FIT2004 S1/2015: Lab questions for week 6

THIS PRAC IS ASSESSED!
(Weight: 6 Marks)

CLASS: This programming exercise has to be prepared in advance. Your 3 hour lab is mainly to iron out bugs, clarify any doubts you have with your demonstrator, and finalize your implementation. Once past the first hour of your assigned lab (or earlier if someone volunteers), your demonstrator will come to you based on a random draw and assess your work. This lab assignment should be implemented in Python programming language. Practical work is marked on the performance of your program **and also on your understanding of the program**. A *perfect* program with zero understanding implies you will get **zero** marks! "Forgetting" is not an acceptable explanation for lack of understanding. You must write all the code yourself, and may not use any external library routines that will invalidate the assessment of your learning. The usual input/output and other basic routines are exempted.
DEMONSTRATORS are not obliged to mark programs that do not compile or that crash. Time allowing, they will try to help in tracking down errors, but they are not required to mark programs in such a state, particularly those that do not compile. Therefore keep backup copies of working partial solutions.
OBJECTIVE: This exercise will test your understanding of lookup using *hashing* and solving problems using *dynamic programming*. You will be handling a simplified version of a useful, real-world problem. This problem instance has several learning elements that will enhance your ability to understand and implement a (relatively) nuanced algorithm. Repeated practice of thinking and writing programs will help you evolve into a proficient programmer.

Supporting Material

For this exercise use the supporting material uploaded on the Unit's Moodle site under Week 6

Background

Consider a set of n search keys $\{K_1, K_2, \ldots, K_n\}$ such that the following precedence holds $K_1 < K_2 < \ldots < K_n$. Using these keys in some random permutation, one could construct a *binary search tree* (BST). It is easy to see that, depending on the order of inserts while constructing the tree, the resultant BST may differ (sometimes radically) in its tree structure (and balance). In other words, there are many possible BSTs for the same given set of keys.

For example, when $n = 3$, with keys $K_1 = 1, K_2 = 2$, and $K_3 = 3$, there are *five* possible BSTs:

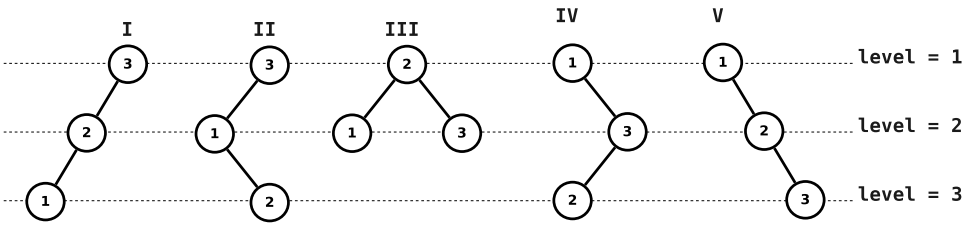

Figure 1: Five possible binary search trees for keys $K_1 = 1, K_2 = 2$ and $K_3 = 3$

In general, there are $\frac{^{2n}C_n}{n+1} \approx \frac{4^n}{\sqrt{\pi}n^{1.5}}$ possible binary search trees with n keys/nodes.

Let us assume some large text (file) is given to you, and it has n distinct words (keys) in it: $K_1 < K_2 < \ldots < K_n$. Clearly there is a distribution of number of occurrences of each distinct word in the given text, where some words may occur only once while others may occur many times. That is, each distinct word in the text has a certain *frequency* of occurrence. If, for each word, its frequency is divided by the *total* number of words in the text, we get the relative frequency which is the empirical *probability* of that word in the source text. We will use $\Pr(K_i)$ to denote the empirical probability of a key K_i.

For a binary search tree $T(K_1, \ldots, K_n)$ containing n keys (or nodes) with associated probabilities $\{\Pr(K_i), \ldots, \Pr(K_n)\}$, the **cost**, $\zeta(K_1, \ldots, K_n)$, of any possible BST over the set of keys is computed using the formula:

$$\zeta(K_1, \ldots, K_n) = \sum_{i=1}^{n} \Pr(K_i) \times d(\text{root}\,(T)\,, K_i)$$

where $\Pr(K_i)$ is the *probability* of each key K_i, and $d(\text{root}\,(T)\,, K_i)$ denotes the *level*[‡] at which the node with the key K_i appears in the BST. For example, the cost of each of the five trees shown in Figure 1 are:

BST I:	$3\Pr(K_1) + 2\Pr(K_2) + \Pr(K_3)$
BST II:	$2\Pr(K_1) + 3\Pr(K_2) + \Pr(K_3)$
BST III:	$2\Pr(K_1) + \Pr(K_2) + 2\Pr(K_3)$
BST IV:	$\Pr(K_1) + 3\Pr(K_2) + 2\Pr(K_3)$
BST V:	$\Pr(K_1) + 2\Pr(K_2) + 3\Pr(K_3)$

This sets up the problem of finding the **minimum cost** binary search tree for a given set of keys, $\{K_1 < K_2 < \ldots < K_n\}$ with corresponding probabilities $\{\Pr(K_1), \Pr(K_2), \ldots, \Pr(K_n)\}$.

Why do we care about the minimum cost binary search tree? The reason is plainly because such a BST *minimizes* the average *search time* when searching keys with respect to the given probabilities of the keys.

It turns out that finding the minimum cost binary search tree is solvable using a **dynamic programming** approach developed by Donald Knuth. The approach builds on the primary insight that *all subtrees of a minimum cost binary search tree are also BSTs of minimum cost.*

[‡]The *level* of a key in the tree is same as the *number of comparisons* required to lookup that key in the BST. If the key is at the root, the number of comparisons (or level) = 1; if it is a level below the root, then level = 2, and so on. Equivalently, level = one plus the depth of the key, where root is at depth = 0.

In other words, if the minimum cost binary search tree $T(K_1, \ldots, K_n)$ has as its **root** node as some $\text{root}(T) = K_r$, $1 \leq r \leq n$, then the equation

$$\zeta(K_1, \ldots, K_n) = \sum_{i=1}^{n} \Pr(K_i) \times d(\text{root}\,(T)\,, K_i)$$

can be expanded as

$$\zeta(K_1, \ldots, K_n) = \underbrace{\Pr(K_r)}_{\text{lookup cost of root node}} + \underbrace{\sum_{i=1}^{r-1} \Pr(K_i) \times d(\text{root}\,(T)\,, K_i)}_{\text{lookup cost of nodes in left subtree}} + \underbrace{\sum_{i=r+1}^{n} \Pr(K_i) \times d(\text{root}\,(T)\,, K_i)}_{\text{lookup cost of nodes in right subtree}}$$

However, note that $d(\text{root}\,(T)\,, K_i)$ terms of the above equation in the left and right subtree parts are measured from the root K_r of the whole tree T. In fact, $d(\text{root}\,(T)\,, K_i)$ is **one more** than what it would be if we measured the level of each K_i in the left and right subtrees from the root of the corresponding subtrees. Therefore, we can rewrite the above equation in terms of the root of left (T_{left}) and right (T_{right}) subtrees.

$$\begin{aligned}
\underbrace{\zeta(K_1, \ldots, K_n)}_{\text{cost of the whole BST}} &= \Pr(K_r) + \sum_{i=1}^{r-1} \Pr(K_i)(1 + d(\text{root}\,(T_{\text{left}})\,, K_i)) + \sum_{i=r+1}^{n} \Pr(K_i)(1 + d(\text{root}\,(T_{\text{right}})\,, K_i)) \\
&= \sum_{i=1}^{n} \Pr(K_i) + \sum_{i=1}^{r-1} \Pr(K_i) \times d(\text{root}\,(T_{\text{left}})\,, K_i) + \sum_{i=r+1}^{n} \Pr(K_i) \times d(\text{root}\,(T_{\text{right}})\,, K_i) \\
&= \underbrace{\sum_{i=1}^{n} \Pr(K_i)}_{\text{constant term}} + \underbrace{\zeta(K_1, \ldots, K_{r-1})}_{\text{cost of left subtree}} + \underbrace{\zeta(K_{r+1}, \ldots, K_n)}_{\text{cost of right subtree}}
\end{aligned}$$

The optimal substructure for this problem can be **gleaned**[§] from the above, where the minimum cost BST can be computed recursively from smaller subtrees which are in turn of minimum cost themselves. To achieve this, let, for any $0 \leq i \leq j \leq n$, $c(i, j)$ store the minimum cost BST with keys/nodes $K_{i+1} < \ldots < K_j$ with corresponding probabilities $\{\Pr(K_{i+1}), \ldots, \Pr(K_j)\}$. Let $w(i, j)$ store the sum of probabilities $\sum_{x=i+1}^{j} \Pr(K_x) = \Pr(K_{i+1}) + \ldots + \Pr(K_j)$.

The following general dynamic programming recurrence holds for this problem:

$$\begin{aligned}
c(i, j) &= 0, && \text{when } i = j \\
&= w(i, j) + \min_{i < r \leq j} \left[c(i, r-1) + c(r, j)\right], && \forall i < j
\end{aligned}$$

To solve this problem, three arrays need to be maintained:

$c[i, j]$,	$\forall 0 \leq i \leq j \leq n$, storing the cost of the tree $T(K_{i+1}, \ldots, K_j)$
$r[i, j]$,	$\forall 0 \leq i < j \leq n$, storing the root of the tree $T(K_{i+1}, \ldots, K_j)$
$w[i, j]$,	$\forall 0 \leq i \leq j \leq n$, storing the sum of probabilities of keys in $T(K_{i+1}, \ldots, K_j)$

If $i = j$ the tree is treated as `null`. Otherwise, the left subtree of $T(K_{i+1}, \ldots, K_j)$ is $T(K_{i+1}, \ldots, K_{r[i,j]-1})$ and its right subtree is $T(K_{r[i,j]+1}, \ldots, K_j)$. The initializations of these arrays are **intentionally** withheld. The procedure to solve this problem involves starting from singleton node BSTs (which are already of minimum cost by definition) and systematically finding larger and larger minimum cost BSTs until the full problem is solved.

Based on this background, your lab assignment is to address the following questions:

[§]We have not proven this yet, note! As a non-assessed exercise during your self-study, attempt to prove why this optimal substructure holds for this problem.

Lab Assignment Questions

1. Access the supporting material folder in the Week 6 section of moodle. You will find two text files: `essaysOfFrancisBacon.txt` and `hamletByShakespeare.txt`. Write a python program that:

   - takes any one of the above text files as an argument;
   - reads through each character of the file;
   - retains all blank space characters (ASCII 32) intact, and converts any other character that is **NOT** from the English alphabet, [a-z] or [A-Z], into a blank space;
   - converts all lower case characters from the English alphabet, `[a-z]`, to its corresponding upper case, `[A-Z]`;
   - splits the resultant transformed text into words (delimited by blank space);**
   - writes all the split words into an output file `splitwords.txt`††, ignoring all single letter words.

2. Write another python program that takes as an argument `splitwords.txt`, and is able to identify distinct words in the file and count their frequencies by implementing a hash table using quadratic probing to resolve collisions (see lecture slides `l5.1.pdf`). Note, you **have** to implement your own hash table and are NOT ALLOWED to use any Python in-built hashing data structures. You will also have to write your own hash function that maps any key string to a table index. Your hash table must support basic functions like `lookup`, `insert`, `delete` as well as `resize`. All the choices you will make to implement this hash table should to be examined carefully for reasonable amount of efficiency. Finally, this program should output a space separated two column text file, `keyword-frequency.txt`†† with the distinct keys in the first column and their corresponding frequencies in the second. Before generating this output, this information should be sorted on the lexicographic order of the keys using any sort algorithm you have learnt in this unit – again, you will have to implement this sorting yourself.

3. Write a python program that accepts the file `keyword-frequency.txt` as an argument and, using the background information, should implement the dynamic programming method to compute the minimum cost binary search tree. The output should be written to a file `mincostbst.txt`†† with four space-separated columns containing the following information:

COL 1 Key (K_i)	COL 2 Freq(K_i)	COL 3 Level(K_i) in Mincost BST	COL 4 Cumulative sum of Pr(K_i)*level(K_i)

-=o0o=-
END
-=o0o=-

**You are free to use Python's in-built method(s) for splitting, but kudos for implementing your own.

††Sample output formats have been provided via the supporting material folder in Week 6 section of moodle.