

# 08. Dynamic Programming for Longest Common Subsequence and Optimal Binary Search Trees

CPSC 535

Kevin A. Wortman



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

## Big Idea: Alternative Kinds of Solutions

- ▶ So far
  - ▶ Step 2. Derive a **recurrence** for an optimal value.
  - ▶ Recall rod cutting:

$$r_i = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

- ▶ Recall matrix chain multiplication:

$$r_{i,j} = \min_{i \leq k \leq j} r_{i,k} + r_{k+1,j} + p_{i-1}p_kp_j$$

- ▶ Now: longest common subsequence (LCS)
  - ▶ not simply minimizing/maximizing one expression
  - ▶ instead, choose between **three alternatives**
  - ▶ **2D table**, like matrix chain

## Subsequences

- ▶ Let  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  be two sequences
- ▶ Define **prefix** notation:  $X_k = \langle x_1, \dots, x_k \rangle$ ;  $X_0 = \langle \rangle$ 
  - ▶ if  $X = \langle 2, 7, 8, 1, 7, 1, 2 \rangle$  then  $X_3 = \langle 2, 7, 8 \rangle$
- ▶ Informally: a **subsequence** of  $Y$  is a copy of  $Y$  with some elements removed
- ▶ Formally:  $X$  is a **subsequence** of  $Y$  if there exists an increasing sequence of indices  $\langle i_1, i_2, \dots, i_k \rangle$  such that, for all  $j \in [1, k]$ ,  $x_j = y_{i_j}$
- ▶ Example: for  $X = \langle B, C, D, B \rangle$  and  $Y = \langle A, B, C, B, D, A, B \rangle$ ,  $X$  is a subsequence of  $Y$  with index sequence  $\langle 2, 3, 5, 7 \rangle$

## Common Subsequence

- ▶  $Z$  is a **common subsequence** of  $X$  and  $Y$ , if  $Z$  is a subsequence  $X$  and  $Z$  is a subsequence of  $Y$
- ▶ a **longest common subsequence** is a common subsequence of maximum length
- ▶ Example: let  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$
- ▶  $Z = \langle B, C, A \rangle$  is a common subsequence
- ▶  $Z = \langle B, C, B, A \rangle$  is a longest common subsequence

# Longest Common Subsequence

*Longest Common Subsequence (LCS) solution problem*

**input:** sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$

**output:** a longest common subsequence of  $X$  and  $Y$

*Longest Common Subsequence (LCS) value problem*

**input:** (same)

**output:** the length of a longest common subsequence of  $X$  and  $Y$

## Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
  - 4.1 **top-down** alternative: add table base case (**memoization**)
  - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

## Longest Common Subsequence Step 1

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
  - ▶ **solution**: a sequence e.g.  $\langle B, C, B, A \rangle$
  - ▶ **value**: integer length of a sequence e.g. 4
  - ▶ eventual goal is solution
  - ▶ start with value

## Longest Common Subsequence Step 2

2. Derive a **recurrence** for an optimal value.

- ▶ Recall input:  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$
- ▶ Recall *prefix*:  $X_i$  is first  $i$  elements of  $X$
- ▶ Define  $LCS(X, Y) \equiv$  length of longest common subsequence of  $X$  and  $Y$
- ▶ We need to define  $LCS$  recursively



## Longest Common Subsequence Step 2

2. Derive a **recurrence** for an optimal value.

- ▶ **Idea:** If last symbols  $x_m = y_n$  match, then extend a shorter common subsequence:

$$LCS(X, Y) = LCS(X_{m-1}, Y_{n-1}) + 1$$

- ▶ Else ( $x_m \neq y_n$ ), have to omit  $x_m$  or  $y_n$ 
  - ▶ Omit  $x_m$ :  $LCS(X, Y) = LCS(X_{m-1}, Y)$
  - ▶ Omit  $y_n$ :  $LCS(X, Y) = LCS(X, Y_{n-1})$
  - ▶ Want **longest** so

$$LCS(X, Y) = \max(LCS(X_{m-1}, Y), LCS(X, Y_{n-1}))$$

## Example

- ▶ Suppose  $X = \langle A, B, A, D \rangle$  and  $Y = \langle B, B, A, C, D \rangle$
- ▶ Last symbols match,  $x_4 = y_5 = D$ , so

$$\begin{aligned} LCS(X, Y) &= LCS(X_{m-1}, Y_{n-1}) + 1 \\ &= LCS(\langle A, B, A \rangle, \langle B, B, A, C \rangle) + 1 \end{aligned}$$

- ▶ Now suppose  $X = \langle A, B, A, D \rangle$  and  $Y = \langle B, B, A, C, C \rangle$
- ▶ Last symbols differ ( $x_4 = D$  but  $y_5 = C$ ), so

$$\begin{aligned} LCS(X, Y) &= \max(LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})) \\ &= \max(LCS(\langle A, B, A \rangle, \langle B, B, A, C, C \rangle), LCS(\langle A, B, A, D \rangle, \langle B, B, A, C \rangle)) \end{aligned}$$

## Longest Common Subsequence Step 2

2. Derive a **recurrence** for an optimal value.

$$LCS(X_m, Y_n) = \begin{cases} 0 & m = 0 \\ 0 & n = 0 \\ LCS(X_{m-1}, Y_{n-1}) + 1 & x_m = x_n \\ \max(LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})) & \text{otherwise} \end{cases}$$

## Longest Common Subsequence Step 3

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
1: function LCS-DC( $X[1..m]$ ,  $Y[1..n]$ )
2:   if  $m == 0$  or  $n == 0$  then
3:     return 0
4:   else if  $X[m] == Y[n]$  then
5:     return LCS-DC( $X[1..m-1]$ ,  $Y[1..n-1]$ ) + 1
6:   else
7:     return max(LCS-DC( $X[1..m-1]$ ,  $Y[1..n]$ ), LCS-DC( $X[1..m]$ ,  $Y[1..n-1]$ ))
8:   end if
9: end function
```

## Matrix Chain Multiplication Step 4.a

4. Design a dynamic programming algorithm that computes an **optimal value**.

4.1 **top-down** alternative: add table base case (**memoization**)

- ▶ Recall **memoization**: use a hash dictionary to make a “memo” of pre-calculated solutions
- ▶ create hash table  $T$
- ▶ use pair  $(m, n)$  as key in table  $T$ , storing  $LCS(X_m, Y_n)$

## Matrix Chain Multiplication Step 4.a

```
1: function LCS-MEMOIZED( $X[1..m]$ ,  $Y[1..n]$ )
2:   HASH-TABLE-CREATE( $T$ )
3:   return LCS-M( $T$ ,  $X$ ,  $Y$ )
4: end function
5: function LCS-M( $T$ ,  $X[1..m]$ ,  $Y[1..n]$ )
6:    $q$  = HASH-TABLE-SEARCH( $T$ ,  $(m, n)$ )
7:   if  $q \neq \text{NIL}$  then
8:     return  $q$ 
9:   end if
10:  if  $m == 0$  or  $n == 0$  then
11:     $q = 0$ 
12:  else if  $X[m] == Y[n]$  then
13:     $q = \text{LCS-M}(T, X[1..m-1], Y[1..n-1]) + 1$ 
14:  else
15:     $q = \max(\text{LCS-M}(X[1..m-1], Y[1..n]), \text{LCS-M}(X[1..m], Y[1..n-1]))$ 
16:  end if
17:   $q.\text{key} = (m, n)$ 
18:  HASH-TABLE-INSERT( $q$ )
19:  return  $q$ 
20: end function
```

## Memoized Algorithm Analysis

- ▶  $T$  contains  $\Theta(n^2)$  pairs  $(m, n)$
- ▶ each entry is inserted exactly once
- ▶ in the general case, LCS-M takes  $\Theta(1)$  expected time
- ▶  $\Rightarrow$  LCS-MEMOIZED takes  $\Theta(n^2)$  expected time

## Longest Common Subsequence Step 4.b

4. Design a dynamic programming algorithm that computes an **optimal value**.
    - 4.1 **top-down** alternative: add table base case (**memoization**)
    - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
- ▶ create 2D array  $c$  where  $c[i][j] = LCS(X_i, Y_j)$
  - ▶ **bottom-up**: write an explicit **for** loop that computes and stores every general case
  - ▶ need to order loops so we never use an uninitialized element
  - ▶  $\therefore$  initialize all base cases before any general case



## Longest Common Subsequence Step 4.b

```
1: function LCS-BU( $X[1..m]$ ,  $Y[1..n]$ )
2:   Create array  $c[0..m][0..n]$ 
3:   for  $i$  from 0 to  $m$  do
4:      $c[i][0] = 0$ 
5:   end for
6:   for  $j$  from 1 to  $n$  do
7:      $c[0][j] = 0$ 
8:   end for
9:   for  $i$  from 1 to  $m$  do
10:    for  $j$  from 1 to  $n$  do
11:      if  $X[i] == Y[j]$  then
12:         $c[i][j] = c[i-1][j-1] + 1$ 
13:      else
14:         $c[i][j] = \max(c[i-1][j], c[i][j-1])$ 
15:      end if
16:    end for
17:  end for
18:  return  $c[m][n]$ 
19: end function
```

▷ unusual index range

▷ only initialize  $c[0][0]$  once

## Bottom-Up Analysis

- ▶ LCS-BU is clearly  $\Theta(n^2)$  time
- ▶ (easy analysis)

## Longest Common Subsequence Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

▶ **idea:** for each  $(i, j)$ , record which alternative sub-solution defines  $c[i][j]$  :

- ▶  $\nwarrow \equiv c[i-1][j-1]$
- ▶  $\uparrow \equiv c[i-1][j]$
- ▶  $\leftarrow \equiv c[i][j-1]$

▶ define

$$b[i][j] \in \{\nwarrow, \uparrow, \leftarrow\}$$

▶ rewrite  $\max(c[i-1][j], c[i][j-1])$  as **if/else** so we can update  $b[i][j]$

## Longest Common Subsequence Step 5

```

1: function LCS-SOLUTION( $X[1..m]$ ,  $Y[1..n]$ )
2:   Create arrays  $c[0..m][0..n]$  and  $b[1..m][1..n]$ 
3:   for  $i$  from 0 to  $m$  do
4:      $c[i][0] = 0$ 
5:   end for
6:   for  $j$  from 1 to  $n$  do
7:      $c[0][j] = 0$ 
8:   end for
9:   for  $i$  from 1 to  $m$  do
10:    for  $j$  from 1 to  $n$  do
11:      if  $X[i] == Y[j]$  then
12:         $c[i][j] = c[i-1][j-1] + 1$ 
13:         $b[i][j] = \nwarrow$ 
14:      else if  $c[i-1][j] \geq c[i][j-1]$  then
15:         $c[i][j] = c[i-1][j]$ 
16:         $b[i][j] = \uparrow$ 
17:      else
18:         $c[i][j] = c[i][j-1]$ 
19:         $b[i][j] = \leftarrow$ 
20:      end if
21:    end for
22:  end for
23:  return LCS-BTRACK( $b, X, m, n$ )
24: end function

```

▷ different index ranges

▷ only initialize  $c[0][0]$  once

## Longest Common Subsequence Step 5

```
1: function LCS-BTRACK( $b[1..m][1..n], X[1..m], i, j$ )
2:   if  $i == 0$  or  $j == 0$  then
3:     return  $\langle \rangle$ 
4:   end if
5:   if  $b[i][j] == \nwarrow$  then
6:     return  $\text{LCS-BTRACK}(b, X, i - 1, j - 1) + \langle X[i] \rangle$ 
7:   else if  $b[i][j] == \uparrow$  then
8:     return  $\text{LCS-BTRACK}(b, X, i - 1, j)$ 
9:   else
10:    return  $\text{LCS-BTRACK}(b, X, i, j - 1)$ 
11:  end if
12: end function
```

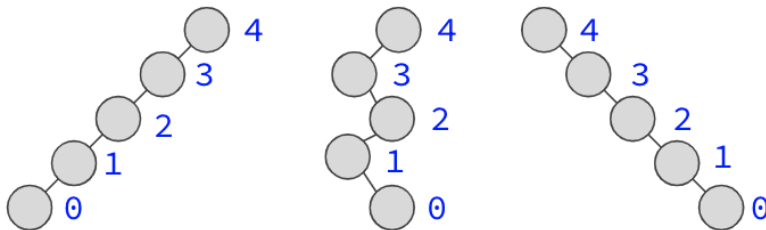
▷ empty sequence

▷ append

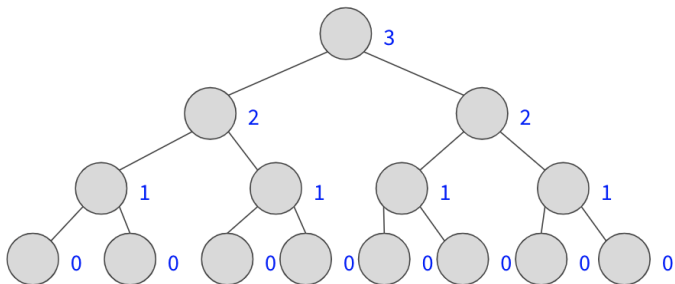
## Review: Binary Search Trees

- ▶ Recall **Binary Search Tree (BST)**: fundamental data structure
- ▶ **Depth** of node  $x$  = length of path from root to  $x$
- ▶ **Height** of tree = maximum depth of any node
- ▶ Time of a search = **depth** of search path
- ▶ Height
  - ▶ worst case =  $\Theta(n)$
  - ▶ best case =  $\Theta(\log n)$
- ▶ self-balancing BST maintains  $\Theta(\log n)$  height

## Worst-Case BSTs



## Best-Case BST





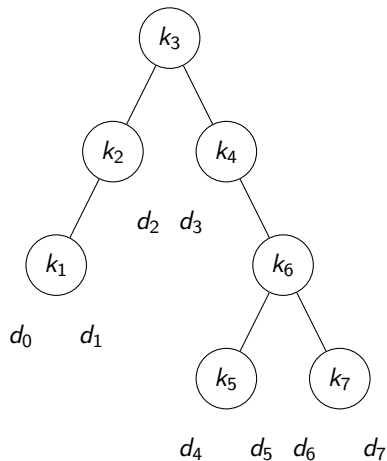
## Optimal BSTs

- ▶ Fix the sequence of search operations
- ▶ **Optimal** BST: minimizes total search time
  - ▶ including constant factors
- ▶ Total time for  $n$  elements and  $k$  searches:
  - ▶ any self-balancing BST:  $O(k \log n)$
  - ▶ optimal BST:  $O(k \log n)$  with lowest possible constant factor
- ▶ Goal
  - ▶ frequently-visited elements near root
  - ▶ rarely-visited elements near leaves
  - ▶ tricky because a path visits multiple nodes; all count

## Problem Setup

- ▶ Given:
  - ▶ ordered keys  $K = \langle k_1, k_2, \dots, k_n \rangle$
  - ▶ “dummy” values  $d_0, d_1, \dots, d_n$  represent values of failed searches, between keys
- ▶ for a given search and index  $i$ ,
  - ▶  $p_i$  = probability that this is a successful search for  $k_i$
  - ▶  $q_i$  = probability that this is a failed search for value  $d_i$

## Problem Setup



## Expected Search Cost

Every search ends in a key  $k_i$  or dummy  $d_i$ , so

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1.$$

For tree  $T$ ,

$$\begin{aligned} E[\text{search in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i)) \cdot q_i \\ &= 1 + \sum_{i=1}^n (\text{depth}_T(k_i)) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i)) \cdot q_i \end{aligned}$$

Have: probabilities  $p_1, \dots, p_n$  and  $q_0, \dots, q_n$

Need: shape  $T$  to minimize sum

## Optimal BST Problem

*Optimal Binary Search Tree (BST) solution problem*

**input:** keys  $K = \langle k_1, k_2, \dots, k_n \rangle$ ; successfull-search probabilities  $p_1, p_2, \dots, p_n$ ; and failed-search probabilities  $q_0, q_1, \dots, q_n$

**output:** a BST  $T$  that contains  $K$  with minimum expected search cost

*Optimal Binary Search Tree (BST) value problem*

**input:** successfull-search probabilities  $p_1, p_2, \dots, p_n$ ; and failed-search probabilities  $q_0, q_1, \dots, q_n$

**output:** the minimum expected search cost of a tree that contains  $K$

(Note: keys  $K$  unneeded for value problem.)

## Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
  - 4.1 **top-down** alternative: add table base case (**memoization**)
  - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

## Optimal BST Step 1

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
  - ▶ **solution:** a BST  $T$
  - ▶ **value:**  $E[\text{search in } T] = 1 + \sum_{i=1}^n (\text{depth}_T(k_i)) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i)) \cdot q_i$
  - ▶ goal is value

## Optimal BST Step 2

2. Derive a **recurrence** for an optimal value.
  - ▶ Make one decision and recurse for the rest
  - ▶ Decision: **choose some key to be root**
  - ▶ Define  $e[i, j] = E[\text{search in optimal tree containing } k_i, \dots, k_j]$
  - ▶ Denote empty tree with  $j = i - 1$
  - ▶ Base case: empty tree; cost is  $q_{i-1}$
  - ▶ General case:
    - ▶ choose a split index  $r$
    - ▶ recursively compute left subtree  $e[i, r - 1]$
    - ▶ recursively compute right subtree  $e[r + 1, j]$
    - ▶ add root on top; increases depths of subtrees



## Optimal BST Step 2

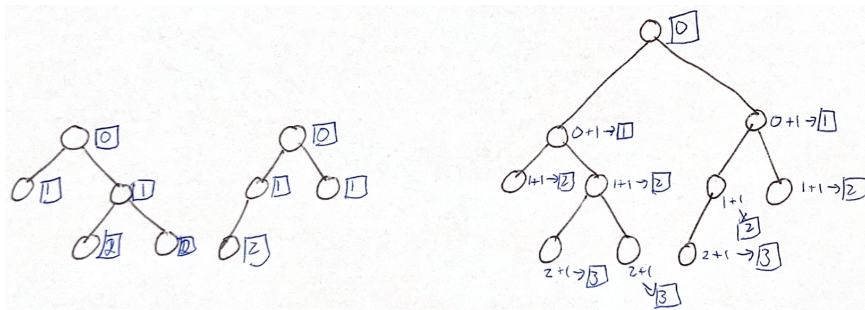
- ▶ Place root atop two subtrees
- ▶ +1 to path length of every descendant
- ▶ Recall

$$\begin{aligned} E[\text{search in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i)) \cdot q_i \\ &= 1 + \sum_{i=1}^n (\text{depth}_T(k_i)) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i)) \cdot q_i \end{aligned}$$

- ▶ +1 to each path increases  $E[\text{search in } T]$  by  $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i$
- ▶ Define

$$w(i, j) = \sum_{k=1}^i p_k + \sum_{k=0}^j q_k$$

## Adding a Root Increments Path Lengths



## Optimal BST Step 2

For a chosen root index  $r$ ,

$$e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j)$$

Optimize by choosing whichever root has minimal total cost:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min_{r \in [i:j]} (e[i, r - 1] + e[r + 1, j] + w(i, j)) & \text{if } i \leq j \end{cases}$$

## Optimal BST Step 3 – core function

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
1: function OBST-REC( $p[1..n]$ ,  $q[0..n]$ ,  $i, j$ )
2:   if  $j == (i - 1)$  then
3:     return  $q[i - 1]$ 
4:   end if
5:    $e = \infty$ 
6:   for  $r$  from  $i$  to  $j$  do
7:      $t = \text{OBST-REC}(p, q, i, r - 1) + \text{OBST-REC}(p, q, r + 1, j) + W(p, q, i, j)$ 
8:     if  $t < e$  then
9:        $e = t$ 
10:    end if
11:  end for
12:  return  $e$ 
13: end function
```

## Optimal BST Step 3 – helper functions

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
1: function OBST-DC( $p[1..n]$ ,  $q[0..n]$ )
2:   return OBST-DC-REC( $p$ ,  $q$ , 1,  $n$ )
3: end function
4: function W( $p[1..n]$ ,  $q[0..n]$ ,  $i$ ,  $j$ )
5:    $w = 0$ 
6:   for  $k$  from  $i$  to  $j$  do
7:      $w = w + p[k]$ 
8:   end for
9:   for  $k$  from  $i - 1$  to  $j$  do
10:     $w = w + q[k]$ 
11:   end for
12:   return  $w$ 
13: end function
```

## Optimal BST Step 4.a

4. Design a dynamic programming algorithm that computes an **optimal value**.

4.1 **top-down** alternative: add table base case (**memoization**)

- ▶ create hash table  $T$
- ▶ use pair  $(i, j)$  as key in table  $T$ , storing  $\text{OBST-REC}(p, q, i, j)$

## Optimal BST Step 4.a – helper functions

```
1: function OBST-MEMOIZED( $p[1..n]$ ,  $q[0..n]$ )
2:   HASH-TABLE-CREATE( $T$ )
3:   return OBST-DC-REC( $p$ ,  $q$ ,  $T$ , 1,  $n$ )
4: end function
5: function W( $p[1..n]$ ,  $q[0..n]$ ,  $i$ ,  $j$ )
6:    $w = 0$ 
7:   for  $k$  from  $i$  to  $j$  do
8:      $w = w + p[k]$ 
9:   end for
10:  for  $k$  from  $i - 1$  to  $j$  do
11:     $w = w + q[k]$ 
12:  end for
13:  return  $w$ 
14: end function
```

## Optimal BST Step 4.a – core function

```
1: function OBST-M( $p[1..n]$ ,  $q[0..n]$ ,  $T$ ,  $i$ ,  $j$ )
2:    $q = \text{HASH-TABLE-SEARCH}(T, (i, j))$ 
3:   if  $q \neq \text{NIL}$  then
4:     return  $q$ 
5:   end if
6:   if  $j == (i - 1)$  then
7:     return  $q[i - 1]$ 
8:   end if
9:    $e = \infty$ 
10:  for  $r$  from  $i$  to  $j$  do
11:     $t = \text{OBST-M}(p, q, T, i, r - 1) + \text{OBST-M}(p, q, T, r + 1, j) + W(p, q, i, j)$ 
12:    if  $t < e$  then
13:       $e = t$ 
14:    end if
15:  end for
16:   $e.\text{key} = (i, j)$ 
17:  HASH-TABLE-INSERT( $e$ )
18:  return  $e$ 
19: end function
```



## Optimal BST Step 4.b

4. Design a dynamic programming algorithm that computes an **optimal value**.
    - 4.1 **top-down** alternative: add table base case (**memoization**)
    - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
- ▶ create 2D array  $e$  where  $e[i][j] = \text{OBST-REC}(p, q, i, j)$
  - ▶ **bottom-up**: write an explicit **for** loop that computes and stores every general case

## Optimal BST Step 4.b

```

1: function OBST-BU( $p[1..n]$ ,  $q[0..n]$ )
2:   Create array  $e[1..n+1][0..n]$ 
3:   for  $i$  from 1 to  $n+1$  do
4:      $e[i][i-1] = q[i-1]$ 
5:   end for
6:   for  $\ell$  from 1 to  $n$  do
7:     for  $i$  from 1 to  $n-\ell+1$  do
8:        $j = i + \ell - 1$ 
9:        $e[i][j] = \infty$ 
10:      for  $r = i$  to  $j$  do
11:         $t = e[i][r-1] + e[r+1][j] + W(p, q, i, j)$ 
12:        if  $t < e[i][j]$  then
13:           $e[i][j] = t$ 
14:        end if
15:      end for
16:    end for
17:  end for
18:  return  $e[1][n]$ 
19: end function

```

▷ unusual index range

▷ base cases

## Optimal BST Bottom-Up Analysis

- ▶ Create array  $e$ :  $\Theta(n^2)$
- ▶ Base cases:  $\Theta(n)$
- ▶ General cases:
  - ▶ **for** loop over  $\ell$ :  $\Theta(n)$  iterations
  - ▶ nested **for** loop over  $i$ :  $\Theta(n)$  iterations
  - ▶ nested **for** loop over  $r$ :  $\Theta(n)$  iterations
  - ▶ call  $W(p, q, i, j)$ :  $\Theta(n)$  time
- ▶ total  $\Theta(n^4)$  time
- ▶ bottleneck is calls to  $W$
- ▶ can precompute and cache  $W$  values in their own table

# Optimal BST Final Draft

```

1: function OBST-BU( $p[1..n]$ ,  $q[0..n]$ )
2:   Create array  $e[1..n+1][0..n]$ 
3:   Create array  $w[1..n+1][0..n]$ 
4:   for  $i$  from 1 to  $n+1$  do
5:      $e[i][i-1] = q[i-1]$ 
6:      $w[i][i-1] = q[i-1]$ 
7:   end for
8:   for  $\ell$  from 1 to  $n$  do
9:     for  $i$  from 1 to  $n-\ell+1$  do
10:       $j = i + \ell - 1$ 
11:       $e[i][j] = \infty$ 
12:       $w[i][j] = w[i][j-1] + p[j] + q[j]$ 
13:      for  $r$  from  $i$  to  $j$  do
14:         $t = e[i][r-1] + e[r+1][j] + w[i][j]$ 
15:        if  $t < e[i][j]$  then
16:           $e[i][j] = t$ 
17:        end if
18:      end for
19:    end for
20:  end for
21:  return  $e[1][n]$ 
22: end function

```

▷ unusual index range

▷  $w[i][j] = W(p, q, i, j)$

▷ base cases

## Optimal BST Final Draft Analysis

- ▶ three nested loop
- ▶ body of innermost loop is now only  $\Theta(1)$
- ▶ OBST-BU takes  $\Theta(n^3)$  time