

# Daily Coding Problem #1

## Problem

This problem was recently asked by Google.

Given a list of numbers and a number k, return whether any two numbers from the list add up to k.

For example, given [10, 15, 3, 7] and k of 17, return true since 10 + 7 is 17.

Bonus: Can you do this in one pass?

## Solution

This problem can be solved in several different ways.

Brute force way would involve a nested iteration to check for every pair of numbers:

```
def two_sum(lst, k):
    for i in range(len(lst)):
        for j in range(len(lst)):
            if i != j and lst[i] + lst[j] == k:
                return True
    return False
```

This would take  $O(N^2)$ . Another way is to use a set to remember the numbers we've seen so far. Then for a given number, we can check if there is another number that, if added, would sum to k. This would be  $O(N)$  since lookups of sets are  $O(1)$  each.

```
def two_sum(lst, k):
    seen = set()
    for num in lst:
        if k - num in seen:
            return True
        seen.add(num)
```

```
return False
```

Yet another solution involves sorting the list. We can then iterate through the list and run a binary search on  $K - \text{lst}[i]$ . Since we run binary search on  $N$  elements, this would take  $O(N \log N)$  with  $O(1)$  space.

```
from bisect import bisect_left

def two_sum(lst, K):
    lst.sort()

    for i in range(len(lst)):
        target = K - lst[i]
        j = binary_search(lst, target)

        # Check that binary search found the target and that it's not in the same index
        # as i. If it is in the same index, we can check lst[i + 1] and lst[i - 1] to see
        # if there's another number that's the same value as lst[i].
        if j == -1:
            continue
        elif j != i:
            return True
        elif j + 1 < len(lst) and lst[j + 1] == target:
            return True
        elif j - 1 >= 0 and lst[j - 1] == target:
            return True
    return False

def binary_search(lst, target):
    lo = 0
    hi = len(lst)
    ind = bisect_left(lst, target, lo, hi)

    if 0 <= ind < hi and lst[ind] == target:
        return ind
    return -1
```

# Daily Coding Problem #2

## Problem

This problem was asked by Uber.

Given an array of integers, return a new array such that each element at index  $i$  of the new array is the product of all the numbers in the original array except the one at  $i$ .

For example, if our input was [1, 2, 3, 4, 5], the expected output would be [120, 60, 40, 30, 24]. If our input was [3, 2, 1], the expected output would be [2, 3, 6].

Follow-up: what if you can't use division?

## Solution

This problem would be easy with division: an optimal solution could just find the product of all numbers in the array and then divide by each of the numbers.

Without division, another approach would be to first see that the  $i^{\text{th}}$  element simply needs the product of numbers before  $i$  and the product of numbers after  $i$ . Then we could multiply those two numbers to get our desired product.

In order to find the product of numbers before  $i$ , we can generate a list of prefix products. Specifically, the  $i^{\text{th}}$  element in the list would be a product of all numbers including  $i$ . Similarly, we would generate the list of suffix products.

```
def products(nums):
    # Generate prefix products
    prefix_products = []
    for num in nums:
        if prefix_products:
            prefix_products.append(prefix_products[-1] * num)
        else:
            prefix_products.append(num)
```

```
# Generate suffix products
suffix_products = []
for num in reversed(nums):
    if suffix_products:
        suffix_products.append(suffix_products[-1] * num)
    else:
        suffix_products.append(num)
suffix_products = list(reversed(suffix_products))

# Generate result
result = []
for i in range(len(nums)):
    if i == 0:
        result.append(suffix_products[i + 1])
    elif i == len(nums) - 1:
        result.append(prefix_products[i - 1])
    else:
        result.append(prefix_products[i - 1] * suffix_products[i + 1])
return result
```

This runs in  $O(N)$  time and space, since iterating over the input arrays takes  $O(N)$  time and creating the prefix and suffix arrays take up  $O(N)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #3

## Problem

This problem was asked by Google.

Given the root to a binary tree, implement `serialize(root)`, which serializes the tree into a string, and `deserialize(s)`, which deserializes the string back into the tree.

For example, given the following Node class

```
class Node:  
    def __init__(self, val, left=None, right=None):  
        self.val = val  
        self.left = left  
        self.right = right
```

The following test should pass:

```
node = Node('root', Node('left', Node('left.left')), Node('right'))  
assert deserialize(serialize(node)).left.left.val == 'left.left'
```

## Solution

There are many ways to serialize and deserialize a binary tree, so don't worry if your solution differs from this one. We will be only going through one possible solution.

We can approach this problem by first figuring out what we would like the serialized tree to look like. Ideally, it would contain the minimum information required to encode all the necessary information about the binary tree. One possible encoding might be to borrow [S-expressions](#) from Lisp. The tree `Node(1, Node(2), Node(3))` would then look like '(1 (2 () ()) (3 () ()))', where the empty brackets denote nulls.

To minimize data over the hypothetical wire, we could go a step further and prune out some unnecessary brackets. We could also replace the 2-character '()' with '#'. We can then infer leaf nodes by their form 'val # #' and thus get the structure of the tree that way. Then our tree would

look like 1 2 # # 3 # #.

```
def serialize(root):
    if root is None:
        return '#'
    return '{} {} {}'.format(root.val, serialize(root.left), serialize(root.right))

def deserialize(data):
    def helper():
        val = next(vals)
        if val == '#':
            return None
        node = Node(int(val))
        node.left = helper()
        node.right = helper()

        return node
    vals = iter(data.split())
    return helper()
```

This runs in  $O(N)$  time and space, since we iterate over the whole tree when serializing and deserializing.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #4

## Problem

This problem was asked by Stripe.

Given an array of integers, find the first missing positive integer in linear time and constant space. In other words, find the lowest positive integer that does not exist in the array. The array can contain duplicates and negative numbers as well.

For example, the input [3, 4, -1, 1] should give 2. The input [1, 2, 0] should give 3.

You can modify the input array in-place.

## Solution

Our lives would be easier without the linear time constraint: we would just sort the array, while filtering out negative numbers, and iterate over the sorted array and return the first number that doesn't match the index. However, sorting takes  $O(n \log n)$ , so we can't use that here.

Clearly we have to use some sort of trick here to get it running in linear time. Since the first missing positive number must be between 1 and  $\text{len}(\text{array}) + 1$  (why?), we can ignore any negative numbers and numbers bigger than  $\text{len}(\text{array})$ . The basic idea is to use the indices of the array itself to reorder the elements to where they should be. We traverse the array and swap elements between 0 and the length of the array to their value's index. We stay at each index until we find that index's value and keep on swapping.

By the end of this process, all the first positive numbers should be grouped in order at the beginning of the array. We don't care about the others. This only takes  $O(N)$  time, since we swap each element at most once.

Then we can iterate through the array and return the index of the first number that doesn't match, just like before.

```
def first_missing_positive(nums):
    if not nums:
        return 1
    for i, num in enumerate(nums):
```

```
while i + 1 != nums[i] and 0 < nums[i] <= len(nums):
    v = nums[i]
    nums[i], nums[v - 1] = nums[v - 1], nums[i]
    nums[v - 1] = v
    if nums[i] == nums[v - 1]:
        break
for i, num in enumerate(nums, 1):
    if num != i:
        return i
return len(nums) + 1
```

Another way we can do this is by adding all the numbers to a set, and then use a counter initialized to 1. Then continuously increment the counter and check whether the value is in the set.

```
def first_missing_positive(nums):
    s = set(nums)
    i = 1
    while i in s:
        i += 1
    return i
```

This is much simpler, but runs in  $O(N)$  time and space, whereas the previous algorithm uses no extra space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #5

## Problem

This problem was asked by Jane Street.

`cons(a, b)` constructs a pair, and `car(pair)` and `cdr(pair)` returns the first and last element of that pair. For example, `car(cons(3, 4))` returns 3, and `cdr(cons(3, 4))` returns 4.

Given this implementation of `cons`:

```
def cons(a, b):
    def pair(f):
        return f(a, b)
    return pair
```

Implement `car` and `cdr`.

## Solution

This is a really cool example of using [closures](#) to store data. We must look at the signature type of `cons` to retrieve its first and last elements. `cons` takes in `a` and `b`, and returns a new anonymous function, which itself takes in `f`, and calls `f` with `a` and `b`. So the input to `car` and `cdr` is that anonymous function, which is `pair`. To get `a` and `b` back, we must feed it yet another function, one that takes in two parameters and returns the first (if `car`) or last (if `cdr`) one.

```
def car(pair):
    return pair(lambda a, b: a)

def cdr(pair):
    return pair(lambda a, b: b)
```

Fun fact: `cdr` is pronounced "cudder"!

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #6

## Problem

This problem was asked by Google.

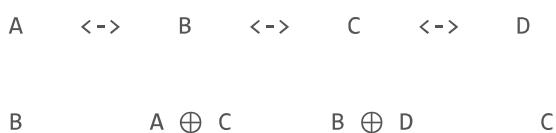
An XOR linked list is a more memory efficient doubly linked list. Instead of each node holding next and prev fields, it holds a field named both, which is an XOR of the next node and the previous node. Implement an XOR linked list; it has an add(element) which adds the element to the end, and a get(index) which returns the node at index.

If using a language that has no pointers (such as Python), you can assume you have access to `get_pointer` and `dereference_pointer` functions that converts between nodes and memory addresses.

## Solution

For the head, both will just be the address of next, and if it's the tail, it should just be the address of prev. And intermediate nodes should have an XOR of next and prev.

Here's an example XOR linked list which meets the above conditions:



Let's work through get first, assuming that the above conditions are maintained. Then, given a node, to go to the next node, we have to XOR the current node's both with the previous node's address. And to handle getting the next node from the head, we would initialize the previous node's address as 0.

So in the above example, A's both is B which when XOR'd with 0 would become B. Then B's both is A ⊕ C, which when XOR'd with A becomes C, etc.

To implement add, we would need to update current tail's both to be XOR'd by its current both the new node's memory address. Then the new node's both would just point to the memory address of the current tail. Finally, we'd update the current tail to be equal to the new node.

```
import ctypes

# This is hacky. It's a data structure for C, not python.

class Node(object):
    def __init__(self, val):
        self.val = val
        self.both = 0

class XorLinkedList(object):
    def __init__(self):
        self.head = self.tail = None
        self.__nodes = [] # This is to prevent garbage collection

    def add(self, node):
        if self.head is None:
            self.head = self.tail = node
        else:
            self.tail.both = id(node) ^ self.tail.both
            node.both = id(self.tail)
            self.tail = node

        # Without this line, Python thinks there is no way to reach nodes between
        # head and tail.
        self.__nodes.append(node)

    def get(self, index):
        prev_id = 0
        node = self.head
        for i in range(index):
            next_id = prev_id ^ node.both

            if next_id:
                prev_id = id(node)
                node = _get_obj(next_id)
            else:
                raise IndexError('Linked list index out of range')

        return node
```

```
    return node

def _get_obj(id):
    return ctypes.cast(id, ctypes.py_object).value
```

add runs in O(1) time and get runs in O(N) time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #7

## Problem

This problem was asked by Facebook.

Given the mapping a = 1, b = 2, ... z = 26, and an encoded message, count the number of ways it can be decoded.

For example, the message '111' would give 3, since it could be decoded as 'aaa', 'ka', and 'ak'.

You can assume that the messages are decodable. For example, '001' is not allowed.

## Solution

This looks like a problem that is ripe for solving with recursion. First, let's try to think of a recurrence we can use for this problem. We can try some cases:

- "", the empty string and our base case, should return 1.
- "1" should return 1, since we can parse it as "a" + "".
- "11" should return 2, since we can parse it as "a" + "a" + "" and "k" + "".
- "111" should return 3, since we can parse it as:
  - "a" + "k" + ""
  - "k" + "a" + ""
  - "a" + "a" + "a" + "".
- "011" should return 0, since no letter starts with 0 in our mapping.
- "602" should also return 0 for similar reasons.

We have a good starting point. We can see that the recursive structure is as follows:

- If string starts with zero, then there's no valid encoding.
- If the string's length is less than or equal to 1, there is only 1 encoding.
- If the first two digits form a number k that is less than or equal to 26, we can recursively count the number of encodings assuming we pick k as a letter.
- We can also pick the first digit as a letter and count the number of encodings with this assumption.

```
def num_encodings(s):
    if s.startswith('0'):
        return 0
    elif len(s) <= 1: # This covers empty string
        return 1

    total = 0

    if int(s[:2]) <= 26:
        total += num_encodings(s[2:])

    total += num_encodings(s[1:])
    return total
```

However, this solution is not very efficient. Every branch calls itself recursively twice, so our runtime is  $O(2^n)$ . We can do better by using dynamic programming.

All the following code does is repeat the same computation as above except starting from the base case and building up the solution. Since each iteration takes  $O(1)$ , the whole algorithm now takes  $O(n)$ .

```
from collections import defaultdict

def num_encodings(s):
    # On lookup, this hashmap returns a default value of 0 if the key doesn't exist
    # cache[i] gives us # of ways to encode the substring s[i:]
    cache = defaultdict(int)
    cache[len(s)] = 1 # Empty string is 1 valid encoding

    for i in reversed(range(len(s))):
        if s[i].startswith('0'):
            cache[i] = 0
        elif i == len(s) - 1:
            cache[i] = 1
        else:
            cache[i] = cache[i+1]
            if int(s[i:i+2]) <= 26:
                cache[i] += cache[i+2]
```

```
        else:  
            if int(s[i:i + 2]) <= 26:  
                cache[i] = cache[i + 2]  
                cache[i] += cache[i + 1]  
  
    return cache[0]
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #8

## Problem

This problem was asked by Google.

A unival tree (which stands for "universal value") is a tree where all nodes under it have the same value.

Given the root to a binary tree, count the number of unival subtrees.

For example, the following tree has 5 unival subtrees:

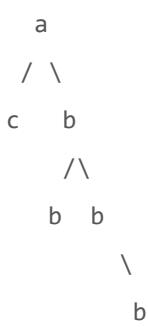
```
0
/
1  0
 / \
1   0
 / \
1   1
```

## Solution

To start off, we should go through some examples.

```
a
/
a  a
 /\
a  a
 \
A
```

This tree has 3 unival subtrees: the two 'a' leaves, and the one 'A' leaf. The 'A' leaf causes all its parents to not be counted as a unival tree.



This tree has 5 unival subtrees: the leaf at 'c', and every 'b'.

We can start off by first writing a function that checks whether a tree is unival or not. Then, perhaps we could use this to count up all the nodes in the tree.

To check whether a tree is a unival tree, we must check that every node in the tree has the same value. To start off, we could define an `is_unival` function that takes in a root to a tree. We would do this recursively with a helper function. Recall that a leaf qualifies as a unival tree.

```

def is_unival(root):
    return unival_helper(root, root.value)

def unival_helper(root, value):
    if root is None:
        return True
    if root.value == value:
        return unival_helper(root.left, value) and unival_helper(root.right, value)
    return False

```

And then our function that counts the number of subtrees could simply use that function:

```

def count_unival_subtrees(root):
    if root is None:
        return 0
    left = count_unival_subtrees(root.left)
    right = count_unival_subtrees(root.right)
    return 1 + left + right if is_unival(root) else left + right

```

However, this runs in  $O(n^2)$  time. For each node of the tree, we're evaluating each node in its subtree again as well. We can improve the runtime by starting at the leaves of the tree, and keeping track of the unival subtree count and value as we percolate back up. This should evaluate each node only once, making it run in  $O(n)$  time.

```

def count_unival_subtrees(root):
    count, _ = helper(root)
    return count

# Also returns number of unival subtrees, and whether it is itself a unival subtree.

```

```
def helper(root):
    if root is None:
        return 0, True

    left_count, is_left_unival = helper(root.left)
    right_count, is_right_unival = helper(root.right)
    total_count = left_count + right_count

    if is_left_unival and is_right_unival:
        if root.left is not None and root.value != root.left.value:
            return total_count, False
        if root.right is not None and root.value != root.right.value:
            return total_count, False
        return total_count + 1, True
    return total_count, False
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #9

## Problem

This problem was asked by Airbnb.

Given a list of integers, write a function that returns the largest sum of non-adjacent numbers. Numbers can be 0 or negative.

For example, [2, 4, 6, 2, 5] should return 13, since we pick 2, 6, and 5. [5, 1, 1, 5] should return 10, since we pick 5 and 5.

Follow-up: Can you do this in O(N) time and constant space?

## Solution

This problem seems easy from the surface, but is actually quite tricky. It's tempting to try to use a greedy strategy like pick the largest number (or first), then the 2nd-largest if it's non-adjacent and so on, but these don't work -- there will always be some edge case that breaks it.

Instead, we should look at this problem recursively. Say we had a function that already returns the largest sum of non-adjacent integers on smaller inputs. How could we use it to figure out what we want?

Say we used this function on our array from `a[1:]` and `a[2:]`. Then our solution should be `a[1:]` OR `a[0] + a[2:]`, whichever is largest. This is because choosing `a[1:]` precludes us from picking `a[0]`. So, we could write a straightforward recursive solution like this:

```
def largest_non_adjacent(arr):
    if not arr:
        return 0

    return max(
        largest_non_adjacent(arr[1:]),
        arr[0] + largest_non_adjacent(arr[2:]))
```

However, this solution runs in  $O(2^n)$  time, since with each call, we're making two further recursive calls. We could memoize the results, or use dynamic programming to store, in an array, the largest sum of non-adjacent numbers from index 0 up to that point. Like so:

```
def largest_non_adjacent(arr):
    if len(arr) <= 2:
        return max(0, max(arr))

    cache = [0 for i in arr]
    cache[0] = max(0, arr[0])
    cache[1] = max(cache[0], arr[1])

    for i in range(2, len(arr)):
        num = arr[i]
        cache[i] = max(num + cache[i - 2], cache[i - 1])

    return cache[-1]
```

This code should run in  $O(n)$  and in  $O(n)$  space. But we can improve this even further. Notice that we only ever use the last two elements of the cache when iterating through the array. This suggests that we could just get rid of most of the array and just store them as variables:

```
def largest_non_adjacent(arr):
    if len(arr) <= 2:
        return max(0, max(arr))

    max_excluding_last = max(0, arr[0])
    max_including_last = max(max_excluding_last, arr[1])

    for num in arr[2:]:
        prev_max_including_last = max_including_last

        max_including_last = max(max_including_last, max_excluding_last + num)
        max_excluding_last = prev_max_including_last

    return max(max_including_last, max_excluding_last)
```



# Daily Coding Problem #10

## Problem

This problem was asked by Apple.

Implement a job scheduler which takes in a function  $f$  and an integer  $n$ , and calls  $f$  after  $n$  milliseconds.

## Solution

We can implement the job scheduler in many different ways, so don't worry if your solution is different from ours. Here is just one way:

First, let's try the most straightforward solution. That would probably be to spin off a new thread on each function we want to delay, sleep the requested amount, and then run the function. It might look something like this:

```
import threading
from time import sleep

class Scheduler:
    def __init__(self):
        pass

    def delay(self, f, n):
        def sleep_then_call(n):
            sleep(n / 1000)
            f()
        t = threading.Thread(target=sleep_then_call)
        t.start()
```

While this works, there is a huge problem with this method: we spin off a new thread each time we call delay! That means the number of threads we use could easily explode. We can get around this by having only one dedicated thread to call the functions, and storing the functions we need to call in some data structure. In this case, we use a list. We also have to do some sort of polling now to check when to run a function. We can store each function along with a unix epoch timestamp that tells it when it should run by. Then we'll poll some designated tick amount and check the list for any jobs that are due to be run, run them, and then remove them from the list.

```
from time import sleep
import threading

class Scheduler:
    def __init__(self):
        self.fns = [] # tuple of (fn, time)
        t = threading.Thread(target=self.poll)
        t.start()

    def poll(self):
        while True:
            now = time() * 1000
            for fn, due in self.fns:
                if now > due:
                    fn()
            self.fns = [(fn, due) for (fn, due) in self.fns if due > now]
            sleep(0.01)

    def delay(self, f, n):
        self.fns.append((f, time() * 1000 + n))
```

We'll stop here, but you can go much farther with this. Some extra credit work:

- Extend the scheduler to allow calling delayed functions with variables
- Use a heap instead of a list to keep track of the next job to run more efficiently
- Use a condition variable instead of polling (it just polls lower in the stack)
- Use a threadpool or other mechanism to decrease the chance of starvation (one thread not being able to run because of another running thread)

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #11

## Problem

This problem was asked by Twitter.

Implement an autocomplete system. That is, given a query string  $s$  and a set of all possible query strings, return all strings in the set that have  $s$  as a prefix.

For example, given the query string de and the set of strings [dog, deer, deal], return [deer, deal].

Hint: Try preprocessing the dictionary into a more efficient data structure to speed up queries.

## Solution

The naive solution here is very straightforward: we need to only iterate over the dictionary and check if each word starts with our prefix. If it does, then add it to our set of results and then return it once we're done.

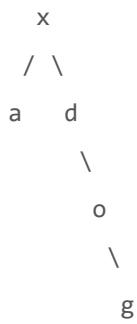
```
WORDS = ['foo', 'bar', ...]  
def autocomplete(s):  
    results = set()  
    for word in WORDS:  
        if word.startswith(s):  
            results.add(word)  
    return results
```

This runs in  $O(N)$  time, where  $N$  is the number of words in the dictionary. Let's think about making this more efficient. We can preprocess the words, but what data structure would be best for our problem?

If we pre-sort the list, we could use binary search to find the first word that includes our prefix and then the last, and return everything in between.

Alternatively, we could use a tree for this. Not a binary tree, but a tree where each child represents

one character of the alphabet. For example, let's say we had the words 'a' and 'dog' in our dictionary. Then the tree would look like this:



Then, to find all words beginning with 'do', we could start at the root, go into the 'd' child, and then the 'o', child, and gather up all the words under there. We would also some sort of terminal value to mark whether or not 'do' is actually a word in our dictionary or not. This data structure is known as a [trie](#).

So the idea is to preprocess the dictionary into this tree, and then when we search for a prefix, go into the trie and get all the words under that prefix node and return those. While the worst-case runtime would still be  $O(n)$  if all the search results have that prefix, if the words are uniformly distributed across the alphabet, it should be much faster on average since we no longer have to evaluate words that don't start with our prefix.

```
ENDS_HERE = '__ENDS_HERE'

class Trie(object):
    def __init__(self):
        self._trie = {}

    def insert(self, text):
        trie = self._trie
        for char in text:
            if char not in trie:
                trie[char] = {}
            trie = trie[char]
        trie[ENDS_HERE] = True

    def elements(self, prefix):
        d = self._trie
        for char in prefix:
            if char in d:
                d = d[char]
            else:
                return []
        return self._elements(d)
```

```
def _elements(self, d):
    result = []
    for c, v in d.items():
        if c == ENDS_HERE:
            subresult = ['']
        else:
            subresult = [c + s for s in self._elements(v)]
        result.extend(subresult)
    return result

trie = Trie()
for word in words:
    trie.insert(word)

def autocomplete(s):
    suffixes = trie.elements(s)
    return [s + w for w in suffixes]
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #12

## Problem

This problem was asked by Amazon.

There exists a staircase with N steps, and you can climb up either 1 or 2 steps at a time. Given N, write a function that returns the number of unique ways you can climb the staircase. The order of the steps matters.

For example, if N is 4, then there are 5 unique ways:

- 1, 1, 1, 1
- 2, 1, 1
- 1, 2, 1
- 1, 1, 2
- 2, 2

What if, instead of being able to climb 1 or 2 steps at a time, you could climb any number from a set of positive integers X? For example, if X = {1, 3, 5}, you could climb 1, 3, or 5 steps at a time.

## Solution

It's always good to start off with some test cases. Let's start with small cases and see if we can find some sort of pattern.

- N = 1: [1]
- N = 2: [1, 1], [2]
- N = 3: [1, 2], [1, 1, 1], [2, 1]
- N = 4: [1, 1, 2], [2, 2], [1, 2, 1], [1, 1, 1, 1], [2, 1, 1]

What's the relationship?

The only ways to get to N = 3, is to first get to N = 1, and then go up by 2 steps, or get to N = 2 and go up by 1 step. So  $f(3) = f(2) + f(1)$ .

Does this hold for  $N = 4$ ? Yes, it does. Since we can only get to the 4th step by getting to the 3rd step and going up by one, or by getting to the 2nd step and going up by two. So  $f(4) = f(3) + f(2)$ .

To generalize,  $f(n) = f(n - 1) + f(n - 2)$ . That's just the [Fibonacci sequence!](#)

```
def staircase(n):
    if n <= 1:
        return 1
    return staircase(n - 1) + staircase(n - 2)
```

Of course, this is really slow ( $O(2^N)$ ) — we are doing a lot of repeated computations! We can do it a lot faster by just computing iteratively:

```
def staircase(n):
    a, b = 1, 2
    for _ in range(n - 1):
        a, b = b, a + b
    return a
```

Now, let's try to generalize what we've learned so that it works if you can take a number of steps from the set  $X$ . Similar reasoning tells us that if  $X = \{1, 3, 5\}$ , then our algorithm should be  $f(n) = f(n - 1) + f(n - 3) + f(n - 5)$ . If  $n < 0$ , then we should return 0 since we can't start from a negative number of steps.

```
def staircase(n, X):
    if n < 0:
        return 0
    elif n == 0:
        return 1
    else:
        return sum(staircase(n - x, X) for x in X)
```

This is again, very slow ( $O(|X|^N)$ ) since we are repeating computations again. We can use dynamic programming to speed it up.

Each entry  $\text{cache}[i]$  will contain the number of ways we can get to step  $i$  with the set  $X$ . Then, we'll build up the array from zero using the same recurrence as before:

```
def staircase(n, X):
    cache = [0 for _ in range(n + 1)]
    cache[0] = 1
    for i in range(1, n + 1):

        cache[i] += sum(cache[i - x] for x in X if i - x >= 0)
    return cache[n]
```

This now takes  $O(N * |X|)$  time and  $O(N)$  space.

---

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #13

## Problem

This problem was asked by Amazon.

Given an integer  $k$  and a string  $s$ , find the length of the longest substring that contains at most  $k$  distinct characters.

For example, given  $s = "abcba"$  and  $k = 2$ , the longest substring with  $k$  distinct characters is "bcb".

## Solution

The most obvious brute force solution here is to simply try every possible substring of the string and check whether it contains at most  $k$  distinct characters. If it does and it is greater than the current longest valid substring, then update the current one. This takes  $O(n^2 * k)$  time, since we use  $n^2$  to generate each possible substring, and then take  $k$  to check each character.

```
def longest_substring_with_k_distinct_characters(s, k):
    current_longest_substring = ''
    for i in range(len(s)):
        for j in range(i + 1, len(s) + 1):
            substring = s[i:j]
            if len(set(substring)) <= k and len(substring) > len(current_longest_substring):
                current_longest_substring = substring
    return len(current_longest_substring)
```

We can improve this by instead keeping a running window of our longest substring. We'll keep a dictionary that maps characters to the index of their last occurrence. Then, as we iterate over the string, we'll check the size of the dictionary. If it's larger than  $k$ , then it means our window is too big, so we have to pop the smallest item in the dictionary and recompute the bounds. If, when we add a character to the dictionary and it doesn't go over  $k$ , then we're safe -- the dictionary hasn't been filled up yet or it's a character we've seen before.

```
def longest_substring_with_k_distinct_characters(s, k):
```

```
if k == 0:
    return 0

# Keep a running window
bounds = (0, 0)
h = {}
max_length = 0
for i, char in enumerate(s):
    h[char] = i
    if len(h) <= k:
        new_lower_bound = bounds[0] # lower bound remains the same
    else:
        # otherwise, pop last occurring char
        key_to_pop = min(h, key=h.get)
        new_lower_bound = h.pop(key_to_pop) + 1

    bounds = (new_lower_bound, bounds[1] + 1)
    max_length = max(max_length, bounds[1] - bounds[0])

return max_length
```

This takes  $O(n * k)$  time and  $O(k)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #14

## Problem

This problem was asked by Google.

The area of a circle is defined as  $\pi r^2$ . Estimate  $\pi$  to 3 decimal places using a Monte Carlo method.

Hint: The basic equation of a circle is  $x^2 + y^2 = r^2$ .

## Solution

Monte Carlo methods rely on random sampling. In this case, if we take a cartesian plane and inscribe a circle with radius  $r$  inside a square with lengths  $2r$ , then the area of the circle will be  $\pi r^2$  while the area of the square will be  $(2r)^2 = 4r^2$ . Then, the ratio of the areas of the circle to the square is  $\pi / 4$ .

So, what we can do is the following:

- Set  $r$  to be 1 (the unit circle)
- Randomly generate points within the square with corners  $(-1, -1)$ ,  $(1, 1)$ ,  $(1, -1)$ ,  $(-1, 1)$
- Keep track of the points that fall inside and outside the circle
  - You can check whether a point  $(x, y)$  is inside the circle if  $x^2 + y^2 < r^2$ , which is another way of representing a circle
- Divide the number of points that fall inside the circle to the total number of points -- that should give us an approximation of  $\pi / 4$ .

```
from random import uniform  
from math import pow
```

```
def generate():
    return (uniform(-1, 1), uniform(-1, 1))

def is_in_circle(coords):
    return coords[0] * coords[0] + coords[1] * coords[1] < 1

def estimate():
    iterations = 10000000
    in_circle = 0
    for _ in range(iterations):
        if is_in_circle(generate()):
            in_circle += 1

    pi_over_four = in_circle / iterations
    return pi_over_four * 4
```

Note that this doesn't give a perfect approximation -- we need more iterations to get a closer estimate. We want the digits of pi up to 3 decimal places. This translates to an error of  $< 10^{-3}$ . The error scales with the square root of the number of guesses, which means we need  $10^6$  iterations to get to our desired precision. If we want more precision, we'll have to crank up the iterations.

This problem is embarrassingly parallel. None of the estimations have any dependent computations, so we can parallelize this problem easily -- divide up the workload into P processes you have, and then add up all the points in the circle in the end. Extra credit: make this program multi-process.



# Daily Coding Problem #15

## Problem

This problem was asked by Facebook.

Given a stream of elements too large to store in memory, pick a random element from the stream with uniform probability.

## Solution

Naively, we could process the stream and store all the elements we encounter in a list, find its size, and pick a random element from  $[0, \text{size} - 1]$ . The problem with this approach is that it would take  $O(N)$  space for a large  $N$ .

Instead, let's attempt to solve using loop invariants. On the  $i$ th iteration of our loop to pick a random element, let's assume we already picked an element uniformly from  $[0, i - 1]$ . In order to maintain the loop invariant, we would need to pick the  $i$ th element as the new random element at  $1 / (i + 1)$  chance. For the base case where  $i = 0$ , let's say the random element is the first one. Then we know it works because

- For  $i \geq 0$ , before the loop began, any element  $K$  in  $[0, i - 1]$  had  $1 / i$  chance of being chosen as the random element. We want  $K$  to have  $1 / (i + 1)$  chance of being chosen after the iteration. This is the case since the chance of having been chosen already but not getting swapped with the  $i$ th element is  $1 / i (1 - (1 / (i + 1)))$  which is  $1 / i i / (i + 1)$  or  $1 / (i + 1)$

Let's see how the code would look:

```
import random
```

```
def pick(big_stream):
    random_element = None

    for i, e in enumerate(big_stream):
        if random.randint(1, i + 1) == 1:
            random_element = e
    return random_element
```

Since we are only storing a single variable, this only takes up constant space!

By the way, this is called **reservoir sampling**!

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #16

## Problem

This problem was asked by Twitter.

You run an e-commerce website and want to record the last N order ids in a log. Implement a data structure to accomplish this, with the following API:

- `record(order_id)`: adds the `order_id` to the log
- `get_last(i)`: gets the `i`th last element from the log. `i` is guaranteed to be smaller than or equal to `N`.

You should be as efficient with time and space as possible.

## Solution

It seems like an array would be the perfect fit for this problem. We can just initialize the array to have size `N`, and index it in constant time. Then, when we record any orders, we can pop off the first order and append it to the end. Getting the `i`th last order would then just be indexing the array at `length - i`.

```
class Log(object):  
    def __init__(self, n):  
        self._log = []  
        self.n = n  
  
    def record(self, order_id):  
        if len(self._log) >= self.n:  
            self._log.pop(0)  
        self._log.append(order_id)
```

```
def get_last(self, i):
    return self._log[-i]
```

This is one issue with this solution, however: when we have to pop off an element when the array is full, we have to move every other element down by 1. That means `record` takes  $O(N)$  time. How can we improve this?

What we can do to avoid having to move every element down by 1 is to keep a current index and move it up each time we record something. For `get_last`, we can simply take `current - i` to get the appropriate element. Now, both `record` and `get_last` should take constant time.

```
class Log(object):
    def __init__(self, n):
        self.n = n
        self._log = []
        self._cur = 0

    def record(self, order_id):
        if len(self._log) == self.n:

            self._log[self._cur] = order_id
        else:
            self._log.append(order_id)
        self._cur = (self._cur + 1) % self.n

    def get_last(self, i):
        return self._log[self._cur - i]
```

By the way, this is called a ring buffer or [circular buffer](#)!



# Daily Coding Problem #17

## Problem

This problem was asked by Google.

Suppose we represent our file system by a string in the following manner:

The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:

```
dir
  subdir1
  subdir2
    file.ext
```

The directory `dir` contains an empty sub-directory `subdir1` and a sub-directory `subdir2` containing a file `file.ext`.

The string

"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\t\tsubdir2\n\t\t\tsubsubdir2\n\t\t\t\tfile2.ext"

represents:

```
dir
  subdir1
    file1.ext
    subsubdir1
  subdir2
    subsubdir2
      file2.ext
```

The directory `dir` contains two sub-directories `subdir1` and `subdir2`. `subdir1` contains a file `file1.ext` and an empty second-level sub-directory `subsubdir1`. `subdir2` contains a second-level sub-directory `subsubdir2` containing a file `file2.ext`.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "`dir/subdir2/subsubdir2/file2.ext`", and its length is 32 (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to a file in the abstracted file system. If there is no file in the system, return 0.

Note:

The name of a file contains at least a period and an extension.

The name of a directory or sub-directory will not contain a period.

## Solution

There are two steps in solving this question: we must first parse the string representing the file system and then get the longest absolute path to a file.

### Step 1: Parsing the file system

Ideally, we would initially parse the string given into a dictionary of some sort. That would mean a string like:

```
dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\t\tsubdir2\n\t\t\tsubsubdir2\n\t\t\t\tfile2.ext
```

would become:

```
{  
    "dir": {  
        "subdir1": {  
            "file1.ext": True,  
            "subsubdir1": {}  
        },  
        "subdir2": {  
            "subsubdir2": {  
                "file2.ext": True  
            }  
        }  
    }  
}
```

where each key with a dictionary as its value represents a directory, and a key with True as its value represents an actual file.

To achieve this, we can first split the string by the newline character, meaning each item in our array represents a file or directory. Then, we create an empty dictionary to represent our parsed file system and traverse the file system on each entry. We keep track of the last path we've seen so far in `current_path` because we may need to return to some level in that path, depending on the number of tabs. Once we are at the correct place to put down the new directory or file, we check the name for a `.` and set the correct value to either `True` (if file) or `{}` (if directory).

```
def build_fs(input):  
    fs = {}  
    files = input.split('\n')  
  
    current_path = []  
    for f in files:  
        if '.' in f:  
            # file  
            name = f[f.rfind('\\')+1:f.rfind('.')]  
            if current_path[-1] == name:  
                fs[current_path[-1]] = True  
            else:  
                fs[current_path[-1]] = {}  
                current_path.append(name)  
                fs[name] = True  
        else:  
            # directory  
            name = f[f.rfind('\\')+1:]  
            if current_path[-1] == name:  
                fs[current_path[-1]] = True  
            else:  
                fs[current_path[-1]] = {}  
                current_path.append(name)  
                fs[name] = {}
```

```

indentation = 0
while '\t' in f[:2]:
    indentation += 1
    f = f[1:]

current_node = fs
for subdir in current_path[:indentation]:
    current_node = current_node[subdir]

if '.' in f:
    current_node[f] = True
else:
    current_node[f] = {}

current_path = current_path[:indentation]
current_path.append(f)

return fs

```

### Step 2: Computing the longest path

After we've constructed a native representation of the file system, we can write a fairly straightforward recursive function that takes the current root, recursively calculates the `longest_path` of all the subdirectories and files under the root, and returns the longest one. Remember that since we specifically want the longest path to a file to discard any paths that do not have a `.` in them. And if there are no paths starting at this root, then we can simply return the empty string.

```

def longest_path(root):
    paths = []
    for key, node in root.items():
        if node == True:
            paths.append(key)
        else:
            paths.append(key + '/' + longest_path(node))
    # filter out unfinished paths
    paths = [path for path in paths if '.' in path]
    if paths:
        return max(paths, key=lambda path:len(path))
    else:
        return ''

```

### Step 3: Putting it together

Now that the hard part is done, we just need to put the two together:

```

def longest_absolute_path(s):
    return len(longest_path(build_fs(s)))

```

This runs in  $O(n)$ , since we iterate over the input string twice to build the file system, and then in the worst case we go through the string again to compute the longest path.

---

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #18

## Problem

This problem was asked by Google.

Given an array of integers and a number  $k$ , where  $1 \leq k \leq$  length of the array, compute the maximum values of each subarray of length  $k$ .

For example, given array = [10, 5, 2, 7, 8, 7] and  $k = 3$ , we should get: [10, 7, 8, 8], since:

- $10 = \max(10, 5, 2)$
- $7 = \max(5, 2, 7)$
- $8 = \max(2, 7, 8)$
- $8 = \max(7, 8, 7)$

Do this in  $O(n)$  time and  $O(k)$  space. You can modify the input array in-place and you do not need to store the results. You can simply print them out as you compute them.

## Solution

Even though the question states  $O(n)$ , in an interview it's always useful to first write out a brute force solution, which may provide us with some insight on some deeper structure in the problem.

So let's first write out a naive solution: we can simply take each subarray of  $k$  length and compute their maxes.

```
def max_of_subarrays(lst, k):
    for i in range(len(lst) - k + 1):
        print(max(lst[i:i + k]))
```

This takes  $O(n * k)$  time, which doesn't get us quite to where we want. How can we make this faster?

One possible idea is this: we could use a max-heap of size k and add the first k elements to the heap initially, and then pop off the max and add the next element for the rest of the array. This is better, but adding and extracting from the heap will take  $O(\log k)$ , so this algorithm will take  $O(n * \log k)$ , which is still not enough. How can we do better?

Notice that, for example, the input [1, 2, 3, 4, 5, 6, 7, 8, 9] and  $k = 3$ , after evaluating the max of first range, since 3 is at the end, we only need to check whether 4 is greater than 3. If it is, then we can print 4 immediately, and if it isn't, we can stick with 3.

On the other hand, for the input [9, 8, 7, 6, 5, 4, 3, 2, 1] and  $k = 3$ , after evaluating the max of the first range, we can't do the same thing, since we can't use 9 again. We have to look at 8 instead, and then once we move on to the next range, we have to look at 7.

These two data points suggest an idea: we can keep a double-ended queue with max size k and only keep what we need to evaluate in it. That is, if we see [1, 3, 5], then we only need to keep [5], since we know that 1 and 3 cannot possibly be the maxes.

So what we can do is maintain an ordered list of indices, where we only keep the elements we care about, that is, we will maintain the loop invariant that our queue is always ordered so that we only keep the indices we care about (i.e, there are no elements that are greater after, since we would just pick the greater element as the max instead).

It will help to go over an example. Consider our test input: [10, 5, 2, 7, 8, 7] and  $k = 3$ . Our queue at each step would look like this (recall that these are indices):

## Preprocessing

After processing 10: [0] After processing 5: [0, 1] # 5 is smaller than 10, and 10 is still valid until we hit the 3rd index After processing 2: [0, 1, 2] # 2 is smaller than 5, and 10 is still valid

## Main Loop

Print value of first element in our queue: **10**

After processing 7: [4] # 10 is no longer valid (we can tell since the current index - 0 > k), so we dequeue from the front. 7 is bigger than 5 and 2, so we get rid of them from the back and replace it with the 7

Print value of first element in our queue: **7**

After processing 8: [5] # 8 is bigger than 7, so no point in keeping 7 around. We get rid of it from the back and replace it with the 8

Print value of first element in our queue: **8**

After processing 7: [5, 4] # 7 is smaller than 8, so we enqueue it from the back

Print value of first element in our queue: **8**

## Code

```
from collections import deque

def max_of_subarrays(lst, k):
    q = deque()
    for i in range(k):
        while q and lst[i] >= lst[q[-1]]:
            q.pop()
        q.append(i)

    # Loop invariant: q is a list of indices where their corresponding values are in descending order.

    for i in range(k, len(lst)):
        print(lst[q[0]])
        while q and q[0] <= i - k:
            q.popleft()
        while q and lst[i] >= lst[q[-1]]:
            q.pop()
        q.append(i)
    print(lst[q[0]])
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #20

## Problem

This problem was asked by Google.

Given two singly linked lists that intersect at some point, find the intersecting node. The lists are non-cyclical.

For example, given A = 3 -> 7 -> 8 -> 10 and B = 99 -> 1 -> 8 -> 10, return the node with value 8.

In this example, assume nodes with the same value are the exact same node objects.

Do this in  $O(M + N)$  time (where M and N are the lengths of the lists) and constant space.

## Solution

We might start this problem by first ignoring the time and space constraints, in order to get a better grasp of the problem.

Naively, we could iterate through one of the lists and add each node to a set or dictionary, then we could iterate over the other list and check each node we're looking at to see if it's in the set. Then we'd return the first node that is present in the set. This takes  $O(M + N)$  time but also  $O(\max(M, N))$  space (since we don't know initially which list is longer). How can we reduce the amount of space we need?

We can get around the space constraint with the following trick: first, get the length of both lists. Find the difference between the two, and then keep two pointers at the head of each list. Move the pointer of the larger list up by the difference, and then move the pointers forward in conjunction and check if they match.

```
def length(head):
    if not head:
        return 0
    return 1 + length(head.next)

def intersection(a, b):
```

```
m, n = length(a), length(b)
cur_a, cur_b = a, b

if m > n:
    for _ in range(m - n):
        cur_a = cur_a.next
else:
    for _ in range(n - m):
        cur_b = cur_b.next

while cur_a != cur_b:
    cur_a = cur_a.next
    cur_b = cur_b.next

return cur_a
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #21

## Problem

This problem was asked by Snapchat.

Given an array of time intervals (start, end) for classroom lectures (possibly overlapping), find the minimum number of rooms required.

For example, given [(30, 75), (0, 50), (60, 150)], you should return 2.

## Solution

First, notice that the minimum number of classroom halls is the maximum number of overlapping intervals.

Now let's consider the naive approach. We could go through each interval and check every other interval and see if it overlaps, keeping track of the largest number of overlapping intervals.

```
def overlaps(a, b):
    start_a, end_a = a
    start_b, end_b = b
    # It doesn't overlap if it's like this:
    # |start_a .... end_a|  <--> |start_b ... end_b|
    # or like this:
    # |start_b .... end_b|  <--> |start_a ... end_a|
    # so return not or either of these
    return not (end_a < start_b or start_a > end_b)

def max_overlapping(intervals):
    current_max = 0
    for interval in intervals:
        num_overlapping = sum(overlaps(interval, other_interval)
            for other_interval in intervals
            if interval is not other_interval)
        current_max = max(current_max, num_overlapping)
```

```
return current_max
```

This would take  $O(n^2)$  time, since we're checking each interval pairwise. Can we do any better?

One solution is to extract the start times and end times of all the intervals and sort them. Then we can start two pointers on each list, and consider the following:

- If the current start is before the current end, then we have a new overlap. Increment the start pointer.
- If the current start is after the current end, then our overlap closes. Increment the end pointer.

All that's left to do is keep a couple variables to keep track of the maximum number of overlaps we've seen so far and the current number of overlaps.

```
def max_overlapping(intervals):  
    starts = sorted(start for start, end in intervals)  
    ends = sorted(end for start, end in intervals)  
  
    current_max = 0  
    current_overlap = 0  
    i, j = 0, 0  
    while i < len(intervals) and j < len(intervals):  
        if starts[i] < ends[j]:  
            current_overlap += 1  
            current_max = max(current_max, current_overlap)  
            i += 1  
        else:  
            current_overlap -= 1  
            j += 1  
    return current_max
```

This runs in  $O(n \log n)$  time, since we have to sort the intervals.

# Daily Coding Problem #22

## Problem

This problem was asked by Microsoft.

Given a dictionary of words and a string made up of those words (no spaces), return the original sentence in a list. If there is more than one possible reconstruction, return any of them. If there is no possible reconstruction, then return null.

For example, given the set of words 'quick', 'brown', 'the', 'fox', and the string "thequickbrownfox", you should return ['the', 'quick', 'brown', 'fox'].

Given the set of words 'bed', 'bath', 'bedbath', 'and', 'beyond', and the string "bedbathandbeyond", return either ['bed', 'bath', 'and', 'beyond'] or ['bedbath', 'and', 'beyond'].

## Solution

We might be initially tempted to take a greedy approach to this problem, by for example, iterating over the string and checking if our current string matches so far. However, you should immediately find that that can't work: consider the dictionary {'the', 'theremin'} and the string 'theremin': we would find 'the' first, and then we wouldn't be able to match 'remin'.

So this greedy approach doesn't work, since we would need to go back if we get stuck. This gives us a clue that we might want to use [backtracking](#) to help us solve this problem. We also have the following idea for a recurrence: If we split up the string into a prefix and suffix, then we can return the prefix extended with a list of the rest of the sentence, but only if they're both valid. So what we can do is the following:

- Iterate over the string and split it into a prefix and suffix
- If the prefix is valid (appears in the dictionary), then recursively call on the suffix
- If that's valid, then return. Otherwise, continue searching.
- If we've gone over the entire sentence and haven't found anything, then return empty.

We'll need a helper function to tell us whether the string can actually be broken up into a sentence as well, so let's define `find_sentence_helper` that also returns whether or not the sentence is valid.

```
def find_sentence(dictionary, s):
    sentence, valid = find_sentence_helper(dictionary, s)
    if valid:
        return sentence

def find_sentence_helper(dictionary, s):
    if len(s) == 0:
        return [], True

    result = []
    for i in range(len(s) + 1):
        prefix, suffix = s[:i], s[i:]
        if prefix in dictionary:
            rest, valid = find_sentence_helper(dictionary, suffix)
            if valid:
                return [prefix] + rest, True
    return [], False
```

This will run in  $O(2^N)$  time, however. This is because in the worst case, say, for example,  $s = "aaaaab"$  and  $\text{dictionary} = ["a", "aa", "aaa", "aaaa", "aaaaa"]$ , we will end up exploring every single path, or every combination of letters, and the total number of combinations of characters is  $2^N$ .

We can improve the running time by using dynamic programming to store repeated subcomputations. This reduces the running time to just  $O(N^2)$ . We'll keep a dictionary that maps from indices to the last word that can be made up to that index. We'll call these starts. Then, we just need to do two nested for loops, one that iterates over the whole string and tries to find a start at that index, and a loop that checks each start to see if a new word can be made from that start to the current index.

Now we can simply take the start at the last index and build our sentence backwards:

```
def find_sentence(s, dictionary):
    starts = {0: ''}
    for i in range(len(s) + 1):
        new_starts = starts.copy()
        for start_index, _ in starts.items():
            word = s[start_index:i]
            if word in dictionary:
                new_starts[i] = word
        starts = new_starts.copy()
```

```
result = []
current_length = len(s)
if current_length not in starts:
    return None
while current_length > 0:
    word = starts[current_length]
    current_length -= len(word)
    result.append(word)

return list(reversed(result))
```

Now this runs in  $O(N^2)$  time and  $O(N)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #23

## Problem

This problem was asked by Google.

You are given an M by N matrix consisting of booleans that represents a board. Each True boolean represents a wall. Each False boolean represents a tile you can walk on.

Given this matrix, a start coordinate, and an end coordinate, return the minimum number of steps required to reach the end coordinate from the start. If there is no possible path, then return null. You can move up, left, down, and right. You cannot move through walls. You cannot wrap around the edges of the board.

For example, given the following board:

```
[[f, f, f, f],  
 [t, t, f, t],  
 [f, f, f, f],  
 [f, f, f, f]]
```

and start = (3, 0) (bottom left) and end = (0, 0) (top left), the minimum number of steps required to reach the end is 7, since we would need to go through (1, 2) because there is a wall everywhere else on the second row.

## Solution

The idea here is to use either BFS or DFS to explore the board, starting from the start coordinate, and keep track of what we've seen so far as well as the steps from the start until we find the end coordinate.

In our case, we'll use BFS. We'll create a queue and initialize it with our start coordinate, along with a count of 0. We'll also initialize a seen set to ensure we only add coordinates we haven't seen before.

Then, as long as there's something still in the queue, we'll dequeue from the queue and first check if it's our target coordinate -- if it is, then we can just immediately return the count. Otherwise, we'll get the valid neighbours of the coordinate we're working with (valid means not off the board and not a wall), and enqueue them to the end of the queue.

To make sure the code doesn't get too messy, we'll define some helper functions: `walkable`, which returns whether or not a tile is valid, and `get_walkable_neighbours` which returns the valid neighbours of a coordinate.

```
from collections import deque

# Given a row and column, returns whether that tile is walkable.
def walkable(board, row, col):
    if row < 0 or row >= len(board):
        return False
    if col < 0 or col >= len(board[0]):
        return False
    return not board[row][col]

# Gets walkable neighbouring tiles.
def get_walkable_neighbours(board, row, col):
    return [(r, c) for r, c in [
        (row, col - 1),
        (row - 1, col),
        (row + 1, col),
        (row, col + 1)]
        if walkable(board, r, c)
    ]]

def shortest_path(board, start, end):
    seen = set()
    queue = deque([(start, 0)])
    while queue:
        coords, count = queue.popleft()
        if coords == end:
            return count
        seen.add(coords)
        neighbours = get_walkable_neighbours(board, coords[0], coords[1])
        queue.extend((neighbour, count + 1) for neighbour in neighbours
                    if neighbour not in seen)

board = [[False, False, False, False],
         [True, True, True, True],
         [False, False, False, False],
```

```
[False, False, False, False]]  
  
print(shortest_path(board, (3, 0), (0, 0)))
```

This code should run in  $O(M * N)$  time and space, since in the worst case we need to examine the entire board to find our target coordinate.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #24

## Problem

This problem was asked by Google.

Implement locking in a binary tree. A binary tree node can be locked or unlocked only if all of its descendants or ancestors are not locked.

Design a binary tree node class with the following methods:

- `is_locked`, which returns whether the node is locked
- `lock`, which attempts to lock the node. If it cannot be locked, then it should return false. Otherwise, it should lock it and return true.
- `unlock`, which unlocks the node. If it cannot be unlocked, then it should return false. Otherwise, it should unlock it and return true.

You may augment the node to add parent pointers or any other property you would like. You may assume the class is used in a single-threaded program, so there is no need for actual locks or mutexes. Each method should run in  $O(h)$ , where  $h$  is the height of the tree.

## Solution

A relatively easy way to implement this would be to augment each node with an `is_locked` attribute as well as a parent pointer. We can then implement the methods in a straightforward manner:

- `is_locked` simply returns the node's attribute
- `lock` searches the node's children and parents for a true `is_locked` attribute. If it is set to true on any of them, then return false. Otherwise, set the current node's `is_locked` to

true and return true.

- unlock simply changes the node's attribute to false. If we want to be safe, then we should search the node's children and parents as in lock to make sure we can actually unlock the node, but that shouldn't ever happen.

While `is_locked` is  $O(1)$  time, `lock` and `unlock` will take  $O(m + h)$  time where  $m$  is the number of nodes in the node's subtree (since we have to traverse through all its descendants) and  $h$  is the height of the node (since we have to traverse through the node's ancestors).

We can improve the performance of `lock` and `unlock` by adding another field to the node that keeps tracks of the count of locked descendants. That way, we can immediately see whether any of its descendants are locked. This will reduce our `lock` and `unlock` functions to only  $O(h)$ . We can maintain this field by doing the following:

- When locking, if the locking succeeds, traverse the node's ancestors and increment each one's count
- When unlocking, traverse the node's ancestors and decrement each one's count

The code will look something like the following:

```
class LockingBinaryTreeNode(object):  
    def __init__(self, val, left=None, right=None, parent=None):  
        self.val = val  
        self.left = left  
        self.right = right  
  
        self.parent = parent  
        self.is_locked = False  
        self.locked_descendants_count = 0  
  
    def _can_lock_or_unlock(self):  
        if self.locked_descendants_count > 0:  
            return False  
  
        cur = self.parent  
        while cur:  
            if cur.is_locked:  
                return False  
            cur = cur.parent  
        return True  
  
    def is_locked(self):  
        return self.is_locked
```

```

def lock(self):
    if self.is_locked:
        return False # node already locked

    if not self._can_lock_or_unlock():
        return False

    # Not locked, so update is_locked and increment count in all ancestors
    self.is_locked = True

    cur = self.parent
    while cur:
        cur.locked_descendants_count += 1
        cur = cur.parent
    return True

def unlock(self):
    if not self.is_locked:
        return False # node already unlocked

    if not self._can_lock_or_unlock():
        return False

    self.is_locked = False

    # Update count in all ancestors
    cur = self.parent
    while cur:
        cur.locked_descendants_count -= 1
        cur = cur.parent
    return True

```

Now, `is_locked` is still  $O(1)$ , but `lock` and `unlock` are both  $O(h)$  instead of  $O(m + h)$ .

Terms of Service  
Press

# Daily Coding Problem #26

## Problem

This problem was asked by Google.

Given a singly linked list and an integer  $k$ , remove the  $k^{\text{th}}$  last element from the list.  $k$  is guaranteed to be smaller than the length of the list.

The list is very long, so making more than one pass is prohibitively expensive.

Do this in constant space and in one pass.

## Solution

If we didn't have the constraint of needing only to make one pass, this problem would be trivial to implement. We could simply iterate over the whole list to find out the total length  $N$  of the list, and then restart from the beginning and iterate  $N - k$  steps and remove the node there. That would take constant space as well.

However, given that we have the constraint of needing to make only one pass, we have to find some way of getting the  $N - k^{\text{th}}$  node in the list in one shot.

What we can do, then, is this:

- Set up two pointers at the head of the list (let's call them `fast` and `slow`)
- Move `fast` up by  $k$
- Move both `fast` and `slow` together until `fast` reaches the end of the list
- Now `slow` is at the  $N - k^{\text{th}}$  node, remove it

That only makes one pass and is constant time. The code should look something like this:

```
class Node:  
    def __init__(self, val, next=None):  
        self.val = val  
        self.next = next
```

```
self.next = next

def __str__(self):
    current_node = self
    result = []
    while current_node:
        result.append(current_node.val)
        current_node = current_node.next
    return str(result)

def remove_kth_from_linked_list(head, k):
    slow, fast = head, head
    for i in range(k):
        fast = fast.next

    prev = None
    while fast:
        prev = slow
        slow = slow.next
        fast = fast.next

    prev.next = slow.next

head = Node(1, Node(2, Node(3, Node(4, Node(5)))))
print(head)
remove_kth_from_linked_list(head, 3)
print(head)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #27

## Problem

This problem was asked by Facebook.

Given a string of round, curly, and square open and closing brackets, return whether the brackets are balanced (well-formed).

For example, given the string "`([]){}`", you should return true.

Given the string "`([])`" or "`((()`", you should return false.

## Solution

In this case, it's easy to start with a simplified case of the problem, which is dealing with only round brackets. Notice that in this case, we just need to keep track of the current number of open brackets -- each closing bracket should be matched with the rightmost open bracket. So we can keep a counter and increment it for every open bracket we see and decrement it on every closing bracket. If we get to the end of the string and have a non-zero number, then it means it's unbalanced. A negative number would indicate more closing brackets than open ones, and a positive number would indicate the opposite.

In the case of round, curly, and square brackets, we need to also keep track of what *kind* of brackets they are as well, because we can't match a round open bracket with a curly square. In this case, we can use a stack to keep track of the actual bracket character and push onto it whenever we encounter an open bracket, and pop if we encounter a matching closing bracket to the top of the stack. If the stack is empty or it's not the correct matching bracket, then we'll return false. If, by the end of the iteration, we have something left over in the stack, then it means it's unbalanced -- so we'll return whether it's empty or not.

```
def balance(s):
    stack = []
    for char in s:
        if char in [")", "]", "}"]:
            stack.append(char)
        elif char in ["(", "[", "{"]:
            if len(stack) == 0 or stack[-1] != char:
                return False
            stack.pop()
    return len(stack) == 0
```

```
else:
    # Check character is not unmatched
    if not stack:
        return False

    # Char is a closing bracket, check top of stack if it matches
    if (char == ")" and stack[-1] != "(") or \
       (char == "]" and stack[-1] != "[") or \
       (char == "}" and stack[-1] != "{"):
        return False
    stack.pop()

return len(stack) == 0
```

Fun fact: "()" is not a palindrome, nor is "()()". "()()" is a palindrome, though.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #28

## Problem

This problem was asked by Palantir.

Write an algorithm to justify text. Given a sequence of words and an integer line length k, return a list of strings which represents each line, fully justified.

More specifically, you should have as many words as possible in each line. There should be at least one space between each word. Pad extra spaces when necessary so that each line has exactly length k. Spaces should be distributed as equally as possible, with the extra spaces, if any, distributed starting from the left.

If you can only fit one word on a line, then you should pad the right-hand side with spaces.

Each word is guaranteed not to be longer than k.

For example, given the list of words ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"] and k = 16, you should return the following:

```
["the quick brown", # 1 extra space on the left  
"fox jumps over", # 2 extra spaces distributed evenly  
"the lazy dog"] # 4 extra spaces distributed evenly
```

## Solution

It seems like the justification algorithm is independent from the groupings, so immediately we should figure out two things:

- How to group lines together so that it is as close to k as possible (without going over)
- Given a grouping of lines, justifying the text by appropriately distributing spaces

To solve the first part, let's write a function `group_lines` that takes in all the words in our input sequence as well as our target line length k, and return a list of lists of words that represents the

sequences as well as our target line length  $k$ , and return a list of lists of words that represents the lines that we will eventually justify. Our main strategy will be to iterate over all the words, keep a list of words for the current line, and because we want to fit as many words as possible per line, estimate the current line length, assuming only one space between each word. Once we go over  $k$ , then save the word and start a new line with it. So our function will look something like this:

```
def min_line(words):
    return ' '.join(words)

def group_lines(words, k):
    ...
    Returns groupings of |words| whose total length, including 1 space in between,
    is less than |k|.
    ...
    groups = []
    current_sum = 0
    current_line = []
    for i, word in enumerate(wordwordss):
        # Check if adding the next word would push it over
        # the limit. If it does, then add |current_line| to
        # group. Also reset |current_line| properly.
        if len(min_line(current_line + [word])) > k:
            groups.append(current_line)
            current_line = []
        current_line.append(word)

    # Add the last line to groups.
    groups.append(current_line)
    return groups
```

Then, we'll want to actually justify each line. We know for sure each line we feed from `group_lines` is the maximum number of words we can pack into a line and no more. What we can do is first figure out how many spaces we have available to distribute between each word. Then from that, we can calculate how much base space we should have between each word by dividing it by the number of words minus one. If there are any leftover spaces to distribute, then we can keep track of that in a counter, and as we rope in each new word we'll add the appropriate number of spaces. We can't add more than one leftover space per word.

```
def justify(words, length):
    ...
    Precondition: |words| can fit in |length|.
    Justifies the words using the following algorithm:
        - Find the smallest spacing between each word (available_spaces / spaces)
        - Add a leftover space one-by-one until we run out
    ...
    if len(words) == 1:
```

```
word = words[0]
num_spaces = length - len(word)
spaces = ' ' * num_spaces
return word + spaces

spaces_to_distribute = length - sum(len(word) for word in words)
number_of_spaces = len(words) - 1
smallest_space = floor(spaces_to_distribute / number_of_spaces)
leftover_spaces = spaces_to_distribute - (number_of_spaces * smallest_space)
justified_words = []
for word in words:
    justified_words.append(word)
    current_space = ' ' * smallest_space
    if leftover_spaces > 0:
        current_space += ' '
        leftover_spaces -= 1
    justified_words.append(current_space)
return ''.join(justified_words).rstrip()
```

The final solution should just combine our two functions:

```
def justify_text(words, k):
    return [justify(group, k) for group in group_lines(words, k)]
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #29

## Problem

This problem was asked by Amazon.

Run-length encoding is a fast and simple method of encoding strings. The basic idea is to represent repeated successive characters as a single count and character. For example, the string "AAAABBBCCDAA" would be encoded as "4A3B2C1D2A".

Implement run-length encoding and decoding. You can assume the string to be encoded have no digits and consists solely of alphabetic characters. You can assume the string to be decoded is valid.

## Solution

We can implement encode by iterating over our input string and keeping a current count of whatever the current character is, and once we encounter a different one, appending the count (as a string) and the actual character to our result string.

```
def encode(s):
    if not s:
        return ''

    result = ''
    current_char = s[0]
    current_count = 1

    for i, char in enumerate(s, 1):
        if char == current_char:
            current_count += 1
        else:
            result += str(current_count) + current_char
            current_char = char
            current_count = 1

    result += str(current_count) + current_char
```

```
return result
```

We can implement decode by iterating over the encoded string and checking each character for a digit. If it is, then calculate the correct count, and once we find its corresponding character, extend the result with the character count number of times and then reset the count.

```
def decode(s):
    count = 0
    result = ''
    for char in s:
        if char.isdigit():
            count = count * 10 + int(char)
        else:
            # char is alphabetic
            result += char * count
            count = 0
    return result
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #31

## Problem

This problem was asked by Google.

The edit distance between two strings refers to the minimum number of character insertions, deletions, and substitutions required to change one string to the other. For example, the edit distance between "kitten" and "sitting" is three: substitute the "k" for "s", substitute the "e" for "i", and append a "g".

Given two strings, compute the edit distance between them.

## Solution

First, notice that we can probably define this problem recursively. How can we notice this? If we look at the example (kitten -> sitting) and its solution path (kitten -> sitten -> sittin -> sitting), we can see that it's the minimum distance between sitten and sitting plus one.

The recurrence, then, looks like this:

- If either  $s_1$  or  $s_2$  are empty, then return the size of the larger of the two strings (since we can trivially turn an empty string into a string by inserting all its characters)
- Otherwise, return the minimum between:
  - The edit distance between each string and the last  $n - 1$  characters of the other plus one
  - If the first character in each string is the same, then the edit distance between  $s_1[1:]$  and  $s_2[1:]$ , otherwise the same edit distance + 1

So, the naive recursive solution would look like this:

```
def distance(s1, s2):
```

```

if len(s1) == 0 or len(s2) == 0:
    return max(len(s1), len(s2))

return min(distance(s1[1:], s2) + 1,
           distance(s1, s2[1:]) + 1,
           distance(s1[1:], s2[1:]) if s1[0] == s2[0]
           else distance(s1[1:], s2[1:]) + 1)

```

However, this runs very slowly due to repeated subcomputations. We can speed it up by using dynamic programming and storing the subcomputations in a 2D matrix. The index at  $i, j$  will contain the edit distance between  $s1[:i]$  and  $s2[:j]$ . Then, once we fill it up, we can return the value of the matrix at  $A[-1][-1]$ .

```

def distance(s1, s2):
    x = len(s1) + 1 # the length of the x-coordinate
    y = len(s2) + 1 # the length of the y-coordinate

    A = [[-1 for i in range(x)] for j in range(y)]
    for i in range(x):
        A[0][i] = i

    for j in range(y):
        A[j][0] = j

    for i in range(1, y):
        for j in range(1, x):
            if s1[j- 1] == s2[i - 1]:
                A[i][j] = A[i - 1][j - 1]
            else:
                A[i][j] = min(
                    A[i - 1][j] + 1,
                    A[i][j - 1] + 1,
                    A[i - 1][j - 1] + 1
                )

    return A[y - 1][x - 1] # return the edit distance between the two strings

```

This now takes  $O(N * M)$  time and space, where  $N$  and  $M$  are the lengths of the strings.

Press

# Daily Coding Problem #35

## Problem

This problem was asked by Google.

Given an array of strictly the characters 'R', 'G', and 'B', segregate the values of the array so that all the Rs come first, the Gs come second, and the Bs come last. You can only swap elements of the array.

Do this in linear time and in-place.

For example, given the array ['G', 'B', 'R', 'R', 'B', 'R', 'G'], it should become ['R', 'R', 'R', 'G', 'G', 'B', 'B'].

## Solution

It may be easier to first consider an easier problem: one with only two possible values, say 'R' and 'G'. Then we could maintain the following loop invariant quite easily:

- Maintain three sections of the array using two indices, `low` and `high`:
  - Strictly 'R's: `array[:low]`
  - Unknown: `array[low:high]`
  - Strictly 'G's: `array[high:]`

Initially, `low` will be 0 and `high` will be `len(array) - 1`, since the whole array is unknown. As we iterate over the array, we'll swap any 'G's we see to the third section and decrement `high`. If we see an 'R', then we just need to increment `low`, since that's where it belongs. We can terminate once `low` crosses `high`. So we can gradually shrink our unknown section through the following algorithm:

```
def partition(arr):  
    low, high = 0, len(arr) - 1  
    while low <= high:  
        if arr[low] == 'R':
```

```

    low += 1
else:
    arr[low], arr[high] = arr[high], arr[low]
    high -= 1

```

This correctly partitions our array into two separate categories. How can we extend this to three partitions? Let's maintain four sections using 3 indices, `low`, `mid`, and `high`:

- Strictly 'R's: `array[:low]`
- Strictly 'G's: `array[low:mid]`
- Unknown: `array[mid:high]`
- Strictly 'B's: `array[high:]`

We'll initialize `low` and `mid` both to 0, and `high` to `len(array) - 1` so that our unknown section is the whole array, as before. To maintain this invariant, we should do the following:

- Look at `array[mid]`:
  - If it's R, then swap `array[low]` with `array[high]` and increment `low` and `mid`
  - If it's G, then just increment `mid`; it's where it should be
  - If it's B, then swap `array[mid]` with `array[high]` and decrement `high`

Once `mid` crosses over with `high`, then our unknown section is gone and we can terminate.

Our solution looks like this:

```

def partition(arr):
    low, mid, high = 0, 0, len(arr) - 1
    while mid <= high:
        if arr[mid] == 'R':
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 'G':
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1

```

P.S. This problem is also called the [Dutch national flag problem!](#)

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #42

## Problem

This problem was asked by Google.

Given a list of integers S and a target number k, write a function that returns a subset of S that adds up to k. If such a subset cannot be made, then return null.

Integers can appear more than once in the list. You may assume all numbers in the list are positive.

For example, given S = [12, 1, 61, 5, 9, 2] and k = 24, return [12, 9, 2, 1] since it sums up to 24.

## Solution

Let's consider the brute force method: selecting all subsets, summing them, and checking if they equal k. That would take  $O(2^N * N)$  time, since generating all subsets takes  $O(2^N)$  and we need to sum everything in the subset.

We can do a little better by implicitly computing the sum. That is, for each call, we can basically choose whether to pick some element (let's say the last) in our set and recursively looking for  $k - \text{last}$  in the remaining part of the list, or exclude the last element and keep on looking for  $k$  in the remaining part of the list recursively.

```
def subset_sum(nums, k):
    if k == 0:
        return []
    if not nums and k != 0:
        return None

    nums_copy = nums[:]
    last = nums_copy.pop()
```

```

with_last = subset_sum(nums_copy, k - last)
without_last = subset_sum(nums_copy, k)
if with_last is not None:
    return with_last + [last]
if without_last is not None:
    return without_last

```

This will run in  $O(2^N)$  theoretically, but practically, since we copy the whole array on each call, it's still  $O(2^N * N)$ , which is worse than exponential.

Let's try to improve the running time by using dynamic programming. We have the recursive formula nailed down. How can we use bottom-up dynamic programming to improve the runtime?

We can construct a table A that's size  $\text{len}(\text{nums}) + 1$  by  $k + 1$ . At each index  $A[i][j]$ , we'll keep a subset of the list from  $0..i$  (including lower, excluding upper bound) that can add up to  $j$ , or null if no list can be made. Then we will fill up the table using pre-computed values and once we're done, we should be able to just return the value at  $A[-1][-1]$ . Let's first initialize the list:

```
A = [[None for _ in range(k + 1)] for _ in range(len(nums) + 1)]
```

To begin, we can initialize each element of the first row ( $A[i][0]$  for  $i$  in  $\text{range}(\text{len}(\text{nums}) + 1)$ ) with the empty list, since any subset of the list can make 0: just don't pick anything!

```

for i in range(len(nums) + 1):
    A[i][0] = []

```

Each element of the first column ( $A[0][j]$  for  $j$  in  $\text{range}(1, \text{len}(\text{nums}))$ ) starting from the first row should be null, since we can't make anything other than 0 with the empty set. Since we've initialized our whole table to be null, then we don't need to do anything here.

```

[], [], [], [], ..., ...
None, None, None, ...
None, None, None, ...
None, None, None, ...
...

```

Now we can start populating the table. Iterating over each row starting at 1, and then each column starting at 1, we can use the following formula to compute  $A[i][j]$ :

- First, let's consider the last element of the list we're looking at:  $\text{nums}[i - 1]$ . Let's call this `last`.

- If `last` is greater than `j`, then we definitely can't make `j` with `nums[:i]` including `last` (since it would obviously go over). So let's just copy over whatever we had from `A[i - 1][j]`. If we can make `j` without `last`, then we can still make `j`. If we can't, then we still can't.
- If `last` smaller than or equal to `j`, then we still might be able to make `j` using `last`
  - If we can make `j` without `last` by looking up the value at `A[i - 1][j]` and if it's not null, then use that.
  - Else, if we can't make `j` without `last`, check if we can make it *with* `last` by looking up the value at `A[i - 1][j - last]`. If we can, then copy over the list from there and append the `last` element to it.
  - Else, we can't make it with or without `j`, so set `A[i][j]` to null.

```

for i in range(1, len(nums) + 1):
    for j in range(1, k + 1):
        last = nums[i - 1]
        if last > j:
            A[i][j] = A[i - 1][j]
        else:
            if A[i - 1][j] is not None:
                A[i][j] = A[i - 1][j]
            elif A[i - 1][j - last] is not None:
                A[i][j] = A[i - 1][j - last] + [last]
            else:
                A[i][j] = None

```

Putting it all together:

```

def subset_sum(nums, k):
    A = [[None for _ in range(k + 1)] for _ in range(len(nums) + 1)]

    for i in range(len(nums) + 1):
        A[i][0] = []

    for i in range(1, len(nums) + 1):
        for j in range(1, k + 1):

            last = nums[i - 1]
            if last > j:
                A[i][j] = A[i - 1][j]
            else:
                if A[i - 1][j] is not None:
                    A[i][j] = A[i - 1][j]
                elif A[i - 1][j - last] is not None:
                    A[i][j] = A[i - 1][j - last] + [last]
                else:
                    A[i][j] = None

```

```
else:  
    if A[i - 1][j] is not None:  
        A[i][j] = A[i - 1][j]  
    elif A[i - 1][j - last] is not None:  
        A[i][j] = A[i - 1][j - last] + [last]  
    else:  
        A[i][j] = None  
  
return A[-1][-1]
```

This runs in  $O(k * N)$  time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #43

## Problem

This problem was asked by Amazon.

Implement a stack that has the following methods:

- `push(val)`, which pushes an element onto the stack
- `pop()`, which pops off and returns the topmost element of the stack. If there are no elements in the stack, then it should throw an error or return null.
- `max()`, which returns the maximum value in the stack currently. If there are no elements in the stack, then it should throw an error or return null.

Each method should run in constant time.

## Solution

Implementing the stack part (push and pop) of this problem is easy -- we can just use a typical list to implement the stack with `append` and `pop`. However, getting the max in constant time is a little trickier. We could obviously do it in linear time if we popped off everything on the stack while keeping track of the maximum value, and then put everything back on.

We can use a secondary stack that *only* keeps track of the max values at any time. It will be in sync with our primary stack, as in it will have the exact same number of elements as our primary stack at any point in time, but the top of the stack will always contain the maximum value of the stack.

We can then, when pushing, check if the element we're pushing is greater than the max value of the secondary stack (by just looking at the top), and if it is, then push that instead. If not, then maintain the previous value.

```
class MaxStack:  
    def __init__(self):
```

```
self.stack = []
self.maxes = []

def push(self, val):
    self.stack.append(val)
    if self.maxes:
        self.maxes.append(max(val, self.maxes[-1]))
    else:
        self.maxes.append(val)

def pop(self):
    if self.maxes:
        self.maxes.pop()
    return self.stack.pop()

def max(self):
    return self.maxes[-1]
```

Everything should run in O(1) time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #46

## Problem

This problem was asked by Amazon.

Given a string, find the longest palindromic contiguous substring. If there are more than one with the maximum length, return any one.

For example, the longest palindromic substring of "aabcdcb" is "bcdcb". The longest palindromic substring of "bananas" is "anana".

## Solution

We can compute the longest palindromic contiguous substring in  $O(N^3)$  using brute force. We can just iterate over each substring of the array and check if it's a palindrome.

```
def is_palindrome(s):
    return s[::-1] == s

def longest_palindrome(s):
    longest = ''
    for i in range(len(s) - 1):
        for j in range(i + 1, len(s)):
            substring = s[i:j]
            if is_palindrome(substring) and len(substring) > len(longest):
                longest = substring
    return longest
```

We can improve the running time of this algorithm by using dynamic programming to store any repeated computations. Let's keep an  $N \times N$  table  $A$ , where  $N$  is the length of the input string. Then, at each index  $A[i][j]$  we'll keep whether or not the substring made from  $s[i:j]$  is a palindrome. We'll make use of the following relationships:

- All strings of length 1 are palindromes

- $s$  is a palindrome if  $s[1:-1]$  is a palindrome and the first and last character of  $s$  are the same

So, when we fill up our table, we can do the following:

- First, set each item along the diagonal  $A[i:i]$  to true, since strings of length 1 are always palindromes
- Then, check  $A[i:i+1]$  and set it to true if  $A[i] == A[i + 1]$  and false otherwise (check all strings of length 2)
- Finally, iterate over the matrix from top to bottom, left to right, only examining the upper diagonal. Note that it doesn't make sense for  $j$  to be smaller than  $i$ , so that's why we only need to deal with half of the matrix. Set  $A[i][j]$  to true only if  $A[i + 1][j - 1]$  is true and  $A[i] == A[j]$ .
- Keep track of the longest palindromic substring we've seen so far.

Let's go through an example with the word "bananas".

**b a n a n a s**

b	t	f	f	f	f	f	f
a		t	f	t	f	t	f
n			t	f	t	f	f
a				t	f	f	f
n					t	f	f
a						t	f
s							t

We can see from this table that the longest palindromic substring here is "ananas", since  $A[1:5]$  is the longest substring that's true in the table.

```
def longest_palindrome(s):
    if not s:
        return ''

    longest = s[0]
    A = [[None for _ in range(len(s))] for _ in range(len(s))]

    # Set all substrings of length 1 to be true
    for i in range(len(s)):
        A[i][i] = True

    # Try all substrings of length 2
    for i in range(len(s) - 1):
        if s[i] == s[i + 1]:
            A[i][i + 1] = True
        else:
            A[i][i + 1] = False

    # Try all substrings of length 3
    for i in range(len(s) - 2):
        if s[i] == s[i + 2] and A[i + 1][i + 1]:
            A[i][i + 2] = True
        else:
            A[i][i + 2] = False

    # Try all substrings of length 4
    for i in range(len(s) - 3):
        if s[i] == s[i + 3] and A[i + 1][i + 2]:
            A[i][i + 3] = True
        else:
            A[i][i + 3] = False

    # Try all substrings of length 5
    for i in range(len(s) - 4):
        if s[i] == s[i + 4] and A[i + 1][i + 3]:
            A[i][i + 4] = True
        else:
            A[i][i + 4] = False

    # Try all substrings of length 6
    for i in range(len(s) - 5):
        if s[i] == s[i + 5] and A[i + 1][i + 4]:
            A[i][i + 5] = True
        else:
            A[i][i + 5] = False

    # Try all substrings of length 7
    for i in range(len(s) - 6):
        if s[i] == s[i + 6] and A[i + 1][i + 5]:
            A[i][i + 6] = True
        else:
            A[i][i + 6] = False

    return longest
```

```
A[i][i + 1] = s[i] == s[i + 1]

i, k = 0, 3
while k <= len(s):
    while i < (len(s) - k + 1) :
        j = i + k - 1
        A[i][j] = A[i + 1][j - 1] and s[i] == s[j]
        # Update longest if necessary
        if A[i][j] and len(s[i:j + 1]) > len(longest):
            longest = s[i:j + 1]
        i += 1
    k += 1
    i = 0
return longest
```

This runs in  $O(N^2)$  time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #47

## Problem

This problem was asked by Facebook.

Given a array of numbers representing the stock prices of a company in chronological order, write a function that calculates the maximum profit you could have made from buying and selling that stock once. You must buy before you can sell it.

For example, given [9, 11, 8, 5, 7, 10], you should return 5, since you could buy the stock at 5 dollars and sell it at 10 dollars.

## Solution

The brute force solution here is to iterate over our list of stock prices, and for each price, find the largest profit you can make from selling that stock price (with the formula future - current), and keep track of the largest profit we can find. That would look like this:

```
def buy_and_sell(arr):
    max_profit = 0
    for i in range(len(arr) - 1):
        for j in range(i, len(arr)):
            buy_price, sell_price = arr[i], arr[j]
            max_profit = max(max_profit, sell_price - buy_price)
    return max_profit
```

This would take  $O(N^2)$ . Can we speed this up?

The maximum profit comes from the greatest difference between the highest price and lowest price, where the higher price must come after the lower one. But if we see a high price x and then a higher price y afterwards, then we can always discard x. So, if we keep track of the highest price in the future for each variable, we can immediately find how much profit buying at that price can make.

That means we can look at the array backwards and always keep track of the highest price we've

seen so far. Then, at each step, we can look at the current price and check how much profit we would have made buying at that price by comparing with our maximum price in the future. Then we only need to make one pass!

```
def buy_and_sell(arr):
    current_max, max_profit = 0, 0
    for price in reversed(arr):
        current_max = max(current_max, price)
        potential_profit = current_max - price
        max_profit = max(max_profit, potential_profit)
    return max_profit
```

This runs in  $O(N)$  time and  $O(1)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #49

## Problem

This problem was asked by Amazon.

Given an array of numbers, find the maximum sum of any contiguous subarray of the array.

For example, given the array [34, -50, 42, 14, -5, 86], the maximum sum would be 137, since we would take elements 42, 14, -5, and 86.

Given the array [-5, -1, -8, -9], the maximum sum would be 0, since we would not take any elements.

Do this in O(N) time.

## Solution

The brute force approach here would be to iterate over every contiguous subarray and calculate its sum, keeping track of the largest one seen.

```
def max_subarray_sum(arr):
    current_max = 0
    for i in range(len(arr) - 1):
        for j in range(i, len(arr)):
            current_max = max(current_max, sum(arr[i:j]))
    return current_max
```

This would run in  $O(N^3)$  time. How can we make this faster?

We can work backwards from our desired solution by iterating over the array and looking at the maximum possible subarray that can be made ending at each index. At each index, either we can include that element in our sum or we exclude it.

We can then keep track of the maximum subarray we've seen so far in a variable `max_so_far`, compute the maximum subarray that includes `x` at each iteration, and choose whichever one is bigger.

```
def max_subarray_sum(arr):
    max_ending_here = max_so_far = 0
    for x in arr:
        max_ending_here = max(x, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

This algorithm is known as Kadane's algorithm, and it runs in  $O(N)$  time and  $O(1)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #50

## Problem

This problem was asked by Microsoft.

Suppose an arithmetic expression is given as a binary tree. Each leaf is an integer and each internal node is one of '+', '-', '\*', or '/'.

Given the root to such a tree, write a function to evaluate it.

For example, given the following tree:

```
*  
 / \  
+   +  
/ \ / \  
3 2 4 5
```

You should return 45, as it is  $(3 + 2) * (4 + 5)$ .

## Solution

This problem should be straightforward from the definition. It will be recursive. We check the value of the root node. If it's one of our arithmetic operators, then we take the evaluated value of our node's children and apply the operator on them.

If it's not an arithmetic operator, it has to be a raw number, so we can just return that.

```
class Node:  
    def __init__(self, val, left=None, right=None):  
        self.val = val  
        self.left = left  
        self.right = right  
  
    PLUS = "+"  
    MINUS = "-"
```

```
TIMES = "*"
DIVIDE = "/"
def evaluate(root):
    if root.val == PLUS:
        return evaluate(root.left) + evaluate(root.right)
    elif root.val == MINUS:
        return evaluate(root.left) - evaluate(root.right)
    elif root.val == TIMES:
        return evaluate(root.left) * evaluate(root.right)
    elif root.val == DIVIDE:
        return evaluate(root.left) / evaluate(root.right)
    else:
        return root.val
```

This runs in O(N) time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #52

## Problem

This problem was asked by Google.

Implement an LRU (Least Recently Used) cache. It should be able to be initialized with a cache size  $n$ , and contain the following methods:

- `set(key, value)`: sets key to value. If there are already  $n$  items in the cache and we are adding a new item, then it should also remove the least recently used item.
- `get(key)`: gets the value at key. If no such key exists, return null.

Each operation should run in  $O(1)$  time.

## Solution

To implement both these methods in constant time, we'll need to use a hash table along with a linked list. The hash table will map keys to nodes in the linked list, and the linked list will be ordered from least recently used to most recently used. Then, for set:

- First look at our current capacity. If it's  $< n$ , then create a node with the val, set it as the head, and add it as an entry in the dictionary.
- If it's equal to  $n$ , then add our node as usual, but also evict the least frequently used node by deleting the tail of our linked list and also removing the entry from our dictionary. We'll need to keep track of the key in each node so that we know which entry to evict.

For get:

- If the key doesn't exist in our dictionary, then return null.
- Otherwise, look up the relevant node through the dictionary. Before returning it, update the

linked list by moving the node to the front of the list.

To help us out, we can use the following tricks:

- Using dummy nodes for the head and tail of our list, which will simplify creating the list when nothing's initialized.
- Implementing the helper class `LinkedList` to reuse code when adding and removing nodes to our linked list.
- When we need to bump a node to the back of list (like when we fetch it), we can just remove it and readd it.

In the end, the code would look like this:

```
class Node:  
    def __init__(self, key, val):  
        self.key = key  
        self.val = val  
        self.prev = None  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        # dummy nodes  
        self.head = Node(None, 'head')  
        self.tail = Node(None, 'tail')  
        # set up head <-> tail  
        self.head.next = self.tail  
        self.tail.prev = self.head  
  
    def get_head(self):  
        return self.head.next  
  
    def get_tail(self):  
        return self.tail.prev  
  
    def add(self, node):  
        prev = self.tail.prev  
        prev.next = node  
        node.prev = prev  
        node.next = self.tail  
        self.tail.prev = node
```

```

def remove(self, node):
    prev = node.prev
    nxt = node.next
    prev.next = nxt
    nxt.prev = prev

class LRUCache:

    def __init__(self, n):
        self.n = n
        self.dict = {}
        self.list = LinkedList()

    def set(self, key, val):
        if key in self.dict:
            self.dict[key].delete()
        n = Node(key, val)
        self.list.add(n)
        self.dict[key] = n
        if len(self.dict) > self.n:
            head = self.list.get_head()
            self.list.remove(head)
            del self.dict[head.key]

    def get(self, key):
        if key in self.dict:
            n = self.dict[key]
            # bump to the back of the list by removing and adding the node
            self.list.remove(n)
            self.list.add(n)
            return n.val

```

All operations run in O(1) time.

# Daily Coding Problem #53

## Problem

This problem was asked by Apple.

Implement a queue using two stacks. Recall that a queue is a FIFO (first-in, first-out) data structure with the following methods: enqueue, which inserts an element into the queue, and dequeue, which removes it.

## Solution

We can implement this by noticing that while a stack is LIFO (last in first out), if we empty a stack one-by-one into another stack, and then pop from the other stack we can simulate a FIFO (first in first out) list.

Consider enqueueing three elements: 1, 2, and then 3:

```
stack1: [1, 2, 3]
stack2: []
```

Then emptying stack1 into stack2:

```
stack1: []
stack2: [3, 2, 1]
```

Then dequeuing three times:

```
1
2
3
```

Which retains the original order. So when enqueueing, we can simply push to our first stack. When dequeuing, we'll first check our second stack to see if any residual elements are there from a previous emptying, and if not, we'll empty all of stack one into stack two immediately so that the order of elements is correct (we shouldn't empty some elements into stack two, pop only some of

them, and then empty some more, for example).

```
class Queue:

    def __init__(self):
        self.s1 = []
        self.s2 = []

    def enqueue(self, val):
        self.s1.append(val)

    def dequeue(self):
        if self.s2:
            return self.s2.pop()
        if self.s1:
            # empty all of s1 into s2
            while self.s1:
                self.s2.append(self.s1.pop())
            return self.s2.pop()
        return None
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #55

## Problem

This problem was asked by Microsoft.

Implement a URL shortener with the following methods:

- `shorten(url)`, which shortens the url into a six-character alphanumeric string, such as `zLg6wl`.
- `restore(short)`, which expands the shortened string into the original url. If no such shortened string exists, return null.

Hint: What if we enter the same URL twice?

## Solution

Clearly, we need a random string generator for this problem. If you're in an interview and you don't know how to generate a random string by heart, that's fine -- you can just assume you have access to a function that generate N random characters. In this case, we'll create a helper function called `_generate_short` that does it for us.

The idea for this problem is to generate a shortened url and store it in a dictionary where the shortened url is the key and the actual url is the value. Then, when retrieving the actual url we can just look it up in the dictionary.

However, we need to be careful in that we don't accidentally overwrite an existing entry when shortening a url. So what we'll do is continuously generate urls until we find one that doesn't already exist, and then use that one. We do that in the helper function `generate_unused_short`.

```
import random
import string

class URLShortener:
    def __init__(self):
```

```

self.short_to_url = {}

def _generate_short(self):
    return ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(6))

def _generate_unused_short(self):
    t = self._generate_short()
    while t in self.short_to_url:
        t = self._generate_short()
    return t

def shorten(self, url):
    short = self._generate_unused_short()
    self.short_to_url[short] = url
    return short

def restore(self, short):
    return self.short_to_url.get(short, None)

```

We can improve this a bit. What if we shorten the same url twice? We could potentially re-use the existing shortened url, but we don't know how to access it without querying all values in our dict!

So we can create a second dict that maps urls to shortened urls and update that appropriately. When we see a url we've seen before, we can just then just re-use that shortened url.

```

import random
import string

class URLShortener:
    def __init__(self):
        self.short_to_url = {}
        self.url_to_short = {}

    def _generate_short(self):
        return ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(6))

    def _generate_unused_short(self):
        t = self._generate_short()
        while t in self.short_to_url:
            t = self._generate_short()
        return t

    def shorten(self, url):
        short = self._generate_unused_short()
        if url in self.url_to_short:
            return self.url_to_short[url]
        self.url_to_short[url] = short
        self.short_to_url[short] = url
        return short

```

```
self.short_to_url[short] = url  
self.url_to_short[url] = short  
return short  
  
def restore(self, short):  
    return self.short_to_url.get(short, None)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #57

## Problem

This problem was asked by Amazon.

Given a string  $s$  and an integer  $k$ , break up the string into multiple texts such that each text has a length of  $k$  or less. You must break it up so that words don't break across lines. If there's no way to break the text up, then return null.

You can assume that there are no spaces at the ends of the string and that there is exactly one space between each word.

For example, given the string "the quick brown fox jumps over the lazy dog" and  $k = 10$ , you should return: ["the quick", "brown fox", "jumps over", "the lazy", "dog"]. No string in the list has a length of more than 10.

## Solution

We can break up the string greedily. First we'll break up  $s$  into an array  $\text{words}$ . Then, we can use a buffer as a current string and tentatively add words to it, checking that the newly-added-to line can fit within  $k$ . If we overflow with the new word, then we flush out the current string into an array  $\text{all}$  and restart it with the new word.

Notice that if any word is longer than  $k$ , then there's no way to break up the text, so we should return None. It's helpful to define a helper function that returns the length of a list of words with spaces added in between.

Finally, we return  $\text{all}$ , which should contain the texts we want.

```
def break(s, k):
    words = s.split()

    if not words:
        return []

    all = []
    current = ''
    for word in words:
        if len(current) + len(word) + 1 > k:
            all.append(current)
            current = word
        else:
            current += ' ' + word
    all.append(current)

    return all
```

```
current = []
all = []

for i, word in enumerate(words):
    if length(current + [word]) <= k:
        current.append(word)
    elif length([word]) > k:
        return None
    else:
        all.append(current)
        current = [word]
all.append(current)

return all

def length(words):
    if not words:
        return 0
    return sum(len(word) for word in words) + (len(words) - 1)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #58

## Problem

This problem was asked by Amazon.

An sorted array of integers was rotated an unknown number of times.

Given such an array, find the index of the element in the array in faster than linear time. If the element doesn't exist in the array, return null.

For example, given the array [13, 18, 25, 2, 8, 10] and the element 8, return 4 (the index of 8 in the array).

You can assume all the integers in the array are unique.

## Solution

We can obviously do this problem in linear time if we looked at each element in the array. However, we need to do it faster than linear time. A big clue should be that the array of integers was previously sorted, and then rotated. If it was just sorted, we could do a binary search. However, this array was also rotated, so we can't do a regular binary search. We can modify it slightly to get to where we want it, however.

In our solution, we first find the rotation point using binary search. We do this by:

- Checking the midpoint for the rotation point (by comparing it to the previous number and seeing if it's larger)
- Moving our check up or down the array:
  - If the number we're looking at is larger than the first item in the array, then the rotation must occur later, so add dist
  - If not, then it must occur before, so subtract dist
- And then update dist by dividing it by 2 and taking its floor (so it's proper binary search).

Then, once we have the rotation point, we can do binary search as usual by remembering to offset the correct amount.

The code would look like this:

```
def shifted_array_search(lst, num):
    # First, find where the breaking point is in the shifted array
    i = len(lst) // 2
    dist = i // 2
    while True:
        if lst[0] > lst[i] and lst[i - 1] > lst[i]:
            break
        elif dist == 0:
            break
        elif lst[0] <= lst[i]:
            i = i + dist
        elif lst[i - 1] <= lst[i]:
            i = i - dist
        else:
            break
        dist = dist // 2

    # Now that we have the bottom, we can do binary search as usual,
    # wrapping around the rotation.
    low = i
    high = i - 1
    dist = len(lst) // 2
    while True:
        if dist == 0:
            return None

        guess_ind = (low + dist) % len(lst)
        guess = lst[guess_ind]
        if guess == num:
            return guess_ind

        if guess < num:
            low = (low + dist) % len(lst)
        if guess > num:
            high = (len(lst) + high - dist) % len(lst)

        dist = dist // 2
```

This solution runs in  $O(\log n)$ . However, this is definitely not the only solution! There are many other possible ways to implement this, but as long as you have the idea of doing binary search, you've got it.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #62

## Problem

This problem was asked by Facebook.

There is an N by M matrix of zeroes. Given N and M, write a function to count the number of ways of starting at the top-left corner and getting to the bottom-right corner. You can only move right or down.

For example, given a 2 by 2 matrix, you should return 2, since there are two ways to get to the bottom-right:

- Right, then down
- Down, then right

Given a 5 by 5 matrix, there are 70 ways to get to the bottom-right.

## Solution

Notice that, to get to any cell, we only have two ways: either directly from above, or from the left, unless we can't go up or left anymore, in which case there's only one way. This leads to the following recurrence:

- If either N or M is 1, then return 1
- Otherwise,  $f(n, m) = f(n - 1, m) + f(n, m - 1)$

This is very similar to the staircase problem from Daily Coding Problem #12.

The recursive solution would look like this:

```
def num_ways(n, m):  
    if n == 1 or m == 1:  
        return 1  
    return num_ways(n - 1, m) + num_ways(n, m - 1)
```

However, just like in the staircase problem (or fibonacci), we will have a lot of repeated subcomputations. So, let's use bottom-up dynamic programming to store those results.

We'll initialize an N by M matrix A, and each entry  $A[i][j]$ , will contain the number of ways we can get to that entry from the top-left. Then, once we've filled up the matrix using our recurrence (by checking directly above or directly left), we can just look at the bottom-right value to get our answer.

```
def num_ways(n, m):
    A = [[0 for _ in range(m)] for _ in range(n)]
    for i in range(n):
        A[i][0] = 1
    for j in range(m):
        A[0][j] = 1
    for i in range(1, n):
        for j in range(1, m):
            A[i][j] = A[i - 1][j] + A[i][j - 1]
    return A[-1][-1]
```

This runs in  $O(N * M)$  time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #63

## Problem

This problem was asked by Microsoft.

Given a 2D matrix of characters and a target word, write a function that returns whether the word can be found in the matrix by going left-to-right, or up-to-down.

For example, given the following matrix:

```
[[ 'F', 'A', 'C', 'I'],
 ['O', 'B', 'Q', 'P'],
 ['A', 'N', 'O', 'B'],
 ['M', 'A', 'S', 'S']]
```

and the target word 'FOAM', you should return true, since it's the leftmost column. Similarly, given the target word 'MASS', you should return true, since it's the last row.

## Solution

This problem should be quite straightforward: we can go through each entry in the array, try to create the word going right and down, and check if the word matches our word. To make bounds checking simple, we'll just try to grab everything from where we're looking at till the end, and then use the slice operator to just get what we want.

```
def build_word_right(matrix, r, c, length):
    return ''.join([matrix[r][i] for i in range(c, len(matrix[0]))])[:length]

def build_word_down(matrix, r, c, length):
    return ''.join([matrix[i][c] for i in range(r, len(matrix))])[:length]

def word_search(matrix, word):
    for r in range(len(matrix)):
        for c in range(len(matrix[0])):
            word_right = build_word_right(matrix, r, c, len(word))
            word_down = build_word_down(matrix, r, c, len(word))
            if word == word_right or word == word_down:
                return True
    return False
```

```

word_down = build_word_down(matrix, r, c, len(word))
if word in (word_right, word_down):
    return True
return False

```

However, if the matrix was really big, then we would be grabbing the whole row or column just to shorten it. We can improve our `build_word_right` and `build_word_down` functions a bit by just taking what we need, which is whichever is shorter between the length of the word and the end of the row or column:

```

def build_word_right(matrix, r, c, length):
    row_len = len(matrix[0])
    return ''.join([matrix[r][i] for i in range(c, min(row_len, length))])

def build_word_down(matrix, r, c, length):
    col_len = len(matrix)
    return ''.join([matrix[i][c] for i in range(r, min(col_len, length))])

```

However, let's say our word were both really big. If we notice, when we're checking the current row or column, that the first few letters are off, then we can quit the search early.

The Python built-in function `zip` is very handy:

```

def check_word_right(matrix, r, c, word):
    word_len = len(word)
    row_len = len(matrix[0])
    if word_len != row_len - c:
        return False
    for c1, c2 in zip(word, (matrix[r][i] for i in range(c, row_len))):
        if c1 != c2:
            return False
    return True

def check_word_down(matrix, r, c, word):
    word_len = len(word)
    col_len = len(matrix)
    if word_len != col_len - r:
        return False
    for c1, c2 in zip(word, (matrix[i][c] for i in range(r, col_len))):
        if c1 != c2:
            return False
    return True

    return ''.join([matrix[i][c] for i in range(r, min(col_len, length))])

def word_search(matrix, word):

```

```
for r, row in enumerate(matrix):
    for c, val in enumerate(row):
        if check_word_right(matrix, r, c, word):
            return True
        if check_word_down(matrix, r, c, word):
            return True
return False
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #65

## Problem

This problem was asked by Amazon.

Given a N by M matrix of numbers, print out the matrix in a clockwise spiral.

For example, given the following matrix:

```
[[1,  2,  3,  4,  5],  
 [6,  7,  8,  9,  10],  
 [11, 12, 13, 14, 15],  
 [16, 17, 18, 19, 20]]
```

You should print out the following:

```
1  
2  
3  
4  
5  
10  
15  
20  
19  
18  
17  
16  
11  
6  
7  
8  
9  
14  
13  
12
```

# Solution

As you might imagine, there are many possible solutions for this problem. Ours involves keeping track of our current position and direction. As we move along and print each value, we set it to None. Then once we've either hit the edge or another None value (indicating we've seen it before), we change directions counterclockwise and keep on going.

We use an enum to define the directions, and some helper functions `next_direction`, `next_position`, and `should_change_direction` to help us lay out the code cleanly.

```
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

DIRECTIONS = [RIGHT, DOWN, LEFT, UP]

def next_direction(direction):
    if direction == RIGHT:
        return DOWN
    elif direction == DOWN:
        return LEFT
    elif direction == LEFT:
        return UP
    elif direction == UP:
        return RIGHT

def next_position(position, direction):
    if direction == RIGHT:
        return (position[0], position[1] + 1)
    elif direction == DOWN:
        return (position[0] + 1, position[1])
    elif direction == LEFT:
        return (position[0], position[1] - 1)
    elif direction == UP:
        return (position[0] - 1, position[1])

def should_change_direction(M, r, c):
    in_bounds_r = 0 <= r < len(M)
    in_bounds_c = 0 <= c < len(M[0])
    return not in_bounds_r or not in_bounds_c or M[r][c] is None

def matrix_spiral_print(M):
    remaining = len(M) * len(M[0])
```

```
current_direction = RIGHT
current_position = (0, 0)
while remaining > 0:
    r, c = current_position
    print(M[r][c])
    M[r][c] = None
    remaining -= 1

    possible_next_position = next_position(current_position, current_direction)
    if should_change_direction(M, possible_next_position[0], possible_next_position[1]):
        current_direction = next_direction(current_direction)
        current_position = next_position(current_position, current_direction)
    else:
        current_position = possible_next_position
```

This takes  $O(M * N)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #67

## Problem

This problem was asked by Google.

Implement an LFU (Least Frequently Used) cache. It should be able to be initialized with a cache size  $n$ , and contain the following methods:

- `set(key, value)`: sets key to value. If there are already  $n$  items in the cache and we are adding a new item, then it should also remove the least frequently used item. If there is a tie, then the least recently used key should be removed.
- `get(key)`: gets the value at key. If no such key exists, return null.

Each operation should run in  $O(1)$  time.

## Solution

This problem is similar to the LRU cache problem (Problem #52), but requires a different perspective. In that problem, we used a doubly linked list of nodes and a hash table that mapped keys to the nodes. When we evicted from the cache, we just had to look at the head of the linked list.

In this solution, we keep two dictionaries: one mapping from keys to values (and their frequencies), and another mapping from frequency counts to a deque of keys.

When we set a key, we first check if we need to evict another key. If we do, then we'll look at the entry in our frequency map with the lowest frequency and pop from the left (since we'll be appending, the left will be the least recently used entry). Then we can add our mapping to the dicts: we'll add our key and value (along with a frequency of one) to our value mapping, and also to our frequency mapping at key 1.

If we're updating a key (the key already exists), then it's a different story. Here, we will need to basically only update the value mapping by setting a new value and increment the frequency. For the frequency mapping, we'll need to move our key to the next frequency bucket, creating it if necessary via `defaultdict`.

Getting a key has similar logic to updating it, without actually updating it.

```
from collections import defaultdict
from collections import deque

class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.val_map = {}
        self.freq_map = defaultdict(deque)
        self.min_freq = 0

    def get(self, key):
        # If key doesn't exist, return None.
        if key not in self.val_map:
            return None

        # First, we look up the val and frequency in our val_map.
        val, freq = self.val_map[key]

        # We need to then increment the frequency of our key,
        # and move it to the next frequency bucket.
        self.freq_map[freq].remove(key)
        if not self.freq_map[freq]:
            del self.freq_map[freq]
        self.freq_map[freq + 1].append(key)
        self.min_freq = min(self.freq_map.keys())
        return val

    def set(self, key, value):
        if self.capacity == 0:
            return

        if key in self.val_map:
            freq = self.freq_map[self.val_map[key][1]]
            freq.remove(key)
            if not freq:
                del self.freq_map[self.val_map[key][1]]
            self.freq_map[self.val_map[key][1] + 1].append(key)
            self.min_freq = min(self.freq_map.keys())
            self.val_map[key] = (value, self.freq_map[self.val_map[key][1]] + 1)
        else:
            if len(self.val_map) == self.capacity:
                # Remove the least frequently used item.
                freq = self.freq_map[min(self.freq_map.keys())]
                freq.pop(0)
                if not freq:
                    del self.freq_map[min(self.freq_map.keys())]
            self.val_map[key] = (value, 1)
            self.freq_map[1].append(key)
            self.min_freq = 1
```

```

# so we'll take it out of the current bucket and put it
# into the next frequency's bucket. If it was the last thing
# in the current bucket and the lowest frequency, (e.g. 1 to 2),
# then we'll make sure to update our min_freq so we can keep
# track of what to evict.

self.freq_map[freq].remove(key)

if not self.freq_map[freq]:
    del self.freq_map[freq]
    if self.min_freq == freq:
        self.min_freq += 1

# Update our dicts as usual.

self.val_map[key] = (val, freq + 1)
self.freq_map[freq + 1].append(key)
return val

def set(self, key, val):
    if self.capacity == 0:
        return

    if key not in self.val_map:
        # Evict the least frequently used key by popping left
        # from the lowest-frequency key, since it's ordered by
        # time (because we use append).

        if len(self.val_map) >= self.capacity:
            to_evict = self.freq_map[self.min_freq].popleft()
            if not self.freq_map[self.min_freq]:
                del self.freq_map[self.min_freq]
            del self.val_map[to_evict]

        # Add our key to val_map and freq_map
        self.val_map[key] = (val, 1)
        self.freq_map[1].append(key)
        self.min_freq = 1
    else:
        # Update the entry and increase the frequency of the key,
        # updating the minimum frequency if necessary.

        _, freq = self.val_map[key]
        self.freq_map[freq].remove(key)
        if not self.freq_map[freq]:
            if freq == self.min_freq:
                self.min_freq += 1
            del self.freq_map[freq]
        self.val_map[key] = (val, freq + 1)
        self.freq_map[freq + 1].append(key)

```

These operations should run in O(1) time.

# Daily Coding Problem #69

## Problem

This problem was asked by Facebook.

Given a list of integers, return the largest product that can be made by multiplying any three integers.

For example, if the list is [-10, -10, 5, 2], we should return 500, since that's  $-10 * -10 * 5$ .

You can assume the list has at least three integers.

## Solution

If all the integers were positive, then we would simply need to take the three largest numbers of the array. Then, we can just sort it and return the last three elements.

However, we need to account for negative numbers in the array. If the largest product that can be made includes a negative number, we would need to have two so as to cancel out the negatives. So, we can take the larger of:

- The three largest numbers
- The two smallest (most negative) numbers, and the largest number

```
def maximum_product_of_three(lst):  
    lst.sort()  
    third_largest, second_largest, first_largest = lst[-3], lst[-2], lst[-1]  
    first_smallest, second_smallest = lst[0], lst[1]  
    return max(third_largest * second_largest * first_largest,  
               first_largest * first_smallest * second_smallest)
```

This runs in  $O(N \log N)$  time since we have to sort the input array.

We could also do this in  $O(N)$  time by using select or looking for the largest elements manually.

```
from math import inf
```

```
def maximum_product_of_three(lst):
    max1, max2, max3, min1, min2 = -inf, -inf, -inf, inf, inf

    for x in lst:
        if x > max1:
            max3 = max2
            max2 = max1
            max1 = x
        elif x > max2:
            max3 = max2
            max2 = x
        elif x > max3:
            max3 = x

        if x < min1:
            min2 = min1
            min1 = x
        elif x < min2:
            min2 = x

    return max(max1 * max2 * max3, max1 * min1 * min2)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #73

## Problem

This problem was asked by Google.

Given the head of a singly linked list, reverse it in-place.

## Solution

We can do this recursively and cleverly, using Python's default argument feature. Basically, we call `reverse` on the node's next, but not before cleaning up some pointers first:

```
def reverse(head, prev=None):
    if not head:
        return prev
    tmp = head.next
    head.next = prev
    return reverse(tmp, head)
```

This runs in  $O(N)$  time. But it also runs in  $O(N)$  space, since Python doesn't do [tail-recursion elimination](#).

We can improve the space by doing this iteratively, and keeping track of two things: a `prev` pointer and a `current` pointer. The `current` pointer will iterate over through the list and the `prev` pointer will follow, one node behind. Then, as we move along the list, we'll fix up the `current` node's `next` to point to the previous node. Then we update `prev` and `current`.

```
def reverse(head):
    prev, current = None, head
    while current is not None:
        tmp = current.next
        current.next = prev
        prev = current
        current = tmp
```

```
prev = current  
current = tmp  
return prev
```

Now this only uses constant space!

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #75

## Problem

This problem was asked by Microsoft.

Given an array of numbers, find the length of the longest increasing subsequence in the array. The subsequence does not necessarily have to be contiguous.

For example, given the array [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15], the longest increasing subsequence has length 6: it is 0, 2, 6, 9, 11, 15.

## Solution

This naive, brute force way to solve this is to generate each possible subsequence, testing each one for monotonicity and keeping track of the longest one. That would be prohibitively expensive: generating each subsequence already takes  $O(2^N)$ !

Instead, let's try to tackle this problem using recursion and then optimize it with dynamic programming.

Assume that we already have a function that gives us the length of the longest increasing subsequence. Then we'll try to feed some part of our input array back to it and try to extend the result. Our base cases are: the empty list, returning 0, and an array with one element, returning 1.

Then,

- For every index  $i$  up until the second to last element, calculate `longest_increasing_subsequence` up to there.
- We can only extend the result with the last element if our last element is greater than `arr[i]` (since otherwise, it's not increasing).
- Keep track of the largest result.

```
def longest_increasing_subsequence(arr):
    if not arr:
        return 0
```

```
if len(arr) == 1:
    return 1

max_ending_here = 0
for i in range(len(arr)):
    ending_at_i = longest_increasing_subsequence(arr[:i])
    if arr[-1] > arr[i - 1] and ending_at_i + 1 > max_ending_here:
        max_ending_here = ending_at_i + 1
return max_ending_here
```

This is really slow due to repeated subcomputations (exponential in time). So, let's use dynamic programming to store values to recompute them for later.

We'll keep an array A of length N, and A[i] will contain the length of the longest increasing subsequence ending at i. We can then use the same recurrence but look it up in the array instead:

```
def longest_increasing_subsequence(arr):
    if not arr:
        return 0
    cache = [1] * len(arr)
    for i in range(1, len(arr)):
        for j in range(i):
            if arr[i] > arr[j]:
                cache[i] = max(cache[i], cache[j] + 1)
    return max(cache)
```

This now runs in  $O(N^2)$  time and  $O(N)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #77

## Problem

This problem was asked by Snapchat.

Given a list of possibly overlapping intervals, return a new list of intervals where all overlapping intervals have been merged.

The input list is not necessarily ordered in any way.

For example, given [(1, 3), (5, 8), (4, 10), (20, 25)], you should return [(1, 3), (4, 10), (20, 25)].

## Solution

We can do this by sorting all the intervals by their start time. This way, when looking at the current interval, if it overlaps with the previous one we can just combine them.

```
def merge(intervals):
    result = []
    for start, end in sorted(intervals, key=lambda i: i[0]):
        # If current interval overlaps with the previous one, combine them
        if result and start <= result[-1][1]:
            prev_start, prev_end = result[-1]
            result[-1] = (prev_start, max(end, prev_end))
        else:
            result.append((start, end))
    return result
```

Since we have to sort the intervals, this runs in  $O(N \log N)$  time.

[Privacy Policy](#)  
[Terms of Service](#)

[Press](#)

# Daily Coding Problem #78

## Problem

This problem was asked by Google.

Given k sorted singly linked lists, write a function to merge all the lists into one sorted singly linked list.

## Solution

A brute force solution here might be to gather all the values of the linked lists into one large array, sort the array, and then recreate a linked list with the values from the array. That would look like this:

```
def merge(lists):
    # Combine all nodes into an array
    arr = []
    for head in lists:
        current = head
        while current:
            arr.append(current.val)
            current = current.next

    new_head = current = Node(-1) # dummy head
    for val in sorted(arr):
        current.next = Node(val)
        current = current.next

    return new_head.next
```

This would take  $O(KN \log KN)$  time and  $O(KN)$  space, where K is the number of lists and N is the number of elements in the largest list.

A better way would be to take advantage of the inherent sortedness of the input lists. We can keep track, using pointers, of where we are at each list, and pick the minimum of all the pointers. Once we've picked one, we can move that pointer up. This would run in  $O(KN * K)$  time and  $O(K)$  space.

```
def merge(lists):
    new_head = current = Node(-1)
    while all(lst is not None for lst in lists):
        # Get min of all non-None lists
        current_min, i = min((lst.val, i) for i, lst in enumerate(lists) if lst is not None)
        lists[i] = lists[i].next
        current.next = Node(current_min)
        current = current.next
    return new_head.next
```

An even faster way would be to use a heap to keep track of all the pointers instead. Then we can do this in  $O(KN * \log K)$  time, since we'll be using the heapsort ordering to figure out the min in  $O(\log K)$  time instead of  $O(K)$  time.

```
def merge(lists):
    new_head = current = Node(-1)
    heap = [(lst.val, i) for i, lst in enumerate(lists)]
    heapq.heapify(heap)
    while heap:
        current_min, i = heapq.heappop(heap)
        # Add next min to merged linked list.
        current.next = Node(current_min)
        current = current.next
        # Add next value to heap.
        if lists[i] is not None:
            heapq.heappush(heap, (lists[i].val, i))
            lists[i] = lists[i].next
    return new_head.next
```

# Daily Coding Problem #79

## Problem

This problem was asked by Facebook.

Given an array of integers, write a function to determine whether the array could become non-decreasing by modifying at most 1 element.

For example, given the array [10, 5, 7], you should return true, since we can modify the 10 into a 1 to make the array non-decreasing.

Given the array [10, 5, 1], you should return false, since we can't modify any one element to get a non-decreasing array.

## Solution

In this problem, we can count each time an element goes down. Then, if it has went down more than twice, we can return False right away. But if count is one, and the element is one that cannot be erased by modifying only one endpoint of that downtick, then we should return False as well.

```
def check(lst):
    count = 0
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            if count > 0:
                return False
            if i - 1 >= 0 and i + 2 < len(lst) and lst[i] > lst[i + 2] and lst[i + 1] < lst[i - 1]:
                return False
            count += 1
    return True
```

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #80

## Problem

This problem was asked by Google.

Given the root of a binary tree, return a deepest node. For example, in the following tree, return d.

```
a  
/\  
b c  
/  
d
```

## Solution

Base case for this question actually can't be null, because it's not a real result that can be combined (null is not a node). Here we should use the leaf node as the base case and return itself.

The recursive step for this problem is a little bit tricky because we can't actually use the results of the left and right subtrees directly. So we need to ask, what other information do we need to solve this question? It turns out if we tag each subresult node their depths, we could get the final solution by picking the higher depth leaf and then incrementing it:

```
def deepest(node):  
    if node and not node.left and not node.right:  
        return (node, 1) # Leaf and its depth  
  
    if not node.left: # Then the deepest node is on the right subtree  
        return increment_depth(deepest(node.right))  
    elif not node.right: # Then the deepest node is on the left subtree  
        return increment_depth(deepest(node.left))  
  
    return increment_depth(  
        max(deepest(node.left), deepest(node.right),
```

```
key=lambda x: x[1])) # Pick higher depth tuple and then increment its depth
```

```
def increment_depth(node_depth_tuple):  
    node, depth = node_depth_tuple  
    return (node, depth + 1)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #81

## Problem

This problem was asked by Yelp.

Given a mapping of digits to letters (as in a phone number), and a digit string, return all possible letters the number could represent. You can assume each valid number in the mapping is a single digit.

For example if {"2": ["a", "b", "c"], 3: ["d", "e", "f"], ...} then "23" should return ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

## Solution

There is a relatively straight forward substructure to this problem.

Let's assume that we knew the result of the function with the same digits except the first character. Then, we could reconstruct the final result by: for each character the first digit maps to, prepend that character to each permutation from the recursive call.

For example, if the digits are '12', and the mapping is {'1': ['a', 'b', 'c'], '2': ['d', 'e', 'f']} then without the first digit, the result would be ['d', 'e', 'f']. If we prepend 'a', 'b', and 'c', to each permutation, we would get 'ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf'.

```
def get_permutations(digits, mapping):
    digit = digits[0]

    if len(digits) == 1:
        return mapping[digit]

    result = []
    for char in mapping[digit]:
        for perm in get_permutations(digits[1:], mapping):
            result.append(char + perm)

    return result
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #83

## Problem

This problem was asked by Google.

Invert a binary tree.

For example, given the following tree:

```
    a
   / \
  b   c
 / \ /
d   e f
```

should become:

```
    a
   / \
  c   b
  \   /
   f e d
```

## Solution

Assuming we could invert the current node's left and right subtrees, all we'd need to do is then switch the left to now become right, and right to become left. The base case is when the node is None and we can just return None for that case. Then we know this works for the leaf node case since switching left and right subtrees doesn't do anything (since they're both None).

```
def invert(node):
    if not node:
        return node
```

```
left = invert(node.left)
right = invert(node.right)

node.left, node.right = right, left
return node
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #84

## Problem

This problem was asked by Amazon.

Given a matrix of 1s and 0s, return the number of "islands" in the matrix. A 1 represents land and 0 represents water, so an island is a group of 1s that are neighboring whose perimeter is surrounded by water.

For example, this matrix has 4 islands.

```
1 0 0 0 0  
0 0 1 1 0  
0 1 1 0 0  
0 0 0 0 0  
1 1 0 0 1  
1 1 0 0 1
```

## Solution

This problem can be solved by keeping track of a `visited` table that keeps track of the land we've visited. Then, every time we see a piece of land that hasn't been visited, we can floodfill explore.

This takes  $O(N)$  (where  $N$  is the number of cells) since each cell is only visited twice: once in our outer for loop and then once in our fill.

```
def num_islands(board):  
    num_rows = len(board)  
    num_cols = len(board[0])  
    count = 0  
  
    visited = [[False for _ in range(num_cols)] for _ in range(num_rows)]  
    for row in range(len(board)):  
        for col in range(len(board[row])):
```

```
if board[row][col] == 1 and not visited[row][col]:
    fill(board, visited, row, col)
    count += 1

return count

def fill(board, visited, row, col):
    moves = [(0, 1),
              (0, -1),
              (1, 0),
              (-1, 0)]
    visited[row][col] = True

    for move_row, move_col in moves:
        new_row, new_col = (row + move_row, col + move_col)
        if (inside_board(board, new_row, new_col) and
            board[new_row][new_col] == 1 and
            not visited[new_row][new_col]):
            fill(board, visited, new_row, new_col)

def inside_board(board, row, col):
    return 0 <= row < len(board) and 0 <= col < len(board[0])
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #85

## Problem

This problem was asked by Facebook.

Given three 32-bit integers  $x$ ,  $y$ , and  $b$ , return  $x$  if  $b$  is 1 and  $y$  if  $b$  is 0, using only mathematical or bit operations. You can assume  $b$  can only be 1 or 0.

## Solution

We can solve this problem by seeing that if we multiply  $x$  with  $b$ , it solves half the problem. Since we want  $y$  to behave in opposite, we can get the same behavior by multiplying  $y$  with  $(1 - b)$ .

Now,  $(x * b)$  gives  $x$  when  $b$  is 1 and 0 otherwise. Similarly,  $(y * (1 - b))$  gives  $y$  when  $b$  is 0 and 0 otherwise. We can just combine the two formulas with either  $a +$  or  $|$ ,

```
def switch(x, y, b):
    return (x * b) | (y * (1 - b))
```

# Daily Coding Problem #86

## Problem

This problem was asked by Google.

Given a string of parentheses, write a function to compute the minimum number of parentheses to be removed to make the string valid (i.e. each open parenthesis is eventually closed).

For example, given the string "()())()", you should return 1. Given the string ")()", you should return 2, since we must remove all of them.

## Solution

For a string to be considered valid, each open parenthesis should eventually be closed. Other parentheses that don't satisfy this condition should be counted as invalid.

Whenever we encounter an unmatched closing parenthesis, we can count it as invalid. After that, we also add the number of unmatched opening parentheses to our invalid count. This runs in  $O(N)$ .

```
def count_invalid_parenthesis(string):
    opened = 0
    invalid = 0
    for c in string:
        if c == '(':
            opened += 1
        elif c == ')':
            if opened > 0:
                opened -= 1
            else:
                invalid += 1
    # Count as invalid all unclosed parenthesis
    invalid += opened
    return invalid
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #89

## Problem

This problem was asked by LinkedIn.

Determine whether a tree is a valid binary search tree.

A binary search tree is a tree with two children, `left` and `right`, and satisfies the constraint that the key in the `left` child must be less than or equal to the root and the key in the `right` child must be greater than or equal to the root.

## Solution

To solve this problem, we need to recall the definition of a binary search tree. Each node of a BST has the following properties:

- A node's left subtree contains only nodes with keys less than the nodes' key.
- A node's right subtree contains only nodes with keys greater than the nodes' key.
- Both the left and right subtrees must be valid BSTs.

From the properties above, we can construct a recursive solution. It's tempting to write a solution which checks whether the left node's key is less than the current node's key and the right node's key is greater than the current node's key. However, we have to make sure that the property holds for the entire subtree, not just the children. For example, the following binary tree would be considered valid if we only checked the children:

```
2
 / \
1   3
 \
4
```

We can iterate through the entire left and right subtrees to determine whether the keys are valid.

However, the work would be doing can be simplified into one recursive method.

Let's call our recursive method `is_bst()`. At each call in our recursive method, we can maintain a range of valid values for the node's keys -- we'll call the lower bound `min_key` and upper bound `max_key`. If the current node's key is *outside* the range of `min_key` to `max_key`, then return `false`. Otherwise, we call the method on the left and right child nodes, returning `true` if both calls return `true`. If a node is `null`, we should return `true`.

When we call `is_bst()` on the children, we limit the range of valid keys based on our current key. If we call `is_bst()` on the *left* node, then `min_key` should remain the same, while `max_key` should be updated to the current node's key. Similarly, if we call `is_bst()` on the *right* node, then `max_key` should remain the same, while `min_key` should be updated to the current node's key.

```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.key = key

    def is_bst(self):
        def is_bst_helper(root, min_key, max_key):
            if root is None:
                return True
            if root.key <= min_key or root.key >= max_key:
                return False
            return is_bst_helper(root.left, min_key, root.key) and \
                   is_bst_helper(root.right, root.key, max_key)

        return is_bst_helper(self, float('-inf'), float('inf'))
```

The time complexity of this solution is  $O(N)$ , as it requires visiting every node in the tree.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #92

## Problem

This problem was asked by Airbnb.

We're given a hashmap associating each courseId key with a list of courseIds values, which represents that the prerequisites of courseId are courseIds. Return a sorted ordering of courses such that we can finish all courses.

Return null if there is no such ordering.

For example, given {'CSC300': ['CSC100', 'CSC200'], 'CSC200': ['CSC100'], 'CSC100': []}, should return ['CSC100', 'CSC200', 'CSC300'].

## Solution

This is a classic topological sorting question. One way to think about this problem is to think about how would you go about solving it manually? We can divide it into these two steps:

1. Put all courses with no pre-requisites into our todo list.
2. For each course C in the todo list, find each course D which have C as a prerequisite and remove C from its list. Add D to the todo list.

If in the end we couldn't take some courses, this means that there was a circular dependency.

```
def courses_to_take(course_to_prereqs):  
    # Copy list values into a set for faster removal.  
    course_to_prereqs = {c: set(p) for c, p in course_to_prereqs.items()}  
  
    todo = [c for c, p in course_to_prereqs.items() if not p]  
  
    # Used to find courses D which have C as a prerequisite  
    prereq_to_courses = {}  
  
    for course in course_to_prereqs:  
        for p in course_to_prereqs[course]:
```

```
for prereq in course_to_prereqs[course]:
    if prereq not in prereq_to_courses:
        prereq_to_courses[prereq] = []

    prereq_to_courses[prereq].append(course)

result = [] # courses we need to take in order

while todo:
    prereq = todo.pop()
    result.append(prereq)

    # Find which courses are now free to take

    for c in prereq_to_courses.get(prereq, []):
        course_to_prereqs[c].remove(prereq)
        if not course_to_prereqs[c]:
            todo.append(c)

# Circular dependency
if len(result) < len(course_to_prereqs):
    return None
return result
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #95

## Problem

This problem was asked by Palantir.

Given a number represented by a list of digits, find the next greater permutation of a number, in terms of lexicographic ordering. If there is not greater permutation possible, return the permutation with the lowest value/ordering.

For example, the list [1, 2, 3] should return [1, 3, 2]. The list [1, 3, 2] should return [2, 1, 3]. The list [3, 2, 1] should return [1, 2, 3].

Can you perform the operation without allocating extra memory (disregarding the input memory)?

## Solution

The brute force approach to this problem would be to generate all permutations of the number/list. Once we have all of the permutations, we choose the one that comes right after our input list, by taking the minimum of their distances. This approach would take  $O(N!)$  time to generate the permutations and find the one that is the closest to our given number.

We can instead observe a pattern in how the next permutation is obtained. First, consider the case where the entire sequence is decreasing (e.g. [3, 2, 1]). Regardless of the values, we cannot generate a permutation where the value is greater. We can see this since, if there were two possible values a and b that could be swapped to obtain a higher value of a, b must be greater than a. Since b comes after a in the sequence and the sequence is decreasing, we have a contradiction. Next, we have to generate the lowest possible value from the sequence. By similar logic, the smallest number we can generate is from an increasing sequence -- or the reverse of the decreasing sequence.

Now, we can break down other problems in terms of this subproblem. We will start from the right side of the list, and try to find the smallest digit that we can swap and obtain a higher number. While the sublist is a decreasing sequence, we cannot get a higher permutation. Instead, we try adding to the sublist by appending the digit to the left. Once we get a digit that is less than the one to the right, we swap it with the smallest digit that is higher than it. Then, we must make sure the remaining digits to the right are in their lowest-value permutation. We can do this by ordering

the remaining digits on the right from least-to-greatest. Since our swap preserves the fact that the right sublist is decreasing, we simply reverse it. For example:

```
[1, 3, 5, 4, 2]
    ^ decreasing sublist
[1, 3, 5, 4, 2]
    ^ ^ decreasing sublist
[1, 3, 5, 4, 2]
    ^     ^ decreasing sublist
[1, 3, 5, 4, 2]
    ^         ^ non-decreasing, so swap with next-highest digit
[1, 4, 5, 3, 2]
    ^         ^ reverse digits to the right, sorting least-to-greatest
[1, 4, 2, 3, 5]
```

```
def nextPermutation(self, nums):
    def swap(nums, a, b):
        # Perform an in-place swap
        nums[a], nums[b] = nums[b], nums[a]

    def reverse(nums, a, b):
        # Reverses elements at index a to b (inclusive) in-place
        nums[a:b+1] = reversed(nums[a:b+1])

    # Find first index where nums[idx] < nums[idx + 1]
    pivot = len(nums) - 2
    while pivot >= 0 and nums[pivot] >= nums[pivot + 1]:
        pivot -= 1

    if pivot >= 0:
        # Find the next-largest number to swap with
        successor = len(nums) - 1
        while (successor > 0 and nums[successor] <= nums[pivot]):
            successor -= 1
        swap(nums, pivot, successor)

    reverse(nums, pivot + 1, len(nums) - 1)
```

# Daily Coding Problem #96

## Problem

This problem was asked by Microsoft.

Given a number in the form of a list of digits, return all possible permutations.

For example, given [1,2,3], return [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]].

## Solution

There are a few ways to do this, and most solutions will have the same run-time. We will need to generate all  $N!$  permutations, so our algorithm will have  $O(N!)$  run time.

The most straightforward method is to use recursion. We can think of the problem in terms of subproblems, where we can generate permutations of a sublist. A permutation of a single digit (e.g. [1]) would return simply the single digit. To get permutations of size  $n$ , we get all permutations of size  $n-1$  and add the next character within each position (index 0 to  $n$ ). For

example, one permutation of the sublist [2,3] is [2,3]. We add 1 to three positions to obtain [1,2,3], [2,1,3], and [2,3,1].

```
def permute(nums):
    if (len(nums) == 1):
        return [nums]

    output = []
    for l in permute(nums[1:]):
        for idx in range(len(nums)):
            output.append(l[:idx] + [nums[0]] + l[idx:])
    return output
```

An alternative way we can formulate the recursion is by generating all permutations of length  $n-1$ , but with all digits allowed. The permutations of size 1 would return the input array (e.g. [[1],[2],[3]]). Then, we append the  $n$ th digit to the front of the permutations.

```
def permute(nums):
    def helper(nums, index, output):
        if index == len(nums) - 1:
            output.append(nums.copy())
        for i in range(index, len(nums)):
            nums[index], nums[i] = nums[i], nums[index]
            helper(nums, index + 1, output)
            nums[index], nums[i] = nums[i], nums[index]

    output = []
    helper(nums, 0, output)
    return output
```

Both solutions run in  $O(N!)$  time and space, where  $N$  is the size of the input list.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #97

## Problem

This problem was asked by Stripe.

Write a map implementation with a get function that lets you retrieve the value of a key at a particular time.

It should contain the following methods:

- `set(key, value, time)`: sets key to value for  $t = \text{time}$ .
- `get(key, time)`: gets the key at  $t = \text{time}$ .

The map should work like this. If we set a key at a particular time, it will maintain that value forever or until it gets set at a later time. In other words, when we get a key at a time, it should return the value that was set for that key set at the most recent time.

Consider the following examples:

```
d.set(1, 1, 0) # set key 1 to value 1 at time 0  
d.set(1, 2, 2) # set key 1 to value 2 at time 2  
d.get(1, 1) # get key 1 at time 1 should be 1  
d.get(1, 3) # get key 1 at time 3 should be 2
```

```
d.set(1, 1, 5) # set key 1 to value 1 at time 5  
d.get(1, 0) # get key 1 at time 0 should be null  
d.get(1, 10) # get key 1 at time 10 should be 1
```

```
d.set(1, 1, 0) # set key 1 to value 1 at time 0  
d.set(1, 2, 0) # set key 1 to value 2 at time 0  
d.get(1, 0) # get key 1 at time 0 should be 2
```

## Solution

One possible way to solve this question is using a map of maps, where each key has its own map of time-value pairs. That would mean something like:

```
{  
    key: {  
        time: value,  
        time: value,  
        ...  
    },  
    key: {  
        time: value,  
        time: value,  
        ...  
    },  
    ...  
}
```

Also, if a particular time does not exist on the time-value map, we must be able to get the value of the nearest previous time (or null if doesn't have one). A sorted map would fit the bill, but python standard library doesn't have one. So, let's see how this map would look:

```
class TimeMap:  
    def __init__(self):  
        self.map = dict()  
        self.sorted_keys_cache = None  
  
    def get(self, key):  
        value = self.map.get(key)  
        if value is not None:  
            return value  
        if self.sorted_keys_cache is None:  
            self.sorted_keys_cache = sorted(self.map.keys())  
        i = bisect.bisect_left(self.sorted_keys_cache, key)  
        if i == 0:  
            return None  
        else:  
            return self.map.get(self.sorted_keys_cache[i - 1])  
  
    def set(self, key, value):  
        self.sorted_keys_cache = None  
        self.map[key] = value
```

This is a map with a list of sorted keys. To find out the nearest previous time we use the binary search algorithm provided by the [bisect](#).

Any write operation on this map wipes the key's cache, causing a full sort of the keys on the next get call, which in python's [TimSort](#) averages as  $O(n \log n)$  complexity.

For mixed workloads, a more suitable approach is to use arrays under the hood. Something like this:

```
class TimeMap:

    def __init__(self):
        self.keys = []
        self.values = []

    def get(self, key):
        if self.keys is None:
            return None
        i = bisect.bisect_left(self.keys, key)
        if len(self.keys) == i:
            return self.values[i - 1]
        elif self.keys[i] == key:
            return self.values[i]
        elif i == 0:
            return None
        else:
            return self.values[i - 1]

    def set(self, key, value):
        i = bisect.bisect_left(self.keys, key)
        if len(self.keys) == i:
            self.keys.append(key)
            self.values.append(value)
        elif self.keys[i] == key:
            self.values[i] = value
        else:
            self.keys.insert(i + 1, key)
            self.values.insert(i + 1, value)
```

In this way, both get and set behave more predictable from the performance standpoint, it's just a binary search, and for set two array reallocations in the worst case.

The last missing part to solve this question is the first level map, which the code would look this:

```
class MultiTimeMap:

    def __init__(self):
        self.map = defaultdict(TimeMap)

    def set(self, key, value, time):
        self.map[key].set(time, value)
```

```
def get(self, key, time):
    time_map = self.map.get(key)
    if time_map is None:
        return None
    else:
        return time_map.get(time)
```

Now each key can have its own TimeMap, initialized by `defaultdict` when needed.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #98

## Problem

This problem was asked by Coursera.

Given a 2D board of characters and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example, given the following board:

```
[  
    ['A', 'B', 'C', 'E'],  
    ['S', 'F', 'C', 'S'],  
    ['A', 'D', 'E', 'E']  
]
```

`exists(board, "ABCED")` returns true, `exists(board, "SEE")` returns true, `exists(board, "ACB")` returns false.

## Solution

We can view the provided board as a graph, where the characters are nodes, whose edges point to adjacent characters. We have the choice of performing a depth-first or breadth-first search. DFS is usually simpler to implement, so we will make that our choice. Also, since we can exit the search once we've reached the length of the word, the recursive depth will be limited by the length.

```
def search(board, row, col, word, index, visited):  
    def is_valid(board, row, col):  
        return row >= 0 and row < len(board) and col >= 0 and col < len(board[0])  
  
        if not is_valid(board, row, col):  
            return False  
        if visited.contains((row, col)):  
            return False  
  
        visited.add((row, col))  
  
        if word[index] != board[row][col]:  
            return False  
  
        if index == len(word) - 1:  
            return True  
  
        return search(board, row + 1, col, word, index + 1, visited) or  
               search(board, row - 1, col, word, index + 1, visited) or  
               search(board, row, col + 1, word, index + 1, visited) or  
               search(board, row, col - 1, word, index + 1, visited)  
  
    if not is_valid(board, row, col):  
        return False  
    if visited.contains((row, col)):  
        return False  
  
    visited.add((row, col))  
  
    if word[index] != board[row][col]:  
        return False  
  
    if index == len(word) - 1:  
        return True  
  
    return search(board, row + 1, col, word, index + 1, visited) or  
           search(board, row - 1, col, word, index + 1, visited) or  
           search(board, row, col + 1, word, index + 1, visited) or  
           search(board, row, col - 1, word, index + 1, visited)
```

```

        return False

    if board[row][col] != word[index]:
        return False

    if index == len(word) - 1:
        return True

    visited.add((row, col))

    for d in ((0, -1), (0, 1), (-1, 0), (1, 0)):
        if search(board, row + d[0], col + d[1], word, index + 1):
            return True

    visited.remove((row, col)) # Backtrack

    return False

def find_word(board, word):
    int M = len(board)
    int N = len(board[0])

    for row in range(M):
        for col in range(N):
            visited = set()
            if search(board, row, col, word, 0, visited):
                return True

```

The worst-case time complexity of this solution is  $O(MN * 4^L)$  where  $L$  is the length of the word and  $M$  and  $N$  are the dimensions of the board.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #99

## Problem

This problem was asked by Microsoft.

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, given [100, 4, 200, 1, 3, 2], the longest consecutive element sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in  $O(n)$  complexity.

## Solution

We can see that if the array of integers was sorted, we'd be able to find the longest consecutive sequence fairly easily. First, we sort the array in ascending order. Then, we traverse the array, keeping the count of the current sequence. If the next element is one higher than the current element, then we increment the count. Otherwise, we start the count over at 1. We simply return the maximum count we've seen overall. The overall run-time of this solution would be  $O(n \log n)$ , since we have to sort the input array. Depending on the sorting implementation and whether or not we are able to modify the array, the space complexity would be  $O(n)$  or  $O(1)$ .

We can improve our solution by using extra space to cache the bounds of sequences we've seen so far. If we get a new number, then we start with a sequence length of 1. If we get a number we've seen already, we can ignore it since none of the bounds should change.

```
def longest_consecutive(self, nums):
    max_len = 0
    bounds = dict()
    for num in nums:
        if num in bounds:
            continue
        left_bound, right_bound = num, num
        if num - 1 in bounds:
            left_bound = bounds[num - 1][0]
```

```
if num + 1 in bounds:  
    right_bound = bounds[num + 1][1]  
    bounds[num] = left_bound, right_bound  
    bounds[left_bound] = left_bound, right_bound  
    bounds[right_bound] = left_bound, right_bound  
    max_len = max(right_bound - left_bound + 1, max_len)  
  
return max_len
```

This solution has a time complexity of  $O(N)$ , and a space complexity of  $O(N)$ .

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #100

## Problem

This problem was asked by Google.

You are in an infinite 2D grid where you can move in any of the 8 directions:

```
(x,y) to  
(x+1, y),  
(x - 1, y),  
(x, y+1),  
(x, y-1),  
(x-1, y-1),  
(x+1,y+1),  
(x-1,y+1),  
(x+1,y-1)
```

You are given a sequence of points and the order in which you need to cover the points. Give the minimum number of steps in which you can achieve it. You start from the first point.

Example:

Input: [(0, 0), (1, 1), (1, 2)]

Output: 2

It takes 1 step to move from (0, 0) to (1, 1). It takes one more step to move from (1, 1) to (1, 2).

## Solution

We can see that the minimum number of steps would be to walk as many diagonal steps as possible, and then walk directly to the second point. If we were to walk directly vertically and then horizontally, it would be a greater number of steps. We can break down the diagonal and vertical/horizontal components by taking the adding the minimum of the vertical or horizontal differences with the remaining distance.

```

// X and Y co-ordinates of the points in order.
// Each point is represented by (X.get(i), Y.get(i))
public int coverPoints(ArrayList<Integer> X, ArrayList<Integer> Y) {
    int totalDistance = 0;
    for (int i = 1; i < X.size(); i++) {
        totalDistance += getDistance(X.get(i - 1), Y.get(i - 1), X.get(i), Y.get(i));
    }
    return totalDistance;
}

private int getDistance(int x1, int y1, int x2, int y2) {
    /* Get diagonal distance component */
    int dist1 = (int)Math.min(Math.abs(x2 - x1), Math.abs(y2 - y1));
    /* Get horizontal/vertical distance component */
    int dist2 = (int)Math.max(Math.abs(x2 - x1), Math.abs(y2 - y1)) - dist1;
    return dist1 + dist2;
}

```

Or, we can simply take the maximum of the vertical and horizontal distances -- this is the total distance.

```

// X and Y co-ordinates of the points in order.
// Each point is represented by (X.get(i), Y.get(i))
public int coverPoints(ArrayList<Integer> X, ArrayList<Integer> Y) {
    int totalDistance = 0;
    for (int i = 1; i < X.size(); i++) {
        totalDistance += getDistance(X.get(i - 1), Y.get(i - 1), X.get(i), Y.get(i));
    }
    return totalDistance;
}

private int getDistance(int x1, int y1, int x2, int y2) {
    return (int)Math.max(Math.abs(x2 - x1), Math.abs(y2 - y1));
}

```





# Daily Coding Problem #101

## Problem

This problem was asked by Alibaba.

Given an even number (greater than 2), return two prime numbers whose sum will be equal to the given number.

A solution will always exist. See [Goldbach's conjecture](#).

Example:

Input: 4

Output: 2 + 2 = 4

If there are more than one solution possible, return the lexicographically smaller solution.

If [a, b] is one solution with a <= b, and [c, d] is another solution with c <= d, then

[a, b] < [c, d]

If a < c OR a==c AND b < d.

## Solution

We can search for the smallest solution by iterating through possible values of the first potential prime. We check whether the first and second numbers are prime, and if so, return them.

```
private static boolean isPrime(int n) {  
    for (int i = 2; i < Math.sqrt(n); i++)  
        if (n % i == 0)  
            return false;  
    return true;  
}
```

```
public ArrayList<Integer> primesum(int a) {  
    for (int i = 2; i <= a / 2; i++) {  
        if (isPrime(i) && isPrime(a - i)) {  
            ArrayList<Integer> output = new ArrayList<>();  
            output.add(i);  
            output.add(a - i);  
            return output;  
        }  
    }  
  
    return null;  
}
```

```
def primesum(self, n):  
    for i in xrange(2, n):  
        if self.is_prime(i) and self.is_prime(n - i):  
            return i, n - i  
  
def is_prime(self, n):  
    if n < 2:  
        return False  
  
    for i in xrange(2, int(n**0.5) + 1):  
        if n % i == 0:  
            return False  
  
    return True
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #102

## Problem

This problem was asked by Lyft.

Given a list of integers and a number K, return which contiguous elements of the list sum to K.

For example, if the list is [1, 2, 3, 4, 5] and K is 9, then it should return [2, 3, 4].

## Solution

The naive approach to solve this question is using brute-force, which has a time complexity of  $O(N^2)$ . We might look at each possible contiguous sublist and check if its sum matches K. Something like this:

```
def find_continuous_k(list, k):
    for first_idx in xrange(len(list)):
        sum = 0
        for last_idx in xrange(first_idx, len(list)):
            sum += list[last_idx]
            if sum == k:
                return list[first_idx:last_idx + 1]
    return None
```

Note that for this function work as expected with negative numbers we must keep looking even if  $sum > k$ .

This question doesn't have a complexity requirement, but here's another way to solve it. Take a look at this code:

```
def find_continuous_k(list, k):
    previous = dict()

    sum = 0
    # sublist starting at the zeroth position work.
```

```
previous[0] = -1
for last_idx, item in enumerate(list):
    sum += item
    previous[sum] = last_idx
    first_idx = previous.get(sum - k)
    if first_idx is not None:
        return list[first_idx + 1:last_idx + 1]
return None
```

As you can see, we store all previous sums and their positions to look for a specific number required to satisfy K. This way, we have O(n) time complexity and O(n \* 2) space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #103

## Problem

This problem was asked by Square.

Given a string and a set of characters, return the shortest substring containing all the characters in the set.

For example, given the string "figehaeci" and the set of characters {a, e, i}, you should return "aeci".

If there is no substring containing all the characters in the set, return null.

## Solution

Here's an O(n) solution. The basic idea is simple: for each starting index, find the least ending index such that the substring contains all of the necessary letters. The trick is that the least ending index increases over the course of the function, so with a little data structure support, we consider each character at most twice.

```
from collections import defaultdict

def smallest(s1, s2):
    assert s2 != ''
    d = defaultdict(int)
    nneg = [0] # number of negative entries in d
    def incr(c):
        d[c] += 1
        if d[c] == 0:
            nneg[0] -= 1
    def decr(c):
        if d[c] == 0:
            nneg[0] += 1
        d[c] -= 1
    for c in s2:
        decr(c)
    for start in range(len(s1)):
        while nneg[0] < len(s2):
            end = start + 1
            while end < len(s1) and nneg[0] < len(s2):
                incr(s1[end])
                end += 1
            if nneg[0] == len(s2):
                return s1[start:end]
        decr(s1[start])
```

```
minlen = len(s1) + 1
j = 0
for i in xrange(len(s1)):
    while nneg[0] > 0:
        if j >= len(s1):
            return minlen
        incr(s1[j])
        j += 1
    minlen = min(minlen, j - i)
    decr(s1[i])
return minlen
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #104

## Problem

This problem was asked by Google.

Determine whether a doubly linked list is a palindrome. What if it's singly linked?

For example, 1 -> 4 -> 3 -> 4 -> 1 returns true while 1 -> 4 returns false.

## Solution

This problem can be solved for both singly and doubly linked list by using iterators.

We can then create an iterator over the first node, and another iterator over the reverse of the linked list. Then compare that the two are the same.

This would take  $O(N)$  space and time.

```
def get_values(node):
    while node:
        yield node.val
        node = node.next

def is_palindrome(node):
    values = get_values(node)
    values_reversed = reversed(list(get_values(node))) # O(N) space

    return all(x == y for x, y in zip(values, values_reversed))
```

Terms of Service

Press

# Daily Coding Problem #106

## Problem

This problem was asked by Pinterest.

Given an integer list where each number represents the number of hops you can make, determine whether you can reach to the last index starting at index 0.

For example, [2, 0, 1, 0] returns true while [1, 1, 0, 1] returns false.

## Solution

This problem can be solved by using dynamic programming:

We can see we can reach the Kth step if we can reach a step  $0 \leq j < K$  and  $j + \text{hops}[j] \geq K$ .

Another approach that would use O(1) space is to start at the first step and keep a variable `steps_left` which represents the maximum number of steps we can move forward. If we store this value and ensure that we can reach the end of each step without this value becoming 0 (except for the last step), then we know we can reach the last step.

```
def can_reach_end(hops):
    steps_left = 1

    for i in range(len(hops) - 1):
        steps_left = max(steps_left - 1, hops[i])
        if steps_left == 0:
            return False
    return True
```

Terms of Service

Press

# Daily Coding Problem #107

## Problem

This problem was asked by Microsoft.

Print the nodes in a binary tree level-wise. For example, the following should print 1, 2, 3, 4, 5.

```
1
/
2   3
 / \
4   5
```

## Solution

We can solve this problem by using a queue, initialized with the root, and continuously grabbing the first element and adding its left child and then its right child to the back of the queue, like so:

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    from queue import Queue

    def print_level_order(root):
        queue = Queue()
        queue.put(root)
```

```
while not queue.empty():
    node = queue.get()
    if node.left:
        queue.put(node.left)
    if node.right:
        queue.put(node.right)
    print(node.val)

root = Node(1, Node(2), Node(3, Node(4), Node(5)))
print_level_order(root)
```

This takes  $O(N)$  time and space, since we have to look at the whole tree.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #108

## Problem

This problem was asked by Google.

Given two strings A and B, return whether or not A can be shifted some number of times to get B.

For example, if A is abcde and B is cdeab, return true. If A is abc and B is acb, return false.

## Solution

If the strings are not the same length, then we can immediately return false.

One solution might be to use a doubly nested for loop, and compare each character starting at different offsets and verifying that they all match up:

```
def is_shifted(a, b):
    if len(a) != len(b):
        return False

    for i in range(len(a)):
        if all(a[(i + j) % len(a)] == b[j] for j in range(len(a))):
            return True

    return False
```

Another cleaner way to solve this might be to concatenate one of the strings to itself (like  $a + a$ ), and try looking for the other string in this concatenated string. If the string is shifted, we should find it in the concatenated string.

```
def is_shifted(a, b):
    if len(a) != len(b):
        return False
```

```
return b in a + a
```

These solutions both take O( $N^2$ ) time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #109

## Problem

This problem was asked by Cisco.

Given an unsigned 8-bit integer, swap its even and odd bits. The 1st and 2nd bit should be swapped, the 3rd and 4th bit should be swapped, and so on.

For example, 10101010 should be 01010101. 11100010 should be 11010001.

Bonus: Can you do this in one line?

## Solution

We can do this by applying a bitmask over all the even bits, and another one over all the odd bits. Then we shift the even bitmask right by one and the odd bitmask left by one.

```
def swap_bits(x):
    EVEN = 0b10101010
    ODD = 0b01010101
    return (x & EVEN) >> 1 | (x & ODD) << 1
```

In one line, that would be:

```
def swap_bits(x):
    return (x & 0b10101010) >> 1 | (x & 0b01010101) << 1
```

Press

# Daily Coding Problem #110

## Problem

This problem was asked by Facebook.

Given a binary tree, return all paths from the root to leaves.

For example, given the tree

```
1
/
2   3
 / \
4   5
```

it should return `[[1, 2], [1, 3, 4], [1, 3, 5]]`.

## Solution

A binary tree is a data structure in which each node has at most two children (left / right). To solve this question, we can iterate over the tree to check nodes without children, also known as leaf nodes.

The `BSTNode` class might be defined like this:

```
class BSTNode:
    def __init__(self, value, left=None, right=None):
        self.parent = None
        self.value = value
        self.left = left
        self.right = right
        if left:
            left.parent = self
        if right:
            right.parent = self
```

```
def path(self):
    path = []
    current = self
    while current:
        path = [current.value] + path
        current = current.parent
    return path
```

We conveniently store the parent node for the path function, so we can find the ancestors to get their path to the root. We can initialize our binary tree with this code:

```
"""
For the binary tree shown below:

      1
     / \
    2   3
   / \
  4   5
"""

root = Node(
    value=1,
    left=Node(2),
    right=Node(3, Node(4), Node(5))
)
```

Sometimes programmers may be tempted to use a recursive function, like this:

```
def find_leaves(node):
    if not node.left and not node.right:
        return [node.path()]
    leaves = []
    if node.left:
        leaves += find_leaves(node.left)
    if node.right:
        leaves += find_leaves(node.right)
    return leaves
```

We should avoid this practice on large or unknown trees because of the possibility of a [call stack overflow](#).

A safer way is to use a FIFO queue, linked list or a simple list. Like this:

```
def find_leaves(node):
    leaves = []
    queue = list()
    queue.append(node)
    while len(queue):
        node = queue.pop()
        if not node.left and not node.right:
            leaves += [node.path()]
            continue
        if node.right:
            queue.append(node.right)
        if node.left:
            queue.append(node.left)
    return leaves
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #111

## Problem

This problem was asked by Google.

Given a word  $W$  and a string  $S$ , find all starting indices in  $S$  which are anagrams of  $W$ .

For example, given that  $W$  is "ab", and  $S$  is "abxaba", return 0, 3, and 4.

## Solution

### Brute force

The brute force solution here would be to go over each word-sized window in  $S$  and check if they're anagrams, like so:

```
from collections import Counter

def is_anagram(s1, s2):
    return Counter(s1) == Counter(s2)

def anagram_indices(word, s):
    result = []
    for i in range(len(s) - len(word) + 1):
        window = s[i:i + len(word)]
        if is_anagram(window, word):
            result.append(i)
    return result
```

This would take  $O(|W| * |S|)$  time. Can we make this any faster?

### Count difference

Notice that moving along the window seems to mean recomputing the frequency counts of the

entire window, when only a little bit of it actually updated. This insight lead us to the following strategy:

- Make a frequency dictionary of the target word
- Continuously diff against it as we go along the string
- When the dict is empty, the window and the word matches

We diff in our frequency dict by incrementing the new character in the window and removing old one.

```
class FrequencyDict:  
    def __init__(self, s):  
        self.d = {}  
        for char in s:  
            self.increment(char)  
  
    def _create_if_not_exists(self, char):  
        if char not in self.d:  
            self.d[char] = 0  
  
    def _del_if_zero(self, char):  
        if self.d[char] == 0:  
            del self.d[char]  
  
    def is_empty(self):  
        return not self.d  
  
    def decrement(self, char):  
        self._create_if_not_exists(char)  
        self.d[char] -= 1  
        self._del_if_zero(char)  
  
    def increment(self, char):  
        self._create_if_not_exists(char)  
        self.d[char] += 1  
        self._del_if_zero(char)  
  
def anagram_indices(word, s):  
    result = []  
  
    freq = FrequencyDict(word)  
  
    for char in s[:len(word)]:  
        freq.decrement(char)
```

```
if freq.is_empty():
    result.append(0)

for i in range(len(word), len(s)):
    start_char, end_char = s[i - len(word)], s[i]
    freq.increment(start_char)
    freq.decrement(end_char)
    if freq.is_empty():
        beginning_index = i - len(word) + 1
        result.append(beginning_index)

return result
```

This should run in  $O(S)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #112

## Problem

This problem was asked by Twitter.

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. Assume that each node in the tree also has a pointer to its parent.

According to the definition of [LCA on Wikipedia](#): "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."

## Solution

When given a problem statement, make sure to clarify the question, as well as its inputs and outputs. In this problem, we are given a binary tree, and not necessarily a binary search tree. We will be given the root of a binary tree as input, as well as two nodes a and b. We are to return the node that satisfies the LCA definition of the two nodes, or null if no such nodes exist.

It's also important to identify edge cases, and provide examples of these cases. One important edge case is where both a and b point to the same node. Another case that may be worth mentioning is if nodes a and b do not share any ancestors. In this case, a and b must not be in the same tree. You can clarify with your interviewer whether or not you can assume a and b are in the tree.

The key to this problem is that we are given parent pointers. Although this problem can be solved without parent pointers, the iterative solution is simplified when given these. The interviewer may also try to direct your solution to use the pointers.

In order to find the LCA, we have to identify the first node in which the paths from a and b intersect while traversing upwards to the root. One way we can do this is by using two loops. For each node in a's path, check whether there is a node in b's path that is the same. This solution would have an overall time complexity of  $O(\text{depth}(a) * \text{depth}(b))$ , which could be  $O(N^2)$  in the worst-case, where a and b are leaf nodes, and the tree is unbalanced (close to a linked list). The space complexity is  $O(1)$ .

We can reduce the time complexity by trading off space. We first store the nodes seen in the path from a in a hash set. Then, while iterating from b to the root, we check whether the current node is found in the set. We return the first node that meets this condition. Then, we have a solution which runs in linear (or  $O(\text{depth}(a) + \text{depth}(b))$ ) time, and linear space.

However, we can improve the space complexity on this solution by observing that both paths must converge and end at the root, as long as both nodes are in the same tree. Then, we can reduce this problem to finding the first node of two linked lists' intersection. First, we get the lengths of both nodes' paths (their depth in the tree). Then, we move the longer path's pointer forward so both pointers are the same distance from the root. Finally, we move both pointers in lock-step and return when we find the first node that is the same from both paths.

```
class TreeNode:
    def __init__(self, val):
        self.left = None
        self.right = None
        self.parent = None
        self.val = val

def lca(root, a, b):
    def depth(node):
        count = 0
        while node:
            count += 1
            node = node.parent

        return count

    depth_a, depth_b = depth(a), depth(b)
    if depth_a < depth_b:
        while depth_a < depth_b:
            b = b.parent
            depth_b -= 1
    elif depth_a > depth_b:
        while depth_a > depth_b:
            a = a.parent
            depth_a -= 1

    while a and b and (a is not b):
        a = a.parent
        b = b.parent

    return a if (a is b) else None
```

Now, our solution has a linear time complexity, and uses constant space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #113

## Problem

This problem was asked by Google.

Given a string of words delimited by spaces, reverse the words in string. For example, given "hello world here", return "here world hello"

Follow-up: given a mutable string representation, can you perform this operation in-place?

## Solution

One way we can solve this problem is by splitting the input string into a list of words, and then reversing the order.

```
def reverse_words(string):
    words = string.split(' ')
    words = reversed(words)
    return ' '.join(words)
```

This solution works in  $O(N)$  time, and uses  $O(N)$  space since we create a new list to hold the words, and a new string to represent the output.

If we were given a mutable string representation instead of an immutable string, we can perform the reverse operation in-place. Let's assume we are given a list of characters, which we will modify in-place. We can easily reverse a substring within a string in  $O(N)$  time and  $O(1)$  space. However, simply reversing the entire string or reversing each word won't get return the correct solution -- we must do both operations. First, we reverse the entire string to get the string "ereh dlrow olleh". Then, we reverse each word within the string to obtain the original words: "here world hello".

```
def reverse_words(string_list):
    # Helper function to reverse string in place
    def reverse(l, start, end):

        # Reverses characters from index start to end (inclusive)
```

```
while start < end:
    l[start], l[end] = l[end], l[start]
    start += 1
    end -= 1

# Reverse the entire string
reverse(string_list, 0, len(string_list) - 1)

# Reverse each word in the string
start = 0
for end in range(len(string_list)):
    if string_list[end] == ' ':
        print(start, end)
        reverse(string_list, start, end - 1)
        start = end + 1
# Reverse the last word
reverse(string_list, start, len(string_list) - 1)

return string_list
```

This solution has a worst-case time complexity of  $O(N)$ , since each character is swapped a maximum of two times. This solution also gives us a worst-case space complexity of  $O(1)$ .

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #115

## Problem

This problem was asked by Google.

Given two non-empty binary trees  $s$  and  $t$ , check whether tree  $t$  has exactly the same structure and node values with a subtree of  $s$ . A subtree of  $s$  is a tree consists of a node in  $s$  and all of this node's descendants. The tree  $s$  could also be considered as a subtree of itself.

## Solution

We can solve this problem by solving the subproblem of whether the tree  $t$  has the same structure and node values of any subtree of  $s$ . We can implement a helper function `is_equal(s, t)` that checks whether the two trees have the same structure and values. Then, for each node  $u$  in  $s$ , we return True if `is_equal(u, t)` for any  $u$ , otherwise we return False. The implementation of `is_equal` is straightforward, and takes  $O(\min(M, N))$  time in the worst case, where  $M$  and  $N$  are the number of nodes in trees  $u$  and  $t$ , respectively.

Once we have the `is_equal(u, t)` function, we can write a recursion for the overall answer. We should return True if the current node is equal to  $t$ , or if either of the subtrees is equal to  $t$ . The base case should handle an empty subtree.

```
def is_subtree(s, t):
    def is_equal(s, t):
        if s is None and t is None:
            return True
        if s is None or t is None:
            return False
        if s.val != t.val:
            return False
        return is_equal(s.left, t.left) and is_equal(s.right, t.right)

    if s is None:
        return False
    if is_equal(s, t):
        return True
    return is_subtree(s.left, t) or is_subtree(s.right, t)
```

```

if is_equal(s, t):
    return True
return is_subtree(s.left, t) or is_subtree(s.right, t)

```

Overall, the time complexity of this algorithm is  $O(M * N)$  in the worst case, where  $M$  is the number of nodes in tree  $s$  and  $N$  is the number of nodes in tree  $t$ , since we would perform up to  $N$  calls to `is_subtree`. The space complexity of this solution is  $O(M)$ , since the depth of the recursion can go up to the number of nodes of tree  $s$  in the worst case.

Another way we can solve this problem is by encoding both trees using a pre-order traversal (with null markers). We return whether the string representation of subtree  $s$  is found within the string representation of tree  $t$ . Since we are using a pre-order traversal, subtrees of  $t$  will each be found in one contiguous substring, enabling us to search for the substring from the traversal of tree  $s$ . We must mark null pointers during our traversal, as pre-order traversals without these can be ambiguous in how the tree should be reconstructed. We also need to wrap the start and end of the traversal string, to avoid edge cases such as 12 and 1.

```

def is_subtree(s, t):
    def preorder(root):
        traversal = []
        stack = [root]
        while stack:
            n = stack.pop()
            if n is None:
                traversal.append('.') # null marker
                continue
            else:
                traversal.append(str(n.val))
            stack.append(n.right)
            stack.append(n.left)
        return ',' + ','.join(traversal) + ',' # Wrap result

    s_str = preorder(s)
    t_str = preorder(t)
    return t_str in s_str

```

The `preorder(root)` function takes  $O(N)$  time, and up to  $O(N)$  space in the worst case. The overall runtime depends on how the method checking for whether the substring is contained is implemented. The Python 3 implementation uses a variant of the [Boyer–Moore string search algorithm](#), which runs in  $O(M + N)$  time in the average case, and  $O(M * N)$  in the worst case.

Finally, we could also check for subtree quality by using a hashing algorithm. One frequently-used data structure for comparing trees using hashing is a [Merkle tree](#).

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #116

## Problem

This problem was asked by Jane Street.

Generate a finite, but an arbitrarily large binary tree quickly in O(1).

That is, generate() should return a tree whose size is unbounded but finite.

## Solution

### Eager tree generation

If we ignore the O(1) generation constraint, we can create an unbounded tree by using randomness.

That is, we can generate the left and right sub-trees recursively X% of the time.

Since the question didn't have any constraint about the values the nodes can have, it's arbitrarily set to 0.

```
import random

class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def generate():
        root = Node(0)

        if random.random() < 0.5:
```

```

root.left = generate()

if random.random() < 0.5:
    root.right = generate()

return root

```

## Lazy tree generation

The trick here is that we can generate the tree **lazily**. Here we use Python's `property` keyword, which lets us define a property that of an object at look-up time.

When a `left` or `right` property is looked up, we check if that sub-tree has been evaluated. If not, we recursively create a new node half the time. If it has been, then we just return that node.

The object is  $O(1)$  to create since nothing happens when it's created.

```

class Node:

    def __init__(self, val, left=None, right=None):
        self.val = val
        self._left = left
        self._right = right

        self._is_left_evaluated = False
        self._is_right_evaluated = False

    @property
    def left(self):
        if not self._is_left_evaluated:
            if random.random() < 0.5:
                self._left = Node(0)

        self._is_left_evaluated = True
        return self._left

    @property
    def right(self):
        if not self._is_right_evaluated:
            if random.random() < 0.5:
                self._right = Node(0)

        self._is_right_evaluated = True
        return self._right

def generate():
    return Node(0)

```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #117

## Problem

This problem was asked by Facebook.

Given a binary tree, return the level of the tree with minimum sum.

## Solution

A binary tree is a data structure in which each node has at most two children (left and right), and a level is how many parents a node has until it reaches the root.

To better explain this question, let's see this example:

```
1      (Level 0 = 1)
 / \
2   3   (Level 1 = 2 + 3)
 / \
4   5   (Level 2 = 4 + 5)
```

One possible way to solve this problem is iterate level by level on the tree's nodes, so we can get theirs respective sum. Like this code:

```
from Queue import Queue
from collections import defaultdict

class Node:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def smallest_level(root):
    queue = Queue()
    queue.put((root, 0))
    level_to_sum = defaultdict(int) # Maps level to its sum

    while not queue.empty():
        node, level = queue.get()
        level_to_sum[level] += node.value

        if node.right:
            queue.put((node.right, level + 1))

        if node.left:
            queue.put((node.left, level + 1))

    return min(level_to_sum, key=level_to_sum.get)
```

The complexity of this function is O(N), and works both with positive or negative values. It works by doing a [breadth first search](#) and keeping track of the sum in each level.

You can try yourself with these examples:

```
"""
    1
   / \
 -2   -3      (level 1 is the minimum)
  / \
 4   -5

"""
root = Node(
    value=1,
    left=Node(-2),
    right=Node(-3, Node(4), Node(-5))
)
print(minimum_level_sum(root))
```

```
"""
    1
   /     \
  2       3
 / \       \
4  5       6
  \       / \
 -1   -7   -8  (level 3 is the minimum)

"""
root = Node(
    value=1,
    left=Node(2, Node(4), Node(5, Node(-1))),
    right=Node(
        value=3,
        right=Node(6, Node(-7), Node(-8))
    )
)
print(minimum_level_sum(root))
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #118

## Problem

This problem was asked by Google.

Given a sorted list of integers, square the elements and give the output in sorted order.

For example, given [-9, -2, 0, 2, 3], return [0, 4, 4, 9, 81].

## Solution

A brute force method would be to simply square and sort the list, like so: `sorted([x ** 2 for x in lst])`. This would result in  $O(n \log n)$  time.

A faster way to do this would be to notice that there are two natural sublists in `lst`: The positive numbers and negative numbers.

The positive numbers, if sorted, would still remain sorted, while negative numbers, if sorted, would be reverse sorted. So by reversing the negative numbers and then sorting it we get two sorted sections in `lst`. Then we can apply a `merge` operation, similar to merge sort.

```
def square_sort(lst):
    negatives = [x for x in lst if x < 0]
    non_negatives = [x for x in lst if x >= 0]

    negatives_square_sorted = [x ** 2 for x in reversed(negatives)]
    non_negatives_square_sorted = [x ** 2 for x in non_negatives]

    return _merge(negatives_square_sorted, non_negatives_square_sorted)

def _merge(left_lst, right_lst):
    result = []

    i = j = 0
```

```
while i < len(left_lst) and j < len(right_lst):
    if left_lst[i] < right_lst[j]:
        result.append(left_lst[i])
        i += 1
    elif left_lst[i] > right_lst[j]:
        result.append(right_lst[j])
        j += 1
    else:
        result.append(left_lst[i])
        result.append(right_lst[j])
        i += 1
        j += 1

result.extend(left_lst[i:])
result.extend(right_lst[j:])
return result
```

This takes O(N) time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #119

## Problem

This problem was asked by Google.

Given a set of closed intervals, find the smallest set of numbers that covers all the intervals. If there are multiple smallest sets, return any of them.

For example, given the intervals [0, 3], [2, 6], [3, 4], [6, 9], one set of numbers that covers all these intervals is {3, 6}.

## Solution

This problem becomes clearer if we sort the intervals by the starting points. For example, intervals [[10, 20], [1, 6], [3, 8], [7, 12]] should become [[1, 6], [3, 8], [7, 12], [10, 20]].

Now, to cover the first interval, [1, 6], we must pick a number in between the interval. However, if the next interval intersects, then we can solve an easier interval problem of picking a point between their intersection. This would let us use 1 less point to cover the intervals. Then, we can look at the third intersection and so forth to find the first k intervals which all intersect. Once we find an interval that doesn't intersect, we can pick a point in the intersection of all the previous intervals. Then we repeat the process starting from the current interval.

In the above example, the intersection of [[1, 6], [3, 8]] is [3, 6] while the intersection of [7, 12], [10, 20] is [10, 12].

```
def covering(intervals):
    intervals.sort(key=lambda x: x[0])

    result = []
    i = 0

    while i < len(intervals):
        interval = intervals[i]
```

```
while i < len(intervals) and intersecting(intervals[i], interval):
    interval = (max(intervals[i][0], interval[0]), min(intervals[i][1], interval[1]))
    i += 1

result.append(interval[1])
return result

def intersecting(x, y):
    return not (x[0] > y[1] or y[0] > x[1])
```

The main while loop takes  $O(n)$  since we iterate through the intervals, and sorting the interval takes  $O(n \log n)$ , so this takes  $O(n \log n)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #120

## Problem

This problem was asked by Microsoft.

Implement the singleton pattern with a twist. First, instead of storing one instance, store two instances. And in every even call of `getInstance()`, return the first instance and in every odd call of `getInstance()`, return the second instance.

## Solution

This question is more about programming and design patterns than computer science.

The singleton pattern allows you to limit the number of objects of a class to one instance. This is helpful in a large application either to conserve resources such as memory or to make correctness easier to reason about. For example, to represent configuration of a system, it would be helpful to have one centralized object.

In this particular question, we ask for a twist on the classic singleton by allowing two instances of a class. We do this by adding another static field as well as `calls` variable to keep track of the number of calls made to `getInstance`.

```
public class Service {  
    private static Service instanceOne = null;  
    private static Service instanceTwo = null;  
  
    private static int calls = 0;  
  
    private Service() {  
        // Disallow creation through the constructor  
    }  
  
    public static Service getInstance() {  
        if(instanceOne == null) {  
            instanceOne = new Service();  
            instanceTwo = new Service();  
        }  
        if(calls % 2 == 0) {  
            return instanceOne;  
        } else {  
            return instanceTwo;  
        }  
    }  
}
```

```
    }

    if (calls++ % 2 == 0) {
        return instanceOne;
    }
    return instanceTwo;
}

}
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #122

## Problem

This question was asked by Zillow.

You are given a 2-d matrix where each cell represents number of coins in that cell. Assuming we start at `matrix[0][0]`, and can only move right or down, find the maximum number of coins you can collect by the bottom right corner.

For example, in this matrix

```
0 3 1 1  
2 0 0 4  
1 5 3 1
```

The most we can collect is  $0 + 2 + 1 + 5 + 3 + 1 = 12$  coins.

## Solution

Notice the recursive structure of this problem.

- If we're at the bottom-right corner of the matrix, the most amount of coins we can collect is that cell.
- If we're at the bottom of the matrix, the most amount of coins we can collect is that cell and amount we can collect by moving right.
- If we're at the rightmost part of the matrix, the most amount of coins we can collect is that cell and amount we can collect by moving down.
- Otherwise, the most amount of coins we can collect is that cell and the max of either the most amount we can collect from moving right or the most amount we can collect from moving down.

For example, in the following matrix,

0 2 3  
5 4 9  
6 7 1

- In the 1 cell, bottom-right cell, the most we can collect is that cell: 1.
- In the 7 cell, a bottom cell, we can only collect by taking coins from the right:  $7 + 1$ .
- In the 9 cell, a right-most cell, we can only collect by taking coins from below  $9 + 1$ .
- In the 4 cell, a non-edge cell, we can collect 4 plus the max of going right or going down:  
 $4 + \max(9 + 1, 7 + 1) == 14$ .

This leads itself to the following code:

```
def collect_coins(matrix, r=0, c=0, cache=None):  
    if cache is None:  
        cache = {}  
  
    num_rows = len(matrix)  
    num_cols = len(matrix[0])  
  
    is_bottom = r == num_rows - 1  
    is_rightmost = c == num_cols - 1  
  
    if (r, c) not in cache:  
  
        if is_bottom and is_rightmost:  
            cache[r, c] = matrix[r][c]  
        elif is_bottom:  
            cache[r, c] = matrix[r][c] + collect_coins(matrix, r, c + 1, cache)  
        elif is_rightmost:  
            cache[r, c] = matrix[r][c] + collect_coins(matrix, r + 1, c, cache)  
        else:  
            cache[r, c] = matrix[r][c] + max(collect_coins(matrix, r + 1, c, cache),  
                                              collect_coins(matrix, r, c + 1, cache))  
  
    return cache[r, c]
```

This is  $O(MN)$  both space and time since we compute the solution for each cell at most once.



# Daily Coding Problem #123

## Problem

This problem was asked by LinkedIn.

Given a string, return whether it represents a number. Here are the different kinds of numbers:

- "10", a positive integer
- "-10", a negative integer
- "10.1", a positive real number
- "-10.1", a negative real number
- "1e5", a number in scientific notation

And here are examples of non-numbers:

- "a"
- "x 1"
- "a -2"
- "\_"

## Solution

We can solve this problem bottom-up, starting from positive integers:

- A positive integer contains only digits.
- A negative integer starts with '-' and the rest is a positive integer.
- A positive decimal contains one '.' and the substrings before and after '.' are positive integers.
- A negative decimal starts with '-' and the rest is a positive decimal.
- A positive number is either a positive integer or decimal.

- A negative number is either a negative integer or decimal.
- A scientific notation number contains one 'e' and the substrings before and after 'e' are each either a positive or negative number.
- And finally, a number is either a positive number, a negative number, or a scientific number.

```

def is_number(s):
    return is_positive_number(s) or is_negative_number(s) or is_scientific_number(s)

def is_scientific_number(s):
    if s.count('e') != 1:
        return False

    before_e, after_e = s.split('e')

    return ((is_positive_number(before_e) or is_negative_number(before_e))
            and (is_positive_number(after_e) or is_negative_number(after_e)))

def is_positive_number(s):
    return is_positive_integer(s) or is_positive_real(s)

def is_negative_number(s):
    return is_negative_integer(s) or is_negative_real(s)

def is_negative_real(s):
    return s.startswith('-') and is_positive_real(s[1:])

def is_positive_real(s):
    if s.count('.') != 1:
        return False

    integer_part, decimal_part = s.split('.')

    return is_positive_integer(integer_part) and is_positive_integer(decimal_part)

def is_negative_integer(s):
    return s.startswith('-') and is_positive_integer(s[1:])

def is_positive_integer(s):
    return s.isdigit()

```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #125

## Problem

This problem was asked by Google.

Given the root of a binary search tree, and a target K, return two nodes in the tree whose sum equals K.

For example, given the following tree and K of 20

```
    10
   /   \
  5    15
 /   \
11   15
```

Return the nodes 5 and 15.

## Solution

This question is similar to the two-sum problem with a list. We can actually reduce this problem into that one by turning the tree into a list. To save some space, we'll use [generators](#), which are like list-like but are generated on-the-fly.

```
def two_sum(root, K):
    seen = {} # Map of val to node

    for node in iter_tree(root):
        if K - node.val in seen:
            return (node, seen[K - node.val])
        seen[node.val] = node

    return None
```

```
def iter_tree(root):
    if root:
        for node in iter_tree(root.left):
            yield node

        yield root

        for node in iter_tree(root.right):
            yield node
```

Another solution is to simply iterate over each node and do a binary tree search for  $K - \text{node.val}$ . This takes  $O(N \log N)$  time since for each node, we do a search which takes  $\log N$ . However, it will only take  $O(\log N)$  space because the call stack gets  $\log N$  deep.

```
def two_sum(root, K):
    for node_one in iter_tree(root):
        node_two = search(root, K - node_one.val)

        if node_two:
            return (node_one, node_two)

    return None

def search(node, val):
    if not node:
        return None

    if node.val == val:
        return node
    elif node.val < val:
        return search(node.right, val)
    else:
        return search(node.left, val)
```

# Daily Coding Problem #126

## Problem

This problem was asked by Facebook.

Write a function that rotates a list by k elements. For example, [1, 2, 3, 4, 5, 6] rotated by two becomes [3, 4, 5, 6, 1, 2]. Try solving this without creating a copy of the list. How many swap or move operations do you need?

## Solution

We can naively rotate a list without creating a copy by simply moving each element down by one, k times. Don't forget to wrap around:

```
def rotate(lst, k):
    for _ in range(k):
        # Move each element down by one.
        first_element = lst[0]
        for i in range(len(lst) - 1):
            lst[i] = lst[i + 1]
        lst[len(lst) - 1] = first_element
    return lst
```

Although this takes constant space, this will take  $O(nk)$  time. Can we do this any faster?

We can view this problem as transforming the list into  $lst[k:] + lst[:k]$ . By reversing these subarrays and then reversing the whole array we can effectively rotate the array in linear time and without copying.

Take our example, [1, 2, 3, 4, 5, 6] and  $k = 2$ .

- First reverse from 0 to k: [2, 1, 3, 4, 5, 6]
- Then reverse from k to n: [2, 1, 6, 5, 4, 3]
- Then reverse from 0 to n: [3, 4, 5, 6, 1, 2]

```
def rotate(lst, k):
    reverse(lst, 0, k - 1)
    reverse(lst, k, len(lst) - 1)
    reverse(lst, 0, len(lst) - 1)

def reverse(lst, i, j):
    while i < j:
        lst[i], lst[j] = lst[j], lst[i]
        i += 1
        j -= 1
```

Since reversing a list takes  $O(n)$  and we do a constant number of reversals (3), this algorithm takes  $O(n)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #127

## Problem

This problem was asked by Microsoft.

Let's represent an integer in a linked list format by having each node represent a digit in the number. The nodes make up the number in reversed order.

For example, the following linked list:

1 -> 2 -> 3 -> 4 -> 5

is the number 54321.

Given two linked lists in this format, return their sum in the same linked list format.

For example, given

9 -> 9

5 -> 2

return 124 (99 + 25) as:

4 -> 2 -> 1

## Solution

We can add two numbers using the same process as elementary grade school addition — add the least significant digits with a carry.

We'll start at the head of the two nodes, and compute  $\text{sum} \% 10$ . We write this down, move the two nodes up and add a carry if the sum was greater than 10. A tricky part here is to finding the terminating condition. We can see that this happens when there is no more carry and the two linked lists have reached the end. Since this is tricky to represent in code, we instead extend the

nodes one more until they are both None and carry is 0 and then simply return None.

```
class Node:  
    def __init__(self, val, next=None):  
        self.val = val  
        self.next = next  
  
def add(node0, node1, carry=0):  
    if not node0 and not node1 and not carry:  
        return None  
  
    node0_val = node0.val if node0 else 0  
    node1_val = node1.val if node1 else 0  
    total = node0_val + node1_val + carry  
  
    node0_next = node0.next if node0 else None  
    node1_next = node1.next if node1 else None  
    carry_next = 1 if total >= 10 else 0  
  
    return Node(total % 10, add(node0_next, node1_next, carry_next))
```

This will run in  $O(\max(m, n))$  time, where  $m$  and  $n$  are the lengths of the linked lists.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #129

## Problem

Given a real number  $n$ , find the square root of  $n$ . For example, given  $n = 9$ , return 3.

## Solution

This is a classic question that was first solved by Heron of Alexandria after the first century.

Alexandra's algorithm starts with a guess and iteratively improves until convergence. In each iteration, we improve our guess by averaging guess and  $n / \text{guess}$ . This formula comes from the fact that if guess is an overestimate, then  $n / \text{guess}$  would be an underestimate. For example, If  $n$  is 9, then a guess of 4 is an overestimate and  $9 / 4$  is an underestimate. On the other hand, if guess is an underestimate, then  $n / \text{guess}$  is an overestimate. The process converges when guess is 3 which is equal to  $9 / 3$ . For the full proof, please see [here](#).

```
def squareroot(n, error=0.00001):
    guess = 1

    while abs(guess ** 2 - n) >= error:
        guess = (guess + n / guess) / 2.0
    return guess
```

A more realistic answer, in an interview setting, would be to use binary search. We can pick an underestimate  $\text{lo} = 0$  and an overestimate  $\text{hi} = n$  to start. And we can keep the loop invariant that the true  $\text{squareroot}(n)$  would always lie between  $[\text{lo}, \text{hi}]$ . To do this, we see if  $\text{guess} = (\text{lo} + \text{hi}) / 2$  is an overestimate, and if it is, bring the  $\text{hi}$  down to  $\text{guess}$ . Otherwise, we bring the  $\text{lo}$  up to  $\text{guess}$ . The loop finishes when  $\text{guess} ** 2$  is very close to  $n$  (plus or minus `error`):

```
def squareroot(n, error=0.00001):
    lo = 0.0
    hi = n

    guess = (lo + hi) / 2.0
```

```
while abs(guess ** 2 - n) >= error:  
    if guess ** 2 > n:  
        hi = guess  
    else:  
        lo = guess  
    guess = (lo + hi) / 2.0  
  
return guess
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #130

## Problem

This problem was asked by Facebook.

Given an array of numbers representing the stock prices of a company in chronological order and an integer  $k$ , return the maximum profit you can make from  $k$  buys and sells. You must buy the stock before you can sell it, and you must sell the stock before you can buy it again.

For example, given  $k = 2$  and the array  $[5, 2, 4, 0, 1]$ , you should return 3.

## Solution

This problem is quite a bit harder than Daily Coding Problem #47, where we only needed to buy and sell a stock once to get the max profit.

Let's consider the last price in the array. Either we used that price in a transaction or we didn't. Thus, the max profit for our input should be the max of either:

- The max profit of  $\text{prices}[:-1]$  using  $k$  transactions (we didn't use the last price)
- The best you can get by selling the stock on the last day (we use the last price)

We can use dynamic programming to represent the maximum profit we can get. Let  $\text{dp}[i][j]$  represent the maximum profit using at most  $i$  transactions up to day  $j$  (inclusive).

So the recurrence is:

```
dp[0][j] = 0 # If k = 0 then no way to profit
dp[i][0] = 0 # If no prices then no way to profit

dp[i][j] = max(
    dp[i][j - 1], # We don't use the last price
    max(price[j] - price[m] + dp[i - 1][m] for m in range(j)) # Best we can do by completing last
transaction on jth day
)
```

So let's build our matrix bottom-up and fill it according to the recurrence:

```
from math import inf

def max_profit(k, prices):
    n = len(prices)
    dp = [[0 for _ in range(n)] for _ in range(k + 1)]

    for i in range(k + 1):
        dp[i][0] = 0

    for j in range(n):
        dp[0][j] = 0

    for i in range(1, k + 1):
        for j in range(1, n):
            best_so_far = -inf
            for m in range(j):
                best_so_far = max(best_so_far, prices[j] - prices[m] + dp[i - 1][m])
            dp[i][j] = max(best_so_far, dp[i][j - 1])

    return dp[k][n - 1]
```

This will run in  $O(kn^2)$  time and take  $O(kn)$  space.

We can improve the time complexity by noticing that we're redoing some of the same calculations in the innermost for loop. In the term  $\text{prices}[j] - \text{prices}[m] + \text{dp}[i - 1][m]$ ,  $\text{prices}[j]$  stays constant and the rest of the term can be factored into the outer loop.

```
from math import inf

def max_profit(k, prices):
    n = len(prices)
    dp = [[0 for _ in range(n)] for _ in range(k + 1)]

    for i in range(k + 1):
        dp[i][0] = 0

    for j in range(n):
        dp[0][j] = 0

    for i in range(1, k + 1):
        best_so_far = -inf
        for j in range(1, n):
            best_so_far = max(best_so_far, dp[i - 1][j - 1] - prices[j - 1])
        dp[i][j] = max(best_so_far, dp[i][j - 1])

    return dp[k][n - 1]
```

```
dp[i][j] = max(dp[i][j - 1], prices[j] + best_so_far)
```

```
return dp[k][n - 1]
```

Now that we got rid of the inner for loop, this runs in  $O(kn)$  time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #131

## Problem

This question was asked by Snapchat.

Given the head to a singly linked list, where each node also has a “random” pointer that points to anywhere in the linked list, deep clone the list.

## Solution

This problem has a straightforward solution using  $O(n)$  space:

- Create a clone of the linked list, disregarding random pointers.
- Make a hashmap that maps from an original node to its cloned counterpart.
- Iterate through both the clone and originals at the same time. For a given clone node, find the original's random clone counterpart in the hashmap, and set it as its random node.

However, there's a clever way to use even less space.

- First, double the linked list by interleaving it with cloned nodes (without random set). For example, given  $1 \rightarrow 2 \rightarrow 3$ , becomes  $1 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 3$ .
- Set the cloned nodes' random by following the original, previous node's `random.next`.
- Restore the linked lists by separating them. For example, each original nodes need to set `node.next = node.next.next`.

```
def clone(node):
    node = double(node)
    set_random_pointers(node)

    clone_head = node.next

    while node:
```

```
clone_match = node.next

    if clone_match.next:
        node.next, clone_match.next = node.next.next, clone_match.next.next
    else:
        node.next, clone_match.next = node.next.next, None

    node = node.next

return clone_head

def set_random_pointers(node):
    while node:
        clone_match = node.next
        clone_match.random = node.random.next

        node = node.next.next

def double(node):
    root = node
    while node:
        copy = Node(node.val)
        next = node.next

        node.next = copy
        copy.next = next
        node = next

    return root
```

Since we only store pointers, this only takes O(1) extra space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #132

## Problem

This question was asked by Riot Games.

Design and implement a HitCounter class that keeps track of requests (or hits). It should support the following operations:

- `record(timestamp)`: records a hit that happened at `timestamp`
- `total()`: returns the total number of hits recorded
- `range(lower, upper)`: returns the number of hits that occurred between timestamps `lower` and `upper` (inclusive)

Follow-up: What if our system has limited memory?

## Solution

Let's first assume the timestamps in [Unix time](#), that is, integers.

We can naively create a HitCounter class by simply using an unsorted list to store all the hits, and implement range by querying over each hit one by one:

```
class HitCounter:  
    def __init__(self):  
        self.hits = []  
  
    def record(self, timestamp):  
        self.hits.append(timestamp)  
  
    def total(self):  
        return len(self.hits)  
  
    def range(self, lower, upper):  
        count = 0
```

```
    for hit in self.hits:
        if lower <= hit <= upper:
            count += 1
    return count
```

Here, `record()` and `total()` would take constant time, but `range()` would take  $O(n)$  time.

One tradeoff we could make here is to use a sorted list or binary search tree to keep track of the hits. That way, `range()` would now take  $O(\lg n)$  time, but so would `record()`.

We'll use python's `bisect` library to maintain sortedness:

```
import bisect

class HitCounter:
    def __init__(self):
        self.hits = []

    def record(self, timestamp):
        bisect.insort_left(self.hits, timestamp)

    def total(self):
        return len(self.hits)

    def range(self, lower, upper):
        left = bisect.bisect_left(self.hits, lower)
        right = bisect.bisect_right(self.hits, upper)
        return right - left
```

This will still take up a lot of space, though -- one element for each timestamp.

To address the follow-up question, we can make several possible trade-offs.

One possible trade-off would be to sacrifice accuracy for memory by grouping together timestamps in a coarser granularity, such as minute or even hours. That means we'll lose some accuracy around the borders but we'd be using up to a constant factor less space.

For our solution, we'll keep track of each group in a tuple where the first item is the timestamp in minutes and the second is the number of hits occurring within that minute. We'll sort the tuples by minute for  $O(\lg n)$  record:

```
import bisect
from math import floor

class HitCounter:
    def __init__(self):
        self.counter = 0
```

```
self.hits = [] # (timestamp in minutes, # of times)

def record(self, timestamp):
    self.counter += 1

    minute = floor(timestamp / 60)
    i = bisect.bisect_left([hit[0] for hit in self.hits], minute)

    if i < len(self.hits) and self.hits[i][0] == minute:
        self.hits[i] = (minute, self.hits[i][1] + 1)
    else:
        self.hits.insert(i, (minute, 1))

def total(self):
    return self.counter

def range(self, lower, upper):
    lower_minute = floor(lower / 60)
    upper_minute = floor(upper / 60)
    lower_i = bisect.bisect_left([hit[0] for hit in self.hits], lower_minute)
    upper_i = bisect.bisect_right([hit[0] for hit in self.hits], upper_minute)

    return sum(self.hits[i][1] for i in range(lower_i, upper_i))
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #133

## Problem

This problem was asked by Amazon.

Given a node in a binary search tree, return the next bigger element, also known as the inorder successor.

For example, the inorder successor of 22 is 30.

```
10
/
5   30
/
22  35
```

You can assume each node has a parent pointer.

## Solution

We can use case-analysis to break the problem down to two steps.

- First, if there is a right child of node, then the leftmost descendant of `node.right` (or just `node.right` if it has none) is simply the inorder successor.
- Otherwise, we can find the inorder successor by traversing through the parent pointers, keeping track of the current node and parent. When we find a parent whose left child is equal to node, then we know this is the inorder successor.

Let's look at an example.

```
10
/
5   30
/
22  35
```

```
/ \  
22   35  
\  
  25
```

- The inorder successor of 10 is 22 since it has a right child and 22 is the leftmost child of `node.right`.
- The inorder successor of 25 is 30 since 30 is the first parent where `parent.left` is `node`.

```
class Node:  
    def __init__(self, val, left=None, right=None, parent=None):  
        self.val = val  
        self.left = left  
        self.right = right  
        self.parent = parent  
  
    def inorder_successor(self):  
        if self.right:  
            return leftmost(self.right)  
  
        parent = self.parent  
  
        while parent and parent.left is not self:  
            parent, self = parent.parent, parent  
  
        return parent  
  
    def leftmost(self):  
        while self.left:  
            self = self.left  
        return self
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #134

## Problem

This problem was asked by Facebook.

You have a large array with most of the elements as zero.

Use a more space-efficient data structure, `SparseArray`, that implements the same interface:

- `init(arr, size)`: initialize with the original large array and size.
- `set(i, val)`: updates index at `i` with `val`.
- `get(i)`: gets the value at index `i`.

## Solution

Since the original array is mostly zeroes, we should be able to manage our array with a lot less space by only keeping track of the non-zero values and indices. We can use a dictionary to keep track of those, and default to zero when the key is not found in our dictionary.

Remember to also check the bounds when setting or getting `i`, and to clean up any indices if we're setting an index to zero again.

```
class SparseArray:  
    def __init__(self, arr, n):  
        self.n = n  
        self._dict = {}  
        for i, e in enumerate(arr):  
            if e != 0:  
                self._dict[i] = e  
  
    def _check_bounds(self, i):  
        if i < 0 or i >= self.n:  
            raise IndexError('Out of bounds')
```

```
def set(self, i, val):
    self._check_bounds(i)
    if val != 0:
        self._dict[i] = val
    return
elif i in self._dict:
    del self._dict[i]

def get(self, i):
    self._check_bounds(i)
    return self._dict.get(i, 0)
```

This will use as much space as there are non-zero elements in the array.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #135

## Problem

This question was asked by Apple.

Given a binary tree, find a minimum path sum from root to a leaf.

For example, the minimum path in this tree is [10, 5, 1, -1], which has sum 15.

```
10
/
5   5
 \
2   1
 /
-1
```

## Solution

This question can be solved using [structural induction](#).

- Assuming that we can find the `min_sum_path` of `node.left` and `node.right`, how could we use these results for `node`?
- We can pick one of `min_sum_path(node.left)` `min_sum_path(node.right)`, whichever has the min sum, and add `node` to the end of that list.
- For the base case, we simply return `[]` since no such path exists.

Since the above code returns the list in reversed order, we can create a wrapper method that can reverse this list to make it in proper order:

```
def min_sum_path(node):
    return list(reversed(_min_sum_path(node)))
```

```
def _min_sum_path(node):
    if node:
        left_list = _min_sum_path(node.left)
        right_list = _min_sum_path(node.right)

        min_list = min(left_list, right_list, key=lambda lst: sum(node.val for node in lst))
        min_list.append(node)

    return min_list

return []
```

This code takes  $O(N)$  time since it traverses the whole tree but takes  $O(K)$  memory where  $K$  is the height of the tree.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #136

## Problem

This question was asked by Google.

Given an  $N$  by  $M$  matrix consisting only of 1's and 0's, find the largest rectangle containing only 1's and return its area.

For example, given the following matrix:

```
[[1, 0, 0, 0],  
 [1, 0, 1, 1],  
 [1, 0, 1, 1],  
 [0, 1, 0, 0]]
```

Return 4.

## Solution

The brute force method for solving this problem would involve enumerating every possible subrectangle in the matrix, checking if they are all 1s and summing them up if they are, keeping track of the largest subrectangle of 1s so far. This would take constant space but  $O(M^3N^3)$  time, since we have to enumerate every subrectangle, which is  $O(N^2M^2)$ , and for each subrectangle we may have to sum up all the area inside it, which is another  $O(MN)$ .

The code for the brute force method would look like this:

```
def is_valid(matrix, top_left_row, top_left_col, bottom_right_row, bottom_right_col):  
    for i in range(top_left_row, bottom_right_row):  
        for j in range(top_left_col, bottom_right_col):  
            if matrix[i][j] == 0:  
                return False  
    return True  
  
def area(top_left_row, top_left_col, bottom_right_row, bottom_right_col):
```

```

    return (bottom_right_row - top_left_row) * (bottom_right_col - top_left_col)

def largest_rectangle(matrix):
    n, m = len(matrix), len(matrix[0])
    max_so_far = 0
    for top_left_row in range(n):
        for top_left_col in range(m):
            for bottom_right_row in range(n, top_left_row, -1):
                for bottom_right_col in range(m, top_left_col, -1):
                    if is_valid(
                            matrix,
                            top_left_row,
                            top_left_col,
                            bottom_right_row,
                            bottom_right_col):
                        max_so_far = max(
                            max_so_far,
                            area(
                                top_left_row,
                                top_left_col,
                                bottom_right_row,
                                bottom_right_col))
    return max_so_far

```

To improve the runtime, we can actually only look at the whole matrix once. If we keep a row in our cache, then we can keep incrementing the elements in our row for each column if we encounter a 1, or reset it to 0 if we encounter a 0. Then we can infer the largest subrectangle we've seen so far, and keep track of the largest.

Let's go through an example:

```

101
011
111

```

For this example, our cache starts off as [0, 0, 0]. After we read in the first row, it becomes [1, 0, 1]. The largest subrectangle here is either at column 0 or 2, with an area of 1. When we read in the second row, we'll reset to zero each column we see a 0 in and increment each column we see a 1 in, so it'll be [0, 1, 2].

Now we know that we've seen one 1 in the middle column and two 1s in a row in the last column, so we can infer that the largest subrectangle of 1s so far has area 2.

After reading the third row, we get [1, 2, 3], and again infer that the largest area is  $2 * 2 = 4$ .

How can we infer the area? We're given a row of heights, similar to a histogram, and we want the largest rectangle we can make given these values. We can do this using brute force: calculating the

area of each rectangle starting and ending at each index by multiplying the min height by the width.

The code for this would look like this:

```
def infer_area(cache):
    max_area = 0
    for i in range(len(cache)):
        for j in range(i + 1, len(cache) + 1):
            current_rectangle = min(cache[i:j]) * (j - i)
            max_area = max(max_area, current_rectangle)
    return max_area

def largest_rectangle(matrix):
    n, m = len(matrix), len(matrix[0])
    max_so_far = 0

    cache = [0 for _ in range(m)]
    for row in matrix:
        for i, val in enumerate(row):
            if val == 0:
                cache[i] = 0
            elif val == 1:
                cache[i] += 1
        max_so_far = max(max_so_far, infer_area(cache))

    return max_so_far
```

This would take  $O(MN^3)$  time and  $O(N)$  space, since we keep one row in memory (as the cache), and we run a double for loop over each row to calculate the area.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #137

## Problem

This problem was asked by Amazon.

Implement a bit array.

A bit array is a space efficient array that holds a value of 1 or 0 at each index.

- `init(size)`: initialize the array with `size`
- `set(i, val)`: updates index at `i` with `val` where `val` is either 1 or 0.
- `get(i)`: gets the value at index `i`.

## Solution

Although Python has arbitrary precision, let's assume each Python `int` has 32 bits.

We can use a list of ints to store `size` bits. Since `size` may not be divisible by 32, we find the ceiling of `size / 32` to figure out the number of integers we need in the list.

Using the above strategy, we would store the first 32 bits in `list[0]`, the next 32 bits in `list[1]`, and so forth.

For the `get(i)` method, we first find the index in the list that holds the  $i^{\text{th}}$  bit, which is at `i / BITS_PER_INT`. Then we find the offset in the integer which holds the bit, which is at `i % BITS_PER_INT`. After these steps, we can extract the bit like so: `(self._list[list_idx] >> int_idx) & 1`

As for `set(i, val)`, we use the following formula:

```
self._list[list_idx] ^= (-val ^ self._list[list_idx]) & (1 << int_idx)
```

The above formula works since if `val` is 0, then the  $i^{\text{th}}$  bit ends up getting xor'd by itself, resulting in 0. If `val` is 1, then, the  $i^{\text{th}}$  bit gets xor'd by itself again, which results in 1.

```
import math

BITS_PER_INT = 32

class BitArray(object):
    def __init__(self, size):
        self._list = [0] * int(math.ceil(size / float(BITS_PER_INT)))
        self._size = size

    def get(self, i):
        if i < 0 or i >= self._size:
            raise IndexError('Index out of bounds')

        list_idx = i / BITS_PER_INT
        int_idx = i % BITS_PER_INT

        return (self._list[list_idx] >> int_idx) & 1

    def set(self, i, val):
        if i < 0 or i >= self._size:
            raise IndexError('Index out of bounds')

        list_idx = i / BITS_PER_INT
        int_idx = i % BITS_PER_INT

        self._list[list_idx] ^= (-val ^ self._list[list_idx]) & (1 << int_idx)
```

get(i) and set(i) takes O(1) and the data structure takes O(N) space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #138

## Problem

This problem was asked by Google.

Find the minimum number of coins required to make n cents.

You can use standard American denominations, that is, 1¢, 5¢, 10¢, and 25¢.

For example, given n = 16, return 3 since we can make it with a 10¢, a 5¢, and a 1¢.

## Solution

If we think about this problem recursively, we can assume we already have a `minimum_coin(n)` function and do the following:

- Calculate the minimum number of coins to make n - denomination for each denomination if it's valid
- Return the min of these, plus one (to account for adding that coin).

Then, we can figure out the base cases:

- If n is 0, then we can make that with 0 coins.
- If n is one of the denominations 1, 5, 10, 25, then return 1.

Here's the code:

```
DENOMINATIONS = [1, 5, 10, 25]
```

```
def minimum_coins(n):
    if n == 0:
        return 0
    elif n in DENOMINATIONS:
```

```
    return 1
else:
    return min(1 + minimum_coins(n - d) for d in DENOMINATIONS if n - d >= 0)
```

This will run really slowly -- in exponential time, since for each call we're making up to 4 calls recursively and only decrementing our input by a constant value. To improve performance, we can use dynamic programming to keep an of all of our previous results. Now we can just look up values in the array instead of repeating subcomputations.

```
DENOMINATIONS = [1, 5, 10, 25]

def minimum_coins_dp(n):
    cache = [0 for _ in range(n + 1)]

    for d in DENOMINATIONS:
        if d < len(cache):
            cache[d] = 1

    for i in range(1, n + 1):
        cache[i] = min(1 + cache[i - d] for d in DENOMINATIONS if i - d >= 0)

    return cache[n]
```

Now this runs in  $O(n)$  time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #139

## Problem

This problem was asked by Google.

Given an iterator with methods `next()` and `hasNext()`, create a wrapper iterator, `.PeekableInterface`, which also implements `peek()`. `peek` shows the next element that would be returned on `next()`.

Here is the interface:

```
class PeekableInterface(object):
    def __init__(self, iterator):
        pass

    def peek(self):
        pass

    def next(self):
        pass

    def hasNext(self):
        pass
```

## Solution

This problem can be solved by storing an instance variable `_next` in our class that holds the following invariant:

`_next` always holds the next item that would be returned on `next()`.

We can follow this by setting `_next` to `next(self.iterator)` in the constructor and then updating on each `next()` call.

Using this invariant, we can then implement `peek` by simply returning `_next` and we can

implement `hasNext` by checking that it's not `None`.

```
class PeekableInterface(object):  
    def __init__(self, iterator):  
        self.iterator = iterator  
        self._next = next(self.iterator)  
  
    def peek(self):  
        return self._next  
  
    def next(self):  
        result = self._next  
        self._next = next(self.iterator)  
        return result  
  
    def hasNext(self):  
        return self._next is not None
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #142

## Problem

This problem was asked by Google.

You're given a string consisting solely of (, ), and \*. \* can represent either a (, ), or an empty string. Determine whether the parentheses are balanced.

For example, ((())\* and (\*) are balanced. )\*( is not balanced.

## Solution

The brute force method here would be to keep a count of current open parentheses that gets incremented every time we see ( and decremented every time we see ). When we encounter a \*, we'll recursively try every path (so with count, count - 1, and count + 1) to see if we can get a valid string.

```
def balanced(s, count=0):
    if not s and count == 0:
        return True

    c = count
    for i, char in enumerate(s):
        if char == '(':
            c += 1
        elif char == ')':
            c -= 1
        elif char == '*':
            return balanced(s[i + 1:], c) or balanced(s[i + 1:], c + 1) or balanced(s[i + 1:], c - 1)

        if c < 0:
            return False
```

```
return c == 0
```

This takes exponential time though since we're recursively calling `balanced` 3 times on each call.

We can use a faster trick, though. Notice from the brute force solution that `*` basically means a range from `c - 1` to `c + 1`. We can keep just two variables, `low` and `high`. We'll maintain the following invariants:

- `low` represents the minimum number of unbalanced open parentheses
- `high` represents the maximum number of unbalanced open parentheses

where `low` and `high` will diverge when we encounter an `*`. We'll keep track of these counts by doing the following:

- If we encounter `(`, then we increment both `low` and `high` - they both contribute to the counts of unbalanced open parentheses.
- If we encounter `)`, then we decrement both `low` and `high` - we have one less unbalanced open parenthesis.
- If we encounter `*`, then we decrement `low` but increment `high`, since it could mean either a `)` or `(`.

```
def balanced(s):
    low = 0
    high = 0
    for char in s:
        if char == '(':
            low += 1
            high += 1
        elif char == ')':
            low = max(low - 1, 0)
            high -= 1
        elif char == '*':
            low = max(low - 1, 0)
            high += 1

        if high < 0:
            return False
    return low == 0
```

This will take only  $O(1)$  space and  $O(n)$  time.

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #143

## Problem

This problem was asked by Amazon.

Given a pivot  $x$ , and a list  $lst$ , partition the list into three parts.

- The first part contains all elements in  $lst$  that are less than  $x$
- The second part contains all elements in  $lst$  that are equal to  $x$
- The third part contains all elements in  $lst$  that are larger than  $x$

Ordering within a part can be arbitrary.

For example, given  $x = 10$  and  $lst = [9, 12, 3, 5, 14, 10, 10]$ , one partition may be  $[9, 3, 5, 10, 10, 12, 14]$ .

## Solution

This question has a relatively simple  $O(1)$  space and  $O(n)$  time solution involving few passes.

- In the first pass, put all elements in  $lst < x$  to the front
- In the second pass, put all elements in  $lst > x$  to the end

One way to do it in one pass is to keep three variables,  $i$ ,  $j$ , and  $k$ , with these invariants:

- All elements in  $lst[:i]$  are less than  $x$
- All elements in  $lst[i:j]$  are equal to  $x$
- All elements in  $lst[k+1:]$  are greater than  $x$

Then we iterate with  $j$  and put  $lst[j]$  according to the above invariants.

```
def partition(lst, x):  
    i = 0  
    j = 0  
    k = len(lst) - 1  
  
    while j < k:  
        if lst[j] == x:  
            j += 1  
        elif lst[j] < x:  
            lst[i], lst[j] = lst[j], lst[i]  
            i += 1  
            j += 1  
        else:  
            lst[j], lst[k] = lst[k], lst[j]  
            k -= 1  
  
    return lst
```

This will take only  $O(1)$  space and  $O(n)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #144

## Problem

This problem was asked by Google.

Given an array of numbers and an index  $i$ , return the index of the nearest larger number of the number at index  $i$ , where distance is measured in array indices.

For example, given  $[4, 1, 3, 5, 6]$  and index  $0$ , you should return  $3$ .

If two distances to larger numbers are the equal, then return any one of them. If the array at  $i$  doesn't have a nearest larger integer, then return null.

Follow-up: If you can preprocess the array, can you do this in constant time?

## Solution

We can do this the brute force method by simply iterating over the range of the array, and trying each offset in both directions until we find a number larger than  $\text{arr}[i]$ :

```
def nearest(arr, i):
    for j in range(1, len(arr)):
        low = i - j
        high = i + j
        if 0 <= low and arr[low] > arr[i]:
            return low
        if high < len(arr) and arr[high] > arr[i]:
            return high
```

This takes  $O(1)$  space but  $O(n)$  time, since we may need to iterate over the whole array.

To speed up the querying, we can preprocess the array to build a lookup array. We'll store the closest index for each corresponding index in the array. To build it fast, we can simply look at pairwise elements along the array to see which one's bigger, and set the index of the lookup array appropriately.

However, there may be cases where we end up with nulls, such as with our example input:

[4, 1, 3, 5, 6] would become [null, 2, 3, 4, null] -- the first null shouldn't be there!

We'll fallback to our original method to fill in the nulls:

```
def _nearest(arr, i):
    for j in range(1, len(arr)):
        low = i - j
        high = i + j
        if 0 <= low and arr[low] > arr[i]:
            return low
        if high < len(arr) and arr[high] > arr[i]:
            return high

def preprocess(arr):
    cache = [None for _ in range(len(arr))]

    for j in range(len(arr) - 1):
        if arr[j] > arr[j + 1]:
            cache[j + 1] = j
        elif arr[j + 1] > arr[j]:
            cache[j] = j + 1

    for i, val in enumerate(cache):
        if val is None:
            cache[i] = _nearest(arr, i)

    return cache
```

Now we can call `preprocess(arr)` on our array and query the index at `i`.

If the elements in the original array are distinct, then this will take linear time. Otherwise, if there are duplicates, then it could take  $O(n^2)$  time in the worst case.

# Daily Coding Problem #145

## Problem

This problem was asked by Google.

Given the head of a singly linked list, swap every two nodes and return its head.

For example, given  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ , return  $2 \rightarrow 1 \rightarrow 4 \rightarrow 3$ .

## Solution

This problem can be solved either recursively or iteratively, and either by mutating next pointers or val values. The easiest implementation which yields the best space and time complexity is iteratively changing val.

In the following code, we maintain and see these properties:

- curr holds the current node that needs to be swapped with its next node.
- If curr.next is None, then there isn't anything to be swapped with, so the loop exits.
- In the loop, curr is able to jump to curr.next.next since we know curr.next exists.

```
def swap_every_two(node):
    curr = node

    while curr and curr.next:
        curr.val, curr.next.val = curr.next.val, curr.val
        curr = curr.next.next

    return node
```

This takes  $O(1)$  space and  $O(n)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #146

## Problem

This question was asked by BufferBox.

Given a binary tree where all nodes are either 0 or 1, prune the tree so that subtrees containing all 0s are removed.

For example, given the following tree:

```
0
/
1  0
 / \
1   0
 / \
0   0
```

should be pruned to:

```
0
/
1  0
 /
1
```

We do not remove the tree at the root or its left child because it still has a 1 as a descendant.

## Solution

If we think about prune recursively and assume that it works, then we can run prune on our input tree's left and right child and it should get rid of all subtrees that are wholly 0s. By that logic, if there is still a left or right child then this the current tree cannot be wholly 0s. So the only remaining cases are:

- When the root itself is null, just return null
- When the root's value is 0 and it is a leaf, then it should return null

Remember to prune the tree first before checking its subtrees.

```
def prune(root):  
    if root is None:  
        return root  
  
    root.left, root.right = prune(root.left), prune(root.right)  
  
    if root.left is None and root.right is None and root.val == 0:  
        return None  
  
    return root
```

This takes  $O(n)$  time and  $O(h)$  space, since we traverse the entire tree.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #147

## Problem

Given a list, sort it using this method: `reverse(lst, i, j)`, which sorts `lst` from `i` to `j`.

## Solution

This type of sorting is called [pancake sorting](#) and can be solved in a similar way as selection sort.

We iteratively put the maximum element to the end of the list using this strategy:

- First, let `size` be the size of the list that we're concerned with sorting at the moment.
- Then, we can find the position where the maximum element is in `list[:size + 1]`, say `max_ind`.
- Then, reverse the sublist from 0 to `max_ind` to put the element at the front.
- Then, reverse the sublist from 0 to `size` to put the max element to the end.
- Decrement `size` and repeat, until `size` is 0.

```
def pancake_sort(lst):  
    for size in reversed(range(len(lst))):  
        max_ind = max_pos(lst[:size + 1])  
        reverse(lst, 0, max_ind)  
        reverse(lst, 0, size)  
    return lst
```

```
def max_pos(lst):  
    return lst.index(max(lst))
```

```
def reverse(lst, i, j):  
    while i < j:  
        lst[i], lst[j] = lst[j], lst[i]  
  
        i += 1
```

j -= 1

This takes  $O(n^2)$  time and  $O(1)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #148

## Problem

This problem was asked by Apple.

**Gray code** is a binary code where each successive value differ in only one bit, as well as when wrapping around. Gray code is common in hardware so that we don't see temporary spurious values during transitions.

Given a number of bits  $n$ , generate a possible gray code for it.

For example, for  $n = 2$ , one gray code would be  $[00, 01, 11, 10]$ .

## Solution

We can solve this problem recursively—given that we have a gray code for  $n - 1$ , how can we extend that solution to work with  $n$  bits?

Let's take the base case, the solution for  $n = 1$ :  $[0, 1]$ .

Now, for  $n = 2$ , we obviously need to extend the solution by adding an extra bit. Notice that if we simply duplicate the solution twice, we get close to a solution:

$[00, 01, 10, 11]$

The only parts that don't work here are at the edges of where we're duplicating. Instead, if we duplicated the solution, prepend  $0$  to one copy, and prepend  $1$  to the other and reverse it, we get one where the edges match up:

$[00, 01, 11, 10]$

This works because between the copies, the only differences are at the most significant bit. We want to exploit this symmetry and flip the second copy so that at the edges, the only thing that changes is that significant bit.

```
def gray_code(n):  
    if n == 0:
```

```
return []

elif n == 1:
    return [0, 1]

prev_gray_code = gray_code(n - 1)

result = []
for code in prev_gray_code:
    result.append(code)

for code in reversed(prev_gray_code):
    result.append((1 << n - 1) + code)

return result
```

This takes  $O(2^n)$  time, since we're generating twice as many results for each recursive call and each recursive call reduces the input by a constant amount.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #149

## Problem

This problem was asked by Goldman Sachs.

Given a list of numbers  $L$ , implement a method  $\text{sum}(i, j)$  which returns the sum from the sublist  $L[i:j]$  (including  $i$ , excluding  $j$ ).

For example, given  $L = [1, 2, 3, 4, 5]$ ,  $\text{sum}(1, 3)$  should return  $\text{sum}([2, 3])$ , which is 5.

You can assume that you can do some pre-processing.  $\text{sum}()$  should be optimized over the pre-processing step.

## Solution

This problem has many different possible tradeoffs you could make:

- Have no pre-processing step and compute  $\text{sum}$  on the fly.
- Pre-process the sum of every  $i, j$  combination s.t.  $i < j$  and store it in a list.
- Create a [segment tree](#) representing sums of many segments of the list.

Let's explore the segment tree since it has a good balance of pre-processing time and query time.

Basically, a segment tree's nodes would have these fields:

- `start_ind` and `end_ind` which represents the segment in the list this node represents.
- `val`, which represents the sum of this segment.
- `left`, for the left node.
- `right`, for the right node.

With these fields we can see how we might query for a segment.

- If the node's segment is equivalent to the query, return its `val`

- Otherwise, query for the left node if it encloses the starting index and add it to our result.
- Also query for the right node if it encloses the ending index and add it to our result.

```
result = 0

if left.start_ind <= start_ind <= left.end_ind:
    result += query(left, start_ind, min(end_ind, left.end_ind))

if right.start_ind <= end_ind <= right.end_ind:
    result += query(right, max(start_ind, right.start_ind), end_ind)
```

The full code:

```
class Node:
    def __init__(self, val, start_ind, end_ind, left=None, right=None):
        self.val = val
        self.start_ind = start_ind
        self.end_ind = end_ind
        self.left = left
        self.right = right

    @property
    def interval(self):
        return (self.start_ind, self.end_ind)

def make_segment_tree(lst):
    return _make_segment_tree(lst, 0, len(lst) - 1)

def _make_segment_tree(lst, start_ind, end_ind):
    if start_ind == end_ind:
        assert(len(lst) == 1)
        val = lst[0]
        return Node(val, start_ind, end_ind)

    mid = len(lst) // 2

    left = _make_segment_tree(lst[:mid], start_ind, start_ind + mid - 1)
    right = _make_segment_tree(lst[mid:], start_ind + mid, end_ind)

    root_val = left.val + right.val

    return Node(root_val, start_ind, end_ind, left, right)
```

```
def query(node, start_ind, end_ind):
    if node.start_ind == start_ind and node.end_ind == end_ind:
        return node.val

    result = 0
    left = node.left
    right = node.right

    if left.start_ind <= start_ind <= left.end_ind:
        result += query(left, start_ind, min(end_ind, left.end_ind))

    if right.start_ind <= end_ind <= right.end_ind:
        result += query(right, max(start_ind, right.start_ind), end_ind)

    return result
```

This takes  $O(N \log N)$  time and space during pre-processing while querying takes  $O(\log N)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #150

## Problem

This problem was asked by LinkedIn.

Given a list of points, a central point, and an integer k, find the nearest k points from the central point.

For example, given the list of points  $[(0, 0), (5, 4), (3, 1)]$ , the central point  $(1, 2)$ , and  $k = 2$ , return  $[(0, 0), (3, 1)]$ .

## Solution

One strategy to solve this would use a max-heap or a priority queue. We store the first k elements in the queue with their distances from the center point as their priority. After that, we go through the rest of the points and compare their distances with the max priority of the queue/heap. If the current point is closer, then pop that max element from the queue/heap and add in the current one.

```
import math

def nearest_k_points(points, center, k):
    if len(points) <= k:
        return points

    pq = PriorityQueue()

    for point in points:
        if len(pq) < k:
            pq.push(point, distance(point, center))
        else:
            dist = distance(point, center)
            if dist < pq.peek()[1]:
                pq.pop()
                pq.push(point, dist)

    return [pq.pop() for _ in range(k)]
```

```

    pq.push(point, dist)

return [pq.pop()[0] for i in range(k)]
```

def distance(p1, p2):  
 x1, y1 = p1  
 x2, y2 = p2  
 return math.sqrt((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2)

Assuming the priority queue uses a heap, this will take  $O(n \log k)$  time and  $O(k)$  space.

An implementation of a priority queue using a heap follows:

```

class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, ele, priority):
        self.heap.append((priority, ele))
        self._bubble_up(len(self.heap) - 1)

    def pop(self):
        if self.is_empty():
            raise IndexError("get from empty heap")

        self._swap(0, len(self.heap) - 1)
        priority, ele = self._pop()
        self._bubble_down(0)

        return (ele, priority)

    def peek(self):
        priority, ele = self.heap[0]
        return (ele, priority)

    def is_empty(self):
        return not self.heap

    def _bubble_down(self, ind):
        length = len(self.heap)
        heap = self.heap

        while True:
            lc, rc = self._left_child(ind), self._right_child(ind)
            if lc >= length and rc >= length:
                break
```

```
        elif lc >= length:
            replace = rc
        elif rc >= length:
            replace = lc
        else:
            replace = min(lc, rc, key=lambda i: self.heap[i])

    if heap[replace] > heap[ind]:
        self._swap(ind, replace)
        ind = replace
    else:
        break

def _bubble_up(self, ind):
    par = self._parent(ind)
    heap = self.heap

    while par >= 0:
        if heap[ind] > heap[par]:
            self._swap(ind, par)
            ind = par
            par = self._parent(ind)
        else:
            break

def _parent(self, ind):
    return (ind - 1) // 2

def _left_child(self, ind):
    return (ind * 2) + 1

def _right_child(self, ind):
    return (ind * 2) + 2

def __len__(self):
    return len(self.heap)

def _swap(self, i, j):
    _, item0 = self.heap[i]
    _, item1 = self.heap[j]

    self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

def _pop(self):
    priority, ele = self.heap.pop()
    return (priority, ele)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #151

## Problem

Given a 2-D matrix representing an image, a location of a pixel in the screen and a color C, replace the color of the given pixel and all adjacent same colored pixels with C.

For example, given the following matrix, and location pixel of (2, 2), and 'G' for green:

```
B B W  
W W W  
W W W  
B B B
```

Becomes

```
B B G  
G G G  
G G G  
B B B
```

## Solution

A simplistic strategy to floodfill is to use [depth-first-search](#). The algorithm works as follows:

First, Fill the current coord to color. Mark as visited. For each neighboring new\_coord, if it hasn't been visited, is inside the matrix, and is the same color as current coord color, recursively floodfill that coordinate.

```
def floodfill(matrix, coord, color, visited=None):  
    if visited is None:  
        visited = set()  
  
    visited.add(coord)
```

```

r, c = coord
prior_color = matrix[r][c]
matrix[r][c] = color

coords = [(r + 1, c), (r, c + 1), (r - 1, c), (r, c - 1)]

for new_coord in coords:
    new_r, new_c = new_coord
    if (new_coord not in visited
        and in_matrix(matrix, new_coord)
        and matrix[new_r][new_c] == prior_color):

        visited.add(new_coord)
        floodfill(matrix, new_coord, color, visited)

def in_matrix(matrix, coord):
    rows = len(matrix)
    cols = len(matrix[0])

    r, c = coord

    return 0 <= r < rows and 0 <= c < cols

```

This will take  $O(V + E)$  time and  $O(V)$  space. In other words, it will take time proportional to the size of the matrix.

Another way to do this problem would be to use breadth first search, using a queue. This would also yield the same complexities.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #153

## Problem

Find an efficient algorithm to find the smallest distance (measured in number of words) between any two given words in a string.

For example, given words "hello", and "world" and a text content of "dog cat hello cat dog dog hello cat world", return 1 because there's only one word "cat" in between the two words.

## Solution

We can translate this problem into a more algorithmic format. We can first find all the indices of `word0` and `word1` in `text`. Then with the two indices lists, we have the problem of finding a number from each of the lists that minimizes their difference.

For example, given [1, 10, 33] and [5, 6, 15, 32], the two numbers would be 33 and 32.

To solve this problem, we can use a greedy strategy. We keep two pointers `i`, for the `word0` indices, and `j`, for the `word1` indices. Then we explore different `i` and `j` while keeping the minimum distance we've seen of their values.

For example, this is the initial state:

[1, 10, 33] ^ i

[5, 6, 15, 32] ^ j

And the minimum distance would begin with `abs(5 - 1) == 4`. Then we iterate until `i` or `j` is out of index: if the value indexed by `i`, `word0_indices[i]`, is lower than at `j`, we increment `i`, and otherwise increment `j`.

This process must work since the optimal solution must make the value of the lower number higher in order to minimize the difference.

```
text_words = [w.strip() for w in text.split(' ')]

print text_words

word0_indices = [i for i, w in enumerate(text_words) if w == word0]
word1_indices = [i for i, w in enumerate(text_words) if w == word1]

if not word0_indices or not word1_indices: # one of the words doesn't exist.
    return float('inf')

i = j = 0

min_distance = abs(word0_indices[i] - word1_indices[j])

while i < len(word0_indices) and j < len(word1_indices):

    current_distance = abs(word0_indices[i] - word1_indices[j])
    min_distance = min(min_distance, current_distance)

    if word0_indices[i] < word1_indices[j]:
        i += 1
    else:
        j += 1

return min_distance - 1 # Don't count the last step to get to word1
```

This takes O(n) space and time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #154

## Problem

This problem was asked by Amazon.

Implement a stack API using only a heap. A stack implements the following methods:

- `push(item)`, which adds an element to the stack
- `pop()`, which removes and returns the most recently added element (or throws an error if there is nothing on the stack)

Recall that a heap has the following operations:

- `push(item)`, which adds a new key to the heap
- `pop()`, which removes and returns the max value of the heap

## Solution

One way to solve this problem is to store a timestamp as the keys to a max-heap. This way, the most recently added item will always be at the top, and we can extract it by calling `pop()` on the heap. This should cause the heap to bubble-down and bring the next most recently added item to the top.

```
from time import time

class Stack:
    def __init__(self):
        self.max_heap = MaxHeap()

    def push(self, item):
        t = time()
        self.max_heap.push(item, t)
```

```
def pop(self):
    item, _ = self.max_heap.pop()
    return item

import heapq

class MaxHeap:
    def __init__(self):
        self._heap = []

    def push(self, item, priority):
        heapq.heappush(self._heap, (-priority, item))

    def pop(self):
        _, item = heapq.heappop(self._heap)
        return item
```

The stack operations will take  $O(\log n)$  time and  $O(n)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #155

## Problem

This problem was asked by MongoDB.

Given a list of elements, find the majority element, which appears more than half the time ( $> \text{floor}(\text{len}(1st) / 2.0)$ ).

You can assume that such element exists.

For example, given [1, 2, 1, 1, 3, 4, 0], return 1.

## Solution

One way to solve this problem is to simply use a hashmap to store the mapping of an element to its frequency. Each insertion would take O(1) for a total of O(n) time. At the end of the loop, we could query for the element which has the highest frequency:

```
def majority(elements):
    element_to_count = {}
    for element in elements:
        if element not in element_to_count:
            element_to_count[element] = 0
        element_to_count[element] += 1
    # Find the element with most count
    return max(element_to_count, key=element_to_count.get)
```

Another way is to use a voting strategy. We keep the current majority element as well as their frequency during the loop. When we see an element that's the same as the majority, we vote or increment the frequency. Otherwise, we decrement it.

```
def majority(elements):
    for i, e in enumerate(elements):
        if i == 0 or count == 0:
            current_element = e
            count = 1
        else:
            if current_element == e:
                count += 1
            else:
                count -= 1
    return current_element
```

```
majority = e
count = 1
elif majority == e:
    count += 1
else:
    count -= 1
return majority
```

We know the majority algorithm exists, so we must increment more than we decrement. Since the final count must be positive, we must have found the majority element.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #156

## Problem

This problem was asked by Facebook.

Given a positive integer  $n$ , find the smallest number of squared integers which sum to  $n$ .

For example, given  $n = 13$ , return 2 since  $13 = 3^2 + 2^2 = 9 + 4$ .

Given  $n = 27$ , return 3 since  $27 = 3^2 + 3^2 + 3^2 = 9 + 9 + 9$ .

## Solution

One naive recursive way of solving this problem would be to do the following:

- Iterate  $i$  from 1 to  $\text{sqrt}(n)$
- Recursively compute the minimum number of squares needed to sum to  $n - i*i$
- Pick the min of those, plus 1

The base case would be when  $n = 0$ .

```
from math import inf

def num_squares_naive(n):
    if n == 0:
        return 0

    min_num_squares = inf

    i = 1
    while n - i*i >= 0:
        min_num_squares = min(min_num_squares, num_squares_naive(n - i*i) + 1)
        i += 1
```

```
return min_num_squares
```

However, this takes exponential time. We can speed things up using a cache with dynamic programming, and using the same logic:

```
def num_squares(n):
    if n == 0:
        return 0

    cache = [inf for i in range(n + 1)]
    cache[0] = 0
    for i in range(1, n + 1):
        j = 1
        while j * j <= i:
            cache[i] = min(cache[i], cache[i - j*j] + 1)
            j += 1

    return cache[n]
```

Now this is  $O(n^2)$  time and  $O(n)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #157

## Problem

This problem was asked by Amazon.

Given a string, determine whether any permutation of it is a palindrome.

For example, `carrace` should return true, since it can be rearranged to form `racecar`, which is a palindrome. `daily` should return false, since there's no rearrangement that can form a palindrome.

## Solution

The brute force solution would be to try every permutation, and verify if it's a palindrome. If we find one, then return true, otherwise return false.

The number of possible permutations is on the order of  $n!$ , so computing all of them and checking their palindromicity would be prohibitively expensive. Instead, notice that in a palindrome each character must be matched by the other side, except at the middle. That means that if we count up all the characters in our input string, at most one can have an odd count.

```
from collections import Counter

def is_permutation_palindrome(s):
    c = Counter(s)

    num_odds = 0 # Number of characters that have an odd count.

    for char, count in c.items():
        if count % 2 != 0:
            num_odds += 1

    return num_odds <= 1
```

This takes  $O(n)$  time and space. We can reduce this to constant space at the expense of higher total memory usage by just keeping one array that maps each index to its corresponding ASCII

character. Then we can even do this in a single pass as we iterate over the string and keep track of the odd counts:

```
def is_permutation_palindrome(s):
    # There are 128 ASCII characters
    arr = [0 for _ in range(128)]

    num_odds = 0
    for char in s:
        i = ord(char)
        arr[i] += 1

        if arr[i] % 2 != 0:
            num_odds += 1
        else:
            num_odds -= 1

    return num_odds <= 1
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #158

## Problem

This problem was asked by Slack.

You are given an N by M matrix of 0s and 1s. Starting from the top left corner, how many ways are there to reach the bottom right corner?

You can only move right and down. 0 represents an empty space while 1 represents a wall you cannot walk through.

For example, given the following matrix:

```
[[0, 0, 1],  
 [0, 0, 1],  
 [1, 0, 0]]
```

Return two, as there are only two ways to get to the bottom right:

- Right, down, down, right
- Down, right, down, right

The top left corner and bottom right corner will always be 0.

## Solution

This problem is similar to Daily Coding Problem #62, but with a twist: there are now obstacles in the original matrix that we can't go through. The basic idea to solve this is the following:

- Notice again that we can only get to any cell from above or from the left.
- We'll store the number of ways to get to a cell  $[i][j]$ , and compute that from looking at  $[i - 1][j]$  and  $[i][j - 1]$ .

- First though, we must check that those spots are not walls, since we can't move from there. If it is, then the number of ways coming from that spot is 0.

We also set the first row and column to be 1 (since we can move straight down or right), but we have to stop when we hit the first wall.

```
EMPTY = 0
WALL = 1

def num_ways(matrix):
    m, n = len(matrix), len(matrix[0])
    num_ways_matrix = [[0 for j in range(n)] for i in range(m)]

    # Fill first row
    for j in range(n):
        if matrix[0][j] == WALL:
            break
        num_ways_matrix[0][j] = 1

    # Fill first col
    for i in range(m):
        if matrix[i][0] == WALL:
            break
        num_ways_matrix[i][0] = 1

    for i in range(1, m):
        for j in range(1, n):
            from_top = num_ways_matrix[i - 1][j] if matrix[i - 1][j] != WALL else 0
            from_left = num_ways_matrix[i][j - 1] if matrix[i][j - 1] != WALL else 0

            num_ways_matrix[i][j] = from_top + from_left

    return num_ways_matrix[m - 1][n - 1]
```

This takes  $O(n * m)$  time and space.



# Daily Coding Problem #159

## Problem

This problem was asked by Google.

Given a string, return the first recurring character in it, or null if there is no recurring character.

For example, given the string "acbbac", return "b". Given the string "abcdef", return null.

## Solution

We can solve this problem by keeping track of the characters we've seen in a set. Once we encounter a character we've seen already, we can return it.

```
def first_recurring(s):
    seen = set()
    for char in s:
        if char in seen:
            return char
        seen.add(char)
    return None
```

This runs in  $O(n)$  time and space, since we have to iterate over the string.

We can improve the space complexity by using bitwise operators to set the bits of characters already seen in the string:

```
def first_recurring(str):
    checker = 0
    for c in str:
        val = ord(c) - ord('a')
        if (checker & (1 << val)) > 0:
            return c
        checker |= (1 << val)
    return None
```

Thanks to Shantanu for this solution.

---

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #160

## Problem

This problem was asked by Uber.

Given a tree where each edge has a weight, compute the length of the longest path in the tree.

For example, given the following tree:



and the weights: a-b: 3, a-c: 5, a-d: 8, d-e: 2, d-f: 4, e-g: 1, e-h: 1, the longest path would be c -> a -> d -> f, with a length of 17.

The path does not have to pass through the root, and each node can have any amount of children.

## Solution

There are two cases: either the longest path goes through the root or it doesn't.

If it doesn't, then we only need to look at the longest path of any of the current node's children. If it does, then the longest path must come from the paths made by combining the two highest childrens' heights.

What we'll do is recursively look at each child's height and longest path, and keep track of the longest path we've seen so far. At the end we select the largest between the longest subpath, or the two largest heights that can be combined to make a new larger path.

The base case here is when we look at a node with no children, in which case the longest path should be 0.

```

from math import inf

class Node:
    def __init__(self, val, children=[]):
        self.val = val
        self.children = children

def longest_path(root):
    height, path = longest_height_and_path(root)
    return path

def longest_height_and_path(root):
    longest_path_so_far = -inf
    highest, second_highest = 0, 0
    for length, child in root.children:
        height, longest_path_length = longest_height_and_path(child)

        longest_path_so_far = max(longest_path_so_far, longest_path_length)

        if height + length > highest:
            highest, second_highest = height + length, highest
        elif height + length > second_highest:
            second_highest = height + length

    return highest, max(longest_path_so_far, highest + second_highest)

```

Since we're looking at every node recursively, this algorithm runs in  $O(n)$  time and  $O(h)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #161

## Problem

This problem was asked by Facebook.

Given a 32-bit integer, return the number with its bits reversed.

For example, given the binary number 1111 0000 1111 0000 1111 0000 1111 0000, return 0000 1111 0000 1111 0000 1111 0000 1111.

## Solution

We can do this by iterating over every bit from 0 to 31, checking whether that bit is on, and then setting that bit on a final result:

```
NUM_BITS = 32

def reverse_bits(n):
    reversed_num = 0
    for i in range(NUM_BITS):
        j = n >> i & 1
        reversed_num += j << (NUM_BITS - i - 1)
    return reversed_num
```

This takes O(n) time where n is the number of bits.

We can also trade off time for space by creating a lookup cache of already reversed bits, let's say 8. Then we only need to iterate at  $32 / 8 = 4$  times if we preprocess the cache.

```
NUM_BITS = 32
NUM_BITS_IN_CACHE = 8

def reverse_naive(n, num_bits):
    reversed_num = 0
```

```

for i in range(num_bits):
    j = n >> i & 1
    reversed_num += j << (num_bits - i - 1)
return reversed_num

def preprocess():
    cache = {}
    for i in range(2 ** NUM_BITS_IN_CACHE):
        cache[i] = reverse_naive(i, NUM_BITS_IN_CACHE)
    return cache

preprocessed = preprocess()

def reverse_bits(n):
    result = 0
    mask = 2 ** NUM_BITS_IN_CACHE - 1
    for i in range(NUM_BITS // NUM_BITS_IN_CACHE):
        relevant = n >> (i * NUM_BITS_IN_CACHE) & mask
        result += preprocessed[relevant] << NUM_BITS - ((i + 1) * NUM_BITS_IN_CACHE)
    return result

```

This will take  $O(2^c)$  time and space to initialize and  $O(n / c)$  time to query, where  $c$  is the number of bits in the cache key.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #162

## Problem

This problem was asked by Square.

Given a list of words, return the shortest unique prefix of each word. For example, given the list:

- dog
- cat
- apple
- apricot
- fish

Return the list:

- d
- c
- app
- apr
- f

## Solution

We can do this by comparing each word to each other and finding the shortest unique prefix for each word. However, this would take  $O(n^2)$  time. We can speed up the process by using a [trie](#) and keeping track of each character's count. Then, after adding each word, we can look for the word in the trie by character until we hit a count of 1 while building the prefix.

```
class Node:  
    def __init__(self, char=None):  
        self.char = char  
        self.children = {}  
  
        self.finished = False  
        self.count = 0  
  
  
class Trie:  
    def __init__(self):  
        self.root = Node()  
  
    def insert(self, word):  
        node = self.root  
        for char in word:  
            if char not in node.children:  
                node.children[char] = Node(char)  
            node.count += 1  
            node = node.children[char]  
        node.finished = True  
  
    def unique_prefix(self, word):  
        node = self.root
```

```
prefix = ''  
for char in word:  
    if node.count == 1:  
        return prefix  
    node = node.children[char]  
    prefix += char  
return prefix  
  
  
def shortest_unique_prefix(lst):  
    trie = Trie()  
    for word in lst:  
        trie.insert(word)  
  
    return [trie.unique_prefix(word) for word in lst]
```

This takes  $O(n)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #163

## Problem

This problem was asked by Jane Street.

Given an arithmetic expression in [Reverse Polish Notation](#), write a program to evaluate it.

The expression is given as a list of numbers and operands. For example: [5, 3, '+'] should return  $5 + 3 = 8$ .

For example, [15, 7, 1, 1, '+', '-', '/', 3, '\*', 2, 1, 1, '+', '+', '-'] should return 5, since it is equivalent to  $((15 / (7 - (1 + 1))) * 3) - (2 + (1 + 1)) = 5$ .

You can assume the given expression is always valid.

## Solution

The way to implement Reverse Polish Notation is to use a stack. When we encounter a value, then we add it to the stack, and if we encounter an operator such as '+', '-', '\*', or '/', then we pop the last two things off the stack, use them as terms on the operator, and then pop the resulting value back on the stack. At the end of the function there should only be one thing remaining on the stack, so we just return that.

```
PLUS = '+'
MINUS = '-'
TIMES = '*'
DIVIDE = '/'

OPERANDS = [PLUS, MINUS, TIMES, DIVIDE]

def rpn(expr):
    stack = []
    for val in expr:
```

```
if val in OPERANDS:  
    term1, term2 = stack.pop(), stack.pop()  
  
    if val == PLUS:  
        stack.append(term1 + term2)  
  
    elif val == MINUS:  
        stack.append(term1 - term2)  
  
    elif val == TIMES:  
        stack.append(term1 * term2)  
  
    elif val == DIVIDE:  
        stack.append(term1 / term2)  
  
    else:  
        stack.append(val)  
  
return stack[0]
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #164

## Problem

This problem was asked by Google.

You are given an array of length  $n + 1$  whose elements belong to the set  $\{1, 2, \dots, n\}$ . By the pigeonhole principle, there must be a duplicate. Find it in linear time and space.

## Solution

One method to solve this is to iterate over the array and look in location  $i$  of the array: if  $lst[i]$  holds  $i$ , then keep going. If  $lst[i]$  holds  $j$ , then swap  $lst[i]$  and  $lst[j]$  and repeat until it's the correct value. If we encounter the same value at  $lst[j]$  then we have found our duplicate.

```
def duplicate(lst):
    i = 0
    while i < len(lst):
        if lst[i] != i:
            j = lst[i]
            if lst[j] == lst[i]:
                return j
            lst[i], lst[j] = lst[j], lst[i]
        else:
            i += 1
    raise IndexError('Malformed input.')
```

This runs in  $O(n)$  time and constant space.

We can also simply sum up all the elements in the array and subtract it by the sum of 1 to  $n$ , using the formulas  $n * (n + 1) / 2$ . We should be left with the duplicate.

```
def duplicate(lst):
    n = len(lst) - 1
    return sum(lst) - (n * (n + 1) // 2)
```

This takes O(n) time and O(1) space.

---

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #165

## Problem

This problem was asked by Google.

Given an array of integers, return a new array where each element in the new array is the number of smaller elements to the right of that element in the original input array.

For example, given the array [3, 4, 9, 6, 1], return [1, 1, 2, 1, 0], since:

- There is 1 smaller element to the right of 3
- There is 1 smaller element to the right of 4
- There are 2 smaller elements to the right of 9
- There is 1 smaller element to the right of 6
- There are no smaller elements to the right of 1

## Solution

A naive solution for this problem would simply be to create a new array, and for each element count all the smallest elements to the right of it.

```
def smaller_counts_naive(lst):
    result = []
    for i, num in enumerate(lst):
        count = sum(val < num for val in lst[i + 1:])
        result.append(count)
    return result
```

This takes  $O(n^2)$  time. Can we do this any faster?

To speed this up, we can try the following idea:

- Iterate backwards over the input list
- Maintain a sorted list  $s$  of the elements we've seen so far
- Look at  $s$  to see where the current element would fit in

The index will be how many elements on the right are smaller.

```
import bisect

def smaller_counts(lst):
    result = []
    s = []
    for num in reversed(lst):
        i = bisect.bisect_left(s, num)
        result.append(i)
        bisect.insort(s, num)
    return list(reversed(result))
```

Now this only takes  $O(n \log n)$  time and  $O(n)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #166

## Problem

This problem was asked by Uber.

Implement a 2D iterator class. It will be initialized with an array of arrays, and should implement the following methods:

- `next()`: returns the next element in the array of arrays. If there are no more elements, raise an exception.
- `has_next()`: returns whether or not the iterator still has elements left.

For example, given the input `[[1, 2], [3], [], [4, 5, 6]]`, calling `next()` repeatedly should output 1, 2, 3, 4, 5, 6.

Do not use `flatten` or otherwise clone the arrays. Some of the arrays can be empty.

## Solution

To minimize the amount of space we use for the 2D iterator, we just need to store two pointers, an outer pointer `i` and an inner pointer `j`. To make the code reusable, we'll implement a private function `_next_coords(i, j)` that gets the coordinates of the next valid element.

Remember that we need to check for the following cases:

- List of lists is empty
- First element hasn't been retrieved yet
- Empty list in beginning, middle, or end
- End of inner list

```
def __init__(self, lsts):
    self._lststs = lststs
    self.i = None
    self.j = None

def _next_coords(self, i, j):
    '''Gets the coordinates of the next valid element.'''
    if not self._lststs or len(self._lststs) == 0:
        return None

    if i is None and j is None:
        i = 0
        while i < len(self._lststs):
            if len(self._lststs[i]) > 0:
                return (i, 0)
            i += 1

    if j + 1 < len(self._lststs[i]):
        return (i, j + 1)

    i += 1
    while i < len(self._lststs):
        if len(self._lststs[i]) > 0:
            return (i, 0)
        i += 1

    return None

def next(self):
    coords = self._next_coords(self.i, self.j)
    if coords is None:
        raise Exception("No more elements")
    else:
        self.i, self.j = coords
        return self._lststs[self.i][self.j]

def has_next(self):
    coords = self._next_coords(self.i, self.j)
    return coords is not None
```

© Daily Coding Problem 2019  
Privacy Policy

Terms of Service

Press

# Daily Coding Problem #167

## Problem

This problem was asked by Airbnb.

Given a list of words, find all pairs of unique indices such that the concatenation of the two words is a palindrome.

For example, given the list ["code", "edoc", "da", "d"], return [(0, 1), (1, 0), (2, 3)].

## Solution

The naive solution here would be to check each possible word pair for palindromicity and add their indices to the result:

```
def is_palindrome(word):
    return word == word[::-1]

def palindrome_pairs(words):
    result = []
    for i, word1 in enumerate(words):
        for j, word2 in enumerate(words):
            if i == j:
                continue
            if is_palindrome(word1 + word2):
                result.append((i, j))
    return result
```

This takes  $O(n^2 * c)$  time where  $n$  is the number of words and  $c$  is the length of the longest word.

To speed it up, we can insert all words into a dictionary or hash table and then check each word's prefixes and postfixes. It will map from a word to its index in the list. If the reverse of a word's

prefix/postfix is in the dictionary and its postfix/prefix is palindromic, then we add it to our list of results. For example, say we're looking at the word aabc. We check all its prefixes:

- Since a is a palindrome, we look for cba in the dictionary. If we find it, then we can make cbaaabc.
- Since aa is a palindrome, we look for cb in the dictionary. If we find it, then we can make cbaabc.
- aab and aabc are not palindromes, so we don't do anything.

And we do the same thing for the postfix.

```
def is_palindrome(word):
    return word == word[::-1]

def palindrome_pairs(words):
    d = {}
    for i, word in enumerate(words):
        d[word] = i

    result = []

    for i, word in enumerate(words):
        for char_i in range(len(word)):
            prefix, postfix = word[:char_i], word[char_i:]

            reversed_prefix, reversed_postfix = prefix[::-1], postfix[::-1]

            if is_palindrome(postfix) and reversed_prefix in d:
                if i != d[reversed_prefix]:
                    result.append((i, d[reversed_prefix]))

            if is_palindrome(prefix) and reversed_postfix in d:
                if i != d[reversed_postfix]:
                    result.append((d[reversed_postfix], i))

    return result
```

This should speed up the time to  $O(n * c^2)$ . Since we will likely be constrained more by the number of words than the number of characters, this seems like an acceptable tradeoff.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #168

## Problem

This problem was asked by Facebook.

Given an N by N matrix, rotate it by 90 degrees clockwise.

For example, given the following matrix:

```
[[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]]
```

you should return:

```
[[7, 4, 1],  
 [8, 5, 2],  
 [9, 6, 3]]
```

Follow-up: What if you couldn't use any extra space?

## Solution

Let's look at where each value should go with a few examples:

Consider the example input matrix:

- (0, 0) should go to (0, 2)
- (0, 1) should go to (1, 2)
- (0, 2) should go to (2, 2)
- (1, 0) should go to (0, 1)
- (1, 1) should go to (1, 1)
- (1, 2) should go to (2, 1)
- (2, 0) should go to (0, 0)
- (2, 1) should go to (1, 1)
- (2, 2) should go to (2, 1)

In general, the rule looks like this:  $[i][j]$  should go to  $[j][n - i - 1]$ . So we can instantiate a new matrix, loop over each element, and assign it to its new location:

```
def rotate_matrix(matrix):  
    n = len(matrix)
```

```

new_matrix = [[None for _ in range(n)] for _ in range(n)]

for r, row in enumerate(matrix):
    for c, val in enumerate(row):
        new_matrix[c][n - r - 1] = val

return new_matrix

```

This takes  $O(n^2)$  time and space. How can we do this without using any extra memory?

It would be hard to perform a rotation by only swapping two elements. Instead, we can look at a value and perform a chain of 4 swaps:

- Top-left with bottom-left
- Top-right with top-left
- Bottom-right with top-right
- Bottom-left with bottom-right

We start with the first row and move down until  $n // 2$ , since the bottom rows should be rotated already by then.

```

def rotate_matrix(matrix):
    n = len(matrix)

    for i in range(n // 2):
        for j in range(i, n - i - 1):
            p1 = matrix[i][j]
            p2 = matrix[j][n - i - 1]
            p3 = matrix[n - i - 1][n - j - 1]
            p4 = matrix[n - j - 1][i]

            matrix[j][n - i - 1] = p1
            matrix[n - i - 1][n - j - 1] = p2

            matrix[n - j - 1][i] = p3
            matrix[i][j] = p4

    return matrix

```

While this still runs in  $O(n^2)$  time, we use no extra space as everything is rotated in-place.

# Daily Coding Problem #170

## Problem

This problem was asked by Facebook.

Given a start word, an end word, and a dictionary of valid words, find the shortest transformation sequence from start to end such that only one letter is changed at each step of the sequence, and each transformed word exists in the dictionary. If there is no possible transformation, return null. Each word in the dictionary have the same length as start and end and is lowercase.

For example, given `start = "dog", end = "cat", and dictionary = {"dot", "dop", "dat", "cat"}`, return `["dog", "dot", "dat", "cat"]`.

Given `start = "dog", end = "cat", and dictionary = {"dot", "tod", "dat", "dar"}`, return null as there is no possible transformation from dog to cat.

## Solution

We can model this problem as a graph: the nodes will be the words in the dictionary, and we can form an edge between two nodes if and only if one character can be modified in one word to get to the other.

Then we can do a typical [breadth-first search](#) starting from `start` and finishing once we encounter `end`:

```
def word_ladder(start, end, words):
    queue = deque([(start, [start]))]

    while queue:
        word, path = queue.popleft()
        if word == end:
            return path
```

```
for i in range(len(word)):  
    for c in ascii_lowercase:  
        next_word = word[:i] + c + word[i + 1:]  
        if next_word in words:  
            words.remove(next_word)  
            queue.append([next_word, path + [next_word]])  
  
return None
```

This takes  $O(n^2)$  time and  $O(n)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #171

## Problem

This problem was asked by Amazon.

You are given a list of data entries that represent entries and exits of groups of people into a building. An entry looks like this:

```
{"timestamp": 1526579928, "count": 3, "type": "enter"}
```

This means 3 people entered the building. An exit looks like this:

```
{"timestamp": 1526580382, "count": 2, "type": "exit"}
```

This means that 2 people exited the building. `timestamp` is in [Unix time](#).

Find the busiest period in the building, that is, the time with the most people in the building. Return it as a pair of (`start`, `end`) timestamps. You can assume the building always starts off and ends up empty, i.e. with 0 people inside.

## Solution

We can solve this by first sorting the entries by the timestamp (assuming they're not already sorted). Then, as we iterate over the sorted entries we can keep track of the current number of people. If we go over the max we've seen so far, then we update the max number of people and the bounds of the period.

```
def busiest_period(entries):
    period = (None, None)
    num_people, max_num_people = 0, 0

    # Sort the entries by timestamp
    sorted_entries = sorted(entries, key=lambda e: e["timestamp"])
```

```
# Keep track of the number of people at each entry.
for i, entry in enumerate(sorted_entries):
    if entry["type"] == "enter":
        num_people += entry["count"]
    else:
        num_people -= entry["count"]

    if num_people > max_num_people:
        max_num_people = num_people
        period = (entry["timestamp"], sorted_entries[i + 1]["timestamp"])

return period
```

Since we have to sort the entries this takes  $O(n \log n)$  time. If they're already sorted, then this just takes  $O(n)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #173

## Problem

This problem was asked by Stripe.

Write a function to flatten a nested dictionary. Namespace the keys with a period.

For example, given the following dictionary:

```
{  
    "key": 3,  
    "foo": {  
        "a": 5,  
        "bar": {  
            "baz": 8  
        }  
    }  
}
```

it should become:

```
{  
    "key": 3,  
    "foo.a": 5,  
    "foo.bar.baz": 8  
}
```

You can assume keys do not contain dots in them, i.e. no clobbering will occur.

## Solution

One solution is to iterate over every key-value in the dict, check whether it's a dictionary, and if it, recursively flatten it and add its values prefixed with its current key.

```
def flatten(d, separator='.'):
```

```
new_dict = {}

for key, val in d.items():
    if isinstance(val, dict):
        flattened_subdict = flatten(val)
        for flatkey, flatval in flattened_subdict.items():
            new_dict[key + separator + flatkey] = flatval
    else:
        new_dict[key] = val

return new_dict
```

This takes O(n) time since we have to look at every key and value in the original dict.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #176

## Problem

This problem was asked by Bloomberg.

Determine whether there exists a one-to-one character mapping from one string  $s_1$  to another  $s_2$ .

For example, given  $s_1 = \text{abc}$  and  $s_2 = \text{bcd}$ , return true since we can map a to b, b to c, and c to d.

Given  $s_1 = \text{foo}$  and  $s_2 = \text{bar}$ , return false since the o cannot map to two characters.

## Solution

We can solve this question by creating a mapping and try to fill it out as we zip along both strings. Let's call the characters at each index  $i$   $\text{char}_1$  and  $\text{char}_2$  for  $s_1$  and  $s_2$  respectively. Then we have to deal with the following cases:

- If the lengths of the strings are different, then return false -- a mapping can't exist.
- If  $\text{char}_1$  doesn't exist in the mapping, then create it and set its value to  $\text{char}_2$ .
- If  $\text{char}_1$  exists in the mapping and if its value is  $\text{char}_2$  then continue.
- If  $\text{char}_1$  exists in the mapping but its value is not  $\text{char}_2$  then we have a conflict, so we can't create a one-to-one mapping, so return false.

```
def mapping_exists(s1, s2):  
    if len(s1) != len(s2):  
        return False  
  
    mapping = {}  
    for char1, char2 in zip(s1, s2):  
        if char1 not in mapping:  
            mapping[char1] = char2  
        else:  
            if mapping[char1] != char2:  
                return False  
    return True
```

```
elif mapping[char1] != char2:  
    return False  
  
return True
```

This takes O(n) time and space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #177

## Problem

This problem was asked by Airbnb.

Given a linked list and a positive integer  $k$ , rotate the list to the right by  $k$  places.

For example, given the linked list  $7 \rightarrow 7 \rightarrow 3 \rightarrow 5$  and  $k = 2$ , it should become  $3 \rightarrow 5 \rightarrow 7 \rightarrow 7$ .

Given the linked list  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$  and  $k = 3$ , it should become  $3 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 2$ .

## Solution

Let's look at the structure of the solution:

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ ,  $k = 2$

becomes

$d \rightarrow e \rightarrow a \rightarrow b \rightarrow c$

To generalize this, we take the  $k$  last elements (in order) and move them to the front. We fix the pointers up so that the last element points to the old head, and the  $k$ th element's next points to null.

We can accomplish this by using fast and slow pointers.

The basic idea is this. First, advance the fast pointer  $k$  steps ahead. Then move the fast and slow pointers together until the fast one hits the end first.

However, to handle the case where  $k$  is larger than the length of the linked list itself, we first get the length of the linked list first  $n$ , and check  $k \% n$  first.

```
def rotate(head, k):
    fast, slow = head, head
```

```
for _ in range(k):
    fast = fast.next

while fast.next is not None:
    slow = slow.next
    fast = fast.next

new_head = slow.next
fast.next = head
slow.next = None

return new_head
```

This takes  $O(n)$  time and  $O(1)$  space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #180

## Problem

This problem was asked by Google.

Given a stack of N elements, interleave the first half of the stack with the second half reversed using only one other queue. This should be done in-place.

Recall that you can only push or pop from a stack, and enqueue or dequeue from a queue.

For example, if the stack is [1, 2, 3, 4, 5], it should become [1, 5, 2, 4, 3]. If the stack is [1, 2, 3, 4], it should become [1, 4, 2, 3].

Hint: Try working backwards from the end state.

## Solution

It's a bit hard to see how we could transform our stack directly to the desired state. So let's consider going backwards from the desired state.

- Given [1, 2, 3, 4, 5] we want [1, 5, 2, 4, 3].
- We can see this is just a pairing of a queue with (5, 4) and a stack of [3, 2, 1] where we first pop off stack and then get from the queue.
- At this point, we can get to the above from a queue of (3, 2, 1, 5, 4)
- Which is just a rotation of (5, 4, 3, 2, 1)

Now let's go forward with these insights.

1. Put all elements into the queue and get (5, 4, 3, 2, 1)
2. Rotate  $\text{len(stack)} / 2$  elements by taking them off the queue (5, 4) and putting them back to get (3, 2, 1, 5, 4)

3. Put  $\text{ceil}(\text{len}(\text{stack}) / 2)$  elements into the stack. The queue is now (5, 4) and stack is [3, 2, 1]
4. Pair them up  $\text{len}(\text{stack}) / 2$  times. If stack is still not empty, pop one more time
5. Move all elements from the queue to the stack

```
from Queue import Queue
import math

def interleave(stack):
    size = len(stack)
    queue = Queue()
    # Step 1.
    while stack:
        queue.put(stack.pop())
    # Step 2.
    for _ in range(size / 2):
        queue.put(queue.get())
    # Step 3.
    for _ in range(int(math.ceil(size / 2.0))):
        stack.append(queue.get())

    # Step 4.
    for _ in range(size / 2):
        queue.put(stack.pop())
        queue.put(queue.get())
    if stack:
        queue.put(stack.pop())
    # Step 5.
    while not queue.empty():
        stack.append(queue.get())
return stack
```

Since each step is at most  $O(N)$ , this runs in  $O(N)$  time, and since we use an extra queue, it takes up  $O(N)$  space.



# Daily Coding Problem #181

## Problem

This problem was asked by Google.

Given a string, split it into as few strings as possible such that each string is a palindrome.

For example, given the input string racecarannakayak, return ["racecar", "anna", "kayak"].

Given the input string abc, return ["a", "b", "c"].

## Solution

The naive, brute force way to solve this would be to recursively look at each possible split of s and get their palindrome splits. We'd keep track of the minimum cut at each split. The base case is when the string itself is a palindrome, leading to a min cut of 1.

```
def is_palindrome(s):
    return s == s[::-1]

def split_into_palindromes(s):
    if not s:
        return []
    if is_palindrome(s):
        return [s]

    min_cut = None
    for i in range(1, len(s)):
        curr_cut = split_into_palindromes(s[:i]) + split_into_palindromes(s[i:])
        if min_cut is None or len(curr_cut) < len(min_cut):
            min_cut = curr_cut
    return min_cut
```

However, this takes  $O(2^n)$  time, since at each step we're recursively calling ourselves twice, which is far too slow. Let's try to speed it up.

Since every function call relies on getting substrings that are palindromes of the original string s, we can build up and use a cache to keep track of all possible subpalindromes so that lookups are faster. So, we'll build a n by n matrix A from the bottom up such that every value  $A[i][j]$  is true if  $s[i:j]$  is a palindrome and false if it isn't.

We can then use dynamic programming and keep an array P of length n with the following invariant:

- $P[i]$  contains the smallest number of palindromes made from the string  $s[:i]$ .

To accomplish this, we'll define the following recurrence:

- $P[i] = \min_{j < i} P[j] + [s[j:i]]$  if  $s[j:i]$  is a palindrome

```
def is_palindrome(s):
    return s == s[::-1]

def split_into_palindromes(s):
    A = [[None for _ in range(len(s))] for _ in range(len(s))]

    # Set all substrings of length 1 to be true
    for i in range(len(s)):
        A[i][i] = True

    # Try all substrings of length 2
    for i in range(len(s) - 1):
        A[i][i + 1] = s[i] == s[i + 1]

    i, k = 0, 3
    while k <= len(s):
        while i < (len(s) - k + 1):
            j = i + k - 1
            A[i][j] = A[i + 1][j - 1] and s[i] == s[j]
            i += 1
        k += 1
        i = 0

    P = [None for _ in range(len(s) + 1)]
    P[0] = []
    for i in range(len(P)):
        for j in range(i):
            matrix_i = i - 1

            if A[j][matrix_i]:
                if P[i] is None or (len(P[j]) + 1 < len(P[i])):
                    P[i] = P[j] + [s[j:i]]

    return P[-1]
```

This takes  $O(n^3)$  time, since we have to build up the matrix (which is  $O(n^2)$  time), and also create a new list with size up to  $O(n)$  for each inner loop of the cache. This also takes  $O(n^2)$  space.



# Daily Coding Problem #182

## Problem

This problem was asked by Facebook.

A graph is minimally-connected if it is connected and there is no edge that can be removed while still leaving the graph connected. For example, any binary tree is minimally-connected.

Given an undirected graph, check if the graph is minimally-connected. You can choose to represent the graph as either an adjacency matrix or adjacency list.

## Solution

Suppose we have a graph that is not minimally-connected. That means that there exists an edge  $(u, v)$  in the graph that, if we remove it, would still have a path from  $u$  to  $v$ . This means that there must be a cycle in the graph along  $u$  and  $v$ . Conversely, if there is no cycle, then the graph is minimally-connected. We can run DFS on the graph to detect a cycle, and return false if there is one.

However, an easier way to look at this is to simply notice that each vertex must have exactly one edge coming from it, which means that there must be  $n - 1$  edges in the graph. So we can also simply count up the number of edges in the graph and check if it's equal to  $n - 1$ .

```
def minimally_connected(graph):
    n = len(graph)
    num_edges = 0
    for v, edges in graph.items():
        num_edges += len(edges)
    num_edges //= 2
    return n - 1 == num_edges
```

This takes  $O(V + E)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #184

## Problem

This problem was asked by Amazon.

Given n numbers, find the greatest common denominator between them.

For example, given the numbers [42, 56, 14], return 14.

## Solution

Because the greatest common divisor is associative, gcd of multiple numbers: say, a, b, c is equivalent to  $\text{gcd}(\text{gcd}(a, b), c)$ . Intuitively, this is because if d divides  $\text{gcd}(a, b)$  and c, it must divide a and b as well by the definition of the greatest common divisor.

Thus the greatest common divisor of multiple numbers a, b, ..., z can be obtained by iteratively computing the gcd of a and b, and gcd of the result of that with c, and so on.

```
def gcd(nums):
    n = nums[0]
    for num in nums[1:]:
        n = _gcd(n, num)
    return n
```

How to implement `_gcd()` though? A naive implementation might try every integer from 1 to  $\min(a, b)$  and see if it divides the larger:

```
def _gcd_naive(a, b):
    smaller, larger = min(a, b), max(a, b)
    for d in range(smaller, 0, -1):
        if larger % d == 0:
            return d
```

However, a much more efficient method is the Euclidean algorithm to find the the greatest

common divisor, which follows the recursive formula:

```
gcd(a, 0) = a  
gcd(a, b) = gcd(b, a % b)
```

```
def _gcd(a, b):  
    if b == 0:  
        return a  
    return _gcd(b, a & b)
```

A more memory-efficient method that works bottom up:

```
def _gcd(a, b):  
    while b:  
        a, b = b, a % b  
    return a
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #189

## Problem

This problem was asked by Google.

Given an array of elements, return the length of the longest subarray where all its elements are distinct.

For example, given the array [5, 1, 3, 5, 2, 3, 4, 1], return 5 as the longest subarray of distinct elements is [5, 2, 3, 4, 1].

## Solution

The brute force solution here would be to test every possible subarray for distinctness, and keep track of the longest:

```
def is_distinct(arr):
    d = {}
    for e in arr:
        if e in d:
            return False
        d[e] = True

    return True

def distinct_subarray(arr):
    max_distinct_subarray = []
    for i in range(len(arr)):
        for j in range(i + 1, len(arr) + 1):
            subarray = arr[i:j]
            if is_distinct(subarray) and len(subarray) > len(max_distinct_subarray):
                max_distinct_subarray = subarray

    return len(max_distinct_subarray)
```

This takes  $O(n^3)$  time and  $O(n)$  space, since we need to get  $O(n^2)$  subarrays, and then iterate over each subarray which can be up to  $O(n)$  in length.

We can make this faster by keeping track of the indices of the last occurring elements as well as the running longest distinct subarray. Thus, when we look at the element at the next index, there are two cases for the longest distinct subarray ending at that index:

- If the element doesn't exist in the dictionary, then the new longest distinct subarray is the same as the previous one with the current element appended
- If it does exist in the dictionary, then the longest distinct subarray starts from  $d[i] + 1$  to the current index.

```
def distinct_subarray(arr):  
    d = {} # most recent occurrences of each element  
  
    result = 0  
    longest_distinct_subarray_start_index = 0  
    for i, e in enumerate(arr):  
        if e in d:  
            # If d[e] appears in the middle of the current longest distinct subarray  
            if d[e] >= longest_distinct_subarray_start_index:  
                result = max(result, i - longest_distinct_subarray_start_index)  
                longest_distinct_subarray_start_index = d[e] + 1  
            d[e] = i  
  
    return max(result, len(arr) - longest_distinct_subarray_start_index)
```

This runs in  $O(n)$  time and  $O(1)$  space.

# Daily Coding Problem #190

## Problem

This problem was asked by Facebook.

Given a circular array, compute its maximum subarray sum in  $O(n)$  time. A subarray can be empty, and in this case the sum is 0.

For example, given [8, -1, 3, 4], return 15 as we choose the numbers 3, 4, and 8 where the 8 is obtained from wrapping around.

Given [-4, 5, 1, 0], return 6 as we choose the numbers 5 and 1.

## Solution

This question is very similar to Daily Coding Problem #49, which asked to find the maximum subarray sum of an array, although in this case it's possible that the subarray can wrap around.

So we can split this problem into two parts. One part is the same as before: find the maximum subarray sum that doesn't wrap around. We also can find the maximum subarray sum that *does* wrap around, and take the maximum of the two.

To find the maximum subarray sum that wraps around, we can find the minimum subarray sum that doesn't wrap around by similar logic as #49, and subtract the total sum of the array by it.

```
def maximum_circular_subarray(arr):
    max_subarray_sum_wraparound = sum(arr) - min_subarray_sum(arr)
    return max(max_subarray_sum(arr), max_subarray_sum_wraparound)
```

```
def max_subarray_sum(arr):
    max_ending_here = max_so_far = 0
    for x in arr:
        max_ending_here = max(x, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

```
def min_subarray_sum(arr):
    min_ending_here = min_so_far = 0
    for x in arr:
        min_ending_here = min(x, min_ending_here + x)
        min_so_far = min(min_so_far, min_ending_here)
    return min_so_far
```

This takes O(n) time and O(1) space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #191

## Problem

This problem was asked by Stripe.

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Intervals can "touch", such as [0, 1] and [1, 2], but they won't be considered overlapping.

For example, given the intervals (7, 9), (2, 4), (5, 8), return 1 as the last interval can be removed and the first two won't overlap.

The intervals are not necessarily sorted in any order.

## Solution

We can do this problem greedily. If we sort the intervals by the end time, and continuously choose the first interval that ends first, we'll minimize the number of overlapping intervals with that one. We can compute overlapping intervals quickly by just keeping track of the end of the last interval we haven't overlapped with:

```
from math import inf

def non_overlapping_intervals(intervals):
    current_end = -inf
    overlapping = 0

    for start, end in sorted(intervals, key=lambda i: i[1]):
        if start >= current_end:
            current_end = end
        else:
            overlapping += 1
```

```
return overlapping
```

This takes  $O(n \log n)$  time, since we have to sort the intervals. It also takes  $O(1)$  time.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #192

## Problem

This problem was asked by Google.

You are given an array of nonnegative integers. Let's say you start at the beginning of the array and are trying to advance to the end. You can advance at most, the number of steps that you're currently on. Determine whether you can get to the end of the array.

For example, given the array [1, 3, 1, 2, 0, 1], we can go from indices 0 -> 1 -> 3 -> 5, so return `true`.

Given the array [1, 2, 1, 0, 0], we can't reach the end, so return `false`.

## Solution

It's tempting to use a greedy strategy and to try to immediately take the largest step we see. However, fails since it might get stuck into a local optimum. Consider [2, 2, 0, 0]: it's better to not take the first 2 and instead do 0 -> 1 -> 3.

Instead, the basic idea is this: we keep track of the absolute possible furthest step we can reach, and not only compute the furthest step we can reach from the current step but all steps in between as well. This works because each step is stateless and gets "reset" whenever you land on a new index. We also can't look past the furthest step, so break if we're past it.

```
def can_reach_end(arr):
    furthest_so_far = 0
    for i in range(len(arr)):
        if i > furthest_so_far:
            break
        furthest_so_far = max(furthest_so_far, i + arr[i])
    return furthest_so_far >= len(arr) - 1
```

This takes O(n) time and constant space.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #196

## Problem

This problem was asked by Apple.

Given the root of a binary tree, find the most frequent subtree sum. The subtree sum of a node is the sum of all values under a node, including the node itself.

For example, given the following tree:

```
5
/ \
2 -5
```

Return 2 as it occurs twice: once as the left leaf, and once as the sum of 2 + 5 - 5.

## Solution

A straightforward way to solve this problem would be to keep a dictionary or counter, and then traverse the tree while recursively computing each subtree sum. We'll do this by computing the leaves first, and then use the solution to that to calculate the subtree sums of its ancestors. At each step we'll increment the counter for that specific sum.

Once we've finished, we can look at the counter and find the key with the highest occurring value -- that's our most frequent subtree sum.

```
from collections import Counter

class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def frequent_subtree_sum(root):
    if root is None:
        return None

    c = Counter()

    def get_subtree_sum(node):
        if node is None:
            return 0
        else:
            return node.val + get_subtree_sum(node.left) + get_subtree_sum(node.right)

    for node in [root]:
        c[get_subtree_sum(node)] += 1

    return c.most_common(1)[0][0]
```

```
s = node.val + get_subtree_sum(node.left) + get_subtree_sum(node.right)
c[s] += 1
return s

get_subtree_sum(root)
return c.most_common(1)[0]
```

This takes  $O(n)$  time and space, since we have to traverse the entire tree once and store each subtree sums' counts.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #197

## Problem

This problem was asked by Amazon.

Given an array and a number  $k$  that's smaller than the length of the array, rotate the array to the right  $k$  elements in-place.

## Solution

One way to perform a circular shift in Python is to slice off the end of the array and attach it to the front:

```
def rotate(array, k):
    n = len(array) - k
    return array[n:] + array[:n]
```

However, this uses  $O(n)$  extra space, as we are creating a new array.

To solve this in place, note that we can think about the array above as consisting of two parts:

- the part before the slice, which we will call A, and
- the part after, which we will call B

We are looking for a way to turn AB into BA, but without actually slicing. Is this possible?

It turns out that it is! We can first reverse the elements in A to create A', and reverse the elements in B to create B'. Finally, we can reverse the entire array, which will give us the desired result in  $O(n)$  time and no extra space.

To give a concrete example, let's say we want to rotate [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] by 3, which will have the same effect as slicing [7, 8, 9] and placing it in front. We first rotate the

individual parts, creating:

```
[6, 5, 4, 3, 2, 1, 0, 9, 8, 7]
```

Finally, we reverse the entire array, to create [7, 8, 9, 0, 1, 2, 3, 4, 5, 6].

Here it is in code:

```
def reverse(array, left, right):
    while left < right:
        array[left], array[right] = array[right], array[left]
        left += 1
        right -= 1

def rotate(array, k):
    n = len(array)
    reverse(array, 0, n - k - 1)
    reverse(array, n - k, n - 1)
    reverse(array, 0, n - 1)
    return array
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #198

## Problem

This problem was asked by Google.

Given a set of distinct positive integers, find the largest subset such that every pair of elements in the subset  $(i, j)$  satisfies either  $i \% j = 0$  or  $j \% i = 0$ .

For example, given the set [3, 5, 10, 20, 21], you should return [5, 10, 20]. Given [1, 3, 6, 24], return [1, 3, 6, 24].

## Solution

The brute force solution would generate all subsets of numbers and, for each one, compare all pairs of numbers to check divisibility.

Since there are  $2^N$  subsets of any set, and looking at all pairs of each subset is  $O(N^2)$ , this would take  $O(2^N * N^2)$ . We must find a better solution.

Note that, for any number  $a$  and  $b$ , if  $a | b$ , then every element that divides  $a$  will also divide  $b$ . So if we have a sorted list, knowing how many divisors each element has before  $k$  will also tell us how many divisors the  $k$ th element has- just one more than that of its greatest divisor. Therefore, we can use dynamic programming to find the largest subset that includes a given number by looking at the sizes of previously computed subsets.

To make this more concrete, suppose we are using the list [5, 10].

Now we look at the second element. Since  $5 | 10$ , and 5 had one divisor,  $\text{num\_divisors}[1] = \text{num\_divisors}[0] + 1 = 2$ .

Finally, for each element in the solution subset, we store the index where we can find the next highest element in the subset. In other words, if  $a < b < c$ , then  $\text{prev\_divisor\_index}[c]$  would be the index of  $b$ , and  $\text{prev\_divisor\_index}[b]$  would be the index of  $a$ .

Let's see how this looks in code:

```
def largest_divisible_subset(nums):
```

```

if not nums:
    return []

nums.sort()

# Keep track of the number of divisors of each element, and where to find
# its last divisor.
num_divisors = [1 for _ in range(len(nums))]
prev_divisor_index = [-1 for _ in range(len(nums))]

# Also track the index of the last element in the best subset solution so far.
max_index = 0

# For each element, check if a previous element divides it. If so, and if adding
# the element will result in a larger subset, update its number of divisors
# and where to find its last divisor.
for i in range(len(nums)):
    for j in range(i):
        if (nums[i] % nums[j] == 0) and (num_divisors[i] < num_divisors[j] + 1):
            num_divisors[i] = num_divisors[j] + 1
            prev_divisor_index[i] = j

    if num_divisors[max_index] < num_divisors[i]:
        max_index = i

# Finally, go back through the chain of divisors and get all the subset elements.
result = []
i = max_index
while i >= 0:

    result.append(nums[i])
    i = prev_divisor_index[i]

return result

```

Since we are looping through the list twice and storing lists of size N, this has time complexity O( $N^2$ ) and space complexity O(N).



# Daily Coding Problem #199

## Problem

This problem was asked by Facebook.

Given a string of parentheses, find the balanced string that can be produced from it using the minimum number of insertions and deletions. If there are multiple solutions, return any of them.

For example, given "()", you could return "(())". Given "))()(", you could return "())()".

## Solution

To solve this problem, first note that there is no significant difference between using insertions or deletions. To see this, think about what we would need to do to balance an unmatched "(" in a string. Whether we decide to delete it, or to append a ")" immediately after, it would still take one operation. So for simplicity, we will choose to use only insertions.

Once this is clear, we can see that a greedy solution is indeed possible, similar to what we would do to determine if the string were balanced. Namely, we can keep a counter, and increment it for open parentheses and decrement it for close parentheses. If the counter is in danger of going negative (indicating an unbalanced string), we must insert an open parenthesis. Otherwise, no change is needed to the original string.

Finally, we may have unmatched open parentheses when we finish traversing the string. To deal with these, we can append a number of close parentheses corresponding to the counter value.

Since we traverse the string once and build up a new string of length at most  $2*N$ , this algorithm has  $O(N)$  time and space complexity.

The code for this problem would look like this:

```
def get_closest_string(s):
    closest_string = []
    open_parens = 0
```

```
for char in s:
    if char == "(":
        open_parens += 1
        closest_string.append(char)

    else:
        if not open_parens:
            closest_string += "("
        else:
            open_parens -= 1
            closest_string.append(char)

while open_parens:
    open_parens -= 1
    closest_string.append(")")

return ''.join(closest_string)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #200

## Problem

This problem was asked by Microsoft.

Let  $X$  be a set of  $n$  intervals on the real line. We say that a set of points  $P$  "stabs"  $X$  if every interval in  $X$  contains at least one point in  $P$ . Compute the smallest set of points that stabs  $X$ .

For example, given the intervals  $[(1, 4), (4, 5), (7, 9), (9, 12)]$ , you should return  $[4, 9]$ .

## Solution

One naive solution would be to evaluate every possible subset of start points and endpoints. In other words, we would first generate all subsets of points from the interval boundaries. Then,

beginning with the smaller subsets and proceeding up to the larger ones, check each one to see if it intersects every interval at some point. Once a satisfactory subset is found, we return it.

Here is how this might look:

```
from itertools import combinations

def get_points(intervals):
    points = set([element for pair in intervals for element in pair])

    subsets = []
    for size in range(1, len(points)):
        subsets.extend(list(combinations(points, size)))

    for start, end in intervals:
        for subset in subsets:
            # Is there any point that is between the start and end of all intervals?
            if all(any(start <= point <= end for point in subset)
                  for start, end in intervals):
```

```
    return subset
```

However, this is horribly inefficient. We have  $O(2^N)$  subsets, and for each one we are matching its points against every interval start point and endpoint, so this will be  $O(2^N * N^2)$ .

To gain some insight, let's take a look at the following intervals:  $[(1, 4), (4, 4), (3, 9)]$ . It is clear that if we choose 4, we can satisfy every interval. So we might guess that choosing the earliest endpoint is an optimal strategy. Can we prove this?

In fact, yes! First, we know that we must choose a point less than or equal to the earliest endpoint. Otherwise, the interval that contains that endpoint will never get stabbed. On the other hand, we know that that endpoint is at least as good as any lesser point in that interval, because it intersects every interval that the lesser point does, and potentially others. For instance, in the example above, if we chose 3 instead of 4, we would not have been able to catch  $(4, 4)$ .

This shows that an optimal greedy strategy exists. First, we sort the intervals by ascending endpoint. Then, as we traverse the list, whenever we reach an interval that is not stabbed by any points in our solution, we take its endpoint and add it to our solution. Since the intervals are sorted, we need only consider the most recently added endpoint to determine if there is an intersection.

```
def get_points(intervals):
    intervals.sort(key=lambda x: (x[1], x[0]))

    points = []
    latest_endpoint = None

    for start, end in intervals:
        if start <= latest_endpoint:
            continue
        else:
            points.append(end)
            latest_endpoint = end

    return points
```

Sorting the intervals is  $O(n * \log n)$ , and we only pass through the list once, so the time complexity is  $O(n * \log n)$ .

Press

# Daily Coding Problem #201

## Problem

This problem was asked by Google.

You are given an array of arrays of integers, where each array corresponds to a row in a triangle of numbers. For example, `[[1], [2, 3], [1, 5, 1]]` represents the triangle:

```
1  
2 3  
1 5 1
```

We define a path in the triangle to start at the top and go down one row at a time to an adjacent value, eventually ending with an entry on the bottom row. For example,  $1 \rightarrow 3 \rightarrow 5$ . The weight of the path is the sum of the entries.

Write a program that returns the weight of the maximum weight path.

## Solution

A brute force solution would be to generate all possible paths and take the maximum of their sums. Since there are  $2^N$  ways of getting from the top to the bottom of our triangle, this would be prohibitively expensive.

Let's look at the example above. If we start at the top, we must either go down to 2 or 3. Our maximum weight, then, is either  $1 + (\text{the maximum weight of a path starting with } 2)$  or  $1 + (\text{the maximum weight of a path starting with } 3)$ . So to find a solution we can figure out which of the left and right subpaths has a greater maximum weight, and add 1 to that value. This points the way to a recursive solution:

- If we are at the bottom level of our triangle, the maximum weight of a path is simply the value at a given index.

- Otherwise, it will be the value at that index, plus the larger of the maximum weights of its left and right subpaths.

```
def max_path_weight(arrays, level=0, index=0):
    if level == len(arrays) - 1:
        return arrays[level][index]
    else:
        return arrays[level][index] + max(
            max_path_weight(arrays, level + 1, index), max_path_weight(arrays, level + 1, index + 1)
        )
```

Although this is neater than a brute force solution, it isn't more efficient, because we are still traversing every path. Instead, we can use dynamic programming to speed things up by storing precomputed values for subpaths.

We already know that the maximum weight for any index on the bottom row is the value at that index. So starting with the second-to-last row, we move up one row at a time, incrementing the value at each of its indices with the the maximum weight of its left or right subpath. Let's see this in action:

```
1
2 3
1 5 1
```

We start with the second row, looking at  $2 + \max(1, 5) = 5$ , so we increment 2 by 5 to get 7. Similarly when we look at 3,  $\max(5, 1) = 5$ , so we increment 3 by 5 to get 8. This results in the following triangle.

```
1
7 8
1 5 1
```

Finally, we replace the 1 in the first row with  $1 + \max(7, 8)$ , giving us 9, our solution. Since we examine each element once, this algorithm is  $O(N)$ , where  $N$  is the total number of elements.

```
def max_path_weight(arrays):
    for level in range(len(arrays) - 2, -1, -1):
        for index in range(level + 1):
            arrays[level][index] += max(arrays[level + 1][index], arrays[level + 1][index + 1])

    return arrays[0][0]
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #202

## Problem

This problem was asked by Palantir.

Write a program that checks whether an integer is a palindrome. For example, 121 is a palindrome, as well as 888. 678 is not a palindrome. Do not convert the integer into a string.

## Solution

Without the restriction on using strings, this problem would be trivial:

```
def check_palindrome(num):
    return str(num) == ''.join(reversed(str(num)))
```

Taking an idea from the above solution, we can instead try to *mathematically* create the reversed integer, and check to see that the original and reversed numbers are the same. How might we do this?

Recall that because we use the decimal system, any n-digit integer  $x_1x_2\dots x_n$  can be represented as  $x_1 * 10^{n-1} + x_2 * 10^{n-2} + \dots + x_n * 10^0$ .

As a result, we can get the digits in reverse order by repeatedly taking the number mod 10 and dividing by 10. For example, if we start with the number 678, we would do the following:

- Take  $678 \% 10$  as the first digit, then divide by 10
- Take  $67 \% 10$  as the second digit, then divide by 10
- Take  $6 \% 10$  as the third digit

Then, if we want to build the reversed number, we can repeatedly add each digit to 10 times our current sum. So to obtain 876, we would do the following:

- Begin with 8
- Multiply  $8 * 10$ , then add our new number 7

- Multiply  $87 * 10$ , then add our new number 6

We can combine these steps as follows:

```
def check_palindrome(num):
    tmp = num
    reversed_num = 0

    while tmp != 0:
        reversed_num = (reversed_num * 10) + (tmp % 10)
        tmp /= 10

    return num == reversed_num
```

This algorithm runs in  $O(N)$  time, where  $N$  is the number of digits in our input, and uses some extra space to store the reversed number. What if we weren't allowed that extra space?

One way to do this is as follows: we begin with the first and last digit of our number. If the digits match, we create a new number by removing those digits. We continue in this way, comparing first and last digits and stripping them from our number, until either there is a mismatch or there are no digits left. If we end up without any digits left, we'll know that we have a palindrome.

The tricky part about this is figuring out how to get and remove the first and last digits mathematically. Let  $n$  be the number of digits in our integer. Then we can perform these operations as follows:

- Get the first digit:  $\text{int} // 10^n$
- Get the last digit:  $n \% 10$
- Remove the first digit:  $\text{int} \% 10^n$
- Remove the last digit:  $n // 10$

Putting this all together, we have the following algorithm:

```
def check_palindrome(num):
    # Find the divisor needed to obtain the first digit.
    divisor = 1

    while (num / divisor >= 10):
        divisor *= 10

    while (num != 0):
        first = num / divisor
        last = num % 10

        if (first != last):
            return False

        num /= divisor

    return True
```

```
num = (num % divisor) / 10

# The number of digits has been reduced by 2, so our divisor is reduced by 10 ** 2.
divisor = divisor / 100

return True
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #203

## Problem

This problem was asked by Uber.

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. Find the minimum element in  $O(\log N)$  time. You may assume the array does not contain duplicates.

For example, given [5, 7, 10, 3, 4], return 3.

## Solution

The  $O(\log N)$  in the instructions may tip you off to the fact that we can use binary search to solve this problem.

Let's look at what happens if we split our list at some point. If the value at our split point is less than the last element in the list, we know the right half is sorted, so the minimum element must lie in the left half (including the midpoint). Otherwise, it must lie in the right half.

We can apply this routine repeatedly to cut in half the area where the minimum element must be. Eventually we will reach a point where the lowest and highest indices we are looking at will be the same, and the element at this index will be our solution.

```
def helper(arr, low, high):
    if high == low:
        return arr[low]

    mid = (high + low) / 2
    if arr[mid] < arr[high]:
        high = mid
    else:
        low = mid + 1
```

```
return helper(arr, low, high)

def find_min_element(arr):
    low, high = 0, len(arr) - 1
    return helper(arr, low, high)
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #231

## Problem

This problem was asked by IBM.

Given a string with repeated characters, rearrange the string so that no two adjacent characters are the same. If this is not possible, return None.

For example, given "aaabbc", you could return "ababac". Given "aaab", return None.

## Solution

Intuitively, the following greedy algorithm may come to mind: keep popping the most frequent character from the string (that is different from the last one) and adding it to a new list, until there are no more letters left. If we have used all the letters, return the joined list; otherwise, return None.

One tricky part is that we must make sure to pop each new letter before adding back the last one, to avoid repeats.

Here is how we could implement this.

```
from collections import defaultdict

def rearrange(string):
    frequencies = defaultdict(int)
    for letter in string:
        frequencies[letter] += 1

    char, count = max(frequencies.items(), key=lambda x: x[1])
    frequencies.pop(char)
    result = [char]

    while frequencies:
```

```

last_char, last_count = char, count

char, count = max(frequencies.items(), key=lambda x: x[1])
frequencies.pop(char)
result.append(char)

if last_count > 1:
    frequencies[last_char] = last_count - 1

if len(result) == len(string):
    return "".join(result)
else:
    return None

```

This will take  $O(N^2)$ , where  $N$  is the number of letters in the string. This is because finding the maximum value in the dictionary is  $O(N)$ , and we must do this every time we locate the next letter to add to our result.

We can improve this by using a better data structure. In particular, a heap will allow us to pop and push the most frequent letter in  $O(\log N)$ , reducing our time complexity to  $O(N * \log N)$ .

Similar to before, our algorithm will be divided into two steps. First we insert the counts of each character into a heap. Then, we pop the most frequent element one at a time and add it to the result. Once this element is popped, we reinsert the previous element, decrementing its count by one.

```

from collections import defaultdict
import heapq

def rearrange(string):
    frequencies = defaultdict(int)
    for letter in string:
        frequencies[letter] += 1

    heap = []
    for char, count in frequencies.items():
        heapq.heappush(heap, (-count, char))

    count, char = heapq.heappop(heap)
    result = [char]

    while heap:
        last = (count + 1, char)
        count, char = heapq.heappop(heap)
        result.append(char)

```

```
if last[0] < 0:  
    heapq.heappush(heap, last)  
  
if len(result) == len(string):  
    return "".join(result)  
else:  
    return None
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #232

## Problem

This problem was asked by Google.

Implement a `PrefixMapSum` class with the following methods:

- `insert(key: str, value: int)`: Set a given key's value in the map. If the key already exists, overwrite the value.
- `sum(prefix: str)`: Return the sum of all values of keys that begin with a given prefix.

For example, you should be able to run the following code:

```
mapsum.insert("columnar", 3)
assert mapsum.sum("col") == 3

mapsum.insert("column", 2)
assert mapsum.sum("col") == 5
```

## Solution

Depending on how efficient we want our `insert` and `sum` operations to be, there can be several solutions.

If we care about making `insert` as fast as possible, we can use a simple dictionary to store the value of each key inserted. As a result insertion will be  $O(1)$ . Then, if we want to find the `sum` for a given key, we would need to add up the values for every word that begins with that prefix. If  $N$  is the number of words inserted so far, and  $k$  is the length of the prefix, this will be  $O(N * k)$ .

This could be implemented as follows:

```

class PrefixMapSum:

    def __init__(self):
        self.map = {}

    def insert(self, key: str, value: int):
        self.map[key] = value

    def sum(self, prefix):
        return sum(value for key, value in self.map.items() if key.startswith(prefix))

```

On the other hand, perhaps we will rarely be inserting new words, and need our `sum` retrieval to be very efficient. In this case, every time we `insert` a new key, we can recompute all its prefixes, so that finding a given sum will be  $O(1)$ . However, insertion will take  $O(k^2)$ , since slicing is  $O(k)$  and we must do this  $k$  times.

```

from collections import defaultdict

class PrefixMapSum:
    def __init__(self):
        self.map = defaultdict(int)
        self.words = set()

    def insert(self, key: str, value: int):
        # If the key already exists, increment prefix totals by the difference of old and new values.
        if key in self.words:
            value -= self.map[key]
        self.words.add(key)

        for i in range(1, len(key) + 1):
            self.map[key[:i]] += value

    def sum(self, prefix):
        return self.map[prefix]

```

A hybrid solution would involve using a trie data structure. When we insert a word into the trie, we associate each letter with a dictionary that stores the letters that come after it in various words, as well as the total at any given time.

For example, suppose that we wanted to perform the following operations:

```

mapsum.insert("bag", 4)
mapsum.insert("bath", 5)

```

For the first `insert` call, since there is nothing already in the trie, we would create the following:

```
{"b": "total": 4,
 {"a": "total": 4,
  {"g": "total": 4}
 }
}
```

When we next insert `bath`, we will step letter by letter through the trie, incrementing the total for the prefixes already in the trie, and adding new totals for prefixes that do not exist. The resulting dictionary would look like this:

```
{"b": "total": 9,
 {"a": "total": 9,
  {"g": "total": 4},
  {"t": "total": 5,
   {"h": "total": 5}
  }
 }
}
```

As a result, `insert` and `sum` will involve stepping through the dictionary a number of times equal to the length of the prefix. Since finding the dictionary values at each level is constant time, this algorithm is  $O(k)$  for both methods.

```
from collections import defaultdict

class Trie:
    def __init__(self):
        self.letters = defaultdict(dict)
        self.total = 0

class PrefixMapSum:
    def __init__(self):
        self.root = Trie()
        self.map = {}

    def insert(self, key: str, value: int):
        # If the key already exists, increment prefix totals by the difference of old and new
        # values.
        value -= self.map.get(key, 0)
        self.map[key] = value

        curr = self.root
        for char in key:
            curr = curr.letters.setdefault(char, Trie())
            curr.total += value
```

```
def sum(self, prefix):
    curr = self.root
    for char in prefix:
        curr = curr.letters[char]
    return curr.total if curr else 0
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #233

## Problem

This problem was asked by Apple.

Implement the function `fib(n)`, which returns the  $n^{\text{th}}$  number in the Fibonacci sequence, using only  $O(1)$  space.

## Solution

Finding the  $n^{\text{th}}$  Fibonacci sequence is a classic problem with a variety of solutions.

Recall that the Fibonacci sequence can be defined such that the first two numbers are 0 and 1, and thereafter each number is the sum of the two that came before it. The first few numbers are 0, 1, 1, 2, 3, 5, ....

Because of the property that each number depends only on the two preceding numbers, it is straightforward to use recursion to find the  $n^{\text{th}}$  Fibonacci number.

```
def fib(n: int):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

As  $n$  gets very large, however, this quickly becomes inefficient. The reason for this is that every call to `fib` generates two additional calls, meaning that we must invoke this function on the order of  $2^n$  times.

We can reduce the number of calls by saving previous results in a cache. If we have already calculated `fib(n)` for some arbitrary  $n$ , we return the value stored in our cache. If not, we make sure to set the value in the cache before returning.

```
cache = {0: 0, 1: 1}
```

```
def fib(n: int):
    if n in cache:
        return cache[n]
    else:
        cache[n] = fib(n - 1) + fib(n - 2)
        return cache[n]
```

Neither of these solutions satisfy the requirement that we only use constant space. Note that each number in the sequence only depends on the two preceding ones. So we can find the  $n^{\text{th}}$  Fibonacci number by continually updating these two numbers, according to the formula  $a$ ,  $b = b$ ,  $a + b$ . For example, 3 and 5 will turn into 5 and 8. Once the updates are done, we simply take the last number.

```
def fib(n: int):
    if n <= 1:
        return n
    else:
        a, b = 0, 1
        for _ in range(n - 1):
            a, b = b, a + b
    return b
```

Finally, it is in fact possible to write an  $O(1)$  time *and* space algorithm, since there is a closed form mathematical solution. While you probably will not be expected to provide this in an interview, it may still be interesting to know.

```
from math import sqrt

PHI = (1 + sqrt(5)) / 2

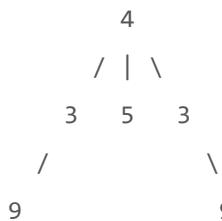
def fib(n: int):
    return int(PHI ** n / sqrt(5) + 0.5)
```

# Daily Coding Problem #237

## Problem

This problem was asked by Amazon.

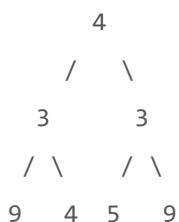
A tree is symmetric if its data and shape remain unchanged when it is reflected about the root node. The following tree is an example:



Given a k-ary tree, determine whether it is symmetric.

## Solution

When solving problems with k-ary trees, it is often helpful to consider the simpler case of a binary tree first. Let's analyze the example below.



Here are the checks we would perform:

- `root == root`
- `root.left == root.right`
- `root.left.left == root.right.right`
- `root.left.right == root.right.left`

For the last comparison, since `4 != 5`, we would return `False`.

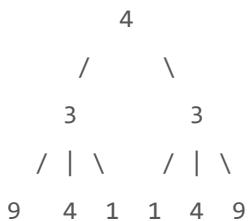
We can turn this into a recursive solution. For each `Node` we traverse, starting with the root, we check that the values of its left and right child nodes are equal, and that the grandchild nodes form mirror images.

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def is_symmetric(left, right):
        if not left and not right:
            return True
        elif not left or not right:
            return False
        return left.val == right.val and \
               is_symmetric(left.left, right.right) and is_symmetric(left.right, right.left)

assert is_symmetric(root, root)
```

For a k-ary tree, we will need to compare up to k children, but a similar principle applies. Suppose the number of children for two nodes we are comparing is k. Then we can loop through the list of children for each node, comparing `left[0]` to `right[k - 1]`, `left[1]` to `right[k - 2]`, and so on.



For the example above, each of the root's child nodes have three children. Comparing them in the way described above, we find that `9 == 9`, `4 == 4`, and `1 == 1`, so this tree is indeed symmetric.

A recursive implementation of the above is as follows:

```

class Node:

    def __init__(self, val, children=[]):
        self.val = val
        self.children = children


def is_symmetric(left, right):
    if left.val != right.val:
        return False

    if not left.children and not right.children:
        return True

    if len(left.children) != len(right.children):
        return False

    k = len(left.children)
    for i in range(k):
        if not is_symmetric(left.children[i], right.children[k - 1 - i]):
            return False

    return True

assert is_symmetric(root, root)

```

The complexity of this algorithm is  $O(N)$  for both binary and  $k$ -ary trees, since in either case we only examine each node once.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #241

## Problem

This problem was asked by Palantir.

In academia, the h-index is a metric used to calculate the impact of a researcher's papers. It is calculated as follows:

A researcher has index  $h$  if at least  $h$  of her  $N$  papers have  $h$  citations each. If there are multiple  $h$  satisfying this formula, the maximum is chosen.

For example, suppose  $N = 5$ , and the respective citations of each paper are [4, 3, 0, 1, 5]. Then the h-index would be 3, since the researcher has 3 papers with at least 3 citations.

Given a list of paper citations of a researcher, calculate their h-index.

## Solution

The simplest way to solve this is to sort the papers in decreasing order by number of citations, and find the last index such that the value at that index is at least as great as the index.

In the example above, for example, [5, 4, 3, 1, 0], we see that `citations[3] >= 3`, whereas `citations[4] < 4`.

```
def h_index(citations):
    n = len(citations)
    citations.sort(reverse=True)

    h = 0
    while h < n and citations[h] >= h + 1:
        h += 1

    return h
```

However, sorting the citations is  $O(N * \log N)$ . A more efficient solution is to bucket sort the citations, putting all citations with count greater than  $N$  in their own bucket. Then, we can descend from  $N$  to  $0$ , accumulating the counts for each bucket. The accumulated total at a given index represents the number of papers with at least that many citations. So once we reach a point where `total >= index`, we have found our answer.

In the example above, counts would be `[1, 1, 0, 1, 1, 1]`, and the total for the third index would be 3.

```
def h_index(citations):
    n = len(citations)
    counts = [0 for _ in range(n + 1)]

    for citation in citations:
        if citation >= n:
            counts[n] += 1
        else:
            counts[citation] += 1

    total = 0
    for i in range(n, -1, -1):
        total += counts[i]
        if total >= i:
            return i
```

The bucket sort and countdown each take  $O(N)$  time, so this algorithm is  $O(N)$ .

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)

# Daily Coding Problem #244

## Problem

This problem was asked by Square.

The Sieve of Eratosthenes is an algorithm used to generate all prime numbers smaller than N. The method is to take increasingly larger prime numbers, and mark their multiples as composite.

For example, to find all primes less than 100, we would first mark [4, 6, 8, ...] (multiples of two), then [6, 9, 12, ...] (multiples of three), and so on. Once we have done this for all primes less than N, the unmarked numbers that remain will be prime.

Implement this algorithm.

Bonus: Create a generator that produces primes indefinitely (that is, without taking N as an input).

## Solution

Despite being very old, the Sieve of Eratosthenes is a fairly efficient method of finding primes. As described above, here is how it could be implemented:

```
def primes(n):
    is_prime = [False] * 2 + [True] * (n - 1)

    for x in range(n):
        if is_prime[x]:
            for i in range(2 * x, n, x):
                is_prime[i] = False

    for i in range(n):
        if is_prime[i]:
            yield i
```

There are a few ways we can improve this. First, note that for any prime number  $p$ , the first useful multiple to check is actually  $p^2$ , not  $2 * p$ ! This is because all numbers  $2 * p, 3 * p, \dots, i * p$  where  $i < p$  will already have been marked when iterating over the multiples of  $2, 3, \dots, i$  respectively.

As a consequence of this we can make another optimization: since we only care about  $p^2$  and above, there is no need for  $x$  to range all the way up to  $N$ : we can stop at the square root of  $N$  instead.

Taken together, these improvements would look like this:

```
def primes(n):
    is_prime = [False] * 2 + [True] * (n - 1)

    for x in range(int(n ** 0.5)):
        if is_prime[x]:
            for i in range(x ** 2, n, x):
                is_prime[i] = False

    for i in range(n):
        if is_prime[i]:
            yield i
```

Finally, to generate primes without limit we need to rethink our data structure, as we can no longer store a boolean list to represent each number. Instead, we must keep track of the lowest unmarked multiple of each prime, so that when evaluating a new number we can check if it is such a multiple, and mark it as composite. This is a good candidate for a heap-based solution.

Here is how we could implement this. We start a counter at 2 and incrementally move up through the integers. The first time we come across a prime number  $p$ , we add it to a min-heap with priority  $p^2$  (using the optimization noted above), and `yield` it. Whenever we come across an integer equal with this priority, we pop the corresponding key and reinsert it with a new priority equal to the next multiple of  $p$ .

For integers between 2 and 10, we would perform the following actions:

```
2: push [4, 2], yield 2
3: push [9, 3], yield 3
4: pop [4, 2], push [6, 2]
5: push [25, 5], yield 5
6: pop [6, 2], push [8, 2]
7: push [49, 7], yield 7
8: pop [8, 2], push [10, 2]
9: pop[9, 3], push [12, 3]
```

An important thing to note is that at any given time the next composite number will be first in the heap, so it suffices to check and update only the highest-priority element.

```
import heapq

def primes():
    composite = []
    i = 2

    while True:
        if composite and i == composite[0][0]:
            while composite[0][0] == i:
                multiple, p = heapq.heappop(composite)
                heapq.heappush(composite, [multiple + p, p])

        else:
            heapq.heappush(composite, [i*i, i])
            yield i

        i += 1
```

The time complexity is the same as above, as we are implementing the same algorithm. However, at the point when our algorithm considers an integer N, we will already have popped all the composite numbers less than N from the heap, leaving only multiples of the prime ones. Since there are approximately  $N / \log N$  primes up to N, the space complexity has been reduced to  $O(N / \log N)$ .

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)



# Daily Coding Problem #246

## Problem

This problem was asked by Dropbox.

Given a list of words, determine whether the words can be chained to form a circle. A word X can be placed in front of another word Y in a circle if the last character of X is same as the first character of Y.

For example, the words ['chair', 'height', 'racket', 'touch', 'tunic'] can form the following circle: chair --> racket --> touch --> height --> tunic --> chair.

## Solution

It will be helpful to examine this problem as a graph. Consider each starting letter as a vertex, and each word as an edge that connects the starting letter and ending letter of that word. We can represent this in an adjacency list like so:

```
def make_graph(words):
    graph = defaultdict(list)

    for word in words:
        graph[word[0]].append(word[-1])

    return graph
```

Finding out whether we can chain these strings together is equivalent to determining whether there is a cycle that goes through all edges, or in other words an [Eulerian cycle](#).

An efficient method of solving this takes advantage of the fact that a graph has a Eulerian cycle if and only if it satisfies the following two properties:

- For each vertex, its in-degree equals its out-degree.
- The graph is strongly connected.

The in-degree of a vertex refers to how many edges are directed into that vertex, while the out-degree refers to how many edges are directed out from that vertex. For our problem, this corresponds to how many words end and start with a given letter, respectively. To check that these values are equal, we can simply iterate over our graph and count them up.

```
def are_degrees_equal(graph):
    in_degree = defaultdict(int)
    out_degree = defaultdict(int)

    for key, values in graph.items():
        for v in values:
            out_degree[key] += 1
            in_degree[v] += 1

    return in_degree == out_degree
```

It remains to determine whether the graph is strongly connected. In other words, we must determine whether every node is reachable from every other node. Instead of trying a depth-first search from every node, there is a straightforward algorithm we can use. Choosing a vertex at random, we first see if we can visit all other vertices from it. Then, we reverse all edges in the graph and see if this is still possible.

```
def find_component(graph, visited, current_word):
    visited.add(current_word)

    for neighbor in graph[current_word]:
        if neighbor not in visited:
            find_component(graph, visited, neighbor)

    return visited

def is_connected(graph):
    start = list(graph)[0]
    component = find_component(graph, set(), start)
```

```
reversed_graph = defaultdict(list)
for key, values in graph.items():
    for v in values:
        reversed_graph[v].append(key)
reversed_component = find_component(graph, set(), start)

return component == reversed_component == graph.keys()
```

Creating the graph and checking the in-degree and out-degree of each vertex are both  $O(N)$ , where  $N$  is the number of words. Determining whether the graph is connected involves two depth-first searches, each with time complexity  $O(V + E)$ . But since the number of edges or vertices can be no greater than the number of strings, the whole algorithm runs in  $O(N)$  time.

The full solution is as follows:

```
from collections import defaultdict

def find_component(graph, visited, current_word):
    visited.add(current_word)

    for neighbor in graph[current_word]:
        if neighbor not in visited:
            find_component(graph, visited, neighbor)

    return visited

def is_connected(graph):
    start = list(graph)[0]
    component = find_component(graph, set(), start)

    reversed_graph = defaultdict(list)
    for key, values in graph.items():
        for v in values:
            reversed_graph[v].append(key)
    reversed_component = find_component(graph, set(), start)

    return component == reversed_component == graph.keys()

def are_degrees_equal(graph):
    in_degree = defaultdict(int)
    out_degree = defaultdict(int)
```

```
for key, values in graph.items():
    for v in values:
        out_degree[key] += 1
        in_degree[v] += 1

return in_degree == out_degree

def make_graph(words):
    graph = defaultdict(list)

    for word in words:
        graph[word[0]].append(word[-1])

    return graph

def can_chain(words):
    graph = make_graph(words)

    degrees_equal = are_degrees_equal(graph)
    connected = is_connected(graph)

    return degrees_equal and connected
```

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)