2. Dictionaries and Sets

Programming and Algorithms II
Degree in Bioinformatics
Fall 2017
Keeping counts. Example

Input contains 9 6 30 -5 9 5 -5 30 4 1 5 9 5 2 4 -5

Output should be ideally the list

[ (-5,3) (1,1) (2,1) (4,2) (5,3) (6,1) (9,3) (30,2) ]

Or perhaps in another order (less good)
A problem: Keeping counts

Problem:

Given:
Input that contains a sequence of integers

Compute:
A list of pairs \((x, c)\) where \(c\) is the number of times that \(x\) appears in the list (only for \(c > 0\))

 Optionally: the list should be sorted by \(x\)
First solution

create an empty list L
for each element x in input
  if x is not in L:
    L.append((x,1))
  else:
    replace (x,c) with (x,c+1) in L
First solution: Cost

N = number of integers in input
D = number of different integers in input
Note len(L) <= D

“if x not in L” cost up to O(D)
even if implemented as L.count(x)

Total cost O(ND)

If D = N (all items different), cost is O(N²)
Second solution: Keeping the list sorted

Finding \( x \) in a list \( L \) is \( O(\text{len}(L)) \)

\( L.\text{count}(x) \) does not do magic

But if the list is sorted, we can do better

**Binary** or **dichotomic** search:

\( O(\log D) \) time, where \( D = \text{len}(L) \)
Binary or dichotomic search

$O(\log D)$ time, where $D = \text{len}(L)$

Only on ordered structures

Only if $O(1)$ Random Access... which lists have

For $D=1,000$, $\log D = 10$

For $D = 1,000,000$ $\log D = 20$
Binary or dichotomic search

In general, we keep two positions $i,j$ such that

$x$ is in $L$ if and only if it is in $L[i:j]$
def binarySearch(L, x):
    i = 0
    j = len(L) - 1
    while i <= j:
        mid = (i+j)//2  # // is integer div
        if L[mid] > x:
            j = mid - 1
        elif L[mid] < x:
            i = mid + 1
        else:
            return mid
    return -1

If \( x \) is in \( L \), returns some position of \( i \) that contains \( x \)
If \( x \) is not in \( L \), returns -1
2 comparisons per iteration
Binary or dichotomic search

```
def binarySearch(L,x):
    i = 0
    j = len(L)-1
    while i < j:
        mid = (i+j)//2
        if L[mid] >= x:
            j = mid
        else:
            i = mid+1
    if i < len(L) and L[i] == x:
        return i
    else:
        return -1
```

If x is in L, returns the first position of L that contains x
If x is not in L, returns -1
1 comparison per iteration
Why $\log_2(n)$

At every iteration, $j-i$ is divided by 2:

- Either $i$ is the same and $j \leq (i+j)/2$
- Or $j$ is the same and $i \geq (i+j)/2$

When $j < i$, we stop

How many times can we divide $\text{len}(L)$ by 2 before we get to 0?

$\text{len}(L) \approx 2^k \iff k \approx \log_2(\text{len}(L))$

example $256 = 2^8 \rightarrow 128 \ 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1 \ 0$
Second solution: Keeping L sorted

Create an empty list L
For each element x in input
    i = binarySearch(L,x)
    if i == -1:
        insert (x,1) in L, in place that keeps L sorted
    else
        x,c = L[i][0],L[i][1]  # get pair (x,c) in pos. i
        L[i] = (x,c+1)
Second solution: Cost

Remember that len(L) <= D

• Looking if x is in L (and where): O(log D)
• Adding it in right place if not in L: O(D)
• Incrementing count if already in L: O(1)
• O(D) cost for every one of D distinct elements

Solution 1: O(ND)
Solution 2: O(N log D) + D*O(D) + (N-D)*O(1) =
            = O(N log D + D^2)

Is still O(N^2) if D=N, but better if D << N
Solution 3

1. read the whole input into a list A
2. sort A
3. go over A sequentially
   for each element A[i]
   if (i==len(L)−1) or (A[i] < A[i+1]) :
     append A[i] to L

1. A=[3,2,1,3,5,3,1,2,1,5,1]
2. A=[1,1,1,1,2,2,3,3,3,5,5]
3. L=[1,2,3,5]
Solution 3

1. read the whole input into a list A
2. sort A
3. go over A sequentially
   for each element A[i]
     if (i==len(L)-1) or (A[i] < A[i+1]) :
       append A[i] to L

1. $O(N)$
2. $O(N \log N)$
3. $O(N)$
Total time $O(N \log N)$
Recap: Costs

Solution 1: $O(ND)$
Solution 2: $O(N \log D + D^2)$
Solution 3: $O(N \log N)$

Solution 3 faster than 2 for $D > (N \log N)^{1/2}$
But uses memory $O(N)$, not $O(D)$.
No good for Big Data

(There are $O(N \log N)$ algorithms for sorting files not in memory)
## The problem

<table>
<thead>
<tr>
<th></th>
<th>Adding $x$</th>
<th>Searching $x$</th>
<th>Getting all elements in order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted list</strong></td>
<td>.append $O(1)$</td>
<td>.count or linear search $O(D)$</td>
<td>.sort $O(D \log D)$</td>
</tr>
<tr>
<td><strong>Sorted list</strong></td>
<td>.insert $O(D)$</td>
<td>binary search $O(\log D)$</td>
<td>Trivial $O(D)$</td>
</tr>
</tbody>
</table>
The problem

<table>
<thead>
<tr>
<th></th>
<th>Adding x</th>
<th>Searching x</th>
<th>Getting all elements in order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list</td>
<td>.append O(1)</td>
<td>.count or linear search O(D)</td>
<td>sort O(D log D)</td>
</tr>
<tr>
<td>Sorted list</td>
<td>.insert O(D)</td>
<td>binary search O(log D)</td>
<td>Trivial O(D)</td>
</tr>
<tr>
<td>Hash table</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(D log D)</td>
</tr>
<tr>
<td>Balanced tree</td>
<td>O(log D)</td>
<td>O(log D)</td>
<td>O(D)</td>
</tr>
</tbody>
</table>

- You will understand how they are built next semester (“Data structures and algorithms”)
- Detail: Hashing is O(1) “on average”
- Python dictionaries use hash tables.
## Python dictionary operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy, d1 = d</td>
<td>O(n)</td>
</tr>
<tr>
<td>Get, x = d[key]</td>
<td>O(1)</td>
</tr>
<tr>
<td>or d.get(key, def)</td>
<td></td>
</tr>
<tr>
<td>Set, d[key] = v</td>
<td>O(1)</td>
</tr>
<tr>
<td>key (not) in d</td>
<td>O(1)</td>
</tr>
<tr>
<td>del d[key]</td>
<td>O(1)</td>
</tr>
<tr>
<td>len(d)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Technically, *average* time. If you have really, really bad luck, all is O(n)
Python dictionary iterators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.items()</td>
<td>[(k1,v1),(k2,v2),..., (kn,vn)]</td>
</tr>
<tr>
<td></td>
<td>IN NO PARTICULAR ORDER</td>
</tr>
<tr>
<td>d.iteritems()</td>
<td>Iterator over the list above</td>
</tr>
<tr>
<td>d.keys()</td>
<td>Iterator over the set of keys of d</td>
</tr>
<tr>
<td>d.itervalues()</td>
<td>Iterator over the set of values of d</td>
</tr>
</tbody>
</table>

Note: In Python2 this really returned a list. In Python 3, it returns an “observer object”, a list that will change if the dictionary changes. Just put the result inside list(...) to get a proper list.
Keeping counts, Solution 4

d = {}
for each x in input
    if x in d:
        c = d[x]
    else:
        c = 0
    d[x] = c + 1

To sort at the end:

L = []
for x in sorted(d.keys()):
    L.append((x,d[x]))
Solution 4, cost

- Checking if x in dictionary $O(1)$
- Getting count from dictionary $O(1)$
- Adding / updating dictionary $O(1)$
- N iterations, each cost $O(1)$
- Cost $O(N)$
- Independent of D!!

- Sorting at the end, $O(D \log D)$
- Cost $O(N + D \log D)$
Dictionaries vs. tables

• Dictionaries preferable if only operations are “find” and “insert/update”

• List have (small) ranges of ints as keys
• Dictionaries take any hashable type as key

• If a list suffices, don’t use a dictionary:
  Same O() but bigger constant in time and memory
Sets

• Special case of Dictionaries, if you want

• No value associated to key

• Just is / is not
<table>
<thead>
<tr>
<th>Operation</th>
<th>Sets</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy, s1 = s</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>s.isdisjoint(s1), s.issubset(s1), s.issuperset(s1)</td>
<td>O(n1+n2)</td>
<td></td>
</tr>
<tr>
<td>s.union(s1), s.intersection(s1), s.difference(s1)</td>
<td>O(n1+n2)</td>
<td></td>
</tr>
<tr>
<td>x (not) in s</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>s.add(x)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>s.remove(x) (exception if not in)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>s.discard(x) (no exception)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>len(s)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>for x in s ...</td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>
To remember

• Dictionaries: Functions keys $\rightarrow$ values

• Sets: add, remove, is / is not

• **Huge speedups** over lists in many problems

• **They are your friends**

• Central to Python