1. Witness of The Tall People

This problem was recently asked by Google:

There are n people lined up, and each have a height represented as an integer. A murder has happened right in front of them, and only people who are taller than everyone in front of them are able to see what has happened. How many witnesses are there?

Example:

```
Input: [3, 6, 3, 4, 1]
Output: 3
```

Explanation: Only `[6, 4, 1]` were able to see in front of them.

```
 #
 #
 # #
#### 
####
#####
36341                    x (murder scene)
```

Here's your starting point:

```
def witnesses(heights):
  # Fill this in.

print witnesses([3, 6, 3, 4, 1])
# 3
```

2. Create a Simple Calculator

Given a mathematical expression with just single digits, plus signs, negative signs, and brackets, evaluate the expression. Assume the expression is properly formed.

Example:

```
Input: - ( 3 + ( 2 - 1 ) )
Output: -4
```

Here's the function signature:

```
def eval(expression):
  # Fill this in.

print eval('- (3 + ( 2 - 1 ) )')
# -4
```

Given that the expression is properly formed, we can iterate through each and determine the actions to do depending on which character we see. The 2 main things we need to keep track of is:

- The current total
- The operation to be operated for the next integer (or bracket)

So if we see a `+` or `-`, we can safely save that into the `op` variable.
If we see a digit, similarly, we can add or subtract accordingly.
If we see an open bracket, some special work needs to be done. A recursive eval call should be done, and should start processing after where the bracket starts. This can be solved by also passing in the index of where to start processing in the expression.
If we see an end bracket, we can simply return our results. We need to also return which index we have processed up to, so the parent eval function knows where to continue off.

```
def eval_helper(expression, index = 0):
  result = 0
  op = '+'
  while index < len(expression):
              char = expression[index]
              if char.isdigit():
              if op == '+':
              result += int(char)
              else:
              result -= int(char)
              elif char in ('+', '-'):
              op = char
              elif char == '(':
              n, index = eval_helper(expression, index + 1)
              if op == '+':
              result += n
              else:
              result -= n
              elif char == ')':
              return (result, index)
              index += 1
  return (result, index)

def eval(expression):
  return eval_helper(expression)[0]

print eval('- (3 + ( 2 - 1 ) )')
# -4
```

The time complexity of this solution is `O(n)` since each character in the string is iterated through once. The space complexity is actually `O(n)`, as for every recursive call some space is used on the stack, and the brackets can be nested proportionally to the length of the string.

3. Maximum In A Stack

Implement a class for a stack that supports all the regular functions (`push`, `pop`) and an additional function of `max()` which returns the maximum element in the stack (return `None` if the stack is empty). Each method should run in constant time.

```
class MaxStack:
 def __init__(self):
        # Fill this in.

 def push(self, val):
        # Fill this in.

 def pop(self):
        # Fill this in.

 def max(self):
        # Fill this in.

s = MaxStack()
s.push(1)
s.push(2)
s.push(3)
s.push(2)
print s.max()
# 3
s.pop()
s.pop()
print s.max()
# 2
```

4. Add two numbers as a linked list

You are given two linked-lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Example:

```
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807.
```

Here is the function signature as a starting point (in Python):

```python
# Definition for singly-linked list.
class ListNode(object):
  def __init__(self, x):
          self.val = x
          self.next = None

class Solution:
  def addTwoNumbers(self, l1, l2, c = 0):
          # Fill this in.

l1 = ListNode(2)
l1.next = ListNode(4)
l1.next.next = ListNode(3)

l2 = ListNode(5)
l2.next = ListNode(6)
l2.next.next = ListNode(4)

result = Solution().addTwoNumbers(l1, l2)
while result:
  print result.val,
  result = result.next
# 7 0 8
```

Why Python? We recommend using Python as a generalist language for interviewing, as it is well-regarded in the tech industry and used across Google/YouTube, Facebook/Instagram, Netflix, Uber, Dropbox, Pinterest, Spotify, etc., It is easy to learn with readable syntax, and very similar in structure to other popular languages like Java, C/C++, Javascript, PHP, Ruby, etc. Python is generally faster to read/write though, which makes it ideal for interviews. You can, of course, use any language you like!

5. Number of Ways to Climb Stairs

You are given a positive integer N which represents the number of steps in a staircase. You can either climb 1 or 2 steps at a time. Write a function that returns the number of unique ways to climb the stairs.

```
def staircase(n):
  # Fill this in.

print staircase(4)
# 5
print staircase(5)
# 8
```

Can you find a solution in `O(n)` time?

6. Word Search

You are given a 2D array of characters, and a target string. Return whether or not the word target word exists in the matrix. Unlike a standard word search, the word must be either going left-to-right, or top-to-bottom in the matrix.

Example:

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

Given this matrix, and the target word `FOAM`, you should return `true`, as it can be found going up-to-down in the first column.

Here's the function signature:

```
def word_search(matrix, word):
  # Fill this in.

matrix = [
  ['F', 'A', 'C', 'I'],
  ['O', 'B', 'Q', 'P'],
  ['A', 'N', 'O', 'B'],
  ['M', 'A', 'S', 'S']]
print word_search(matrix, 'FOAM')
# True
```