Homework 5

2019.03.28

Note: When the exercise asks you to "design an algorithm for…," it always means that "designs an EFFICIENT algorithm for … and ANALYZES your algorithm". You should keep this in mind when writing solutions.

1. **Exercises 15.5-2**

   Determine the cost and structure of an optimal binary search tree for a set of n = 7 keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

2. **Exercises 15.5-4**

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

3. **Problems 15-2: *Longest palindrome subsequence***

   A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, **civic**, **racecar**, and **aibohphobia** (fear of palindromes).

   Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input **character**, your algorithm should return **carac**. How long is the running time of your algorithm?

4. **Problems 15-5: *Edit distance***

   In order to transform one source string of text *x[1..m]* to a target string *y[1..n]*, we can perform various transformation operations. Our goal is, given *x* and *y*, to produce a series of transformations that change *x* to *y*. We use an array *z*—assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially, z is empty, and at termination, we should have *z[j]* = *y[j]* for *j* = 1, 2, …, n. We maintain current indices *i* into *x* and *j* into *z*, and the operations are allowed to alter *z* and these indices. Initially, *i* = *j* = 1. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have *i* = *m* + 1.

We may choose from among six transformation operations:

- **Copy** a character from *x* to *z* by setting *z[j]* = *x[i]* and then incrementing both *i* and *j*. This operation examines *x[i]*.
- **Replace** a character from x by another character c, by setting *z[j]* = c, and then incrementing both *i* and *j*. This operation examines *x[i]*.
- **Delete** a character from x by incrementing *i* but leaving *j* alone. This operation examines *x[i]*.
- **Insert** the character c into *z* by setting *z[j]* = c and then incrementing *j*, but leaving *i* alone. This operation examines no characters of *x*.
- **Twiddle** (i.e., exchange) the next two characters by copying them from *x* to *z* but in the opposite order; we do so by setting *z[j]* = *x[i + 1]* and *z[j + 1]* = *x[i]* and then setting *i* = *i* + 2 and *j* = *j* + 2. This operation examines *x[i]* and *x[i + 1]*.
- **Kill** the remainder of *x* by setting *i* = *m* + 1. This operation examines all characters in *x* that have not yet been examined. This operation, if performed, must be the final operation.

Given two sequences *x[1..m]* and *y[1..n]* and set of transformation-operation costs, the edit distance from *x* to *y* is the cost of the least expensive operation sequence that transforms *x* to *y*. Describe a dynamic-programming algorithm that finds the edit distance from *x[1..m]* to *y[1..n]* and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

5. Find out all LCS of《BADBCBA》and《ABACDBC》.

6. An algorithm to solve the LCS problem of two strings *X* and *Y* has space complexity *O*(|*X*||*Y*|) typically.

a. Design an algorithm to find the length of LCS of two string *X* and *Y* just using only 2·min(|*X*|, |*Y*|) cells for working space.

b. Design an algorithm to find the length of LCS of two string *X* and *Y* just using only 1 + min(|*X*|, |*Y*|) cells for working space.

7. String Alignment

Let σ be an alphabet set, β denote the blank character in σ, and a measure function *F*: $\sigma \times \sigma \rightarrow \boldsymbol{R}$. Where *F* is defined as followings, for any *x* and *y* in σ, *F*(*x*, *y*) ≤ 0 if $x \neq y$ and *F*(*x*, *y*) ≥ 0 if *x* = *y*; whereas *F*(β, β) = -∞. Given *X* and *Y* be two strings of σ^*, let *X*' and *Y*' denote two new strings made by inserting some β into *X* and *Y* respectively. The similarity of *X* and *Y* is defined by measuring the maximal value

of $\sum_{a_i \in X', b_i \in Y'} F(a_i, b_i)$ among all possible *X'* and *Y'*.

a. Design an algorithm to find the similarity of *X* and *Y*.
b. Design an algorithm that describe where the blank characters are inserted to get the similarity.