Chapter 14: Dynamic Programming III

Subsequence of a String

- Let $S = s_1 s_2 \ldots s_m$ be a string of length m
- Any string of the form

$$s_{i_1} \, s_{i_2} \, \ldots \, s_{i_k}$$

with $i_1 < i_2 < \ldots < i_k$ is a subsequence of S

- E.g., if S = farmers

→ fame, arm, mrs, farmers, are some of the subsequences of S

Longest Common Subsequence

- Let S and T be two strings
- If a string is both
  - a subsequence of S and
  - a subsequence of T,

  it is a common subsequence of S and T

- In addition, if it is the longest possible one, it is a longest common subsequence

Longest Common Subsequence

- e.g.,

  S = algorithms

  T = logarithms

- Then, aim, lots, ohms, grit, are some of the common subsequences of S and T
- Longest common subsequences:

  lorithms , lgrithms

Longest Common Subsequence

- Let $S = s_1 s_2 \ldots s_m$ be a string of length m
- Let $T = t_1 t_2 \ldots t_n$ be a string of length n

✓ Can we quickly find a longest common subsequence (LCS) of S and T ?

Optimal Substructure

- Let $X = x_1 x_2 \ldots x_k$ be an LCS of
  $S_{1,i} = s_1 s_2 \ldots s_{i-1} s_i$ and $T_{1,j} = t_1 t_2 \ldots t_{j-1} t_j$
- Lemma:
  - If $s_i = t_j$, then $x_k = s_i = t_j$, and $x_1 x_2 \ldots x_{k-1}$ must be the LCS of $S_{1,i-1}$ and $T_{1,j-1}$
  - If $s_i \neq t_j$, then X must either be
    (i) an LCS of $S_{1,i}$ and $T_{1,j-1}$, or
    (ii) an LCS of $S_{1,i-1}$ and $T_{1,j}$

Optimal Substructure

- Let $len_{i,j}$ = length of the LCS of $S_{1,i}$ and $T_{1,j}$

- Lemma: For any $i, j \geq 1$,
  - if $s_i = t_j$, $len_{i,j} = len_{i-1,j-1} + 1$
  - if $s_i \neq t_j$, $len_{i,j} = \max \{ len_{i,j-1} , len_{i-1,j} \}$

Length of LCS

Define a function Compute_L(i,j) as follows:

```
Compute_L(i, j)   /* Finding len_{i,j} */
  1. if (i == 0  or  j == 0 )  return 0;    /* base case */
  2. if (s_i == t_j)
        return Compute_L(i-1,j-1) + 1;
  3. else
        return max {Compute_L(i-1,j), Compute_L(i,j-1)};
```

Compute_L(m, n) runs in $O(2^{m+n})$ time

Overlapping Subproblems

- To speed up, we can see that :
- To Compute_L(i, j-1) and Compute_L(i-1, j), has a common subproblem: Compute_L(i-1, j-1)
- In fact, in our recursive algorithm, there are many redundant omputations !
- Question: Can we avoid it ?

Bottom-Up Approach

- Let us create a 2D table L to store all $len_{i,j}$ values once they are computed
- BottomUp_L( ) /* Finding min #operations */

1. For all i and j, set L[i,0] = L[0, j] = 0;
2. for (i = 1,2,…, m)

   Compute L[i,j] for all j;

   /* Based on L[i-1,j-1], L[i-1,j], L[i,j-1] */
3. return L[m, n] ;

Running Time = $\Theta(mn)$

Example Run: After Step 1

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0									
I	0									
R	0									
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

$m = 9$

$n = 9$

Example Run: Step 2, i = 1

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1	1	1	1	1	1	1	1	1
I	0									
R	0									
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

Example Run: Step 2, i = 2

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1	1	1	1	1	1	1	1	1
I	0	1	1	1	1	2	2	2	2	2
R	0									
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

Example Run: Step 2, i = 3

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1	1	1	1	1	1	1	1	1
I	0	1	1	1	1	2	2	2	2	2
R	0	1	1	2	2	2	2	2	3	3
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

Example Run: Step 2, i = 4

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1	1	1	1	1	1	1	1	1
I	0	1	1	1	1	2	2	2	2	2
R	0	1	1	2	2	2	2	2	3	3
T	0	1	1	2	2	2	3	3	3	3
Y	0									
R	0									
O	0									
O	0									
M	0									

Example Run: After Step 2

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1	1	1	1	1	1	1	1	1
I	0	1	1	1	1	2	2	2	2	2
R	0	1	1	2	2	2	2	2	3	3
T	0	1	1	2	2	2	3	3	3	3
Y	0	1	1	2	2	2	3	3	3	4
R	0	1	1	2	2	2	3	3	4	4
O	0	1	2	2	2	2	3	4	4	4
O	0	1	2	2	2	2	3	4	4	4
M	0	1	2	2	3	3	3	4	4	4

Extra information to obtain an LCS

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1↖	1←	1←	1←	1←	1←	1←	1←	1←
I	0									
R	0									
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

Extra Info: Step 2, i = 2

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1↖	1←	1←	1←	1←	1←	1←	1←	1←
I	0	1↑	1↑	1↑	1↑	2↖	2←	2←	2←	2←
R	0									
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

Extra Info: Step 2, i = 3

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1↖	1←	1←	1←	1←	1←	1←	1←	1←
I	0	1↑	1↑	1↑	1↑	2↖	2←	2←	2←	2←
R	0	1↑	1↑	2↖	2←	2↑	2↑	2↑	3↖	3←
T	0									
Y	0									
R	0									
O	0									
O	0									
M	0									

Extra Info: After Step 2

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1↖	1←	1←	1←	1←	1←	1←	1←	1←
I	0	1↑	1↑	1↑	1↑	2↖	2←	2←	2←	2←
R	0	1↑	1↑	2↖	2←	2↑	2↑	2↑	3↖	3←
T	0	1↑	1↑	2↑	2↑	2↑	3↖	3←	3←	3←
Y	0	1↑	1↑	2↑	2↑	2↑	3↑	3←	3←	4↖
R	0	1↑	1↑	2↖	2↑	2↑	3↑	3↑	4↖	4↑
O	0	1↑	2↖	2↑	2↑	2↑	3↑	4↖	4↑	4↑
O	0	1↑	2↖	2↑	2↑	2↑	3↑	4↖	4↑	4↑
M	0	1↑	2↑	2↑	3↖	3←	3←	4↑	4↑	4↑

LCS obtained by tracing from L[m,n]

		D	O	R	M	I	T	O	R	Y
	0	0	0	0	0	0	0	0	0	0
D	0	1↖	1←	1←	1←	1←	1←	1←	1←	1←
I	0	1↑	1↑	1↑	1↑	2↖	2←	2←	2←	2←
R	0	1↑	1↑	2↖	2←	2↑	2↑	2↑	3↖	3←
T	0	1↑	1↑	2↑	2↑	2↑	3↖	3←	3←	3←
Y	0	1↑	1↑	2↑	2↑	2↑	3↑	3←	3←	4↖
R	0	1↑	1↑	2↑	2↑	2↑	3↑	3↑	4↖	4↑
O	0	1↑	2↖	2↑	2↑	2↑	3↑	4↖	4↑	4↑
O	0	1↑	2↖	2↑	2↑	2↑	3↑	4↖	4↑	4↑
M	0	1↑	2↑	2↑	3↖	3←	3←	4↑	4↑	4↑

Computing the length of an LCS

LCS_LENGTH(X,Y)

$m \leftarrow X.length$
$n \leftarrow Y.length$
for $i \leftarrow 1$ **to** m
    $c[i, 0] \leftarrow 0$
for $j \leftarrow 1$ **to** n
    $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
    for $j \leftarrow 1$ **to** n

Computing the length of an LCS

if $x_i == y_j$
    $c[i, j] \leftarrow c[i\text{-}1, j\text{-}1]+1$
    $b[i, j] \leftarrow$ "$\nwarrow$"
elseif $c[i–1, j] \geq c[i, j\text{-}1]$ / * We can use > instead of ≥ */
    $c[i, j] \leftarrow c[i\text{-}1, j]$
    $b[i, j] \leftarrow$ "$\uparrow$"
else $c[i, j] \leftarrow c[i, j\text{-}1]$
    $b[i, j] \leftarrow$ "$\leftarrow$"
return c and b

Remarks

- Again, a slight change in the algorithm allows us to obtain a particular LCS
- Also, we can make minor changes to the recursive algorithm and obtain a memoized version (whose running time is $O(mn)$)

Writing a Translation Program

- In real life, different words may be searched with different frequencies

  e.g., apple may be more often than pizza

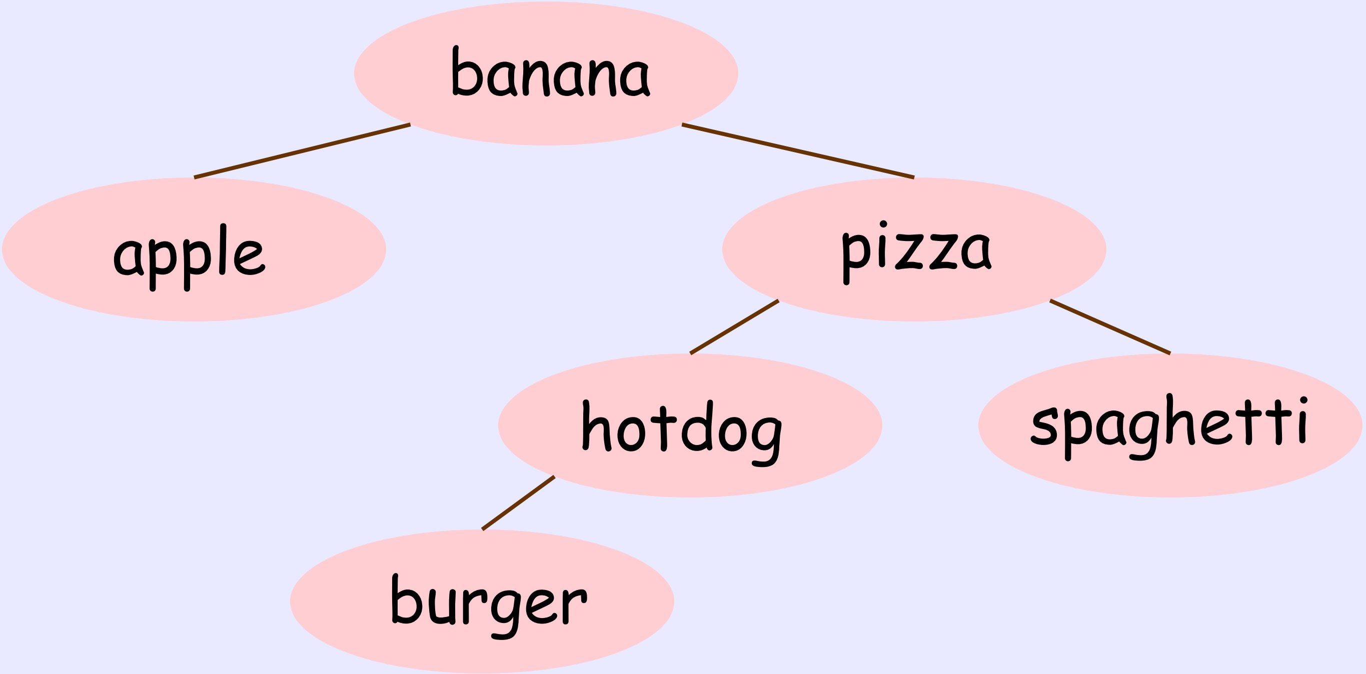
- Also, there may be different frequencies for the unsuccessful searches

  e.g., we may unlikely search for a word in the range (hotdog, pizza)

- Suppose your friend in Google gives you the probabilities of what a search will be:

< apple	0.01	= hotdog	0.02
= apple	0.21	(hotdog, pizza)	0.04
(apple, banana)	0.10	= pizza	0.04
= banana	0.18	(pizza, spaghetti)	0.11
(banana, burger)	0.05	= spaghetti	0.07
= burger	0.01	> spaghetti	0.04
(burger, hotdog)	0.12		

- Given these probabilities, we may want words that are searched more frequently to be nearer the root of the search tree

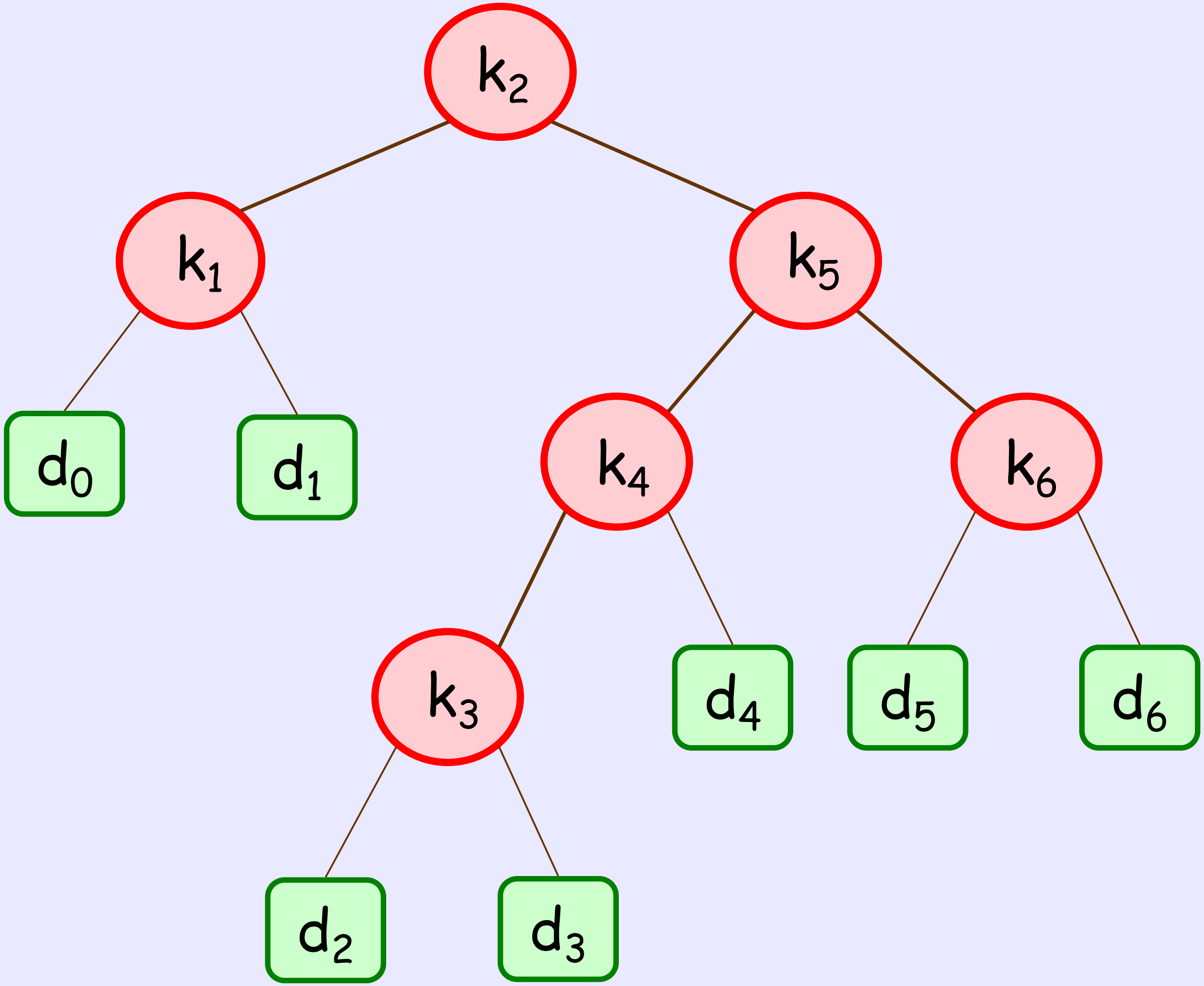


This tree has better expected performance

Expected Search Time

- To handle unsuccessful searches, we can modify the search tree slightly (by adding dummy leaves), and define the expected search time as follows:
- Let $k_1 < k_2 < \ldots < k_n$ denote the n keys, which correspond to the internal nodes
- Let $d_0 < d_1 < d_2 < \ldots < d_n$ be dummy keys for ranges of the unsuccessful search

  ➔ dummy keys correspond to leaves



Modified Search tree for six keys

Search Time

- Lemma: Based on the modified search tree:
  - ✓ when we search for a word k_i, search time = node-depth(k_i)
  - ✓ when we search for a word in range d_j, search time = node-depth(d_j)

Expected Search Time

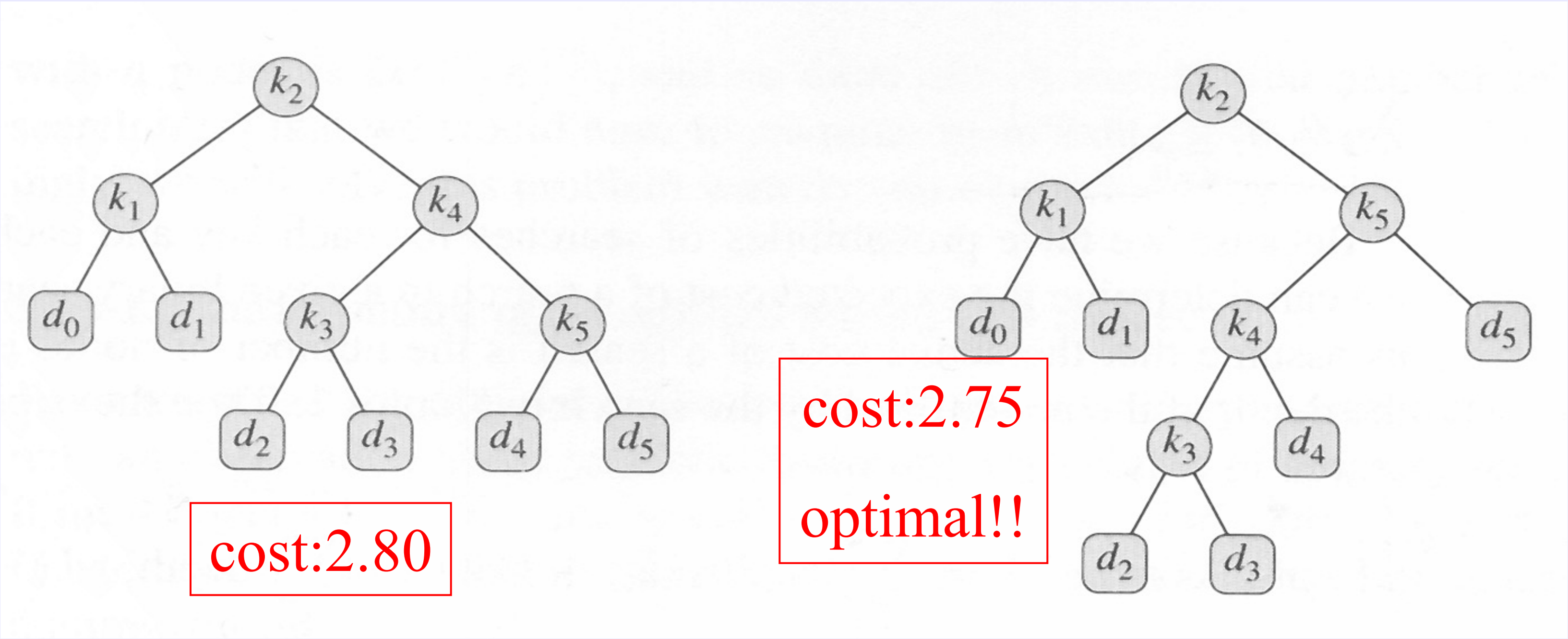
- Let p_i = Pr(k_i is searched)
- Let q_j = Pr(word in d_j is searched)

So, $$\sum_i p_i + \sum_j q_j = 1$$

- Expected search time

$$= \sum_i p_i \text{ node-depth}(k_i) + \sum_j q_j \text{ node-depth}(d_j)$$

Optimal Binary search trees



0.1 + (0.15+ 0.1) x 2 +(0.05 + 0.1 + 0.05 + 0.2) x 3 + (0.05 + 0.05 + 0.05 + 0.1) x 4 = 0.1 + 0.5 + 1.2 + 1.0 = 2.8

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal Binary Search Tree

- Question:

  Given the probabilities p_i and q_j,
  can we construct a binary search tree whose expected search time is minimized?

Such a search tree is called an
Optimal Binary Search Tree

Optimal Substructure

- Let T = optimal BST for the keys
  $(k_i, k_{i+1}, \dots, k_j; d_{i-1}, d_i, \dots, d_j)$.
- Let L and R be its left and right subtrees.

- Lemma: Suppose k_r is the root of T. Then,

✓ L must be an optimal BST for the keys
  $(k_i, k_{i+1}, \dots, k_{r-1}; d_{i-1}, d_i, \dots, d_{r-1})$

✓ R must be an optimal BST for the keys
  $(k_{r+1}, k_{r+2}, \dots, k_j; d_r, d_{r+1}, \dots, d_j)$

Optimal Substructure

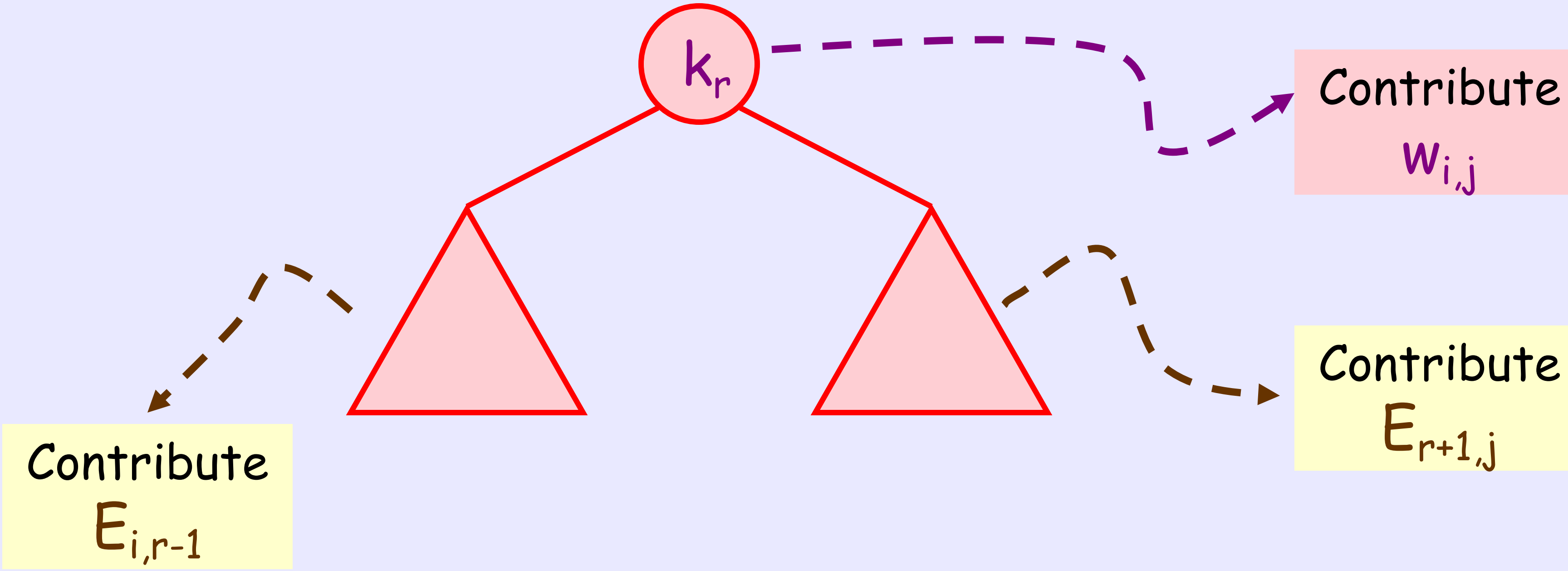
- Let $E_{i,j}$ = expected time spent with the keys $(k_i, k_{i+1}, \ldots, k_j; d_{i-1}, d_i, \ldots, d_j)$ in optimal BST
- Note that, $E_{i,i-1} = (d_{i-1})$ and $E_{j+1,j} = (d_j)$
- Let $w_{i,j} = \Sigma_{s=i \text{ to } j}\, p_s + \Sigma_{t=i-1 \text{ to } j}\, q_t$

  = sum of the probabilities of keys $(k_i, k_{i+1}, \ldots, k_j; d_{i-1}, d_i, \ldots, d_j)$

Optimal Substructure

- Lemma: For any $j \geq i$,

$$E_{i,j} = \min_{i \leq r \leq j} \{ E_{i,r-1} + E_{r+1,j} + w_{i,j} \}$$



Optimal Binary Search Tree

- Define a function Compute_E(i,j) as follows:

Compute_E(i, j) /* Finding $e_{i,j}$ */

1. if (i == j+1) return q_j; /* Exp time with dummy key d_j */
2. min = ∞;
3. for (r = i, i+1, …, j) {

    g = Compute_E(i,r-1) + Compute_E(r+1,j) + $w_{i,j}$;

    if (g < min) min = g;

   }
4. return min ;

Optimal Binary Search Tree

- Question: We want to get Compute_E(1,n) What is its running time?
- Similar to Matrix-Chain Multiplication, the recursive function runs in $\Omega(3^n)$ time
- In fact, it will examine at most once for all possible binary search tree → Running time = $O(C(2n-2,n-1)/n)$

Catalan Number

Overlapping Subproblems

- Here, we can see that :

  To Compute_E(i,j) and Compute_E(i,j+1)
  there are many COMMON subproblems

  Compute_E(i,i-1), Compute_E(i,i),
  Compute_E(i,i+1), ..., Compute_E(i,j-1)

- So, in our recursive algorithm, there are many redundant computations !
- Question: Can we avoid it ?

```
for (r = i, i+1, ..., j) {
      g = Compute_E(i,r-1) + Compute_E(r+1,j) + wi,j ;
```

Bottom-Up Approach

- Let us create a 2D table E to store all $E_{i,j}$ values once they are computed
- Let us also create a 2D table W to store all $w_{i,j} = w_{i,j-1} + p_j + q_j$
- We first compute all entries in W.
- Next, we compute $E_{i,j}$ for $j-i = 0,1,2,\dots,n-1$

$$w_{i,j} = \Sigma_{s=i \text{ to } j}\, p_s + \Sigma_{t=i-1 \text{ to } j}\, q_t$$

Bottom-Up Approach

BottomUp_E(i,j) /* Finding min #operations */

1. Fill all entries of W
2. for i = 0, 1, 2, …, n, set $E[i+1,i] = q_i$;
3. for (length = 0,1,2,…, n-1)

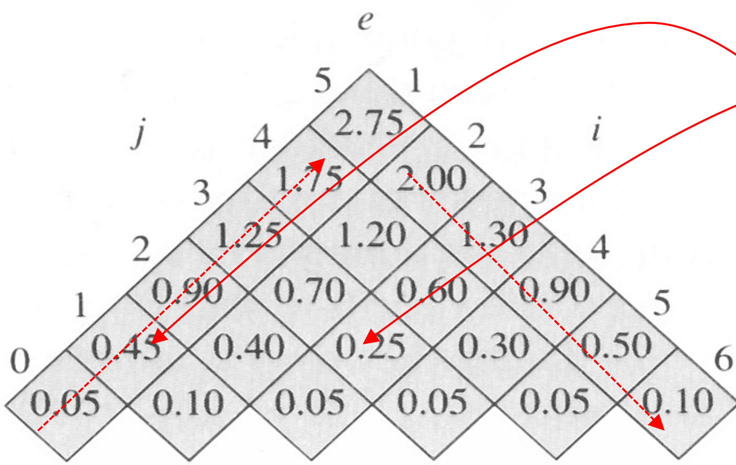
   for (i = 1, 2, ..n-length)

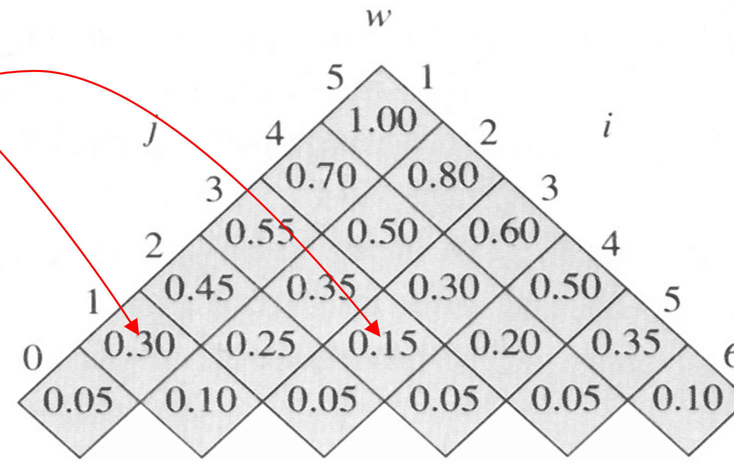
   Compute E[i,i+length];

   /* From W and E[x,y] with |x-y| < length */
4. return E[1,n] ;

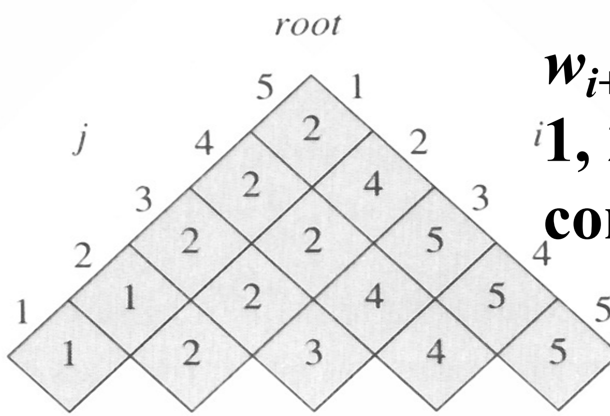
Running Time = $\Theta(n^3)$

The table e[i,j], w[i,j], and root[i,j] compute by OPTIMAL-BST on the key distribution.





$e(i+1, i) = q_i$ for $i = 0, 1, \ldots, 5$ boundary condition



$w_{i+1, i} = q_i$ for $i = 0, 1, 2, \ldots, 5$ boundary condition

$w_{i,j} = w_{i,j-1} + p_j + q_j$

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

For any $j \geq i$,

$e_{i,j} = min_{i \leq r \leq j} \{ e_{i, r-1} + e_{r+1, j} + w_{i,j} \}$

For Example

- e(1, 1) = min (e(1, 0) + e(2, 1) + w1,1) = 0.05 ++ 0.1 + 0.3 = 0.45
- e(2, 2) = min (e(2, 1) + e(3, 2) + w(2,2) = 0.1 + 0.25 + 0.05 = 0.4
- e(1,2) = min (e(1, 0) + e(2, 2) + w(1, 2), e(1, 1) + e (3, 2) + w(1, 2)) = min (0.05 + 0.4 + 0.45 = 0.9, 0.45 + 0.05 + 0.45 = 0.95) = 0.9
- …
- e(1, 5) = min (e(1, 0) + e(2, 5), e(1, 1) + e(3, 5), e(1, 2) + e(4, 5), e(1, 3) + e(5, 5), e(1, 4) + e (6, 5)) + w(1,5)

Remarks

- Again, a slight change in the algorithm allows us to get the exact structure of the optimal binary search tree

- Also, we can make minor changes to the recursive algorithm and obtain a memoized version (whose running time is $O(n^3)$)

Practice at home

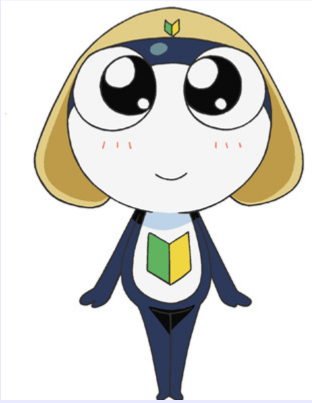
- Exercise 14.4-1, 14.4-3, 14.4-4
- Exercises 14.5-2, 14.5-3

Sharing Gold Coins

Five lucky pirates has discovered a treasure box with 1000 gold coins …





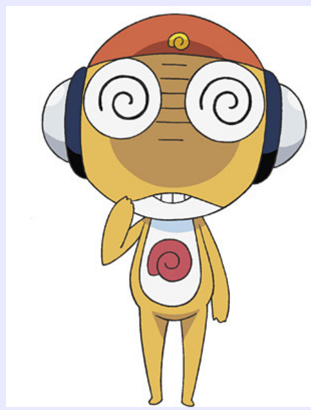


Sharing Gold Coins

There are rankings among the pirates:

1

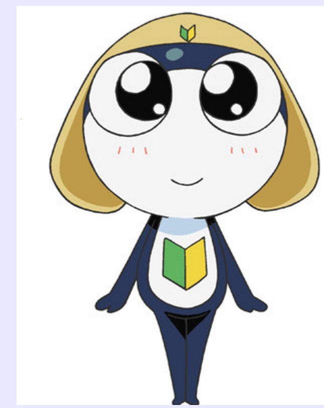
2



3

4

5



… and they decide to share the gold coins in the following way:

Sharing Gold Coins

First, Rank-1 pirate proposes how to share the coins…

- If at least half of them agree, go with the proposal
- Else, Rank-1 pirate is out of the game

Hehe, I am going to make the first proposal … but there is a danger that I cannot share any coins

Sharing Gold Coins

In general, if Rank-1, Rank-2, …, Rank-k pirates are out, then Rank-(k+1) pirate proposes how to share the coins…

- If at least half of the remaining agree, go with the proposal
- Else, Rank-(k+1) pirate is out of the game

Question: If all the pirates are smart, who will get the most coin? Why?

Sharing Gold Coins

If Rank-1 pirate is out, then Rank-2 pirate proposes how to share the coins…

- If at least half of the remaining agree, go with the proposal
- Else, Rank-2 pirate is out of the game

Hehe, I get a chance to propose if Rank-1 pirate is out of the game

Brainstorm

- 外遇村裡有50對夫妻，丈夫全都有外遇。妻子都知道所有有外遇的男人，就是不知道自己的先生有外遇，妻子之間彼此也不會互相交換訊息。村子有一個規定，妻子若能證明自己的先生外遇，就必需在當天晚上12:00殺死他。有一天，公認不會說謊的皇后來到外遇村，宣布至少有一位丈夫不忠。結果會發生甚麼事？