CSCM712: Solving software and efficiency problem
Hashing, for hashtables

Cécilia Pradic

March 23rd 2026

What is hashing? (technically correct but desperately incomplete answer)

A **hash** function takes a set of values into a fixed range of integers.

Sets of values are typically "objects of some specified type/class".

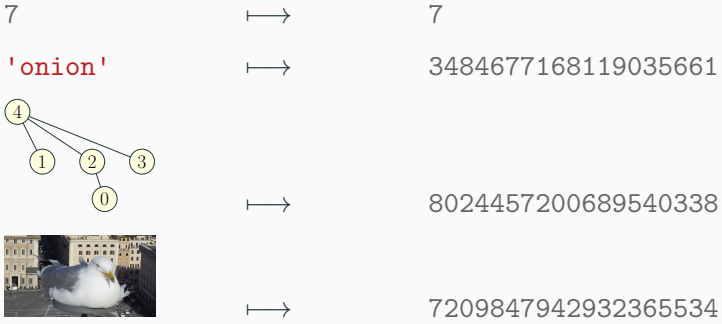


Figure 1: a number, a string, a tree and a png hashed into numbers

What is hashing, really?

A **hash** function takes a set of values into a fixed range of integers.

A *good* hash function $h : U \to \{0, \ldots, 2^{64} - 1\}$ should (at least):

- be extremely efficient ($\mathcal{O}(1)$) (and deterministic)
- scramble evenly the inputs in the range
  - i.e. the probability of $h(x) = h(y)$ should be small if $x \neq y$
  - a pair (x, y) with $x \neq y$ and $h(x) = h(y)$ is called a **collision**

What is hashing, really?

A **hash** function takes a set of values into a fixed range of integers.

A *good* hash function $h : U \to \{0, \ldots, 2^{64} - 1\}$ should (at least):

- be extremely efficient ($\mathcal{O}(1)$) (and deterministic)
- scramble evenly the inputs in the range
  - i.e. the probability of $h(x) = h(y)$ should be small if $x \neq y$
  - a pair (x, y) with $x \neq y$ and $h(x) = h(y)$ is called a **collision**

Sometimes we also want other guarantees of the same flavor.

Why is this a concept?

Hashing functions can be used to:

- short, plausible IDs (**digests**)
  - example: certifying integrity of downloads
- for all sorts of **cryptographic protocols** (signing, etc)
  - for this you also want adversarial hardness of generating collisions
  - i.e., that there be no efficient probabilistic algorithm $\mathcal{A}$ such that $\mathcal{A}(x)$ returns y with $x \neq y$ and $h(x) = h(y)$ with non-negligible probability.
- to implement **hash tables** (python's *dictionaries*)
- and python's sets (just use the keys, not the values)

(sorry I inadvertantly lied! python sets are not based on AVLs, I misread something)

```
{ 'seagull' : [1, 4], 'anemone' : [5,3,6,7], 'cat' : [] }
```

Why is this a concept?

Hashing functions can be used to:

- short, plausible IDs (**digests**)
  - example: certifying integrity of downloads
- for all sorts of **cryptographic protocols** (signing, etc)
  - for this you also want adversarial hardness of generating collisions
  - i.e., that there be no efficient probabilistic algorithm $\mathcal{A}$ such that $\mathcal{A}(x)$ returns y with $x \neq y$ and $h(x) = h(y)$ with non-negligible probability.
- to implement **hash tables** (python's *dictionaries*)
- and python's sets (just use the keys, not the values)

  (sorry I inadvertantly lied! python sets are not based on AVLs, I misread something)

```
{ 'seagull' : [1, 4], 'anemone' : [5,3,6,7], 'cat' : [] }
```

In this module, we will only explore the last two aspects.

What is a hash table?

A collection of *values* indexed by unique *keys*

- where keys are not necessarily numbers
- such that the operations below are **almost surely** $\mathcal{O}(1)$

  (for this, we need a good hash function for keys)
- (taking up a reasonable amount of space)

```
class hashTable:
  def __init__(self, capacity = 50):
  # create a new empty hash table
  def __getitem__(self, key):
  # retrieve an item
  def __setitem__(self, key, value):
  # store an item at a given key
  def __contains__(self, key):
  # is key in the hash table?
```

Implementing a hash table

Naive, hopeful, collision-free

```
class hopelessHashTable:
  def __init__(self, capacity = 50):
     self.table = [ None for _ in range(0, 2 * capacity)]
  def __getitem__(self, key):
     return self.table[hash(key) % len(self.table)]
  def __setitem__(self, key, value):
     self.table[hash(key) % len(self.table)] = value
  def __contains__(self, key):
      return self.table[hash(key) % len(self.table)] != None
```

But of course collisions happen sometimes

Let's add a dove, a magpie, a gull and a swan to a hashtable.

```
hash('dove') % 4 = 2
hash('magpie') % 4 = 3
hash('gull') % 4 = 0
hash('swan') % 4 = 3
```

- even if collisions are rare, we need to handle them properly

(raising an exception is not an option)

One resolution: chaining

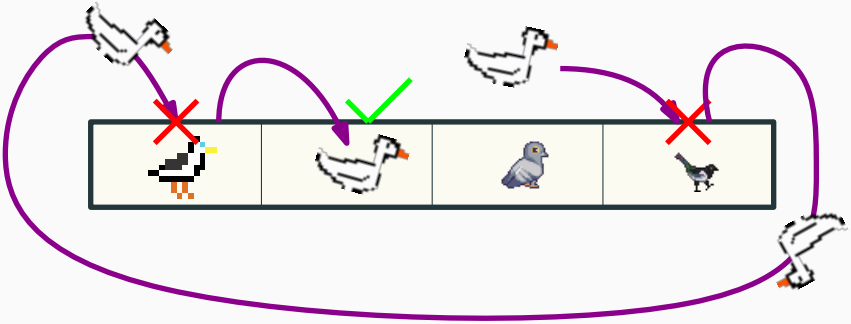
Idea: store a *set* instead of a single key

- if there are collisions, `__getitem__` still need to go through the whole bucket
- traditionally a bucket is implemented by a linked list

```
class chainingHashTable:
  def __init__(self, capacity = 50):
     self.table = [ {} for _ in range(0, 2 * capacity)]
  def __getitem__(self, key):
     for (k, v) in self.table[hash(key) % len(self.table)]:
       if k == key:
         return v
     raise KeyError(k)
  # ... and more
```

Another resolution: linear probing

Idea: if the spot is occupied by something else, try the next spot until you find something or an unoccupied spot.



Linear probing: looking up

Alas, then the index of a key k might be different from `hash`(k) mod `len`.

Example of clustering

Adding a, dove, magpie, swan and then gull:

idx	0	1	2	3	4	5	6	7
hash	0			3	4	4	3	
key	rat			dove	magpie	swan	gull	
val	22			55	21	54	785	

What this means for lookup

Probe the next cell until empty.

E.g. to lookup gull, the whole cluster (cells 3 to 6) needs to be traversed.

Linear probing: additional complication: deletion

To delete some key, we need to leave a **tombstone mark** ⊓

- ⊓ counts as occupied for lookup
- ⊓ counts as free when adding an item

Why?

Consider the previous table where swan and rat were deleted.

idx	0	1	2	3	4	5	6	7
hash				3	4		3	
key	⊓			dove	magpie	⊓	gull	
val				55	21		785	

What happens when looking up gull? Setting swan again?

Probing: other strategies

- Linear probing = (next cell = `lambda x: (x + 1) % len(table)`)
- Otherwise, the construction describe works for alternative strategies

(the book discusses for instance quadratic probing)

How to compare probing strategies

See the impact on **clustering**, which leads to more expensive lookups.

Size of the internal table

Important metric

$$\frac{\#\text{ stored keys}}{\text{size(table)}} \quad = \quad \text{load factor } \lambda$$

- if $\lambda \geq 1$, you **will** have collisions
- the smaller λ, the less collisions you'll have

(assuming your hash function is sensible and you are not unlucky)

Necessary conditions to have $\mathcal{O}(1)$ operation on average

$\lambda < 1$ and low probability of collision for `hash`

Corollary

When λ grows too large, a hash table needs **rehashing**

- allocate a new table, reinsert everything $\mathcal{O}(n)$
- (similar rationale as dynamic array; should not be done often)

Demo time

Let's play with `hashing.py` on canvas.

It contains:

- a naive hash table class without collision handling
- a chaining hash table class without the `delitem` method
- a probing hash table class without the `delitem` method
- a helper function and some tests

Lab 9

You will have to implement `delitem` for the last two.

OK but what is `hash`?

Python's hash function

To build dictionaries, python uses hash.

```
class bla:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __hash__(self):
    return x + 31 * y

  def __eq__(self, other):
    return self.x == other.x and self.y == other.y

assert(hash(bla(5, 1)) == 36)
```

- hash(x) gets resolved to the __hash__ method of x.
- if __hash__ is defined, so should __eq__

Hashing for built-in python datatypes

Implemented for:

- `str`, `int`, `float`
- tuples (assuming the components are hashable)

Not implemented for

- sets, lists, dictionaries

For new classes

- default implementation
- but only based on the reference (location in memory)

Hashing for built-in python datatypes

Implemented for:

- `str`, `int`, `float`
- tuples (assuming the components are hashable)

Not implemented for

- sets, lists, dictionaries

For new classes

- default implementation
- but only based on the reference (location in memory)

$\Longrightarrow$ we typically want to override that

How is the hash computed for built-ins?

In python:

- for `int`, `hash(n) = n`
- for tuples and strings, I am not sure
  - (lookup the source code of CPython if you want)
  - note that currently the hash get **salted** with a random value

  → you won't be able to reproduce runs

(unless you set the environment variable `PYTHONHASHSEED` to 0)

Java implementation of hash for string

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++)
            h = 31 * h + val[i];
        hash = h;
    }
    return h;
}
```

- Note the hash value is **cached**
  - $\mathcal{O}(n)$ on the first run, $\mathcal{O}(1)$ afterwards
- Essentially pick a prime number, regard your string as a list of numbers, and evaluate a polynomial
- Overflows just trigger an implicit modulo

What about our custom classes?

Two basic things you should ensure:

- define and be consistent with `__eq__` so that

$$\texttt{x == y} \quad \Longrightarrow \quad \texttt{hash(x) == hash(y)}$$

- keep your hash function efficient (i.e., $\mathcal{O}(1)$; most of the time)
  - if dealing with a recursive class
    $\Longrightarrow$ cache the hash in an attribute to avoid recomputation
  - corollary: recompute when a method **mutates** the content (e.g. `__setitem__`) (might be tricky!)

Vague heuristics (to minimize collisions)

- Make `hash` vary with *all* things that influence ==
- Re-use existing `hash` for labels
- Re-use existing `hash` for *tuples* to bundle stuff together
- Don't hesitate to `hash` something built out of `hash`s

Example # 1

```
class bla:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __hash__(self):
    return hash(self.x, self.y) # reusing hash for tuples
                                # (and the __hash__ methods of x and y)
  def __eq__(self, other):
    return self.x == other.x and self.y == other.y

assert(hash(bla(5, 1)) == 36)
```

Example # 2

```
class btree:
  def __init__(self, x, l = None, r = None):
    self.label = x
    self.left = l
    self.right = r
    self.hash = self.hashMyself()

  def hashMyself():  # implicitly recursive
    return hash((self.label, self.left, self.right))

  def __hash__(self):
    return self.hash

  def __eq__(self, other):
    return hash(self) == hash(other) and\ # probable shortcut when !=
           blablablabla
```

Example # 2 continued: mutability

```
class binarySearchTree(btree):
  def insert(self, k):
     # here you should do something recursive, afterwhich you know
     # hash is set correctly in all subtrees
     self.hash = self.hashMyself()
     # and then you need to recompute the cached hash!
```

Is it provably the best?

Probably not.

- If you want to optimize, you really need to have a precise idea of what is your possible set of keys for your application.
- You may also want to optimize a the efficiency of `__hash__` rather than meet some theoretical bound on the probability of collisions

Is it provably the best?

Probably not.

- If you want to optimize, you really need to have a precise idea of what is your possible set of keys for your application.
- You may also want to optimize a the efficiency of `__hash__` rather than meet some theoretical bound on the probability of collisions

Some keywords to read upon for some theory

Universal hashing and **Perfect hashing**

Suggested applications (possibly for the lab) (you may have other in mind)

- countsort for non-contiguous sets of numbers
- dynamic programming/memoization
- optimizing slightly equality checks
- **hash-consing**: setting up tree-like datastructures for **maximal sharing**

Did we gain transferrable knowledge for crypto/security?

NO!

In crypto contexts

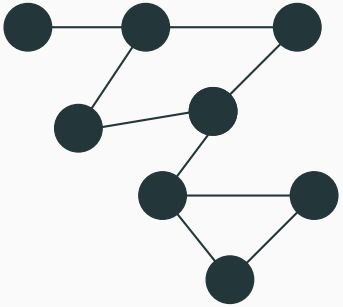
- You don't only care about **random** collisions ...
- ... but **adversarial** collisions
- i.e., is it possible for an attacker to efficiently generate a collision

Clearly, `hash` in python is not meant to be secure in that way.

Look up the documentation with `hashlib` and proceed with caution (maybe actually know something about cryptography, at least as a user)

What next?

Either something about graphs.



Or we jump to questions and/or revisions!