Dynamic Programming

Key ideas of dynamic programming: identify subproblems (not too many) and an order of solving them such that each subproblem can be solved by combining previously solved subproblems.
Constructing optimum binary search trees

Given items 1 ... n
probabilities \( p_1 \ldots p_n \)
Construct a binary search tree
to minimize search cost \( \sum_i p_i \text{ depth}(i) \)

\( p = \ldots = p_5 = \frac{1}{5} \)

\[
\text{search cost} = \frac{1}{5} + 2 \cdot \frac{2}{5} + 2 \cdot \frac{3}{5}
\]
\( = \frac{7}{5} \quad \text{# nodes} \rightarrow \text{depth} \)

\( p_1 = .6 \quad p_2 = p_3 = p_4 = p_5 = .1 \)

\[
\begin{align*}
\text{cost:} & = 1(0.6) + 2(0.1) + 3(0.6) + 3(0.1) \\
& = .6 + .2 + 1.8 + .3 = 2.9
\end{align*}
\]

[In case you've seen optimum Huffman trees, this is different in that leaf ordering is fixed.

To apply dynamic programming:
subproblems: opt. binary search tree for items i ... j
order subproblems by # items, i.e., by j-i
to solve i ... j
tree for \( i \ldots k-1 \)

Try all choices for k

tree for \( k+1 \ldots j \)
Details

\[ M[i,j] = \min_{k=i}^{j} \left[ \sum_{k=i}^{j} M[i,k-1] + M[k+1,j] \right] + \sum_{t=i}^{j} P_t \]

- independent of \( k \) choice of \( k \)

How to compute \( \sum_{t=i}^{j} P_t \)

First compute \( P[j] = \sum_{j=1}^{j} P_j \) \( P[0] = 0 \)

then we can get \( \sum_{t=i}^{j} P_t \) as \( P[j] - P[i-1] \)

for \( i = 1 \ldots n \) \( M[i,i] \leftarrow P[i] \) \( M[i,i-1] \leftarrow 0 \)

for \( d = 1 \ldots n-1 \) \( \& d \) is \( j-i \) in above

for \( i = 1 \ldots n-d \)

* solve for \( M[i,i+d] \)

best \( \leftarrow \infty \) \& or a very large number

for \( k = i \ldots i+d \)

\( \text{temp} \leftarrow M[i,k-1] + M[k+1,i+d] \)

if \( \text{temp} < \) best then \( \text{best} \leftarrow \text{temp} \)

end

\( M[i,i+d] \leftarrow \) best + \( P[i+d] - P[i-1] \)

end

end

\# subproblems

Run time \( O(n^2 \cdot n) = O(n^3) \) time per subproblem
Dynamic Programming for 0-1 Knapsack

Recall the knapsack problem: Given items 1, 2, ..., n, where item i has weight $w_i$ and value $v_i$ ($w_i, v_i \in \mathbb{Z}$) choose a subset $S$ of items s.t. \( \sum_{i \in S} w_i \leq W \) and \( \sum_{i \in S} v_i \) is maximized.

Recall that we considered the fractional version (can use fractions of items e.g. flour, rice) where greedy alg. works. Today we consider the 0-1 version where items are indivisible (e.g. flashlight, tent).

First attempts like weighted interval scheduling, distinguish whether item n is in or out:

- if $n \notin S$ - look for optimal solution for 1...n-1
- if $n \in S$ - want subset $S$ of 1...n-1 with
  \[
  \sum_{i \in S} w_i \leq W - w_n
  \]
  the space left in the knapsack.

We must solve a subproblem with different weight capacity.
Subproblems: one for each pair \(i, w\), \(i=0 \ldots n\), \(w=0 \ldots W\)

Find subset \(S \subseteq \{1 \ldots i\}\) s.t.
\[
\sum_{i \in S} w_i \leq w \quad \text{and} \quad \sum_{i \in S} v_i \quad \text{is maximized}
\]

\(\text{Let } M(i, w) = \max_{S} \sum_{i \in S} v_i\)

To find \(M(i, w)\):

* if \(w_i > w\) then \(M(i, w) \leftarrow M(i-1, w)\)
* else \(M(i, w) \leftarrow \max \left\{ M(i-1, w) \right\} \quad \text{if don't use } i
\)
\[
\left\{ v_i + M(i-1, w-w_i) \right\} \quad \text{if use } i
\]

Pseudocode and ordering of subproblems:

Use matrix \(M[0 \ldots n, 0 \ldots W]\)

Initialize \(M[0, w] \leftarrow 0\) \(w=0 \ldots W\)

for \(i=1 \ldots n\)

for \(w=0 \ldots W\)

compute \(M[i, w]\) using

Analysis:

We have a nested loop

So \(O(n \cdot W)\)

This is not a polynomial time algorithm. It is pseudo-polynomial time.

The input is \(w_1 \ldots w_n, v_1 \ldots v_n, W\)

Size of input is sum of # bits.
W is one of the numbers in the input. The size of the input counts the size of W — let's say it has k bits, \( k = \Theta(\log W) \).

But the algorithm takes \( O(n \cdot W) \) — that is, \( O(n \cdot 2^k) \) so it's exponential in the input size.

Run-time is polynomial in the value of W rather than size of W.

Finding the actual solution for knapsack: Two methods:
1. backtracking
2. alter above code to store more info.

1. Backtracking: Use \( M \) to recover solution
   \( i = n, w = W, S = \emptyset \)
   while \( i > 0 \)
     if \( M(i, w) = M(i-1, w) \) /* didn't use \( i \)
       \( i = i-1 \)
     else /* used \( i \)
       \( S = S \cup \{i\} \)
       \( i = i-1, w = w - w_i \)
   end
2. enhance original code

- when we set \( M(i, w) \)
- also set \( \text{Flag}(i, w) \) — do we use item \( i \) or not to get \( M(i, w) \) (we still need backtracking)

- or even store \( \text{Soln}(i, w) \) — list of items to get \( M(i, w) \) (no backtracking needed)

Trade-offs:

(2) uses more space
(1) duplicates tests used to compute \( M \)
Memoization

- use recursion, rather than explicitly solving all subproblems bottom-up as we've been doing so far.
- danger - that you solve the same subproblem over and over (possibly taking exponential time, e.g.
  \[ T(n) = 2T(n-1) + O(1) \] is exponential)
- fix - when you solve a subproblem, store the solution. Before (re)-solving, check if you have a stored solution. Solutions can be stored in a matrix or in a hash table.
- advantage - maybe you don't solve all subproblems.

- disadvantages
  - harder to analyze run-time
  - overhead of recursive approach takes more time.
Common subproblems in dynamic programming

1. Input $x_1, \ldots, x_n$
   - Subproblems $x_i, \ldots, x_i$
   - Weighted interval scheduling

2. Input $x_1, \ldots, x_n$
   - Subproblems $x_i, \ldots, x_j$
   - # subproblems $O(n^2)$
   - Optimal binary search tree

3. Input $x_1, \ldots, x_n, y_1, \ldots, y_m$
   - Subproblems $x_i, \ldots, x_i$ and $y_1, \ldots, y_j$
   - # subproblems $O(n \cdot m)$
   - Edit distance

4. Input rooted tree on $n$ nodes
   - # subproblems $O(n)$
   - Not covered in this course.