CS 290-01 Algorithms in the Real World: HW3

Due 10:15 am on Friday, September 18

Submission. You will submit materials for homework number n on Gradescope under two separate assignments: HWn Report and HWn Code. You should turn in a pdf containing all of your written solutions for the report, and a source code file containing all of the code you wrote for the assignment. When you submit your code, the interface may mention an autograder; you can ignore this, as the assignment will not be autograded. Your grade will be recorded on the report; the source code is to show your work and for reference during grading.

Report. The pdf you submit for the report should be typed, and solutions to individual questions should be clearly labeled. Show your work or explain your reasoning for your answers. You should use LaTeX (which is free) or some other editor of your choice (Microsoft Word, Google Docs, etc.) to prepare your reports. If you use an editor like Microsoft Word, make sure to convert the final document to a pdf, confirm that the symbolic math from the equation editor is properly formatted, and submit the pdf.

Collaboration. You may complete this assignment independently or in a group of two students. If you work with another student you should submit a single report and a single code file with both of your names, and should use the group feature when submitting on gradescope to indicate that you worked as a group. Do not split up the assignment, and each only complete half of the problems. Instead, complete each portion of the assignment by working together synchronously (for example, by pair programming with screen sharing over zoom or some other similar service) or by working independently and then coming together to merge solutions and check one another?s work.

Allowed Materials. You can use any standard library functions and data structures in your programming language of choice (Java or Python 3 are recommended). You may also use any slides or notes from class or reference materials posted on the course website. You may search the internet for basic definitions, terminology, and language documentation (for example, checking the syntax for array slicing in python), but you may not use anyone else's code (either another student's or from the internet), nor may you search the internet for solutions or descriptions of solutions to the homework problems.

Problem 1 (Industrial Grade Hash Functions). This problem will provide an introduction to the use of industrial grade hash functions (in particular, SHA-256) which will be used throughout the rest of the assignment and possibly in subsequent assignments. We will provide directions for Python 3 and Java; SHA-256 is readily available through standard library support in both languages. Read the directions for your preferred language to get started.

- *Python 3.* See `https://docs.python.org/3/library/hashlib.html` for the full documentation. You will need to import the `hashlib` library to get started. To initialize a SHA-256 hashing object, you can use a command like `hasher = hashlib.sha256()`. To pass the key you would like to hash, you will need to encode it as a bytes object. You can do this directly with the `encode()` function and pass it to the hash object. Say you want to hash a string called `my_string`, then you just call `hasher.update(my_string.encode("utf-8"))`. Then to get the hash, the code is just `hash = hasher.digest()`, which will return another bytes object. You can directly access these bytes with standard indexing in Python so that `hash[0]` will give you an integer between 0-255 corresponding to the first byte, `hash[1]` will give you an integer between 0-255 corresponding to the second byte, and so on. NOTE: Calling update multiple times *concatenates* strings with previous calls to update. To get the hash of multiple different strings, you should create a new SHA-256 hashing object every time.

- *Java.* The Java instructions are very similar. You can see the documentation at `https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/MessageDigest.html`. You will need to import java.security.MessageDigest to get started. To initialize a SHA-256 hashing object, you can use a command like `MessageDigest hasher = MessageDigest.getInstance("SHA-256");`. To pass the key you would like to hash, you will need to encode it as a byte array. You can do this directly with the `getBytes()` method, pass it to the hash object, and get the hash value. For example, say the string you want to hash is called `myString`. Then the code to get the hash is just `byte[] hash = hasher.digest(myString.getBytes());`. The result is another byte array. As in Python, `hash[0]` will give you the first byte, `hash[1]` will give you the second byte, and so on, but unlike in Python, these values will be *signed* 8-bit integers between -128 and 127. To get the corresponding unsigned integer from 0-255 in Java, you simply need need to convert with some binary addition; the code is `hash[0] & 0xFF` to get the 0-255 int corresponding to the first byte, `hash[1] & 0xFF` to get the 0-255 int corresponding to the second byte, and so on.

Note that the above directions implement the same SHA-256 algorithm, so you should get the same results regardless of which language you use. As an example to check your understanding, consider the SHA-256 hash of `"Duke"`. The first byte (as a 0-255 integer) is 79, the second is 226, the third is 75, ..., and the last (32nd) byte is 168. Note that it is case sensitive.

(a) Using the above directions/libraries, implement a function/method `get_bytes`[1] that takes a string and an integer k as input and returns the first k bytes of the SHA-256 hash digest of the string, representing each byte as an integer between 0 and 255. For example, `get_bytes("Duke", 3)` would return `[79, 226, 75]`. What are the first three bytes (as 0-255 integers) of the SHA-256 hash of `"hello"`? Note that it is case sensitive.

(b) Implement a simple hash table (i.e., a do it yourself version of a very simple hash map in Java or dictionary in Python). Your hash table should store (key, value) pairs where the keys will be strings and the values will be integers. Your hash table should be of size 256 (that is, there should be 256 "buckets" for keys to hash to), and you should hash as follows: for a given key, use your get_bytes function to get the first byte of the SHA-256 hash digest as an integer between 0 and 255; use that as the index in the hash table where you store the (key, value) pair.[2] You should use simple chaining to handle collisions in your hash table, meaning that each "bucket" in your hash table should be a list to which you append the (key, value) pair when hashing a new key, and you simply search through the corresponding list when looking up a particular key for its value.[3]

(c) The file "stream.txt" contains a stream of the words appearing in the English novel *A Tale of Two Cities*.[4] The stream has been pre-processed for you to only contain lowercase words without punctuation, separated by spaces, with no newline characters. Using your simple hash table, count how many times each unique word appears in the stream with a *single* loop over the stream. You are encouraged to use a standard hash table implementation in your language of choice (i.e., a hash map in Java or a dictionary in Python) to validate your own hash table implementation. Report how many times the following words appear in the stream:

    (i) `the`
    (ii) `are`
    (iii) `sydney`
    (iv) `london`

[1]You can name it whatever you want, put it in a class or not, etc, as long as it has the basic functionality. You will need this implementation in order to complete several subsequent tasks in the assignment.

[2]For simplicity, you are only required to implement the special case of 256 buckets for this homework. However, note that it is not too difficult to see how to generalize this basic idea: if you want a hash table with n buckets, simply take the first $\lceil \log_2(n)/8 \rceil$ bytes from the digest and take that value (as an integer) mod n. Feel free to implement this more general version if you want more programming practice.

[3]Note that probing based techniques for handling collisions are only effective when the hash table is larger than the number of keys being stored in it., which will not necessarily be true for this simple fixed-size 256 bucket hash table.

[4]This is actually a very small stream, but will allow us to test all of the basic ideas that would apply to a much larger stream.

(d) How much space/memory (in bytes) does the hash table from part c (after processing the stream) use? To answer this question, assume that each string (the keys you store) requires one byte per character in the string, and each integer (the values you store) requires four bytes (the default size of an int in both Java and Python). For the purposes of this problem, you may ignore the memory required to store pointers in the chaining lists (although we will certainly also count your answer as correct if you choose to complete this more detailed analysis).

Problem 2 (Count-Min Sketch). Recall that a Count Min-Sketch (hereafter, CMS) data structure is specified by multiple hash tables. Unlike a regular hash table, we do *not* store keys, and we do *not* handle collisions. Instead, each "bucket" of each hash table simply stores an integer that functions as an estimate of the count of keys hashed to that bucket. If you like, you can think of the count min sketch as a two dimensional table where every row i is an independent hash table with hash function $h_i()$, and for a given element x, $CMS[i][h_i(x)]$ contains the i'th estimate of the number of times x has occurred in the stream. The dimensions of the CMS are:

- r = The number of "rows" or independent hash tables/functions $h_1, \ldots, h_r$. We will use $r = 5$ hash tables in our testing (of course, feel free to play around with different values).
- n = The size of each hash table, i.e., the number of "buckets" per hash function (or number of "columns"). We will use n = 256 for all of our tests.[5]

Recall that each hash table or row in the CMS provides an estimate of the number of times a given element has appeared in the stream. We take a single pass over the stream, and every time you see an element, you will increment each estimate of that element in the CMS. Our overall estimate for the number of times a given element has appeared in the stream is the minimum of these estimates. For this problem, we will use the first five bytes of the SHA-256 hash digest (as integers from 0 to 255) to implement the $r = 5$ independent hash functions. For example, the first five bytes (as 0-255 integers) of `Duke` are 79, 226, 75, 13, and 255 respectively. So we would store our estimated count for "Duke" at index 79 in the first table/row, index 226 in the second table/row, and so on.

(a) Implement the CMS. You can test your implementation using the stream.txt file. The file contains a long stream of words (separated by spaces). You should take a single pass through the stream, and for each word, increment the CMS estimates. Your final estimates should be the *minimum* of these. For validation purposes, you should get the following estimates for the provided words:

- `paris` 185
- `her` 1186
- `well` 377

(b) Report the overall estimated counts of your CMS for the following words. Also report the error for each of these counts as the ratio of your overall estimated count to the real count (note that you reported the real counts of these words in problem 1). For example, if your estimated count was 125 and the real count was 100, you would report a ratio of 1.25.

(i) `the`
(ii) `are`
(iii) `sydney`
(iv) `london`

(c) An optimization to the CMS is to use *conservative updates.* With conservative updates, when you see a particular word in the stream, rather than incrementing *all* of the estimates for the count of that word, you only increment the current *minimal* estimate(s) for that word. If there is a tie, you increment all

[5] As in problem 1, this value is chosen to make it easy to get the indices from the bytes of a hash digest. Also as in problem 1, it is easy to generalize this to arbitrary table sizes.

of the tied estimates. For example, say your five estimates from your five hash tables / CMS rows for the word `hello` are 7, 9, 12, 7, and 10 respectively. The next time you see `hello` in the stream, you would only increment the two estimates of 7, leaving the other values unchanged. Reporting the overall estimate remains unchanged; you still report the minimum of the individual estimates. Explain why we still *never underestimate* the count of any element, even when using conservative updates.

(d) Implement conservative updates for your CMS and again process the stream. Report the estimated counts of your CMS with conservative updates for the following words. As in part b, also report the error for each of these counts as the ratio of your estimated count to the real count.

  (i) `the`

  (ii) `are`

  (iii) `sydney`

  (iv) `london`

(e) With both the original CMS and with conservative updates, you should observe that the error ratio for more common words is smaller than the error ratio for less common words. Explain why you would expect this to happen. Your answer should refer to how the CMS works.

(f) How much space/memory (in bytes) does your CMS use to estimate the counts from the stream? Answer for the specific setting of 5 hash tables each of size 256 storing 4 byte integers. What is the ratio of the size of your hash table from problem 1 part d to this value (i.e., how many times less memory does your CMS use as compared to your hash table from problem 1)?