CSC 411
Design and Analysis of Algorithms

Chapter 8 - Part 2

Dynamic Programming

Instructor: Minhee Jun

Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- "Programming" here means "planning"
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances ***once***
  - record solutions in a table
  - extract solution to the initial instance from that table

Examples of DP algorithms

- Fibonaccl numbers problem
- Computing a binary coefficient

- Three basic examples
  - Coin row problem
  - Change-Making problem
  - Coin-collecting problem

- Knapsack problem & memory functions

- Optimal binary search tree

- Warshall’s and Floyd’s Algorithms

The Knapsack Problem

- Given n items of known weights $w_1,\ w_2, \cdots, w_n$ and values $v_1,\ v_2, \cdots, v_n$, and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack
  - Let's consider an instance defined by the first i items, and knapsack capacity j.
    - $F(i,\ j)$: the value of an optimal solution to this instance.
      i.e. the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

The Knapsack Problem

- Among the subsets that do not include the ith item, the value of an optimal subset is $F(i-1,\ j)$
- Among the subsets that do include the ith item, the value of an optimal subset is $v_i + F(i-1, j-w_i)$

$$F(i,j) = \begin{cases} \max\{F(i-1,j),\ v_i + F(i-1,j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1,j) & \text{if } j - w_i < 0. \end{cases}$$

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1,\ j-w_i)$	$F(i-1,\ j)$	
$w_i,\ v_i$	i	0		$F(i,\ j)$	
	n	0			goal

The Knapsack Problem

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

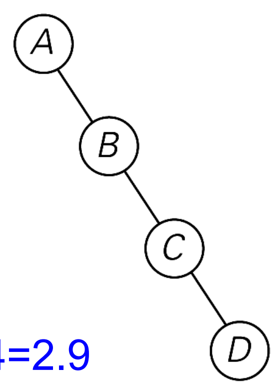
$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

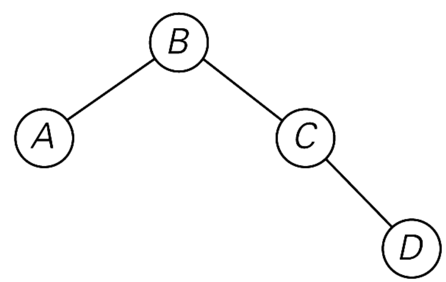
	i	capacity j: 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

6. Optimal Binary Search Trees

- A binary search tree is on e of the most important data structure in computer science.
  - An optimal binary search tree: the average # comparisons in a search is the smallest possible
  - For a given *n* keys $a_1 < a_2 < \cdots < a_n$ and probabilities $p_1,\ p_2,\ \cdots,\ p_n$, searching for them, find a BST with a minimum average # comparisons in successful search
    - For example, four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively



0.1*1+0.2*2+0.4*3+0.3*4=2.9



0.1*2+0.2*1+0.4*2+0.3*3=2.1

Brute Force Approach to Find Optimal Binary Search Trees

- What is the total number of BSTs with *n* nodes ($a_1 < a_2 < \cdots < a_n$)?
  - Example: for 4 nodes with keys *A<B<C<D,*
    we could find the optimal tree by generating all 14 BST.
    .
  - With exhaustive-search approach,
    the total # BSTs with n keys is the nth Catalan number,

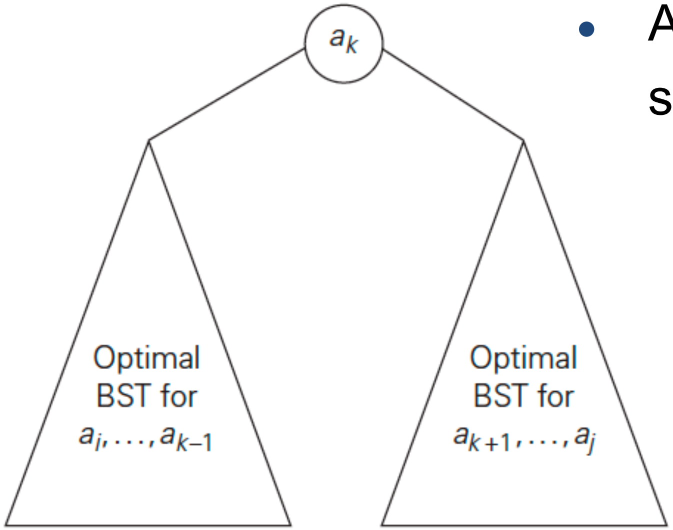
Catalan number : $\mathrm{C(n)} = \binom{2n}{n}\frac{1}{n+1} = \frac{2n!}{(n+1)!n!}$

Is brute force approach is good solution?

This grows exponentially, so brute force is hopeless.

Optimal Binary Search Trees - DP

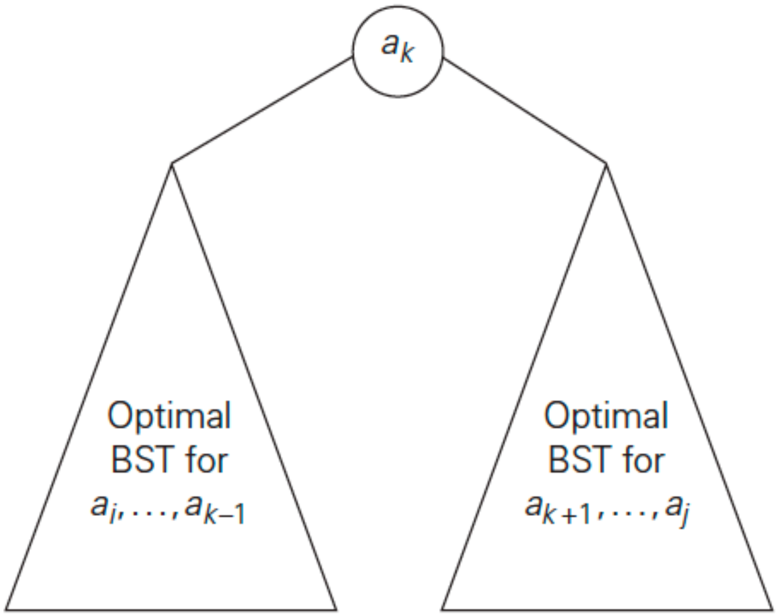
- $C(i, j)$: the smallest average # comparisons made in a successful search in a BST T_i^j, made up of keys a_i, a_{i+1}, $\cdots$, a_j $(1 \le i \le j \le n)$.
- We are just interested in $C(1,n)$
  - We will find values of $C(i, j)$ for all smaller instances of the problem
  - Consider all possible ways to choose a root a_k among the keys a_i, $\cdots$, a_j of a BST T_i^j



- A root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j

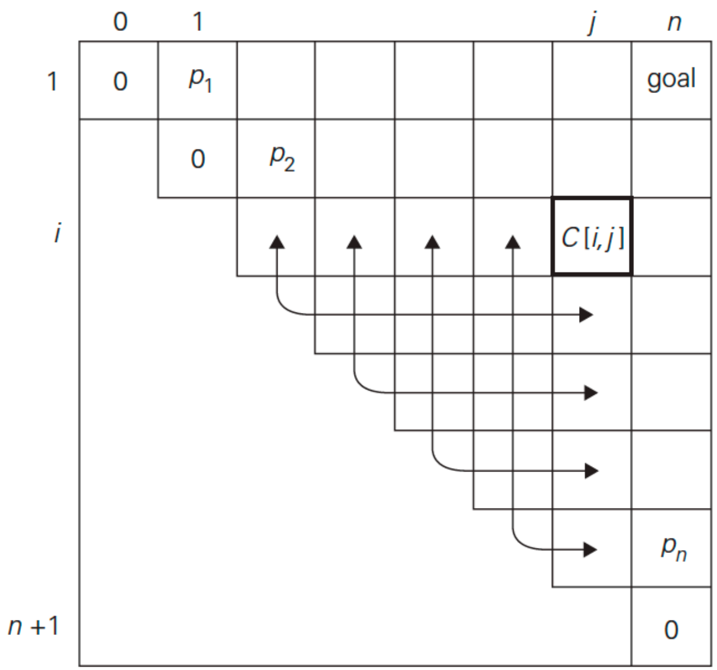
Optimal Binary Search Trees - DP

- $$\begin{aligned} C(i,j) &= \min_{i\le k\le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\} \\ &= \min_{i\le k\le j} \{ \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1}) + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j}) + \sum_{s=i}^{j} p_s\} \\ &= \min_{i\le k\le j} \{C(i,k-1) + C(k+1,j))\} + \sum_{s=i}^{j} p_s \end{aligned}$$
- $C(i,i) = p_i$ for $1 \le i \le n$.



Optimal Binary Search Trees - DP

- $C(i,j) = \min_{i \le k \le j} \{C(i,k-1) + C(k+1,j))\} + \sum_{s=i}^{j} p_s$
- $C(i,i) = p_i$ for $1 \le i \le n$.



Optimal Binary Search Trees: Example

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

$$C(i,j) = \min_{i\le k\le j}\{C(i,k-1) + C(k+1,j))\} + \sum_{s=i}^{j} p_s \qquad C(i,i) = p_i \text{ for } 1 \le i \le n.$$

$$C(1,2) = \min\begin{Bmatrix} k=1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{Bmatrix}$$

$$= 0.4.$$

Optimal Binary Search Trees: Example

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

$$C(i,j) = \min_{i \le k \le j} \{C(i,k-1) + C(k+1,j))\} + \sum_{s=i}^{j} p_s$$

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Optimal Binary Search Trees: Example

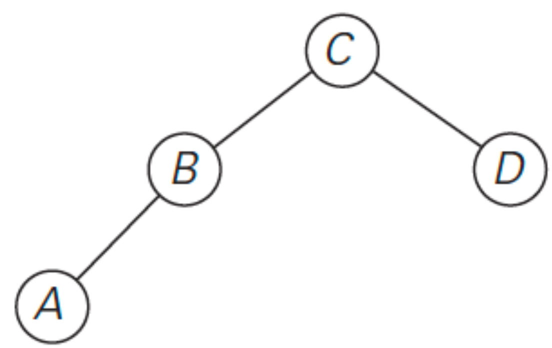
main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

- The root of the optimal tree: the third key, i.e., C
  - $R(1, 4) = 3$
- The root of the optimal tree containing A and B: B
  - $R(1,2) = 2$



Optimal Binary Search Trees: Pseudocode

```
ALGORITHM  OptimalBST(P[1..n])
    //Finds an optimal binary search tree by dynamic programming
    //Input: An array P[1..n] of search probabilities for a sorted list of n keys
    //Output: Average number of comparisons in successful searches in the
    //        optimal BST and table R of subtrees' roots in the optimal BST
    for i ← 1 to n do
        C[i, i − 1] ← 0
        C[i, i] ← P[i]
        R[i, i] ← i
    C[n + 1, n] ← 0
    for d ← 1 to n − 1 do    //diagonal count
        for i ← 1 to n − d do
            j ← i + d
            minval ← ∞
            for k ← i to j do
                if C[i, k − 1] + C[k + 1, j] < minval
                    minval ← C[i, k − 1] + C[k + 1, j];   kmin ← k
            R[i, j] ← kmin
            sum ← P[i];   for s ← i + 1 to j do sum ← sum + P[s]
            C[i, j] ← minval + sum
    return C[1, n], R
```

Time efficiency: $\Theta(n^2)$

CSC 411 Design and Analysis of Algorithm

- Chapter 1: Introduction
- Chapter 2: Fundamentals of Analysis of Algorithm Efficiency
- Chapter 3: Brute Force and Exhaustive Search
- Chapter 4: Decrease-and-Conquer
- Chapter 5: Divide-and-Conquer
- Chapter 6: Transform-and-Conquer
- Chapter 9: Greedy Technique
- Chapter 8: Dynamic Programming

Thank you!