Optimizing DFS on Large Graphs using Bloom Filters & False +ve Caching

Set Membership Problem in Computer Science

- Although we all know the various ways to represent a set, like the Roster Form or the Set Builder Form we studied in the field of Discrete Mathematics, storing sets in the computer's memory, in a way that its elements can be both inserted and accessed efficiently has proved to be very challenging due to the fundamental nature of tradeoffs between Space/Time Complexity. But the another often neglected factor while weighing such tradeoffs is Accuracy.

- We will introduce you to a Probabilistic Data Structure called a Bloom Filter and see the Discrete Math behind tuning their Accuracy and compare how they improve upon traditional non probabilistic, discrete Algorithms like Depth First Search on very large graphs. Towards the end we will draw some insights from the mathematical analysis to discuss optimization using caching.

Representing a Set in the Computer’s Memory

- To perform membership test on a set in the computer’s memory, it’s necessary to understand how a set is represented in the computer’s memory in the first place. Based on which we can think of ways to perform the Set Membership test in an optimal way.

- The drawbacks of these approaches become clearer as the cardinality of the set grows and tends to a very large number*. So we ask you to assume that we are dealing with sets of such large cardinalities for the sake of explanation and analysing worst case scenarios and bottlenecks.

*Not ∞ as we are dealing with finite memory, ≅ 2^{64}-1 which is the maximum value an unsigned long long int can hold on 64 bit CPUs.

Representing a Set as a Linear Array

- The naivest and least optimal way to represent a set in a computer's memory is by using an array because in the worst case we cannot conclude if an element belongs to a set till we reach and read the last element of the array.

- So the time taken by the Set Membership Test on a set represented as an array depends linearly on the cardinality of the set(N), leading to a O(N) Time and Space Complexity.
    - Insertion is an O(N) Operation too as we need to perform a set membership test prior to insertion to avoid inserting duplicate members in the set!

- At the end we can either be 0% or 100% sure if the element exists in the set.

Representing a Set as a Balanced Binary Tree

- A more optimal way to represent a set in a computer's memory is by using a balanced binary tree because in the worst case we can conclude that an element doesn't belongs to a set if a node containing the same value doesn't exists in the tree in O(log N) Time.*

- Insertion is an O(log n) operation too as maintaining the balanced binary tree invariant and performing set membership test to inserting duplicate members in the set takes O(log n) time.

- At the end we can either be 0% or 100% sure if the element exists in the set.

*Inductive Proof: https://www.cs.cornell.edu/courses/cs211/2006sp/Lectures/L06-Induction/binary_search.html

Representing a Set as a Balanced Binary Tree

- Although O(log n) is relatively small even for large numbers, the devil lies in the inefficiencies of storing Trees in a computer's memory which is exacerbated when the entire set needs to be loaded into the RAM.
- Assuming that a the node of the balanced binary tree contains left and right pointers on a 64-Bit CPU Architecture, we can say that the smallest and the largest possible memory address will be 64 bits(8 bytes) long. So to store a 4 byte integer, we will have to spare 8+8=16 bytes for pointers to child nodes.
    - For a large set, the metadata will be 4 times the size of the data itself, which is unacceptable.
    - So we are trading space compactness to improve the Time Complexity from O(N) to O(log N).
    - Although we can represent the tree in an array, it will implicitly set an upper bound to the set's cardinality, hence leading to Amortized O(log N) Time Complexity.*

*As once in a while an insertion will be O(N) instead of O(log N) due to overhead of copying all existing elements into the larger array.

Understanding Google's Bottleneck

- There are around 2 Billion Gmail IDs created till date, with a conservative estimate of each ID being 10 characters long where each character occupies a single byte in the computer's memory, to store all 2 billion email ids using a set implemented using a tree will require 20 Billion Bytes of data and 80 Billion Bytes of metadata even though there is no useful relationship between different gmail IDs(nodes).

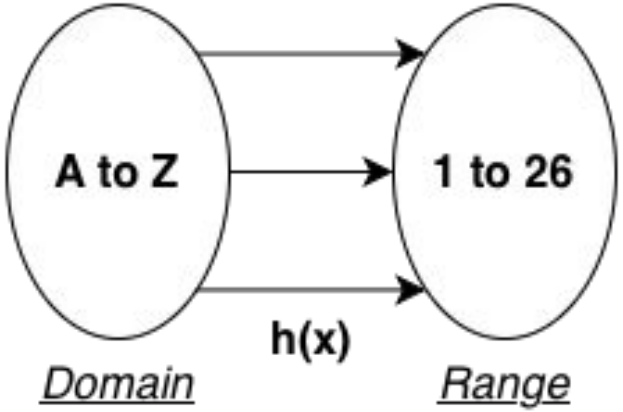
- If a user wants to create a new ID, then Google verifies if the ID is already taken after every single keystroke! For such large datasets under such severe time constraints logarithmic time complexity can still fall short of expectation when it comes to testing for set membership of the user input in the gmail database(set of all IDs created till date, N=2 Billion) after every keystroke in a blink of an eye.

Hashing: Optimizing Time at the cost of Space

- A hash function allows us to generate a mapping between a larger domain set and a smaller range set by bounding its result using the modulus operator %

- Suppose the domain is the set of all alphabets from A-Z and we want to create a set of alphabets that can store it's subset(range set). So we create an array of size 26+1 and use a hash function h(x) to generate a mapping between the domain and range such that:

$$h(x) = \begin{cases} 1, & \text{if } x='A' \\ 2, & \text{if } x='B' \\ 3, & \text{if } x='C' \\ \vdots & \\ 24, & \text{if } x='X' \\ 25, & \text{if } x='Y' \\ 26, & \text{if } x='Z' \end{cases}$$



Hashing: Optimizing Time at the cost of Space

- Now, to check the membership of any alphabet stored in variable char x, we again use the same hash function h(x) which tells us which index of the array needs to be checked, if the character stored at that index is same as x then we can conclude that x is a member of the set instead of having to read each element of the underlying linear array sequentially.
- The drawback to this approach is the wastage of space in the provisioned array, as the array should have a size equal to the cardinality N of the domain set regardless of the cardinality of the range set. So we get a O(1) time lookup a the cost of O(N) Space Complexity.
- A good hash function should ensure uniform distribution throughout the array. Still if the cardinality of the domain set is ∞(i.e. an infinite set) and the cardinality of the set to be stored is N such that N<<<∞, then **By pigeon hole principle, number of pigeons=∞ but number of pigeonholes=N. So there must be more than one elements from domain set that can be mapped to the same index by h(x).**

Hashing: Optimizing Time at the cost of Space

- Such a situation is also called a Collision. **In the previous example the domain set was a set of all alphabets so the cardinality of the domain was constant = 26. Hence we were able to avoid collisions due to a bijective hash function.**

- Now let's assume h(x) accepts a string and returns the (length of x)%10 that allows us to store the string x in the array of size 10 at that index.
- To insert the CAT in the null set, we store it in the 3rd index of the array of size 10.

- Now h(x) will map any 3 letter word like DOG, RAT, etc. to the same 3rd index.
  - In fact all lengths with the digit 3 in the units place will be mapped to the 3rd index!
  - If CAT is stored at the 3rd index and we are asked if DOG is a member of the set then we can read the 3rd index and say that it doesn't store the word DOG inside it so DOG isn't a member of the set if DOG was never inserted after inserting CAT.
  - Inserting words like DOG, RAT would lead to collision and require resolution methods.
  - At the end we can either be 0% or 100% sure if the element exists in the set.

Hashing: Optimizing Time at the cost of Space

- We can use collision resolution methods and multiple hash functions to reduce the number of collisions but cannot entirely avoid it for infinite domain sets.

- But we will still require to store the entire object in the indices of the array.
  - This is the reason behind the atomic nature of the accuracy(assuming no collisions).

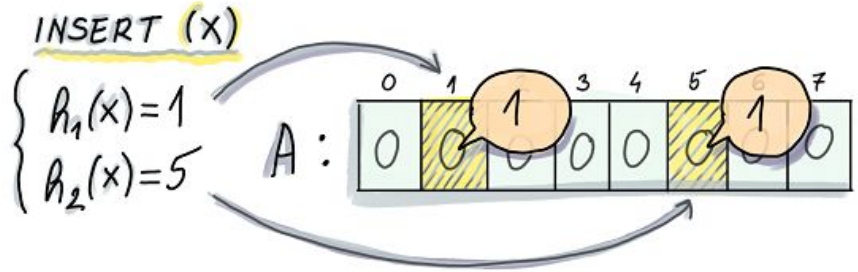
- Bloom filters don't store the entire object in the indices of their underlying array, which drops the atomic nature of the accuracy, making the data structure probabilistic instead of deterministic thereby providing exceptional space complexity and compactness without coming at the cost of time complexity.

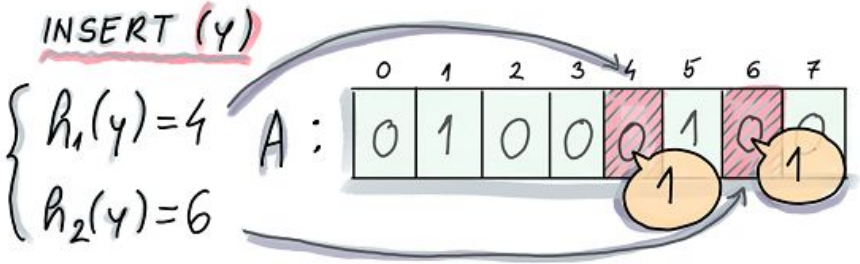
- This also exposes several variables that can be tuned to set the expected accuracy of the bloom filter.

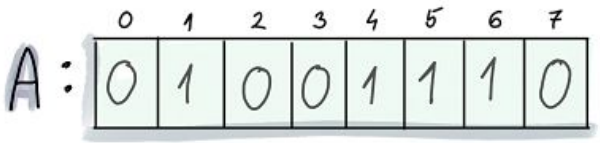
Bloom Filters: Optimizing Space at the cost of Accuracy

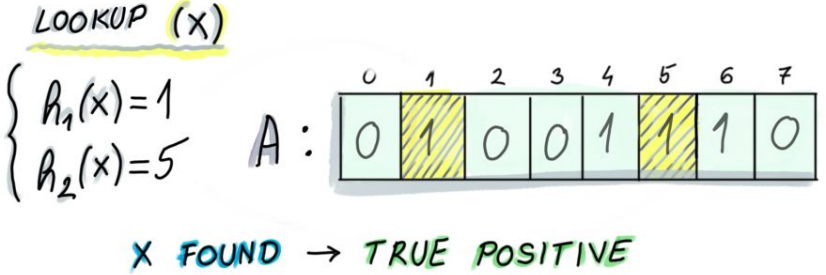
- So what exactly are we storing in the bloom filter if not the elements themselves?
- Well the bloom filter is a very large array of bits(0s & 1s) where a bit is set to 1 acts as a beacon to mark the probable existence of an element. While the 0s mark the guaranteed absence of an element from the set.
- It's cheaper to create such a large sized array of bits as each index can either store a 0 or a 1 which is 1 bit long, compared to an array of integers where every index is 32 bits(4 bytes) long.
- The size of a fundamental block that can be wasted is reduced from 32 bits to 1 bit.
- The accuracy can be improved by using multiple hash functions.

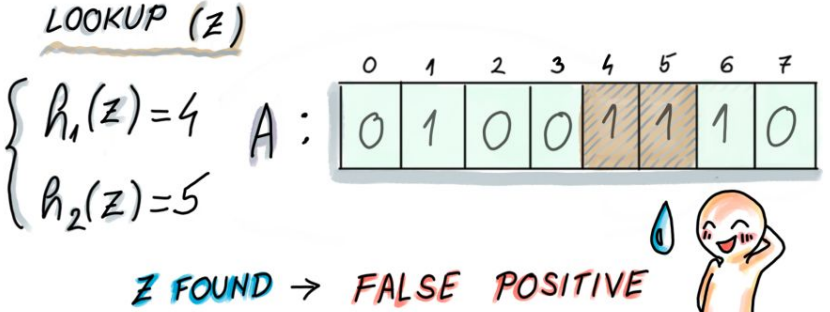
Working of Bloom Filters











If $h_1(z)$ *or* $h_2(z)$ *point to an index storing 0*
Then surely z isn't a member of the set.
Hence there are no False -ve and only True -ve.

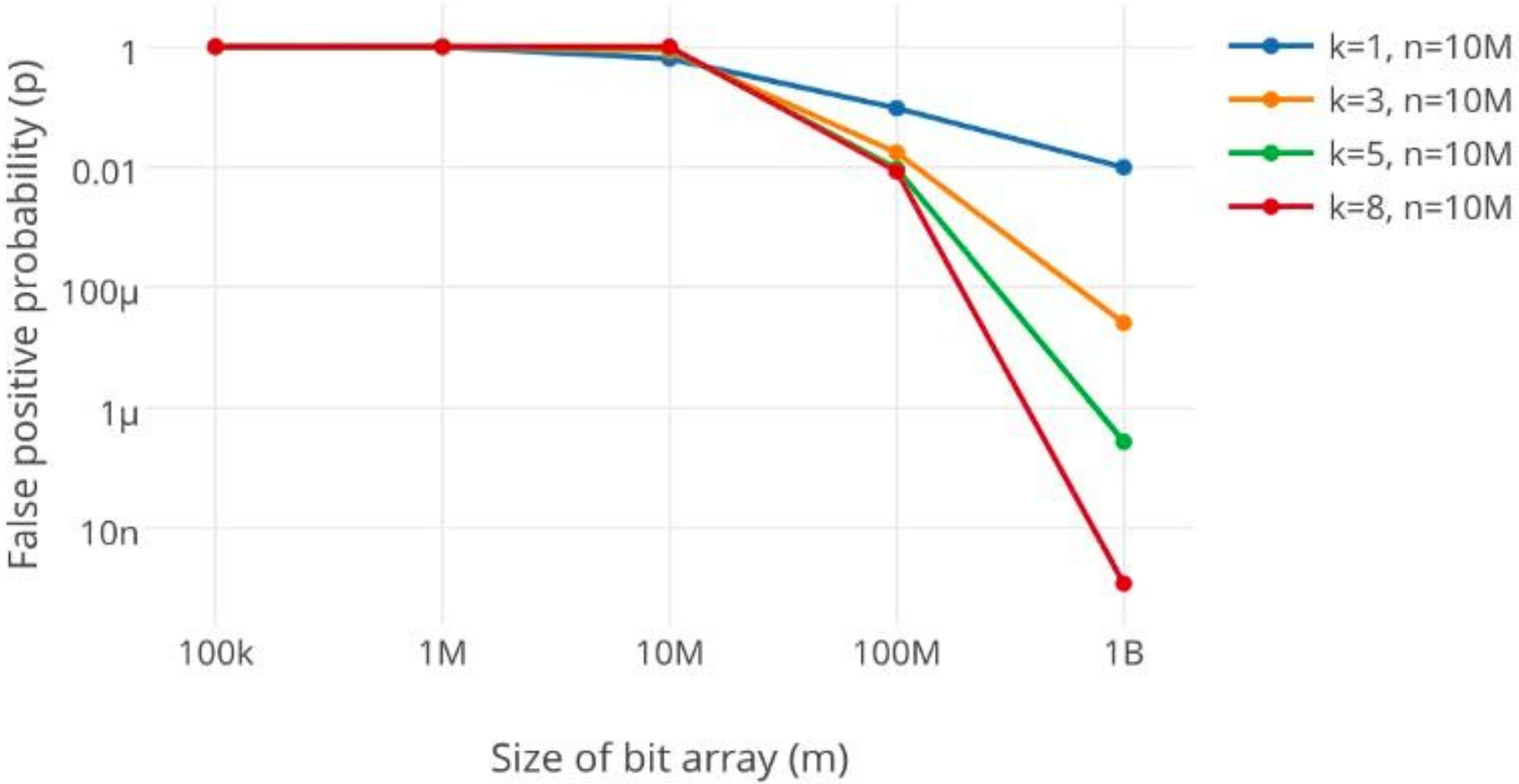
Tunable parameters of Bloom Filters

- The load factor of a bloom filter = number of 1s/size of bloom filter
- When the load factor is 1, all bits are set to 1 so the False Positivity Rate is 100%
- If the load factor is > 0.5, we should increase the size of bloom filter and rehash all the elements. Hence we will require another copy of all the data.

- A Bloom filter has two main components:
  1. A bit array `A[0..m-1]` with all slots initially set to 0,
  2. *k* hash functions h_1, h_2,..., h_k, each **mapping keys onto a range [0,*m*−1]**
- A Bloom filter has two tunable parameters:
  1. **n**: max items in the bloom filter at any given time,
  2. **p**: your acceptable false positive rate {0..1} (e.g. `0.01` → 1%)

- The number of bits needed in the bloom filter: **`m`** `= -n*ln(p) / (ln(2)^2)`
- The number of hash functions we should apply: **`k`** `= m/n * ln(2)`

https://www.di-mgt.com.au/bloom-calculator.html, https://hur.st/bloomfilter/


False positive probability (p)
1
0.01
100µ
1µ
10n
100k
1M
10M
100M
1B
Size of bit array (m)
k=1, n=10M
k=3, n=10M
k=5, n=10M
k=8, n=10M

False +ve Caching of Bloom Filters

- We can find the lower limit to the number of false positives = np

- So we can store False +ve results in a linked list, and whenever the bloom filter returns TRUE we can check if we are dealing with a false +ve or not, if we are then we return FALSE and move the node in cache to head so that it can be queried faster next time, thereby implicitly moving the least frequently queried cached items towards the tail. If a False +ve is found even after checking the cache then it gets cached by inserting itself at the head.

- The limitation of the approach is that if new data is to be inserted in the bloom filter we need to make sure that it is not cached as a False +ve, if it is we need to remove it from the cache. Hence it’s preferable to have False +ve caching for fixed sized sets(CDNs).

Application Of Bloom Filters

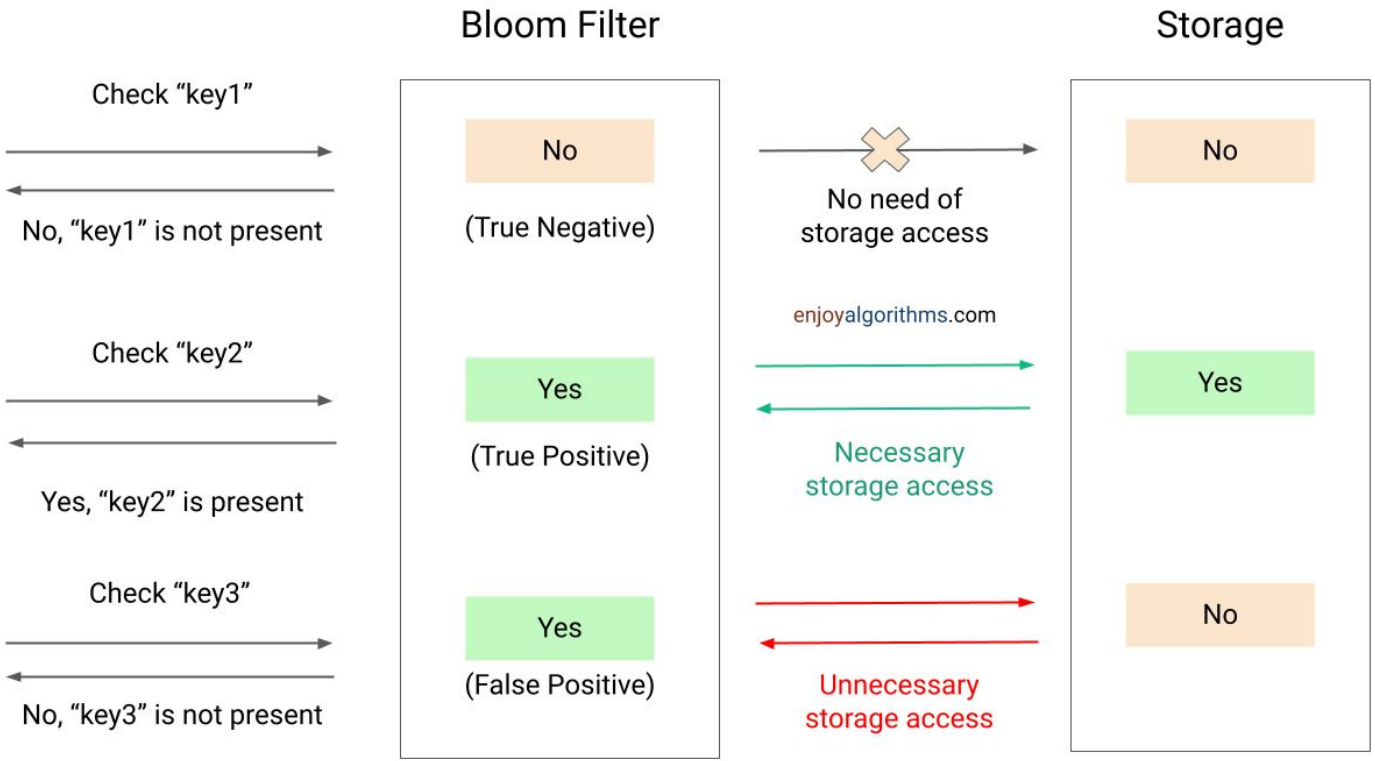
- For any company dealing with huge datasets, if they want to check if something doesn’t exists in their database without reading the entire database can use bloom filters. They will significantly reduce the unnecessary reads on the database as the database will only be read if there is a high probability that the query exists in the database.

- It can be used to detect weak passwords while creating a new password.

- A bloom filter of malware signatures can detect similar malware.

- Google Chrome uses Bloom Filter to check if an URL is a threat or not. If Bloom Filter says that it is a threat, then it goes to another round of testing before alerting the user.

Application Of Bloom Filters



Designing a graph for demonstration

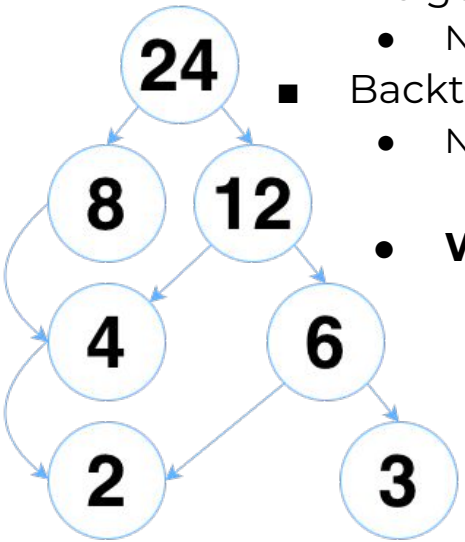
- We initially build a graph that is represented as an Adjacency List where each node stores a number and points directly to all its factors.
  - This simple yet powerful model allows to to easily scale up the cardinality.

- Then we create a bloom filter for every row in the adjacency list by inserting all factors of the node in the bloom filter for the corresponding node.

- Then we compress the adjacency list by removing redundancies.
  - Example earlier 8 would point to both 4 and 2 while 4 would also point to 2.
  - After compressing, 8 will only point to 4 while 4 will point to 2.

Checking if 3 is a factor of 24 using DFS

- We start at node 24 and find out that it points to node 8 & 12.
  - We go to node 8 which points to node 4 and the node 4 points to the node 2.
    - Node 2 is a leaf node as it's a prime number, so backtrack .
  - Backtracking from 2,4 & 8 we reach 24, node 12 on its right points to 4 and 6.
    - We go to node 4 which points to the node 2.
      - Node 2 is a leaf node as it's a prime number, so backtrack.
    - Backtracking from 2,4 we reach 12, node 6 on its right points to 2 and 3.
      - Node 2 is a leaf node as it's a prime number, so backtrack.
        - **So we backtrack to 6 and explore its right node 3.**
      - **We found 3 by doing a DFS from 24, so 3 must be a factor of 24!**



Checking if 3 is a factor of 24 using Bloom Filters & False +ve Caching

- We ask the bloom filter stored inside node 24 if 3 is its member.
  - If it says False, means that 3 is definitely not a factor of 24.
  - If it says True, it's a probable true and could be a false +ve.
    - If the 3,24 is a cached false +ve then 3 is definitely not a factor of 24.
      - We move the node in the cache holding 3,24 to the head of the list.
    - Else we perform a DFS
      - If the DFS from 24 finds 3 then we can say 3 is a factor of 24.
      - Else we found a false +ve that is not in cache.
        - So we create a new node containing 3,24 and insert it at head.
        - And we can say that 3 is not a factor of 24.
- So over a period of time, if the bloom filters doesn't change, we will be able to cache all possible False +ves and the DFS will run only when it's almost certain that the queried element is a member of the set, so if some associated details are also stored for that member they can be returned efficiently with quicker misses.

Measuring Improvements from Bloom Filters & False +ve Caching

```
Factors of 1999998 : 54054 153846 181818 285714 666666 999999
Factors of 1999999 : 1207 28169 117647
Factors of 2000000 : 400000 1000000


[1] Query using DFS
[2] Query using DFS+Caching+BloomFilter
[3] Compare Execution Time of [1] and [2]
[0] Exit

→ 3
DFS took: 353 Microseconds
DFS + Bloom Filter with False+ve Caching took: 251 Microseconds

→ 3
DFS took: 278 Microseconds
DFS + Bloom Filter with False+ve Caching took: 218 Microseconds

→ 3
DFS took: 324 Microseconds
DFS + Bloom Filter with False+ve Caching took: 233 Microseconds
```

```
→ 3
DFS took: 306 Microseconds
DFS + Bloom Filter with False+ve Caching took: 213 Microseconds

→ 3
DFS took: 295 Microseconds
DFS + Bloom Filter with False+ve Caching took: 266 Microseconds

→ 3
DFS took: 386 Microseconds
DFS + Bloom Filter with False+ve Caching took: 179 Microseconds
```

Link to code:
https://github.com/JayaswalPrateek/DFSusingBloomFilter

References

https://en.wikipedia.org/wiki/Bloom_filter

https://llimllib.github.io/bloomfilter-tutorial/

https://www.youtube.com/watch?v=RSwjdlTp108

https://www.youtube.com/watch?v=kfFacplFY4Y

https://www.youtube.com/watch?v=mItewjU-YG8

https://stackoverflow.com/questions/658439/how-many-hash-functions-does-my-bloom-filter-need

https://hackernoon.com/probabilistic-data-structures-bloom-filter-5374112a7832

https://freecontent.manning.com/all-about-bloom-filters/

https://systemdesign.one/bloom-filters-explained/

https://www.enjoyalgorithms.com/blog/bloom-filter

https://redis.io/docs/data-types/probabilistic/bloom-filter/

https://www.spiceworks.com/tech/big-data/articles/what-is-a-bloom-filter/