6.006 Recitation

Build 2008.10
Coming Up Next...

- Hashing in theory and in Python
- Bad hash functions
- Mutable dictionary keys
- Hashes for basic data types in Python
Why Hashing

- Useless from a theoretical standpoint
- $O(N)$ / op worst-case, not fit for proofs
- Used everywhere (dictionaries, indices)
- $O(1)$ / op is smokin’ hot / fast
- Simple - small constant factor
- Relies on black magic
Hashing pwns BSTs?

- **BSTs**
  - $O(\log(N)) / \text{op}$
  - guaranteed upper bound (worst-case)
  - comparison model (an order relation on keys is sufficient)
  - pwns in real-time

- **Hashing**
  - $O(1) / \text{op \ avg-case}$
  - no guarantees for worst-case -- $O(N)$
  - intimate knowledge of keys (via magic inside the hash function)
  - rocks for most cases
Real Life Hashing I

- Application: Keeping library cards
- 4x6” card for each book
- filing by the 1st letter of the book title
  - e.g. “Differential Equations” goes to D
- no sorting asides from mechanism above
Real Life Hashing II

- **filing** is uncool, let’s think of **bucketing**
- 26 buckets, labeled ‘A’ - ‘Z’
- Books are bucketed by 1st letter in title
- Time to find a book ~ bucket size
Real Life Hashing III

• What sucks in the scheme above?
  • Common prefixes
    • “The …”, “Introduction to…”
  • Uneven distribution
    • Many words start with E
    • Few words start with X
Real Life Hashing IV

- Solutions to issues above?
- Ignore “The...”, “Introduction...”
  - e.g. bucket “The Invisibles” under I
- Break up E’s bucket: ‘Ea-Em’, ‘En-Ez’
- Merge X’s bucket with W/Y
- Bucketing function gets hairy :( 
Hashing in Codeworld

- Memory is a block of cells
- Buckets are numbered 0 to N-1
- Each bucket is a list of the objects in it
- Fancy name for the bucketing method: hashing function
Hash Functions

• Theory
  • Maps the universe of keys to small (bounded) numbers

• Practice
  • Black magic that allows us to beat the log(N) theoretical bound on a daily basis
Good Hash Functions

- Convenient universe size (16/32/64-bit ints)
- Uniform distribution of keys
  - No obvious bad behavior
- Correct
  - Equal keys always hash to the same value
- Fast
Hashing Hall of Shame

• String hashing
  • numeric code for first letter
  • sum of numeric code for all letters
  • permutations hash to the same value
• polynomial value: \( \Sigma str[i] \cdot 256^i \)
  • grows without bound
Hashing Hall of Shame

• String hashing II
  
  • \((\sum \text{str}[i] \cdot 256^i) \mod 2^{32}\)
  
  • takes \(N^2\) to compute
  
  • \((\sum \text{str}[i] \cdot (256^i \mod 2^{32})) \mod 2^{32}\)
  
  • only takes first 4 letters into account
  
  • still sucks for table sizes = powers of 2
Hashing Wisdom

- Good functions are hard to come up with
- Use built-in functions whenever possible
Python Hashing 101

• Want hash() to work for your own objects?
  • def __hash__(self)
    • hash to a 32-bit number, not -1

• Want your objects as dictionary keys?
  • def __eq__(self, other)
    • return True/False (self equals other?)
Application: Screw Python

• I want lists as dictionary keys!

• Plan:

  1. SuperList object, encapsulating a list
  2. implement ___hash___ and ___eq___
  3. prepare Turing award acceptance speech
Behold, it’s SuperList!!!

```python
1 def make32(x):
2     x = x % (2**32)
3     if x >= 2**31: x = x - 2**32
4     return int(x)
5 class SuperList(object):
6     def __init__(self, list):
7         self.list = list
8     def __hash__(self):
9         m = 1000003
10        x = 0x345678
11        v = self.list
12        for i in range(len(v)):
13            y = v[i].__hash__()
14                if y == -1: return -1
15                x = make32((x^y)*m)
16                m = make32(m + 82520 + 2*((len(v)-i-1)))
17        x = make32(x+97531)
18        if x == -1:
19            x = -2
20        return x
21     def __eq__(self, other):
22         return self.list.__eq__(other.list)
```
>>> from super_list import SuperList
>>> k1 = SuperList([1, 2, 3])
k1 = SuperList([1, 2, 3])
>>> k2 = SuperList([1, 2, 3])
k2 = SuperList([1, 2, 3])
>>> k3 = SuperList([4, 5, 6])
k3 = SuperList([4, 5, 6])

>>> k1 == k2
True

>>> k1 == k3
False

>>> d = {}
d = {}
>>> d[k1] = 'a'
d[k1] = 'a'
>>> d[k2] = 'b'
d[k2] = 'b'
>>> d[k3] = 'c'
d[k3] = 'c'

>>> print d
<super_list.SuperList object at 0x69870>: 'c',
<super_list.SuperList object at 0x69930>: 'b'}

>>> print d[k1], d[k2], d[k3]
b b c
```
>>> k1.list.append(4)
>>> k1 == k2
False
>>> k1 == k3
False
>>> hash(k1)
89902565
>>> hash(k3)
448334556
>>> d[k1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: <super_list.SuperList object at 0x69930>
>>> d[k2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: <super_list.SuperList object at 0x698b0>
>>> d[k3]
'c'
```
What have we learned?

- Dictionary keys must be immutable
Hashing Basic Data

- Examine Python’s hashing functions for the built-in data types
- Examples of reasonable hash functions, avoiding common pitfalls
- Know your language
  - Especially its cost model
PyHash: the Plan

```python
def hash(v):
    ""
    A Python implementation that is identical
to the underlying builtin Python function 'hash'
for integers, longs, strings, instances, and tuples thereof.
This returns -1 only when the object is unhashable.
(Floats not yet implemented.)
    ""
    if type(v) == type(1): return int_hash(v)
    if type(v) == type(1L): return long_hash(v)
    if type(v) == type(" "): return string_hash(v)
    if type(v) == type((1,)): return tuple_hash(v)
    x = dummy
    if type(v) == type(x): return id(v)
    return -1
```
PyHash: Short Integers

```python
1  def make32(x):
2      """
3      Convert x into a 32-bit signed integer.
4      """
5      x = x % (2**32)
6      if x >= 2**31:
7          x = x - 2**32
8      x = int(x)
9      return x
10
11  def int_hash(v):
12      if v == -1: v = -2
13      return v
```
def string_hash(v):
    if v == "":
        return 0
    else:
        x = ord(v[0])<<7
        m = 1000003
        for c in v:
            x = make32((x*m)^ord(c))
            x ^= len(v)
        if x == -1:
            x = -2
        return x
PyHash: Tuples

```python
def tuple_hash(v):
    
    """
    The addend 82520, was selected from the range(0, 1000000) for generating the greatest number of prime multipliers for tuples upto length eight:
    1082527, 1165049, 1082531, 1165057, 1247581, 1330103, 1082533, 1330111, 1412633, 1165069, 1247599, 1495177, 1577699
    """

    m = 1000003
    x = 0x345678
    for i in range(len(v)):
        y = v[i].__hash__()  # Invoke built-in python hash
        if y == -1: return -1
        x = make32((x^y)*m)
        m = make32(m + 82520 + 2*((len(v)-i-1)))
    x = make32(x+97531)
    if x == -1:
        x = -2
    return x
```
PyHash: Long Integers

def long_hash(v):
    sign = 1
    if v<0:
        v, sign = abs(v), -1
    SHIFT = 15  # for a 32-bit machine
    LONG_BIT_SHIFT = 32 - SHIFT
    BASE = 1 << SHIFT
    MASK = (BASE - 1)
    digits = []
    while v>0:
        digits.append(v % BASE)
        v = v>>SHIFT
    digits.reverse()  # process digits high-order to low-order
    x = 0
    for digit in digits:
        x = (((x << SHIFT) & ~MASK) | ((x >> LONG_BIT_SHIFT) & MASK))
        x += digit
    x = make32(x)
    x = x * sign
    if x == -1:
        x = -2
    return x