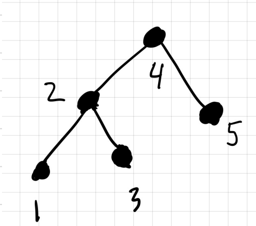
Constructing Optimum Binary Search trees

Given items $1..n$ and probabilities $p_1..p_n$, construct a binary search tree to minimize the search cost $\sum_i p_i \underbrace{\text{ProbeDepth}(i)}_{\text{\# probes into tree to find item } i}$.

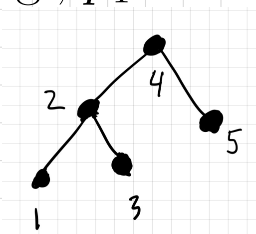
e.g., $p_1 = \cdots = p_5 = \frac{1}{5}$



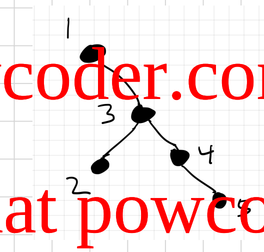
search cost $= 1 \cdot \frac{1}{5} + 2 \cdot 2 \cdot \frac{1}{5} + 2 \cdot 3 \cdot \frac{1}{5} = \frac{7}{5}$

\# nodes depth

e.g., $p_1 = .6 \quad p_2 = p_3 + p_4 = p_5 = .1$



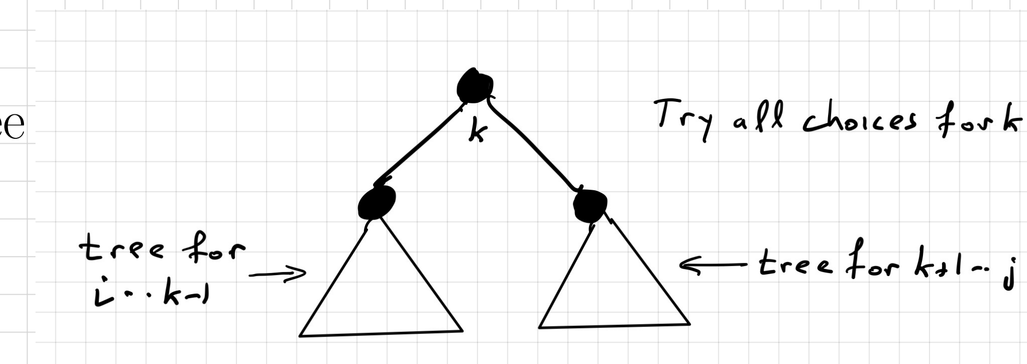
cost $= 1(.1) + 2 \cdot 2(.1) + 3(.6) + 3(.1) = 2.6$



cost $= 1(.6) + 2 \cdot 2(.1) + 2 \cdot 3(.1) + 4(.1) = 1.8$

To apply dynamic programming:

- subproblems: optimal binary search tree for items $i..j$
- order subproblems by # items (i.e., by $j - i$) to solve $i..j$



Details

independent of choice of k

$$M[i,j] = \min_{k=1..j}\{M[i,k-1]+M[k+1,j]\} + \underbrace{\sum_{t=i}^{j} p_t}_{\text{because every node gets 1 deeper}}$$

How to compute $\sum_{t=i}^{j} p_t$

First compute $P[i] = \sum_{j=1}^{i} p_j \qquad P[0] = 0$

then we can get $\sum_{t=i}^{j} p_t$ as $P[j] - P[i-1]$.

---

for i **from** 1 **to** n **do**
  $M[i,i] := p_i$
  $M[i,i-1] := 0$
od
for d **from** 1 **to** $n-1$ **do**   # d is $j-i$ in above
  for i **from** 1 **to** $n-d$
    # solve for $M[i,i+d]$
    best $:= \infty$ # or a very large number
    for k **from** i **to** $i+d$ **do**
      temp $:= M[i,k-1] + M[k+1,i+d]$
      if temp $<$ best **then** best $:=$ temp **fi**;
    od
    $M[i,i+d] :=$ best $+ P[i+d] - P[i-1]$
  od
od

Runtime $O(\overbrace{n^2}^{\#\text{ subproblems}} \cdot \underbrace{n}_{\text{time per subproblem}}) = O(n^3)$

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 $\sum_{i \in S} v_i$ is maximized.
> capacity of knapsack

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 $n \notin S$ — look for optimal solution for $1..n-1$

- if $n \in S$ — want subset S of $1..n-1$ with

$$\sum_{i \in S} w_i \leq \underbrace{W - w_n}_{\text{the space left in the knapsack}}$$

$\Rightarrow$ we must solve a subproblem with different weight capacity

Subproblems: one for each pair i, w, $i = 0..n$, $w = 0..W$

note: no special order of items

Find subset $S \subseteq \{1..i\}$ s.t.

$$\sum_{i \in S} w_i \leq w \quad \text{and} \quad \sum_{i \in S} v_i \text{ is maximized}$$

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 $M(i, w) := \max \begin{cases} M(i-1, w) & \text{\# don't use } i \\ v_i + M(i-1, w - w_i) & \text{\# use i} \end{cases}$

Pseudocode and ordering of subproblems:

Use matrix $M[0..n, 0..W]$
Initialize $M[0, w] := 0$ for $w = 0..W$
for i **from** 1 **to** n **do**
  for w **from** 0 **to** W **do**
    compute $M[i, w]$ using *
  od
od

Analysis: $n \cdot W \cdot c$

subproblems

work per subproblem

constant work for *

loop for w

loop for i

So $O(n \cdot W)$

This is not a polynomial time algorithm. It is pseudo-polynomial time.
The input is $w_1..w_n$, $v_1..v_n$, 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 <u>value</u> of W rather than the <u>size</u> of W.

Finding the actual solution for knapsack. Two methods:

1. Backtracking: Use M to recover solution

   $i := n$; $w := W$; $S := \emptyset$
   while $i > 0$ **do**
       if $M(i, w) = M(i-1, w)$ # didn't use i
           $i := i - 1$
       else # used i
           $S := S \cup \{i\}$; $i := i-1$; $w := w - w_i$
       fi
   od

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


```
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 sub problems in dynamic programming

1. input $x_1..x_n$

   subproblems $x_1..x_i$

   # subproblems n

   weighted interval scheduling

2. input $x_1..x_n$

   subproblems $x_i..x_j$

   # subproblems $O(n^2)$ optimal binary search tree

3. input $x_1..x_n$ $y_1..y_m$

   subproblems $x_1..x_i$ and $y_1..y_j$

   # subproblems $O(nm)$ edit distance