Solution to the Problem 3.4: Binary Knapsack

Main Idea

Our weights are powers of 2, and the total weight w could also be considered as a sum of powers of 2, if we look into its binary representation (e.g. $13 = 1101_2 = 8 + 4 + 1$). So basically what we want is to find the best item for each term in the expansion of w, e.g. the most valuable item of weight 8, the most valuable item of weight 4 and so on. However, it's not that simple — instead of taking a single item of weight 4 we could take two items of weight 2, if it's better in terms of value. Or it could be even more complicated — one item of weight 8 could be replaced by one item of weight 4, one item of weight 2 and two items of weight 1. And it could also be the case that it's impossible or not optimal to get total weight of exactly w. So what do we do?

The key idea is to look at one particular weight at a time, when all smaller weights are already considered, and all larger weights don't affect anything yet.

We start from the lightest items. Say we have several items of weight 1, and w is odd. Then we could safely take one of these items and decrease w by 1 — because no matter what other items we will take, their sum will be even (as all other weights are even), and nothing will take place of the last one unit of weight. Of course, we should take the most valuable item out of those with weight 1 — because if there is an item of weight 1 in the knapsack, then the most valuable item of weight 1 must also be there, otherwise we could swap them and improve the total value. And nothing stops us from taking the most valuable item of weight 1 right away, as the order of putting items into knapsack doesn't matter.

So w is even — if it was odd we either took an item of weight 1, or there were no items of weight 1, in this case we could just decrease w by 1 as all other weights are even anyway. And all items have even weight, except maybe some items of weight 1 which we haven't put into the knapsack yet. If we consider the optimal solution, either none of them is taken — then it just doesn't matter what we'll do with them. Or, some are taken — then there must be at least two of them in the knapsack, because if we take just one there'll surely be an empty space of weight 1 in the knapsack as anything else is even. So if we have at least two remaining items of weight 1, we could safely combine two best-valued of them into one new item of weight 2 (and the values sum up of course). Because if we take an item of weight 1 then we could safely take the second, and these items must be best-valued among all items of weight 1. And we combine again while there are at least two remaining items. Eventually, it's either zero or one one-weighted items remaining. The first case is simple — we just don't have to do anything. In the second case, we could just combine this last item with an "empty" item — if we ever take this worst-valued item, then nothing could occupy free space beside it as all other one-weighted items are already taken. So we lose nothing by assigning it a weight of two.

Now all remaining items have even weight, and the total weight is even. So we could just divide every weight by two (powers of two are still powers of two after that), and do exactly the same. We repeat until either there are no more items, or the remaining weight becomes zero.

Implementation Details

In the implementation, it's more natural to iterate over powers of 2 instead of dividing everything by 2 on each iteration. If you use a signed 32-bit integer for the current weight, there's also a catch: if you write the loop like

```
for (int weight = 1; weight <= (1 << 30); weight *= 2) {
```

it will never terminate — as `int` can take only values up to $2^{31}-1$, on the last iteration the value has to become $2 \cdot 2^{30}$, but due to overflow it's actually -2^{31} which is less than 2^{30}. And when it doubles the next time, it just becomes zero. So you need to use unsigned ints or 64-bit integers, or rewrite the terminating condition, or iterate over the exponent (notice the handy << binary shift operator — you could easily get the value 2^k as `1 << k`).

Use an array to store all candidates of current weight — we need only the value for each candidate. After you've fixed the weight, you could just iterate over all items and add those which have this particular weight to the array. Actually there could also be something in here from the previous iteration — when we've bundled together spare items of smaller weight. Sort the candidate array to get more valuable items first.

We need to check if the current weight 2^k is in the binary expansion of w (or if the k-th binary digit in w is 1, which is the same). For that we could use a bitwise AND operator — `w & weight` will be exactly `weight` when it's true, and zero otherwise. So if the value `w & weight` is nonzero, then we take the most valuable item — which is, you need to add its value to the answer and remove from the candidate array (you don't actually need to decrease w, as we'll only look at its larger bits afterwards).

After that, we need to do the combining step: make a new candidate list, then combine two most valuable items from the old candidate list until there's at most one left (each time we put the sum of two values into the new candidate list), and if one is still remaining then just put its value into the new list. Substitute the old list with the new one, and the iteration step is done!