Advance Analysis of Algorithms

- Dr. Qaiser Abbas
- Computer Science & IT
- University of Sargodha
- qaiser.abbas@uos.edu.pk

21-Dec-20

LCS Problem

- Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the ***same relative order***, but **not necessarily contiguous**.

For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of **“abcdefg”**. So, a string of length **n has 2^n** different possible subsequences.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files) and has applications in bioinformatics.

Examples

- LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
- LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.
- The naive (simple) solution for this problem is to **generate all subsequences** of both given sequences and **find the longest** matching subsequence. This solution is **exponential** in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP).

1) Optimal Substructure

- Let the input sequences be **X[0..m-1]** and **Y[0..n-1]** of lengths m and n respectively and let **L(X[0..m-1], Y[0..n-1])** be the length of LCS of the two sequences X and Y.
- Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).
  - If the last characters of both sequences **match** (if X[m-1] == Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])
  - If the last characters of both sequences **do not match** (if X[m-1] != Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Examples

1. Consider the input strings "**AGGTAB**" and "**GXTXAYB**". Last characters match for the strings. So, length of LCS can be written as:
   L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")
2. Consider the input strings "**ABCDGH**" and "**AEDFHR**". Last characters do not match for the strings. So, length of LCS can be written as:
   L("ABCDGH", "AEDFHR") = MAX (L("ABCDG", "AEDFH**R**"), L("ABCDG**H**", "AEDFH"))

   – So, the LCS problem has optimal substructure property as the main problem **can be solved using solutions to subproblems.**

2) Overlapping Subproblems

- Recursive implementation of the LCS problem.

```
/* A Naive recursive implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if (X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  getchar();
  return 0;
}
```

2) Overlapping Subproblems

- Time complexity of the above naive recursive approach is **O(2^n)** in worst case and worst case happens when all characters of X and Y **mismatch** i.e., length of LCS is 0.
- Considering the previous implementation, following is a **partial recursion tree** for input strings "**AXYT**" and "**AYZX**"

2) Overlapping Subproblems

- In the above partial recursion tree, **lcs("AXY", "AYZ") is being solved twice**. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So, this problem has **Overlapping Substructure property** and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

```
                         lcs("AXYT", "AYZX")
                       /                 \
         lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
         /            \                  /               \
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

LCS Algorithm

- Time Complexity of the above implementation is **O(mn)** which is much better than the worst case time complexity of Naive Recursive implementation.

	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

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Exercise

- Algorithm discussed returns only length of LCS. Please augment the algorithm for printing the LCS.

Assignment # 2

15.4-1

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

15.4-3

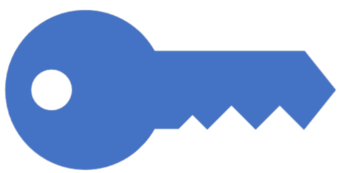
Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

15.4-4

Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

Break

Optimal Binary Search Trees (BSTs)



Given a sorted array ***keys[0.. n-1]*** of search keys and an array ***freq[0.. n-1]*** of frequency counts, where *freq[i]* is the number of searches to *keys[i]*.

Construct a binary search tree of all keys such that the total cost of all the searches would be as small as possible.

Cost of BSTs

```
10                      12
  \                    /
   12                10
  I                    II
```

- Let us first define the cost of a BST. The **cost of a BST node is level of that node multiplied by its frequency**. Let level of root is 1.
- **Example 1**
  - Input: **keys[] = {10, 12}, freq[] = {34, 50}**
  - There can be the following two possible BSTs.
  - Frequency of searches of 10 and 12 are 34 and 50 respectively.
  - The cost of tree I is **34*1 + 50*2 = 134**
  - The cost of tree II is **50*1 + 34*2 = 118**

Cost of BSTs

- **Example 2**
  - Input: **keys[] = {10, 12, 20}, freq[] = {34, 8, 50}**
  - There can be following possible BSTs.

```
10                12                  20       10                20
  \              /  \                /           \              /
   12          10    20            12             20          10
     \                            /              /              \
      20                        10             12                12
 I               II              III           IV                V
```

  - Among all BSTs, **cost of the fifth BST is minimum**.
  - Cost of the fifth BST is **1*50 + 2*34 + 3*8 = 142**

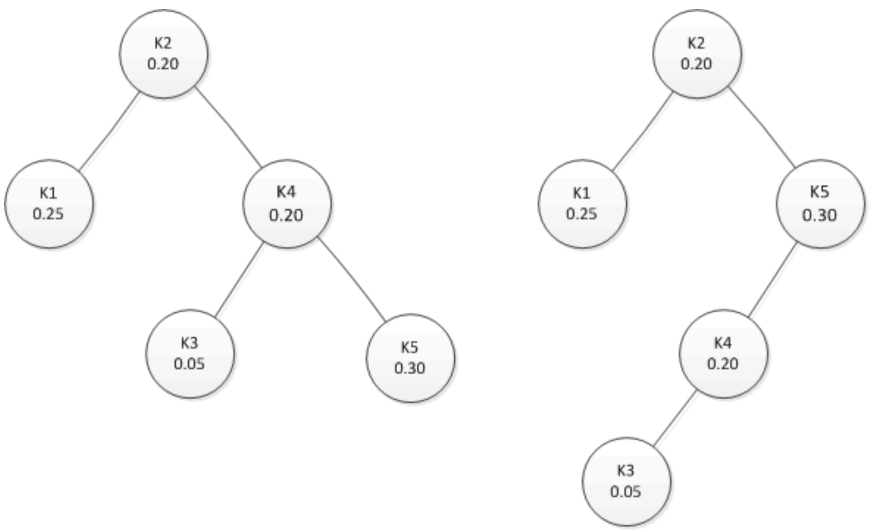
Cost of BSTs

- Problem:
  - Sorted set of keys $k_1, k_2, \ldots, k_n$
  - Key probabilities: $p_1, p_2, \ldots, p_n$
  - What tree structure has lowest expected cost?
  - Cost of searching for node i: $\text{cost}(k_i) = \text{depth}(k_i) + 1$

$$\begin{aligned}
\text{Expected Cost of tree} &= \sum_{i=1}^{n} \text{cost}(k_i) p_i \\
&= \sum_{i=1}^{n} (\text{depth}(k_i) + 1) p_i \\
&= \sum_{i=1}^{n} \text{depth}(k_i) p_i + \sum_{i=1}^{n} p_i \\
&= \left(\sum_{i=1}^{n} \text{depth}(k_i) p_i \right) + 1
\end{aligned}$$

Cost of BSTs

- **Example 3:**
- Probability table (p_i is the probability of key k_i:



i	1	2	3	4	5
Ki	K1	K2	K3	K4	K5
Pi	0.25	0.20	0.05	0.20	0.30

Cost of BSTs

- Given: $k1<k2<k3<k4<k5$
- **Tree 1:**
- *k*2/[*k*1,*k*4]/[*nil*,*nil*],[*k*3,*k*5]
- cost = 0(0.20) + 1(0.25+0.20) +2(0.05+0.30) + 1 = 1.15 + 1
- **Tree 2:**
- *k*2/[*k*1,*k*5]/[*nil*,*nil*],[*k*4,*nil*]/[*nil*,*nil*],[*nil*,*nil*],[*k*3,*nil*],[*nil*,*nil*]
- cost = 0(0.20) + 1(0.25+0.30) +2(0.20) + 3(0.05) + 1 = 1.10 + 1

- Notice that a deeper tree has expected lower cost

Optimal Substructure

Add sum of frequencies from i to j (first part)

$$optCost(i,j) = \sum_{k=i}^{j} freq[k] + \min_{r=i}^{j}[optCost(i,r-1) + optCost(r+1,j)]$$

One by one try all nodes as root (r varies from i to j in second part) and recursively calculate optimal cost from i to r-1 and r+1 to j.

optCost(0, n-1) will give final optimal result.

Optimal Substructure

- Following is recursive implementation that simply follows the recursive structure mentioned.

```
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
   // Base cases
   if (j < i)      // If there are no elements in this subarray
     return 0;
   if (j == i)     // If there is one element in this subarray
     return freq[i];

   // Get sum of freq[i], freq[i+1], ... freq[j]
   int fsum = sum(freq, i, j);

   // Initialize minimum value
   int min = INT_MAX;

   // One by one consider all elements as root and recursively find cost
   // of the BST, compare the cost with min and update min if needed
   for (int r = i; r <= j; ++r)
   {
       int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
       if (cost < min)
          min = cost;
   }

   // Return minimum value
   return min + fsum;
}

// The main function that calculates minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
     // Here array keys[] is assumed to be sorted in increasing order.
     // If keys[] is not sorted, then add code to sort keys, and rearrange
     // freq[] accordingly.
     return optCost(freq, 0, n-1);
}
```

Optimal Substructure

```
// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

- Time complexity of this recursive approach is exponential.

Overlapping Subproblems

- Since same subproblems are called again, this problem has Overlapping Subproblems property.
- Optimal BST problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems,
- Re-computations of same subproblems can be avoided by constructing a temporary array cost[][] in bottom-up manner.

Dynamic Programming Solution

An auxiliary array cost[n][n] to store the solutions of subproblems and cost[0][n-1] will hold the final result.

All diagonal values must be filled first, then the values which lie on the line just above the diagonal.

In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values.

The idea used in the implementation is same as [Matrix Chain Multiplication problem](#).

Dynamic Programming Solution

```
// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates minimum cost of
   a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree that can be
       formed from keys[i] to keys[j].
       cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ... .
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and chain length L
            int j = i+L-1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
               // c = cost when keys[r] becomes root of this subtree
               int c = ((r > i)? cost[i][r-1]:0) +
                       ((r < j)? cost[r+1][j]:0) +
                       sum(freq, i, j);
               if (c < cost[i][j])
                  cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}
```

```
// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

The BST Notes

THE TIME COMPLEXITY OF THE DP SOLUTION IS O(N^4) WHICH CAN BE REDUCED TO O(N^3) BY PRE-CALCULATING SUM OF FREQUENCIES INSTEAD OF CALLING SUM() AGAIN AND AGAIN.

IN THIS SOLUTIONS, WE HAVE COMPUTED OPTIMAL COST ONLY WHICH CAN BE MODIFIED TO STORE THE STRUCTURE OF BSTs.

AUXILIARY ARRAY OF SIZE N CAN BE USED TO STORE THE STRUCTURE OF TREE USING THE VALUE OF ‘R’ IN THE INNERMOST LOOP.

Example Optimal BST

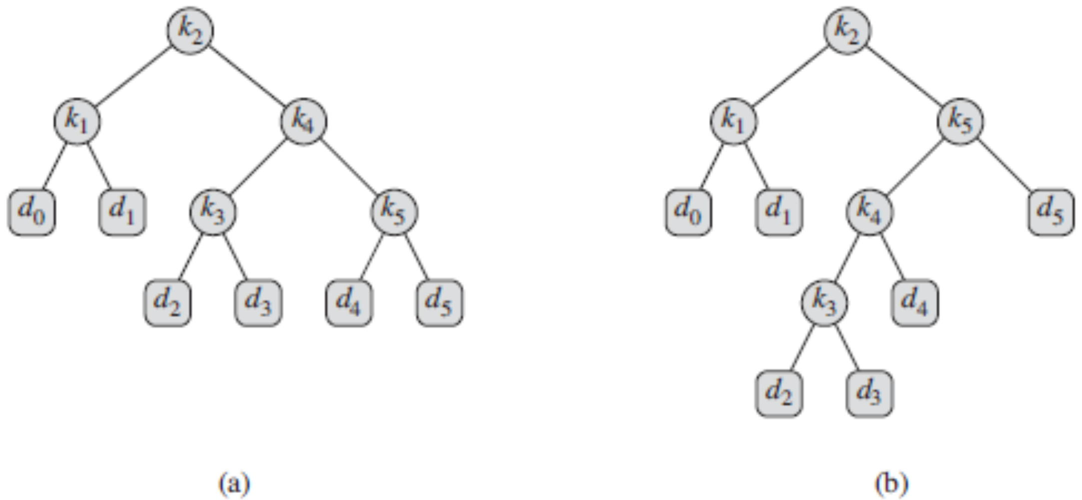


Figure 15.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

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

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.

Optimal BST Algorithm

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

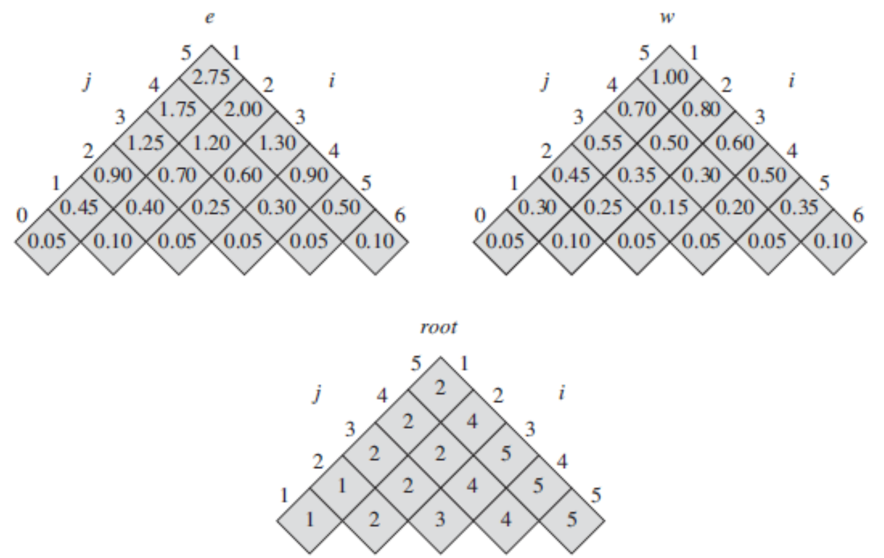


Figure 15.10 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 15.9. The tables are rotated so that the diagonals run horizontally.

Quiz

Quiz will be updated on the Google Class that you have to submit there within the deadline.

Assignment # 2

15.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST(*root*) which, given the table *root*, outputs the structure of an optimal binary search tree. For the example in Figure 15.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 15.9(b).

Assignment # 2

15.5-3

Suppose that instead of maintaining the table $w[i, j]$, we computed the value of $w(i, j)$ directly from equation (15.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

15.5-4 ★

Knuth [212] has shown that there are always roots of optimal subtrees such that $root[i, j-1] \leq root[i, j] \leq root[i+1, j]$ for all $1 \leq i < j \leq n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.