CS4311
Design and Analysis of Algorithms

Lecture 11: Dynamic Programming III
About this lecture

• We will see more examples today
Writing a Translation Program

• Suppose we want to design a program to translate English texts on food to Chinese

• First problem to solve:

  Given an English word, can we quickly search for its Chinese equivalent?

• E.g., Apple $\rightarrow$ 蘋果, Banana $\rightarrow$ 香蕉, Pizza $\rightarrow$ 比薩, Burger $\rightarrow$ 漢堡, Hotdog $\rightarrow$ 熱狗, Spaghetti $\rightarrow$ 意麺
Writing a Translation Program

• However, some English words may not be common to have a Chinese equivalent
  • In this case, we report not found

• E.g., Biryani (a South Asian dish)
  Burrito (a common Mexican food)
  Jambalaya (a famous Louisiana dish)
  Okonomiyaki (a kind of Japanese pizza)
Writing a Translation Program

• Let $n = \#$ of English words in our database with Chinese equivalent

Solution 1: Hashing
  • Good, but need a good hash function

Solution 2: Balanced Binary Search Tree
  • worst-case $O(\log n)$ time per query
Balanced Binary Search Tree

Keys = words in the database
Writing a Translation Program

• In real life, different words do not appear with the same frequencies
  E.g., apple may be more often than pizza

• Also, there may be different frequencies for the unsuccessful searches
  E.g., we may unluckily search for a word in the range (hotdog, pizza) more often than in the range (spaghetti, $+\infty$)
• Suppose your friend in Google gives you probabilities of what a search will be:

<table>
<thead>
<tr>
<th></th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; apple</td>
<td>0.01</td>
</tr>
<tr>
<td>= apple</td>
<td>0.21</td>
</tr>
<tr>
<td>(apple, banana)</td>
<td>0.10</td>
</tr>
<tr>
<td>= banana</td>
<td>0.18</td>
</tr>
<tr>
<td>(banana, burger)</td>
<td>0.05</td>
</tr>
<tr>
<td>= burger</td>
<td>0.01</td>
</tr>
<tr>
<td>(burger, hotdog)</td>
<td>0.12</td>
</tr>
<tr>
<td>= hotdog</td>
<td>0.02</td>
</tr>
<tr>
<td>(hotdog, pizza)</td>
<td>0.04</td>
</tr>
<tr>
<td>= pizza</td>
<td>0.04</td>
</tr>
<tr>
<td>(pizza, spaghetti)</td>
<td>0.11</td>
</tr>
<tr>
<td>= spaghetti</td>
<td>0.07</td>
</tr>
<tr>
<td>&gt; spaghetti</td>
<td>0.04</td>
</tr>
</tbody>
</table>
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

• We can modify the search tree slightly (by adding dummy leaves), and define the expected search time as follows:

• Let $k_1 < k_2 < ... < k_n$ denote the $n$ keys, which correspond to the internal nodes

• Let $d_0 < d_1 < d_2 < ... < d_n$ be dummy keys for ranges of the unsuccessful search

⇒ dummy keys correspond to leaves
Search tree of Page 9 after modification
Expected Search Time

Lemma: Based on the modified search tree:

- when we search for a word $k_i$, 
  search time = $\text{depth}(k_i) + 1$

- when we search for a word in range $d_j$, 
  search time = $\text{depth}(d_j) + 1$
Expected Search Time

• Let \( p_i = \text{Pr}( k_i \text{ is searched} ) \)
• Let \( q_j = \text{Pr}( \text{word in } d_j \text{ is searched} ) \)

\[ \sum_i p_i + \sum_j q_j = 1 \]

Then, expected search time

\[ = \sum_i p_i (\text{depth}(k_i) + 1) + \sum_j q_j (\text{depth}(d_j) + 1) \]

\[ = 1 + \sum_i p_i \text{ depth}(k_i) + \sum_j q_j \text{ depth}(d_j) \]
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 $T = \text{optimal BST for the keys}$

$(k_i, k_{i+1}, \ldots, k_j; d_{i-1}, d_i, \ldots, d_j)$. 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
  $(k_i, k_{i+1}, \ldots, k_{r-1}; d_{i-1}, d_i, \ldots, d_{r-1})$
- $R$ must be an optimal BST for the keys
  $(k_{r+1}, k_{r+2}, \ldots, k_j; d_r, d_{r+1}, \ldots, d_j)$
Optimal Substructure

Let $e_{i,j}$ denote expected search time within an optimal BST for the keys 

$$( k_i, k_{i+1}, \ldots, k_j; d_{i-1}, d_i, \ldots, d_j )$$

$\Rightarrow e_{i,i-1} = Pr(d_{i-1}) \times 1 = q_{i-1}$

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

$$= \sum_{s=i}^{j} p_s + \sum_{t=i-1}^{j} q_t$$

$\Rightarrow w_{i,i-1} = Pr(d_{i-1}) = q_{i-1}$
Lemma: For any $j \geq i$, 

$$e_{i,j} = \min_r \{ p_r + e_{i,r-1} + w_{i,r-1} + e_{r+1,j} + w_{r+1,j} \}$$

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

Contribute $w_{i,r-1}$

Contribute $p_r$

Contribute $e_{i,r-1}$

Contribute $e_{r+1,j}$

Contribute $w_{r+1,j}$
Define a function $\text{Compute}_E(i, j)$ as follows:

$\text{Compute}_E(i, j) \quad /* \text{Finding } e_{i,j} */$

1. if ($i == j+1$) return $q_j$;
2. $m = \infty$;
3. for ($r = i, i+1, ..., j$) {
   
   $g = \text{Compute}_E(i, r-1) + \text{Compute}_E(r+1, j) + w_{i,j}$;
   
   if ($g < m$) $m = g$;
  }
4. return $m$ ;
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
• Also it will examine at most once for all possible binary search tree

\[ \text{Running time} = O(C(2n-2,n-1)/n) \]
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,j-1)

So, in our recursive algorithm, there are many redundant computations!

Question: Can we avoid it?
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 $w_{i,j}$

We first compute all entries in $W$.
Next, we compute $e_{i,j}$ for $j-i = 0,1,2,...,n-1$
Bottom-Up Approach

**BottomUp_E() /* Finding min #operations */**

1. Fill all entries of $W$
2. for $j = 1, 2, ..., n$, set $E[j+1, j] = q_j$
3. for (length = 0, 1, 2, ..., $n-1$)
   Compute $E[i, i+\text{length}]$ for all $i$;
   // From $W$ and $E[x, y]$ with $|x-y| < \text{length}$
4. return $E[1, n]$;

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