Assignment 3: Hash Function Behavior
Posted Friday March 3, 2017
Due Friday March 10, 2017
20 points

As we discussed in lecture and recitation, the basic algorithm for hashing strings using the division method is straightforward. Here is the pseudocode covered in class, except that the indexing of the characters in the input string increases from first character to last:

```
hash(inStr, r, m)
   h = 0
   for i = 0 to inStr.length()-1
       do h = (h*r + (int) inStr[i]) % m
   return h
```

In this assignment, you are to implement this algorithm in Java or Python such that the values `r` (the radix, or base) and `m` (the modulus, which is also the table size) are passed as arguments to the hash function.

The directory ~csci311/HW04 contains two Java files, `TestHash.java` and `Hash.java` and two Python files, `TestHash.py` and `Hash.py`. `TestHash.java` and `TestHash.py` are complete; the program reads two integers (the radix and the modulus respectively) and a string (the name of a file of words) from the command line. You need to complete the `hash()` method in the file `Hash.java` or `Hash.py`. This should be easy to do.

Test Runs

The more interesting part of this homework involves test runs using your hash function. In lecture we pointed out that the choice of radix and modulus (table size) is important. Expanding on what was covered in lecture, various problems can occur depending on relationships between the radix and the modulus. Examples include:

- The modulus is a factor of the radix, or the radix shares a factor with the modulus. To minimize this problem, **choose the radix and modulus to be relatively prime** (that is, they have no common factors). As a rule of thumb, **it is a good idea to choose a prime number for the modulus** to make this easier.
- The modulus is a prime of the form $2^p - 1$. [Number-theoretical fact: $2^p - 1$ is prime only if p is prime, but it's not prime for all prime p. If $2^p - 1$ is prime, it is called a *Mersenne prime*.] We saw in this case that the modulus and radix could interact in a way the turned the hash function into the sum of the character values mod m. In this case, all permutations of a string hashed to the same value.

We also noted that we could change the radix to a prime number to make it relatively prime to the modulus, which is chosen as a different prime number.

The directory ~csci311/HW04 contains four files of words: `wordList`, an arbitrary list of words designed to demonstrate some issues with the choice of radix and modulus, and three files containing legal five-letter words for Scrabble players. The first, `5lw-s.dat`, contains only twelve words; the second, `5lw-m.dat`, contains 1390 selected words, and the last, `5lw.dat`, contains the full list of 8938 words. You will use these files as samples of the universe of five-letter keys to see how well your choices of modulus and radix distribute keys.

Start by running your program using various pairs of radix and modulus values for the files wordList and `5lw-s.dat`. Here are a few interesting choices you might try:

radix	modulus
128	32
128	127
128	97

Observe what happens, and note the behavior for various keys. For the **first two** radix/modulus pairs listed above, **write up what you notice about the correlations of the hash values for certain keys.** These correlations will be most obvious when you use the `wordList` file. What properties of the keys lead to equal hash values for each of those radix/modulus pairs? **Record your answers in your handin.**

Now **modify your `TestHash.java` or `TestHash.py` program** to count how many of each hash value are generated by the words in your input file. How many different hash values can be created with a particular choice of modulus? Set up an array of counters of that size in your program. After you process the input by reading the words from the file, generating a hash value for each, and incrementing the appropriate counter, **output a list of your counts for each hash value into a file** (do not generate or hand in a list of all words and corresponding hash values). With a reasonable choice of modulus (e.g., 127 or less) you should be able to look at the distribution of hash values. This will help you debug your program or experiment with different radixes using the medium-size 1390 words) word list. Once you have this working, experiment with various radix/modulus pairs for the large word list.

Hand in output files of counts for two runs using the largest input file, `5lw.dat`. One run should be for a radix/modulus pair that gives a good distribution of hash values the other for a radix/modulus pair that gives a poor distribution. For both distributions, choose your moduli `m` with appropriate properties that will give you an **average in the range of forty to sixty** words per hash value. For moduli of this size, it will help to compute some simple measurements such as which hash values have maximum and minimum counts, how many zero counts there are, and statistics including mean, variance, and standard deviation for the distribution of your output. Displaying your data as a histogram using a program such as Matlab or Excel will be helpful.

For your run that illustrates a **poor distribution**, answer the following questions:

- Why did you choose the radix and modulus? What did you expect to see?
- Characterize the distribution you see in your output. Comment on why certain hash values are overrepresented, and others are underrepresented. What properties of keys cause large numbers of collisions? What key properties may have caused certain hash values to occur infrequently, or not to occur at all?

For your run that gives a **good distribution** of hash values, answer the following questions:

- Why did you choose the radix and modulus? What did you expect to see?
- Characterize the distribution you see in your output. Why do you think the distribution is good? (Note that even with a good hash function and random generation of keys, you would not see uniform numbers across all hash values for the set of input keys. Determining the quality of hash function is a statistical problem.)

What we are looking for in your reports is thoughtful analysis that is consistent with what you see in your data. Superficial responses will lose points. Note that actual words do not form a random sample from the set of all possible five letter strings; think about how this will affect your output. If you analyze your output statistically, please explain what you did and provide your results as supporting information.

Be careful to do more than just eyeball the data. What looks good may with closer analysis exhibit some bias. Look at the difference between the maximum and minimum counts, and at statistical moments such as mean, average, and standard deviation for the number of words that hash to each value. Think about what might cause such variations: the hash function? The nature of the universe of keys (actual words versus all possible strings)? Explain your thinking in your handin.

Handin

Email your `Hash.java` or `Hash.py` file to your instructor. Please use the `Subject:` line "CSCI 311 Hash Function Handin". Also attach your writeup of your analysis and the files containing outputs from the specified test runs (I want to see aggregate outputs, not long lists of almost 9000 hash values). Your writeup should be formatted to make it readable and understandable; your data files should be labeled with radix, modulus, and any other details. If you run programs to support your analysis, you can hand in analysis data as well.