Project 1

Author Name
Anqi (Stella) Liu

1 Problem Statement

We are required to analyze the following program/code sample.

```
int j = 2
while (j < n) {
int k = j
while (k < n) {
Sum += a[j]*b[k]
k=k*k
}
j=2*j
}
```

2 Theoretical Analysis

We observe that the outer loop on counter j increments by two times. So that loop runs in O(logn) and the inner loop runs on counter k getting squared every time.

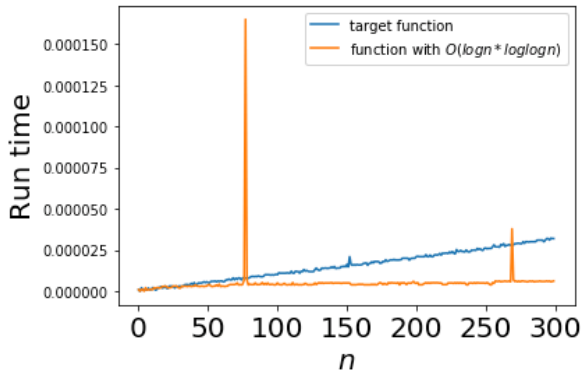
Suppose that after m iterations, k becomes 2^(2^m) and 2^(2^m)>=n, that is, m>=loglog(n).

Thus, the two nested loops run in ***O(lognloglogn)*** time

Use some simple function with O(n) to compare.

Then we can compare the dp_constant function with O(n) and the dummy function with O(logn * loglogn). As we can see, the dummy function takes less time than O(n)

```
plot(300, fun_dp_constant, dummyfun)
print("Figure. The run time of the dp_constant solution and a dummy function with O(logn * loglogn).")
```



```
import time
%matplotlib inline
import matplotlib.pyplot as plt

# Create a dummy function with O(logn * loglogn) for comparison
def dummyfun(n):
    i = 2
    while (i < n):
        i = 2*i
        j = i
        while j < n:
            j = j**2

def plot(n, fun, dummyfun):
    """
    Plot the run time of a function.

    Parameters
    ----------
    n : a number
    function : a function
    """

    x = [i for i in range(n)]
    y1 = []
    y2 = []

    for i in x:
        start = time.time()
        fun(i)
        end = time.time()
        y1.append(end - start)

    for i in x:
        start = time.time()
        dummyfun(i)
        end = time.time()
        y2.append(end - start)

    plt.plot(x, y1, label='target function')
    plt.plot(x, y2, label='function with $O(logn * loglogn)$')
    plt.xlabel('$n$', fontsize=20)
    plt.ylabel('Run time', fontsize=20)
    plt.xticks(fontsize=20)
#     plt.yticks([0, max(y1,y2) // 2, max(y1,y2)], fontsize=20)
    plt.legend(loc='best')
    plt.tight_layout()
    plt.show()
```

1 Experimental Analysis

1.1 Program Listing

By running pseudocode combined with O(log n) such as binary search and O(log log n) in

n= 10, 30, 50, 70,90,100,300,500,700,900,1000,3000 ,5000,7000,9000,10000,30000,50000,70000,90000,100000

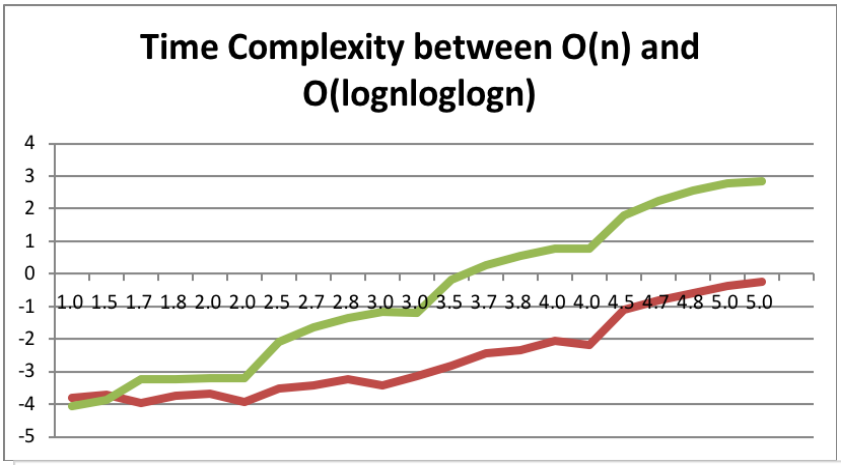
n	O(n)	experimental	Theoretical O(logn log log n)
10	0.000158548	8.77E-05	0.000659227
30	0.0001907	1.37E-04	0.006119967
50	0.0001109	5.76E-04	0.009887695
70	0.0001755	5.83E-04	0.00252223
90	0.0002041	6.08E-04	0.00854373
100	0.0001192	6.38E-04	0.008004189
300	0.0002935	0.0078814	0.009958506
500	0.0003896	0.0225573	0.055912256
700	0.0005858	0.044884	0.074111938
900	0.0003893	0.0688686	0.107934475
1000	0.000754595	0.065732241	0.125576735
3000	0.0015447	0.6495638	0.302310944
5000	0.0036855	1.8409483	0.539046288
7000	0.0044348	3.5880029	0.805282354
9000	0.0086715	5.8057451	1.059251785
10000	0.0064185	6.0600863	1.286026955
30000	0.0817621	64.008932	3.574962854
50000	0.1504788	178.06875	5.779388428
70000	0.2606556	354.55102	8.858988285
90000	0.4177749	588.2923	10.72217631
100000	0.580467939	715.9476097	13.10009623

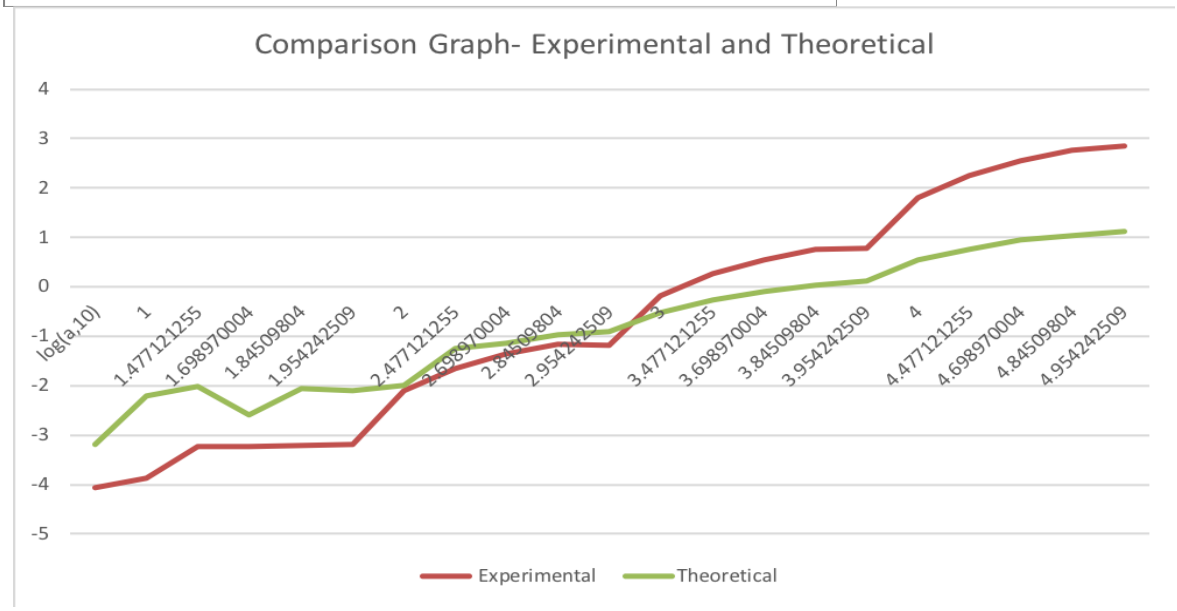
1.2 Data Normalization Notes

To Plot, we take a log based 10 of all of these

log(a, 10)	*log(b, 10)*	*log(c, 10)*	*log(d, 10)*
1	-3.79984	-4.05681	-3.18096
1.477121	-3.71957	-3.86224	-2.21325
1.69897	-3.95521	-3.23974	-2.0049
1.845098	-3.75578	-3.23403	-2.59822
1.954243	-3.69019	-3.21578	-2.06835
2	-3.92369	-3.19507	-2.09668
2.477121	-3.5324	-2.1034	-2.00181
2.69897	-3.40941	-1.64671	-1.25249
2.845098	-3.23225	-1.34791	-1.13011
2.954243	-3.40967	-1.16198	-0.96684

3	-3.12229	-1.18222	-0.90109
3.477121	-2.81115	-0.18738	-0.51955
3.69897	-2.43351	0.265042	-0.26837
3.845098	-2.35312	0.554853	-0.09405
3.954243	-2.0619	0.763858	0.024999
4	-2.19257	0.782479	0.10925
4.477121	-1.08745	1.806241	0.553272
4.69897	-0.82252	2.250588	0.761882
4.845098	-0.58393	2.549679	0.947384
4.954243	-0.37906	2.769593	1.030283
5	-0.23622	2.854881	1.117274



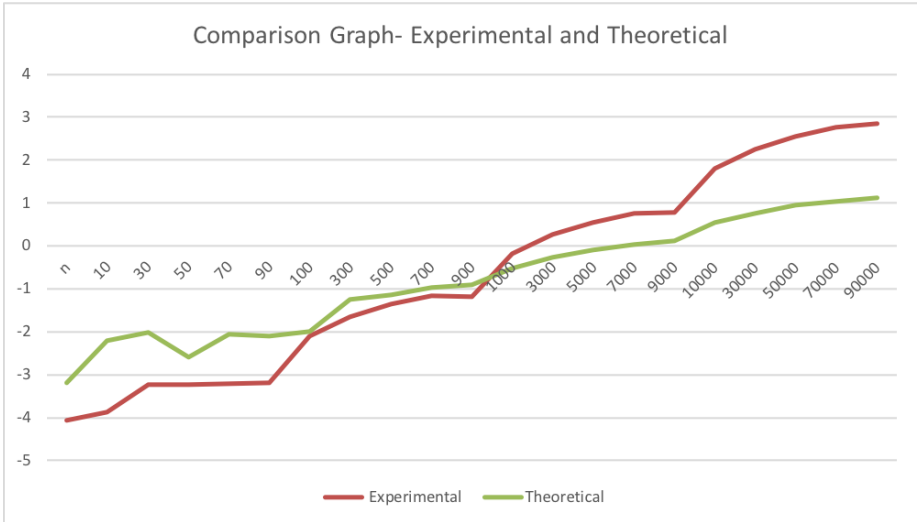


What the curve shows is that while the functions are similar, they are still diverging
We observe that experimental time is growing faster compared to theoretical.

1.3 Output Numerical Data

n	Experimental	Theoretical
10	-4.056812086	-3.180964769
30	-3.862237421	-2.213250955
50	-3.23974277	-2.004904925
70	-3.234025935	-2.598215287
90	-3.215779235	-2.068352496
100	-3.195066981	-2.09668269
300	-2.103396467	-2.001805827
500	-1.646713681	-1.252492982
700	-1.347908771	-1.130111827
900	-1.161978512	-0.966839817
1000	-1.182221563	-0.901090814
3000	-0.187378193	-0.519546131
5000	0.265041602	-0.268373941
7000	0.554852788	-0.094051817
9000	0.763857966	0.024999205
10000	0.782478805	0.109250071
30000	1.806240579	0.553271534
50000	2.250587699	0.761881884
70000	2.549678742	0.947384127
90000	2.769593165	1.030282944
100000	2.854881243	1.117274486

1.4 Graph



2 Conclusions

The curve shows that the functions are similar with higher n and the analysis (and the subsequent hypothesis) may be correct.

Task2

Problem Statement

a.Analyze Union Find data structure. Implement it. Run experiments on it, and show that the time complexity of m unions and n finds matches the theoretical prediction.

Theoretical Analysis

We can now implement the operations as follows:

MakeSet(x): just set x->head=x. This takes constant time.
Find(x): just return x->head. Also takes constant time.
Union(x, y): To perform a union operation we merge the two lists/trees together, and reset the head pointers on one of the lists/trees to point to the head of the other.

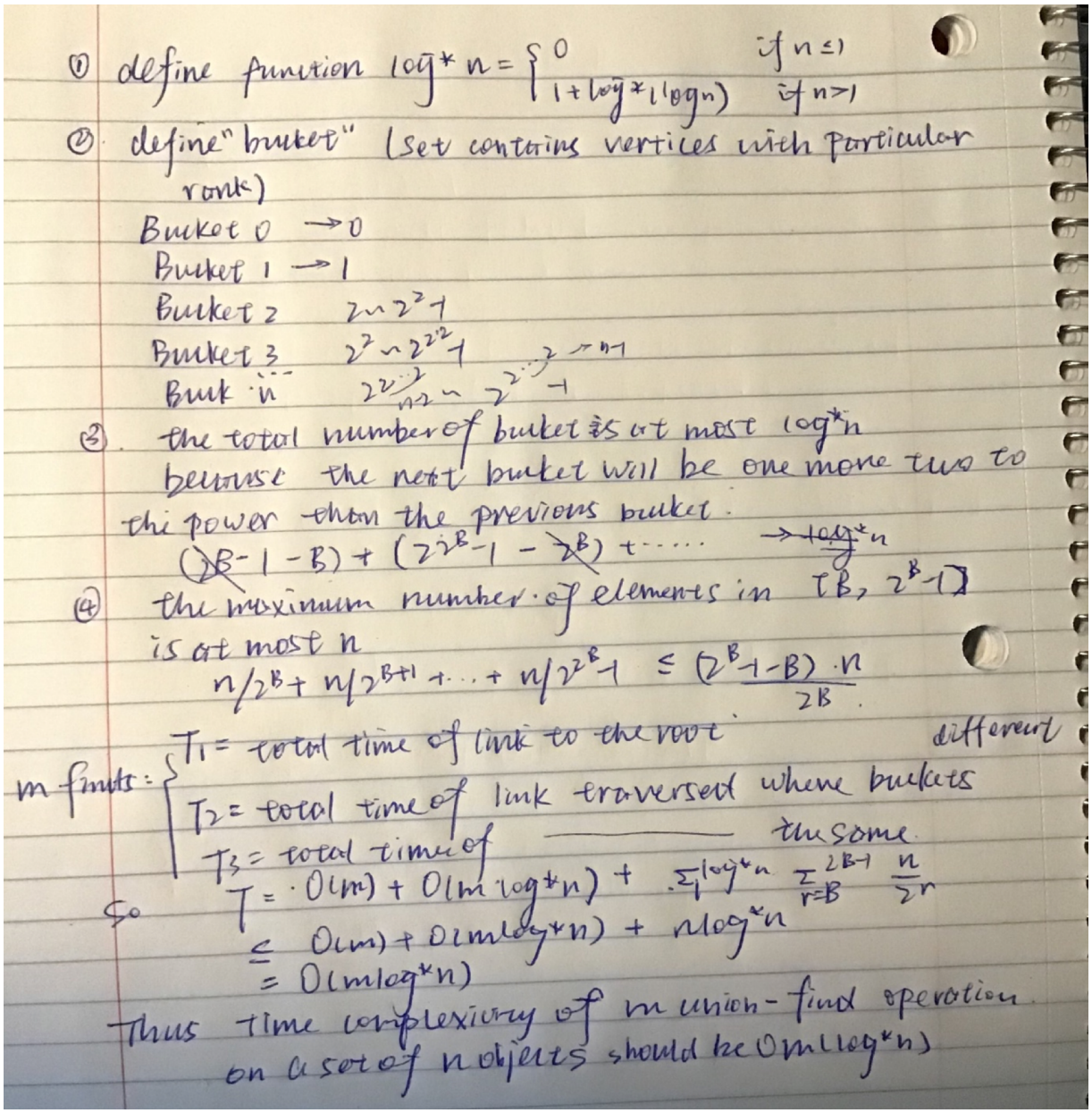
List-Based:
The Find and MakeSet operations are constant time so they are covered by the O(m) term. Each Union operation has cost proportional to the length of the list whose head pointers get updated. So, we need to find some way of analyzing the total cost of the Union operations. Here is the key idea: we can pay for the union operation by charging O(1) to each element whose head pointer is updated. So, all we need to do is sum up the costs charged to all the elements over the entire course of the algorithm. Then looking from the point of view of some lowly element x. Over time, at most log n times will x get walked on and have its head pointer updated. The reason is that we only update head pointers on the smaller of the two lists being joined, so every time x gets updated, the size of the list it is in at least doubles, and this can happen at most log n times. So, we were able to pay for unions by charging the elements whose head pointers are updated, and no element gets charged more than O(log n) total, so the total cost for unions is O(n log n), or O(m + n log n) for all the operations together.

Tree_based:
MakeSet(x): set x's rank to 0 and its parent pointer to itself. This takes constant time.

Find(x): starting from x, follow the parent pointers until you reach the root, updating x and all the nodes we pass over to point to the root. This is called path compression. The running time for Find(x) is proportional to (original) distance of x to its root.

Union(x, y): Let Union(x, y) = Link(Find(x), Find(y)), where Link(root1,root2) behaves as follows. If the one of the roots has larger rank than the other, then that one becomes the new root, and the other (smaller rank) root has its parent pointer updated to point to it. If the two roots have equal rank, then one of them (arbitrarily) is picked to be the new root and its rank is increased by 1. This procedure is called union by rank.

As for the time complexity of m unions and n, it should be O(mlog*n)
Proof:



reference:
https://www.slideshare.net/WeiLi73/time-complexity-of-union-find-55858534?from_action=save

Time complexity analysis

Quick find
The time complexity of find() operation is O (1)
The time complexity of Union () operation is O (n)
If we want to call Union () n times repeatedly, the time complexity will be O (n ^ 2)

Quick union
The worst-case time complexity of the find() operation is O (n).
The worst case time complexity of the Union () operation is O (1).

Weighted Quick union

The worst-case time complexity of the find() operation is O (logn).
The reason is that we connect the connection set with fewer objects to the connection set with larger objects every time, so the height of the generated connection set in the worst case is O (logN).
The worst case time complexity of the Union () operation is O (logn).
The reason is the same as find().

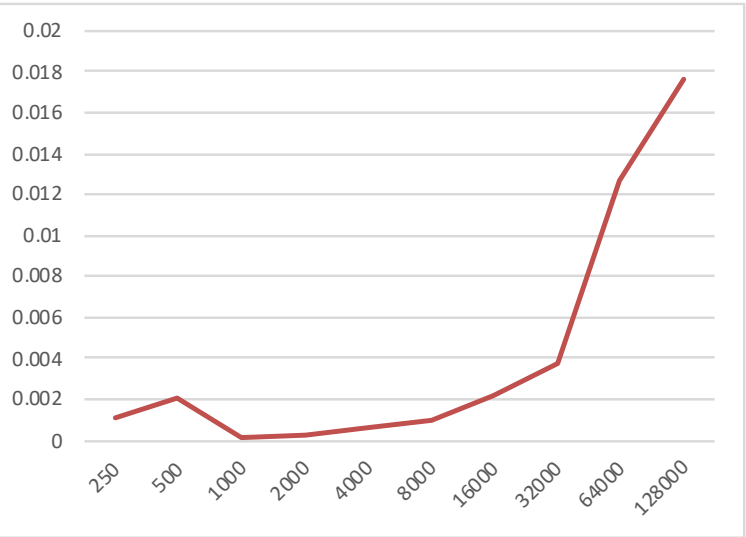
Path compression
Union (): O (logn) at worst.
Find(): the worst case is O (logn).

Weighted quick union with path compression time complexity analysis
Theoretically, starting from a completely disconnected set of n objects, the time required for any sequential m-call of union() is O (n + mlog * n).
The iterated logarithm of a real number means that the result will be less than or equal to 1 only after several successive logarithm operations on the real number. This function increases very slowly and can be regarded as an approximate constant (for example, the iterated logarithm of 2 ^ 65535 is 5).
Therefore, we can think that the weighted quick union with path compression is a linear time algorithm.

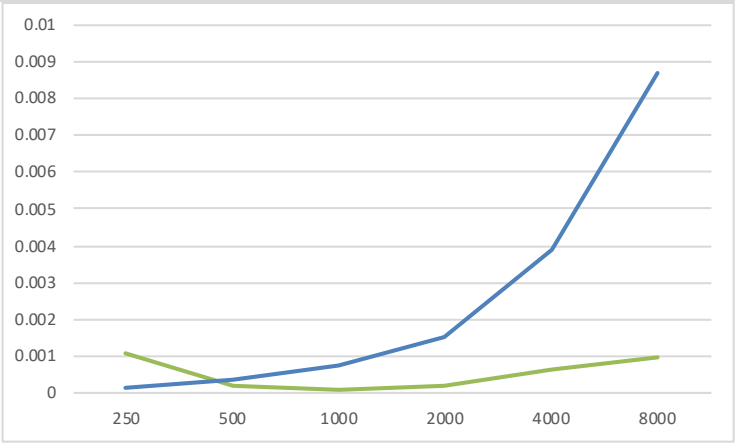
Experimental Analysis

Program Listing

250	0.00109005
500	2.08E-03
1000	0.000111341
2000	0.000212669
4000	0.000643253
8000	0.00100565
16000	0.002141714
32000	0.003737688
64000	0.012708664
128000	0.017648458



Time complexity compared with O(n):



Conclusion:

Theoretically, starting from a completely disconnected set of n objects, the time required for any sequential m-call of union() is O (n + m log * n). However, in experiment, this function increases very slowly and can be regarded as an approximate constant. Therefore, we can think that the weighted quick union with path compression is a linear time algorithm.

Task2

Problem Statement

2b. Analyze Bloom Filter data structure. Implement it. Run experiments on it, and show that the time complexity of m unions and n finds matches the theoretical prediction

Theoretical Analysis

Now we will analyze the probability of a false positive. The probability that one hash do not set a given bit is 1 – 1/m given the setting in previous section. The probability that it is not set by any of

the hash functions is $(1-\frac{1}{m})^k$ .Hence, after all n elements of S have been inserted into the Bloom filter, the probability that a specific bit is still 0 is:

$$f = (1-\frac{1}{m})^{kn} = e^{-\frac{kn}{m}}$$

(Note that this uses the assumption that the hash functions are independent and perfectly random.) The probability of a false positive is the probability that a specific set of k bits are 1, which is

$$(1-(1-\frac{1}{m})^{kn})^k = (1-e^{-\frac{kn}{m}})^k$$

This shows that there are three performance metrics for Bloom filters that can be traded off: computation time (corresponds to the number k of hash functions), size (corresponds to the number m of bits), and probability of error (corresponds to the false positive rate).

Suppose we are given the ratio $\frac{m}{n}$ and want to optimize the number k of hash functions to minimize the false positive rate f. Note that more hash functions increase the precision but also the number of 1's in the filter, thus making false positives both less and more likely at the same time. Let

$$g = ln(f) = kln(1-e^{-\frac{kn}{m}})$$

Suppose $p = e^{-\frac{kn}{m}}$. Then, the derivative of g is

$$\frac{dg}{dk} = ln(1-p) + \frac{kn}{m}\cdot\frac{p}{1-p}$$

We find the optimal k, or right number of hash functions to use, when the derivative is 0. The solution is $k = (ln2)\cdot\frac{m}{n}$. So for this optimal value of k, the false positive rate is:

$$(\frac{1}{2})^k = (0.6185)^{\frac{m}{n}}$$

As m grows in proportion to n, the false positive rate decreases.

False positve rates for choices of k given m/n

m/n	k	k=1	k=2	k=3	k=4	k=5
2	1.39	0.393	0.400			
3	2.08	0.283	0.237	0.253		
4	2.77	0.221	0.155	0.147	0.160	
5	3.46	0.181	0.109	0.092	0.092	0.101
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347
8	5.55	0.118	0.0489	0.0306	0.024	0.0217

From http://www.cs.wisc.edu/~cao/papers/summary-cache/

Time /space complexity Analysis

Time. The time factor is the average time required to reject a message as a member of the given set. In measuring this factor, the unit used is the time required to calculate a single bit address in the hash area, to access the addressed bit, and to make an appropriate test of the bit's contents.

For the conventional hash-coding method, the test is a comparison of the addressed bit in the hash area with the corresponding bit of the message. For method 1, the test is a comparison of the hash area bit with a corresponding bit of a code derived from the message. For method 2, the test is simply to determine the contents of the hash area bit; e.g. is it 1? For the analysis to follow, it is assumed that the unit of time is the same for all three methods and for all bits in the hash area.[1]

The time factor measured in these units is called the normalized time measure, and the space/time trade-offs will be analyzed with respect to this factor. The normalized time measure is

$$T = \underset{m_i \in \tilde{a}}{\text{mean}} \ (t_i), \qquad (2)$$

where: M is the given set of messages; a is the set of messages identified (correctly or falsely) as members of M;

$\bar{a}$ is the set of messages identified as nonmembers of M; m_i is the ith message; and t_i is the time required to reject the ith message.

The hash area has N bits and is organized into h cells of $b + 1$ bits each, of which n cells are filled with the n messages in M. Let ϕ represent the fraction of cells which are empty. Then

$$\phi = \frac{h - n}{h} = \frac{N - n \cdot (b + 1)}{N}. \tag{3}$$

Solving for N yields

$$N = \frac{n \cdot (b + 1)}{1 - \phi}. \tag{4}$$

Let us now calculate the normalized time measure, T. T represents the expected number of bits to be tested during a typical rejection procedure. T also equals the expected number of bits to be tested after a nonempty cell has been accessed and abandoned. That is, if a hash-addressed cell contains a message other than the message to be tested, on the average this will be discovered after, say, E bits are tested. Then the procedure, in effect, starts over again.

Since ϕ represents the fraction of cells which are empty, then the probability of accessing a nonempty cell is $(1 - \phi)$, and the probability of accessing an empty cell is ϕ. If an nonempty cell is accessed, the expected number of bits to be tested is $E + T$, since E represents the expected number of bits to be tested in rejecting the nonempty accessed cell, and T represents the expected number of bits to be tested when the procedure is repeated. If an empty cell is accessed, then only one bit is tested to discover this fact. Therefore

$$T = (1 - \phi)(E + T) + \phi. \tag{5}$$

In order to calculate a value for E, we note that the conditional probability that the first x bits of a cell match those of a message to be tested, and the $(x + 1)$th bit does not match, given that the cell contains a message other than the message to be tested, is $(\frac{1}{2})^x$. (The reader should remember that the first bit of a message always matches the first bit of a nonempty cell, and consequently the exponent is x rather than $x + 1$, as would otherwise be the case.) Thus, for $b \gg 1$, the expected value of E is approximated by the following sum:

$$\sum_{x=1}^{\infty} (x + 1) \cdot (\tfrac{1}{2})^x = 3. \tag{6}$$

Therefore

$$T = (3/\phi) - 2, \tag{7}$$

$$N = n \cdot (b + 1) \cdot \frac{T + 2}{T - 1}. \tag{8}$$

Equation (8) represents the space/time trade-off for the conventional hash-coding method.

reference:Burton Bloom published the original article on this idea, Space-Time Trade-offs in Hash Coding with Allowable Errors, back in 1970. By 2000, there were many new applications of Bloom filters!

In conclusion,

Given a Bloom filter with *m* bits and *k* hashing functions, both insertion and membership testing are *O(k)*. That is, each time we want to add an element to the set or check set membership, we just need to run the element through the *k* hash functions and add it to the set or check those bits.

Experimental Analysis

Test performance of BloomFilter at a set capacity and error rate

run benchmarks.py in my github to test performance

```
capacity=100
```

0.003 seconds to add to capacity, 34328.89 entries/second
Number of Filter Bits: 960
Number of slices: 4
Bits per slice: 240

Fraction of 1 bits at capacity: 0.342
0.002 seconds to check false positives, 60857.57 checks/second
Requested FP rate: 0.1000
Experimental false positive rate: 0.0200
Projected FP rate (Goel/Gupta): 0.013748

```
capacity=1000
```

0.024 seconds to add to capacity, 42035.94 entries/second
Number of Filter Bits: 9588
Number of slices: 4
Bits per slice: 2397

Fraction of 1 bits at capacity: 0.343
0.017 seconds to check false positives, 59350.56 checks/second
Requested FP rate: 0.1000
Experimental false positive rate: 0.0140
Projected FP rate (Goel/Gupta): 0.013564

```
capacity=10000
```

0.077 seconds to add to capacity, 130096.68 entries/second
Number of Filter Bits: 95852
Number of slices: 4
Bits per slice: 23963

Fraction of 1 bits at capacity: 0.339
0.054 seconds to check false positives, 183995.40 checks/second
Requested FP rate: 0.1000
Experimental false positive rate: 0.0142
Projected FP rate (Goel/Gupta): 0.013553

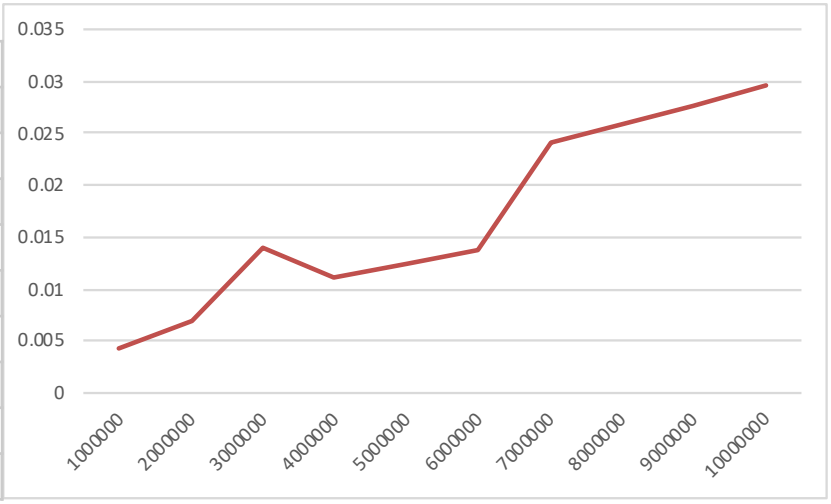
Experimental Analysis

Program Listing

run bloom.py in GitHub project

Graph

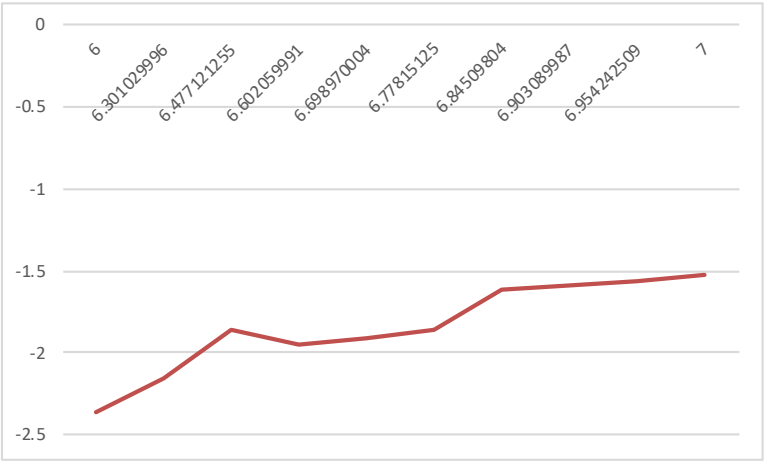
1000000	0.004306316
2000000	0.006885052
3000000	0.013880014
4000000	0.011113882
5000000	0.01237607
6000000	0.013819933
7000000	0.024103165
8000000	0.025764942
9000000	0.027500868
10000000	0.029488802



Data Normalization

take log

6	-2.36589
6.301029996	-2.16209
6.477121255	-1.85761
6.602059991	-1.95413
6.698970004	-1.90742
6.77815125	-1.85949
6.84509804	-1.61793
6.903089987	-1.58897
6.954242509	-1.56065
7	-1.53034



Conclusion

Given a Bloom filter with *m* bits and *k* hashing functions, both insertion and membership testing are *O(k)*. That is, each time we want to add an element to the set or check set membership, we just need to run the element through the *k* hash functions and add it to the set or check those bits.

Task 3

Meta Option C (Asymptotic Design) Project

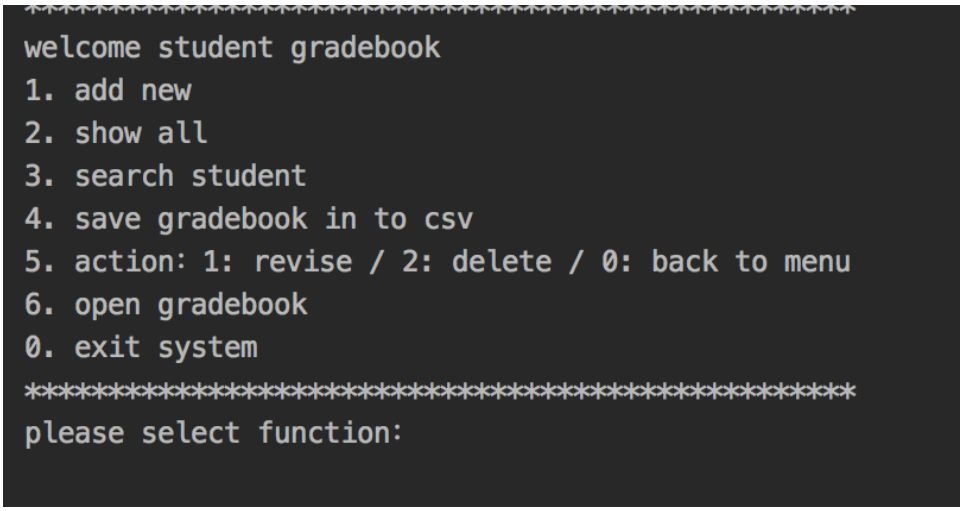
Design a gradebook software, which allows the following features:

- Import a gradebook (CSV file), with fields: “firstname, lastname, email, phone, city, score (0-1000 integer value), department”
- Search the gradebook, with information like: “search key”
- Search should run in O(log n + k) time searching on ALL of its fields, where k is the number of results returned by the search

please check my GitHub project for reference
https://github.com/Stella2019/6212-project

Here are some findings and ideas about the project:

run cards_tools.py, it has the following functions:



First, I used binary search to get O(logn), but sometimes have mistakes in finding.
It is because my application allows to add and delete student info manually.
So the id may not be continued and ordered.

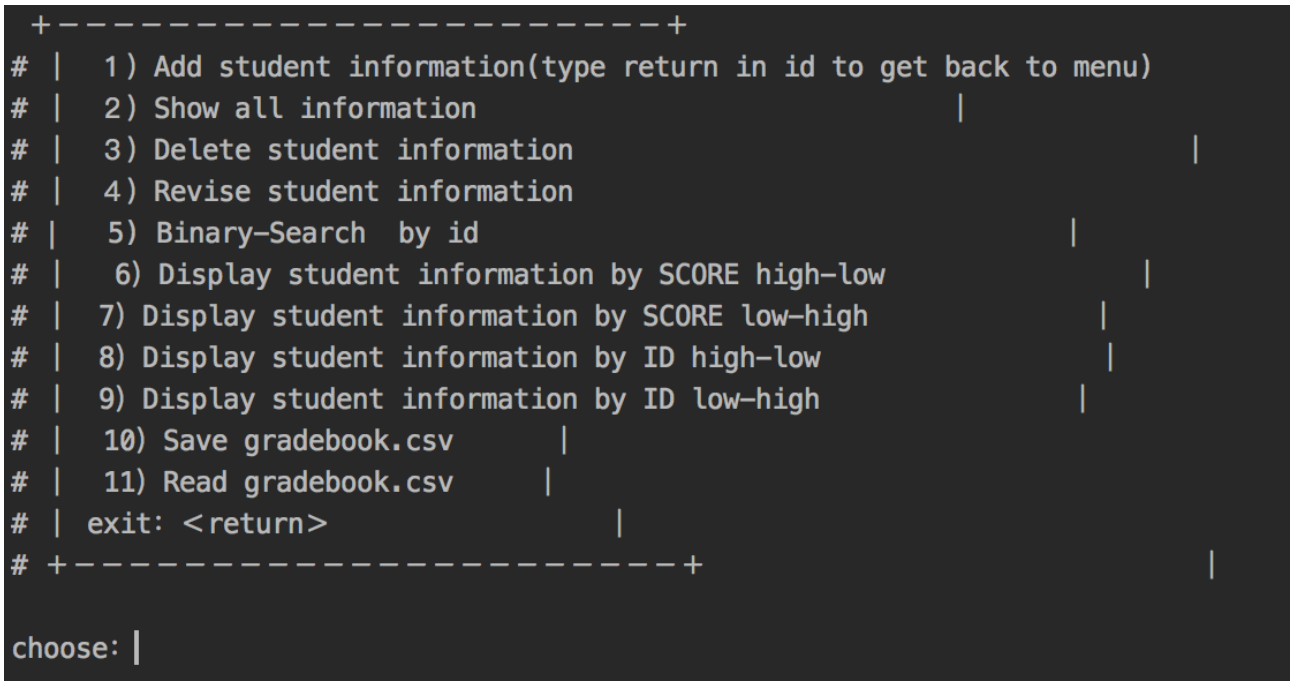
```
    return left

In [47]: arr2 = list(range(1,20,2))
         arr2

Out[47]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

In [48]: # uncontinous array
         def binarysearch2(arr, target):
             left, right = 0, len(arr) - 1

             while (left < right):
                 mid = (left + right) // 2
                 if arr[mid] == target:
                     return arr[mid]
                 elif arr[mid] < target:
                     left = mid + 1
                 else:
                     right = mid - 1

             return arr[left]

In [49]: binarysearch2(arr2, 4)

Out[49]: 5
```

Then I make some changes to improve and make sure to get right id from binary search.

```
In [47]: arr2 = list(range(1,20,2))
         arr2

Out[47]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

In [54]: # uncontinous array
         def binarysearch2(arr, target):
             left, right = 0, len(arr) - 1

             while (left < right):
                 mid = (left + right) // 2
                 if arr[mid] == target:
                     return arr[mid]
                 elif arr[mid] < target:
                     left = mid + 1
                 else:
                     right = mid - 1
             if arr[left] > target:
                 return False
             if arr[right] < target:
                 return False
             return arr[left]

In [58]: binarysearch2(arr2, 14)

Out[58]: False
```

However, if the info is unordered, the binary search may not be efficient under such circumstances. So I want to try more ways like you mentioned in today's lecture such as Select(A,k) to make some improvements.

To improve that, I wrote a new one– gradebook.py with sorting and still working on it.



Appendix

Task2 Union Find

```
class Quick_Find:
    def __init__(self,N):
        self.count = N
        self.ids = [i for i in range(self.count)]
    def connect(self,p,q):
        return self.find[p] == self.find(q)
    def find(self,p):
        return self.ids[p]
    def union(self,p,q):
        pId = self.find(p)
        qId = self.find(q)
        if pId == qId:
            return
        for i in range(len(self.ids)):
            if self.ids[i] == pId:
                self.ids[i] = qId
        self.count-=1
    def getcount(self):
        return self.count
class Quick_Union:
    def __init__(self,N):
        self.count = N
        self.ids = [i for i in range(N)]
    def connect(self,p,q):
        return self.find(p) == self.find(q)
    def find(self,p):
        while self.ids[p] != p:
            p = self.ids[p]
        return p
    def union(self,p,q):
        pID = self.find(p)
        qID = self.find(q)
        if pID == qID:
            return
```

```
        self.ids[pID] = qID
        self.count -= 1
    def getcount(self):
        return self.count

class Weighted_Union_Find:
    def __init__(self,N):
        self.count = N
        self.ids = [i for i in range(N)]
        self.size = [1 for i in range(N)]
    def connect(self,p,q):
        return self.find(p) == self.find(q)
    def find(self,p):
        while self.ids[p] != p:
            p = self.ids[p]
        return p
    def union(self,p,q):
        pID = self.find(p)
        qID = self.find(q)
        if pID == qID:
            return
        if self.size[pID] < self.size[qID]:
            self.ids[pID] = qID
            self.size[qID] += self.size[pID]
        else:
            self.ids[qID] = pID
            self.size[pID] += self.size[qID]
        self.count-=1

    def getcount(self):
        return self.count
if __name__ == '__main__':
    N,M = list(map(int,input().split()))
    # uf = Quick_Find(N)
    # uf = Quick_Union(N)
    uf = Weighted_Union_Find(N)
    for i in range(M):
        p,q = list(map(int,input().split()))
```

```
        if not uf.connect(p,q):
            uf.union(p,q)

    print(uf.getcount())
```

Task2 Bloom filter

```
import math
import hashlib
from struct import unpack, pack, calcsize

try:
    import bitarray
except ImportError:
    raise ImportError('pybloom requires bitarray >= 0.3.4')

__version__ = '1.1'
__author__  = "Jay Baird <jay@mochimedia.com>, Bob Ippolito <bob@redivi.com>,\
            Marius Eriksen <marius@monkey.org>,\
            Alex Brasetvik <alex@brasetvik.com>"

def make_hashfuncs(num_slices, num_bits):
    if num_bits >= (1 << 31):
        fmt_code, chunk_size = 'Q', 8
    elif num_bits >= (1 << 15):
        fmt_code, chunk_size = 'I', 4
    else:
        fmt_code, chunk_size = 'H', 2
    total_hash_bits = 8 * num_slices * chunk_size
    if total_hash_bits > 384:
        hashfn = hashlib.sha512
    elif total_hash_bits > 256:
        hashfn = hashlib.sha384
    elif total_hash_bits > 160:
        hashfn = hashlib.sha256
    elif total_hash_bits > 128:
        hashfn = hashlib.sha1
    else:
        hashfn = hashlib.md5
    fmt = fmt_code * (hashfn().digest_size // chunk_size)
    num_salts, extra = divmod(num_slices, len(fmt))
    if extra:
        num_salts += 1
    salts = [hashfn(hashfn(pack('I', i)).digest()) for i in range(num_salts)]
    def _make_hashfuncs(key):
        #if isinstance(key, unicode):
        #    key = key.encode('utf-8')
        #else:
        #    key = str(key)
        key = str(key).encode("utf-8")
        rval = []
        for salt in salts:
            h = salt.copy()
            h.update(key)
            rval.extend(uint % num_bits for uint in unpack(fmt, h.digest()))
        del rval[num_slices:]
```

```
        return rval
    return _make_hashfuncs


class BloomFilter(object):
    FILE_FMT = '<dQQQQ'

    def __init__(self, capacity, error_rate=0.001):
        """Implements a space-efficient probabilistic data structure

        capacity
            this BloomFilter must be able to store at least *capacity* elements
            while maintaining no more than *error_rate* chance of false
            positives
        error_rate
            the error_rate of the filter returning false positives. This
            determines the filters capacity. Inserting more than capacity
            elements greatly increases the chance of false positives.

        >>> b = BloomFilter(capacity=100000, error_rate=0.001)
        >>> b.add("test")
        False
        >>> "test" in b
        True

        """
        if not (0 < error_rate < 1):
            raise ValueError("Error_Rate must be between 0 and 1.")
        if not capacity > 0:
            raise ValueError("Capacity must be > 0")
        # given M = num_bits, k = num_slices, p = error_rate, n = capacity
        # solving for m = bits_per_slice
        # n ~= M * ((ln(2) ** 2) / abs(ln(P)))
        # n ~= (k * m) * ((ln(2) ** 2) / abs(ln(P)))
        # m ~= n * abs(ln(P)) / (k * (ln(2) ** 2))
        num_slices = int(math.ceil(math.log(1 / error_rate, 2)))
        # the error_rate constraint assumes a fill rate of 1/2
        # so we double the capacity to simplify the API
        bits_per_slice = int(math.ceil(
            (2 * capacity * abs(math.log(error_rate))) /
            (num_slices * (math.log(2) ** 2))))
        self._setup(error_rate, num_slices, bits_per_slice, capacity, 0)
        self.bitarray = bitarray.bitarray(self.num_bits, endian='little')
        self.bitarray.setall(False)

    def _setup(self, error_rate, num_slices, bits_per_slice, capacity, count):
        self.error_rate = error_rate
        self.num_slices = num_slices
        self.bits_per_slice = bits_per_slice
        self.capacity = capacity
        self.num_bits = num_slices * bits_per_slice
        self.count = count
        self.make_hashes = make_hashfuncs(self.num_slices, self.bits_per_slice)

    def __contains__(self, key):
        """Tests a key's membership in this bloom filter.

        >>> b = BloomFilter(capacity=100)
        >>> b.add("hello")
        False
```

```
        >>> "hello" in b
        True

        """
        bits_per_slice = self.bits_per_slice
        bitarray = self.bitarray
        if not isinstance(key, list):
            hashes = self.make_hashes(key)
        else:
            hashes = key
        offset = 0
        for k in hashes:
            if not bitarray[offset + k]:
                return False
            offset += bits_per_slice
        return True

    def __len__(self):
        """Return the number of keys stored by this bloom filter."""
        return self.count

    def add(self, key, skip_check=False):
        """ Adds a key to this bloom filter. If the key already exists in this
        filter it will return True. Otherwise False.

        >>> b = BloomFilter(capacity=100)
        >>> b.add("hello")
        False
        >>> b.add("hello")
        True

        """
        bitarray = self.bitarray
        bits_per_slice = self.bits_per_slice
        hashes = self.make_hashes(key)
        if not skip_check and hashes in self:
            return True
        if self.count > self.capacity:
            raise IndexError("BloomFilter is at capacity")
        offset = 0
        for k in hashes:
            self.bitarray[offset + k] = True
            offset += bits_per_slice
        self.count += 1
        return False

    def copy(self):
        """Return a copy of this bloom filter.
        """
        new_filter = BloomFilter(self.capacity, self.error_rate)
        new_filter.bitarray = self.bitarray.copy()
        return new_filter

    def union(self, other):
        """ Calculates the union of the two underlying bitarrays and returns
        a new bloom filter object."""
        if self.capacity != other.capacity or \
            self.error_rate != other.error_rate:
            raise ValueError("Unioning filters requires both filters to have \
both the same capacity and error rate")
```

```
        new_bloom = self.copy()
        new_bloom.bitarray = new_bloom.bitarray | other.bitarray
        return new_bloom

    def __or__(self, other):
        return self.union(other)

    def intersection(self, other):
        """ Calculates the union of the two underlying bitarrays and returns
        a new bloom filter object."""
        if self.capacity != other.capacity or \
            self.error_rate != other.error_rate:
            raise ValueError("Intersecting filters requires both filters to \
have equal capacity and error rate")
        new_bloom = self.copy()
        new_bloom.bitarray = new_bloom.bitarray & other.bitarray
        return new_bloom

    def __and__(self, other):
        return self.intersection(other)

    def tofile(self, f):
        """Write the bloom filter to file object `f'. Underlying bits
        are written as machine values. This is much more space
        efficient than pickling the object."""
        f.write(pack(self.FILE_FMT, self.error_rate, self.num_slices,
                     self.bits_per_slice, self.capacity, self.count))
        self.bitarray.tofile(f)

    @classmethod
    def fromfile(cls, f, n=-1):
        """Read a bloom filter from file-object `f' serialized with
        ``BloomFilter.tofile''. If `n' > 0 read only so many bytes."""
        headerlen = calcsize(cls.FILE_FMT)

        if 0 < n < headerlen:
            raise ValueError('n too small!')

        filter = cls(1)  # Bogus instantiation, we will `_setup'.
        filter._setup(*unpack(cls.FILE_FMT, f.read(headerlen)))
        filter.bitarray = bitarray.bitarray(endian='little')
        if n > 0:
            filter.bitarray.fromfile(f, n - headerlen)
        else:
            filter.bitarray.fromfile(f)
        if filter.num_bits != filter.bitarray.length() and \
               (filter.num_bits + (8 - filter.num_bits % 8)
                != filter.bitarray.length()):
            raise ValueError('Bit length mismatch!')

        return filter

    def __getstate__(self):
        d = self.__dict__.copy()
        del d['make_hashes']
        return d

    def __setstate__(self, d):
        self.__dict__.update(d)
        self.make_hashes = make_hashfuncs(self.num_slices, self.bits_per_slice)
```

```
class ScalableBloomFilter(object):
    SMALL_SET_GROWTH = 2 # slower, but takes up less memory
    LARGE_SET_GROWTH = 4 # faster, but takes up more memory faster
    FILE_FMT = '<idQd'

    def __init__(self, initial_capacity=100, error_rate=0.001,
                 mode=SMALL_SET_GROWTH):
        """Implements a space-efficient probabilistic data structure that
        grows as more items are added while maintaining a steady false
        positive rate

        initial_capacity
            the initial capacity of the filter
        error_rate
            the error_rate of the filter returning false positives. This
            determines the filters capacity. Going over capacity greatly
            increases the chance of false positives.
        mode
            can be either ScalableBloomFilter.SMALL_SET_GROWTH or
            ScalableBloomFilter.LARGE_SET_GROWTH. SMALL_SET_GROWTH is slower
            but uses less memory. LARGE_SET_GROWTH is faster but consumes
            memory faster.

        >>> b = ScalableBloomFilter(initial_capacity=512, error_rate=0.001, \
                                    mode=ScalableBloomFilter.SMALL_SET_GROWTH)
        >>> b.add("test")
        False
        >>> "test" in b
        True
        >>> unicode_string = u'¡'
        >>> b.add(unicode_string)
        False
        >>> unicode_string in b
        True
        """
        if not error_rate or error_rate < 0:
            raise ValueError("Error_Rate must be a decimal less than 0.")
        self._setup(mode, 0.9, initial_capacity, error_rate)
        self.filters = []

    def _setup(self, mode, ratio, initial_capacity, error_rate):
        self.scale = mode
        self.ratio = ratio
        self.initial_capacity = initial_capacity
        self.error_rate = error_rate

    def __contains__(self, key):
        """Tests a key's membership in this bloom filter.

        >>> b = ScalableBloomFilter(initial_capacity=100, error_rate=0.001, \
                                    mode=ScalableBloomFilter.SMALL_SET_GROWTH)
        >>> b.add("hello")
        False
        >>> "hello" in b
        True

        """
        for f in reversed(self.filters):
            if key in f:
```

```
            return True
        return False

    def add(self, key):
        """Adds a key to this bloom filter.
        If the key already exists in this filter it will return True.
        Otherwise False.

        >>> b = ScalableBloomFilter(initial_capacity=100, error_rate=0.001, \
                                    mode=ScalableBloomFilter.SMALL_SET_GROWTH)
        >>> b.add("hello")
        False
        >>> b.add("hello")
        True

        """
        if key in self:
            return True
        filter = self.filters[-1] if self.filters else None
        if filter is None or filter.count >= filter.capacity:
            num_filters = len(self.filters)
            filter = BloomFilter(
                capacity=self.initial_capacity * (self.scale ** num_filters),
                error_rate=self.error_rate * (self.ratio ** num_filters))
            self.filters.append(filter)
        filter.add(key, skip_check=True)
        return False

    @property
    def capacity(self):
        """Returns the total capacity for all filters in this SBF"""
        return sum([f.capacity for f in self.filters])

    @property
    def count(self):
        return len(self)

    def tofile(self, f):
        """Serialize this ScalableBloomFilter into the file-object
        `f'."""
        f.write(pack(self.FILE_FMT, self.scale, self.ratio,
                     self.initial_capacity, self.error_rate))

        # Write #-of-filters
        f.write(pack('<l', len(self.filters)))

        if len(self.filters) > 0:
            # Then each filter directly, with a header describing
            # their lengths.
            headerpos = f.tell()
            headerfmt = '<' + 'Q'*(len(self.filters))
            f.write('.' * calcsize(headerfmt))
            filter_sizes = []
            for filter in self.filters:
                begin = f.tell()
                filter.tofile(f)
                filter_sizes.append(f.tell() - begin)

            f.seek(headerpos)
            f.write(pack(headerfmt, *filter_sizes))
```

```
    @classmethod
    def fromfile(cls, f):
        """Deserialize the ScalableBloomFilter in file object `f'."""
        filter = cls()
        filter._setup(*unpack(cls.FILE_FMT, f.read(calcsize(cls.FILE_FMT))))
        nfilters, = unpack('<l', f.read(calcsize('<l')))
        if nfilters > 0:
            header_fmt = '<' + 'Q'*nfilters
            bytes = f.read(calcsize(header_fmt))
            filter_lengths = unpack(header_fmt, bytes)
            for fl in filter_lengths:
                filter.filters.append(BloomFilter.fromfile(f, fl))
        else:
            filter.filters = []

        return filter

    def __len__(self):
        """Returns the total number of elements stored in this SBF"""
        return sum([f.count for f in self.filters])


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```