PDSP 2026, Lecture 06, 27 August 2026

Dictionaries

- A list is a collection indexed by position
- A list can be thought of as a function
  $f : \{0, 1, \ldots, n-1\} \rightarrow \{v_0, v_1, \ldots, v_{n-1}\}$
  - A list maps positions to values
- Generalize this to a function $f : \{k_0, k_1, \ldots, k_{n-1}\} \rightarrow \{v_0, v_1, \ldots, v_{n-1}\}$
  - Instead of positions, index by an abstract *key*
- **dictionary**: maps keys, rather than positions, to values
- Notation:
  - `d = {k1:v1, k2:v2}`, enumerate a dictionary explicitly
  - `d[k1]`, value in dictionary `d1` corresponding to key `k1`
  - `{}`, empty dictionary (`[]` for lists, `()` for tuples)

In [1]:
```python
myd = {'a':1,'b':17,'c':0}
```

In [2]:
```python
myd['b']
```

Out[2]:
```
17
```

In [3]:
```python
myd['d']  # Invalid key
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
Cell In[3], line 1
----> 1 myd['d']  # Invalid key

KeyError: 'd'
```

In [4]:
```python
myd['d'] = 17
```

In [5]:
```python
myd['d']
```

Out[5]:
```
17
```

- An assignment `d[k] = v` serves two purposes
  - If there is no key `k`, create the key and assign it the value `v`
  - If there is already a key `k`, replace its current value by `v`
- In a list, we cannot create a value at a new position through an assignment
  - If `l` is `[0,1,2,3]`, `l[4] = 4` generate `IndexError`
  - If `myd = {'a':1,'b':17,'c':0}`, `d['d'] = 19` extends `d` with a new key-value pair

Iteration

- `d.keys()` generates a sequence of all keys in `d`
  - Iterate over keys using `for k in d.keys():`
  - `for k in d:` also works --- `d` is implicitly intepreted as `d.keys()`
  - Though the keys do not form a sequence, Python will generate them in the order in which they were created
- Similarly, `d.values()` is the sequence of values present

In [6]:
```
myd = {'a':1,'b':17,'c':0}
```

In [7]:
```
list( myd.keys()), list(myd.values())
```

Out[7]:
```
(['a', 'b', 'c'], [1, 17, 0])
```

In [8]:
```
myd = {'b':17,'c':0,'a':1}
```

In [9]:
```
list(myd.keys()), list(myd.values())
```

Out[9]:
```
(['b', 'c', 'a'], [17, 0, 1])
```

In [10]:
```
for k in myd:
    print(k)
```

```
b
c
a
```

In [11]:
```
for v in myd.values():
    print(v)
```

```
17
0
1
```

Example

- Count frequency of numbers in a list
- Maintain a counter for each value that appears in the list
- Dictionary `freqd` where each key is a number `v` and `freqd[v]` is a positive integer
  - The first time we see a number, need to create a key and assign it the value 1
  - If there is already a key for the current number, increment its count
  - Test if a key `k` is present using `k in d.keys()` (or, shorter, `k in d`)

In [12]:
```
def frequency(l):
    freqd = {}
    for v in l:
        if v in freqd:  # Check if v is already a key
            freqd[v] = freqd[v] + 1
        else:           # Create a new key with count 1
            freqd[v] = 1
    return(freqd)
```

In [13]:
```
d = frequency([1,2,1,3,1,4,2,3,1,5,6,2,7])
```

In [14]:
```
d
```

Out[14]:
```
{1: 4, 2: 3, 3: 2, 4: 1, 5: 1, 6: 1, 7: 1}
```

- Keys are listed in the order they are inserted
- This is guaranteed by current versions of Python, need not hold for dictionaries in general

In [15]:
```
d2 = frequency([1,2,1,3,1,4,2,3,1,5,7,2,6])
```

In [16]:
```
d2
```

Out[16]:
```
{1: 4, 2: 3, 3: 2, 4: 1, 5: 1, 7: 1, 6: 1}
```

List membership

- `v in l` returns `True` iff value `v` is in `l`
- Implicit iteration, same as

```
def element(l,v):
  for x in l:
    if x == v:
      return(True)
  return(False)
```

- Linear scan of the list, examine all elements (worst case) if `v` is not in `l`

Extract unique elements from a list

- Standard loop builds a new list of unique elements
- Check if each element in the original list is already in the new list before adding

In [17]:
```
def uniql(l):
    uniqlist = []
    for x in l:
        if not (x in uniqlist):  # Implicit nested loop
            uniqlist.append(x)
    return(uniqlist)
```

Complexity

- Worst case is when original list has no duplicates
- `l[k]` will be compared to `k` elements in `newl` before being added to `newl`
- Takes $1 + 2 + \cdots n - 1$ steps, which is $\dfrac{n(n-1)}{2}$
  - Proportional to n^2

Using a dictionary

- Cannot have duplicate keys in a dictionary
- Create a dictionary whose keys are values in the original list
  - Value associated with key is not important
  - If we see the same value twice, the key will be updated, not duplicated
- In the end, return the list of keys

In [18]:
```
def uniqd(l):
    uniqdict = {}
    for x in l:
        uniqdict[x] = 1
    return(list(uniqdict))
```

Complexity

- Creating/updating a key in a dictionary takes a fixed amount of time, independent of the size of the dictionary
  - Assuming the hash function works well and there are no (or very few) collisions
- This works effectively in time proportional to n, the length of the list

- We can experimentally verify this by applying both functions to a large list without duplicates
  - In the examples below, we have asked for the length of the list rather than the list itself to avoid large outputs cluttering the page

In [19]:
```
len(uniql(list(range(50000)))) # Takes a long time
```

Out[19]:
```
50000
```

In [20]:
```
len(uniqd(list(range(100000))))  # Almost instantaneous
```

Out[20]:
```
100000
```

In [21]:
```
len(uniqd(list(range(10000000)))) # Python can do about 10^7 ops/sec
```

Out[21]:
```
10000000
```

- "Classical" solution is to sort the list
- In the sorted list, duplicates are bunched together
- Scan the sorted list and retain an element if it is different from the previous one
- Sorting takes time $n \log n$ -- we will see this later
- Scanning for duplicates takes time n
- Overall $n \log n$

In [22]:
```
def uniqs(l):
    if l == []:
        return([])
    lsorted = sorted(l)
    uniqlist,previous = [lsorted[0]],lsorted[0]
```

```
    for x in lsorted[1:]:
        if x != previous:
            uniqlist.append(x)
        previous = x
    return(uniqlist)
```

In [23]:
```
len(uniqs(list(range(50000))))
```

Out[23]: 50000

In [24]:
```
len(uniqs(list(range(10000000))))
```

Out[24]: 10000000

- Only marginally slower than `uniqd`

Intersection of two lists

- For each element `v` of `l1` check if `v` occurs in `l2`
- Nested loop

In [25]:
```
def intersect(l1,l2):
    commonlist = []
    for x in l1:
        for y in l2:
            if x == y:
                if not(x in commonlist):
                    commonlist.append(x)
    return(commonlist)
```

In [26]:
```
len(intersect(list(range(0,100)),list(range(50,150))))
```

Out[26]: 50

In [27]:
```
intersect(list(range(0,100)),list(range(50,150)))
```

Out[27]:
```
[50,
 51,
 52,
 53,
 54,
 55,
 56,
 57,
 58,
 59,
 60,
 61,
 62,
 63,
 64,
 65,
 66,
 67,
 68,
 69,
 70,
 71,
 72,
 73,
 74,
 75,
 76,
 77,
 78,
 79,
 80,
 81,
 82,
 83,
 84,
 85,
 86,
 87,
 88,
 89,
 90,
 91,
 92,
 93,
 94,
 95,
 96,
 97,
 98,
 99]
```

In [28]:
```
len(intersect(list(range(0,30000)),list(range(15000,45000))))
```

Out[28]:
```
15000
```

- While we scan `l1` we can check if the element is in `l2`
- `x in l2` is an implicit nested iteration
- Performance is same as the explicit nested loop above

In [29]:
```
def intersect2(l1,l2):
    commonlist = []
    for x in l1:
        if x in l2: # Hidden nested loop
            if not(x in commonlist):
                commonlist.append(x)
    return(commonlist)
```

In [30]:
```
len(intersect2(list(range(0,100)),list(range(50,150))))
```

Out[30]:
```
50
```

In [31]:
```
len(intersect2(list(range(0,30000)),list(range(15000,45000))))
```

Out[31]:
```
15000
```

- Using dictionaries
  - Create a dictionary whose keys are the elements in `l1`
  - Check each element in `l2` against the keys of this dictionary
- Checking `x in l` takes time proportional to `len(l)`
- Checking `y in d.keys()` takes constant time
- Overall time is proportional to `len(l1) + len(l2)`

In [32]:
```
def intersectd(l1,l2):
    commondict = {}
    l1dict = {}
    for x in l1:
        l1dict[x] = 1
    for y in l2:
        if y in l1dict:
            commondict[y] = 1
    return(list(commondict.keys()))
```

In [33]:
```
len(intersectd(list(range(0,300000)),list(range(150000,450000))))
```

Out[33]:
```
150000
```

- "Classical" solution is to sort and merge
  - Sort both lists
  - Scan the two sorted lists in a single pass and identify common elements
- Sorting takes time $n \log n$
- Merging takes time n
- Overall $n \log n$

In [34]:
```
def intersectm(l1,l2):
    l1sort = sorted(l1)
    l2sort = sorted(l2)
    commonlist = []
    i,j = 0,0
    while i < len(l1sort) and j < len(l2sort):
        if l1sort[i] < l2sort[j]:
            i += 1
```

```python
        elif l1sort[i] > l2sort[j]:
            j += 1
        elif l1sort[i] == l2sort[j]:
            if (not l1sort[i] in commonlist):
                commonlist.append(l1sort[i])
            i += 1
            j += 1
    return(commonlist)
```

In [35]:
```python
len(intersectm(list(range(0,50000)),list(range(25000,75000))))
```

Out[35]: 25000

In [36]:
```python
len(intersectm(list(range(0,100000)),list(range(50000,150000))))
```

Out[36]: 50000

- Noticeably slower than `intersectd`

In [37]:
```python
len(intersectd(list(range(0,100000)),list(range(50000,150000))))
```

Out[37]: 50000