CS 341: ALGORITHMS (S18) — LECTURE 10
DYNAMIC PROGRAMMING III: COINS AND KNAPSACKS

ERIC BLAIS

In this lecture, we complete our exploration of the dynamic programming technique by revisiting problems related to coins and knapsacks.

1. Making change

Recall the Making Change problem that we saw at the beginning of Lecture 6:

Problem 1 (Making change)**.** *Given coins with denominations $d_1 > d_2 > \cdots > d_n = 1$ and a value $V \geq 1$, determine the minimum number of coins whose denominations sum to V.*

We saw that there is a simple greedy algorithm that correctly solves the problem for some, but not all, coin denominations. We can use the dynamic programming method to solve the problem correctly for all coin denominations.

1.1. Dynamic programming algorithm.

The subproblems for the problem correspond to making change for values $v = \{0, 1, 2, \ldots, V\}$. Define $C(v)$ to be the minimum number of coins required to make change for value v. Then $C(0) = 0$ and for every $v \geq 1$, we can try returning a coin of every possible denomination so that

$$C(v) = \min_{i=1,\ldots,n : d_i \leq v} \{1 + C(v - d_i)\}.$$

The resulting algorithm is as simple as the greedy algorithm, but not as efficient.

Algorithm 1: CoinChange$(d_1, \ldots, d_n, V)$

```
C[0] ← 0;
for v = 1, ..., V do
    C[v] ← min_{i≤n : d_i≤v} {1 + C[v − d_i]};
return C[V];
```

Theorem 10.1. *The* CoinChange *algorithm solves the making change problem.*

Proof. Let $\mathrm{OPT}(v)$ denote the minimum number of coins required to return value v in coins with some fixed denominations $d_1, \ldots, d_n$. To establish the correctness of the algorithm, we prove that $C[v] = \mathrm{OPT}(v)$ by induction on $v = 0, 1, 2, \ldots, V$. In the base case, $\mathrm{OPT}(0) = C[0] = 0$ since zero coins are required to make change of value 0.

For the induction step, assume that $C[0] = \mathrm{OPT}(0)$, $C[1] = \mathrm{OPT}(1)$, $\ldots$, $C[v-1] = \mathrm{OPT}(v-1)$. Consider now a minimum set of coins with total value $v \geq 1$. This set includes a coin of denomination d_k, $1 \leq d_k \leq v$, for some $k \in \{1, 2, \ldots, n\}$. This means that $\mathrm{OPT}(v) = 1 + \mathrm{OPT}(v - d_k)$ and that for each $i \neq k$, $\mathrm{OPT}(v) \leq 1 + \mathrm{OPT}(v - d_i)$. By the induction hypothesis, we then have

$$C[v] = \min_{i : d_i \leq v} \{1 + C[v - d_i]\} = \min_{i : d_i \leq v} \{1 + \mathrm{OPT}(v - d_i)\} = 1 + \mathrm{OPT}(v - d_k) = \mathrm{OPT}(v). \quad \square$$

Theorem 10.2. *The time complexity of the* CoinChange *algorithm is* $\Theta(nV)$.

Proof. The for loop is run V times, and for each loop iteration we compare up to n values, so the time complexity of the algorithm is $O(nV)$. Furthermore, when $V \geq 2d_1$, then for $V/2$ iterations of the loops, all n denominations satisfy the condition $d_i \leq v$ so that on those instances the algorithm has time complexity at least $\frac{V}{2} \cdot n = \Omega(nV)$. □

1.2. **Decision problem variant.** If we don't have a coin with denomination 1, then we can't make change for all possible values. In this case, it makes sense to ask whether we can make change that adds up to some value V or not.

Problem 2 (Making change—decision variant)**.** *Given coins with denominations* $d_1 > d_2 > \cdots > d_n > 1$ *and a value* $V \geq 1$*, determine whether it is possible to choose some coins whose denominations sum to* V*.*

Dynamic programming can be used to solve this problem. In fact, the algorithm we designed for the original version of the making change problem can easily be modified to solve this problem as well.

Algorithm 2: CoinChangeDec$(d_1, \ldots, d_n, V)$

```
C[0] ← True;
for v = 1, ..., V do
    C[v] ← False;
    for i = 1, ..., n do
        if d_i ≤ v then C[v] = C[v] ∨ C[v − d_i];
return C[V];
```

Theorem 10.3. *The* CoinChangeDec *algorithm solves the decision variant of the making change problem.*

Proof. Let CanChange(v) be `True` if it is possible to choose coins of some fixed denominations $d_1, \ldots, d_n$ with total value v; `False` otherwise. To establish the correctness of the algorithm, we prove that $C[v] =$ CanChange(v) by induction on $v = 0, 1, 2, \ldots, V$. In the base case, CanChange$(0) = C[0] =$ `True` since an empty set of coins has total value 0.

For the induction step, the induction hypothesis states that $C[v'] =$ CanChange(v') for all $v' < v$. We consider two cases.

(1) If CanChange(v) is `True`, then there exists a non-empty set S of coins with total value v, and we can let d_k be the denomination of one of the coins in this set. Then CanChange$(v - d_k)$ is `True` (by removing a coin of denomination d_k from S) and by the induction hypothesis $C[v - d_k] =$ `True` as well, so

$$C[v] = \bigvee_{i \leq n \,:\, d_i \leq v} C[v - d_i] = \texttt{True} = \text{CanChange}(v).$$

(2) If CanChange(v) is `False`, then for any denomination d_i we must have that CanChange$(v - d_i)$ is also `False`, otherwise we would take the set S of coins that make change for $v - d_i$ and add a coin of denomination d_i to make change for v. So by the induction hypothesis,

$$C[v] = \bigvee_{i \leq n \,:\, d_i \leq v} C[v - d_i] \bigvee_{i \leq n \,:\, d_i \leq v} \text{CanChange}(v - d_i) = \texttt{False} = \text{CanChange}(v). \quad \square$$

2. Knapsack

We saw a variant of the knapsack problem where we were allowed to divide items and only include a fraction of them in our knapsack. In the standard version of the problem, we no longer have that power: we either include or exclude an item in the knapsack.

Definition 10.4 (Knapsack)**.** An instance of the *knapsack problem* is a set of n items that have positive integer weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, as well as a maximum weight capacity W of the knapsack. A valid solution to the problem is a subset $S \subseteq \{1, 2, \ldots, n\}$ of the items that you put in your backpack that satisfies $\sum_{i \in S} w_i \leq W$ and maximizes the total value $V = \sum_{i \in S} v_i$ among all sets that satisfy the weight condition.

To distinguish this problem explicitly from the fractional knapsack problem, it is also sometimes called the 0-1 *knapsack* problem.

We can solve the knapsack problem using the dynamic programming technique. Let's consider the natural way to do this, following the approach that we used in previous lectures. A natural way to break down the problem into smaller subproblems is to consider only the items $1, \ldots, k$ for each $k \in \{1, 2, \ldots, n\}$ (along with the trivial subproblem when $k = 0$).

Then, as in the other problems we consider, we have a simple observation that can let us solve the subproblem with the first k items when we already solved it with the first $k - 1$ items: either k is in the optimal subset $S_k \subseteq \{1, 2, \ldots, k\}$ of items we put in the knapsack, or it is not. If it is not, then the optimal value $V_k = V_{k-1}$. But if it is, we realize that there is a twist that we need to consider: we need to find the maximum subset $S'_{k-1} \subseteq \{1, 2, \ldots, k-1\}$ that fit in a knapsack with capacity $W - w_k$ (not W!) if we are to put these items into the knapsack along with item k.

Therefore, we need to consider subproblems where we consider the first k elements *and* where we fix the capacity of a knapsack to be w, for $k \in \{0, 1, 2, \ldots, n\}$ and for $w = \{0, 1, 2, \ldots, W\}$. We do so by defining

$$M(k, w) = \max_{S \subseteq \{1,2,\ldots,k\} : \sum_{i \in S} w_i \leq w} \sum_{i \in S} v_i.$$

Then for every $w \leq W$ and $k \leq n$,

$$M(0, w) = 0 \qquad \text{and} \qquad M(k, 0) = 0.$$

For $k \geq 1$, we then have two possibilities: either $w_k > w$, in which case the item k does not fit into the knapsack and $M(k, w) = M(k - 1, w)$, or $w_k \leq w$ in which case $M(k, w)$ is the maximum of the optimal value $M(k - 1, w)$ obtained by leaving out item k and the optimal value $v_k + M(k - 1, w - w_k)$ obtained by including item k in the knapsack. So for each $k = \{1, 2, \ldots, n\}$ we have

$$M(k, w) = \begin{cases} M(k-1, w) & \text{if } w_k > w \\ \max\{M(k-1, w), v_k + M(k-1, w - w_k)\} & \text{if } w_k \leq w. \end{cases}$$

The resulting algorithm is as follows.

Theorem 10.5. *The* Knapsack *algorithm solves the knapsack problem.*

Proof. Let $\text{OPT}(k, w)$ denote the maximum value of a subset of items $1, \ldots, k$ that fit into a knapsack with capacity w. We show $M[k, w] = \text{OPT}(k, w)$ for all $k = 0, 1, \ldots, n$ and $w = 0, 1, \ldots, W$ by induction on k and w. In the base cases, when $k = 0$ or $w = 0$, then $M[k, w] = 0 = \text{OPT}(k, w)$.

Algorithm 3: KNAPSACK$(w_1, \ldots, w_n, v_1, \ldots, v_n, W)$

```
for w = 0, 1, 2, ..., W do M[0, w] ← 0;
for k = 0, 1, 2, ..., n do M[k, 0] ← 0;

for k = 1, ..., n do
    for w = 1, ..., W do
        if w_k ≤ w then
            M[k, w] ← max{M[k − 1, w], v_k + M[k − 1, w − w_k]};
        else
            M[k, w] ← M[k − 1, w];
return M[n, W];
```

For the induction step, the induction hypothesis lets us assume that $M[k-1, w'] = \text{OPT}(k-1, w')$ for every $w' \leq w$. Consider now the value $\text{OPT}(k, w)$and a corresponding set $S \subseteq \{1, 2, \ldots, k\}$ of items with total weight at most w and value $\text{OPT}(k, w)$. There are two cases to consider.

(1) If $w_k > w$, then it must be that $k \notin S$ so that $\text{OPT}(k, w) = \text{OPT}(k-1, w)$ and by the induction hypothesis

$$M[k, w] = M[k-1, w] = \text{OPT}(k-1, w) = \text{OPT}(k, w).$$

(2) If $w_k \leq w$, then $\text{OPT}(k, w) = \text{OPT}(k-1, w)$ (if $k \notin S$) or $\text{OPT}(k, w) = v_k + \text{OPT}(k-1, w-w_k)$ (if $k \in S$), whichever is larger. So by the induction hypothesis

$$\begin{aligned} M[k, w] &= \max\{M[k-1, w], v_k + M[k-1, w-w_k]\} \\ &= \max\{\text{OPT}(k-1, w), v_k + \text{OPT}(k-1, w-w_k)\} = \text{OPT}(k, w). \end{aligned}$$

□

Theorem 10.6. *The time complexity of the* KNAPSACK *algorithm is* $\Theta(nW)$.

Proof. The code inside the two nested for loops is run a total of nW times and has complexity $\Theta(1)$. The two initialization for loops have time complexity $\Theta(W)$ and $\Theta(n)$, respectively. So the total time complexity of the algorithm is $\Theta(nW + n + W) = \Theta(nW)$. □

If we want to find the optimal set of items to put in the knapsack, we can again use the backtracking approach to find which items to put in the knapsack based on the $M[k, w]$ values.

3. Pseudopolynomial-time algorithms

We will spend more time exploring polynomial-time algorithms later in the course, but it's worth pausing here to ask whether the algorithms we introduced in this lecture are polynomial-time or not. To answer this question, we need to start with a formal definition.

Definition 10.7 (Polynomial-time algorithm)**.** A *polynomial-time algorithm* is an algorithm A with time complexity that is polynomial in the length of its input.

The key part of the definition is the phrase *the length of its input*: we usually consider the time complexity of an algorithm in terms of n, with n representing different parameters for different problems, but it's not sufficient to show that an algorithm has time complexity polynomial in n to show that it is a polynomial-time algorithm—unless we also show that the input size is polynomial in n.

Is that the case in the Making Change and the Knapsack problems? NO! That's because the values V and W require only $\log V$ and $\log W$ bits to specify in the inputs to the problem, respectively. So while the time complexity of both algorithms is polynomial in n, it is also *exponential* in the length of the other parameter (V or W). These algorithms are two examples of *pseudopolynomial-time algorithms*.

Can we obtain a polynomial-time algorithm for, say, the Knapsack problem? Such an algorithm would have time complexity $O(n^c(\log W)^{c'})$ for some constants c, c'. As we will see in the NP-completeness section of the course, obtaining such an algorithm would be a huge breakthrough, as it would solve the infamous P vs. NP problem and show that *every* problem in NP also can be solved by a polynomial-time algorithm.

4. Bonus: Memoization

All the dynamic programming algorithms we have introduced in this course follow the same pattern of solving the subproblems from smallest to largest, in that order. This is sometimes called "bottom-up dynamic programming", and there is another way to implement essentially all the dyanmic programming algorithms we have seen in a "top-down" fashion using recursve algorithms. What distinguishes the dynamic programming recursive algorithms from the usual ones is that the algorithm stores a global array of values that it computes along the way, so that recursive calls are made only when the value has not already been computed earlier. This top-down dynamic programming technique is known as *memoization*.

For example, the coin change algorithm we described above can be implemented using the memoization technique as follows.

Algorithm 4: CoinChangeRec(v)

```
/* C[0..V] is a global array of precomputed values                    */
/* Initially, C[i] =⊥ for all i ∈ 1,2,...,V and C[0] = 0.             */
/* d_1,...,d_n are also stored globally                               */
if C[v] =⊥ then
    C[v] ← min_{i≤n : d_i≤v} {1 + CoinChangeRec(v − d_i)};
return C[v];
```

In most settings, for the types of examples we have covered in this course, the bottom-up dynamic programming technique is preferable: it is usually simpler to analyze (both for correctness and for time complexity) and the implementations of those algorithms usually outperform their memoization analogues in practice because of lack of overhead with recursion and because of low-level data-access pattern optimizations. But there are settings when the top-down approach can be better—for example if you only need to precompute a small fraction of the subproblems instead of all of them (and it's not clear from the instance which subproblems need to be solved).