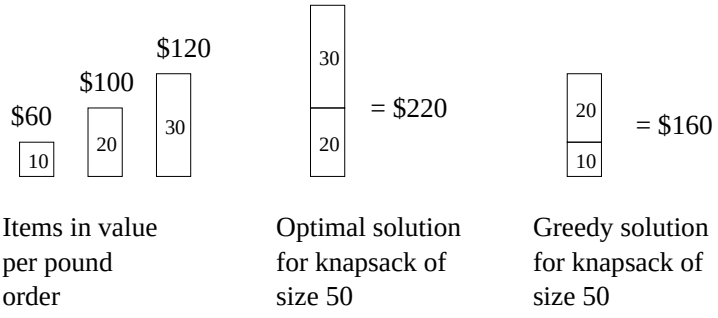
DP Example 3: The 0-1 Knapsack Problem

Module 4:Techniques

1 The $0-1$ knapsack problem

- We are given a knapsack of capacity W (that is, it can hold at most W pounds), which we can fill by choosing any subset of n items; for each item, we know its weight and value; these are given as two arrays, $v[]$ and $w[]$, where item i is worth $v[i]$ and has weight $w[i]$ pounds. The goal is to fill the knapsack in such a way so that the value of all items in the knapsack is maximized. Assume that weights $w[i]$ and the total weight W are integers.
- If this sounds a little abstract, imagine you are in Alibaba's cave; you can't carry everything, so you want to fill your backpack with stuff in such a way that you maximize the value.
- Or perhaps, you pack for a camping trip. You can't take all the stuff you want, so you want to maximize the value you pack. You get the idea.
- How to decide which items to take and which ones to leave? That's the question.
- Straightfirward solution: Enumerate all possible subsets of items; calculate the total weight and the total value of each subset; find the max value among all subsets whose total weight is $< W$. Analysis: How many subsets in a set of n elements? It's 2^n. So the straightfirward solution is exponential.
- How can we improve this? We'll use dynamic programming.
- **Wait a minute. Would something a lot simpler work?** 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*; Sometimes greedy strategies work (we'll talk more about greedy algorithms next week); but not in this case—below is a counter-example showing that the strategy above does not work for the knapsack problem:



- One can imagine a version of the problem called the FRACTIONAL-KNAPSACK PROBLEM in which we can take fractions of items. In this case, we would take $\frac{2}{3}$ of \$120 object and get \$240 solution. The counter-example above would not work anymore, and in fact we'll show that the fractional knapsack problem can be solved with a greedy strategy. More on this next week.

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

- **Optimal substructure**: How can we argue that the 0-1 knapsack problem has optimal sub-structure?

  Consider an optimal solution O, which is essentially a subset of items in $I = \{1, 2, ..., n\}$. We know that O is the maximal way to pack a knapsack of capacity W with items in I.

  Let i be an item in the optimal solution O. What can we say about the the remaining items in O, and the remaining items in I?

  Answer: $O - \{i\}$ must be the optimal way to pack a backpack of weight $W - w[i]$ with items in the set $I - \{i\}$.

  Why? By contradiction. If there was a better way to pack a backpack of weight $W - w[i]$ with items in $I - \{i\}$, then we could take that, add element i, and that would give us a solution that's better than O —- contradiction.

2 Recursive formulation:

The hardest part is coming up with a 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.

- Here's one way we can express the problem: 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.

```
//returns the maximal value obtainable when filling a knapsack of capacity w
//using (a subset of) items 1 through k.
optknapsack(k, w)

    //basecase(s)

    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 }
```

- Correctness: The algorithm considers all possible choices at every step, and picks the best. Once it makes a choice it recurses on the rest (this is correct because of optimal substructure).
- **Running time analysis:** Note that the function optknapsack has two parameters, both k and w. Its running time is function of both.

  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 there are $n \cdot W$ different sub-problems. Each call to *optknapsack*() runs in $\Theta(1)$ if we ignore the recursion, so it must be that the exponential time comes from solving the same sub-problems over and over.

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

3 DP recursive solution with memoization

- How: we'll use a table to store solutions to subproblems, which 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. We modify the algorithm to check this table before launching into computing the solution.

- **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). This table will be shared by all subproblems (think of it as a global variable). 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).

```
//global variable table[1..n][1..W] initialized to 0. Also global v[1..n] and w[1..n].
//returns the maximal value obtainable when filling a knapsack of capacity w
//using (a subset of) items 1 through k.
optknapsackDP(k, w)

    //basecase(s)

    if (w == 0): return 0

    if (k ≤ 0): return 0

    //if solution already computed, return it

    IF (table[k][w] ≠ 0): RETURN table[k][w]

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

    IF (w[k] ≤ 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:** We call $knapsack(n, W)$ and this will trigger recursive calls to smaller problems, which will trigger recursive calls to smaller problems, and so on, until it will reach the base cases(s). On the way back, as recursion returns from the base cases to the initial call $knapsack(n, W)$, the recursion will fill in the table. Let's analyse the running time of the subproblems in the order in which recursion returns, ie assuming that when we compute $knapsack(k, w)$ the values of $knapsack(k-1, w-w_i)$ and $knapsack(k-1, w)$ have already been computed and are retrieved from the table in $\Theta(1)$ each. Then the value of $knapsack(k, w)$ can be computed in $O(1)$ time. In total there are $n \cdot W$ sub-problems (ie entries in the table); we account for the cost of recursion separately by counting that all $n \cdot W$ entries will be computed. As recursion returns, the cost of computing each entry is $\Theta(1)$. Overall computing $knapsack(n, W)$ will run in $O(n \cdot W)$ time.

4 DP iterative solution (bottom-up, no recursion)

- It is possible to eliminate the recursion by filling the table in such an order so that there is no need of recursion. To see this let's try to understand the order in which recursion fills the table.

- 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 (from w=1 to w=W) and top-down (from i=1 to i=n).

optknapsackDP_iterative

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

for $(k = 1; k \leq n; k++)$

  for $(w = 1; w \leq W; w++)$

    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:** $O(n \cdot W)$ Note that this is the same as for the top-down recursive approach. Basically we just get rid of recursion and its overhead, but the asypmtotic complexity stays the same.

5 From finding the optimal value to finding the set of items

When we started this we said we would first focus on computing the **value** of the optimal knapsack, and then we'd extend it to compute the full solution, i.e. the set of items corresponding to the optimal value. Let's do this.

5.1 Computing full solution (without storing additional information):

Input: The table $table[1..n][1..W]$ as computed above, where $table[i][x]$ stores the max value to pack a knapsack of weight x using items $1..i$. Also $w[1..n]$ and $v[1..n]$
Output: the set of items corresponding to $table[n][W]$

$i = n, w = W$
while $(i > 0)$ do:

    //is the value $table[i][w]$ achieved by including item i or not?

    if $table[i][w] == v[i] + table[i-1][w - w[i]]$:

        output item i

        $w = w - w[i], i = i - 1$

    else: $i = i - 1$

Running time: $O(n)$, with no extra space

5.2 Computing full solution (with storing additional information)

Instead of $table[1...n][1..W]$ storing a value, we make it store a pair: $table[i][w].value$ will store the max value packable for a knapsack of weight w using items $1..i$, and $table[i][w].choice$ will store whether this value is achieved *with* or *without* element i. We can extend $knapsack(i, w)$ so that if the maximum is achieved by including item i, we'll set $table[i][w].choice =$ "*with*"; otherwise we'll set it as "*without*". Then we can get the full solution like so:

Input: The table $table[1..n][1..W]$ as computed above. Also $w[1..n]$.
Output: the set of items corresponding to $table[n][W]$

$i = n, w = W$
while ($i > 0$ and $w > 0$) do:

    //is the value $table[i][w]$ achieved by including item i or not?

    if $table[i][w].choice ==$ "*with*":

        output item i

        $w = w - w[i], i = i - 1$

    else: $i = i - 1$

Running time: $O(n)$, with $\Theta(1)$ extra space per entry in the table.