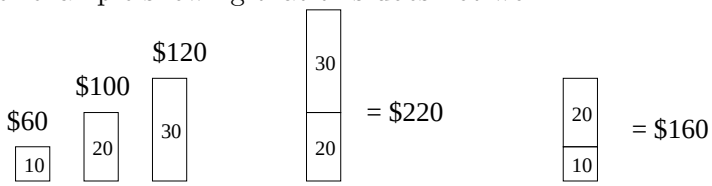
Dynamic Programming: 0-1 Knapsack

- Problem: Given n items, with item i being worth v_i and having weight w_i pounds, fill a knapsack of capacity W pounds with maximal value.
- Perhaps a greedy strategy of picking the item with the biggest value-per-pound might work? Here is a counter-example showing that this does not work:



Items in value per pound order | Optimal solution for knapsack of size 50 | Greedy solution for knapsack of size 50

Note: In the FRACTIONAL-KNAPSACK PROBLEM we can take $\frac{2}{3}$ of \$120 object and get \$240 solution.

- Now we'll show that $0-1$ KNAPSACK PROBLEM can be solved in time $O(n \cdot W)$ using dynamic-programming.
- Often the hardest part is coming up with the recursive formulation. Let us denote by $optknapsack(k, w)$ the maximal value obtainable when filling a knapsack of capacity w using items among items 1 through k.
- To solve our problem we need to compute $optknapsack(n, W)$.
- The idea is to consider each item, one at a time. Let's take item k: either it's part of the optimal solution, or not. We need to compute both options, and chose the best one.

```
optknapsack(k,w)
  IF weight[k] <= w THEN
    with = value[k] + optknapsack(k-1, w - weight[k])
  ELSE
    with = 0
  END IF
  without = optknapsack(k-1,w)
  RETURN max{with, without}
END Knapsack
```

- Analysis: Let $T(n, W)$ be the running time of *optknapsack*(k, w).

$$T(n, W) = T(n-1, W) + T(n-1, W - w[n]) + \Theta(1)$$

We'll look at the worst case where $w[i] = 1$ for all $1 \leq i \leq n$. If $w[i] = 1$ then it is clear that $T(n, W) > 2T(n-1, W-1)$. This recurrence, which runs for $\min(n, W)$ steps, gives that $T(n, W) = \Omega(2^{\min(n,W)})$.

- We now show how to improve the exponential running time with dynamic programming: We create a table T of size $[1..n][1..W]$ in which to store our results of prior runs. Entry $T[i][w]$ will store the result of *optknapsack*(i, w).

  First we initialize all entries in the table as 0 (in this problem we are looking for max values when all item values are positive, so 0 is safe).

  We modify the algorithm to check this table before launching into computing the solution.

```
optknapsack(k,w)

   IF table[k][w] != 0 THEN
     RETURN table[k][w]

   IF w[k] <= w THEN
     with = v[k] + optknapsack(k-1, w-w[k])
   ELSE
     with = 0

   without = optknapsack(k-1,w)

   table[k][w] = max{with, without}

   RETURN max{with, without}
 END
```

- Effectively, the table will prevent a subproblem optknapsack(k,w) to be computed more than once.

- Analysis: This will run in $O(n \cdot W)$ time as we fill each entry in the table at most once, and there are nW spaces in the table.