CSE331 Introduction to Algorithms
Lecture 22
Optimal Binary Search Trees

Antoine Vigneron
antoine@unist.ac.kr

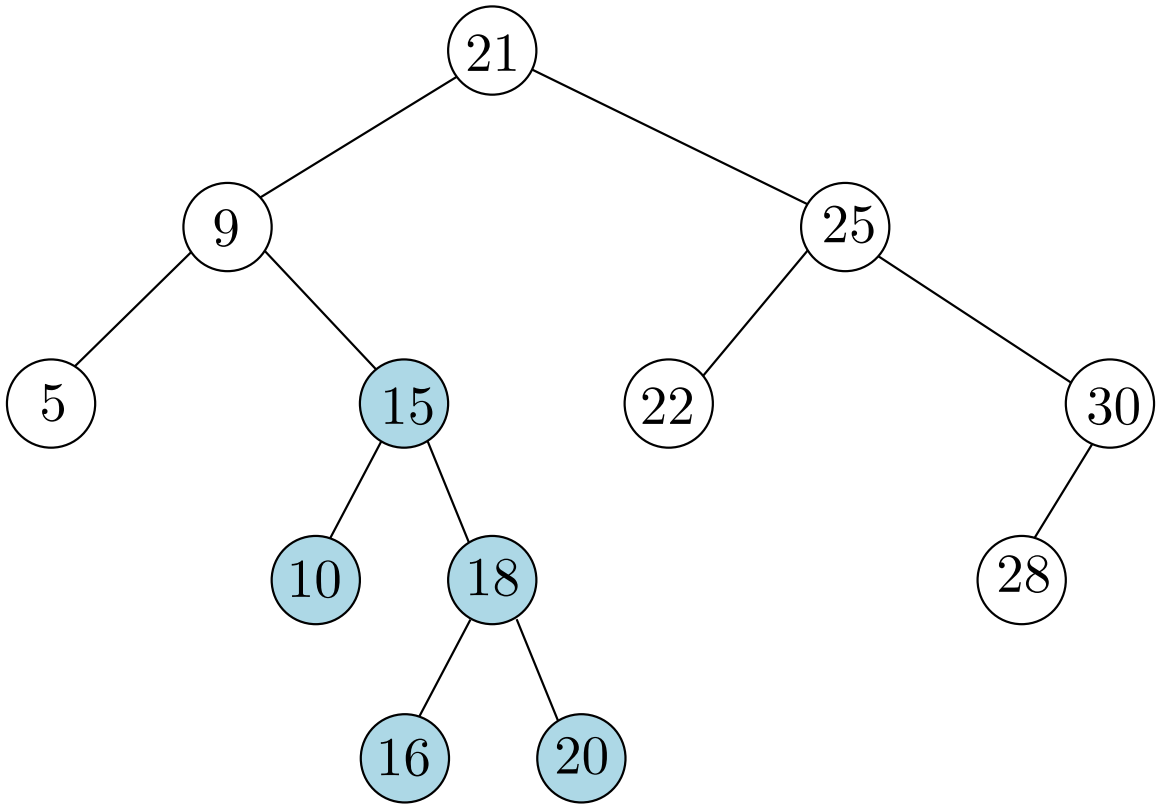
Ulsan National Institute of Science and Technology

July 23, 2021

Introduction

- In this lecture, we will show how to compute optimal *binary search trees* (BST) by dynamic programming.
- **Reference**: Section 12 (p. 286) and Section 15.5 of the textbook (p. 397)
  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

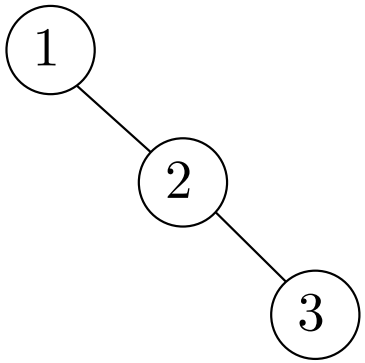
Subtrees of a BST (from previous lecture)



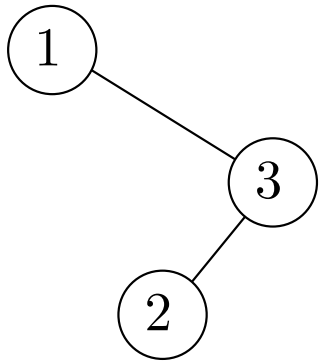
- BST with set of keys $\{5, 9, 10, 15, 16, 18, 20, 21, 22, 25, 28, 30\}$.

BST Insertion Orders (from previous lecture)

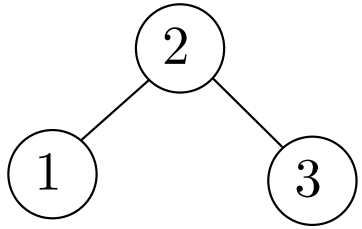
- The shape of a BST depends on the order of insertions.



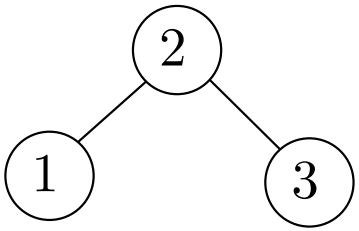
$1 \to 2 \to 3$



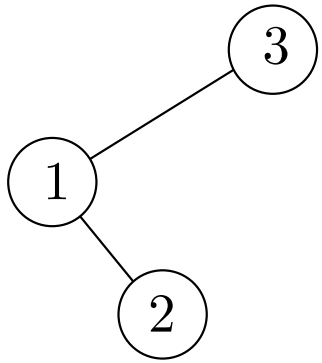
$1 \to 3 \to 2$



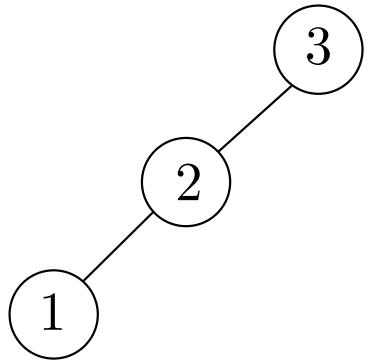
$2 \to 1 \to 3$



$2 \to 3 \to 1$



$3 \to 1 \to 2$



$3 \to 2 \to 1$

Optimal Binary Search Trees

- We are given a set of keys $k_1 < k_2 < \cdots < k_n$.
- We want to build a BST on this set of keys, so as to answer search queries efficiently.
- We know in advance the probability distribution of the queries.

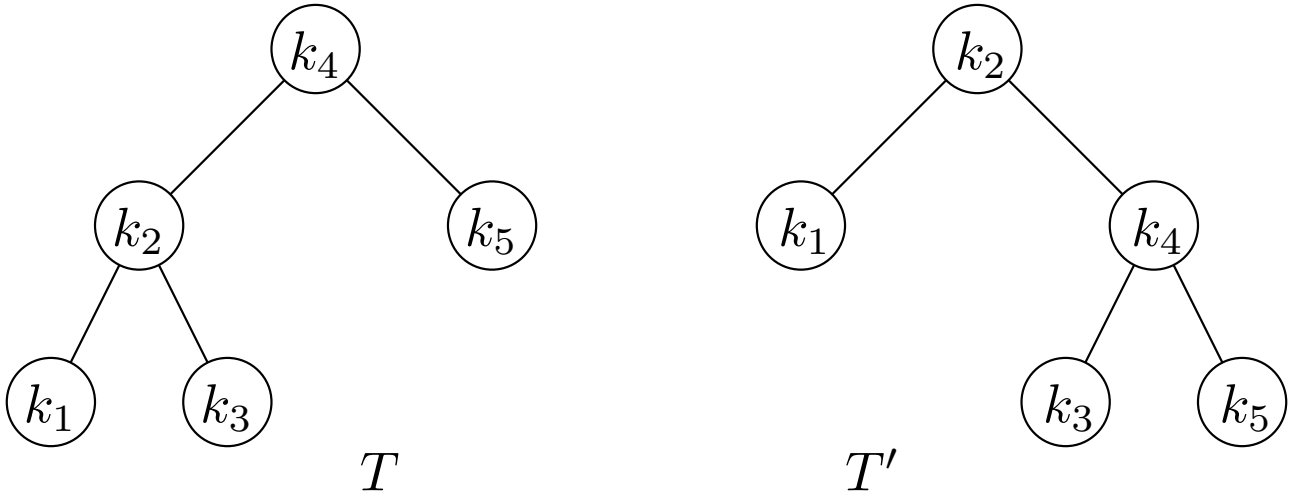
Example

i	1	2	3	4	5
p_i	0.1	0.2	0.1	0.2	0.4

Problem

Given $p_1, \ldots, p_n$, build a BST with keys $k_1, \ldots, k_n$ that minimizes the average search time.

Example

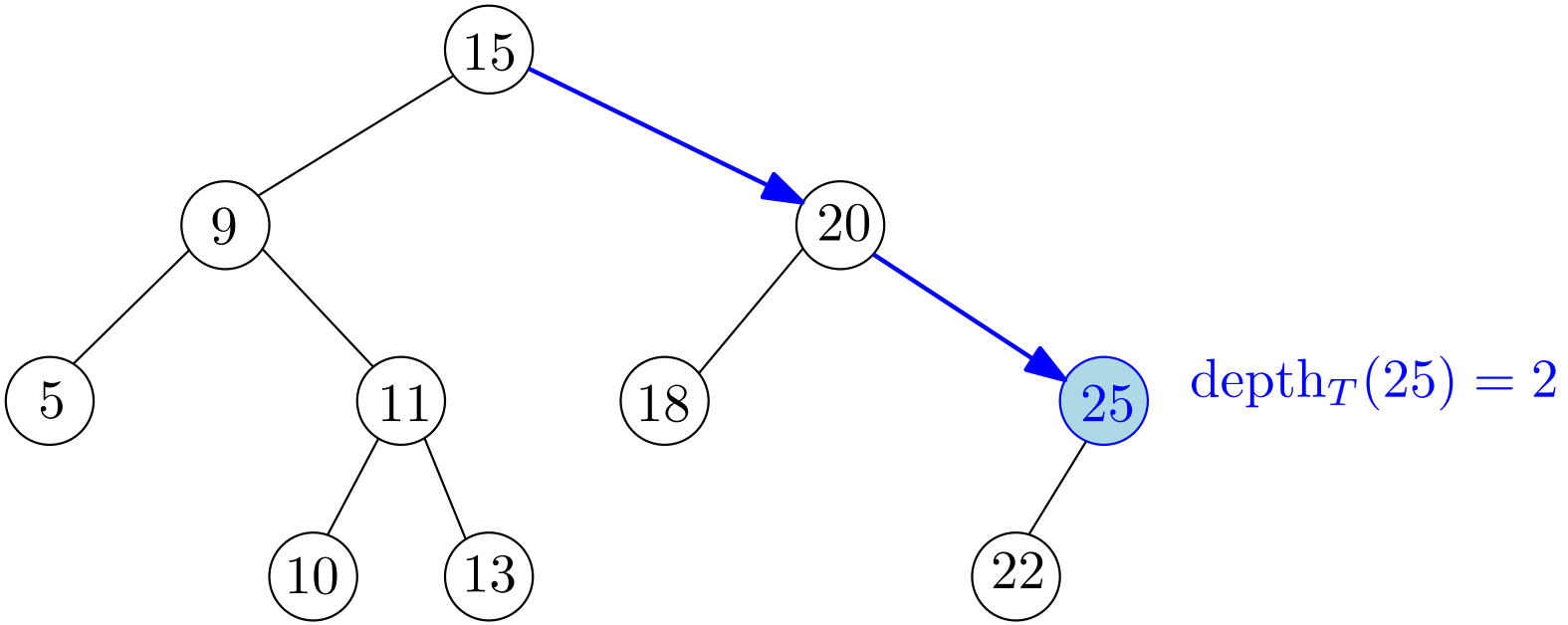


- Probabilities:

i	1	2	3	4	5
p_i	0.1	0.2	0.1	0.2	0.4

- Which tree is better?
- Intuitively, T is better because k_5 is the most frequent key we search, and the search path to k_5 is shorter.

Depth of a Node



Definition

The *depth* of a node v in tree T is the number of edges in the path from the root to v. We denote it by $\text{depth}_T(v)$. As we assume that the keys are distinct, we write $\text{depth}_T(k_i)$ for the depth of the node with key k_i.

Problem Formulation

- We assume that the search time for a node v is proportional to $\text{depth}_T(v) + 1$, which is the number of nodes on the path from the root to v.
- Then the expected search time in a tree T is proportional to

$$\sum_{i=1}^{n} p_i(\text{depth}_T(k_i) + 1) = \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} p_i \,\text{depth}_T(k_i)$$

- So we want to find a tree T that minimizes

$$\text{cost}(T) = \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} p_i \,\text{depth}_T(k_i).$$

- Remark: We will not assume that $\sum_{i=1}^{n} p_i = 1$, even though in the original problem it is true because these numbers are probabilities.

Problem Formulation

Problem

Given keys $k_1 < k_2 < \cdots < k_n$ and weights $p_1, \ldots, p_n$, the optimal BST problem is to find a binary search tree T^* over $k_1, \ldots, k_n$ such that

$$\text{cost}(T^*) = \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} p_i \, \text{depth}_{T^*}(k_i)$$

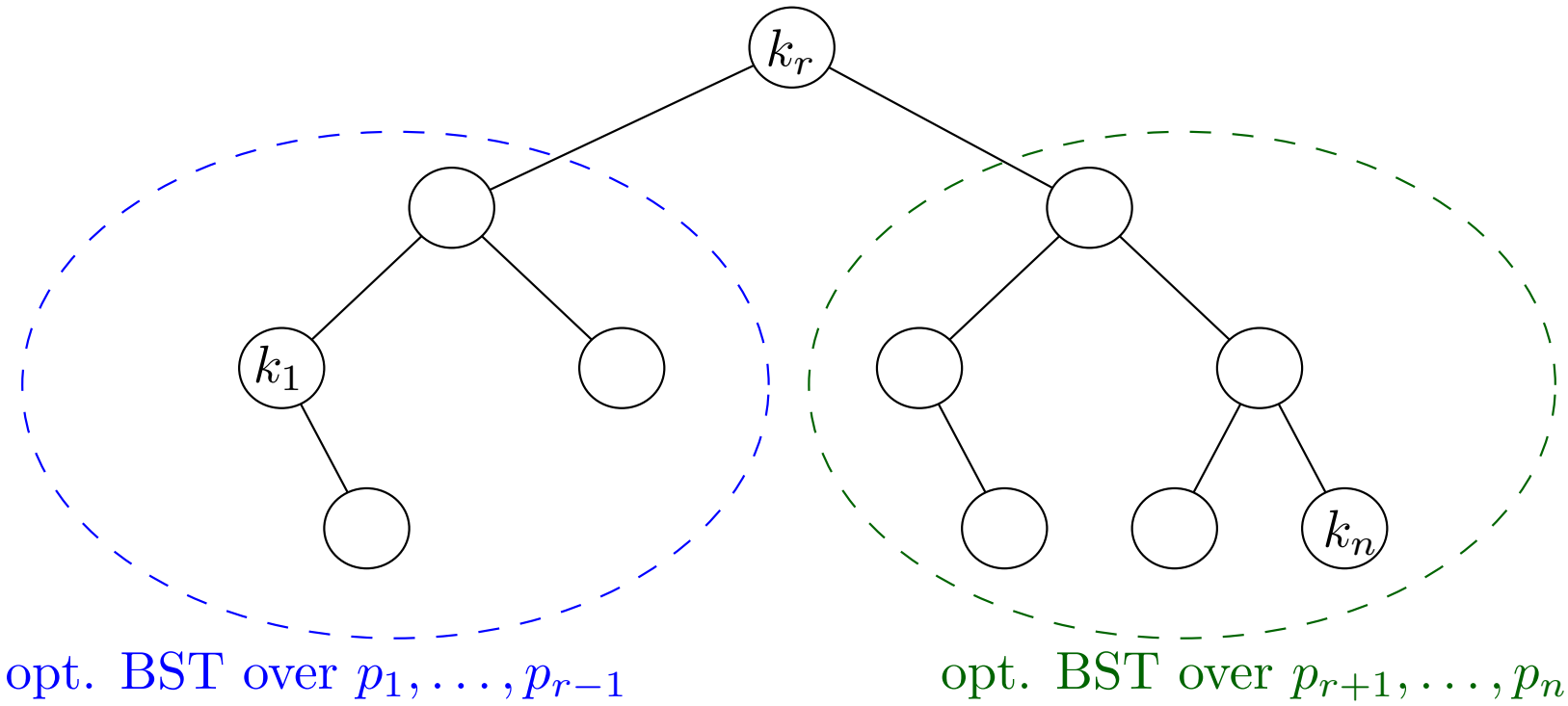
is minimized. The optimal cost is denoted

$$\text{cost}^*(p_1, \ldots, p_n) = \text{cost}(T^*).$$

Brute Force Approach

- A brute force approach would enumerate all possibles BSTs over $k_1, \ldots, k_n$, compute their costs, and return the smallest.
- It would take exponential time because there is an exponential number of such BSTs.
  - Proof?
- So we will try dynamic programming.

Structure of an Optimal BST



Lemma

If T^* is an optimal BST, then the left and right subtrees at his root k_r are optimal BSTs over weights $(p_1, \ldots, p_{r-1})$ and $(p_{r+1}, \ldots, p_n)$, respectively.

- Proof: see exercise set.

Recurrence Relation

- Minimizing over all possible choices for the root, we obtain the following recurrence relation.

Corollary

Let $c[i,j] = \text{cost}^*(p_i, \ldots, p_j)$. *Then*

$$c[i,j] = \left(\sum_{k=i}^{j} p_k\right) + \min_{i \leqslant r \leqslant j} \Big(c[i, r-1] + c[r+1, j]\Big),$$

where $c[i, i-1] = c[j+1, j] = 0$ *is the cost of an empty subtree.*

Recursive Solution

Pseudocode

1: **procedure** $\textsc{Cost}^*(p, i, j)$
2: **if** $j = i - 1$ **then** ▷ base case
3: **return** 0
4: result $\leftarrow \infty$
5: **for** $r \leftarrow i, j$ **do** ▷ minimization
6: result $\leftarrow \min\Big(\text{result}, \textsc{Cost}^*(p, i, r-1) + \textsc{Cost}^*(p, r+1, j)\Big)$
7: **for** $k \leftarrow i, j$ **do**
8: result $\leftarrow$ result $+ p[k]$ ▷ adding $\sum_{k=i}^{j} p_k$
9: **return** result

- What is the problem with this algorithm?
- It runs in exponential time (see exercise set).
- So we will use dynamic programming

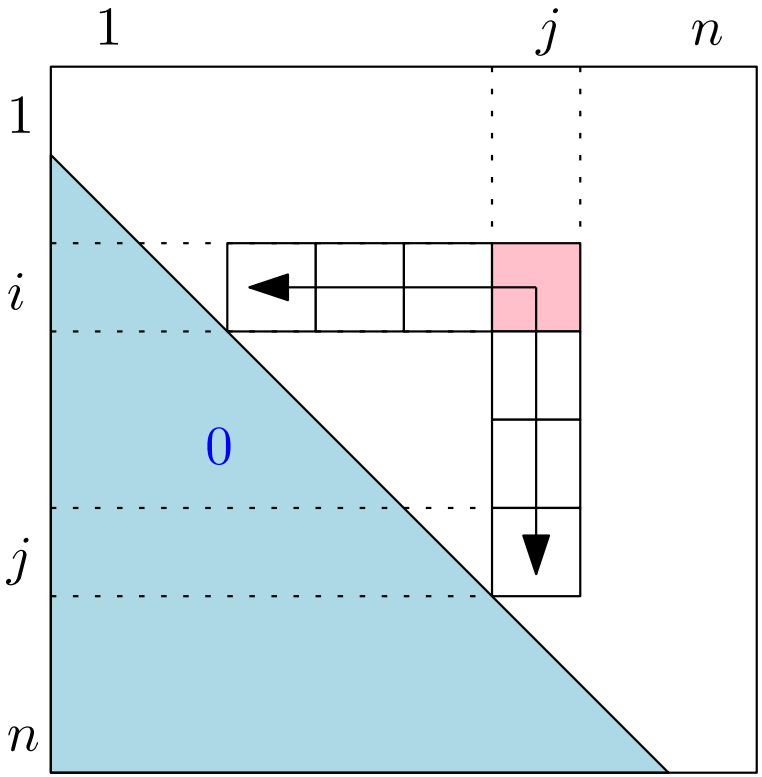
Dynamic Programming: First Attempt

Pseudocode

```
1:  procedure DPCost(p, 1, n)
2:      c[1 ... n+1, 0 ... n] ← new array
3:      for i ← 1, n+1 do                                  ▷ base cases
4:          c[i, i−1] ← 0
5:      for i ← 1, n do                                    ▷ recursive case
6:          for j ← i, n do
7:              c[i, j] ← ∞
8:              for r ← i, j do
9:                  c[i, j] ← min(c[i, j], c[i, r−1] + c[r+1, j])
10:             for k ← i, j do
11:                 c[i, j] ← c[i, j] + p[k]
12:     return c[1, n]
```

Dynamic Programming: First Attempt

- What is the problem with this algorithm?
- It does not compute the values $c[i, j]$ in the right order.
- For instance, $c[1, 2]$ is obtained from $c[1, 1]$ and $c[2, 2]$, but $c[2, 2]$ is computed after $c[1, 2]$.

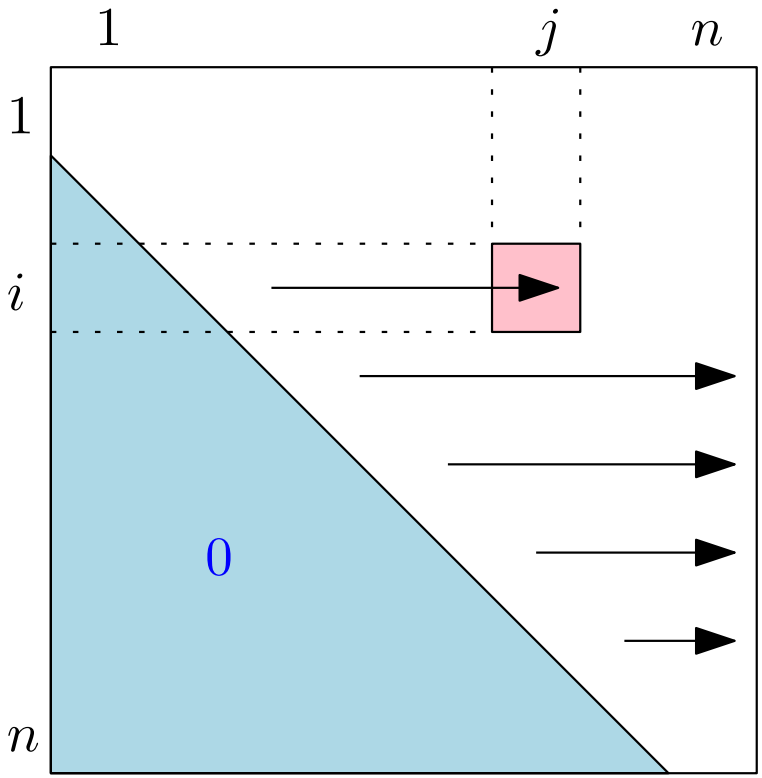


$c[i, j]$ depends on:

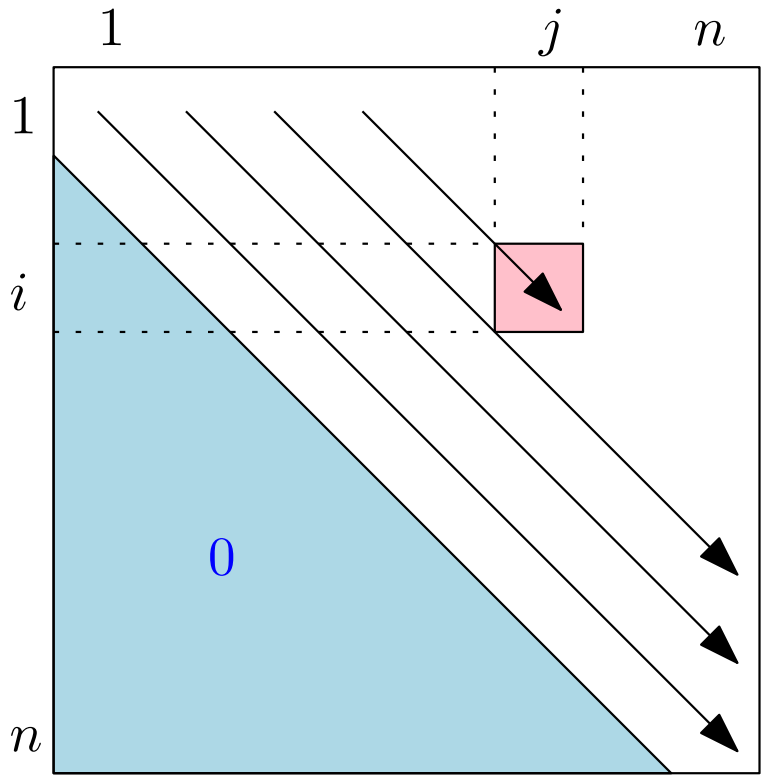
- $c[i, j-1], c[i, j-2], \ldots, c[i, i]$
- $c[i+1, j], c[i+2, j], \ldots, c[j, j]$

Fixing the Problem

- Solution 1: We can always use memoization (left as an exercise)
- Solution 2: Bottom-up approach in a different order



(a)



(b)

Solution (a)

Pseudocode

```
1:  procedure DPCost(p, 1, n)
2:      c[1 ... n+1, 0 ... n] ← new array
3:      for i ← 1, n+1 do                                  ▷ base cases
4:          c[i, i−1] ← 0
5:      for i ← n, 1 do                                    ▷ recursive case
6:          for j ← i, n do
7:              c[i, j] ← ∞
8:              for r ← i, j do
9:                  c[i, j] ← min(c[i, j], c[i, r−1] + c[r+1, j])
10:             for k ← i, j do
11:                 c[i, j] ← c[i, j] + p[k]
12:     return c[1, n]
```

Solution (b)

Pseudocode

```
1:  procedure DPCost(p, 1, n)
2:      c[1 ... n+1, 0 ... n] ← new array
3:      for i ← 1, n+1 do                                  ▷ base cases
4:          c[i, i−1] ← 0
5:      for s ← 0, n−1 do                                  ▷ recursive case
6:          for i ← 1, n−s do
7:              j ← i+s
8:              c[i, j] ← ∞
9:              for r ← i, j do
10:                 c[i, j] ← min(c[i, j], c[i, r−1] + c[r+1, j])
11:             for k ← i, j do
12:                 c[i, j] ← c[i, j] + p[k]
13:     return c[1, n]
```

Comparison

- Solution (a) was easier: We only changed one line of code!
- Solution (b) proceeds by *increasing size* s.
- This is a general approach in bottom-up DP: define a size for the subproblems, and solve the smaller problems first.
- Here the size is $s = j - i$, which is the number of nodes in the tree (minus one).

Analysis

- Both algorithms run in $\Theta(n^3)$ time due to the three nested loops.
  (Detailed analysis: see exercise set.)
- Lines 10-11 on Slide 18 take $\Theta(n^3)$ time as well.
- We can do the same in $\Theta(n^2)$ time. (See next slide.)
- The overall running time is still $\Theta(n^3)$, but it should be faster by a constant factor in practice.

Optimized Solution

Precomputing all sums $P[i,j] = \sum_{k=i}^{j} p_k$

```
1: procedure InitP(p, 1, n, P)
2:     for i ← 1, n do
3:         P[i, i] ← p[i]
4:         for j ← i + 1, n do
5:             P[i, j] ← P[i, j − 1] + p[j]
```

- This procedure takes $\Theta(n^2)$ time.

Optimized Solution

Pseudocode

```
1:  procedure DPCost(p, 1, n)
2:      c[1...n+1, 0...n] ← new array
3:      P[1...n, 1...n] ← new array
4:      InitP(p, 1, n, P)
5:      for i ← 1, n+1 do                                   ▷ base cases
6:          c[i, i−1] ← 0
7:      for i ← n, 1 do                                     ▷ recursive case
8:          for j ← i, n do
9:              c[i, j] ← ∞
10:             for r ← i, j do
11:                 c[i, j] ← min(c[i, j], c[i, r−1] + c[r+1, j])
12:             c[i, j] ← c[i, j] + P[i, j]
13:     return c[1, n]
```

Computing an Optimal BST

Pseudocode

```
1:  procedure DPCost(p, 1, n, R)
2:      c[1...n+1, 0...n] ← new array
3:      P[1...n, 1...n] ← new array
4:      InitP(p, 1, n, P)
5:      for i ← 1, n+1 do                                   ▷ base cases
6:          c[i, i−1] ← 0
7:      for i ← n, 1 do                                     ▷ recursive case
8:          for j ← i, n do
9:              c[i, j] ← ∞
10:             for r ← i, j do
11:                 if c[i, r−1] + c[r+1, j] < c[i, j] then
12:                     c[i, j] ← c[i, r−1] + c[r+1, j]
13:                     R[i, j] ← r                         ▷ root of optimal subtree T*[i, j]
14:             c[i, j] ← c[i, j] + P[i, j]
```

Computing an Optimal BST

Pseudocode

```
1: procedure OPTBST(p, 1, n, R)
2:     R[1...n, 1...n] ← new array
3:     DPCOST(p, 1, n, R)
4:     return RECBST(1, n, R)
```

Pseudocode

```
1: procedure RECBST(i, j, R)
2:     if i > j then
3:         return empty tree
4:     r ← R[i, j]
5:     T ← NEWNODE (k_r)
6:     left(T) ← RECBST(i, r − 1, R)
7:     right(T) ← RECBST(r + 1, j, R)
8:     return T
```

Computing an Optimal BST

- OPTBST takes $O(n)$ time
- As usual, it is easy to reconstruct a solution if we are able to compute its value, by a simple modification of the algorithm.

Concluding Remarks

- The brute force approach takes exponential time, and dynamic programming takes cubic time.
- So once again dynamic programming improved the running time from exponential to polynomial.
- A running time of $\Theta(n^3)$ suggests that we can solve the problem up to roughly $n = 1000$. The brute force approach probably fails for n less than 50, as we saw in the case of binomial coefficients.
- Difference with the textbook: The textbook proves a more general result where we assign a probability q_i to each interval (k_{i-1}, k_i). It makes it more technical, but the main idea is the same.