Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a **technique** (like divide and conquer).
- Developed back in the day when “programming” meant “tabular method”. Doesn’t really refer to computer programming.

  - Invented by American mathematician Richard Bellman in the 1950s
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.

    -Find the best of all possible solutions.
  - Minimization or maximization.

Dynamic Programming

- Like divide and conquer, solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - In the sense that subproblems share subsubproblems.
  - However, solution to one subproblem does not affect the solutions to other subproblems of the same problem. (More on this later.)
- Hence, if divide and conquer approach is used, the same subsubproblem will be solved multiple times.
- DP optimizes by
  - Solving subproblems in a bottom-up fashion.
  - Storing the solution (memoization) to a subproblem in a table the first time it is solved.
  - Performing a lookup for the solution in the table when the subproblem is encountered subsequently.

Principle of Optimality

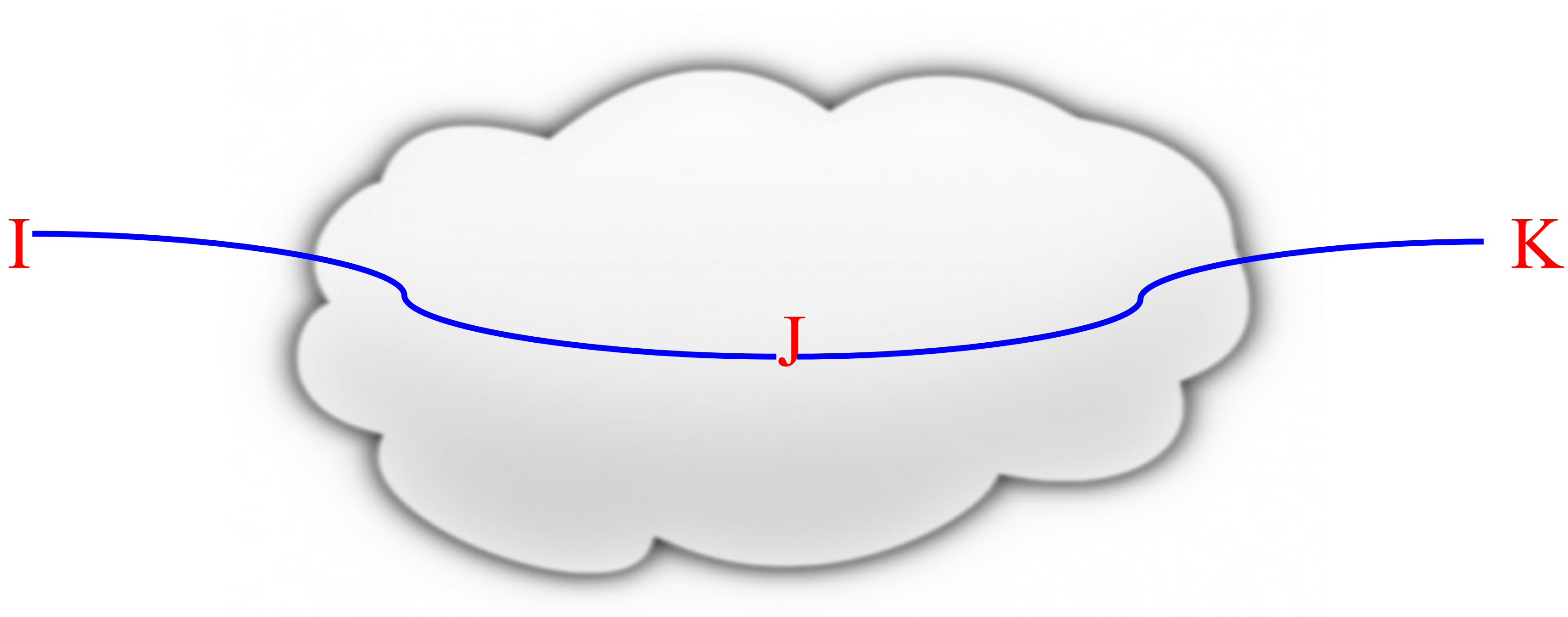
- ***Principle of optimality:*** it says that an optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances.

- This property is also called *optimal substructure property.*

Recall Mergesort. Why isn't it dynamic programming?

Principle of Optimality

- if node J is on the optimal path from node I to node K, then the optimal path from J to K also also falls along the same path



Steps in Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

We'll study these with the help of examples.

Example: Fibonacci Numbers

- Recall definition of Fibonacci numbers:
  - $F(n) = F(n\text{-}1) + F(n\text{-}2)$
  - $F(0) = 0$
  - $F(1) = 1$

- Computing the n^{th} Fibonacci number recursively (top-down):

$$F(n)$$

$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$

$$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$$

…

Example: Fibonacci Numbers (cont.)

– Computing the n^{th} Fibonacci number in bottom-up manner and recording results:

- $F(0) = 0$
- $F(1) = 1$
- $F(2) = 1+0 = 1$
- …
- $F(n-2) =$
- $F(n-1) =$
- $F(n) = F(n-1) + F(n-2)$

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

Efficiency:

- \- time
- \- space

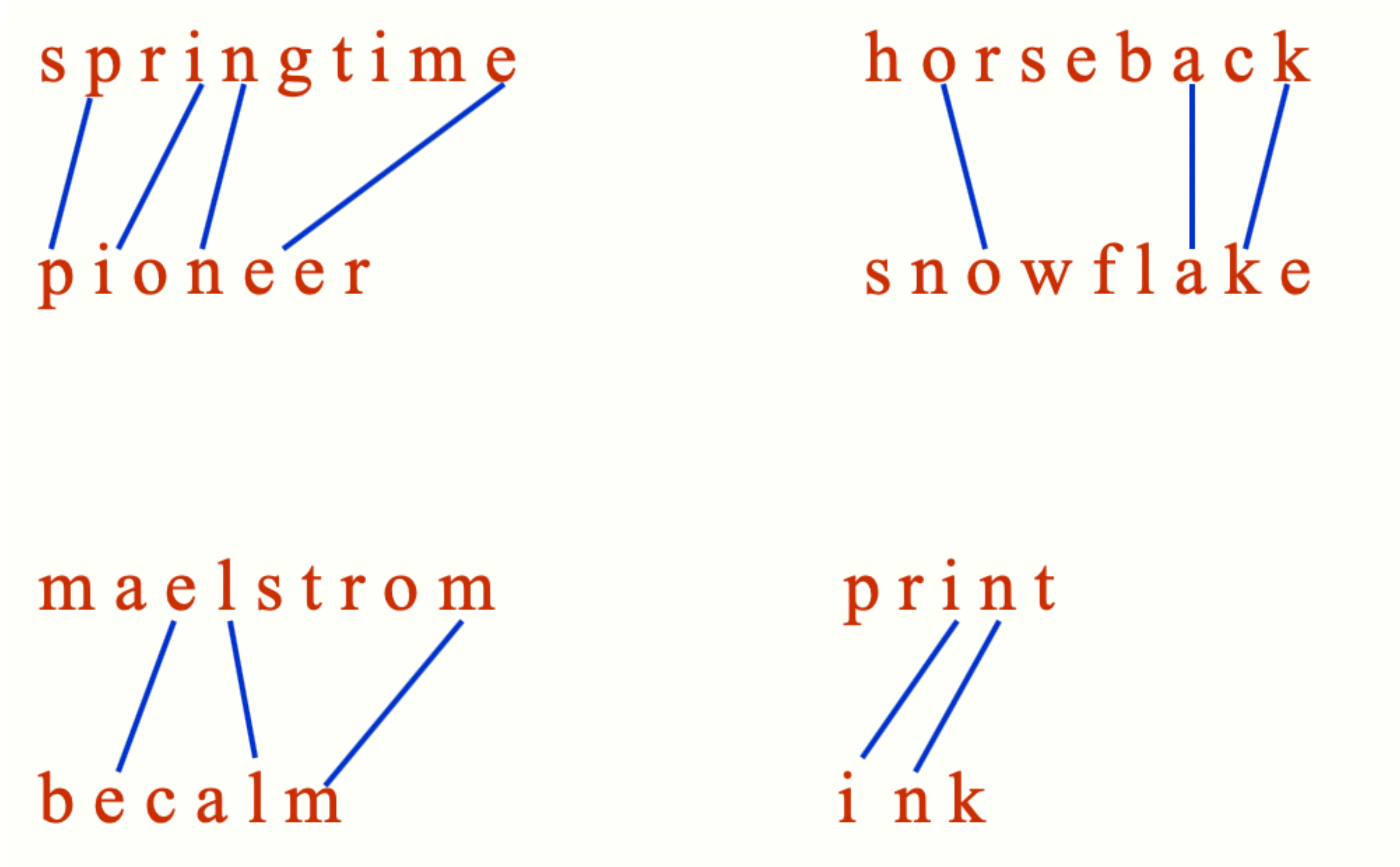
Examples of DP Algorithms

- Matrix Chain Multiplication
- Floyd's algorithm for all-pairs shortest paths
- Longest Common Subsequence
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$.
  - Find a subsequence common to both whose length is longest.
  - A subsequence doesn't have to be consecutive, but it has to be in order.

Examples



springtime
pioneer
horseback
snowflake
maelstrom
becalm
print
ink

Naïve Algorithm

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Optimal Substructure

Notation:

i^{th} prefix of X: X_i = prefix $\langle x_1,\ldots,x_i \rangle$

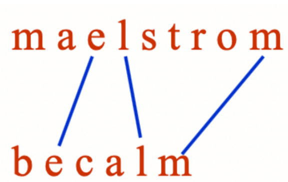
i^{th} prefix of Y: Y_i = prefix $\langle y_1,\ldots,y_i \rangle$

Theorem

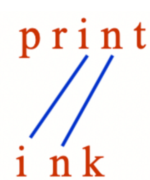
Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k\text{-}1}$ is an LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of $X_{m\text{-}1}$ and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and $Y_{n\text{-}1}$.

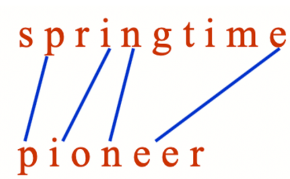
Proof: Straightforward



Case 1



Case 2



Case 3

Recursive Solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y .
1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Can write a recursive algorithm, but it will be inefficient (because subproblems overlap).

Computing the Length of an LCS

<u>LCS-LENGTH (X, Y)</u>

$m \leftarrow length[X]$
$n \leftarrow length[Y]$
for $i \leftarrow 1$ **to** m
    do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
    do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
    do for $j \leftarrow 1$ **to** n
        do if $x_i = y_j$
            then $c[i, j] \leftarrow c[i-1, j-1] + 1$
                $b[i, j] \leftarrow$ "$\nwarrow$"
            else if $c[i-1, j] \geq c[i, j-1]$
                then $c[i, j] \leftarrow c[i-1, j]$
                    $b[i, j] \leftarrow$ "$\uparrow$"
                else $c[i, j] \leftarrow c[i, j-1]$
                    $b[i, j] \leftarrow$ "$\leftarrow$"
return c and b

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

$c[m,n]$ contains the length of an LCS of X and Y.

Time: $O(mn)$

Constructing an LCS

```
PRINT-LCS (b, X, i, j)
1.  if i = 0 or j = 0
2.      then return
3.  if b[i, j] = "↖"
4.      then PRINT-LCS(b, X, i–1, j–1)
5.              print x_i
6.  elseif b[i, j] = "↑"
7.              then PRINT-LCS(b, X, i–1, j)
8.  else PRINT-LCS(b, X, i, j–1)
```

- Initial call is PRINT-LCS (b, X,m, n).
- When $b[i, j] = \nwarrow$, we have extended LCS by one character. So LCS = entries with $\nwarrow$ in them.
- Time: $O(m+n)$

LCS Example

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Optimal Binary Search Trees

- Problem
  - Given sequence $K = k_1, k_2, \ldots, k_n$ of n distinct keys, sorted ($k_1 < k_2 < \cdots < k_\mathrm{n}$).
  - Want to build a binary search tree from the keys.
  - For k_i, have probability p_i that a search is for k_i.
  - Want BST with minimum expected search cost.
  - Search cost = # of items examined.
  - For key k_i, cost = $\mathrm{depth}_T(k_i)+1$, where $\mathrm{depth}_T(k_i)$ = depth of k_i in BST T.
- Note: The root of the tree is at depth 0.

Expected Search Cost

$E[\text{search cost in } T]$

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

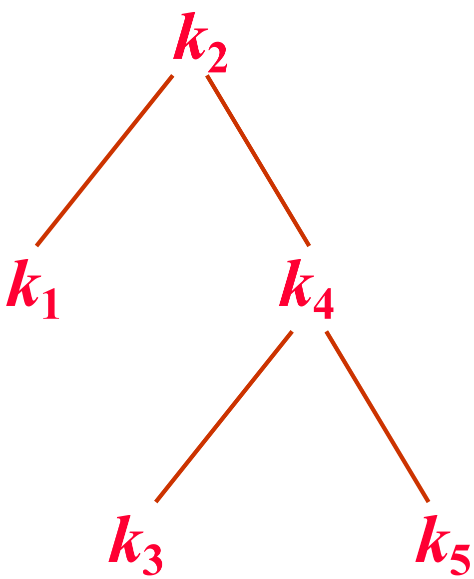
$$= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

Sum of probabilities is 1.

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

Example

- $p_1 = 0.25, p_2 = 0.2, p_3 = 0.05, p_4 = 0.2, p_5 = 0.3.$

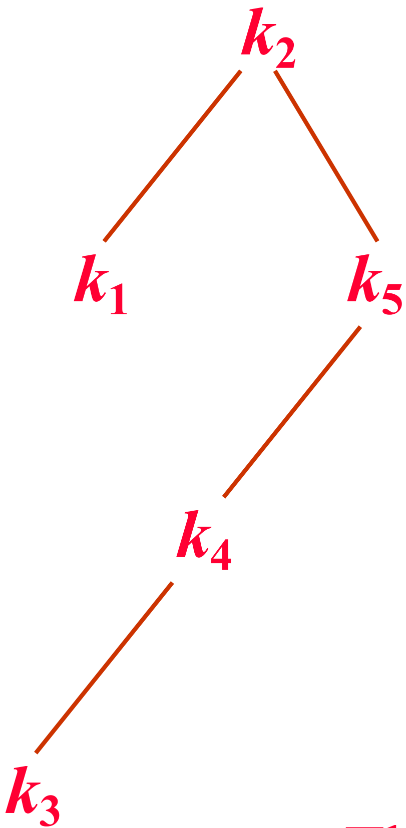


i	$\mathrm{depth}_T(k_i)$	$\mathrm{depth}_T(k_i)\cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Therefore, E[search cost] = 2.15.

Example

- $p_1 = 0.25, p_2 = 0.2, p_3 = 0.05, p_4 = 0.2, p_5 = 0.3.$



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.10

Therefore, E[search cost] = 2.10.

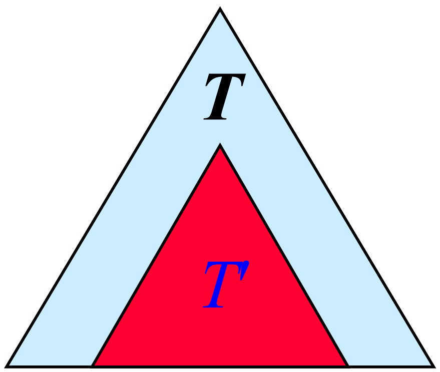
This tree turns out to be optimal for this set of keys.

Example

- **Observations:**
  - Optimal BST might not have smallest height.
  - Optimal BST might not have highest-probability key at root.
- Build by exhaustive checking?
  - Construct each n-node BST.
  - For each, put in keys.
  - Then compute expected search cost.
  - But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

Optimal Substructure

- Any subtree of a BST contains keys in a contiguous range k_i, ..., k_j for some $1 \le i \le j \le n$.

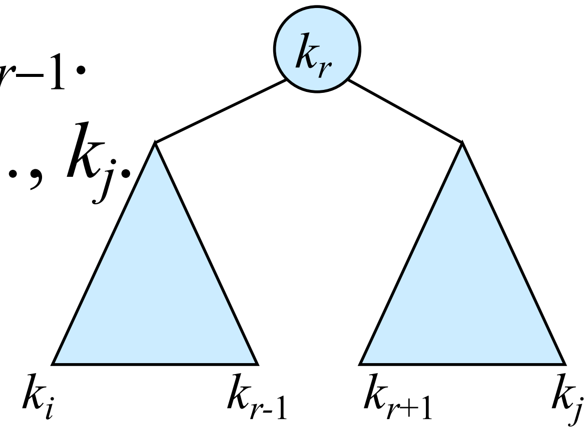


- If T is an optimal BST and T contains subtree T' with keys k_i, ..., k_j , then T' must be an optimal BST for keys k_i, ...,k_j.

Proof: Cut and paste an alternative optimal subtree → contradicts

Optimal Substructure

- For keys $k_i, \ldots, k_j$, one of the keys in $k_i, \ldots, k_j$, k_r, where $i \le r \le j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



- To find an optimal BST:
  – Examine all candidate roots k_r , for $i \le r \le j$.
  – Determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$.

Recursive Solution

- Find optimal BST for k_i, ..., k_j, where $i \geq 1, j \leq n, j \geq i-1$.
- When $j = i-1$, the tree is empty.
- $e[i, j]$ = expected search cost of optimal BST for k_i, ..., k_j.
- If $j = i-1$, then $e[i, j\ ] = 0$.
- If $j \geq i$,
  - Select a root k_r, for some $i \leq r \leq j$.
  - Make an optimal BST with k_i, …, k_{r-1} as the left subtree.
  - Make an optimal BST with k_{r+1}, ..., k_j as the right subtree.

Recursive Solution

- When the OPT subtree becomes a subtree of a node:
  - Depth of every node in OPT subtree goes up by 1.
  - Expected search cost increases by

$$\boxed{w(i,j) = \sum_{l=i}^{j} p_l}$$

- If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

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

$$= e[i,r-1] + e[r+1,j] + w(i,j), \text{ since } w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

- But, we don't know k_r. Hence,

$$\boxed{e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \end{cases}}$$

Computing an Optimal Solution

- Store values in a table:
  - $e[1 \ldots n+1\ ,\ 0 \ldots n]$
- Will use only entries $e[i, j\ ]$, where $j \geq i-1$.
- Will also compute
  - root$[i, j]$ = root of subtree with keys $k_i, \ \ldots, k_j$, for $1 \leq i \leq j \leq n$.
- One other table … don't recompute $w(i, j)$ from scratch every time we need it.
  - Table $w[1 \ldots n+1, 0 \ldots n]$.
  - $w[i, i-1] = 0$ for $1 \leq i \leq n$.
  - $w[i, j] = w[i, j\text{-}1] + p_j$ for $1 \leq i \leq j \leq n$.

Example

	k_1	k_2	k_3	k_4
key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

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					

Let us compute e(1, 2):

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

$$e(1,2) = \min \begin{Bmatrix} r = 1: & e(1,0) + e(2,2) + w(1,2) & = 0 + 0.2 + 0.3 = 0.5 \\ r = 2: & e(1,1) + e(3,2) + w(1,2) & = 0.1 + 0 + 0.3 = 0.4 \end{Bmatrix}$$

$= 0.4.$

$$w(i,j) = \sum_{l=i}^{j} p_l$$

Example

Let us compute e(1, 2):

$$e(1,2) = \min \begin{Bmatrix} r = 1: & e(1,0) + e(2,2) + w(1,2) & = 0 + 0.2 + 0.3 = 0.5 \\ r = 2: & e(1,1) + e(3,2) + w(1,2) & = 0.1 + 0 + 0.3 = 0.4 \end{Bmatrix}$$
$$= 0.4.$$

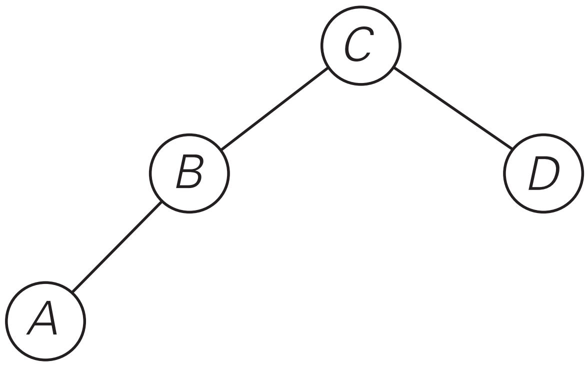
Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

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					



Pseudo-code

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

OPTIMAL-BST(*p*, *n*)

```
for i ← 1 to n + 1
    do e[i, i–1] ← 0
       w[i, i–1] ← 0
for l ← 1 to n
    do for i ← 1 to n–l + 1
        do j ← i + l–1
           e[i, j ] ← ∞
           w[i, j ] ← w[i, j–1] + p_j
           for r ← i to j
               do t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
                  if t < e[i, j ]
                     then e[i, j ] ← t
                          root[i, j ] ← r
return e and root
```

Consider all trees with *l* keys.

Fix the first key.

Fix the last key.

For each possible root

Determine the root of the optimal (sub)tree.

Time: $O(n^3)$