Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Piotr Indyk and Ron Rivest

October 15, 2015
Problem Set 3

Problem Set 3

All parts are due Tuesday, October 27 at 11:59PM.

Name: David Walter

Collaborators: None

Part A

Problem 3-1.

(a) We want b to be at least $log_2(n)$, so that there are at least n b-bit numbers to map the n number of files and directories to. We know the files names and contents only consists of the 26 English lowercase letters and the . symbol, so there are 27 possible values to consider, so as long as the prime number used in the $hash\_strings$ function is greater than 27, then each character will have a unique hash value. The simple case would be if we only had files and no directories. A file has a name and the contents in the file, both of size $O(1)$. Since the name and the contents are both strings, we can simply represent a file as the concatenation of the name followed by the string contents of a file, and the string concatenation can be the string that is hashed by $hash\_string$, and the resulting hash value will be the hash value for the file. Two files with the same name and same contents will have the same hash because their name + contents string representation will hash to the same value, and if the name or contents are not the same, then they will not hash to the same value. For the hashing of the directories, you can multiply the hash values of the files and directories ($O(1)$ number of files and directories) within that directory and multiply that with the hash value of the name of the directory itself, and hash that value, to get a hash value of the directory. If the files or directories are not duplicates, then we can make the SUHA with $hash\_string$ and assume the likelihood of a collision between non-duplicates is very low.

(b) The algorithm needs to start at the files of any given directory and compute the hash values, and work out toward the parent directories and files in the parent directories and compute the hash values in that order. Start by iterating through the directories. At the first directory you will find the first sub-directory and keep going into the first sub-directories until you are in the lowest level of sub-directories and you reach just files. You will then calculate the hash values of all of the files in the given lowest level sub-directory, and the file name and directories will be stored in a hash table. For the last sub-directory that you are in, you will then hash this directory as specified in Part

(a) and put that directory into the hash table. You will then go to the next directory in this lowest-level of directories, and find the next directory, if it exists, and repeat the process for the files within that directory, and hash that directory. Once all of the files and directories of any given sub-directory are hashed and stored in the hash table, you will go to the parent directory, and continue to work your way back to the first level of directories. You repeat the whole process for all of the directories in the first level, or if $root$ contains all of the files and directories (like in the example), once you compute the hash value of $root$ you are done computing all of the hash values. Each time you get to a different file or directory, it takes constant time to compute the hash values and put the file or directory into the hash table. You will do these computations n times.

(c) After all of the files and directories are put into the hash table, the only thing left to do is iterate through the keys in the hash table and check if a given key has duplicates (if the key has more than one value), if so, then append a list of the duplicates to a list that you will return. Iterating through the hash table takes $O(n)$ time and calculating the hash values and storing all of the files and directories takes $O(n)$ time. So the algorithm runs in $O(n)$ time.

Problem 3-2.

(a) For each element in S, you want to input that element into an appropriate hash function, using the $(mod\pi)$ operator and output a hash value, and insert the element into a hash table with the hash value as the key. As you go through S, (assuming SUHA) if you reach a collision, then you found a duplicate, and you return true.

(b) Given a TNA string and a single pair of tuples $(a, b), (c, d)$, and n is the length of the TNA string called T , index into T, and extract $T[a, b] = A$ and $T[c, d] = B$, where each substring could be length n at worst. Now iterate through both substrings, A and B at the same time, and for each base in A and B, find what value the base is equivalent to in $(mod(\pi))$. Then in a hash table of length $|\Sigma|$ at worst (Σ is the set of all of the possible TNA bases) , add $+1$ for an appearance of a certain base permutation a_i , and add -1 for an appearance of a certain base permutation of b_i. After going through A and B, you will have to iterate through the hash table of length $|\Sigma|$ at worst, and check that each value equals zero.

(c) We can use the Rabin-Karp rolling hash algorithm, and assume we know how it works, as described in the class notes. At each step in iterating through the TNA sequence of length n we only need to do a subtraction of the first term, multiplication by the smallest prime number greater than $|\Sigma|$, an addition operation of the next term in the sequence, and the $(mod(P))$ (where P is an appropriately large prime number) operation, all of which can be done in constant time.

(d) In the precomputation, with T being is the entire TNA sequence we are computing with, and n is the length of T, we want to calculate all of the hash values of the substrings from the first value in T, to every other value in T, namely T[0, i], so T[0,1],

T[0,2] ... to T[0, n-1]. Once we have these values stored, they can be accessed again in constant time. At the same time, we will compute β^i, for all i in the integers from 0 to n-1, where β is the length of Σ. This precomputation can be done in $O(n)$ time, because we have n number of operations that all take constant time.

After the precomputation, given the tuples (a, b) and (c, d) we can calculate the hash value of T[a, b] and T[c, d] in constant time. We have $h(T[0, i])$ stored, where i is specified above, and $h(T[0, i])$ is the hash value of T[0, i]. Now we need to be able to calculate the hash value of any T[a, b], and T[c, d]. To compute $h(T[a, b])$ we need to calculate $h(T[0, a]) - h(T[0, b]) * \beta^{b-a}$. This is specified in Recitation 9 notes, in section 2.2. And $h(c, d)$ can be calculated in the same way, and these calculations can be done in constant time. Once you have the hash values of T[a, b] and T[c, d], they can be compared in constant time, so the algorithm runs in $O(1)$ time after $O(n)$ precomputation.

Part B

Problem 3-3.

(a) A word can be represented by a dictionary that maps the letter in a word to how many times that letter appears in that word. Representations of words with different letters cannot be the same. The word 'cart' has a 'c' with a frequency of 1 and the word 'mart' had an 'm' with a frequency of 1, so those representations cannot be the same. Also, the word 'bat' and 'batt' cannot be represented the same way because the representation of 'bat' has a 't' mapped to a 1, and the word 'batt' has a 't' mapped to a 1, so they are not the same.

The entire algorithm runs in $O(n)$ time because all of the operations run in constant time, and there are at worst n of these operations, one for each character in the string.

(b) *Python script implementation submitted.*