COP 3530: Computer Science III
Summer 2005

Dynamic Programming – Part 2

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
CSB 242, (407)823-2790

Course Webpage:
http://www.cs.ucf.edu/courses/cop3530/sum2005

School of Computer Science
University of Central Florida

Pseudo-Polynomial Time Algorithms

- The running time of the algorithm for solving the 0-1 Knapsack problem using dynamic programming was O(nW).
- This means that the running time of the algorithm depends on a parameter *W* that, strictly speaking, is not proportional to the size of the input (the *n* items, together with their weights and benefits, plus the number *W*).
- Assuming that *W* is encoded in some standard way (such as a binary number), then it takes only O(log W) bits to encode *W*.
- If *W* is very large (say $W = 2^n$), then this dynamic programming algorithm would actually be asymptotically slower than the brute force method!
- Thus, technically speaking, this algorithm is not a polynomial time algorithm because its running time is not actually a function of the size of the input.

Pseudo-Polynomial Time Algorithms (cont.)

- It is common to refer to an algorithm such as the 0-1 Knapsack algorithm as being a pseudo-polynomial time algorithm, because its running time depends on the magnitude of a number given in the input, not its encoding size.
- In practice, such algorithms should run much faster than any brute-force algorithm, but it is not correct to say they are true polynomial-time algorithms.
- In fact, the theory of NP-completeness states that it is very unlikely that anyone will every find a true polynomial-time algorithm for the 0-1 Knapsack problem.

Optimal Binary Search Trees

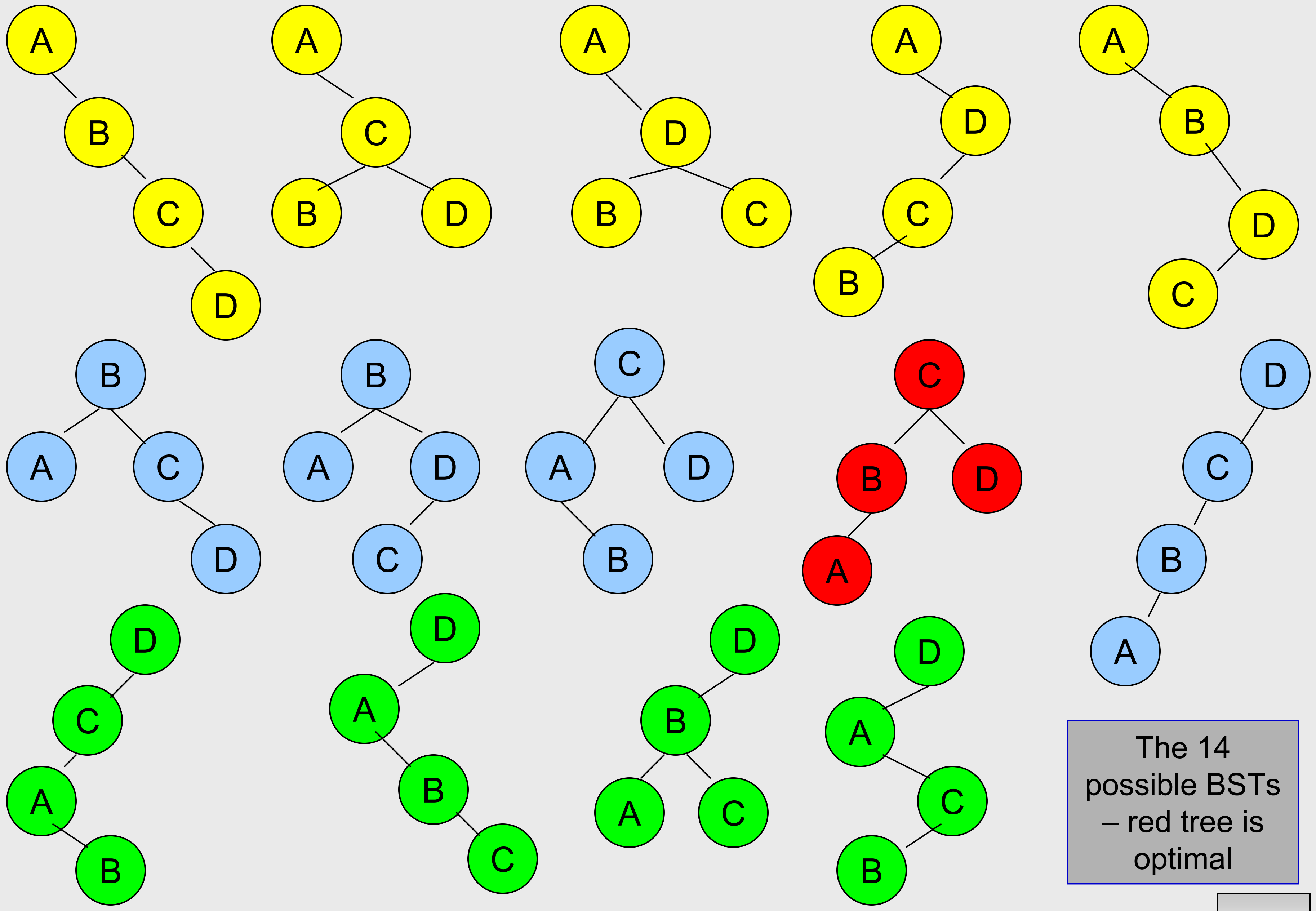
- One of the principle applications of binary search trees (BSTs) is to implement a dictionary.
- A dictionary is a set of elements with the operations of searching, insertion, and deletion.
- If probabilities of searching for elements of a set are known (e.g., from accumulated data about past searches), it is natural to pose a question about an optimal BST for which the average number of comparisons in a search is the smallest possible. (For simplicity, we'll limit the discussion to minimizing the average number of comparisons in a successful search. The method can be extended to include unsuccessful searches as well.)

Optimal Binary Search Trees (cont.)

- As an example, let's consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively.
- The brute force algorithm for this problem would be to generate all of the BSTs which contain these four keys and determine which tree(s) provides the lowest average number of comparisons in a successful search.
- How many BSTs are there for this problem? Can you generate them? How many for a general BST containing *n* keys?

$$c(n) = \binom{2n}{n}\frac{1}{n+1} \quad \text{for } n \geq 0,\ c(0) = 1$$

Catalan number: 1, 2, 5, 14, 42,132, 429,1430, 4862, …. Named after discoverer Eugeni Catalan in mid 1800s. Approaches ∞ as fast as $4^n/n^{1.5}$.



The 14 possible BSTs – red tree is optimal

Finding An Optimal Binary Search Tree

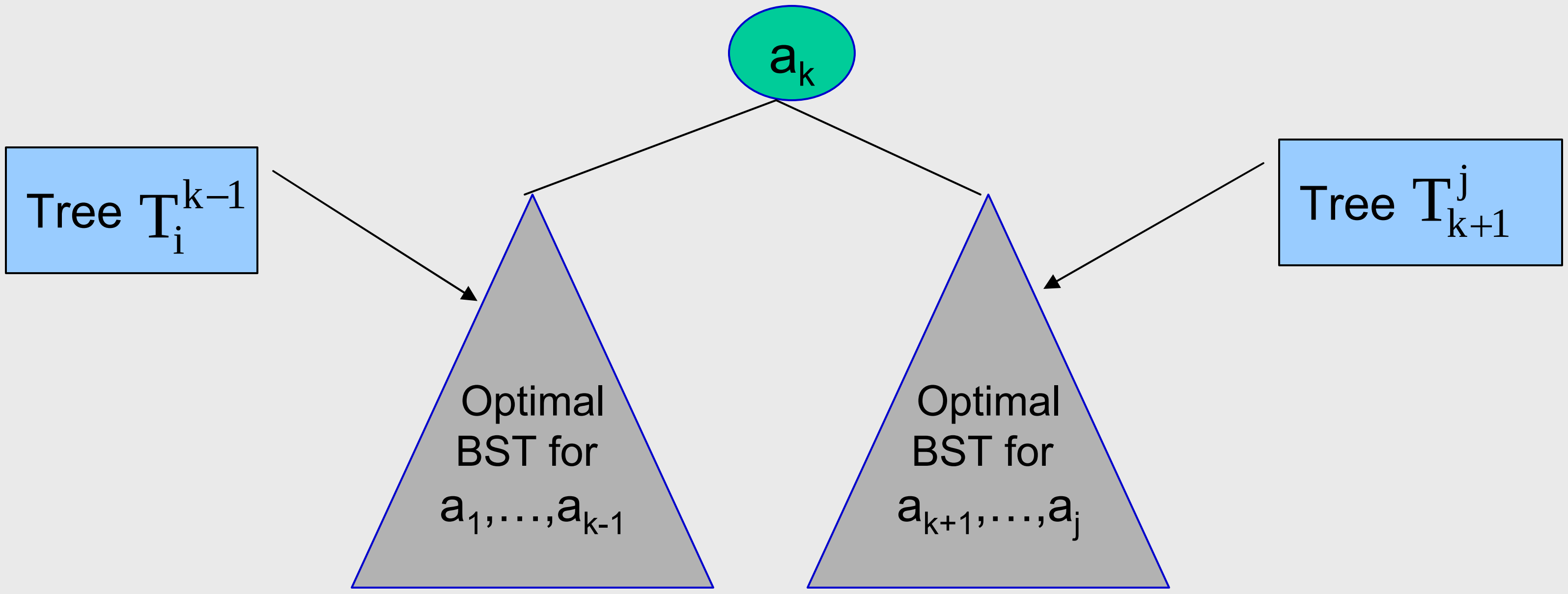
- Let $a_1,\ldots,a_n$ be distinct keys ordered from the smallest to the largest and let $p_1,\ldots,p_n$ be the probabilities of searching for them.
- Let `C[i,j]` be the smallest average number of comparisons made in a successful search in a binary tree T_i^j made up of the keys listed above.
- Following the classic dynamic programming approach, we need to find values of `C[i,j]` for all smaller instances of the problem, although we are interested only in `C[1,n]`.
- We now need to derive the recurrence which underlies the dynamic programming algorithm (i.e., the way to produce all of the `C[i,j]` terms).
- We need to consider all possible ways to choose a root a_k among the keys $a_1,\ldots,a_n$.

Finding An Optimal Binary Search Tree (cont.)

- For such a binary tree (see next page), the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i,\ldots,a_k$ optimally arranges, and the right subtree T_{k+1}^{j} contains the keys $a_{k+1},\ldots,a_j$ also optimally arranged.
- Notice how we are using the principle of optimality in this case.
- If we count the levels in the tree starting with 1 (in order to make the comparison numbers equal the level of the key)., the following recurrence relation is obtained:

$$C[i,j] = \min\left\{ p_k \times 1 + \sum_{s=i}^{k-1} p_s \times (\text{level of } a_s \text{ in } T_i^{k-1} + 1) \right\} + \sum_{s=k+1}^{j} p_s \times (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)$$

Finding An Optimal Binary Search Tree (cont.)


a_k
Tree T_i^{k-1}
Tree T_{k+1}^{j}
Optimal BST for $a_1,\ldots,a_{k-1}$
Optimal BST for $a_{k+1},\ldots,a_j$

- Reducing the recurrence on the previous page gives us:

$$C[i, j] = \min\{C[i, k-1] + C[k+1, j]\} + \sum_{s=k+1}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

Finding An Optimal Binary Search Tree (cont.)

- The recurrence on the previous page indicates that a matrix will be required to hold the dynamic programming solution to the optimal binary search tree problem.
- The recurrence indicates that the solution for C[i,j] requires values that will be in row *i* and the columns to the left of column *j* and in column *j* and the rows below row *i*.
- In the diagram on the next page, the arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of C[i,j].
- This suggests that the matrix should be filled along its diagonals, starting with all zeros on the main diagonal and given probabilities p_i, $1 \leq i \leq n$, right above it and moving toward the upper right corner of the matrix.

EXAMPLE - Finding An Optimal Binary Search Tree

- Let's compute C[1,2]:

$$C[1,2] = \min \begin{cases} k = 1: C[1,0] + C[2,2] + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: C[1,1] + C[3,2] + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{cases}$$

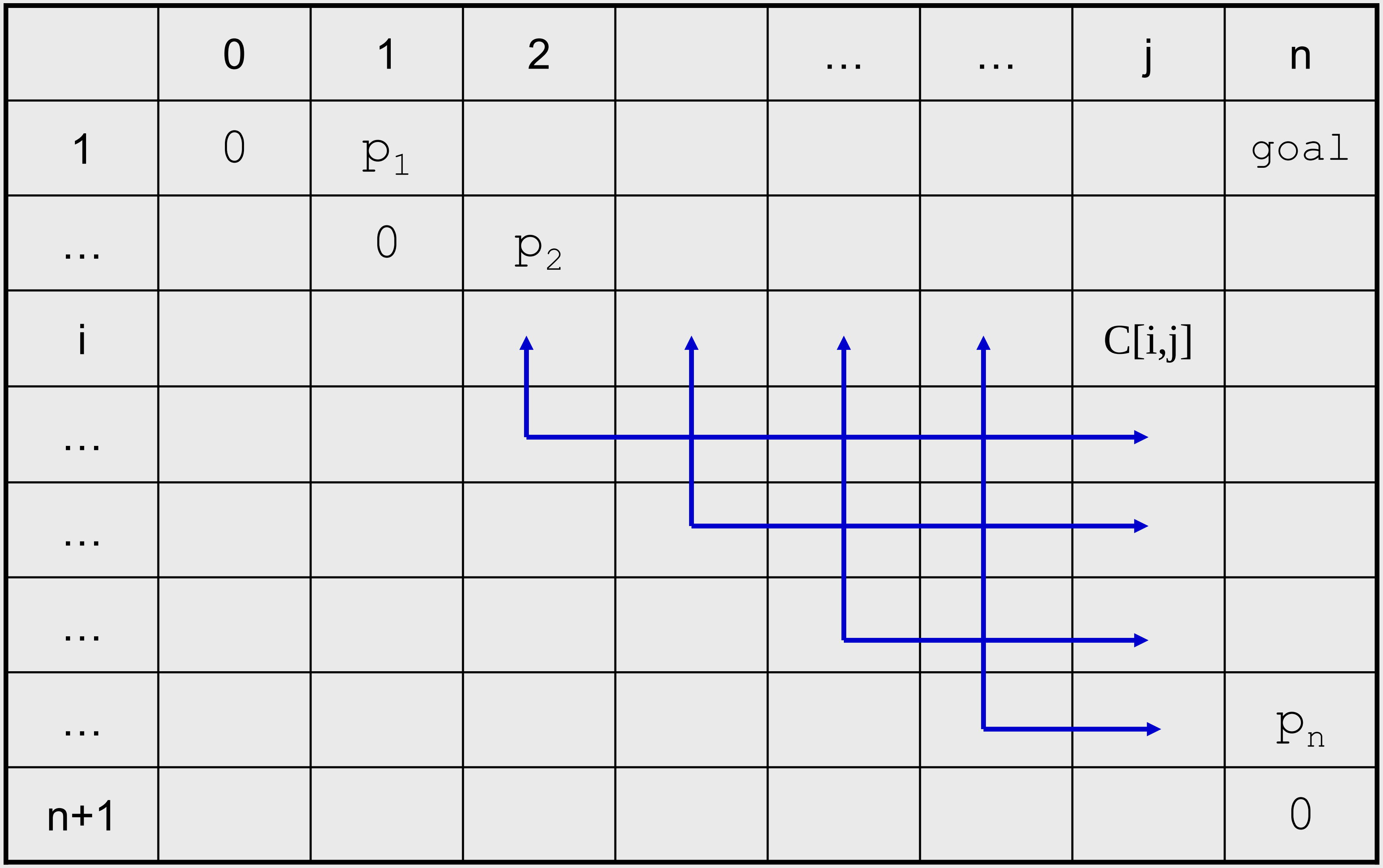
	0	1	2	3	4
1	0	0.1	**0.4**		
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

Main Table

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

Root Table

Dynamic Programming Algorithm Matrix For Optimal BST



	0	1	2		…	…	j	n
1	0	p_1						goal
…		0	p_2					
i							C[i,j]	
…								
…								
…								
…								p_n
n+1								0

Finding An Optimal Binary Search Tree (cont.)

- The technique described computes C[1,n], which is the average number of comparisons for a successful searching the optimal binary search tree.
- If we also would like to produce the optimal tree itself, we need to maintain another matrix, let's call it *R*, to record the value of *k* for which the minimum value in the recurrence is achieved.
- This auxiliary table has the same shape as the previous table and is filled in the same manner, starting with entries *R*[i,i] = *I* for $1 \leq i \leq n$. When the table is filled, its entries indicate indices of the roots of the optimal subtrees, which makes it possible to reconstruct an optimal tree for the entire set which is given.

EXAMPLE - Finding An Optimal Binary Search Tree

- Let's assume the four-key set that we used for the earlier example on page 5.

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

- The initial tables look like the following:

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

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

- Computing C[1,2] we have:

$$C[1,2] = \min \begin{cases} k = 1 : C[1,0] + C[2,2] + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2 : C[1,1] + C[3,2] + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{cases}$$

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

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

- Continuing we have, C[2,3]:

$$C[2,3] = \min \begin{cases} k = 2 : C[2,1] + C[3,3] + \sum_{s=2}^{3} p_s = 0 + 0.4 + 0.6 = 1.0 \\ k = 3 : C[2,2] + C[4,3] + \sum_{s=1}^{3} p_s = 0.2 + 0 + 0.6 = 0.8 \end{cases}$$

	0	1	2	3	4
1	0	0.1	0.4		
2		0	0.2	**0.8**	
3			0	0.4	
4				0	0.3
5					0

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

- Continuing we have, C[3,4]:

$$C[3,4] = \min \begin{cases} k = 3 : C[3,2] + C[4,4] + \sum_{s=3}^{4} p_s = 0 + 0.3 + 0.7 = 1.0 \\ k = 4 : C[3,3] + C[5,4] + \sum_{s=3}^{4} p_s = 0.4 + 0 + 0.7 = 1.1 \end{cases}$$

	0	1	2	3	4
1	0	0.1	0.4		
2		0	0.2	0.8	
3			0	0.4	**1.0**
4				0	0.3
5					0

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

$$C[2,4] = \min \begin{cases} k = 2 : C[2,1] + C[3,4] + \sum_{s=2}^{4} p_s = 0 + 1.0 + 0.9 = 1.9 \\ k = 3 : C[2,2] + C[4,4] + \sum_{s=2}^{4} p_s = 0.2 + 0.3 + 0.9 = 1.4 \\ k = 4 : C[2,3] + C[5,4] + \sum_{s=2}^{4} p_s = 0.8 + 0 + 0.9 = 1.7 \end{cases}$$

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

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

$$C[1,3] = \min \begin{cases} k = 1 : C[1,0] + C[2,3] + \sum_{s=1}^{3} p_s = 0 + 0.8 + 0.7 = 1.5 \\ k = 2 : C[1,1] + C[3,3] + \sum_{s=1}^{3} p_s = 0.1 + 0.4 + 0.7 = 1.2 \\ k = 3 : C[1,2] + C[4,3] + \sum_{s=1}^{3} p_s = 0.4 + 0 + 0.7 = 1.1 \end{cases}$$

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

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

$$C[1,4] = \min \begin{cases} k = 1: C[1,0] + C[2,4] + \sum_{s=1}^{4} p_s = 0 + 1.4 + 1.0 = 2.4 \\ k = 2: C[1,1] + C[3,4] + \sum_{s=1}^{4} p_s = 0.1 + 1.0 + 1.0 = 2.1 \\ k = 3: C[1,2] + C[4,4] + \sum_{s=1}^{4} p_s = 0.4 + 0.3 + 1.0 = 1.7 \\ k = 4: C[1,3] + C[5,4] + \sum_{s=1}^{4} p_s = 1.1 + 0 + 1.0 = 2.1 \end{cases}$$

	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

Main Table

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

Root Table

EXAMPLE - Finding An Optimal Binary Search Tree

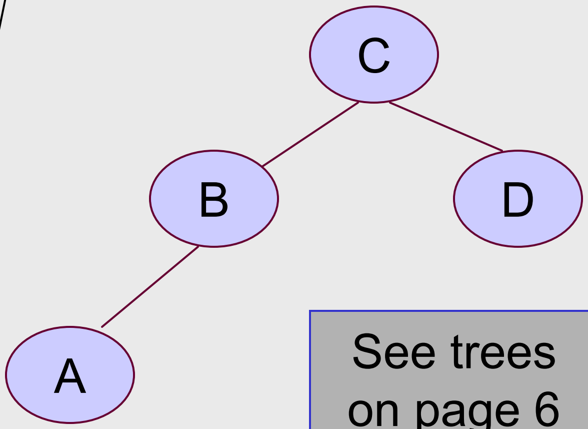
	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

Main Table

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

Root Table

Average number of key comparisons is 1.7.



Optimal BST

See trees on page 6

```
Algorithm OptimalBST(P[1..n])
//Finds optimal BST using 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 key searches in the optimal BST
//        and a table R, of the subtree 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] ← k
          sum ← P[i];
          for s ← i+1 to j do
               sum ← sum + P[s]
          C[i,j] ← minval + sum
return (C[1,n], R)
```