CS3230 Design & Analysis of Algorithms **March 17, 2022**

Lecture 8: Dynamic Programming

Lecturer: Warut Suksompong

The goal of this lecture is to introduce you to the concept of dynamic programming. Some of the contents of the lecture presentation are drawn directly from **Chapter 15 of CLRS**, which you should review. The relevant sections of the chapter are:

- Section 15.3: Elements of dynamic programming
- Section 15.4: Longest common subsequence

You may also take a look at the remaining sections of this chapter, which give other applications of the dynamic programming paradigm.

8.1 Knapsack

In this problem, the input consists of a set of pairs of positive integers $(v_1, w_1), \ldots, (v_n, w_n)$ where v_i denotes the *value* of the i-th item and w_i denotes the *weight* of the i-th item. Also provided as input is an integer W. The objective is to find a subset $S \subseteq [n]$ of items that maximizes $\sum_{i \in S} v_i$ while respecting $\sum_{i \in S} w_i \leq W$. (Here, $[n]$ denotes the set $\{1, 2, \ldots, n\}$.) A short description of this problem is also provided in Section 16.2 of CLRS (as the "0-1 Knapsack Problem").

The naïve algorithm for this problem runs in time $\Omega(2^n)$ as it searches over all the subsets of $\{1, \ldots, n\}$ to find one that maximizes the total value while respecting the bound on the total weight. We show next a dynamic programming solution that solves the problem in $O(nW)$ time. (This is an example of a *pseudo-polynomial time* algorithm as we will discuss in Week 10.)

As usual for dynamic programming, we first design a recursive algorithm with high running time, and then rewrite it in a bottom-up fashion more efficiently. The recursive algorithm relies on the following observation about the optimal substructure property of the knapsack problem.

Lemma 8.1.1 (Optimal Substructure)**.** *Consider an instance of the knapsack problem with $n > 0$ items. Suppose S is an optimal solution.*

(i) If $n \notin S$, then S is an optimal solution to the knapsack sub-problem with inputs $\langle (v_1, w_1), \ldots, (v_{n-1}, w_{n-1}), W \rangle$.

(ii) If $n \in S$, then $S = \{n\} \cup S'$ where S' is an optimal solution to the knapsack sub-problem with inputs $\langle (v_1, w_1), \ldots, (v_{n-1}, w_{n-1}), W - w_n \rangle$.

Proof. The proofs are standard "cut-and-paste".
(Proof of (i)) We know $S \subseteq [n-1]$ and $\sum_{i \in S} w_i \leq W$. Now, suppose S is not an optimal solution to the sub-problem. So, there exists some other set $T \subseteq [n-1]$ such that $\sum_{i \in T} v_i > \sum_{i \in S} v_i$ while also satisfying $\sum_{i \in T} w_i \leq W$. But then T is also a solution to the original knapsack problem with a higher total value, contradicting the assumption that S is optimal.
(Proof of (ii)) Same as in the argument of (i), except that we argue that $S \setminus [n]$ must be an optimal solution to the sub-problem. □

The optimal substructure lemma immediately leads to the following recursive algorithm. Here, *ValWts* is an array with the i-th element being a pair with two attributes, *val* denoting v_i and *wt* denoting w_i.

```
KNAPSACK-REC(ValWts, W)
 1  n = ValWts.length
 2  if n = 0 or W ≤ 0
 3       return ∅
 4  OptWithout = KNAPSACK-REC([ValWts[1], . . , ValWts[n − 1]], W)
 5  OptWith = KNAPSACK-REC([ValWts[1], . . , ValWts[n − 1]], W − ValWts[n].wt)
 6  ValWithout = 0
 7  for i ∈ OptWithout
 8       ValWithout = ValWithout + ValWts[i].val
 9  ValWith = ValWts[n].val
10  for i ∈ OptWith
11       ValWith = ValWith + ValWts[i].val
12  if ValWithout > ValWith
13       return OptWithout
14  else return OptWith ∪ {n}
```

KNAPSACK-REC simply implements the recursion given by the optimal substructure lemma, with the recursion bottoming out when either there are no more items or the weight threshold becomes non-positive. If we denote $T(n, W)$ as the running time, we get the recurrence:

$$T(n, W) \geq T(n-1, W) + T(n-1, W-1) + \Omega(1)$$

in the worst case where the weight of the n-th item is 1. It can be easily checked using the substitution method that $T(n, W) \geq \Omega(2^{\min(n,W)})$.

The exponential time seems too much because the number of distinct sub-problems is at most $n(W+1)$ since in any recursive call, the first argument is one of the n prefixes of *ValWts* and the second argument is a non-negative integer at most W. Denote by $m[i, j]$ the value of an optimal solution to the knapsack sub-problem with inputs $\langle (v_1, w_1), \ldots (v_i, w_i), j \rangle$. Then, we directly get the following recurrence from the optimal substructure lemma:

$$m[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j-w_i] + v_i, m[i-1, j]\} & \text{if } w_i \leq j \\ m[i-1, j] & \text{otherwise} \end{cases} \tag{8.1}$$

The third case handles the situation where the weight of the i-th item itself exceeds the threshold j; in this case, we know that the solution cannot contain item i. Adding the third case explicitly also ensures that j always stays non-negative.

We now have a recipe for recovering the $m[i, j]$ values in a bottom-up fashion. To derive $m[i, j]$, we only need to ensure that $m[i', j']$ values are known where $i' = i - 1$ and $j' < j$. The following table from the lecture slides shows the values of m in a tabular format (rows are i, columns are j) when the input is $\langle (4, 12), (2, 1), (10, 4), (1, 1), (2, 2), 15 \rangle$.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	2	2	10	12	12	12	12	12	12	12	12	12	12	12
4	0	2	3	3	10	12	13	13	13	13	13	13	13	13	13	13
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	**15**

The table is filled out from top left to bottom right, going down the rows, with each row filled left-to-right. This ensures that the sub-problems needed to evaluate $m[i, j]$ are already solved when the entry at (i, j) is being filled out.

Now, we discuss how to recover an optimal solution to the knapsack problem, rather than just its value. Note that in the second case of Equation (8.1), if the first argument of the max is the winner, then item i is in the knapsack solution, but if the second argument is the winner, item i is not. In the first and third case, item i is not in the knapsack solution. In the code below, each entry in the table $M[i, j]$ is a pair where the first entry is the quantity $m[i, j]$ from above, and the second is a boolean value indicating whether i is contained in the returned solution or not. Using this information, we can recover the set of items whose value corresponds to $m[n, W]$.

```
KNAPSACK-DP(ValWts, W)
 1  n = ValWts.length
 2  M = empty (n + 1) × (W + 1)-sized table with rows index 0..n and columns indexed 0..W
 3  for i = 0..n
 4      M[i, 0] = ⟨0, FALSE⟩
 5  for j = 0..W
 6      M[0, j] = ⟨0, FALSE⟩
 7  for i = 1..n
 8      for j = 1..W
 9          ValWithout = M[i − 1, j].first
10          if ValWts[i].wt > W
11              M[i, j] = ⟨ValWithout, FALSE⟩
12          else
13              ValWith = M[i − 1, j − ValWts[i].wt].first + ValWts[i].val
14              if ValWith > ValWithout
15                  M[i, j] = ⟨ValWith, TRUE⟩
16              else
17                  M[i, j] = ⟨ValWithout, FALSE⟩
18  Opt = ∅
19  i = n
20  j = W
21  while i > 0
22      if M[i, j].second = TRUE
23          Opt = Opt ∪ {i}
24          i = i − 1
25          j = j − ValWts[i].wt
26      else
27          i = i − 1
28  return Opt
```

The **while** loop at the end of the code traces back up the table to decide whether each item contributed or not to the calculation of $m[n, W]$. The running time of the algorithm is clearly $O(nW)$.

Remark. If we wanted to recover just $m[n, W]$, and not the subset of corresponding items, we can optimize the space usage to $O(W)$. In particular, we only need to maintain two rows of the table at a time. The running time still remains the same.

Remark. At the end of the lecture, we also talked about the coin changing problem. This was a simple example where the slides are self-explanatory.