1 Overview

Last lecture, we covered some examples of dynamic programming, and applied a recipe for how to come up with DP solutions (for longest common subsequence and maximum weight independent set in trees). This handout will cover two more problems: knapsack (unbounded and 0/1 knapsack).

In general, here are the steps to coming up with a dynamic programming algorithm:

1. **Identify optimal substructure:** how are we going to break up an optimal solution into optimal sub-solutions of sub-problems? We’re looking for a way to do this so that there are overlapping sub-problems, so that a dynamic programming approach will be effective.

2. **Recursively define the value of an optimal solution:** Write down a recursive formulation of the optimum, in terms of sub-solutions.

3. **Find the optimal value:** Turn this recursive formulation into a dynamic programming algorithm to compute the value of the optimal solution.

4. **Find the optimal solution:** Once we’ve figured out how to find the cost of the optimal solution, we can go back and figure out how to keep enough information in our algorithm so that we can find the solution itself.

5. **Tweak the implementation:** Often it’s the case that the solutions that we come up with in the previous steps aren’t implemented in the best way. Maybe they are storing more than they need to, like we saw with our first pass at the Floyd-Warshall algorithm. In this final step (which we won’t go into in too much detail in CS161), we go back through the DP solution we’ve designed, and optimize it for space, running time, and so on.

In this class, we’ll focus mostly on 1, 2, and 3. We’ll see a few examples of 4, and occasionally wave our hands about 5.

2 The Knapsack Problem

This is a classic problem, defined as the following:

We have $n$ items, each with a value and a positive weight. The $i$th item has weight $w_i$ and value $v_i$. We have a knapsack that holds a maximum weight of $W$. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let’s say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>$B$</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>$C$</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>$D$</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it’s having knapsacks of smaller capacities, or maybe it’s having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we will see, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

---

1 We won’t talk too much about this step in CS161, even though it is often important in practice.
2.1 The Unbounded Knapsack Problem

In the example above, we can pick two of item B and two of item D. Then, the total weight is 10, and the total value 42.

We define \( K(x) \) to be the optimal solution for a knapsack of capacity \( x \). Suppose \( K(x) \) happens to contain the \( i \)th item. Then, the remaining items in the knapsack must have a total weight of at most \( x - w_i \). The remaining items in the knapsack must be an optimum solution. (If not, then we could have replaced those items with a more highly valued set of items.) This gives us a nice subproblem structure, yielding the recurrence

\[
K(x) = \max_{i : w_i \leq x} (K(x - w_i) + v_i).
\]

Developing a dynamic programming algorithm around this recurrence is straightforward. We first initialize \( K(0) = 0 \), and then we compute \( K(x) \) values from \( x = 1, \ldots, W \). The overall runtime is \( O(nW) \).

**Algorithm 1: UNBOUNDEDKNAPSACK\( (W, n, w, v) \)**

\[
\begin{align*}
K[0] & \leftarrow 0 \\
& \text{for } x = 1, \ldots, W \text{ do} \\
& \quad K[x] \leftarrow 0 \\
& \quad \text{for } i = 1, \ldots, n \text{ do} \\
& \quad \quad \text{if } w_i \leq x \text{ then} \\
& \quad \quad \quad K[x] \leftarrow \max\{K[x], K[x - w_i] + v_i\}
\end{align*}
\]

return \( K[W] \)

**Remark 1.** This solution is not actually polynomial in the input size because it takes \( \log(W) \) bits to represent \( W \). We call these algorithms “pseudo-polynomial.” If we had a polynomial time algorithm for Knapsack, then a lot of other famous problems would have polynomial time algorithms. This problem is NP-hard.

2.2 The 0-1 Knapsack Problem

Now we consider what happens when we can take at most one of each item. Going back to the initial example, we would pick item A and item C, having a total weight of 10 and a total value of 40.

The subproblems that we need must keep track of the knapsack size as well as which items are allowed to be used in the knapsack. Because we need to keep track of more information in our state, we add another parameter to the recurrence (and therefore, another dimension to the DP table). Let \( K(x,j) \) be the maximum value that we can get with a knapsack of capacity \( x \) considering only items at indices from \( 1, \ldots, j \). Consider the optimal solution for \( K(x,j) \). There are two cases:

1. Item \( j \) is used in \( K(x,j) \). Then, the remaining items that we choose to put in the knapsack must be the optimum solution for \( K(x - w_j,j - 1) \). In this case, \( K(x,j) = K(x - w_j,j - 1) + v_j \).
2. Item \( j \) is not used in \( K(x,j) \). Then, \( K(x,j) \) is the optimum solution for \( K(x,j - 1) \). In this case, \( K(x,j) = K(x,j - 1) \).

So, our recurrence relation is: \( K(x,j) = \max\{K(x - w_j,j - 1) + v_j, K(x,j - 1)\} \). Now, we’re done: we simply calculate each entry up to \( K(W,n) \), which gives us our final answer. Note that this also runs in \( O(nW) \) time despite the additional dimension in the DP table. This is because at each entry of the DP table, we do \( O(1) \) work.


Algorithm 2: \textsc{ZeroOneKnapsack}(W, n, w, v)

\begin{verbatim}
for x = 1, \ldots, W do
    K[x, 0] ← 0
for j = 1, \ldots, n do
    K[0, j] ← 0
for j = 1, \ldots, n do
    for x = 1, \ldots, W do
        K[x, j] ← K[x, j - 1]
        if w_j ≤ x then
            K[x, j] ← max{K[x, j], K[x - w_j, j - 1] + v_j}

return K[W, n]
\end{verbatim}