Sabancı University
Faculty of Engineering and Natural Sciences

CS204 Advanced Programming

Fall 2023-2024

Homework #6

Due: 11/01/2024 - 23:55

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

You HAVE TO write down the code on your own.
You CANNOT HELP any friend while coding.
Plagiarism will not be tolerated!!

1 Introduction

In this homework, you will implement probabilistic data structures, i.e., **sketches**, by using bit operations and polymorphism in combination. in particular, you will implement HyperLogLog, KMinVal and Bloom Filter sketches and test them with varying parameters, i.e., sizes and numbers of hash functions, to efficiently count and query a large number of numbers of strings using low memory.

2 Cardinality estimation problem and sketches

Imagine you're dealing with a **stream** of items. Some of these items are repeated, and all are coming from an extremely large **universal set** - whose size can be infinite. Your first task is to estimate n, the number of unique items in the stream, i.e., **stream cardinality**. To illustrate, think about trying to determine how many distinct individuals visit Facebook in a given week, where each person logs in multiple times a day. Here the universal set is all people in the world, the stream is people's login data (with repetitions), and you are interested in the number of unique users. Section 2 introduces some approaches for this cardinality estimation problem, and Section 3 does the same for another problem; set membership.

Finding the number of unique items is not easy; traditional approaches, such as sorting the items or keeping track of the unique items encountered, are not feasible due to their excessive computational demands or memory requirements. Note that you do not know the number of possible items, i.e., the universal set's cardinality, and even if you know it, it is extremely large. To tackle this challenge, you'll implement a series of data structures and algorithms that **estimates** the correct result with much less computational demand. Since it is an estimation, it will not be accurate, but it should fall within an acceptable range. To begin, we create a hypothetical stream having n unique items but also repetitive entries by following these steps:

- Generate n random integers from an arbitrary distribution.
- Randomly duplicate some of these numbers a variable number of times.
- Shuffle the resulting dataset.

Let's do some observations; we can identify the smallest value (x_min) within the dataset and estimate the count of unique entries as

$$\frac{\text{universal set cardinality}}{x\_min} - 1.$$

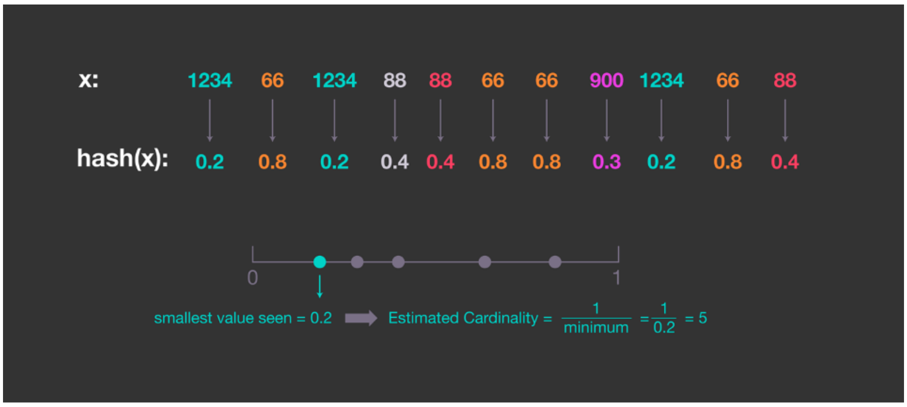
However, since the universal set cardinality is infinite, and the original numbers are distributed arbitrarily (see the first step above), to ensure that the values we deal with are evenly distributed within a limited interval, we will employ a **hash function** and estimate the stream cardinality based on the hashed values rather than the actual entries themselves. You can consider a hash function as a **deterministic, non-random** function generating uniformly distributed numbers. It is deterministic since for the same input, which can be an integer, double, string, any object etc., a hash function outputs the same number. In the homework, the items will be words/strings. A basic hash function family implementation, which you can also use in your HW, is given below:

```
uint64_t hash_func(uint64_t a, uint64_t b, string to_hash) {
    uint64_t hash = b;
    for (char c : to_hash) {
        hash = hash * a + static_cast<uint64_t>(c);
    }
    return (uint64_t) hash;
}
```

When you change a and b, you have a new hash function. In the HW, you will need at most 3 hash functions. You can use the following values:

```
uint64_t a1 = 97102761753631939, b1 = 42506983374872291;
uint64_t a2 = 56842397421741207, b2 = 18456721873196387;
uint64_t a3 = 61811894731001917, b3 = 37217717671130671;
uint64_t a4 = 31415926535897931, b4 = 27182818284590453;
uint64_t a5 = 98765432109876543, b5 = 57548533468232177;
```

The figure below provides a straightforward example illustrating the approach described above, where the hashed values are normalized (by `uint64`$_t$`(-1)`) and exhibit uniform distribution within the range of 0 to 1.



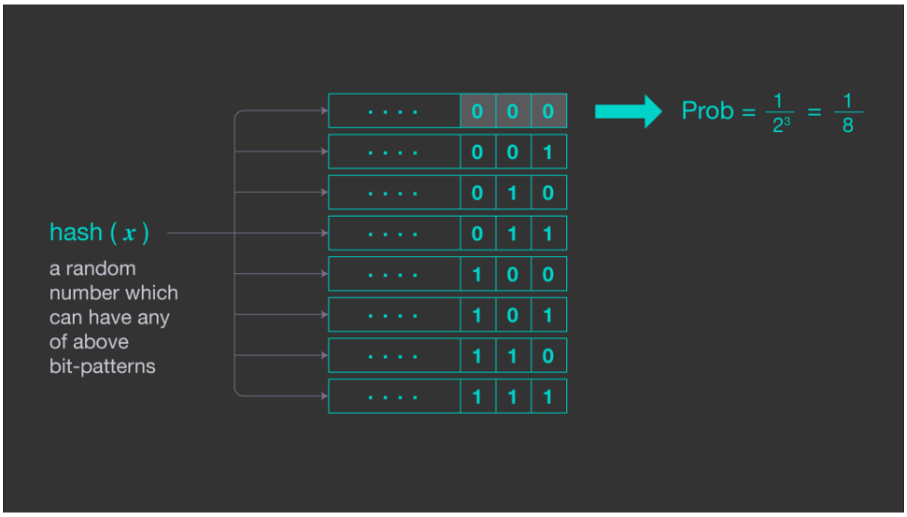
The best thing about this approach, MinVal, is that you need to store only a single integer while going over the stream elements. However, extremely small hash values, which are rare, make MinVal drastically overestimate the cardinality n.

2.1 The first sketch: HyperLogLog

We will come back to the approach above later. But now let's talk about another approach, HyperLogLog in which, unlike MinVal, we will employ the binary representation of hash values. HyperLogLog counts the number of consecutive zero bits at the end of the hashes. This sounds because the probability that a given $hash(x)$ ends with at least ℓ zeros is $1/2^{\ell}$. In other words, on average, a sequence of ℓ consecutive zeros is expected to occur once in the hash values of every 2^{ℓ} distinct entries/items. The figure below illustrates an example of the probability of observing a sequence of three consecutive zeros.

To estimate the number of distinct elements using this pattern, all we need to do is record the length of the longest sequence of consecutive zeros. Mathematically speaking, for a stream $\{x_1, x_2, ..., x_M\}$ with M elements (**with replication**) and n distinct elements, if we denote $\rho(x_i)$ as the number of consecutive zeros in $\text{hash}(x_i)$ **plus one**, a good estimation on n is 2^R, where $R = \max(\rho(x_1), \rho(x_2), ..., \rho(x_M))$.

HyperLogLog is amazing from one aspect; you only store a single number, R, and estimate the cardinality of a stream having billions, trillions of items. However, the estimation over n can still be extremely off due to large ρ values, i.e., the number of consecutive zeros plus one, which is expected to be rare. That is even one entry



whose hash value has too many consecutive zeros yields a drastically inaccurate (overestimated) cardinality. To improve the method, HyperLogLog stores many estimators instead of one and averages the results.

Formally, HyperLogLog uses m buckets, i.e., R values. It again does it with a single hash function but uses part of the binary hash output to split the items into these m buckets. To find which of the m buckets will be used, the first few ($log_2(m)$) bits of the hash value are used to find the bucket index (i.e., the R to be updated) and the rest of the hash is used to compute the length of the sequence having consecutive 0s (**plus one**).

Assume the hash of our incoming item x looks like $hash(x) = 1011011101101100000$ and $m = 16$. Let's use the $log_2(16) = 4$ leftmost bits to find the bucket index. These 4-bits are colored: 1011011101101100000, which gives us the 11th bucket to update (1011 $=$ 11 in decimal). From the remaining, 1011011101101100000, we obtain the length of the consecutive 0s from the rightmost bits, which, in this case, is 5. Plus one is 6. So the 11th R value is replaced with 6 if it is smaller than 6.

This technique, i.e., having m buckets and averaging the estimation, is called *stochastic averaging*. Using this technique, HyperLogLog improves the error by employing the harmonic mean of these values in its estimation with no increase in required storage. Put all these pieces together and we get the HyperLogLog (HLL) estimator for the count of distinct values.

$$\text{CARDINALITY}_{\text{HLL}} = \alpha_m \cdot m \cdot \frac{m}{\sum_{j=1}^{m} 2^{-R_j}}$$

where α_m is a constant equal to:

$$\alpha_m \approx \begin{cases} 0.673, & \text{for } m = 16 \\ 0.697, & \text{for } m = 32 \\ 0.709, & \text{for } m = 64 \\ \frac{0.7213}{1+1.079/m}, & \text{for } m \geq 128 \end{cases}$$

2.2 k-MinVal Sketch

Let's go back to the first approach, MinVal: as mentioned before, only storing the first minimum value can yield extremely off estimates since this value can be small. However, to reduce the error, we can store the smallest k hash values as the following k-MinVal recipe describes:

Step 1: Initialization

You start by initializing a k-MinVal sketch which contains an array of `uint64_t` to store the k minimum hash values found so far. Assume that $k = 3$ (i.e., we keep 3 minimum values)

$$[-1, -1, -1]$$

(-1 denotes that the corresponding entry is initially empty).

Step 2: Hashing Data Elements

For each data element in your stream, apply the hash function and calculate its hash value.
`Example Data Stream: ["apple", "banana", "cherry", "date", "elderberry"]`

Hash Values:

$$apple- > 112$$

$$banana- > 90$$

$$cherry- > 150$$

$$date- > 75$$

$$elderberry- > 130$$

Step 3: Updating the Minimum Values Array

After computing each hash, update the array to maintain only the 'k' smallest unique hash values.

`Add hash of "apple":`	$[112, -1, -1]$
`Add hash of "banana":`	$[90, 112, -1]$
`Add hash of "cherry":`	$[90, 112, 150]$
`Add hash of "date":`	$[75, 90, 112]$
`Add hash of "elderberry":`	$[75, 90, 112]$ (130 is not in the top 3 minimum values).

Step 4: Estimating the Number of Distinct Elements

Once all elements have been processed, use the values in the array to estimate the total number of distinct elements in the data stream. The formula often used is:

$$\text{D}istinctCount \approx \frac{(k-1) \times \text{ Maximum possible hash value}}{k\text{'th minimum hash value}}$$

- k'th minimum hash value = 112 (last value in the array)

- Maximum possible hash value: (depends on hash function implementation - you need to use the appropriate value from `climits`)

- Distinct Count Estimation: Assuming the maximum possible hash value = $2^8 - 1$ (8-bit unsigned hash values are assumed for demonstration)

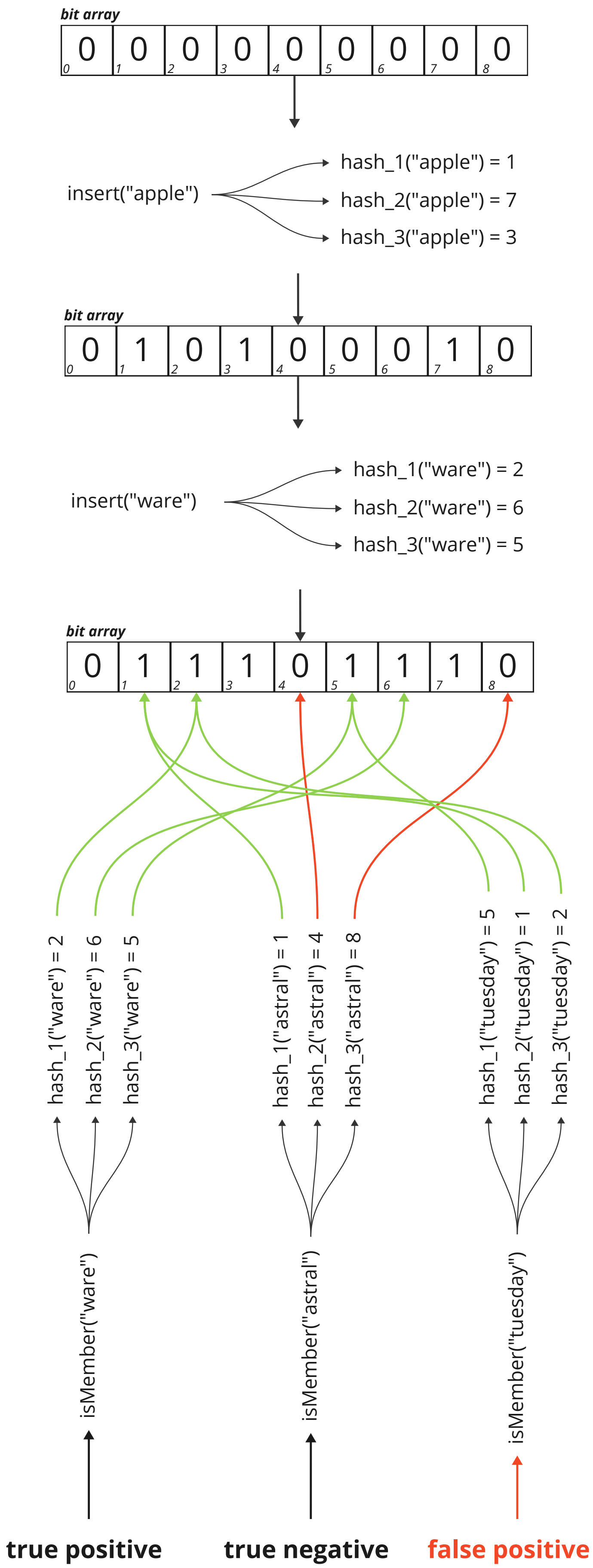
$$DistinctCount = (3-1) * (2^8 - 1)/112 \approx 4.55$$

3 Bloom Filter

A Bloom Filter is a probabilistic data structure used for testing whether an element is a member of a set or not. It is particularly efficient in terms of space when compared to other data structures like hash tables or sets, but it comes with a trade-off in terms of accuracy. A Bloom Filter may sometimes produce **false positives**, meaning it might mistakenly claim that an element is in the set when it's not, but it never produces **false negatives** (if it says an element is not in the set, it is *definitely* not in the set). For the homework, your Bloom Filter class should include a bit array that is made up of `uint64_t` variables and m independent hash functions. Usage of Bloom Filter is as follows;

Step 1: Initialization

- Allocate memory for $size/64$ 64-bit unsigned integers where $size$, i.e., the number of bits the sketch will use, is given as a parameter during construction. You need to set all these bits to 0.


bit array
0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8
insert("apple")
hash_1("apple") = 1
hash_2("apple") = 7
hash_3("apple") = 3
bit array
0 1 0 1 0 0 0 1 0
0 1 2 3 4 5 6 7 8
insert("ware")
hash_1("ware") = 2
hash_2("ware") = 6
hash_3("ware") = 5
bit array
0 1 1 1 0 1 1 1 0
0 1 2 3 4 5 6 7 8
hash_1("ware") = 2
hash_2("ware") = 6
hash_3("ware") = 5
hash_1("astral") = 1
hash_2("astral") = 4
hash_3("astral") = 8
hash_1("tuesday") = 5
hash_2("tuesday") = 1
hash_3("tuesday") = 2
isMember("ware")
isMember("astral")
isMember("tuesday")
true positive
true negative
false positive

- Prepare $k = 3$ hash functions - use the a, b values above. You can put these pairs in an array to loop over them (i.e., to loop over the hash functions).

Step 2: Insertion

- To insert an item, acquire $k = 3$ hash values from independent hash functions (i.e., using different a, b pairs).
- For each hash value, compute the index of the bit to be modified by taking the modulo of the hash value w.r.t. size (i.e., the number of bits).
- Set the corresponding bits of the Bloom Filter (to 1) where the bits' indices are obtained from the above item.

Step 3: Membership Query

To determine if an item is previously inserted into the bloom filter, simply act similar to Step 2.

- To query an item, acquire $k = 3$ hash values from independent hash functions (i.e., using different a, b pairs).
- For each hash value, compute the index of the bit to be modified by taking the modulo of the hash value w.r.t. size (i.e., the number of bits).
- Check the corresponding bits of Bloom Filter where the bits' indices are obtained from the above item.
- If all checked bits are 1 then return **true**. Note that the answer can be wrong.
- If any of the checked bits are 0 then return **false**. This item is **definitely** not inserted into the filter before.

In the HW, you will use 3 hash functions (check the hash function implementation given above and three possible a, b pairs) to test the accuracy of the Bloom Filter sketch with different sizes on membership queries.

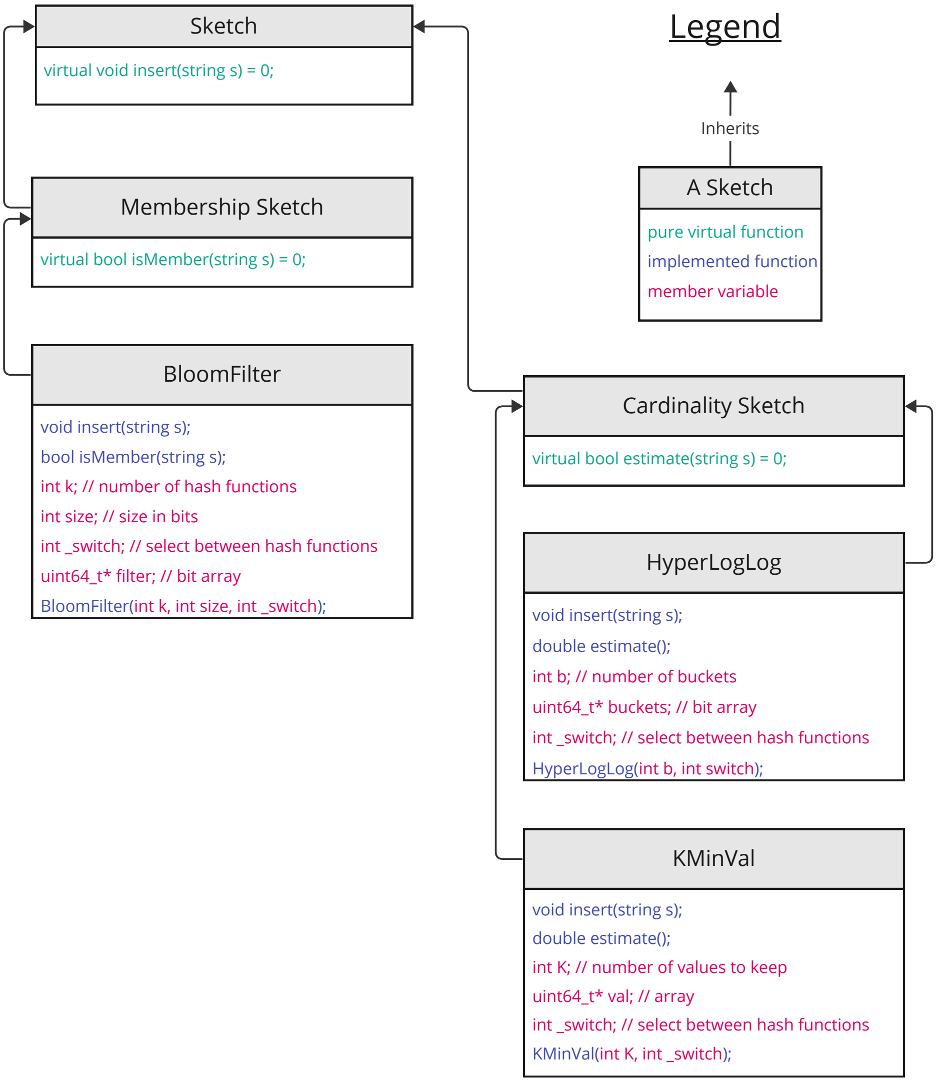
4 Class Structure

You need to employ polymorphism for the implementation of your classes. The class hierarchy is given in the figure below. In short, you need to implement;

- An abstract class named `Sketch`. This is the ancestor class of all other classes.
- An abstract class named `MembershipSketch` that inherits `Sketch` class.
- A concrete class named `BloomFilter` that inherits `MembershipSketch` class. `BloomFilter` needs to implement `void insert(string)` and `bool isMember(string)` functions.
- An abstract class named `CardinalitySketch` that inherits `Sketch` class.
- A concrete class named `HyperLogLog` that inherits `CardinalitySketch`. It needs to implement `void insert(string)` and `double estimate()` functions.
- A concrete class named `KMinVal` that inherits `CardinalitySketch`. It needs to implement `void insert(string)` and `double estimate()` functions as well.

5 The Quality of the Hash Function

The previous hash function we have seen, $ax + b$, is a widely used simple hash function. There are various hash functions in the literature and they differ in terms of their purpose and quality. A good hash function should possess properties like determinism, efficiency, uniform distribution, avalanche effect, collision resistance, deterministic output size etc. MurmurHash is a widely used hash function with notable computational advantages. It is designed for speed and efficiency, making it a preferred choice in various computing applications. MurmurHash is recognized for its capacity to provide uniform distribution of hash values. This means that given a diverse range of input data, the resulting hash values are evenly spread across the possible hash space,


Sketch
virtual void insert(string s) = 0;
Membership Sketch
virtual bool isMember(string s) = 0;
BloomFilter
void insert(string s);
bool isMember(string s);
int k; // number of hash functions
int size; // size in bits
int _switch; // select between hash functions
uint64_t* filter; // bit array
BloomFilter(int k, int size, int _switch);
Legend
Inherits
A Sketch
pure virtual function
implemented function
member variable
Cardinality Sketch
virtual bool estimate(string s) = 0;
HyperLogLog
void insert(string s);
double estimate();
int b; // number of buckets
uint64_t* buckets; // bit array
int _switch; // select between hash functions
HyperLogLog(int b, int switch);
KMinVal
void insert(string s);
double estimate();
int K; // number of values to keep
uint64_t* val; // array
int _switch; // select between hash functions
KMinVal(int K, int _switch);

reducing the likelihood of collisions. Uniform distribution is a crucial property in hash functions as it ensures that different inputs produce distinct hash values, enhancing the overall reliability and effectiveness of data structures like hash tables, dictionaries and data sketches.

When comparing the $ax + b$ string hashing and MurmurHash, it's important to note that MurmurHash is generally designed to offer better quality in terms of distribution, and avalanche effect. MurmurHash aims to minimize collisions and provide even distribution, making it suitable for general-purpose hashing tasks, whereas $ax + b$ hashing may vary in quality depending on the specific parameters chosen, potentially leading to less consistent distribution and potentially weaker security properties. Therefore, when prioritizing hash quality, MurmurHash is often a better choice for a wide range of applications.

Source code for both of these hash functions is provided to you with the homework. Your implementations should be able to employ both of them and choose between the hash functions during construction. Then, you need to compare performance of both of those functions.

6 Some Remarks

- First use the hash function implementation and the parameters given in Section 2 with parameters a and b. Then do the same with the Murmur hash described in the previous section.

- You can't use `std::bitset` or `std::vector` for bitwise operations. You must dynamically allocate memory with an array of `uint64_t` type and implement bitwise operations yourself. Let's say you have a Bloom Filter that is 200 bits in size. To be able to perform operations efficiently, you need to operate on individual bits of `uint64_t` integers. For example, to flip the 112^{th} bit of the filter, you need to flip the

48^{th} bit of the second `uint64_t` variable in your array.

- The given class hierarchy, functions, and member variables are required for a full grade. You can implement other helper functions and structures inside/outside of the classes. You can add hash functions as member variables to classes.
- Your estimations should be at least in the same order of magnitude with the correct numbers. You need to report your errors, false positive rates etc. in your experiments.
- A main function is given to you in `sketches.cpp`. In this file, you can find hash functions and driver code to perform experiments. You must build your code on top of this file. If your classes comply with the given code, you will get the requested output files easily.
- Submission guidelines are changing according to requirements for each homework. Please read again, and submit accordingly.

7 Deliverables

With this homework, a dataset to test your implementations is given to you. In the dataset naming use the pattern X_total_Y_unique.txt. It means that there are a total of X values in the file and Y of them are unique. If $X = Y$, it means that there are no duplicate items in the file. You can use those files to test your sketches.

For this homework, you need to submit your code along with a report. In your report, explain the details of your program, e.g. how you implemented the hash functions, which parameters you chose for different sketches, and your implementation's error rates for different sizes. Moreover, for every file, submit a corresponding file that contains the output of your program in the format, `output_file_name`. For example, if the input file name is `1000_total_500_unique.txt`, the name of your output file should be `output_1000_total_500_unique.txt`. You need to run your program with every input file provided with the homework and submit your output files together with your code and report.

8 Some Important Rules

In order to get full credit, your programs must be efficient and well presented, the presence of any redundant computation, bad indentation, or missing, irrelevant comments is going to decrease your grades. You also have to use understandable identifier names, informative introductions and prompts. Modularity is also important; you have to use functions wherever needed and appropriate.

What and where to submit (PLEASE READ, IMPORTANT): We will use the standard C++ compiler and libraries while testing your homework. It'd be a good idea to write your name and last name in the program (as a comment line of course).

Submission guidelines are below. Some parts of the grading process are automatic. Students are expected to strictly follow these guidelines in order to have a smooth grading process. If you do not follow these guidelines, depending on the severity of the problem created during the grading process, 5 or more penalty points are to be deducted from the grade.

Name your cpp file that contains your program as follows:

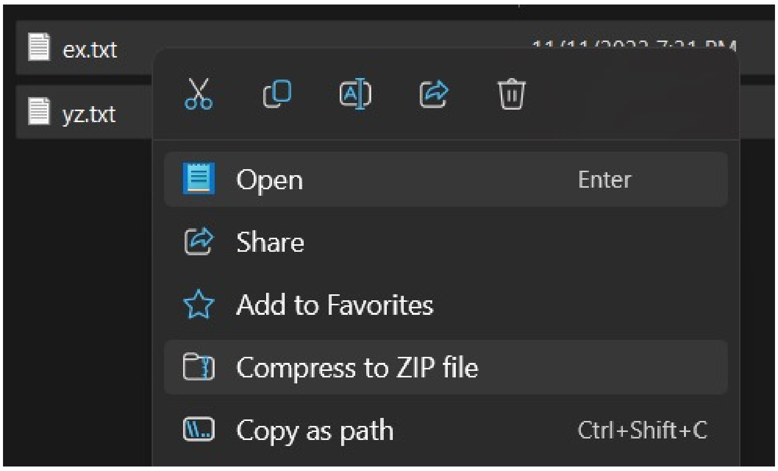
sucourseusername.cpp

Your SUCourse username is actually your SUNet username that is used for checking sabanciuniv e-mails. Do NOT use any spaces, non-ASCII and Turkish characters in the file name. For example, if your name is Alexander Zeus and your username is azeus; Then your cpp file should be named as;

azeus.cpp

Then compress your **files** (you should submit a single cpp for this homework) in a **.zip** file named the same as your cpp file and submit this zip file to sucourse. If your name is Alexander Zeus and your username is azeus, then your zip file should be named

azeus.zip



Please **don't compress folders** into a zip file. When your zip file is opened, it should not expand into folders within folders. Make sure you compressed your files as follows:
Submit via SUCourse ONLY! You will receive no credits if you submit by other means (e-mail, paper, etc.).

Good Luck!

CS204 Team (Fatih Taşyaran)

9 References

Facebook Engineering. *(2018, December 13). HyperLogLog: the analysis of a near-optimal cardinality estimator algorithm. Facebook Engineering.* https://engineering.fb.com/2018/12/13/data-infrastructure/hyperloglog/