2. Dictionaries and Sets

Programming and Algorithms II

Degree in Bioinformatics

Fall 2018

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 $O(N^2)$

Second solution: Keeping the list sorted

Finding x in a list L is O(len(L))

L.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 = len(L)

Binary or dichotomic search

O(log D) time, where D = len(L)

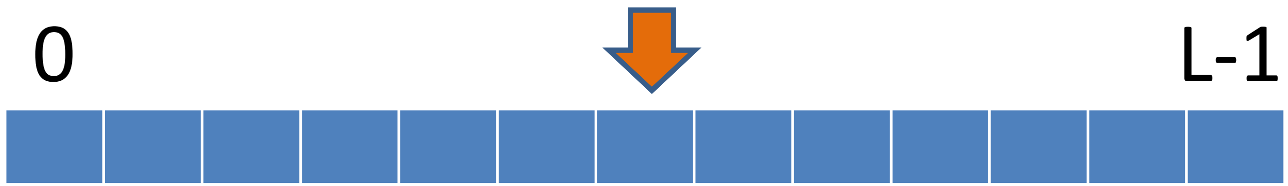
Only on ordered structures

Only if O(1) Random Access… which lists have

For D=1,000,   log D = 10

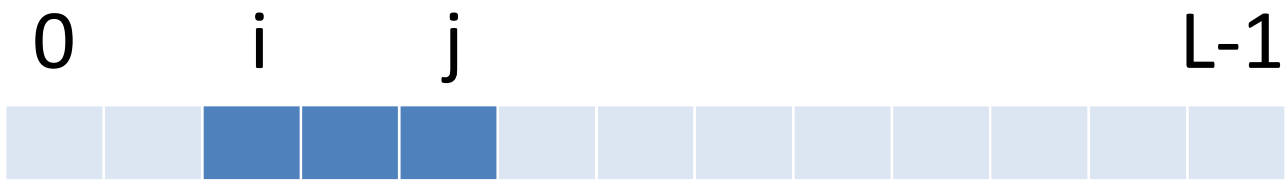
For D = 1,000,000   log D = 20

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]



Binary or dichotomic search

```
def binary_search(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 binary_search(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 <= (i+j)/2
- Or j is the same and i >= (i+j)/2

When j < i, we stop

How many times can we divide len(L) by 2 before we get to 0?

len(L) $\approx 2^k$ ←→ $k \approx \log_2$ (len(L))

example 256 = 2^8 → 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 large w.r.t. N

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

	Adding x	Searching x	Getting all elements in order
Unsorted list	.append O(1)	.count or linear search O(D)	.sort O(D log D)
Sorted list	.insert O(D)	binary seach O(log D)	Trivial O(D)

The problem

	Adding x	Searching x	Getting all elements in order
Unsorted list	.append O(1)	.count or linear search O(D)	sort O(D log D)
Sorted list	.insert O(D)	binary seach O(log D)	Trivial O(D)
Hash table	O(1)	O(1)	O(D log D)
Balanced tree	O(log D)	O(log D)	O(D)

- You will understand how they are built next term
- Detail: Hashing is O(1) “on average”
- Python dictionaries use hash tables

Python dictionary operations

Operation	Time
d1 = d.copy()	O(n)
Get, x = d[key] or d.get(key,def)	O(1)
Set, d[key] = v	O(1)
key (not) in d	O(1)
del d[key]	O(1)
len(d)	O(1)

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

Python dictionary iterators

Operation	Returns
d.items()	[(k1,v1),(k2,v2),...,(kn,vn)] IN NO PARTICULAR ORDER
d.iteritems()	Iterator over the list above
d.keys()	Iterator over the set of keys of d
d.itervalues()	Iterator over the set of values of d

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 <u>hashable</u> 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 in" / "is not in"

Sets

Operation	Time
s1 = s.copy()	O(n)
s.isdisjoint(s1) ,s.issubset(s1), s.issuperset(s1)	O(n1+n2)
s.union(s1), s.intersection(s1), s.difference(s1)	O(n1+n2)
x (not) in s	O(1)
s.add(x)	O(1)
s.remove(x) (exception if not in) s.discard(x) (no exception)	O(1)
len(s)	O(1)
for x in s …	O(n)

To remember

- Dictionaries: Functions keys → values
- Sets: add, remove, is / is not
- Huge speedups over lists in many problems
- They are your friends
- Central to Python