Algorithms and Data Structures

Hashing (Ch. 5)

Morgan Ericsson

Today

- Hashing
  - Seperate chaining
  - Probing
  - Double hashing
  - Rehashing

Hashing

Question

- Assume we have a list of integers
- We can find a given integer
  - $O(N)$ if we use sequential search
  - $O(\log_2 N)$ if we use a binary search tree
- Can we do better?
  - Think back to union find/disjoint set

Binary list

- Let's try to adopt the idea from union find
  - But now we only care if a number is in the list or not
- So, if we want to insert a number
  - `lst[6433] = True`
- If we want to check
  - `lst[2137] is not None`
- Reasonable idea, but how larger should the list be?
- How much space is wasted?

Flawed but good idea

- We use the integer value to map a key to a position
- The problem is the range of the random integers
- If we wanted to store all numbers 1 to 100 it would not be a problem
- But in the general case, the numbers might be to large and the list would be very sparse

Flawed but good idea

```python
import numpy as np

lst = np.zeros(7630+1, dtype=bool)

for v in [7630, 3275, 6433, 5913, 2137]:
  lst[v] = True

print(len(lst[lst == True]))
print(len(lst[lst == False]))
```

```
5
7626
```

General idea

- We use the key to map to an element
- If unlimited space, we can just use the value
- If less then limited space, we need a “better” mapping function
  - What if we use %?

Another way to map

```
1 lst_sz = 5
2 lst = np.zeros(lst_sz, dtype=bool)
3
4 for v in [7630, 3275, 6433, 5913, 2137]:
5   lst[v % lst_sz] = True
6
7 print(len(lst[lst == True]))
8 print(len(lst[lst == False]))
```

```
3
2
```

Hash

- The mapping is called a hashing function or hash(Code)
- The problem we observed is a collison
  - When two (or more) keys map to the same position
- A perfect hashing function should never produce collisions
  - but it can be difficult to define
- Especially since it should also be efficient to compute

Another try

```
 1  lst_sz = 31
 2  lst = np.zeros(lst_sz, dtype=bool)
 3
 4  def hashf(v):
 5    return v % lst_sz
 6
 7  for v in [7630, 3275, 6433, 5913, 2137]:
 8    lst[hashf(v)] = True
 9
10  print(len(lst[lst == True]))
```

```
5
```

More problems

- Without perfect hashing, we can never find the key
  - Hashing is a one-way function
  - `99 % 10 == 9 % 10`
- There is no order
- What if they keys are not integers?

Another example

```
1  lst_sz = 31
2  lst = np.zeros(lst_sz, dtype=bool)
3
4  def hashf(v:str) -> int:
5    pass
6
7  for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
8    lst[hashf(v)] = True
```

Options

- Use the first (or last) k letters
- Can be a bad idea, many domains have common pre- and suffixes
  - Names
  - Phone numbers
  - …
- Use the whole key?

Example

```
1  def hashf(key:str) -> int:
2    hv = 0
3    for c in key:
4      hv += ord(c)
5
6    return hv
```

Example

```
1 lst_sz = 31
2 lst = np.zeros(lst_sz, dtype=bool)
3
4 for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
5   lst[hashf(v) % lst_sz] = True
6
7 print(len(lst[lst == True]))
```

All is good?

```
1 print(hashf('abc') == hashf('acb'))
```

```
True
```

Problem? Depends on our use... Not great for phone numbers, for example...

What can we do?

```
1  def hashf(key:str) -> int:
2    hv = 0
3    for ix, c in enumerate(key, start=2):
4      hv += ix * ord(c)
5
6    return hv
```

Solved?

```
1 print(hashf('abc') == hashf('acb'))
2 print(hashf('abc') == hashf('-10['))
```

```
False
True
```

It is not easy to create a perfect (good) hash function.

Keep in mind

- Try to avoid repetition and “round” numbers
  - It is generally a bad idea to use sizes of even 10s
  - Or powers of two
  - Use prime numbers to break patterns
  - Preferably close to powers of two
- Use as much of the key as possible
  - More bits means more variation

Hashing in Python (and Java)

- Types define a hash value
  - `hash('Olivia')`
  - `hash((1, 3))`
- Similar in Java
- Required for certain things to work, e.g., sets

Simple example

```
 1 class Person:
 2   def __init__(self, n:str, a:int) -> None:
 3     self.name = n
 4     self.age = a
 5
 6   def __str__(self) -> str:
 7     return f'{self.name} ({self.age})'
 8
 9 p1 = Person('Olivia', 34)
10 print(hash(p1))
```

```
708755569
```

For free?

```
1 p1 = Person('Olivia', 34)
2 p2 = Person('Olivia', 34)
3
4 print(hash(p1) == hash(p2))
```

```
False
```

Hash is based on object identity. Problem? Depends on our use...

For free?

```
1  from fastcore.basics import patch
2
3  @patch
4  def __hash__(self:Person) -> int:
5    hv = 17
6    hv = 31 * hv + hash(self.name)
7    hv = 31 * hv + hash(self.age)
8    return hv
```

We can define our own hash function

New try

```
1 p1 = Person('Olivia', 34)
2 p2 = Person('Olivia', 34)
3
4 print(hash(p1) == hash(p2))
```

```
True
```

Based on object values rather than identify

Using our function

```
1 plst = [None] * 31
2
3 p1 = Person('Olivia', 34)
4 p2 = Person('Mia', 11)
5
6 plst[hash(p1) % 31] = p1
7 plst[hash(p2) % 31] = p2
8
9 print(plst[hash(p1) % 31])
```

```
Olivia (34)
```

Suppose Olivia had a birthday ...

```
1 p1.age += 1
2
3 print(plst[hash(p1) % 31])
```

```
None
```

Custom hash functions and mutability is a problem...

Simple hash table

```
 1  class HT:
 2    def __init__(self):
 3      self.sz = 31
 4      self.table = [None] * 31
 5
 6    def insert(self, key):
 7      self.table[hash(key) % self.sz] = key
 8
 9    def contains(self, key):
10      return self.table[hash(key) % self.sz] == key
11
12    def __len__(self):
13      return len([v for v in self.table \
14                  if v is not None])
```

Testing it

```
1  import random
2
3  ht = HT()
4  for i in range(10):
5    v = random.randint(1, 100_000)
6    ht.insert(v)
7
8  print(len(ht))
```

```
10
```

Seperate chaining

Collisions

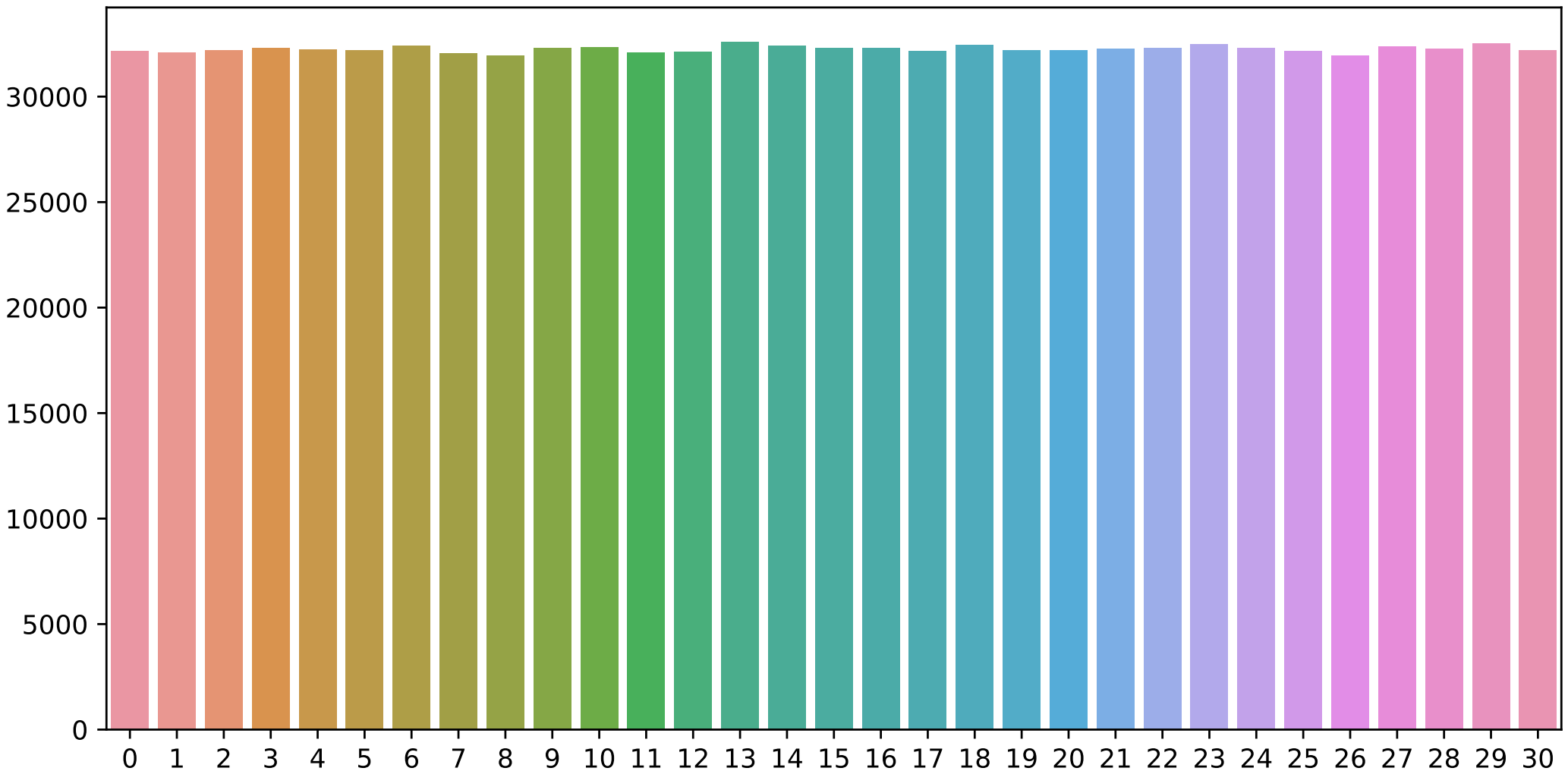
- We have seen that some hashing functions can result in collisions
- Can we manage the collisions?

Hash functions

- We want the hash functions to distribute keys uniformely across the integer interval
- Bins and balls
  - Assume that each key is a ball and each position is a bin
  - If we randomly toss a random ball, it should be equally likely to end up in any of the bins
  - If have m bins and toss n balls, we would expect there to be n/m balls in each bin after a while

Uniformity

Tossing 1 000 000 balls into 31 bins



We also know

- We can expect two balls in the same bin after $\sim \sqrt{\pi \frac{m}{2}}$ tosses
- Every bin has ≥ 1 balls after $\sim m \ln m$ tosses
- After m tosses, the most loaded bin has $\Theta(\frac{\log m}{\log \log m})$ balls

So, how can we deal with collisions?

- Seperate chaining
- We make each bin a linked list
- And place keys that collide in the same bin

A simple linked list

```
1  from dataclasses import dataclass
2
3  @dataclass
4  class LLNode:
5    key: int
6    nxt: 'LLNode|None' = None
```

Seperate chaining

```
1  class HTSC:
2    def __init__(self, m=5):
3      self.sz = m
4      self.table = [None] * self.sz
```

Inserting

```
 1  @patch
 2  def insert(self:HTSC, key):
 3    hv = hash(key) % self.sz
 4    self.table[hv] = self._inschain(self.table[hv], key)
 5
 6  @patch
 7  def _inschain(self:HTSC, n:LLNode|None, key) -> LLNode:
 8    if n is None:
 9      return LLNode(key)
10    n.nxt = self._inschain(n.nxt, key)
11    return n
```

Finding

```
 1  @patch
 2  def find(self:HTSC, key) -> bool:
 3    hv = hash(key) % self.sz
 4    if self.table[hv] is not None:
 5      p = self.table[hv]
 6      while p is not None:
 7        if p.key == key:
 8          return True
 9        p = p.nxt
10    return False
```

Len

```
 1  @patch
 2  def __len__(self:HTSC) -> int:
 3    l = 0
 4    for t in self.table:
 5      l += self._lchain(t)
 6    return l
 7
 8  @patch
 9  def _lchain(self:HTSC, n:LLNode|None) -> int:
10    l = 0
11    while n is not None:
12      l += 1
13      n = n.nxt
14    return l
```

Testing it

```
1 ht = HTSC()
2
3 for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
4   ht.insert(v)
5
6 print(len(ht))
```

```
5
```

Testing it some more

```
1 ht = HTSC()
2 for i in range(200):
3   v = random.randint(1, 100_000)
4   ht.insert(v)
5
6 print(len(ht))
```

```
200
```

Linear again?

- With seperate chaining, we need to search the list to determine if the value exists or not
- We know that each list holds on average n / m
  - Where n is the number of keys and m is the size
- So, can be significant better than than $\mathrm{O(N)}$

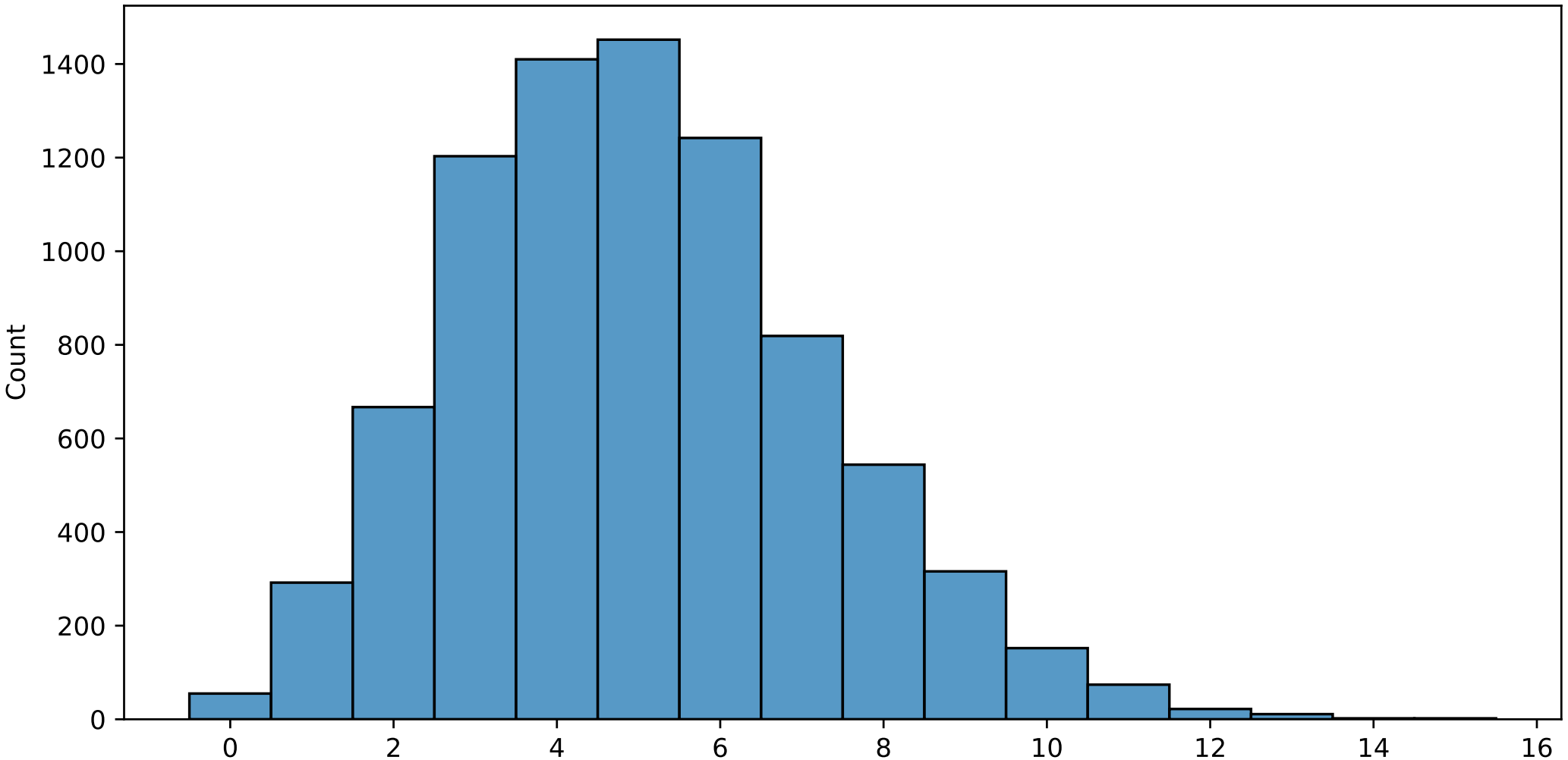
How should we pick *m*?

- If *m* is too small, the lists will be too long
- If *m* is too large, we will waste space
- A good rule of thumb is to set *m* to *n* / 5
  - Then access will be $O(1)$

Testing the idea

```
1  n = 8263 * 5
2  m = 8263
3
4  ht = HTSC(m)
5  for i in range(n):
6    v = random.uniform(0, 1)
7    ht.insert(v)
8
9  ll = [ht._lchain(n) for n in ht.table]
```

Testing the idea



Linear probing

Linear probing

- Seperate chaining works, but:
  - Introduces a second data structure
  - Has overhead in creating nodes
- What if we “chain” in the existing list?

Linear probing

- If a slot is taken, find the next empty one
  - If $\mathrm{hash}(\mathrm{v}) = \mathrm{i}$ and i is taken, try $\mathrm{i} + 1, \mathrm{i} + 2, \ldots$ until an empty slot is found
- Must repeat the same when searching…
- The list must be larger than the number of keys

Linear probing

```
1  class HTLP:
2    def __init__(self, m=5):
3      self.sz = m
4      self.table = [None] * self.sz
```

Inserting

```
1  @patch
2  def insert(self:HTLP, key):
3    hv = hash(key) % self.sz
4    if self.table[hv] is None:
5      self.table[hv] = key
6    else:
7      while self.table[hv] is not None:
8        hv = (hv + 1) % self.sz
9      self.table[hv] = key
```

Finding

```
1  @patch
2  def find(self:HTLP, key) -> bool:
3    hv = hash(key) % self.sz
4    while self.table[hv] is not None:
5      if self.table[hv] == key:
6        return True
7      hv = (hv + 1) % self.sz
8    return False
```

Len

```
1  @patch
2  def __len__(self:HTLP) -> int:
3    return len([v for v in self.table \
4                  if v is not None])
```

Testing it...

```
1 ht = HTLP(7)
2
3 for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
4   ht.insert(v)
5
6 assert ht.find('Liam') == True
7 assert ht.find('John') == False
8 print(len(ht))
```

```
5
```

Knuth's parking problem

- Cars arrive at a (one-way) street with m parking spaces
- Each car desires a specific space i, but will try $i + 1$, $i + 2$, ... if i is taken
- What is the average displacement?
  - with $m / 2$ cars, $\sim 3 / 2$
  - with m cars, $\sim \sqrt{\pi\, m / 8}$

Analysis of linear probing

- Assume we have a list of size m and $n = \alpha m$ keys
- We can then determine the average number of probes if we have a search hit

$$\frac{1}{2}(1 + \frac{1}{1 - \alpha})$$

Analysis of linear probing

» And if we miss/insert

$$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

Analysis of linear probing

- If m is too large, too much wasted space
- If m is too small, search time blows up
- Rule of thumb, $\alpha = n / m \sim 1 / 2$
  - Probes for hit is about $3 / 2$
  - Probes for miss/insert is about $5 / 2$

Testing it

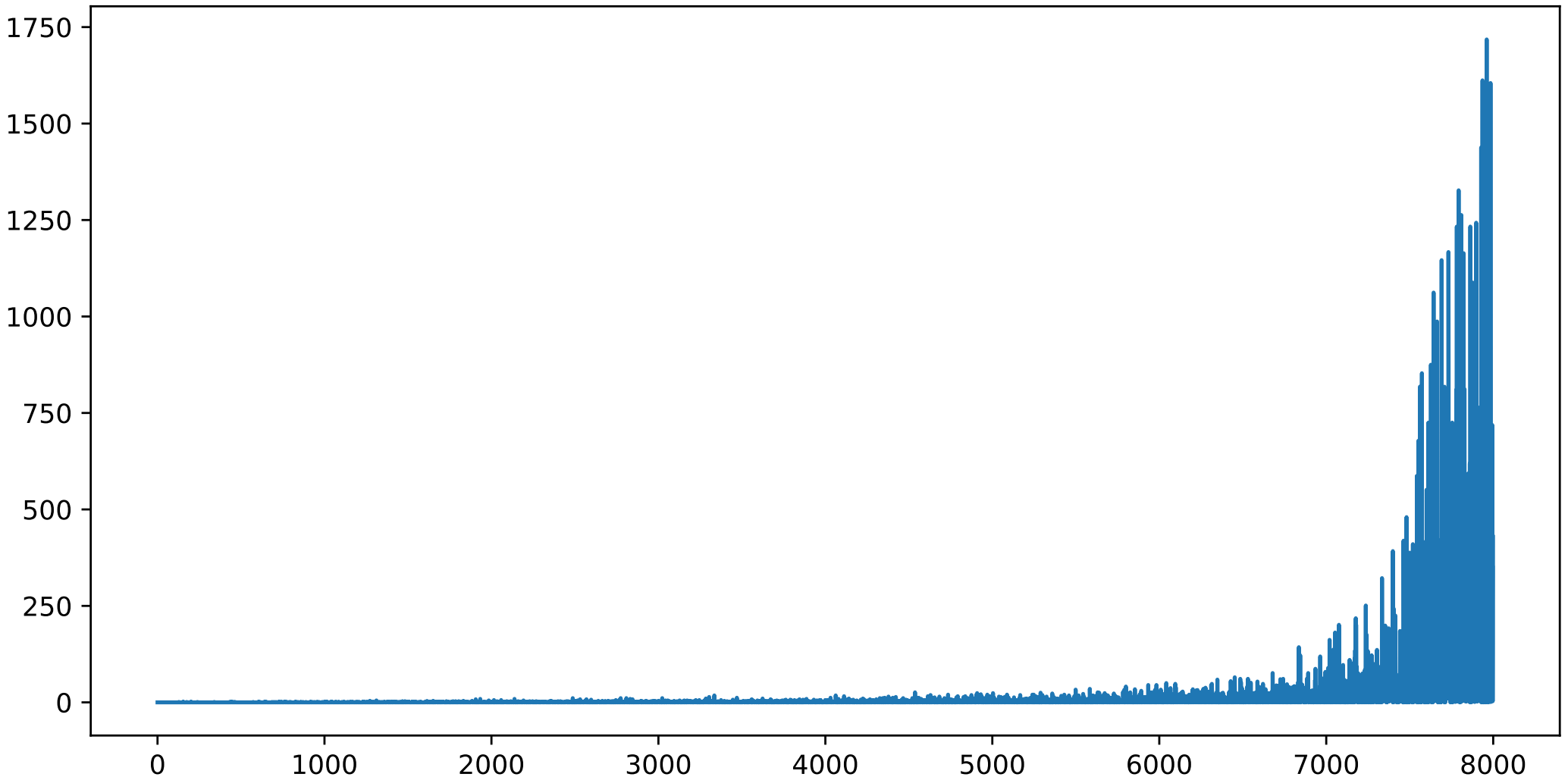
```
 1 @patch
 2 def insert(self:HTLP, key) -> int:
 3   hv = hash(key) % self.sz
 4   if self.table[hv] is None:
 5     self.table[hv] = key
 6     return 0
 7   else:
 8     off = 0
 9     while self.table[hv] is not None:
10       hv = (hv + 1) % self.sz
11       off += 1
12     self.table[hv] = key
13     return off
```

Testing it

```
1  ht = HTLP(7)
2
3  for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
4    print(ht.insert(v))
```

```
0
0
0
0
1
```

Testing it some more



So far

- We can manage collisions with seperate chaining or linear probing
- Within constant time on average
- But logarithmic time worst case
- Assuming uniform hasing

We can do more

- Quadratic probing
- Double hashing
- ...

Reading instructions

Reading instructions

- Ch. 5.1 - 5.5
- Interesting, but not required
  - 5.6 discusses hashing in Java
  - 5.7 discusses more advanced versions of hasing
  - 5.8 discusses universal hash functions
  - 5.9 discusses hashing to secondary storage