Dynamic Programming

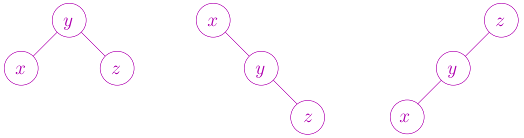
Algorithms: Design and Analysis, Part II

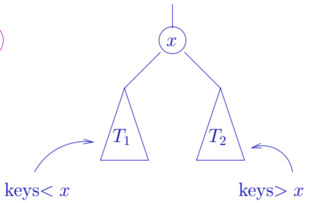
Optimal Binary Search Trees: Problem Definition

A Multiplicity of Search Trees

Recall: For a given set of keys, there are lots of valid search trees.

[say $x < y < z$]





the search tree property

Question: What is the "best" search tree for a given set of keys?

A good answer: A balanced search tree, like a red-black tree. (Recall Part I)

$\Rightarrow$ Worst-case search time $= \Theta(\text{height}) = \Theta(\log n)$

Exploiting Non-Uniformity

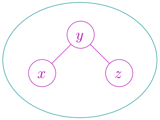
Question: Suppose we have keys $x < y < z$ and we know that:

80% of searches are for x

10% of searches are for y

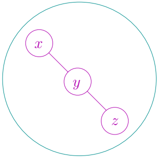
10% of searches are for z

What is the average search time (i.e., number of nodes looked at) in the trees:



$0.8 \cdot 2 + 0.1 \cdot 1 + 0.1 \cdot 2 = 1.9$

and



$0.8 \cdot 1 + 0.1 \cdot 2 + 0.1 \cdot 3 = 1.3$

respectively?

A) 2 and 3 B) 2 and 1

C) 1.9 and 1.2 D) 1.9 and 1.3

Tim Roughgarden

Problem Definition

Input: Frequencies $p_1, p_2, \ldots, p_n$ for items $1, 2, \ldots, n$.
[Assume items in sorted order, $1 < 2 < \ldots < n$]

Goal: Compute a valid search tree that minimizes the weighted (average) search time.

$$C(T) = \sum_{\text{items } i} p_i \underbrace{\text{[search time for } i \text{ in } T]}_{\text{Depth of } i \text{ in } T + 1}$$

Example: If T is a red-black tree, then $C(T) = O(\log n)$.
(Assuming $\sum_i p_i = 1$.)

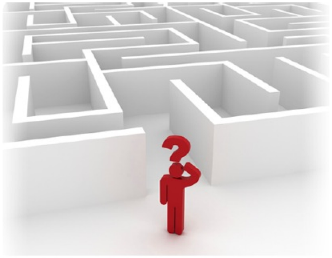
Comparison with Huffman Codes

Similarities:

- Output = a binary tree
- Goal is (essentially) to minimize average depth with respect to given probabilities

Differences:

- With Huffman codes, constraint was prefix-freeness [i.e., symbols only at leaves]
- Here, constraint = search tree property [seems harder to deal with]



Dynamic Programming

Algorithms: Design and Analysis, Part II

Optimal BSTs: Optimal Substructure

Problem Definition

Input: Frequencies $p_1, p_2, \ldots, p_n$ for items $1, 2, \ldots, n$.
[Assume items in sorted order, $1 < 2 < \ldots < n$]

Goal: Compute a valid search tree that minimizes the weighted (average) search time.

$$C(T) = \sum_{\text{items } i} p_i \underbrace{[\text{search time for } i \text{ in } T]}_{\text{Depth of } i \text{ in } T + 1}$$

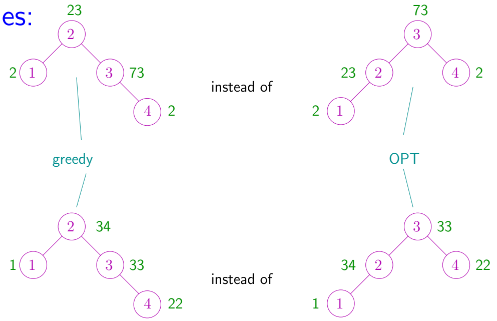
Greedy Doesn't Work

Intuition: Want the most (respectively, least) frequently accessed items closest (respectively, furthest) from the root.

Ideas for greedy algorithms:

- Bottom-up [populate lowest level with least frequently accessed keys]
- Top-down [put most frequently accessed item at root, recurse]

Counter examples:



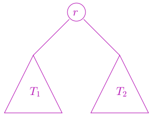
Choosing the Root

Issue: With the top-down approach, the choice of root has hard-to-predict repercussions further down the tree.
[stymies both greedy and naive divide + conquer approaches]

Idea: What if we knew the root?
(i.e., maybe can try all possibilities within a dynamic programming algorithm!)

Optimal Substructure

Question: Suppose an optimal BST for keys $\{1, 2, \ldots, n\}$ has root r, left subtree T_1, right subtree T_2. Pick the strongest statement that you suspect is true.



A) Neither T_1 nor T_2 need be optimal for the items it contains.

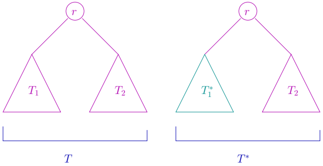
B) At least one of T_1, T_2 is optimal for the items it contains.

C) Each of T_1, T_2 is optimal for the items it contains.

D) <u>T_1 is optimal for the keys $\{1, 2, \ldots, r-1\}$ and T_2 for the keys $\{r+1, r+2, \ldots, n\}$</u>

Proof of Optimal Substructure

Let T be an optimal BST for keys $\{1, 2, \ldots, n\}$ with frequencies $p_1, \ldots, p_n$. Suppose T has root r. Suppose for contradiction that T_1 is not optimal for $\{1, 2, \ldots, r-1\}$ [other case is similar] with $C(T_1^*) < C(T_1)$. Obtain T^* from T by "cutting+pasting" T_1^* in for T_1.



Note: To complete contradiction + proof, only need to show that $C(T^*) < C(T)$.

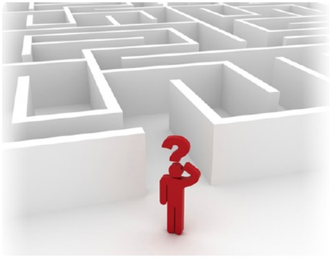
Proof of Optimal Substructure (con'd)

A Calculation:

$$
\begin{aligned}
C(T) &= \sum_{i=1}^{n} p_i \,[\text{search time for } i \text{ in } T] \\
&= p_r \cdot 1 + \sum_{i=1}^{r-1} p_i \,\overbrace{\boxed{[\text{search time for } i \text{ in } T]}}^{=1+\text{search time for } i \text{ in } T_1} \\
&\quad + \sum_{i=r+1}^{n} p_i \,\overbrace{\boxed{[\text{search time for } i \text{ in } T]}}^{=1+\text{search time for } i \text{ in } T_2} \\
&= \underbrace{\boxed{\sum_{i=1}^{n} p_i}}_{\text{a constant (independent of } T\text{)}} + \underbrace{\boxed{\sum_{i=1}^{r-1} p_i \,[\text{search time for } i \text{ in } T_1]}}_{= C(T_1)} \\
&\quad + \underbrace{\boxed{\sum_{i=r+1}^{n} p_i \,[\text{search time for } i \text{ in } T_2]}}_{= C(T_2)}
\end{aligned}
$$

Similarly: $C(T^*) = \sum_{i=1}^{n} p_i + C(T_1^*) + C(T_2)$

Upshot: $C(T_1^*) < C(T_1)$ implies $C(T^*) < C(T)$, contradicting optimality of T. QED!



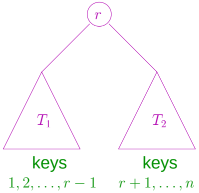
Dynamic Programming

Algorithms: Design and Analysis, Part II

Optimal BSTs: A Dynamic Programming Algorithm

Optimal Substructure

Optimal Substructure Lemma: If T is an optimal BST for the keys $\{1, 2, \ldots, n\}$ with root r, then its subtrees T_1 and T_2 are optimal BSTs for the keys $\{1, 2, \ldots, r-1\}$ and $\{r+1, \ldots, n\}$, respectively.



Note: Items in a subproblem are either a prefix <u>or</u> a suffix of the original problem.

Relevant Subproblems

Question: Let $\{1, 2, \ldots, n\}$ = original items. For which subsets $S \subseteq \{1, 2, \ldots, n\}$ might we need to compute the optimal BST for S?

A) Prefixes ($S = \{1, 2, \ldots, i\}$ for every i)

B) Prefixes and suffixes ($S = \{1, \ldots, i\}$ and $\{i, \ldots, n\}$ for every i)

C) <u>Contiguous intervals ($S = \{i, i+1, \ldots, j-1, j\}$ for every $i \leq j$)</u>

D) All subsets S

The Recurrence

Notation: For $1 \leq i \leq j \leq n$, let C_{ij} = weighted search cost of an optimal BST for the items $\{i, i+1, \ldots, j-1, j\}$ [with probabilities $p_i, p_{i+1}, \ldots, p_j$]

Recurrence: For every $1 \leq i \leq j \leq n$:

$$C_{ij} = \min_{r=i,\ldots,j} \left\{ \sum_{k=i}^{j} p_k + C_{i,r-1} + C_{r+1,j} \right\}$$

(Recall formula $C(T) = \sum_k p_k + C(T_1) + C(T_2)$ from last video)

Interpret $C_{xy} = 0$ if $x > y$

Correctness: Optimal substructure narrows candidates down to $(j - i + 1)$ possibilities, recurrence picks the best by brute force.

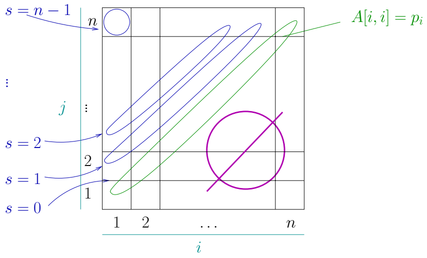
The Algorithm

Important: Solve smallest subproblems (with fewest number $(j-i+1)$ of items) first.

```
Let $A$ = 2-D array. [$A[i,j]$ represents opt BST value of items $\{1,\dots,j\}$]
For $s = 0$ to $n-1$ [$s$ represents $j-i$]
   For $i = 1$ to $n$ [so $i+s$ plays role of $j$]
      $A[i,i+s] = \min_{r=1,\dots,i+s}\{\sum_{k=1}^{i+s} p_k + A[i,r-1] + A[r+1,i+s]\}$
Return $A[1,n]$
```

Interpret as 0 if 1st index > 2nd index. Available for $O(1)$-time lookup

Pictorially:



Running Time

- $\Theta(n^2)$ subproblems
- $\Theta(j - i)$ time to compute $A[i, j]$

$\Rightarrow \Theta(n^3)$ time overall

Fun fact: [Knuth '71, Yoo '80] Optimized version of this DP algorithm correctly fills up entire table in only $\Theta(n^2)$ time [$\Theta(1)$ on average per subproblem]

[Idea: piggyback on work done in previous subproblems to avoid trying all possible roots]