REPORT

KMU 국민대학교
KOOKMIN UNIVERSITY

과목명 | 빅데이터최신기술

담당교수 | 박하명교수

학과 | 소프트웨어학부

학년 |

학번 | 20181703

이름 | 평선호

제출일 |

```python
bit_present = ['a','b','c','d','e','f','g','h','j','k','l','m','n','o','p',
               'aa','bb','cc','dd','ff','gg','hh','ii','jj','kk','ll','mm','nn','oo','pp','qq','ss']
bit_absent = ['q','r','s','t','u','v','w','x','y','z']  # 없는 bit (q ~ z)

bloom = BloomFilter(100, 0.1)
for i in range(len(bit_present)):
    bloom.put(bit_present[i])

bit_present = random.sample(bit_present,len(bit_present))

bit_absent = random.sample(bit_absent,len(bit_absent))

test_bit = bit_present[:10] + bit_absent
test_bit = random.sample(test_bit,len(test_bit))
```

```
d True
ss True
u False
w False
y False
v False
h True
nn True
q False
a True
r False
l True
hh True
x False
j True
s False
oo True
z False
jj True
t False
```

- 1억명의 사용자 계정이 시스템에 저장되어있고, 사용자가 회원가입 중에 동일한 계정명이 서버에 존재하는지 즉각 확인해주는 시스템을 개발하려고 합니다. 이 때, Bloom Filter를 어떻게 활용하면 좋을까요? 비트배열의 크기, false positive 값 등은 어떻게 설정하면 적절할까요? 생각을 서술해보세요.

False positive가 일어날 확률은 해당 수식으로 표현할 수 있다. $\left(1 - e^{-\frac{km}{n}}\right)^k$

False positvie가 최소가 되려면 전체 원소의 개수 m, 비트 배열V의 크기인 n, 해쉬 함수개수 k, 이 3개중 두개의 값을 알면, False positive가 최소가 되도록 다른 변수 값을 구 할 수 있다.

$k = \frac{m}{n}\ln 2$ 가 성립하면 False positive는 최소값을 가진다.

이때 m, 즉 비트 배열의 크기가 클수록 당연히 False positive의 값이 작아지면서 Bloom Filter의 성능은 향상이 된다. 하지만 비트배열의 크기가 메모리가 관리할 수 있는 범위를 벗어나게 되면 메모리 사용량이 과부하가 되어 적절히 조정해야한다.

해당 1억명의 사용자의 계정이 시스템에 저장되어있다 가정했을 때, 해당 bloom filter를 이용하기 위해 약 100MB가 있다 가정하면, false positive는 대략 0.0215 정도 되며 k는 6으로 약 6번이 hash 를 반복하면 최적으로 bloom filter를 사용 할 수 있다.

- Bloom Filter를 사용하면 좋을만한 상황을 얘기하고, 왜 그러한 상황에서 Bloom Filter가 도움이 되는지 설명해보세요.

Bloom Filter 는 공간 효율적인 확률 데이터 구조로, 요소 가 집합의 구성원인지 여부를 테스트하는 데 사용됩니다

k개의 해시 함수로 일부 설정된 요소를 m개의 배열 위치 중 하나로 mapping하거나 hash하여 균일한 무작위 분포를 생성합니다.

해당 Bloom Filter를 특징을 바탕으로 어떠한 상황에서 좋은지 얘기를 하자면,

1. 캐쉬 필터링
   - 콘텐츠 전송 네트워크는 웹 캐시를 배포합니다. 이때 더 나은 성능과 안정성으로 사용자에게 웹 콘텐츠를 캐시하고 제공합니다. 이때 Bloom 필터의 주요 응용 프로그램은 이러한 웹 캐시에 저장할 웹 개체를 효율적으로 결정하는데 사용됩니다.
   일회성 캐싱을 방지하기 위해 Bloom필터를 사용하여 사용자가 액세스 하는 모든 URL을 추적합니다. 이때 웹 개체는 두번째 요청에서 캐시됩니다. 이러한 방식으로 디스크 쓰기 작업량을 줄이고, 1hit는 디스크 캐시에 기록되지 않습니다. 이를 통해 디스크의 캐시 공간이 절약되어 캐시 적중률이 높아집니다.

2. Google Chrome에서의 위험한 사이트 검사
   - Bloom filter를 사용해서 빠르게 검사한 다음, 의심이 가는 사이트인 경우 데이터 베이스에 다시 정확하게 검사한다. 이때 위험 사이트 데이터베이스의 크기가 크고, 검사 요청이 빈번하게 일어나기 떄문에 Bloom filter를 전처리 과정으로 사용해서 데이터베이스 요청 부하를 줄인다. Google에서의 사이트 검사는 굉장히 큰 대용량이기때문에 , 이를 hash함수를 통해 반복수를 줄이고 많은 양의 데이터를 줄여서 공간 효율적으로 검색 할 수 있다.

- 실습 결과물인 Flajolet-Martin Algorithm (Version 2)을 구현하세요.
- Version 1도 구현해보고, Version 1과 2의 정확도를 비교해보세요.
  - 정확도 비교시에 데이터는 적절히 임의로 생성해보세요.
- hash function을 여러개 사용하고 '중앙 값 평균내기' 기법을 활용하여 Version 2의 정확도를 올려보세요.
  - 사용하는 hash function의 수, group의 수 등을 조절해가며 정확도가 어떻게 변화하는지 확인해보세요.

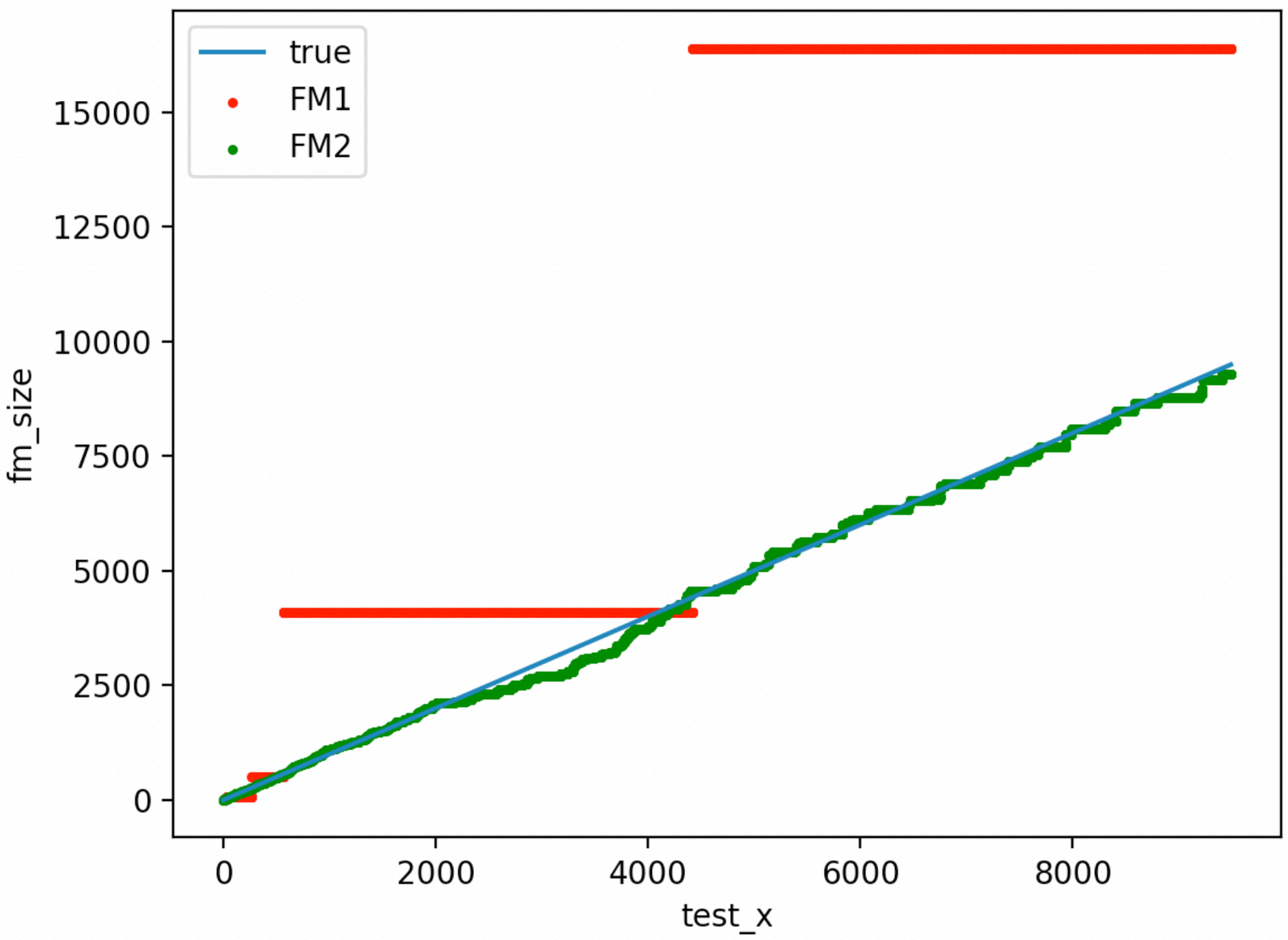
Version1

```
class FM1:
    def __init__(self,domain_size):
        self.bitarray = 0
        self.domain_size = domain_size
        self.n_bits = math.ceil(math.log2(domain_size))
        self.mask = (1 << self.n_bits) -1
        self.seed = random.randint(0,9999999)
        self.r_list = []

    def put(self,item):
        h = mmh3.hash(item,self.seed) & self.mask
        r = 0
        if h == 0:
            return
        while (h & (1 << r)) == 0:
            r += 1
        if len(self.r_list) == 0:
            self.r_list.append(r)
        elif self.r_list[0] < r:
            del self.r_list[0]
            self.r_list.append(r)
        self.bitarray |= (1 << r)
    def size(self):
        R = self.r_list[0]
        #while(self.bitarray & (1 << R) != 0):
        #    R += 1
        return 2 ** R
```

Version2

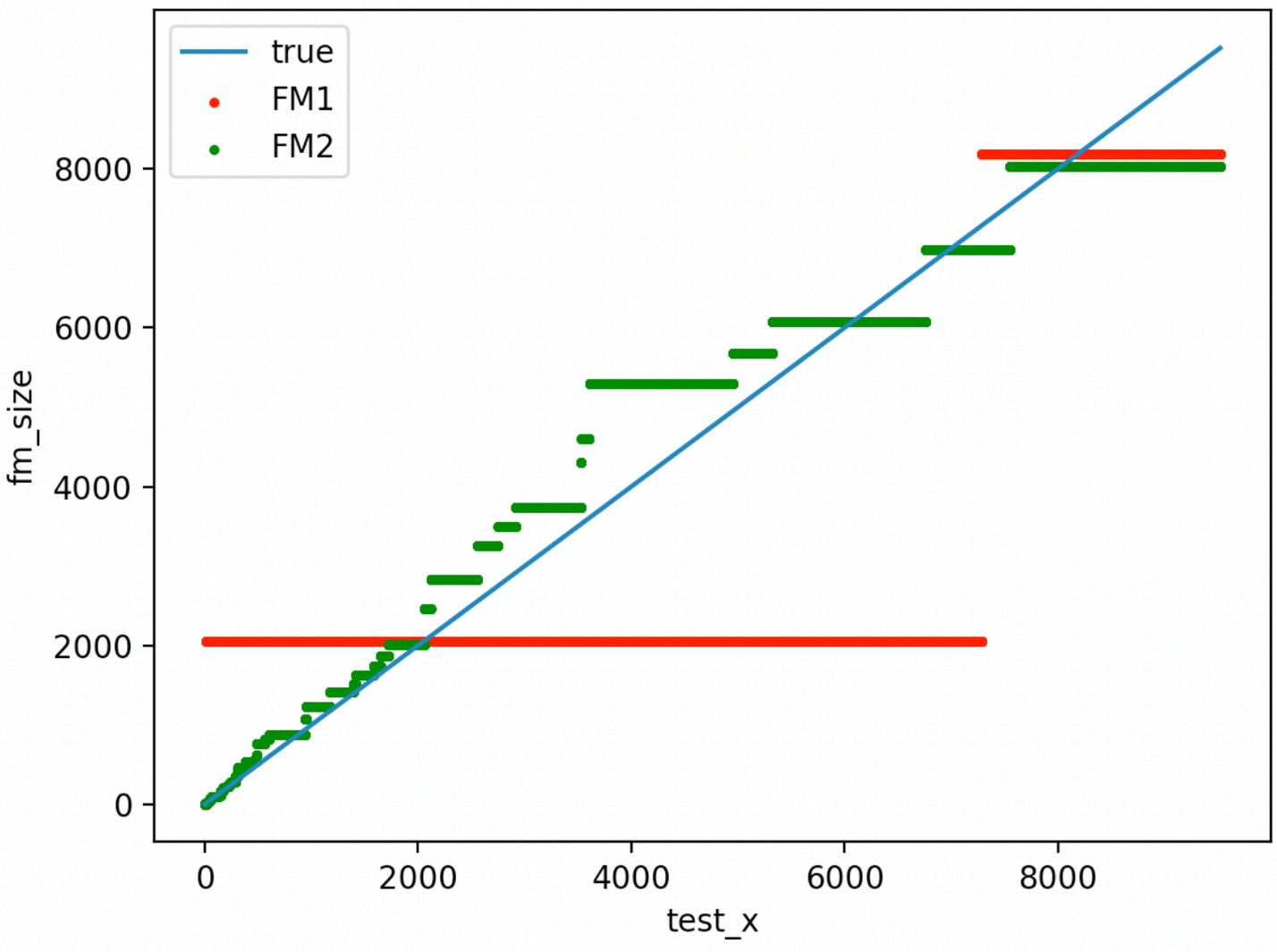
```python
class FM2:
    def __init__(self,domain_size,groupsize):
        self.bitarray = [0 for _ in range(groupsize)]
        self.domain_size = domain_size
        self.n_bits = math.ceil(math.log2(domain_size))
        self.mask = (1 << self.n_bits) - 1
        self.seed = [random.randint(0,9999999) for _ in range(groupsize)]
        self.groupsize = groupsize

    def put(self,item):
        for i in range(self.groupsize):
            h = mmh3.hash(item,self.seed[i]) & self.mask
            r = 0
            if h == 0:
                return
            while (h & (1 << r)) == 0:
                r += 1
            self.bitarray[i] |= (1 << r)
    def size(self):
        sum = 0
        for i in range(self.groupsize):
            R = 0
            while(self.bitarray[i] & (1 << R) != 0):
                R += 1
            sum += R
        R = sum / self.groupsize
        return 2 ** R/0.77351
```

FM1 FM2 실제값 비교화면



Fm1, fm2둘다
Hash function의 수: 1000
group의 수: 1000000

FM1은 R = max(r(a))로 R값을 변경하는데 이때 모든 a에 대해서 r(a)가 우연찮게 너무 크다면 값이 중간에 튈 수가 있다. 따라서 그래프가 툭툭 튀는 경향을 보입니다. 그러나 FM2는 hash함수를 통해 중앙값을 구한 후 이를 반복해서 했기 때문에 실제 값과 굉장히 비슷한 경향을 띄며 진행합니다.



Fm1, fm2둘다
Hash function의 수: 100
group의 수: 10000