UNIVERSITY OF SAN FRANCISCO

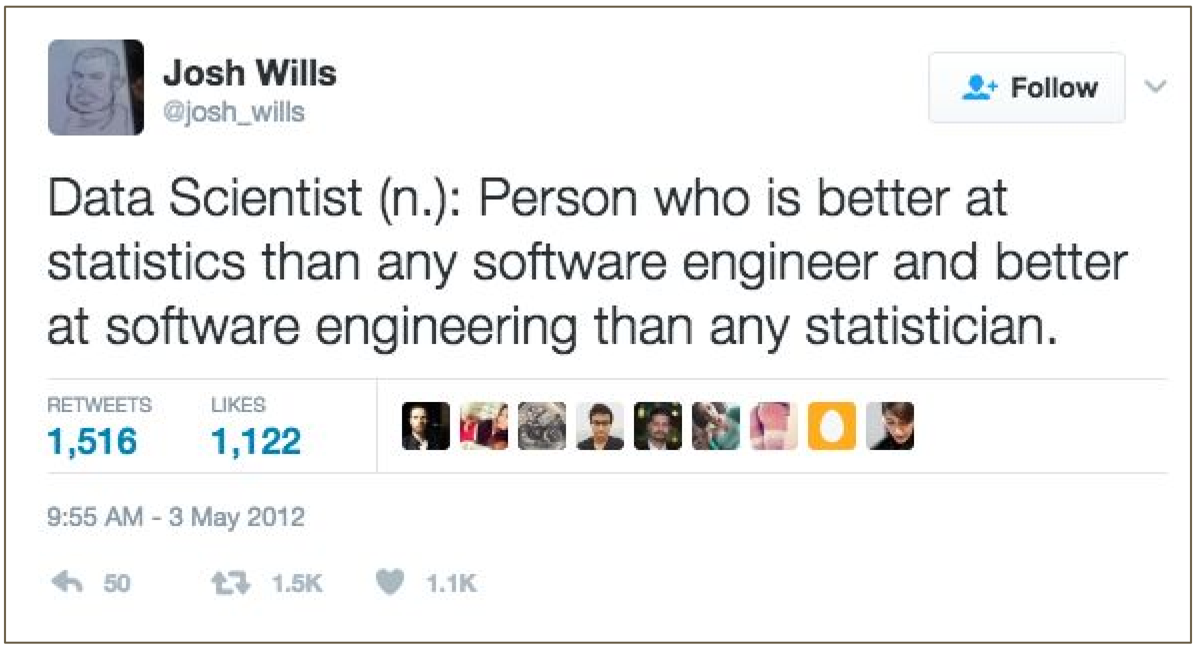
Master of Science in Analytics

# Algorithms

## Interview Skills

# Why Study Algorithms?

- Many people interviewing for data science are computer scientists
  - Until data science comes into its own as a field, many interviewers will be software engineers, database engineers, etc.
  - Algorithms sits at the core of computer science
- Many topics from Algorithms apply to data science
  - Ideas from computational complexity apply to databases (SQL, etc.)
  - The expectation is that you can do both well



# All These Things and More...

- All algorithms and data structures are standard for a CS curriculum
  - All are available in standard CS references; all are available online
  - Some are variations
  - See [Problem Solving with Algorithms and Data Structures using Python]
- What good coders do
  - Understand the concept, memorise the steps
  - Avoid the urge to memorise code

# Arrays

- An array is a (memory-contiguous) series of objects sharing the same type
  - In Python: List
  - Possibly in 1 dimension (vector), 2D (matrix), or N-dimensions (tensor)
- Access
  - Can access any element by name and offset (for example A[5])
  - First element is usually at index 0 (though there are some 1-based languages)
- Assumptions
  - Access to any part of the array can be performed in one read
  - Read operations and write operations have equal cost

# Task: Search

- Task
  - Given: an array (A), a target value (target)
  - Return the index of target in A
  - Return -1 if target is not in A
- Algorithm: Linear Search
  - Iterate from the first element to the last, keeping track of the index
  - At each iteration, if the element is equal to the target, return the index
  - If no element is found at the end of the iteration, return -1
  - Used to implement "in" ala "`if letter in word`"
- Python implementation:

```
def linear_search (A, target):
    for i in range(len(A)):
        if A[i] == target:
            return i
    return -1
```

# Another Search Solution

- Algorithm: Binary Search
  - Same task for search, but assume A is sorted in non-decreasing order
  - Look for the target at the middle of the array; if the middle is equal to the target, return it
  - Recalculate the range where the target may be
    - If the target is greater than the middle element, ignore the lower half of the array
    - If the target is less, ignore the upper half
  - If “upper” and “lower” cross, return -1
- Solutions:
  - There are recursive and iterative implementations
  - The recursive implementation may be more common, but the iterative implementation is more efficient

# Iterative Binary Search (Python)

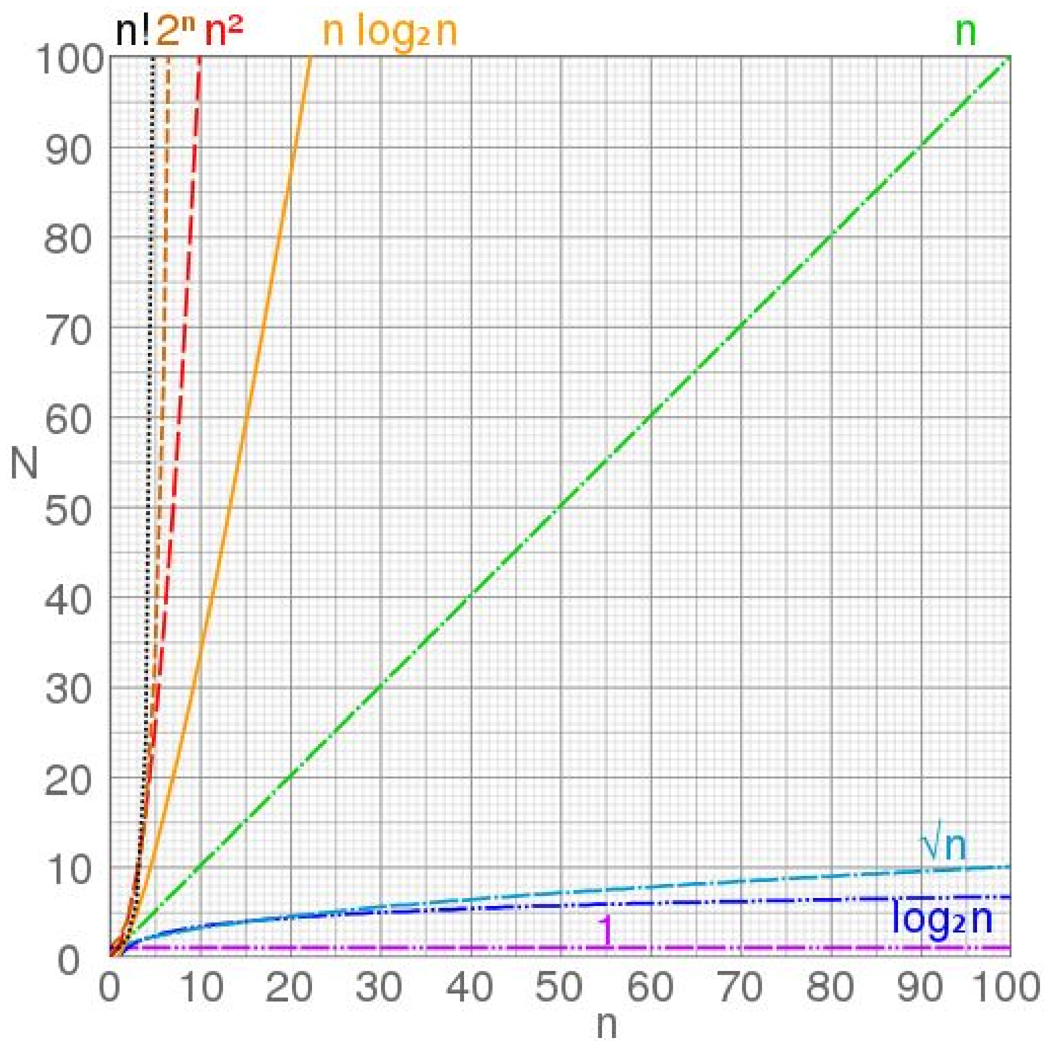
```
def bin_search (A, target):
    lower = 0
    upper = len(A) - 1
    while (lower <= upper):
        middle = (lower + upper) / 2
        if A[middle] == target:
            return middle
        else:
            if target > A[middle]:
                upper = middle - 1
            else:
                lower = middle + 1
    return -1
```

# Computational Complexity

- Compare algorithms primarily by number of array reads and writes
  - Using the number of items in the array (n), the worst case for search (item not in array)
    - Linear search reads all n items
    - Binary search reads about $\log_2(n)$ items
  - Use Big-O notation
    - Formally, if *f* and *g* are two functions, we can say $f(x) = O(g(x))$ if $f(x) <= M(g(x))$ for all $x > x_0$, given a constant $M$
    - Informally: throw out constants, lower-order terms
- Complexity
  - Common to use “worst case” running time
  - It is also possible to use best case, average case
  - Always use “worst case” unless it is possible to prove that it is uncommon
  - Search for our search algorithms (worst case):
    - Linear Search = O(n)
    - Binary Search = O(log n)

# Differences in Speed

- When the size of the array is small, speed differences are negligible
- With large values of n (thousands, millions, etc.), differences are stark



# Task: Sort

- Task
  - Given: an array (A)
  - Reorder elements in A in non-decreasing order
- In-place algorithms:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
- See VisuAlgo for animations
- All use some version of “swap(...)”; in Python:

```
def swap (A, pos1, pos2):
    temp = A[pos1]
    A[pos1] = A[pos2]
    A[pos2] = temp
```

# Selection Sort

- Algorithm:
  - Find the highest-valued item and place it in the last position
  - Eliminate last array position and fill second-to-last position with highest-valued item
  - Fill all subsequent positions similarly
- Python implementation

```
def selection_sort (A):
    for fillslot in range(len(A)-1, 0, -1):
        max = 0
        for location in range(1, max+1):
            if A[location] > A[max]:
                max = location
        swap(A, fillslot, max)
```

- Running time: $O(n^2)$

# Bubble Sort

- Algorithm:
  - Look at each pair of items; swap if in not in order
  - Repeat n-1 times
- Python implementation

```
def bubble_sort (A):
    for pass in range(len(A)-1, 0, -1):
        for i in range(pass):
            if A[i] > A[i+1]:
                swap(A, i, i+1)
```

- Running time: $O(n^2)$
  - Efficiency: stop sorting when there are no swaps
  - Running time is unchanged

# Insertion Sort

- Algorithm:
  - Assume part (first item) of the array is sorted
  - Insert one (the next) item from the unsorted part and shift (if necessary) to maintain sorted order
  - Repeat for all subsequent positions similarly
- Python implementation

```
def insertion_sort (A):
    for index in range(1, len(A)):
        currval = A[index]
        position = index
        while position>0 and A[position-1]>currval
            A[position] = A[position-1]
            position -= 1
        A[position] = curval
```

- Running time: $O(n^2)$

# Merge Sort — Overview

- Algorithm:
  - Recursively split A in half into subarrays until there are 0 or 1 items (which is sorted)
  - Merge elements in adjacent subarrays by selecting lowest item and placing it in the first position
  - Add remaining elements from left half or right half
- Algorithm is recursive
  - Divide-and-conquer
  - Easy split; difficult merge

# Merge Sort — Split

- Algorithm:
  - Divide-and-conquer algorithm
  - 1: Recursively split A in half into subarrays until there are 0 or 1 items (which is sorted)
- Python implementation

```
def merge_sort (A):
    if len(A)>1:
        mid = len(A)//2
        lefthalf = A[:mid]
        righthalf = A[mid:]

    merge_sort(lefthalf)
    merge_sort(righthalf)
```

# Merge Sort — Merge

- Algorithm:
  - Step 2: merge
  - Merge elements in adjacent subarrays by selecting lowest item and placing it in the first position
- Python implementation

```
# Array is now split
i=j=k=0
while i < len(lefthalf) and j < len(righthalf):
    if lefthalf[i] < righthalf[j]:
        A[k] = lefthalf[i]
        i += 1
    else:
        A[k] = righthalf[j]
        j += 1
    k += 1
```

# Merge Sort — Add Remaining

- Algorithm:
  - Step 3: add remaining elements
  - Add remaining elements from left half or right half
- Python implementation

```
# Array is now split and (mostly) merged
while i < len(lefthalf):
    A[k] = lefthalf[i]
    i += 1
    k += 1
while j < len(righthalf):
    A[k] = righthalf[j]
    j += 1
    k += 1
```

# Merge Sort — Review

- Algorithm:
  - Recursively split A in half into subarrays until there are 0 or 1 items (which is sorted)
  - Merge elements in adjacent subarrays by selecting lowest item and placing it in the first position
  - Add remaining elements from left half or right half
- Running time: $O(n \log_2 n)$
  - Each step divides array exactly in half and creates a new level of recursion
  - Each level of recursion perform O(n) reads / writes

# Quick Sort — Overview

- Algorithm:
  - Pick a pivot value from A
  - Partition: move all elements less than pivot to left part of (sub-)array; all elements greater than pivot to right portion
  - Recursively quick sort left part and right part
  - Difficult split; easy merge
- Python implementation

```
def quick_sort (A):
    qs(A, 0, len(A)-1)

def qs(A, first, last):
    if first<last:
        split = partition(A, first, last)
        qs(A, first, split-1)
        qs(A, split+1, last)
```

# Quick Sort — Partition

```
def partition (A, first, last):
    pivot = A[first]  # Other ways to select pivot?
    left = first+1
    right = last
    done = False
    while not done:
        while left <= right and A[left] < pivot
            left += 1
        while A[right] >= pivot and right >= left:
            right -= 1
        if right < left:
            done = True
        else:
            swap(A, left, right)
    swap(A, first, right)
    return right
```

# Quick Sort — Review

- Algorithm:
  - Difficult split; easy merge
  - Partition array and recursively quick sort the left and right halves
- Running time: $O(n \log_2 n)$
  - Complicated analysis because the pivot does not appear in a stable place in the array
  - Worst case: pivot is consistently on left side or right side of array: n levels of recursion, each ~ $O(n)$ reads/writes —> $O(n^2)$
  - The worst case may be rare ~ $(1/n^2)$?
  - The average case is $O(n \log n)$

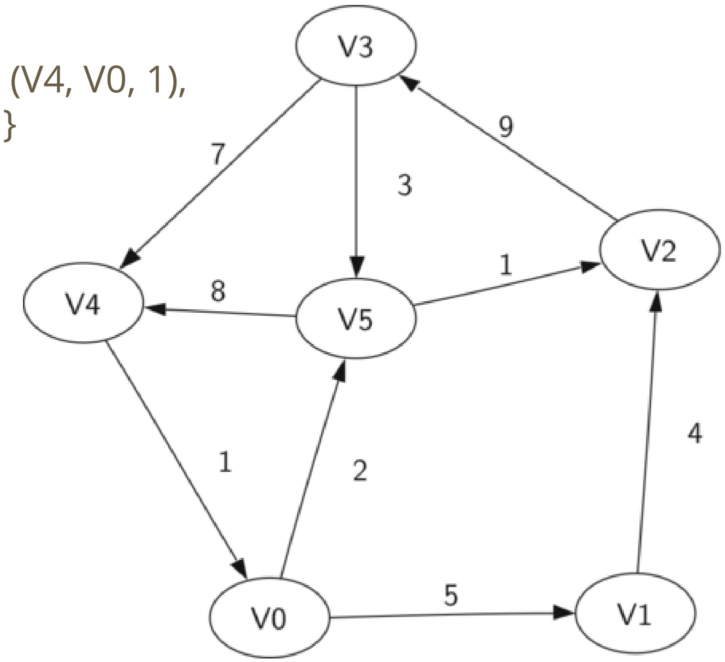
# Hashing

- Concept
  - Store values in an array of (static) size; each entry with a unique position (slot)
  - 1: Convert array index to unique value by scrambling — eg. 54 = 5 + 4 ⇒ 9
  - 2: Ensure index of items being stored fit in array — eg. 54 ⇒ 9 % len(A)
- Hash function
  - Function to convert values to hash positions
  - Strings can be converted to numbers by ASCII value, etc.
  - Should be quick to calculate, result in the fewest collisions (two or more values hashing to the same location)
  - *Folding:* divide key into groups and add individual portions — eg. 415-422-5101 ⇒ [41 + 54 + 22 + 51 + 01] ⇒ 169
  - *Mid-squaring:* square the key and take middle portion — eg. $44^2$ ⇒ 1936 ⇒ 93
- Resolving collisions
  - Rehashing — eg. new_hash = hash(old_hash_value)
  - Open addressing — place an element in next available slot (linear, quadratic, etc.)
  - Chaining — Each slot is a structure (array?) of items — example here

# Graphs — Concept and Storage

- Vocabulary: *G = (V, E)*
  - *Vertices* / nodes; may have a key / name + additional information
  - *Edges* connect vertices; may be weighted or unweighted; directed or bidirectional
  - *Path* — a sequence of vertices connected by edges (in the correct direction)
  - *Cycle* — a path which begins and ends with the same vertex
- Example
  - V = {V1, V1, V2, V3, V4, V5}
  - E = { (V0, V1, 5), (V1, V2, 4), (V2, V3, 9), (V3, V4, 7), (V4, V0, 1),
    (V0, V5, 2), (V5, V4, 8), (V3, V5, 3), (V5, V2, 1) }
- May be stored as a matrix (below) or list

| | V0 | V1 | V2 | V3 | V4 | V5 |
|---|---|---|---|---|---|---|
| V0 | | 5 | | | | 2 |
| V1 | | | 4 | | | |
| V2 | | | | 9 | | |
| V3 | | | | | 7 | 3 |
| V4 | 1 | | | | | |
| V5 | | | 1 | | 8 | |



# Graphs — One Implementation

```
# Only a subset of method implementations are shown

class Vertex:
    def __init__(self, key):
        self.id = key
        self.connected_to = {}

    def add_neighbour(self, nbr, weight=0)
        self.connected_to[nbr] = weight

class Graph:
    def __init__(self):
        self.vertex_list = {}
        self.vertex_count = 0

    def get_vertex(self, key):
        return self.vertex_list[key]
```

# Task: Find Path

- Task
  - Given: two vertices, ($V_S$ and $V_T$) and G = (V, E)
  - Find and return a (best?) path starting at $V_S$ and ending at $V_T$
- Algorithm: Breadth First Search
  - Visit adjacent vertices recursively, starting from $V_S$ (by weight?) until $V_T$ is found
  - Keep a queue  of candidate nodes to visit, set of nodes already visited
  - Some implementations require additional methods to Vertex class
  - Running time = O(V + E)
  - Ignoring $V_T$, the same algorithm can create a (minimum) spanning tree from $V_S$

# BFS in Python

```
def bfs(G, start, target):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in G.get_vertex(vertex) - set(path):
            if next == target:
                yield path + [next]
            else:
                queue.append((next, path + [next]))
```

See http://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/ and others

# Stacks

- Concept
  - LIFO data structure
  - Interface:
    - push(X) — places an item on (the top of) the Stack
    - pop() — removes and returns the item from (the top of) the Stack
    - peek() — returns the item from (the top of) the Stack
    - empty() — returns True if the Stack is empty (a/k/a “is_empty()”)
- Implementation
  - May be implemented as a linked list, etc.
  - Common to implement as an array — O(1) for all operations — eg. List (which has a “pop()” method)

# Queues

- Concept
  - FIFO data structure
  - Interface:
    - enqueue(X) — places an item on (the end of) the Queue
    - dequeue() — removes and returns an item from the start of the Queue
    - size() — returns the number of items on the Queue
  - Variation: dequeue allows insertion & removal from both front & back
- Implementation
  - May be implemented as a linked list, etc.
  - Common to implement as a circular array — O(1) for enqueue(...), dequeue()
- Exercise
  - If we have time, write “Queue” in Python

# Interview Guidelines

- Interviewer will select question for candidate
- Recorded answer limited to 5-minute answer
  - Feel free to answer up to 10-12 mins, but only the first 5 minutes will be recorded
  - Candidates are allowed 1 retry
- As mentioned before, Hour 2 room assignments have changed

# Getting to the Assignment