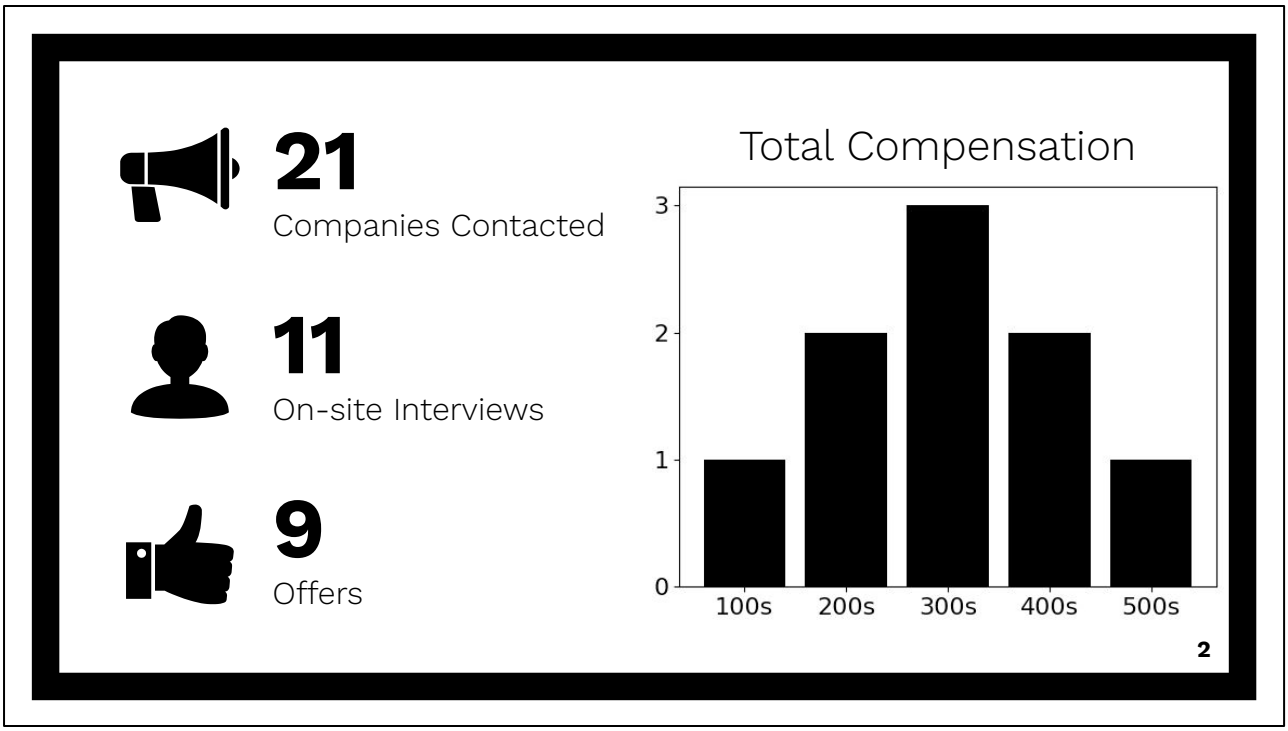
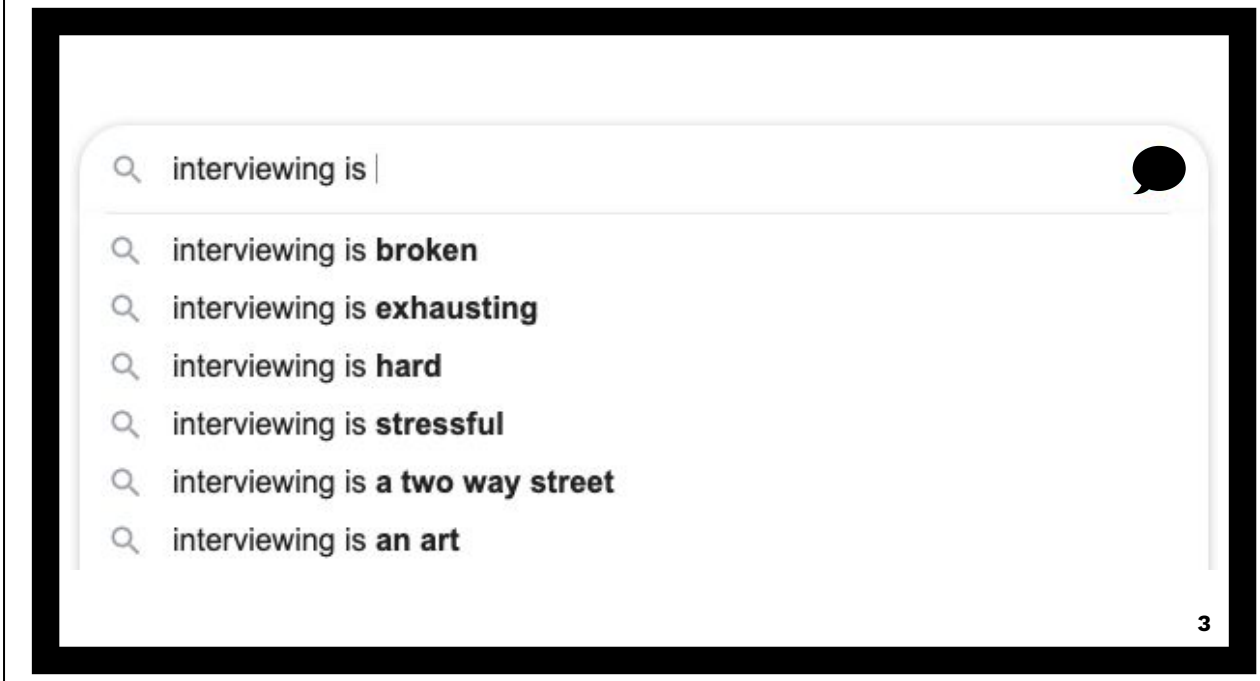
Hi! Smile. My name is Grant Jenks.
Excited to be here.
Let’s get started.



21 companies contacted — 8 ignored my application/referral — 2 rejected me based on resume.
All about referrals!
11 brought me on-site for interviews — 2 rejected me based on interviews.
Facebook, Amazon, Apple, LinkedIn, Google, and startups.
9 made offers.
I did over 145 practice problems in preparation! Wrote over 4,000 lines of code!
Joined LinkedIn July, 2019 as Staff Software Engineer.



Bad attitude: Interviewing is the worst!
Bad attitude: “Don’t hate the player, hate the game.”
Forgive the system for what it is. Set boundaries. Learn to accept or even love it.
The upside is incredible. For example, $10K more per year will accrue millions over your career.

Algorithms

"An algorithm must be seen to be believed."
— Donald Knuth

The algorithmic puzzle, your career, and hundred(s) of thousands of dollars all hanging in the balance.
We're going to look at ten types of algorithms.

itertools.permutations

```
def brute_force_sort(values):
    "Brute-force sort algorithm."
    n = len(values)
    for p in permutations(values, n):
        idx = range(1, n)
        ordered = all(p[i-1] <= p[i] for i in idx)
        if ordered:
            return p
```


Brute-force algorithm.
Worst sort algorithm of all time. Factorial runtime. Hard to do worse, actually.

max() & min()

```
def max_tree_height(root):
    """Choose local maxima at each node for global
    maxima at root."""
    if root is None:
        return 0
    max_left = max_tree_height(node.left)
    max_right = max_tree_height(node.right)
    return max(max_left, max_right) + 1
```

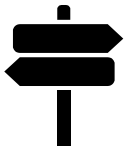

Greedy algorithms using either max() or min().
In order to understand recursion, one must first understand recursion.

from heapq import *

```
def generic_greedy_algorithm():
    pairs = [(weight, option)]
    while pairs:
        _, option = heappop(pairs)
        ...
        for pair in new_options():
            heappush(pair)
```


Greedy algorithm using min-heap.
Maintain min-heap using heapq. One of the more misunderstood modules.
Leverage lexicographic ordering of tuples.
Dijkstra's Shortest Path, Minimum Spanning Tree, Huffman Coding, etc.

list.append & list.pop



```
def find_value_in_tree(root, value):
    stack = [root]
    while stack:
        node = stack.pop()
        if node is None: continue
        if value == node.value: return True
        stack.extend(node.children)
    return False
```


Back-tracking algorithm using stack.
Use explicit stack or implicit stack through recursion.
Example of depth-first search.
Change to breadth-first search by using deque. One-line change!

from bisect import *

```
def grade(score):
    """Grade score given breakpoints and grades.
        >>> scores = [33, 99, 77, 70, 89, 65]
        >>> [grade(score) for score in scores]
        ['F', 'A', 'C', 'C', 'B', 'D']
    """
    breakpoints, grades = [60, 70, 80, 90], 'FDCBA'
    index = bisect_right(breakpoints, score)
    return grades[index]
```


Divide and conquer based on search space.
Use the bisect module for built-in algorithm.
Sometimes it's a data-type like the binary search tree instead.

collections.defaultdict

```
def longest_common_substring_len(seq1, seq2):
    table = defaultdict(lambda: defaultdict(int))
    for i1, e1 in enumerate(seq1):
        for i2, e2 in enumerate(seq2):
            if e1 == e2:
                table[i1][i2] = table[i1-1][i2-1] + 1
            else:
                table[i1][i2] = 0
    return max(max(d.values()) for d in table.values())
```

Dynamic-Programming algorithm.
Memoization bottom-up.
Cache intermediate results using “table” (dictionary of dictionaries).

functools.lru_cache

```
@lru_cache(maxsize=None)
def fibonacci(n):
    """Calculate the nth Fibonacci number.
        >>> fibonacci(100)
        354224848179261915075
    """
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

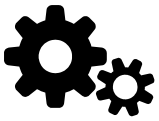
Dynamic-Programming algorithm.
Memoization top-down.
Cache intermediate results using lru_cache.

Recurrence Relations

```
def max_profit_two_transactions(prices):
    buy1, sell1 = buy2, sell2 = float('-inf'), 0
    for price in prices:
        sell2 = max(sell2, buy2 + price)
        buy2  = max(buy2 , sell1 - price)
        sell1 = max(sell1, buy1 + price)
        buy1  = max(buy1 , -price)
    return sell2
```

You're going to get stuck. Story: "Keyboard not found... Press any key to continue."
Recurrence relations occur when you have a dynamic-programming-like solution where the state-space can be shrunk down.
These are tricky devils.
Best you can do on day 1? Buy the stock.
Best on day 2? Sell the stock.
Best on day 3? Buy the stock.
Best on day 4? Sell the stock.

Finite State Machines



```
def valid_parens(characters):
    count = 0
    char_map = {'(': 1, ')': -1}
    for c in characters:
        count += char_map.get(c, 0)
        if count < 0:
            return False
    return count == 0
```

Finite State Machine pattern: very common in parsing.
For each input element, look up transition in map and update state.
Useful tool for complex algorithm problems with multiple scenarios or such.

OOP Design

```
class StatisticsSummary:
    def add(self, value):
        "Add value to the statistics summary."
        ...
    def mean(self, value):
        "Return average of statistics summary."
        ...
    def mode(self, value):
        "Return mode of statistics summary."
```

Actually one of my favorites. All about tradeoffs. Often more practical.
Remember your jargon: is-a and has-a relationships.
Why object-oriented programming: encapsulation, inheritance, and polymorphism.
Unlikely Python-specific. Don't get lost in the object model e.g. no metaclasses.
But remember __init__ and __repr__. Critical for state and debugging.

Coding Interviews

“Talk is cheap. Show me the code.”
— Linus Torvalds

I much prefer a coding interview in front of a computer.
Couple of tips for quick coding with Python.

logging & doctest

```
def solve_linear(a, b):
    """Solve linear function Ax+B=0.
        >>> solve_linear(2, 4)
        -2.0
    """
    logging.debug('Solving %sx+%s', a, b)
    return -b / a
logging.basicConfig(level=logging.DEBUG)
print(doctest.testmod())
```

16

Coding interviews were more common in phone screens. But some companies did it on-site.
Must have modules: logging and doctest.
Using print() feels like buying milk from the store. It's useful. It's wholesome. It's easy.
Using logging feels like buying a house. You don't know what you're getting into. It's not a function. It's a whole framework.
Logging is the Enterprise version of Python's print function.
Doctest is the best BOGO deal in Silicon Valley. Pay for documentation and get testing for free!

%run & %debug

```
In [1]: %run mymath.py
TestResults(failed=0, attempted=1)
In [2]: solve_linear(0, 10)
ZeroDivisionError: division by zero
In [3]: %debug
> /Users/grantjenks/mymath.py(9) solve_linear()
----> 9     return -b / a
ipdb> p a
0
```


Gotta be using IPython. Two killer features: %run and %debug.
%run is so obvious. It types your code in for you. It's just that simple.
You can do something similar in Python with importlib.reload(mymath) but that looks like dark magic.
%debug feels like magic. It goes back in time and starts a debug session at the point of failure.
This is called post-mortem debugging. Not really magic but it's a wonder.
Now is your chance to show the interviewer that you do not fear the debugger.

Hazards

“The enemy is anybody who's going to get you killed, no matter which side he is on.” — Joseph Heller, Catch-22

Resist the temptation to reach for the "big-guns" like numpy for matrices or “deep learning” for a generic problem.
Even though that may be the “right answer”, if you’re not familiar with that, it usually shows.
Programming interviews often require us to make tradeoffs we would not normally make.

Variable Names

```
>>> x = y = z = 0
>>> n = len(data)
>>> for i, e in enumerate(seq):
...     print(f'{i}. {e}')
>>> lo, hi = 0, n-1
>>> for k, v in d.items():
...     print(k, '->', v)
>>> pair = 1989, 'Python'
```


Variable names matter!
x, y, z for variables
n for length or size
i, j, k for indices
e for elements
lo, hi for bounds
k for keys, v for values

int & float

```
>>> 1 // 3
0
>>> -1 // 3
-1
>>> int(float(-1) / 3)
0
>>> round(1.5)
2
>>> round(2.5)
2
```

```
>>> 1 / 3
0.3333333333333333
>>> 0.1 * 5 == 0.5
True
>>> 0.1 * 6 == 0.6
False
>>> 0.1+0.1+0.1 == 0.3
False
>>> decimal.Decimal(0.3)
Decimal('0.29999999...')
```


Integer division in Java does not match floordiv or truediv.

literals

```
>>> list() == []
True
>>> dict() == {}
True
>>> tuple() == ()
True
>>> set([1]) == {1}
True
>>> type(None)
<class 'NoneType'>
```

```
>>> 1_000_000_000_000
1000000000000
>>> 1e100 == 10 ** 100
False
>>> float('inf') > 10**1000
True
>>> 'abc' == b'abc'
False
>>> r'x\ny' == 'x\\ny'
True
```


Many languages lack the literals that Python has.
Lack of type declarations, good and bad. Make it readable.
Prefer d = dict() for interviewers unfamiliar with Python. Spell it out!
None has its own type. Not like null in other languages.
Underscores for thousands separators.
Remember floating-point gotchas.
Floating-point infinity is greater than all integers. Even if they can't be represented in floating point.
Python strings are unicode. Bytes are different.
Raw strings are WYSIWYG. Regular strings are WYSIWYM.
There is no TreeMap as in Java for Python. Nothing for log(n) lookups.

yield from

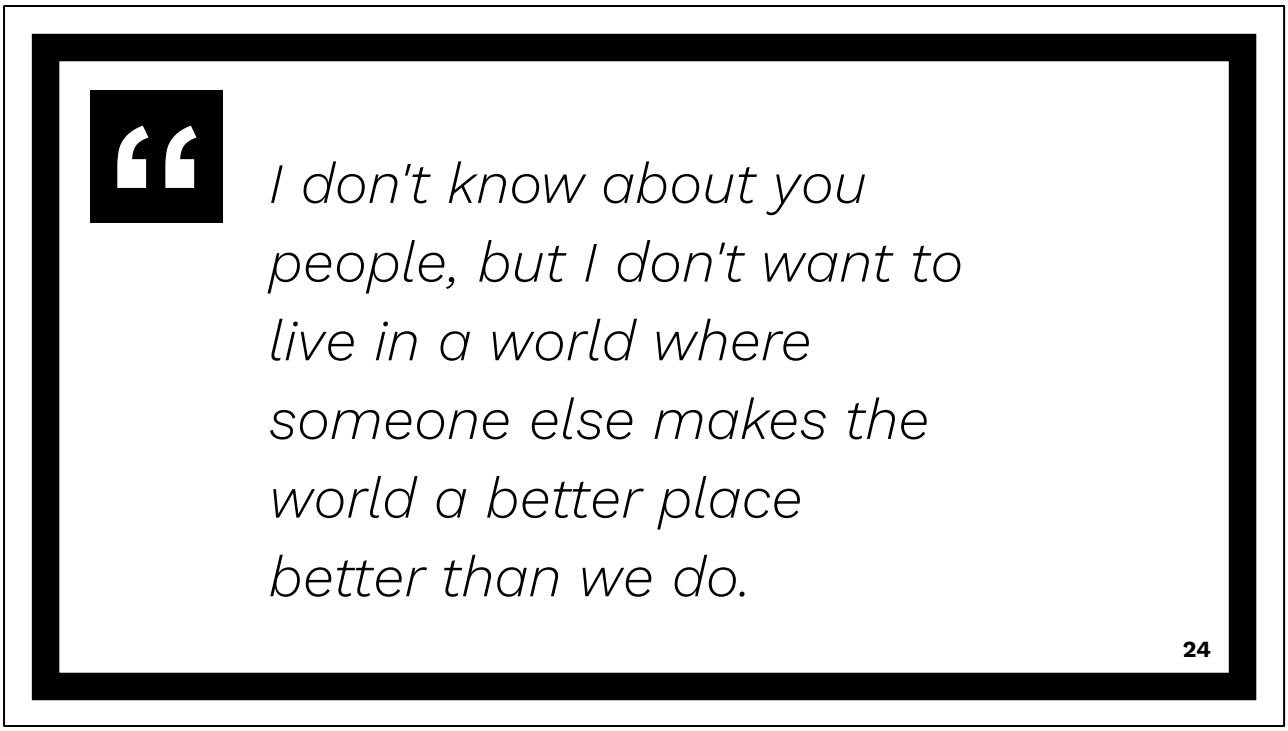
```
fields = ['value', 'left', 'right']
Node = namedtuple('Node', fields)
def in_order_traversal(node):
    "Binary tree in-order iterator."
    if node is None:
        return
    yield from in_order_traversal(node.left)
    yield node.value
    yield from in_order_traversal(node.right)
```


In-order binary tree traversal using generator and “yield from”.
Beware of intermediate topics like iterators and generators.

RecursionError

```
>>> def factorial(n):
...     return 1 if n == 1 else n * factorial(n - 1)
>>> factorial(10_000)
RecursionError: maximum recursion depth exceeded ...
>>> def factorial(n):
...     nums = range(1, n + 1)
...     return functools.reduce(operator.mul, nums)
>>> factorial(10_000)
28462596809170545189064132121198688901480514017 02...
```


Lot's of interview questions require recursion.
No tail recursion :(
Recursion limit can be changed. But if you're getting into that then you're probably off in the weeds.
Python has its own so-called "Zen". Don't fight it. Embrace it.



Fictional character Gavin Belson from HBO’s Silicon Valley.
Companies are competitive. Hiring is extremely competitive.
Good news/bad news of hiring — it’s based on a curve. You don’t have to be good at something just better than everyone else :)
Also an easy tip: read the job req and all the recruiter notes.
Story: prep notes mentioned the four isolation levels provided by SQL. So when I interviewed, I had them memorized and I found a way to work them into the conversation with one of my interviewers.

It Costs Money to Make Money

- $240 LinkedIn Premium Career
- $249 Interview Cake Course
- $159 LeetCode Premium Subscription
- $50 StrengthsFinder Assessment

25

$698 spent in explicit costs.
Checkers Story — Ok to lose a piece if it means gaining two.
LinkedIn Premium — Resume Builder, InMails, Who's Viewed
Interview Cake — Do the whole thing. Great walkthroughts. Solid approach. Not enough alone.
LeetCode Premium — Questions are standardized at companies. Employees may only ask questions from an internal list.
CliftonStrengths Assessment — Classic question: What are your strengths? What are your weaknesses?
Take Strengths Finder and look at top 5-10 and look at Blind Spots section.

I wish you all good fortune in hiring. Go now with Python and get hired! Thanks for letting me share.

Appendix

1.

Python

What is the air-speed velocity of an unladen swallow?
What do you mean? An African or a European swallow?

Python is an ideal language for many coding interviews!
Let's review some core Python features for technical interviews.

import builtins

`MutableSequence`, `Sequence`

- `list`, `tuple`, `str`, `bytes`, `range`, `deque`

`MutableMapping`

- `dict`, `Counter`, `defaultdict`, `ChainMap`

`MutableSet`, `Set`

- `set`, `frozenset`

Built-in data types fit in three categories: Sequence, Mapping, and Set.
These are table stakes. Need a working knowledge of the API and deeper knowledge of implementation.
Often referred to as the ABCs or Abstract Base Classes.
Most sequences are laid out in a memory as a continuous array of pointers or data.
The “range” built-in is a special one. Computes its values. Requires only a few dozen bytes regardless of size.
The “deque” built-in is also special. That’s a doubly-linked list of nodes. Supports fast edits at both ends.
Most mappings are containers of key and value pairs called items. The keys are indexed by hashing.
Most sets are containers of values. The values are indexed by hashing.

How to learn from standard library:
1. Read the docs: collections, itertools, functools
2. Type the examples in yourself and experiment
3. Use them in your code
4. Read the source (be careful here)

O(1) & O(n)

```
>>> %timeit range(100_000)
331 ns
>>> %%timeit -n 1 -r 777 r = range(1_000)
... list(r)
17.8 µs
>>> %%timeit -n 1 -r 777 r = range(100_000)
... list(r)
2.52 ms
```


Story: Teach someone to fish, feed them for a lifetime.
Best way to figure out Big-O is to measure.
Sound smart by calling it “asymptotic time complexity.”
IPython has the fantastic “%timeit” magic command.
Careful when modifying state (like append). Use “-n” and “-r” switches.
When it comes to built-ins, only need to know two classes:
O(1) which is fast, like instant, regardless of container length.
O(n) which is linear, proportional to container length.

Identity & Comparison

```
>>> a = 1234567890
>>> hex(id(a))
'0x109b70ad0'
>>> b = 1234567890
>>> hex(id(b))
'0x109b70af0'
>>> a is b
False
```

```
>>> a == b
True
>>> a.__eq__(b)
True
>>> hash('abc')
3677984258003592727
>>> (1, 'a') < (2, 'b')
True
```


Everything in Python is an object. Ask for its id() or compare using “is”.
Equality can be customized. Operator overloading! Operators: “==” and “!=”.
Implemented by dunder methods like __eq__.
Is there a dunder method for “id()” like __id__? No. Can’t be customized.
Hashing used by dictionaries and sets. If two objects are equal then equal hashes.
Opposite is not true. Equal hashes does not guarantee equal objects. But it’s likely.
Sequences are ordered lexicographically. Like words in English: “apple” before “banana”, and “aardvark” before “apple”.