In some situation finding approximately optimal solution to a maximization or minimization problem is good enough in the application. This may be useful if we either lack all the information as we have to make decisions (online algorithms) or we don't have the time to find a fully optimal solution (say as the problem is too hard). In the remaining part of the course, we will explore algorithms for such situations.

1 Knapsack

First we consider the KNAPSACK problem, which is defined in section 6.4 of the book. This problem is given by n items each with a weight w_i and value v_i, and a maximum allowed weight W. The problem is to selected a subset I of items of maximum total value $\sum_{i\in I} v_i$ whose total weight $\sum_{i\in I} w_i \leq W$ is at most the weight given.

With integer weights w_i and W, section 6.4 offers a dynamic programming algorithm that solves the problem in $O(nW)$ time using dynamic programming. This works well when W is small, but is not polynomial time: when W is large notice that the weights are described with $O(n \log W)$ bits as it is given by n numbers with $\log W$ bits each, and the W in the running time is not polynomial in $\log W$. Or putting it differently, the dynamic programming algorithm in section 6.4 works well when W is small, but can be slow if W is huge. Section 11.8 offers an alternate dynamic programming algorithm for the knapsack problem running in time $O(n^2 \max_i v_i)$ time, assuming that the values v_i are integer. As before this works when $v^* = \max_i v_i$ is not too big but is not polynomial in the problem description which is only $O(n \log v^*)$. In fact, Section 8.8 consider the special case with $w_i = v_i$ and asking the decision question if value W can be achieved. This is the SUBSETSUM problem, which is proved to be NP-complete by a reduction from SAT.

Here we give a simple 2-approximation algorithm for the problem. See also Section 11.8 of the book that we'll cover later that gives a much better approximation algorithm. Our simple approximation algorithm will be a form of greedy algorithm. We will consider two greedy ideas. For both algorithms, we first delete all items with weight $w_i > W$ as they cannot be considered by any solution.

(a) Consider items in order of their value, and add to the set I till the weight is not exceeded.

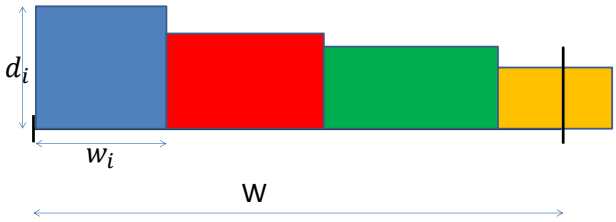
While this algorithm is natural, it is also easy to give a very bad example for this. Say one item has eight $w_n = W$ and value $v_n = W$, and many small items of value $v_i = W - 1$ and weight $w_i = 1$. The item of maximum value is item n which alone fills the knapsack. However, by taking many of the other items, we can take as many as $\min(n-1, W)$ items each of value $v_i = (W-1)$ reaching a much higher total value.

This leads us to a different greedy idea: we should consider the density of value in each item $d_i = v_i/w_i$. Our second greedy algorithm considers items in this order

(b) Consider items in order of their value density d_i, and add to the set I till the weight is not exceeded.

The process can we well represented by a figure, if we have the items each represented by a box of length w_i and height d_i where now the area of the box is $w_i d_i = v_i$ is its value. Items are added to the knapsack in the order decreasing density, as suggested by the figure, where the orange item hanging out on the right of the W size knapsack is the first item that didn't fit into the knapsack.

Unfortunately, this greedy algorithm can also be very bad. For an example of this situation, all we need is two items $w_1 = v_1 = W$, while $v_2 = 1 + \epsilon$ and $w_2 = 1$. Now item 1 has density $d_1 = 1$, while



item 2 has density $d_2 = 1 + \epsilon$. However, after fitting item 2 in the knapsack, item 1 no longer fits! So we end up with a total value of value $1 + \epsilon$, while value W was also possible.

Interestingly, the better of the two greedy algorithm is a good approximation algorithm.

Claim. Running both (a) and (b) greedy algorithm above, and taking the solution of higher value is a 2-approximation algorithm, finding a solution to the knapsack problem with at least 1/2 of the maximum possible value.

Proof. Consider the two greedy algorithms, and let V_a and V_b the value achieved by greedy algorithms (a) and (b) respectively, and let Opt denote the maximum possible value. For algorithm (b) let I be the set of items packed into the knapsack, and let j be the first item that didn't fit into the knapsack (the orange item in the example). Clearly $\sum_{i \in I} v_i = V_b$ and $v_j \leq V_a$, as algorithm (a) starts by taking the single item of maximum value.

I tricky part in thinking about approximation algorithms is how we can compare our solution to the optimum value, without being able having to compute it (as it is NP-hard to compute the optimum value). We'll show below that $\sum_{i \in I} v_i + v_j \geq Opt$, implying that $V_a + V_b \geq Opt$, so the larger of V_a and V_b must be at least 1/2 of OPT.

Claim. Using the notation from algorithm (b) above, $\sum_{i \in I} v_i + v_j \geq Opt$.

Proof. The main observation is that if we could cut a part of the last item so as to exactly fill the knapsack, that would clearly be the optimum solution if taking a partial item was allowed: it uses all items of density $> d_j$ and fills the remaining part of the knapsack with value density d_j, and all items not fully included have density $\leq d_j$. This shows that the optimum value is for the case when we are allowing cutting an item is $\sum_{i \in I} v_i$ plus a fraction of the last item. The true optimum, if cutting items is not permitted, can only be smaller, so we get $\sum_{i \in I} v_i + v_j \geq Opt$ as claimed.