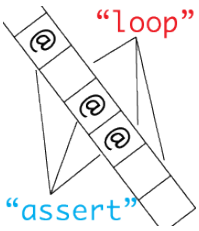
15-122: Principles of Imperative Computation, Spring 2023

Programming Homework 7: Bloom Filters



Due: Thursday 16th March, 2023 by 9pm

In this assignment, you will implement a variation of hash-table-based sets, called *Bloom filters*, and explore some of the applications of Bloom filters.

Download the assignment handout from the course website or Autolab.. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a SEVEN (7) PENALTY-FREE HANDIN LIMIT. Every additional handin will incur a small (5%) penalty (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

To help you write test cases for Task 1, which we want you to do before starting on the later tasks, there will be a separate, unofficial "Bloom Filter Test Case Checker" Autolab assignment. This will only run the part of the autograder that checks Task 1. There is no handin limit for that unofficial autograder. Check the `README.txt` file carefully for details.

1 Bloom Filters

Fundamentally, Bloom filters are an implementation of the *set interface* discussed in the lecture handout:

```
// typedef ______* bloom_t;

bloom_t bloom_new(int capacity)
  /*@requires 0 < capacity; @*/
  /*@ensures \result != NULL; @*/ ;

bool bloom_contains(bloom_t B, string x)
  /*@requires B != NULL; @*/ ;

void bloom_add(bloom_t B, string x)
  /*@requires B != NULL; @*/
  /*@ensures bloom_contains(B, x); @*/ ;
```

The one interesting twist is that the `bloom_contains` function *is allowed to return false positives*. If the Bloom filter says that something *is not* in the set, it has to be right, but if it says that something *is* in the set, it can be wrong. To put it a different way, a Bloom filter answers the question "is `x` in the set?" with either the answer "no" or "maybe."

Here's a summary of how the output of `bloom_contains(B,x)` may relate to whether `x` is in `B`:

	`x` is in `B`	`x` is **not** in `B`
`bloom_contains(B,x)` returns `true`	**SOMETIMES** *(true positive)*	**SOMETIMES!?** *(false positive)*
`bloom_contains(B,x)` returns `false`	**NEVER** *(false negative)*	**ALWAYS** *(true negative)*

The words in all-caps describe when each combination happens. As you can see, false negatives (**SOMETIMES!?**) are problematic: `x` may not yet be in `B`, and yet `bloom_contains(B,x)` returns `true`.

Try to think of a couple of obvious and possibly silly ways you could implement this interface before you continue reading.

There are a lot of possible correct implementations of the interface we gave on the previous page. One always returns `true`, signaling "maybe `x` is in the set." You've been provided with this implementation in `bloom-worst.c0`, and it is, according to the description we gave, a *correct* implementation. It is also very fast and uses very little space! It is terrible in every other possible way.

At the other end of the spectrum, you could implement a proper set. You have the tools to do this in a couple of ways: unbounded arrays (sorted or not), linked lists, and hash tables. You have been provided with one such implementation in `bloom-expensive.c0`. Such an implementation will only signal "maybe `x` is in the set" when `x` is *really, actually* in the set.

This is fine, but it ends up using lots of memory, and what's more, the implementation you were given is quite slow.

The implementations we're going to explore in this assignment will be in-between: they use less memory than a real hash table, but give fewer false positives than a completely non-committal implementation. Before we talk about how to do this, and before we go and do it, let's write some tests!

1.1 Testing Bloom Filters

Task 1 (6 points) In file `bloom-test.c0`, write a testing program that respects the interface on the previous page. It should serve two purposes:

- The testing program should attempt to raise an assertion error on any incorrect implementation of the interface.
- On any correct implementation of the Bloom filter interface, the `main()` function should return a *performance score* from 0 and 100 (inclusive).
  - On the worst possible Bloom filter implementation described above, the performance score should be 0.
  - On an "error free" Bloom filter implementation (such as an actual hash table), the performance score should be 100.
  - On any Bloom filter implementation that has some false positives and some (true) negatives, the performance score should be between 0 and 100.

We ask that you do not call `bloom_new` with a capacity greater than `int_max()/16`.

Generally speaking, worse implementations should have lower performance scores. You will be graded in part based on whether your tests are able to distinguish relatively bad (but not pessimal) implementations from relatively good (but not perfect) ones.

An idea to keep in mind when you are writing your tests is that Bloom filters, like hash tables, have a *load factor*. If n is the total number of distinct elements that have been inserted and m is the table size that was set by `bloom_new(`m`)`, then the load factor is n/m. We will generally expect lots of false positives when the load factor exceeds 1, and vastly fewer false positives when the load factor is much smaller than 1.

You are strongly encouraged to go ahead and submit to the Autolab using the unofficial "Bloom Filter Test Case Checker" autograder before you move on from this task.

1.2 Using Bloom Filters

This section contains motivation for when Bloom filters may come in handy. It may help you with ideas as you're writing test cases, but it's not essential for the rest of the assignment.

Our goal in this assignment will be to develop high-performing Bloom filter: one that return `false` as often as possible while using much less memory than a fully-correct set implementation must use. When can such a data structure be useful?

Simple Rules, Expensive Exceptions When dealing with human concepts like language, maps, traffic law, or time zones, it's sometimes possible to write a simple algorithm that *usually* gives the right answer. However, these simple algorithms almost always have to be augmented with extensive databases containing the idiosyncratic exceptions. A Bloom filter can record all the places where our simplistic algorithm *doesn't* return the right answer. Then, we can quickly ask the Bloom filter "is this one of the exceptions where the simple algorithm doesn't work?" If the answer is "no," we use our simple algorithm. If the answer is "maybe," then we look it up in our carefully-maintained database.

Human language was one of the original motivating examples for Bloom filters.[1] Burton Bloom imagined an extensive database of rules for hyphenating English words in a text editor. A Bloom filter could capture all the words that can't be hyphenated automatically with a simple algorithm, requiring a database lookup.

Fast First Passes If you've ever used a full-featured text editor like Microsoft Word, you've probably had the experience of watching as the spell checker highlights all the misspelled words in a document. A Bloom filter could speed up this process by storing all the correctly-spelled words in a dictionary. On the first pass, Word could report misspellings only when the Bloom filter says a word definitely isn't spelled correctly. Then the maybe-correctly-spelled words can be checked in a second pass to weed out the false positives. In this case, false positives would be incorrectly-spelled words that the Bloom filter did not flag as misspelled.

Making sure that a user doesn't pick a password that is known to be compromised is another application of Bloom filters that falls in this category. You will be implementing it in Section 4 of this assignment.

One-Hit Wonders Wikipedia describes several additional use cases for Bloom filters.[2] Services like Akamai or Netflix try to store copies of frequently-used content physically close to users, but they have limited storage space.

Due to the way people use such services, a good rule in practice is that, if two people in the same region request the same content, it's worth storing a copy of that content near them. This means that "one-hit wonders", content that only one person wants to view, doesn't make use of valuable storage space.

A Bloom filter can help with this problem by storing all the recent requests for content. Whenever new content is requested, the Bloom filter is asked whether anybody else recently requested that material. If the Bloom filter says "maybe," then the server assumes this is the second request and stores it. False positives mean that some one-hit wonders get stored, but the trade-off is sometimes worth it.

[1] Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors." Communications of the ACM, Volume 13 Issue 7, July 1970.

[2] `https://en.wikipedia.org/wiki/Bloom_filter`

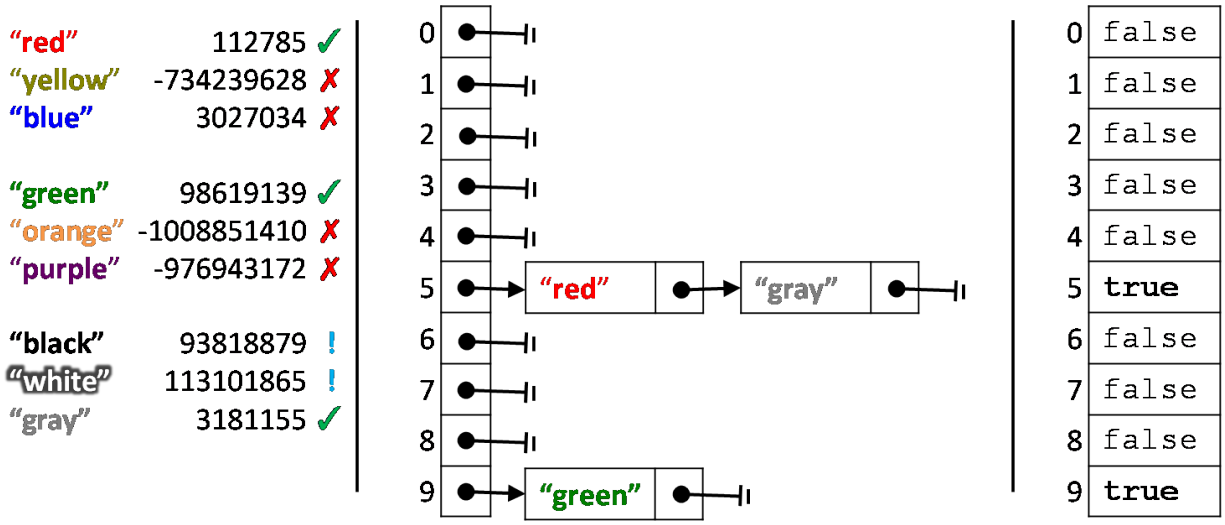


Figure 1: On the left, some strings and their hash values according to the `hash_mul31` hash function.

In the middle, a separate-chaining hash table using this hash function, after the insertion of the strings `"green"`, `"red"`, and `"gray"`. Next to them is an indication of whether the Bloom filter (first implementation) returns a true positive (✔), a false positive (!), or a (true) negative (✘).

On the right, a Bloom filter (first implementation) using this hash function, after the insertion of the strings `"green"`, `"red"`, and `"gray"`.

2 Basic Implementation

The first way we will think about implementing Bloom filters is by taking a regular separate-chaining hash table and getting rid of the chains. Instead of the hash table's main array being a `chain*[]`, we will keep a **`bool`**`[]`. Each index in the Boolean array is `false` if the corresponding hash table bucket is empty, and `true` if the corresponding hash table bucket is non-empty.

In the example from Figure 1, we would give false positives for `"white"` and `"black"`, since those strings collide with strings that are in the set. We would correctly return `false` when asked if other strings, like `"yellow"`, were in the set.

Task 2 (6 points) In the file `bloom1.c0`, implement Bloom filters according to the description above. The type `bloom_t` should be a **`struct`** `bloom_filter*`:

```
struct bloom_filter {
  int capacity;     // capacity < int_max()/4    -- max bool array
  bool[] data;      // capacity == \length(data)
};
```

Note that `capacity` shall be less than `int_max()/4` as C0 won't let you allocate a **`bool`** array that is larger than this.

You must write and correctly use a data structure invariant `is_bloom(B)`. For ease of debugging, you may also want to write a function `print_bloom(B)` that visualizes the internal state of its argument. Calling the constructor `bloom_new(`m`)` should create a Bloom filter whose array has size m. The Bloom filter must use the hash function `hash_mul31` that you implemented in lab, and must compute the hash index from the hash value by modding by the size of the table and then taking the absolute value.

This implementation of the Bloom filter interface uses much less space than an actual hash table. An empty basic Bloom filter with table size m uses one-eighth of the space that the corresponding empty hash table uses. While a hash table has to allocate more space for every element, the basic Bloom filter never allocates any additional space.

3 Better Bloom Filters

In this section, we'll discuss and then implement two improvements to our Bloom filters.

3.1 Multiple Hash Functions

It's inevitable to have collisions in a hash table; we tolerate these collisions because they only make the hash table a little bit slower. In Bloom filters, however, collisions cause us to get false positives. It's worth going to greater lengths to avoid this.

Increasing m, the size of the table, will help some. However, this strategy only takes us so far. Another remarkably effective strategy is implementing *multiple* hash functions, and inserting each item with *every* available hash functions. This means that more of the hash table gets filled up with `true` values (represented as checkmarks in Figure 2). When we test whether an element is in the hash table, then we check all of the indices where that element should hash. If any of them are `false`, we can conclude that the element was never added to the hash table.

The mathematics of why this works better than just growing the array will be a topic for future courses (including Computational Discrete Mathematics, Probability and Computing, and Algorithms). Looking at Figure 2, we see the result of putting our example strings into a Bloom filter. In that figure, the three hash indices we pick are the last three digits of the `hash_mul31` hash values that we saw in Figure 1. In your implementation, you will want to use three entirely different hash functions.

While we still have a false positive for the string `"black"`, the Bloom filter has fewer false positives than before.

Task 3 (3 points) In the file `bloom2.c0`, implement three distinct hash functions, `hash1(s)`, `hash2(s)`, and `hash3(s)`, which take strings and return integers. These will be the three hash functions used by your improved Bloom filter.

You may base your hash functions on any online sources except C0 code you may find on the web. Make sure to cite your sources in comments.

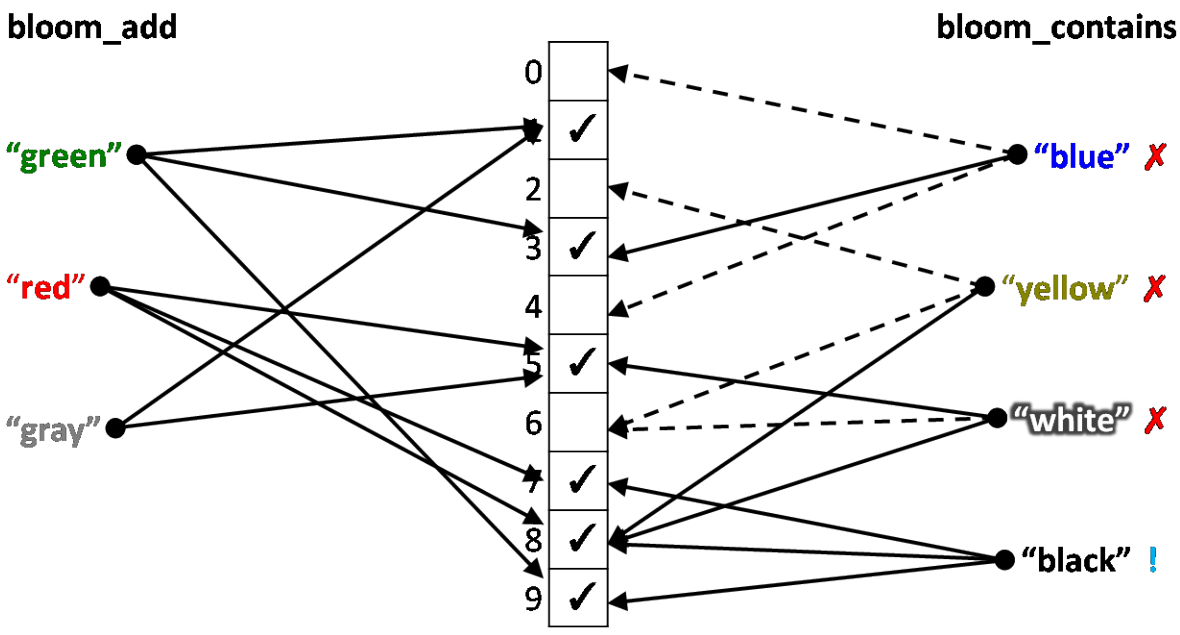


Figure 2: A Bloom filter using multiple hash functions.

You may submit your code to the unofficial autograder (as many times as you want) to get feedback on how good your hash functions are *individually*. Note that good individual performance does not guarantee they will *work well together* in your improved Bloom filter. You also want them to produce hash values that are largely unrelated to each other.

You may also use the collision visualizer you already saw in lab to see how good each hash function is (again, individually). See the file `README.txt` for how to do this.

3.2 Packing Bits

The smallest unit of memory that our computer can efficiently work with is called a *byte*. On most of our modern computers, a value of the **`bool`** type takes up 1 byte, a value of the **`int`** type takes up 4 bytes, and an address (the value of a pointer or an array) takes up 8 bytes. That's why we said that an empty Bloom filter used about one-eighth of the memory that an empty hash table of the same size used.

Recall that our 32-bit integers are made up of 32 bits, each of which are potentially 1 or 0. A **`bool[]`** needs to use m bytes to represent m `true` or `false` values. On the other hand, an **`int[]`** `A` can store 32 `true` or `false` values in the 32 bits of `A[0]`, 32 more `true` or `false` values in the 32 bits of `A[1]`, and so on. An integer array that needs to store m true/false values (bits) needs to only have $\lceil m/32 \rceil$ integers, which takes up $4 \times \lceil m/32 \rceil$ bytes. This is another 8-fold improvement in our memory usage!

Traditionally, we use 0 to represent `false` and 1 to represent `true` in Computer Science.

Task 4 (2 points) In the file `bloom2.c0`, implement two functions which facilitate treating an array of n integers as an array of $32n$ Boolean values:

```
bool get_bit(int[] A, int i)
  /*@requires 0 <= i && i/32 < \length(A); @*/ ;

void set_bit(int[] A, int i)
  /*@requires 0 <= i && i/32 < \length(A); @*/
  /*@ensures get_bit(A, i); @*/ ;
```

Use the provided file `test-pack.c0` to test these functions. Using `get_bit` and `set_bit` should allow you to treat `A` like a `bool[]` that is 32 times longer than `\length(A)`. The function call `get_bit(A,i)` should have the same result that the expression `A[i]` would have for the `bool[]`. The function call `set_bit(A,i)` should have the same result that the statement `A[i] = true;` would have for the `bool[]`.

For a freshly-allocated integer array of all zeros, `get_bit(A,i)` should return `false` for any valid `i`. Subsequent calls to `set_bit(A,i)` turn single bits of the array to `true` (or leave them alone if they are already set).

The exact way that you store 32 `true`/`false` values within an integer is up to you, but it should be similar to the way you stored four 8-bit intensity values in the Pixels assignment. Your approach should be relatively simple and should be documented with comments.

3.3 Implementation

Task 5 (6 points) In the file `bloom2.c0`, implement Bloom filters incorporating the aforementioned improvements. The type `bloom_t` should be a `struct bloom_filter*`:

```
struct bloom_filter {
  int limit;        // limit < int_max()/8    -- max int array
  int[] data;       // limit == \length(data)
};
```

Note that `capacity` shall be less than `int_max()/8` as C0 won't let you allocate an `int` array that is larger than this.

You must write and correctly use a data structure invariant `is_bloom(B)`. For ease of debugging, you may also want to write a function `print_bloom(B)` that visualizes the internal state of its argument. Calling the constructor `bloom_new(`n`)` should create a Bloom filter whose data field is an array of $\lceil n/32 \rceil$ integers. This means that the effective table size will always be a multiple of 32. The effective table size will be $32 \times \lceil n/32 \rceil$, which is between 0 and 31 (inclusive) bits bigger than the requested table size.

Use the three hash functions you implemented in Section 3.1 as your three hash functions. The data structure should only need to access and manipulate the `data` array using the `get_bit` and `set_bit` functions from Section 3.2.

You may submit your code to the unofficial autograder as many times as you want to see how your new Bloom filter implementation compares to your original implementation. This will give you insight on how your three hash functions perform *together*.

4 Using Bloom Filters

Bloom filters come handy in applications where one needs to check whether a value v is present in a given set S but this check is expensive. Here's the idea: first check v against a Bloom filter into which the elements of S have been inserted; if this quick preliminary check returns `false`, we know with confidence that v is not in S; if instead it returns `true`, we need to check v against S itself, which is expensive. We called this idea "quick first passes" earlier.

One such application is making sure that a user does not choose a password that is known to have been compromised. To this end, we will develop a library with the following interface:

```
// typedef ______* pwd_t;

pwd_t pwd_new(int capacity, string pwdfile)
/*@requires capacity > 0;   @*/
/*@ensures \result != NULL; @*/ ;

bool pwd_thoroughcheck(pwd_t B, string s)
/*@requires B != NULL && string_length(s) > 0; @*/ ;

bool pwd_quickcheck(pwd_t B, string s)
/*@requires B != NULL && string_length(s) > 0; @*/ ;

int pwd_check(pwd_t B, string s)
/*@requires B != NULL && string_length(s) > 0; @*/
/*@ensures 0 <= \result && \result <= 2;       @*/ ;

void pwd_stats(pwd_t B)
/*@requires B != NULL; @*/ ;
```

The function `pwd_new` populates a new password database (of type `pwd_t`) with compromised passwords from file `pwdfile`. This bad password database consists of a Bloom filter of capacity `capacity`, of the actual set of password in `pwdfile`, and of some statistic information (discussed below). The function `pwd_stats` prints these statistics.

The function `pwd_quickcheck` returns whether `s` *might be* a compromised password in `B`: if it returns `false`, `s` is definitely not compromised, but if it returns `true` it may or may not be compromised. The function `pwd_thoroughcheck` returns whether `s` *actually is* a compromised password according to `B`.

The function `pwd_check` also checks whether `s` is a compromised password, but it does so in a smart way (by taking advantage of Bloom filters). It returns `0` if `s` is compromised, `1` if it is uncompromised but a thorough check was necessary to establish this, and `2` if it is uncompromised and a quick check was sufficient to conclude this. This function updates the statistic information in `B`. In an actual password setting application, the function `pwd_check` would be called each time a user attempts to set a new password: if the password is found to be compromised, it would be rejected and the user would be prompted to choose a different password.

This interface has been partially implemented for you in file `pwd.c0`. In it, the concrete type `pwd` of bad password databases is defined as follows:

```
struct pwd_header {
  bloom_t iffy;    // != NULL
  string bad;      // string_length(bad) > 0;
  int checks;      // == truepos + trueneg + falsepos
  int truepos;     // 0 <= truepos  <= checks
  int trueneg;     // 0 <= trueneg  <= checks
  int falsepos;    // 0 <= falsepos <= checks
};
typedef struct pwd_header pwd;
```

Besides the underlying Bloom filter (field `iffy`), it records the name (`bad`) of the compromised password file, the number of times `pwd_check` was called (`checks`) as well as the number of times a call to `pwd_check` resulted in a true positive, a true negative and a false positive (`truepos`, `trueneg` and `falsepos`, respectively).

The functions `pwd_thoroughcheck`, `pwd_stats` and the data structure invariant function `is_pwd` have been implemented for you in full. The function `pwd_new` has been partially implemented.

Task 6 (2 points) Complete the implementation of the function `pwd_new` and write the functions `pwd_quickcheck` and `pwd_check`. Make sure to update the statistic fields correctly in the latter.

You may use the file `pwd-test.c0` to run some tests. The file `data/20-badpwd.txt` contains the 20 most common passwords of 2021 (and yes, `"password"` is one of them). Feel free to use other common password files or to make up your own.