Mock Interviews @ ACM

Solution Card

Room Draw Woes

Problem Statements

You and your room draw group are looking to be housed next year! You're hoping that housing won't mess up this time and forget about any of you like they did last time. Can you help them out by solving these problems?

You have a draw group with `N` people and Housing has a specific set of room sizes (i.e. the number of people each room can house) `rooms` they offer. Write a function `roomDrawOne(N, rooms)` returning the *minimum* number of rooms Housing can assign to your draw group so that everyone is accommodated *and* no room is under capacity. (Assume there is an infinite supply of each room.) If it's impossible to accommodate your draw group exactly, output `-1`.

Assumptions. You can assume `N` and the elements of `rooms` (i.e. possible room sizes) will fit in a 32-bit positive integer.

As usual, you should analyze the runtime and space usage of any solution you come up with.

Source: Coin Change from Leetcode.

Hints

Hint 1. Suppose you're only allowed doubles and quads. If you know for fact that the minimum number of rooms needed to accommodate 6 people is 2 and the minimum number of rooms needed to accommodate 8 people is 2, then what can you say about the minimum number of rooms needed to accommodate 10 people?

Hint 2. More generally, let `min[k]` represent the minimum number of rooms needed to accommodate k people. Can you come up with a recurrence for `min[k]`? What technique does this smell of? (dynamic programming)

Solution

Note: This solution can be optimized. See the follow-ups section for how to.

Taking the hint, we can note that if we know the values of min[k - j] for every room size j, then we can deduce that min[k] will be the minimum value of any of the following:

- Any of the min[k - j] plus one, which would represent first accommodating (k - j) people with the minimum number of rooms, then adding a room of size j.
- min[k] itself, since it's possible that there is a room of size k.

(Refer to the hint to see what the min array is defined as.) Thus, if we loop through the min array starting from the leftmost index (0) and simply apply the recursion, we'll be done! Just don't forget to define the base cases: we can accommodate zero people with zero rooms, and for each of the room sizes, we can accommodate that many people with one room.

```java
import java.lang.Integer;
import java.lang.Math;

public int roomDrawOne(int N, int[] rooms) {

    /* This will be our dynamic programming array: min[k] is the minimum number
       of rooms needed to accommodate k people. */
    int[] min = new int[N + 1];

    /* To begin, we'll assume that it takes (roughly) infinitely many rooms to
       accommodate every draw group size except for zero people and numbers
       corresponding directly to room sizes. */
    for (int i = 1; i < N + 1; i++) min[i] = Integer.MAX_VALUE - 1;
    for (int i = 0; i < rooms.length; i++) {
        if (rooms[i] <= N) // Be careful of out-of-bounds errors!
            min[rooms[i]] = 1;
    }
    /* The dynamic programming step. */
    for (int i = 0; i < N + 1; i++) {
        for (int j = 0; j < rooms.length; j++) {
            if (i + rooms[j] <= N) // Be careful of out-of-bounds errors!
                min[i + rooms[j]] = Math.min(min[i + rooms[j]], min[i] + 1);
        }
    }

    // The following returns -1 if min[N] is "infinity" and min[N] otherwise
    return (min[N] == Integer.MAX_VALUE - 1) ? -1 : min[N];
}
```

Analysis

Time. Let M be the number of room types housing can offer (i.e. the size of rooms). The runtime is dominated by the dynamic programming step, which does an O(N) operation for every element in rooms. Altogether, this means that the algorithm's runtime is **O(MN).**

Space. The auxiliary space needed is dominated by the space needed to store the min array. It has N + 1 elements, so the auxiliary space needed is **O(N).**

Tips and Tricks

- You'll notice that we use `Integer.MAX_VALUE - 1` as an "infinity" of sorts. This is a good trick to employ, though it only works in Java. Other languages have their own workarounds, e.g. in Python you can use `float('inf')`.

Follow-Ups

Minor Optimization. There is a way that the code can be rewritten so that we don't need to loop from the beginning of the array every time. If we instead write our code so that the outer loop goes through the room sizes, then the inner loop (that goes through the array) can start at the room size that was selected in the outer loop, rather than at 0. That is, we can change the dynamic programming step to the following:

```java
for (int j = 0; j < rooms.length; j++) {
    for (int i = rooms[j]; i <= N; i++) {
        min[i] = Math.min(min[i], min[i - rooms[j]] + 1);
    }
}
```

Note that this doesn't make any asymptotic improvements to the runtime. But we could see significant runtime improvements if the room sizes are very big.

Time/Space Tradeoff. Imagine the case where we have some enormous number of people to accommodate, and there is just one room size. For example, we may only have quads available and we would want to accommodate a draw group of 1000000. If we run the algorithm above, then many entries of min (specifically those whose index isn't a multiple of 4) never change and remain at infinity. So ¾ of our million allocated bytes for the array are useless, in a sense.

Can you think of a different data structure that could reduce our space consumption in this case, at the expense of a little bit of asymptotic runtime? (Hint: Make min a symbol table containing only draw group sizes that can be accommodated with a finite number of rooms. Let the key be the draw group size and the value be the (running) minimum number of rooms needed to accommodate the draw group size. Then, in your dynamic programming step, only update the `min` value of elements that are greater than any elements already in `min` by the current room size being considered. Your symbol table will need to be able to produce a sorted order of the keys.)