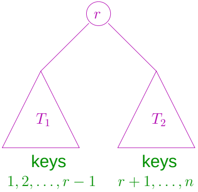
Dynamic Programming

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 $\{1,\ldots,j\}$]

For $s=0$ to $n-1$ [s represents $j-i$]

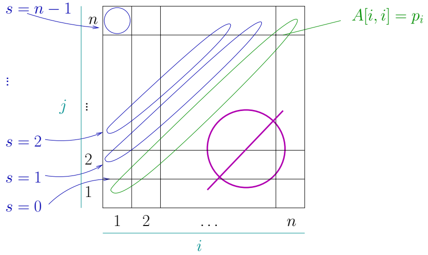
  For $i=1$ to n [so $i+s$ plays role of j]

    $A[i,i+s] = \min_{r=1,\ldots,i+s}\{\sum_{k=1}^{i+s} p_k + \boxed{A[i,r-1]} + \boxed{A[r+1,i+s]}\}$

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]