STA 247 — Assignment #2, Due at start of lecture on November 22

Please submit your assignment on 8 1/2 *by* 11 *inch paper, stapled in the upper-left corner. Do* ***not*** *put it in an envelope or folder. Put your name, student number, and lecture section (Day or Evening) on the first page.*

Late assignments will be accepted only with a valid medical or other excuse.

This assignment is to be done by each student individually.

Question 1: You roll ten fair six-sided dice. Let the sum of the numbers shown on all ten dice be R. You then flip a fair coin R times. Let the number of times the coin lands heads be H and the number of times the coin lands tails be T. (So $H + T$ will be equal to R.)

Find each of the quantities below. You must produce an actual numerical answer, as a simple fraction (eg, 3/8) or decimal number (eg, 0.375). You must also justify how you obtained your answer in terms of theorems in the book.

a) $E(R)$, the expected value of R.

b) VAR(R), the variance of R.

c) $E(H)$, the expected value of H.

d) VAR(H), the variance of H.

e) COV(T, H), the covariance of T and H.

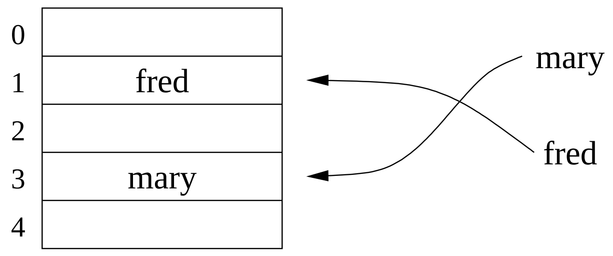
f) CORR(T, H), the correlation of T and H.

Question 2: *Hash tables* are a way of storing information so that it can be quickly looked up. Hash tables are often used by compilers, for example, to store names of variables in the program being compiled. Entries in a hash table are distinguished by having different *keys* — eg, the name of a variable in a program. Other information will usually be stored along with the key (eg, the type of the variable), but for this question we'll assume that we store just the key. We need to have a way of adding new keys to the table, and of seeing whether or not a key is present in the table.

Hash tables are based on the use of a *hash function* that maps keys to integers in some range. This mapping is designed to be more-or-less "random", so that different keys usually map to different integers. One simple way to map a key that is a character string to an integer in the range 0 to $S - 1$ is to add together the codes (eg, ASCII codes) for all the characters in the string, and then take the remainder when dividing by S (ie, we take the value modulo S). For a key that is an integer, we can just take the key itself modulo S. There are many other schemes for hash functions, however, some of which may be better than this scheme. (For instance, this scheme will produce the same hash value for strings that differ only in the order of the characters, which may be undesirable.)

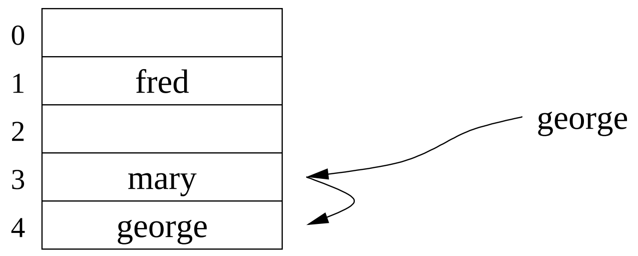
If we use a hash function that maps keys to integers from 0 to $S-1$, we will use a hash table containing S *buckets*, in which keys may be stored, implemented as a vector or array. Initially, these buckets are empty (and marked as such using some special key value). To add a key to the table, we compute its hash value, and then store the key in the bucket indexed by this hash value. (For a language that indexes arrays starting with zero, we can just use the hash value as an index; for a language such as R that indexes arrays starting with one, we need to add one to the hash value to produce an index for the corresponding bucket.)

Here's a picture of a hash table with size $S = 5$ to which we've added keys "mary" and "fred", which we'll suppose have hash values of 3 and 1:



If we now want to find out whether "mary" is in the table, we compute its hash value of 3, look in bucket 3, and see that "mary" is indeed there. If we look for "bert", with hash value 2, we will see that bucket 2 is empty, so "bert" isn't in the table. If we look for "sally", with hash value 1, we will see that "fred" is in bucket 1, so "sally" isn't in the table (but see below for how this procedure will have to be modified). Note that we can quickly find out whether a key is in the table without having to look at all the buckets. This speed of search is why hash tables are attractive.

Suppose we now try to add the key "george", which has a hash value of 3. We have a problem, since bucket 3 is already occupied by "mary" – this is called a *collision*. There are many ways of resolving this problem. One way is to say the each bucket contains not just one key (or nothing), but rather a linked list of keys, in which we can hold as many keys as needed. Another way, which we'll consider for this assignment, is to move on to another bucket when the one indexed by the hash value is already occupied. For example, we might just move to the next bucket, wrapping around to the first bucket if we reach the end of the table. With this scheme, we'd add "george" (whose hash value is 3) to the previous table as follows:



When using this scheme, we can no longer assume that a key that isn't in the bucket corresponding to its hash value isn't in the table. We have to look at the next bucket, the bucket after that, etc. until we either find the key or we reach an empty bucket. This slows down the search for a key in the table.

A problem with this scheme is that we can sometimes end up with solid "clusters" of keys in one section of the table. If we try to add a key whose hash value is in this section, we'll have to move forward many buckets to find a place to put it, and finding it later will be slow. To reduce this problem, we might move forward by a variable number of buckets when a collision occurs, hoping that this will tend to produce smaller clusters. In particular, the number of buckets we move forward could be found by applying a second hash function to the key. For this to be guaranteed to work, the size of the hash table should be a prime number, so that stepping forward many times will bring us back to the original bucket only after we've seen all the other buckets.

For this question, you are to evaluate how well these schemes work. How well a scheme works will depend on how often collisions occur, which we can model as a chance process. On the course web page, you will find R functions that implement a hash table, which you can use.

These functions are summarized below, but see the comments before them for further details. You will need to write additional R functions that simulate using hash tables, and produce summaries and plots that indicate how well the schemes work.

The `hash.table.setup` function initializes the table to have a specified size (the `size` argument), and to use a specified scheme for deciding how many buckets to step forward when a collision occurs (the `step` argument). If the `step` argument is one, collisions are resolved by stepping forward one bucket at a time. If the `step` argument is greater than one, then a second hash function applied to the key is used to select how many buckets to step forward by, which will be between one and the `step` argument.

The `hash.lookup` function looks for a key in the table, and optionally adds it to the table if it is not already present. Keys can be either strings or integers, but must all be either one or the other for a single table. The `hash.lookup` function returns a list containing a `found` flag, indicating whether or not the key was present, as well as a `probes` count of the number of buckets that had to be examined when looking for the key. This count is an indication of how long the search took.

For your tests, you should set the hash table size to 197. You should compare how well the scheme works when using `step` arguments of 1 and 20. For both values of `step`, you should do the following 50 times:

1. Initialize the table with `size` set to 197 and `step` set to either 1 or 20.
2. Add 160 keys to the table, randomly picking your keys (without replacement) from the integers from 1 to 1000.
3. Perform 200 lookup operations on the table, looking for keys that are randomly sampled (with replacement) from the set of integers from 1 to 1000. (So about 16% of these will be in the table, the rest not.)
4. Record how many probes were needed for each of these lookup operations.

After doing this, you will have $50 \times 200 = 10000$ numbers giving the number of probes needed for each of these lookup operations. (These are probably most conveniently stored in a matrix.) You should now use these numbers to compare how well the hash table works with `step` set to 1 and to 20. In particular, you should do the following:

1. Plot two histograms of how many probes were needed with `step` set to 1 and to 20.
2. Estimate the expected number of probes needed with `step` set to 1 and to 20, using the sample mean of the data.
3. Estimate the expected number of probes needed for each of the 50 tables constructed, using the sample means for lookups in each table, and plot two histograms of these means (one for `step` set to 1 and one for `step` set to 20).
4. Estimate the probability that a lookup will require more than 25 probes when `step` is set to 1 and to 20.
5. Say whatever you can about how accurate you think the estimates you found above are.

Finally, you should briefly discuss what the above results mean — ie, summarize what you have learned about the differences with `step` set to 1 and to 20. You should hand in this discussion, the plots and estimates above, and a listing of your R program.

A list of some features of R that may be useful will be put on the web page soon.