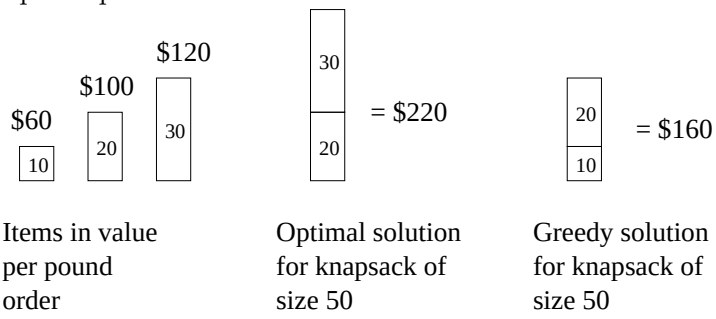
Dynamic Programming: 0-1 Knapsack

- 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:



  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?

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

- **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 in addition.

- **Recursive formulation:** 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.

- 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. When we reach item k, we have two choices: either it's part of the optimal solution, or not. We need to compute both options, and chose the best one.

optknapsack(k, w)

```
//basecase
if (w ≤ 0): return 0
if (k ≤ 0): return 0
//choice 1: take item k in the backpack
IF (weight[k] ≤ w): with = value[k] + optknapsack(k − 1, w − weight[k])
ELSE: with = 0
//choice 2: 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 }
```

- **Running time analysis:**

  Let $T(n, W)$ be the running time of optknapsack(n, W). We have:

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

  The worst case is when $w[i] = 1$ for all $1 \leq i \leq n$.

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

Since $T(n-1,W) > T(n-1,W-1)$ we get that

$$T(n,W) > T2(n-1,W-1)$$

The recursion depth is $\min(n,W)$ steps, and at every step, the time doubles. Therefore $T(n,W) = \Omega(2^{\min(n,W)})$. This is exponential.

- **Why exponential?** How many different sub-problems are there? Answer: Each call to *optknapsack*() has a value for k and a value for w. There are n values for k and W values for w. In total $n \cdot W$ different sub-problems.

- Overlapping calls: Sketch the recursion tree for $n = 5$ and $w[1] = w[2] = w[3] = w[4] = 1$

Dynamic programming: memo-ized recursive solution

- Idea: we'll use a table to store solutions to subproblems. Effectively, the table will prevent a subproblem optknapsack(k,w) to be computed more than once. Note that since a subproblem is of the form $optknapsack(k,w)$, i.e. has two arguments, the table must be two dimensional.

- **The table:** We create a table T of size $[1..n][1..W]$. Entry $T[i][w]$ will store the result of optknapsack(i,w).

- **Initialize:** 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).

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

optknapsackDP(k, w)

//basecase

if $(w \leq 0)$: return 0

if $(k \leq 0)$: return 0

//if solution already computed, return it

IF $(table[k][w] \neq 0)$: RETURN $table[k][w]$

//choice 1: take item k in the backpack (if possible)

IF $(w[k] \leq w)$: $with = v[k] +$ `optknapsackDP`$(k-1, w-w[k])$

ELSE: with $= 0$

//choice 2: 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 $table[k][w]$

- **Analysis:** Ignoring the recursion, each subproblem is computed in $O(1)$ time. 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.

Dynamic programming: iterative solution

It is possible to eliminate recursion by filling the table in the order in which recursion fills it.

Question: In what order is the table filling up? Where are the base-cases in the table?
Answer: Table is filling up left to right and top-down.

optknapsackDP_iterative

create table[0..n][0..W] and initialize all entries to 0

for $(k = 1; k < n; k++)$

for $(w = 1; w < W; w++)$

with = v[k] + table[k-1][w-w[k]]

without = table[k-1][w]

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

RETURN $table[n][W]$

Analysis: $O(n \cdot W)$

From finding the optimal cost to 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 value).