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

1. Optimal binary search trees

Here’s a trick question: which of the following two binary search trees is best?

Without any additional information on how the binary search tree will be used, the first example is better: its depth is 3 (versus 5 for the second tree) which means that the worst-case cost of querying elements in that tree is better in that tree. But what if we know with which probability each element is queried by the application, and we find that element 1 will be queried more often than any other element, and by a large margin: in this case, the second tree will be more efficient.

We can formalize the problem of finding the optimum binary search tree for a set of items with known query probabilities as follows.

Definition 10.1 (Optimal binary search tree problem). Given items $1, 2, \ldots, n$ and probabilities $p_1, \ldots, p_n$, construct a binary search tree $T$ that minimizes the search cost

$$\sum_{i=1}^{n} p_i \cdot \text{depth}_T(i)$$

where $\text{depth}_T(i)$ is the depth of (= number of queries required to reach) element $i$ in the tree $T$.

(Note that when we build a binary search tree, we are not allowed to place the items in arbitrary order; optimal Huffman code problem is the variant where you are allowed to reorder the items in the tree.)

For example, consider the following instances of the optimal binary search tree problem with $n = 5$:  

1

DYNAMIC PROGRAMMING III

ERIC BLAIS
• \( p_1 = p_2 = \cdots = p_5 = \frac{1}{5} \).
  In this case the left tree is optimal and has search cost
  \[
  3 \cdot \frac{1}{5} + 2 \cdot \frac{1}{5} + 3 \cdot \frac{1}{5} + 1 \cdot \frac{1}{5} + 2 \cdot \frac{1}{5} = \frac{11}{5}.
  \]

• \( p_1 = 0.6, p_2 = \cdots = p_5 = 0.1 \).
  In this case, the cost of the left tree is
  \[
  3 \cdot 0.6 + 2 \cdot 0.1 + 3 \cdot 0.1 + 1 \cdot 0.1 + 2 \cdot 0.1 = 2.6
  \]
  and the search cost of the right tree is
  \[
  1 \cdot 0.6 + 2 \cdot 0.1 + 3 \cdot 0.1 + 4 \cdot 0.1 + 5 \cdot 0.1 = 2
  \]
  but neither is optimal: there is another binary search tree with cost 1.8. (Exercise: can you find it?)

Dynamic programming can be used to obtain an efficient algorithm that solves the optimal
binary search tree problem. As always with this method, the key step is in identifying the
subproblems and figuring out in which order we solve them.

With this problem, it is natural to consider building trees “from the bottom up”, so that
the most promising subproblems involve solving the optimal binary search tree problem for
items \( i, i + 1, \ldots, j - 1, j \).

**Subproblems:** For each \( 1 \leq i \leq j \leq n \), define \( M[i,j] \) to be minimum search cost of
binary search tree for items \( i, i + 1, \ldots, j \).

**Order of subproblems:** We solve the subproblems by increasing values of \( j - i \) (i.e.,
for the smallest intervals \( i, \ldots, j \) first, then for longer intervals).

Now, let us see how to solve each subproblem, given our solutions to previous subpro-
blems. To find the optimum binary search tree for elements \( i, \ldots, j \), we need to do three things:

(1) Identify the node \( k \in \{i, \ldots, j\} \) that we choose for the root;
(2) Find the search cost of the optimal subtrees for items \( i, \ldots, k - 1 \) and \( k + 1, \ldots, j \);
and
(3) Compute the search cost of the subtree we construct in this way.

The first item has a simple solution: try all possible choices of \( k \) and choose the best one!
The second item is also easily handled—our solutions to the previous problems will give us
exactly the answers we need. So it remains to compute the search tree of our final subtree.
This turns out to be remarkably simple as well: the total search cost \( C \) of tree for items
\( i, \ldots, j \) with left and right subtrees of costs \( C_L \) and \( C_R \) is
\[
C = C_L + C_R + \sum_{\ell=i}^{j} p_{\ell}.
\]
Putting all three items together, we obtain that the solution to each subproblem is given
by the expression
\[
M[i,j] = \min_{k=i}^{j} \{ M[i,k-1] + M[k+1,j] \} + \sum_{\ell=i}^{j} p_{\ell}.
\]
The full algorithm for the problem is as follows.

The time complexity of the algorithm is \( O(n^3) \) since there are \( O(n^2) \) subproblems and
each subproblem is solved in time \( O(n) \). It’s also possible to design an improved algorithm
Algorithm 1: \textsc{OptimumBST}(p_1, \ldots, p_n)

\begin{algorithmic}
\For {$i = 1, \ldots, n$}
\State $M[i, i] \leftarrow p_i$;
\EndFor
\For {$d = 1, \ldots, n - 1$}
\For {$i = 1, \ldots, n - d$}
\State $j \leftarrow i + d$;
\State $b \leftarrow M[i + 1, j]$;
\For {$k = i + 1, \ldots, j$}
\State $b \leftarrow \min\{b, M[i, k - 1] + M[k + 1, j]\}$;
\EndFor
\State $M[i, j] \leftarrow b + \sum_{\ell = i}^{j} p_\ell$;
\EndFor
\EndFor
\EndFor
\end{algorithmic}

with time complexity $O(n^2)$: interested readers can look to Donald Knuth’s original article titled \textit{Optimum Binary Search Trees} (Acta Informatica, 1971) for all the details.

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.

\textbf{Definition 10.2} (Knapsack). An instance of the \textit{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 \textit{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 \textit{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 $k \leq n$,

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

<table>
<thead>
<tr>
<th>Algorithm 2: Knapsack ((w_1, \ldots, w_n, v_1, \ldots, v_n, W))</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ( w = 0, 1, 2, \ldots, W ) do ( M[0, w] \leftarrow 0 );</td>
</tr>
<tr>
<td>for ( k = 0, 1, 2, \ldots, n ) do ( M[k, 0] \leftarrow 0 );</td>
</tr>
<tr>
<td>for ( k = 1, \ldots, n ) do</td>
</tr>
<tr>
<td>for ( w = 1, \ldots, W ) do</td>
</tr>
<tr>
<td>if ( w_k \leq w ) then</td>
</tr>
<tr>
<td>( M[k, w] \leftarrow \max{M[k - 1, w], v_k + M[k - 1, w - w_k]} );</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>( M[k, w] \leftarrow M[k - 1, w] );</td>
</tr>
<tr>
<td>return ( M[n, W] );</td>
</tr>
</tbody>
</table>

**Theorem 10.3.** 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 \( M[k, w] = 0 = \text{OPT}(k, w) \).

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

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

**Theorem 10.4.** 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 \( \Theta(nW + n + W) = \Theta(nW) \).  \( \square \)
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.