Constructing Optimum Binary Search Trees

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

Example: \(p_1 = \cdots = p_5 = \frac{1}{5}\)

\[
\text{search cost} = \frac{1}{5} \cdot 1 + 2 \cdot \frac{1}{5} \cdot 2 + 2 \cdot \frac{1}{5} \cdot 3 = \frac{7}{5}
\]

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

\[
\text{cost} = 1(.1) + 2 \cdot 2(.1) + 3(.6) + 3(.1) = 2.6
\]

To apply dynamic programming:

- subproblems: optimal binary search tree for items \(i \ldots j\)
- order subproblems by \# items (i.e., by \(j - i\)) to solve \(i \ldots j\)
Details: \( M[i, j] = \min_{k=i...j} \{ M[i, k - 1] + M[k + 1, j] \} + \sum_{t=i}^{j} p_t \)

Because every item gets 1 deeper

How to compute \( \sum_{i=1}^{j} p_t \)? First compute \( P[0] = 0 \) and \( P[i] = \sum_{j=1}^{i} p_j \).

Then we can get \( \sum_{i=1}^{j} p_t \) as \( P[j] - P[i - 1] \).

\[
\text{for } i \text{ from 1 to } n \text{ do}
\quad M[i, i] := p_i
\quad M[i, i - 1] := 0
\text{for } d \text{ from 1 to } n - 1 \text{ do} \quad \# d \text{ is } j - i \text{ in above}
\quad \text{for } i \text{ from 1 to } n - d \text{ do} \quad \# \text{ solve for } M[i, i + d]
\quad \text{best} := \infty
\quad \text{for } k \text{ from } i \text{ to } i + d \text{ do}
\quad \quad \text{temp} := M[i, k - 1] + M[k + 1, i + d]
\quad \quad \text{if} \ \text{temp} < \text{best} \ \text{then} \ \text{best} := \text{temp}
\quad \quad M[i, i + d] := \text{best} + P[i + d] - P[i - 1]
\]

Run-time: \( O(n^2 \cdot n) = O(n^3) \)

\( \leftarrow \) # of subproblems
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 such that

- $\sum_{i \in S} w_i \leq W$ and
  \[ \uparrow \text{ capacity of knapsack} \]
- $\sum_{i \in S} v_i$ is maximized.

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

First attempt: Like weighted interval scheduling, distinguish whether item $n$ is in or out.

- if $n \not\in S$ — look for optimal solution for 1...$n - 1$
- if $n \in S$ — want subset $S$ of 1...$n - 1$ with
  \[ \sum_{i \in S} w_i \leq \underbrace{W - w_n}_{\downarrow \text{ the space left in the knapsack}} \]

$\Rightarrow$ we must solve a subproblem with different weight capacity
Subproblems: one for each pair $i, w$, $i = 0 \ldots n$, $w = 0 \ldots W$  
\[ \text{Note: no special order} \]

Find subset $S \subseteq \{1 \ldots i\}$ s.t. $\sum_{i \in S} w_i \leq w$ and $\sum_{i \in S} v_i$ is maximized.

Let $M(i, w) = \max \sum_{i \in S} v_i$. To find $M(i, w)$

- if $w_i > w$ then $M(i, w) := M(i - 1, w)$
- else $M(i, w) := \max \left\{ M(i - 1, w), v_i + M(i - 1, w - w_i) \right\}$  

Pseudocode and ordering of subproblems:

Use table $M[0 \ldots n, 0 \ldots W]$

Initialize $M[0, w] := 0$ for $w = 0 \ldots W$

for $i$ from 1 to $n$ do

for $w$ from 0 to $W$ do

compute $M[i, w]$ using $\star$

Analysis: $n \cdot W \cdot c \leftarrow O(1)$ for $\star$

loop for $i \leftarrow$ loop for $w$

So $O(n \cdot W)$

Note: This is not polynomial time.

It is \underline{pseudo-polynomial} time. The input is $w_1 \ldots w_n, v_1 \ldots v_n, W$. The size of the input is sum of \# bits. $W$ is one of the numbers in the input. The size of the inputs counts the size of $W$ — let’s say it has $k$ bits: $k \in \Theta(\log W)$.

But the algorithm takes $O(n \cdot W)$ — that’s $O(n \cdot 2^k)$ so it’s exponential in the input size. Runtime is polynomial in the \underline{value} of $W$ rather than the \underline{size} of $W$. 
Finding the actual solution for knapsack. Two methods:


\[
S := \emptyset; \quad w := W
\]

\[
\text{for } i \text{ from } n \text{ downto } 0 \text{ do}
\]

\[
\text{if } M(i, w) \neq M(i - 1, w) \text{ then} \quad \# \text{ used } i
\]

\[
S := S \cup \{i\}; \quad w := w - w_i
\]

2. Enhance original code: when we set $M(i, w)$ also set Flag($i, w$)

   — do we use item $i$ or not to get $M(i, w)$ (we still need backtracking)

Or even store 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 solutions. Before (re)-solving, check if you have a stored solution. Solutions can be stored in a matrix or in a hash table. Example: “option remember” in Maple.

```maple
fib := proc(n)
    option remember;
    if n = 0 then return 0
    elif n = 1 then return 1
    else return fib(n - 1) + fib(n - 2)
    end if
end proc
```

- advantage — maybe you don’t solve all subproblems.

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

1. input $x_1 \ldots x_n$
   subproblems $x_1 \ldots x_i$
   \# subproblems $n$  
   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 $x_1 \ldots x_i$ and $y_1 \ldots y_j$
   \# subproblems $O(nm)$  
   edit distance