Computer science problems.

About the problems. This problem set explores hash tables and hashing functions.

What you need to do. For these problems we ask you to write a program (or programs), as well as write some "paper-and-pencil" solutions (use any text editor that you see fit, or scan an actual handwritten solution; convert the result to pdf format if possible).

You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows (free trial versions do not qualify). It is best to implement each problem as a separate function so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. **Important: do not include your name in comments or in any file names.** If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**.
  Please note that we only need the source files since we will be compiling them before running. The compiled code and executables just clutter your submission - please avoid submitting them if possible.

  In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
  2. The *programming language*, including the *version* (Java 1.20, for instance), the *development framework* (such as VS Code) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your pro-

gram file or as a separate file, clearly named. Please read the instructions for individual problems on the input and output data.
Input/output files are assumed to be at the default location for your program's project. Make it clear in comments where that is.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.
7. Make sure that you clearly specify where input files are supposed to be located, provide an example input file and an example of how the file name would be specified in the input. Use relative paths (from the top of the project or from the executable), not absolute paths.

- Clear, understandable, and well-organized code. This includes:

1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
5. While we are not asking for the fastest program (it's better to make it more readable), you should avoid unnecessary overhead.

Background and main definitions. This year's problem deals with hash tables. This sections gives a brief overview of hash tables. For more information see [2] and the section on hash tables in [1], or any other book on data structure that provides a detailed coverage of hash tables, including collision resolution.

Hash tables are a popular data structure that, under certain assumptions, allows a constant time data lookup (i.e. the time looking for a specific element doesn't depend on the number of elements already stored; an element is found "instantly"). Typically data stored in a hash table has a key that's used for hashing and a value that's bound to that key. Data is inserted into the hash table, but typically not removed.

The basic idea of a hash table is as follows:

1. A hash table is an *array* (or a vector, depending on what's provided by your programming language) of size n. It is initially filled with some value that marks an empty spot. We will refer to this value as `NULL`.

2. In order to insert an element, a function (usually referred to as a *hash function*) is applied to the element. If the element is a key/value pair then the hash function is applied to the key.
   The hash function produces an index in the array.

3. If the index in the array has the `NULL` value, then the element is assigned to that spot.

4. If the spot is not `NULL` (i.e. already has an element in it), that's called a *collision*. When a collision happens, a *collision resolution* procedure is applied. There are two basic approaches: *chaining* which uses additional storage, such as linked lists, and *open addressing* which is trying to find a different spot for the element using another function (known as a *probing function*); the process of determining which index to check next and checking that index is known as a *probe*. If that spot is full as well, that's another collision, and the collision resolution is applied again (so another probe is performed), until the spot is found.

5. The size of the array n is known as the table's *capacity*. The table cannot hold more than n elements.

6. The number of elements in the hash table is referred to as the hash table's *size*; we will use s to denote it. Note that $0 \leq s \leq n$.

7. The fraction $\frac{s}{n}$ (as a percentage value) is referred to as the *load factor*. For example, when the table is $\frac{3}{4}$ full, its load factor is 75%.

8. Hash tables are typically used to retrieve values associated with the keys. In order to look up a key, the process similar to inserting an element is performed: first the hash function is applied, then the location at the index returned by the has function is checked to see if the key there is the one we are looking for. If it is, we return the corresponding value. If it's `NULL`, then the key doesn't appear in the hash table. If it's a different key then collision resolution is performed until probing finds the index with the key we are looking for or with `NULL`.

Some important conventions and assumptions for this problem set:

1. In all examples we will just consider the keys, not associated values.

2. The keys are assumed to be non-negative integers.

3. The keys are assumed to be *distinct*. If there already is an element in the hash table with the exact same key as the one you are inserting, that's an error.

4. Hash tables are usually resizable, i.e. when the load factor becomes high (typically around 75%), a new, larger array is allocated and all the elements are moved over to the new table. However, for our problems we assume that the table is *never resized*.

5. We only use open addressing, never chaining.

6. The indices in the array start at 0.

We will introduce more material in the problems themselves.

Problem 1 (non-programming): the basics of hash tables. Please answer the following questions:

1. Why hash tables keys in a program should be of an immutable type, such as integers or immutable strings?

2. Assuming that there are more possible keys than the capacity of the hash table, why some amount of collisions is inevitable?

3. If you are given a hash table with capacity 100. Why the function $h(k) = k \mod 10$ is not a good hash function for this table? Here k is the key we are inserting and $\mod$ is the modulus operator (see `https://en.wikipedia.org/wiki/Modular_arithmetic` for details).

4. Given open-addressing collision resolution, why deleting a key from a table by replacing it by `NULL` doesn't work? Describe the issue precisely.

Problem 2 (non-programming): typical hash functions and traditional collision resolution. If the keys are integers, the most common hash function is $h(k) = k \mod n$, where n is the table capacity. It is simple and always returns a valid array index.

Recall that if a collision occurs (i.e. the index returned by the hash function is occupied), a probing function is used to find the next spot. In addition to the key k the probing function takes the *probe number* i, where i is 0 when the first attempt is made, i.e. the $h(k)$ is tried, and then $i = 1$ if that spot is occupied, etc. Typically instead of specifying the hash function and the probing function separately, we just write the probing function in such a way that it equals to $h(k)$ when $i = 0$.

Here are several common probing functions, written as $p(k, i)$, where k is the key and i is the probe number:

1. *Linear probing* uses a linear probing function to find the next position of the key, i.e. a function of the form

$$p(k, i) = (h(k) + ci) \mod n$$

Taking the result modulo n guarantees that the result is a valid index. c is called the *step* of linear probing. Most commonly $c = 1$, i.e. linear probing will just try the next index if the one it's probing is unavailable. If $c > 1$, it must be relatively prime with n.

2. *Quadratic probing* uses a function quadratic in i (the probe number) to find the next spot to probe:

$$p(k, i) = (h(k) + c_1 i + c_2 i^2) \mod n$$

Depending on choices made for n (for example, in some hash tables $n = 2^m$ for some m, and in some tables n is chosen to be prime), there are different common choices for c_1 and c_2. The wikipedia page `https://en.wikipedia.org/wiki/Quadratic_probing` provides more details.

3. *Double hashing* involves a "secondary" hash function $g(k)$ for collision resolution:

$$p(k, i) = (h(k) + ig(k)) \mod n$$

If $h(k) = k \mod n$ then $g(k)$ is often chosen to be $g(k) = (k \mod m) + 1$, where $m < n$ and m is relatively prime to n.

Linear probing is the simplest of the three strategies. However, it leads to an issue known as *clustering*: if several keys ended up following the same probing sequence, a new key that collides with any of them would also follow this sequence, so the probing sequences get longer and longer, leading to linear search times, not constant.

Based on these functions, please answer the following questions:

1. What's the point of adding 1 to $k \mod m$ in double hashing? Show an example of a specific issue that this addition is preventing.

2. Explain why linear probing (with c relatively prime to n) and double hashing (with $g(k) = (k \mod m) + 1$, where $m < n$ and **both** m **and** n **are prime**) always find an empty spot for a key being inserted into the table, as long as $s < n$. Note that this is not always the case for quadratic probing - you don't need to explain why.

3. Explain how both quadratic probing and double-hashing are reducing clustering. Be specific about the differences in these two methods.

4. The order of inserting keys into a hash table matters. Consider a hash table with capacity $n = 11$, $h(k) = k \mod 11$. Show the hash table that results from inserting the keys $14, 27, 33, 3, 18, 13, 37, 15, 22$ in this order. Then show the same table with the same hash function and probing function, but with the keys inserted in the opposite order: $22, 15, 37, 13, 18, 3, 33, 27, 14$. How many collisions (i.e. probes with $i > 0$) took place in each of these cases?
   Answer these questions for each of the following collision resolution functions:

   (a) Linear probing with $c = 2$

   (b) Quadratic probing with $c_1 = 0, c_2 = 1$

   (c) Double hashing with $g(k) = (k \mod 3) + 1$

5. You are given the following hash table

   44 _ 13 _ _ 16 24 18 6 _ 21

   (`NULL` values are dented by underscores). Can it be the result of inserting the keys using linear probing with $c = 1$ and $h(k) = k \mod 11$? if yes, give the order of the keys that result in this table. If not, clearly explain why.

Description of the main problem. The main problem of this problem set deals with constructing custom-made hash functions and probing functions for known non-uniform key distributions while trying to minimize the total number of operations performed.

The keys are positive integer numbers. Each input set consists of decimal numbers of a fixed length L (different for different input sets), where $4 \leq L \leq 10$. Leading zeros (if any) will be explicitly written. Each digit of the key has its own non-uniform distribution, i.e. some digits are more likely to appear in a given position than other ones. Some digits may have a frequency of zero in a particular position, i.e. they don't appear in that position at all.

The distributions of each digit will be given as input before the actual keys are given. Each distribution consists of 10 non-negative numbers that add up to 1, where the first number indicates the proportion of 0s, the second proportion of 1s, etc. For example, the distribution

```
0 0 0.2 0 0.3 0 0 0 0 0.5
```

means that this position consists of 20% of 2s, 30% of 4s, and 50% of 9s. No other digits appear.

In order to specify distributions of L-digit numbers we need to provide L distributions. For example, for 6-digit numbers the input may be like this:

```
0 0 0.2 0 0.3 0 0 0 0 0.5
0.2 0.05 0.05 0.05 0.05 0.05 0.05 0.5 0 0
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
0 0 0 0 0 0 0 1 0 0
1/3 0 0 0 0 1/3 0 1/3 0 0
0.1 0.1 0.1 0.1 0.05 0.1 0.1 0.05 0.1 0.2
```

Here is an example of ten 6-digit numbers generated by this combined distribution:

```
408775 942771 208756 943775 915704 476759 470770 956751 945758 238754
```

The hash table may have a capacity $1000 \leq n \leq 1024$, you may chose the capacity that works better with your algorithm. You may set it once and for all or choose it after you have read the digits distribution.

Hash function and probing functions. Your goal is to develop a hash function and the probing function based on the distributions to optimize the hash table performance with large load factors. The performance is measured

based on the total number of operations inserting the numbers into the hash table, including all operations needed for collision resolution.

The following operations will be counted:

- Addition of two numbers of length no more than L,
- Multiplication of two numbers of length no more than L,
- Taking a number of length L modulo another number (of the same length or smaller),
- Performing any bitwise operations, such as shift, &, etc. The result must be no longer that $L + 1$ in decimal representation,
- Copying a number into another memory location. Only a portion of length L or smaller may be copied as one operation,
- Performing a comparison for equality or inequality of two numbers of length L or smaller (checking if the hash table position is NULL doesn't count).

The length of the number is the length of its decimal representation. The actual storage in computer memory (i.e. storing it as an `int` or a `long`) doesn't matter.

Operations on numbers with the length of $L + 1$ are disallowed. You may compute such operations iteratively by taking the result modulo something of length L or smaller after each iteration (for example, when computing exponentiation).

Example of counting operations. Suppose are using double hashing, i.e.

$$p(k, i) = (h(k) + ig(k)) \mod n,$$

where $g(k) = (k \mod m)+1$ for some m. In order to optimize it, we can rewrite it the following way, in pseudocode:

```
if (i = 0)
   return (k mod n)
else
   return ((k mod n) + i *((k mod m) + 1)) mod n
```

This way we are avoiding performing multiplication by 0 in the first probe (when $i = 0$).

Suppose you found the place for the key at $i = 1$, then you performed the following operations:

- The first attempt computed $k \mod n$ and one comparison of i to 0 in the `if` statement, for the total of 2 operations.
- The probe with $i = 1$ computes $((k \mod n) + i(k \mod m) + 1) \mod n$, performing 3 modulo operations, 2 additions, and multiplication by i (even though $i = 1$, the multiplication still needs to be performed). It also performs one comparison of i to 0, so the total for this step is 7 operations.

Thus the total to insert the key into the hash table is 9 operations.

Your functions may use any approaches you would like. If you are using operations not listed here, please ask about their costs and whether they are allowed; be specific.

Problem 3 (programming): implementing custom-made hash functions and collision resolution. Your goal is to write a program that works as following:

- It reads the number L on its own line, followed by the distribution of digits in the format given above (L lines of 10 numbers each, the numbers can be given as ratios or as decimals).
- It checks that the distribution is valid: each line consists of 10 non-negative numbers that add up to 1 within ± 0.00000001. If it's invalid, the program stops and prints an error message, indicating the first line where the error appeared.
- If the distribution is valid, the program determines the hash table capacity n ($1000 \leq n \leq 1024$) and chooses the hash function and the probing function. You may choose them from the list or set some parameters or compute them on the fly if your language allows you to - all of these are valid options. The hash function and the probing function may also be combined.
- The program outputs the hash table size and starts a loop prompting for a file name. The file contains m items generated by the distribution, $n * 0.75 \leq m \leq n * 0.95$ (we are testing the hash table with the load factor between 75% and 95%). The elements are unique. The elements may be separated by spaces, tabs, and newlines.
  The path to the file is relative to your executable or to the default location of your project (Java projects tend to have a default location). In comments clearly specify where the input files are located.
- The program uses its hash function and probing function to insert the elements into the table in the given order.
- After the program reaches the end of file it prints out the total number of operations computed by the hash function and the probing functions (together, on all elements).
- You give the user an option to print the hash table.
- You also give the user an option to search for elements (as many as they would like). For each input number your program responds with an index if the item is found and a message "not in the table" if it's not. If the given number has a wrong length, print an error message and then prompt for another number.

- After these interactions you clear the table, setting it all to `NULL` values, set the operation counter(s) to 0, and prompt for another file name with a different set of test data for the same distribution. Entering Q ends the program.

Some restrictions:

- You may use only small constant (in n) amount of memory in addition to the hash table; you may allocate arrays of length 10 (for 10 digits) in addition to the ones where you store the distributions, but no more than L of such additional arrays
- You may not store any information about the elements you have already inserted
- You may not keep the count of the elements (your probing function, however, may use the probe number in any way you'd like)
- You may not perform any look-ahead in the data

Your program will be graded on:

- Correctness (if it's not always correct, i.e. the hash function doesn't work as specified for insertion and search, then the number of operations doesn't matter; correctness also means that you count the operations correctly)
- Clarity of the algorithms and a clear explanation of how it works and why it's correct
- Minimization of the number of operations on several test runs with different load factors
- Test examples and the quality of testing. Don't forget to test how your functions count the operations
- The setup for input and output
- Clear instructions for compiling and running your program. C/C++ users - please don't use makefiles since they are specific to the operating system! Generally your code should be runnable in a bare-bones command-line setup and not require a specific IDE or tools

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[2] Thomas Mailund. *The Joys of Hashing: Hash Table Programming with C.* APress, 1st edition, 2019.