Advanced Analysis of Algorithms
Fall 2018
Assignment 5
Submission Date: Dec 10, 2018

This is a classic and simple application of a bloom filter. Suppose you are developing a web browser and you have a collection of malicious domains gathered by various resources in a text file. Whenever the user of this browser enters a URL in the address bar, the browser restrict the user from opening the desired web page if it is included in the collection of malicious domains. In order to save space, you will store these domains/URL in a bloom filter and use this filter to identify whether the given URL is malicious or not. Your program will store all the URLs in a bloom filter from the collection first (you are given the collection along with this statement) and then ask the user to enter a URL. If the URL entered by user is in the bloom filter then tell the user its malicious website.

Specs for the Bloom Filter

Let's say that m is the number of bits in the bloom filter B. There are n objects (URL in the collection) to be stored. We will store b bits per object, which means, m=b*n bits in total. (this b=m/n will be a parameter provided by the user at the start of the program and it must be a positive integer in the range 4 - 10)

We derived the following formula in class to compute the probability of getting a false positive in our bloom filter:

$$\Pr[\text{false positive}] <= (1 - e^{-kn/m})^k$$

This quantity is minimized for k = ln2*m/n. So, compute the optimal value of k based on the input parameter b. Which means picking k functions $h_a(x)$ randomly from family of hash functions, H, each one with a random value of **a** chosen from {1,2,…,p-1} where p is a large prime number.

In your program, $p = 2^{23} - 3 = 16777213$ (incidentally, this is a prime number, and also very close to a power of two). Also note that this number will easily fit in a regular 32 bit integer.

How will your program store a password in a bloom filter?

Each URL is first converted into a number by using a polynomial of degree equal to the length of the URL (number of characters) minus 1, where each co-efficient in the polynomial will be a letter in the word and then take modulus with p and m. Following is an example.
: x="navigator", ASCIIs: N=110, a=97, v=118, i=105, g=103, a=97, t=116, o=111, and r=114
So
$P_a(x) = 110a^8+97a^7+118a^6+105a^5+103a^4+97a^3+116a^2+111a+114$
$h_a(x) = (P_a(x) \bmod p) \bmod m$

You should remember that **a** is randomly picked from {1,2,…p-1}, and determines the working of the hash function. By picking a random **a** we basically pick a random hash function from the family:
$H=\{ h_a(x) | a \in \{1,2,..,p-1\}\}$

We shall pick k random hash functions (k random values of a) at the start of the program, and use all k hash functions on each password x, and set all corresponding bits in our bloom filter to 1.

Notes on implementation

Make a BitVector class to perform operations on bits and use its object as member inside the BloomFilter class.

Input the value of **b** and compute the optimal value for **k**

Generate k random values of **a** from the set {1,2,…,p-1}, once at the start of the program, and use the k corresponding $h_a(x)$ functions to insert and lookup.

While computing the value of $P_a(x)$ as describe above, you **should not** compute the entire $P_a(x)$ first and then take remainder with p. Instead, you should take remainder after each single summation. Use the fact that:

(a+b) % p = (a%p + b%p) % p

By taking %p separately of each term and then each sum, you will keep all numbers always smaller than p and within the integer bounds. You will take %m of the final answer in the function $h_a(x)$.

Even more importantly, **you may have to write your own power function.** The number **a** will be large, maybe a 23 bit number, and computing its third or higher power will result in integer overflow, because 32 bit integers cannot store 69 bit numbers! To prevent against overflow, you must write our own power function and use the fact that:

(a * b) %p = (a %p * b%p) % p

In the case of case b=a, so after each multiplication of **a** with itself, the number will be brought back % p

Generation of random number:

We know that our prime number is a 23 bit number. Therefore, we will generate 23 random bits using rand()%2; and this will give us a random 23 bit number with uniform probability for each number between 0 and 2^{23}-1

4) Program Interface

Ask the value of b and load the collection of URLs and show progress of loading.
Ask the user for a URL and if URL is malicious, prompt the user to enter another password.
repeat this until a valid URL is entered