(n) be the minimum number of coins whose values add up to exactly n. Write a recurrence for f(n) in terms of the values of the coins. You may use as many of each type of coin as you wish.

   As an example, suppose the coin values c1, c2, and c3 are 1, 3, 4. Solve the problem for n = 6 using dynamic programming.

8. In the lecture I mentioned a subtle issue that arises when we claim subset sum (or knapsack) takes time and space that are O(N W). The issue is that usually in computer science we write O( ) as a function of the size of the input. However, W is just a single number, not the size of an input.

   Suppose we say that the input consists of number with B bits. How large could W be, in order for the subset sum problem to be non-trivial ?

   Note:
   - The sumset sum problem is trivial if W is greater than the sum of all the weights.
   - Don't bother with this question if you don't yet know how numbers are represented in binary. (If you haven't learned about binary number representations yet, you'll have to wait for the first few lectures of COMP 273. I cover it in COMP 250 but other profs don't.)

Answers

1. Subset sum is a special case of knapsack where vi = wi for all I, that is, the values equal the weights.

2. The columns are numbered w = 1 to 8. The rows are numbered 1 to 4. The bottom right hand corner is the optimal solution, namely, opt(N=4, W= 8).

```
1 1 1 1 1 1 1 1
1 1 1 4 5 5 5 5
1 1 3 4 5 5 7 8
1 1 3 4 5 6 7 8
```

3. opt(4, 7) == opt(3,7) and so we can conclude that there is a solution that does not use w4. But we also note that opt(3, 7 – w4) + w4 = opt(3,1) + w4 = opt(3,7) and so there is also a solution that does use w4. Thus there is more than one solution.

To find a solution that does not use w4, we backtrack to opt(i=3,w=7). We see that opt(3,7) is not equal to op(2,7) and so the solution at opt(3,7) must have used w3=3. We go to opt(2, 7-3) = opt(2,4) = 4 to find the rest of that solution. We see that opt(2,4) is different from opt(1,4) and so we know w2 is part of this solution. Backtracking from opt(2,4) to opt(1, 4-w2) = opt(1,0) and we see we are done. This solution is {w2, w3} i.e 4+3 = 7.
To find a solution that does use w4, we backtrack from opt(4,7) to opt(3,1). We see that opt(3,1) == opt(2,1) and so there is a solution that doesn't use w3, and similarly there is a solution that doesn't use w2. Finally we see opt(1,0) is different from opt(0,0) and so this solution used w1. This solution is thus {w4, w1} i.e. 6+1 = 7.

4. The recursive algorithm begins with an empty table opt[][]. Starting at value (N, W), which is the last row and column in the table, the recursive algorithm for computing opt(N, W) considers two alternatives in the second last row in the table (namely row N-1), as given by the recurrence. It chooses the cell that has the larger of these two values. (If there is a tie, then it has two solutions and chooses one arbitrarily.) Notice that, to evaluate the two cells in row N-1, it need to apply compute them (recursively). In the worst case, it need to evaluate 4 cells in row N-2, and 2^j cells in row N-j. Once 2^j reaches W, the algorithm would (in the worst case) have to evaluate all cells in row N-j and beyond down to row 0. Thus, the recursive method is still O(N W).

5. Consider the value per unit weight, ie. v_i / w_i and sort the items by this "density". Then, start by choosing as much as possible of the most expensive per unit weight i.e. the highest v_i / w_i. If that choice uses all the weight, then you're done. If there's room left, though, then take as much as possible of the next most expensive, etc.

   This method requires O(N log N) for sorting but only requires O(N) for the rest of the algorithm.

6. The idea for the recurrence is as follows. Start with the last coin. You either pick it up or you don't. If you pick it up, then you cannot pick up the second to last coin but you are free to pick up any others. If you don't pick up the last coin, then you are free to pick up any of the others (subject to the problem's constraints). The recurrence that describes this is

   f(n) = max(c_n + f(n – 2), f(n-1)), with base case f(1) = c1, f(0) = 0. You can solve this either iteratively or recursively using dynamic programming.

   For the example given, the maximum value is 17 and uses coins {c1=5, c4=10, c6=2}.

7. To write the recurrence, we consider the case that a coin of type j was used in the optimal solution. Then, if a coin of type j was used, we need to solve the subproblem of finding the minimum number of coins whose values sum up to n – c_j. Thus,
   f(n) = min_{j in 1 to m} { 1 + f(n- c_j) }, f(0) = 0.

For the example given, the solution is two coins of value 3 each.

8. If each weight is represented with B bits, then we would need N * B bits to represent the N weights. With B bits, the weights can be as large as 2^B – 1. For example, with 8 bits, you can represent the numbers 0, … 255. For the subset sum problem, we could therefore define W to be any number up to the sum of the weights, that is, in the worst case it could be any number up to N * (2^B – 1). For even modest size B, e.g. 32 bit integers, the size of W could be quite large.