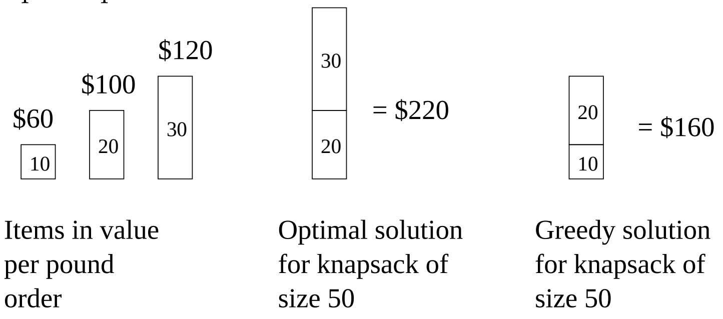
Dynamic Programming: 0-1 Knapsack

Laura Toma, csci2200, Bowdoin College

- The $0-1$ 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. Assume that the weights and values of the n items are given in two arrays; furthermore, assume that weights w_i and the total weight W are integers.

- One very tempting idea is the following: Take the item with the largest value-per-pound, then the second one, and so on, while there is space left in the knapsack. This is called a *greedy algorithm*; we'll talk more about greedy algorithms and see some examples where they do work, next week. Here is a counter-example showing that the strategy above does not work for the knapsack problem:


$60
10
$100
20
$120
30
Items in value per pound order
30
20
= $220
Optimal solution for knapsack of size 50
20
10
= $160
Greedy solution for knapsack of size 50

  Note: One can imagine a version of the problem called the FRACTIONAL-KNAPSACK PROBLEM in which we can take $\frac{2}{3}$ of \$120 object and get \$240 solution.Does the counter-example above still work?

- We'll solve the problem with dynamic programming

- **Optimal substructure**: How would you argue that the problem has optimal sub-structure?

- **Recursive formulation:** Often the hardest part is coming up with the recursive formulation. We note that as we put an item in the knapsack, the set of remaining items to choose from is smaller, and the weight of the knapsack is smaller. This suggests that there are two arguments to the recursive problem: the set of items to chose from, and the available capacity of the knapsack.

- Simplify: As usual, we'll focus on finding out the optimal value that we can place in the knapsck; At the end, we'll show that we can augment the solution to find the actual set of items.

- Notation: 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'll call *optknapsack*(n, W).
- The overall strategy: 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)

    //take item k in the backpack
    IF weight[k] <= w THEN
      with = value[k] + optknapsack(k-1, w - weight[k])
    ELSE
      with = 0
    END IF

    //do not take item k in the backpack
    without = optknapsack(k-1,w)

    //the optimal solution is the best of the two
    RETURN max{with, without}
  END Knapsack
  ```

- **Running time analysis:**

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

  We'll look at the worst case where $w[i] = 1$ for all $1 \le i \le n$. 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 $T(n, W) = \Omega(2^{\min(n,W)})$.

  How many different sub-problems are there?

- We now show how to improve the exponential running time with dynamic programming: We create a table T of size $[1..n][1..W]$. 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 as initial value is safe).

  Effectively, the table will prevent a subproblem optknapsack(k,w) to be computed more than once. We modify the algorithm to check this table before launching into computing the solution.

```
optknapsackDP(k,w)

   //if solution already computed, return it
   IF table[k][w] != 0 THEN
     RETURN table[k][w]

   //take item k in the backpack
   IF w[k] <= w THEN
     with = v[k] + optknapsackDP(k-1, w-w[k])
   ELSE
     with = 0

  //do not take item k in the backpack
   without = optknapsackDP(k-1,w)

   //store solution in the table
   table[k][w] = max{with, without}

   RETURN table[k][w]
 END
```

- **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.
- **Finding the set of items** : Describe how to extend this solution in order to compute the set of items in the backpack (corresponding to the optimal solution).
- **Iterative DP:** Describe how to extend the solution to eliminate the recursion; that is, describe how to fill the table iteratively, without recursion.