Basic Types of Algorithms

Recursive Algorithms

Definition:
- An algorithm that calls itself in its definition.
  - **Recursive case** a conditional statement that is used to trigger the recursion.
  - **Base case** a conditional statement that is used to break the recursion.

What you need to know:
- **Stack level too deep** and **stack overflow**.
  - If you've seen either of these from a recursive algorithm, you messed up.
  - It means that your base case was never triggered because it was faulty or the problem was so massive you ran out of alloted memory.
  - Knowing whether or not you will reach a base case is integral to correctly using recursion.
  - Often used in Depth First Search

Iterative Algorithms

Definition:
- An algorithm that is called repeatedly but for a finite number of times, each time being a single iteration.
  - Often used to move incrementally through a data set.

What you need to know:
- Generally you will see iteration as loops, for, while, and until statements.
- Think of iteration as moving one at a time through a set.
- Often used to move through an array.

Recursion Vs. Iteration :
- The differences between recursion and iteration can be confusing to distinguish since both can be used to implement the other. But know that,
  - Recursion is, usually, more expressive and easier to implement.
  - Iteration uses less memory.
- **Functional languages** tend to use recursion. (i.e. Haskell)
- **Imperative languages** tend to use iteration. (i.e. Ruby)
- Check out this [Stack Overflow post](http://stackoverflow.com/questions/19794739/what-is-the-difference-between-iteration-and-recursion) for more info.

Pseudo Code of Moving Through an Array (this is why iteration is used for this)
```
Recursion                          | Iteration
-----------------------------------|----------------------------------
recursive method (array, n)        | iterative method (array)
  if array[n] is not nil           |   for n from 0 to size of array
    print array[n]                 |     print(array[n])
    recursive method(array, n+1)   |
  else                             |
    exit loop                      |
```

Greedy Algorithm

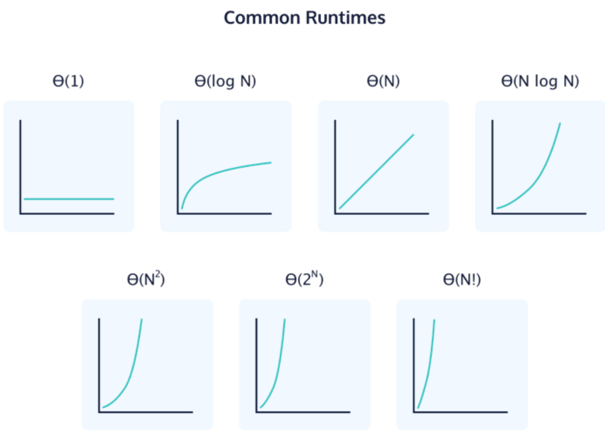
Definition:
- An algorithm that, while executing, selects only the information that meets a certain criteria.
- The general five components, taken from [Wikipedia](http://en.wikipedia.org/wiki/Greedy_algorithm#Specifics):
  - A candidate set, from which a solution is created.
  - A selection function, which chooses the best candidate to be added to the solution.
  - A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
  - An objective function, which assigns a value to a solution, or a partial solution.
  - A solution function, which will indicate when we have discovered a complete solution.

What you need to know:
- Used to find the expedient, though non-optimal, solution for a given problem.
- Generally used on sets of data where only a small proportion of the information evaluated meets the desired result.
- Often a greedy algorithm can help reduce the Big O of an algorithm.

Pseudo Code of a Greedy Algorithm to Find Largest Difference of any Two Numbers in an Array.
```
greedy algorithm (array)
  var largest difference = 0
  var new difference = find next difference (array[n], array[n+1])
  largest difference = new difference if new difference is > largest difference
  repeat above two steps until all differences have been found
  return largest difference
```
This algorithm never needed to compare all the differences to one another, saving it an entire iteration.

Algorithmic Common Runtimes :
The common algorithmic runtimes from fastest to slowest are:

- constant: $\Theta(1)$
- logarithmic: $\Theta(\log N)$
- linear: $\Theta(N)$
- polynomial: $\Theta(N^2)$
- exponential: $\Theta(2^N)$
- factorial: $\Theta(N!)$



Hash Table or Hash Map

Definition:
- Stores data with key value pairs.
- **Hash functions** accept a key and return an output unique only to that specific key.
  - This is known as **hashing**, which is the concept that an input and an output have a one-to-one correspondence to map information.
  - Hash functions return a unique address in memory for that data.
  - Our hash map implementation then takes that hash value mod the size of the array.

What you need to know:
- Designed to optimize searching, insertion, and deletion.
- **Hash collisions** are when a hash function returns the same output for two distinct inputs.
  - All hash functions have this problem.
  - This is often accommodated for by having the hash tables be very large.
- Hashes are important for associative arrays and database indexing.

Colision stategy:
- The first strategy we're going to learn about is called *separate chaining*. The separate chaining strategy avoids collisions by updating the underlying data structure. Instead of an array of values that are mapped to by hashes, it could be an array of linked lists! If a linked list already exists at the address, append the value to the linked list given. This is effective for hash functions that are particularly good at giving unique indices, so the linked lists never get very long. But in the worst-case scenario, where the hash function gives all keys the same index, lookup performance is only as good as it would be on a linked list. If we save both the key and the value, then we will be able to check against the saved key when we're accessing data in a hash map.

- Another popular hash collision strategy is called *open addressing*. In open addressing we stick to the array as our underlying data structure, but we continue looking for a new index to save our data if the first result of our hash function has a different key's data.
    - A common open method of open addressing is called *probing*. Probing means continuing to find new array indices in a fixed sequence until an empty index is found.
    - In a quadratic probing open addressing system, we add increasingly large numbers to the hash code. At the first collision we just add 1, but if the hash collides there too we add 4 ,and the third time we add 9. Having a probe sequence change over time like this avoids clustering.
    - *Clustering* is what happens when a single hash collision causes additional hash collisions. Imagine a hash collision triggers a linear probing sequence to assigns a value to the next hash bucket over. Any key that would hash to this "next bucket" will now collide with a key that, in a sense, doesn't belong to that bucket anyway.

Time Complexity:
- Indexing: Hash Tables: O(1)
- Search: Hash Tables: O(1)
- Insertion: Hash Tables: O(1)

Code sample:

```
class HashMap:
  def __init__(self, array_size):
    self.array_size = array_size
    self.array = [None for item in range(array_size)]

  def hash(self, key, count_collisions=0):
    key_bytes = key.encode()
    hash_code = sum(key_bytes)
    return hash_code + count_collisions

  def compressor(self, hash_code):
    return hash_code % self.array_size

  def assign(self, key, value):
    array_index = self.compressor(self.hash(key))
    current_array_value = self.array[array_index]

    if current_array_value is None:
      self.array[array_index] = [key, value]
      return

    if current_array_value[0] == key:
      self.array[array_index] = [key, value]
      return

    # Collision!

    number_collisions = 1

    while(current_array_value[0] != key):
      new_hash_code = self.hash(key, number_collisions)
      new_array_index = self.compressor(new_hash_code)
      current_array_value = self.array[new_array_index]

      if current_array_value is None:
        self.array[new_array_index] = [key, value]
        return

      if current_array_value[0] == key:
        self.array[new_array_index] = [key, value]
        return

      number_collisions += 1

    return

  def retrieve(self, key):
    array_index = self.compressor(self.hash(key))
    possible_return_value = self.array[array_index]

    if possible_return_value is None:
      return None

    if possible_return_value[0] == key:
      return possible_return_value[1]

    retrieval_collisions = 1

    while (possible_return_value != key):
      new_hash_code = self.hash(key, retrieval_collisions)
```

```
  retrieving_array_index = self.compressor(new_hash_code)
  possible_return_value = self.array[retrieving_array_index]

  if possible_return_value is None:
    return None

  if possible_return_value[0] == key:
    return possible_return_value[1]

  number_collisions += 1

return
```

Linear Data Structure Basics

Array

Definition:
- Stores data elements based on an sequential, most commonly 0 based, index.
- Based on [tuples](http://en.wikipedia.org/wiki/Tuple) from set theory.
- They are one of the oldest, most commonly used data structures.

What you need to know:
- Optimal for indexing; bad at searching, inserting, and deleting (except at the end).
- **Linear arrays**, or one dimensional arrays, are the most basic.
  - Are static in size, meaning that they are declared with a fixed size.
- **Dynamic arrays** are like one dimensional arrays, but have reserved space for additional elements.
  - If a dynamic array is full, it copies its contents to a larger array.
- **Multi dimensional arrays** nested arrays that allow for multiple dimensions such as an array of arrays providing a 2 dimensional spacial representation via x, y coordinates.

Time Complexity:
- Indexing: Linear array: O(1),   Dynamic array: O(1)
- Search: Linear array: O(n),   Dynamic array: O(n)
- Optimized Search: Linear array: O(log n), Dynamic array: O(log n)
- Insertion: Linear array: n/a   Dynamic array: O(n)

Iterative Binary Search

```
def binary_search(sorted_list, target):
  left_pointer = 0
  right_pointer = len(sorted_list)

  # fill in the condition for the while loop
  while left_pointer < right_pointer:
    # calculate the middle index using the two pointers
    mid_idx = (left_pointer + right_pointer) // 2
    mid_val = sorted_list[mid_idx]
    if mid_val == target:
      return mid_idx
    if target < mid_val:
      # set the right_pointer to the appropriate value
      right_pointer = mid_idx
    if target > mid_val:
      # set the left_pointer to the appropriate value
      left_pointer = mid_idx + 1

  return "Value not in list"
}
```

Linked List

Definition:
- Stores data with **nodes** that point to other nodes.
  - Nodes, at its most basic it has one datum and one reference (another node).
  - A linked list _chains_ nodes together by pointing one node's reference towards another node.

What you need to know:
- Designed to optimize insertion and deletion, slow at indexing and searching.
- **Doubly linked list** has nodes that also reference the previous node.
- **Circularly linked list** is simple linked list whose **tail**, the last node, references the **head**, the first node.
- **Stack**, commonly implemented with linked lists but can be made from arrays too.
  - Stacks are **last in, first out** (LIFO) data structures.
  - Made with a linked list by having the head be the only place for insertion and removal.
- **Queues**, too can be implemented with a linked list or an array.
  - Queues are a **first in, first out** (FIFO) data structure.
  - Made with a doubly linked list that only removes from head and adds to tail.

Time Complexity:
- Indexing: Linked Lists: O(n)
- Search: Linked Lists: O(n)
- Optimized Search: Linked Lists: O(n)
- Insertion: Linked Lists: O(1)

Code Sample:
```
class Node:
  def __init__(self, value, next_node=None, prev_node=None):
    self.value = value
    self.next_node = next_node
    self.prev_node = prev_node

  def set_next_node(self, next_node):
    self.next_node = next_node

  def get_next_node(self):
    return self.next_node

  def set_prev_node(self, prev_node):
    self.prev_node = prev_node

  def get_prev_node(self):
    return self.prev_node

  def get_value(self):
    return self.value

class DoublyLinkedList:
  def __init__(self):
    self.head_node = None
    self.tail_node = None

  def add_to_head(self, new_value):
    new_head = Node(new_value)
    current_head = self.head_node

    if current_head != None:
      current_head.set_prev_node(new_head)
      new_head.set_next_node(current_head)

    self.head_node = new_head

    if self.tail_node == None:
      self.tail_node = new_head
```

```
def add_to_tail(self, new_value):
  new_tail = Node(new_value)
  current_tail = self.tail_node

  if current_tail != None:
    current_tail.set_next_node(new_tail)
    new_tail.set_prev_node(current_tail)

  self.tail_node = new_tail

  if self.head_node == None:
    self.head_node = new_tail

def remove_head(self):
  removed_head = self.head_node

  if removed_head == None:
    return None

  self.head_node = removed_head.get_next_node()

  if self.head_node != None:
    self.head_node.set_prev_node(None)

  if removed_head == self.tail_node:
    self.remove_tail()

  return removed_head.get_value()

def remove_tail(self):
  removed_tail = self.tail_node

  if removed_tail == None:
    return None

  self.tail_node = removed_tail.get_prev_node()

  if self.tail_node != None:
    self.tail_node.set_next_node(None)

  if removed_tail == self.head_node:
    self.remove_head()

  return removed_tail.get_value()

def remove_by_value(self, value_to_remove):
  node_to_remove = None
  current_node = self.head_node

  while current_node != None:
    if current_node.get_value() == value_to_remove:
      node_to_remove = current_node
      break

    current_node = current_node.get_next_node()

  if node_to_remove == None:
    return None

  if node_to_remove == self.head_node:
```

```
      self.remove_head()
    elif node_to_remove == self.tail_node:
      self.remove_tail()
    else:
      next_node = node_to_remove.get_next_node()
      prev_node = node_to_remove.get_prev_node()
      next_node.set_prev_node(prev_node)
      prev_node.set_next_node(next_node)

    return node_to_remove

  def stringify_list(self):
    string_list = ""
    current_node = self.head_node
    while current_node:
      if current_node.get_value() != None:
        string_list += str(current_node.get_value()) + "\n"
      current_node = current_node.get_next_node()
    return string_list
```

Queue

Sample code:

```
class Node:
  def __init__(self, value, next_node=None):
    self.value = value
    self.next_node = next_node

  def set_next_node(self, next_node):
    self.next_node = next_node

  def get_next_node(self):
    return self.next_node

  def get_value(self):
    return self.value

class Queue:
  def __init__(self, max_size=None):
    self.head = None
    self.tail = None
    self.max_size = max_size
    self.size = 0

  def enqueue(self, value):
    if self.has_space():
      item_to_add = Node(value)
      print("Adding " + str(item_to_add.get_value()) + " to the queue!")
      if self.is_empty():
        self.head = item_to_add
        self.tail = item_to_add
      else:
        self.tail.set_next_node(item_to_add)
        self.tail = item_to_add
      self.size += 1
    else:
      print("Sorry, no more room!")

  def dequeue(self):
    if self.get_size() > 0:
      item_to_remove = self.head
```

```
      print(str(item_to_remove.get_value()) + " is served!")
      if self.get_size() == 1:
        self.head = None
        self.tail = None
      else:
        self.head = self.head.get_next_node()
      self.size -= 1
      return item_to_remove.get_value()
    else:
      print("The queue is totally empty!")

  def peek(self):
    if self.is_empty():
      print("Nothing to see here!")
    else:
      return self.head.get_value()

  def get_size(self):
    return self.size

  def has_space(self):
    if self.max_size == None:
      return True
    else:
      return self.max_size > self.get_size()

  def is_empty(self):
    return self.size == 0
```

Stack

Sample code:

```
class Node:
  def __init__(self, value, next_node=None):
    self.value = value
    self.next_node = next_node

  def set_next_node(self, next_node):
    self.next_node = next_node

  def get_next_node(self):
    return self.next_node

  def get_value(self):
    return self.value

class Stack:
  def __init__(self, limit=1000):
    self.top_item = None
    self.size = 0
    self.limit = limit

  def push(self, value):
    if self.has_space():
      item = Node(value)
      item.set_next_node(self.top_item)
      self.top_item = item
      self.size += 1
      print("Adding {} to the pizza stack!".format(value))
    else:
      print("No room for {}!".format(value))
```

```
def pop(self):
  if not self.is_empty():
    item_to_remove = self.top_item
    self.top_item = item_to_remove.get_next_node()
    self.size -= 1
    print("Delivering " + item_to_remove.get_value())
    return item_to_remove.get_value()
  print("All out of pizza.")

def peek(self):
  if not self.is_empty():
    return self.top_item.get_value()
  print("Nothing to see here!")

def has_space(self):
  return self.limit > self.size

def is_empty(self):
  return self.size == 0
```

Search Basics :

Merge Sort

Definition:
- A comparison based sorting algorithm
  - Divides entire dataset into groups of at most two.
  - Compares each number one at a time, moving the smallest number to left of the pair.
  - Once all pairs sorted it then compares left most elements of the two leftmost pairs creating a sorted group of four with the smallest numbers on the left and the largest ones on the right.
  - This process is repeated until there is only one set.

What you need to know:
- This is one of the most basic sorting algorithms.
- Know that it divides all the data into as small possible sets then compares them.

Time Complexity:
- Best Case Sort: Merge Sort: O(n)
- Average Case Sort: Merge Sort: O(n log n)
- Worst Case Sort: Merge Sort: O(n log n)

Code sample:

```
def merge_sort(items):
  if len(items) <= 1:
    return items

  middle_index = len(items) // 2
  left_split = items[:middle_index]
  right_split = items[middle_index:]

  left_sorted = merge_sort(left_split)
  right_sorted = merge_sort(right_split)

  return merge(left_sorted, right_sorted)

def merge(left, right):
  result = []

  while (left and right):
    if left[0] < right[0]:
      result.append(left[0])
      left.pop(0)
    else:
      result.append(right[0])
      right.pop(0)

  if left:
    result += left
  if right:
    result += right

  return result
```

Quicksort

Definition:
- A comparison based sorting algorithm
  - Divides entire dataset in half by selecting the middle element and putting all smaller elements to the left of the element and larger ones to the right.
  - It repeats this process on the left side until it is comparing only two elements at which point the left side is sorted.
  - When the left side is finished sorting it performs the same operation on the right side.
- Computer architecture favors the quicksort process.

What you need to know:
- While it has the same Big O as (or worse in some cases) many other sorting algorithms it is often faster in practice than many other sorting algorithms, such as merge sort.
- Know that it halves the data set by the average continuously until all the information is sorted.

Time Complexity:
- Best Case Sort: Merge Sort: O(n)
- Average Case Sort: Merge Sort: O(n log n)
- Worst Case Sort: Merge Sort: O(n^2)

Merge Sort Vs. Quicksort
- Quicksort is likely faster in practice.
- Merge Sort divides the set into the smallest possible groups immediately then reconstructs the incrementally as it sorts the groupings.
- Quicksort continually divides the set by the average, until the set is recursively sorted.

Code sample:

```
def quicksort(list, start, end):
  # this portion of list has been sorted
  if start >= end:
    return
  print("Running quicksort on {0}".format(list[start: end + 1]))
  # select random element to be pivot
  pivot_idx = randrange(start, end + 1)
  pivot_element = list[pivot_idx]
  print("Selected pivot {0}".format(pivot_element))
  # swap random element with last element in sub-lists
  list[end], list[pivot_idx] = list[pivot_idx], list[end]

  # tracks all elements which should be to left (lesser than) pivot
  less_than_pointer = start

  for i in range(start, end):
    # we found an element out of place
    if list[i] < pivot_element:
      # swap element to the right-most portion of lesser elements
      print("Swapping {0} with {1}".format(list[i],
list[less_than_pointer]))
      list[i], list[less_than_pointer] = list[less_than_pointer], list[i]
      # tally that we have one more lesser element
      less_than_pointer += 1
  # move pivot element to the right-most portion of lesser elements
  list[end], list[less_than_pointer] = list[less_than_pointer], list[end]
  print("{0} successfully partitioned".format(list[start: end + 1]))
```

```
  # recursively sort left and right sub-lists
  quicksort(list, start, less_than_pointer - 1)
  quicksort(list, less_than_pointer + 1, end)
```

Bubble Sort

Definition:
- A comparison based sorting algorithm
  - It iterates left to right comparing every couplet, moving the smaller element to the left.
  - It repeats this process until it no longer moves an element to the left.

What you need to know:
- While it is very simple to implement, it is the least efficient of these three sorting methods.
- Know that it moves one space to the right comparing two elements at a time and moving the smaller on to left.

Time Complexity:
- Best Case Sort: Merge Sort: O(n)
- Average Case Sort: Merge Sort: O(n^2)
- Worst Case Sort: Merge Sort: O(n^2)

Code sample:
```
def swap(arr, index_1, index_2):
  temp = arr[index_1]
  arr[index_1] = arr[index_2]
  arr[index_2] = temp

def bubble_sort_unoptimized(arr):
  iteration_count = 0
  for el in arr:
    for index in range(len(arr) - 1):
      iteration_count += 1
      if arr[index] > arr[index + 1]:
        swap(arr, index, index + 1)

  print("PRE-OPTIMIZED ITERATION COUNT: {0}".format(iteration_count))

def bubble_sort(arr):
  iteration_count = 0
  for i in range(len(arr)):
    # iterate through unplaced elements
    for idx in range(len(arr) - i - 1):
      iteration_count += 1
      if arr[idx] > arr[idx + 1]:
        # replacement for swap function
        arr[idx], arr[idx + 1] = arr[idx + 1], arr[idx]

  print("POST-OPTIMIZED ITERATION COUNT: {0}".format(iteration_count))
```

Non-Linear Data Structure Basics

** Tree**

Definition:
Tree data structures are built using tree nodes (a variation on the nodes you created earlier) and are another way of storing information. Specifically, trees are used for data that has a hierarchical structure, such as a family tree or a computer’s file system. The tree data structure you are going to create is an excellent foundation for further variations on trees, including AVL trees, red-black trees, and binary search trees!

Code sample:
```
class TreeNode:
  def __init__(self, value):
    self.value = value # data
    self.children = [] # references to other nodes

  def add_child(self, child_node):
    # creates parent-child relationship
    print("Adding " + child_node.value)
    self.children.append(child_node)

  def remove_child(self, child_node):
    # removes parent-child relationship
    print("Removing " + child_node.value + " from " + self.value)
    self.children = [child for child in self.children
                     if child is not child_node]

  def traverse(self):
    # moves through each node referenced from self downwards
    nodes_to_visit = [self]
    while len(nodes_to_visit) > 0:
      current_node = nodes_to_visit.pop()
      print(current_node.value)
      nodes_to_visit += current_node.children
```

Binary Tree

Definition:
- Is a tree like data structure where every node has at most two children.
  - There is one left and right child node.

What you need to know:
- Designed to optimize searching and sorting.
- A **degenerate tree** is an unbalanced tree, which if entirely one-sided is a essentially a linked list.
- They are comparably simple to implement than other data structures.
- Used to make **binary search trees**.
  - A binary tree that uses comparable keys to assign which direction a child is.
  - Left child has a key smaller than it's parent node.
  - Right child has a key greater than it's parent node.
  - There can be no duplicate node.
  - Because of the above it is more likely to be used as a data structure than a binary tree.

Time Complexity:
- Indexing: Binary Search Tree: O(log n)
- Search: Binary Search Tree: O(log n)

- Insertion: Binary Search Tree: O(log n)

Code sample:
```
class BinarySearchTree:
  def __init__(self, value, depth=1):
    self.value = value
    self.depth = depth
    self.left = None
    self.right = None

  def insert(self, value):
    if (value < self.value):
      if (self.left is None):
        self.left = BinarySearchTree(value, self.depth + 1)
        print(f'Tree node {value} added to the left of {self.value} at depth {self.depth + 1}')
      else:
        self.left.insert(value)
    else:
      if (self.right is None):
        self.right = BinarySearchTree(value, self.depth + 1)
        print(f'Tree node {value} added to the right of {self.value} at depth {self.depth + 1}')
      else:
        self.right.insert(value)

  def get_node_by_value(self, value):
    if (self.value == value):
      return self
    elif ((self.left is not None) and (value < self.value)):
      return self.left.get_node_by_value(value)
    elif ((self.right is not None) and (value >= self.value)):
      return self.right.get_node_by_value(value)
    else:
      return None

  def depth_first_traversal(self):
    if (self.left is not None):
      self.left.depth_first_traversal()
    print(f'Depth={self.depth}, Value={self.value}')
    if (self.right is not None):
      self.right.depth_first_traversal()
```

Heap

Introduction to Heaps :
Heaps are used to maintain a maximum or minimum value in a dataset. Heaps tracking the maximum or minimum value are *max-heaps* or *min-heaps*. We will focus on min-heaps, but the concepts for a max-heap are nearly identical.

Think of the min-heap as a binary tree with two qualities:
- The root is the minimum value of the dataset.
- Every child's value is greater than or equal to its parent.

These two properties are the defining characteristics of the min-heap. By maintaining these two properties, we can efficiently retrieve and update the minimum value.

Notice how by filling the tree from left to right; we're leaving no gaps in the array. The location of each child or parent derives from a formula using the index.
- left child: (index * 2) + 1

- right child: (index * 2) + 2
- parent: (index - 1) / 2 — not used on the root!

Adding an Element: Heapify Up :

Sometimes you will add an element to the heap that violates the heap's essential properties.
We need to restore the fundamental heap properties. This restoration is known as *heapify* or *heapifying*. We're adding an element to the bottom of the tree and moving upwards, so we're *heapifying up*.
As long as we've violated the heap properties, we'll swap the offending child with its parent until we restore the properties, or until there's no parent left. If there is no parent left, that element becomes the new root of the tree.

Removing an Element: Heapify Down :

Maintaining a minimum value is no good if we can never retrieve it, so let's explore how to remove the root node.

In the diagram, you can see removing the top node itself would be messy: there would be two children orphaned! Instead, we'll swap the root node, with the bottom rightmost child: The bottom rightmost child is simple to remove because it has no children.Unfortunately, we've violated the heap property. We'll *heapify down* to
restore the heap property.

This process is similar to heapifying up, except we have two options where we can make a swap. We'll choose the **lesser of the two values** and swap. This is necessary for the heap property. Just like that, we've retrieved the minimum value, allocated a *new* minimum, and maintained the heap property!

Code sample:

```
class MinHeap:
  def __init__(self):
    self.heap_list = [None]
    self.count = 0

  # HEAP HELPER METHODS
  # DO NOT CHANGE!
  def parent_idx(self, idx):
    return idx // 2

  def left_child_idx(self, idx):
    return idx * 2

  def right_child_idx(self, idx):
    return idx * 2 + 1

  # NEW HELPER!
  def child_present(self, idx):
    return self.left_child_idx(idx) <= self.count

  # END OF HEAP HELPER METHODS

  def retrieve_min(self):
    if self.count == 0:
      print("No items in heap")
      return None

    min = self.heap_list[1]
```

```
    print("Removing: {0} from {1}".format(min, self.heap_list))
    self.heap_list[1] = self.heap_list[self.count]
    self.count -= 1
    self.heap_list.pop()
    print("Last element moved to first: {0}".format(self.heap_list))
    self.heapify_down()
    return min

  def add(self, element):
    self.count += 1
    print("Adding: {0} to {1}".format(element, self.heap_list))
    self.heap_list.append(element)
    self.heapify_up()


  def heapify_down(self):
    idx = 1
    while self.child_present(idx):
      print("Heapifying down!")
      smaller_child_idx = self.get_smaller_child_idx(idx)
      child = self.heap_list[smaller_child_idx]
      parent = self.heap_list[idx]
      if parent > child:
        self.heap_list[idx] = child
        self.heap_list[smaller_child_idx] = parent
      idx = smaller_child_idx
    print("HEAP RESTORED! {0}".format(self.heap_list))

  def get_smaller_child_idx(self, idx):
    if self.right_child_idx(idx) > self.count:
      print("There is only a left child")
      return self.left_child_idx(idx)
    else:
      left_child = self.heap_list[self.left_child_idx(idx)]
      right_child = self.heap_list[self.right_child_idx(idx)]
      if left_child < right_child:
        print("Left child is smaller")
        return self.left_child_idx(idx)
      else:
        print("Right child is smaller")
        return self.right_child_idx(idx)

  def heapify_up(self):
    idx = self.count
    while self.parent_idx(idx) > 0:
      if self.heap_list[self.parent_idx(idx)] > self.heap_list[idx]:
        tmp = self.heap_list[self.parent_idx(idx)]
        print("swapping {0} with {1}".format(tmp, self.heap_list[idx]))
        self.heap_list[self.parent_idx(idx)] = self.heap_list[idx]
        self.heap_list[idx] = tmp
      idx = self.parent_idx(idx)
    print("HEAP RESTORED! {0}".format(self.heap_list))
    print("")
```

Graphs and Graphs traversal :

Graph :

Code sample:

```
class Vertex:
  def __init__(self, value):
    self.value = value
    self.edges = {}

  def add_edge(self, vertex, weight = 0):
    self.edges[vertex] = weight

  def get_edges(self):
    return list(self.edges.keys())

class Graph:
  def __init__(self, directed = False):
    self.graph_dict = {}
    self.directed = directed

  def add_vertex(self, vertex):
    self.graph_dict[vertex.value] = vertex

  def add_edge(self, from_vertex, to_vertex, weight = 0):
    self.graph_dict[from_vertex.value].add_edge(to_vertex.value, weight)
    if not self.directed:
      self.graph_dict[to_vertex.value].add_edge(from_vertex.value, weight)

  def find_path(self, start_vertex, end_vertex):
    start = [start_vertex]
    seen = {}
    while len(start) > 0:
      current_vertex = start.pop(0)
      seen[current_vertex] = True
      print("Visiting " + current_vertex)
      if current_vertex == end_vertex:
        return True
      else:
        vertices_to_visit =
set(self.graph_dict[current_vertex].edges.keys())
        start += [vertex for vertex in vertices_to_visit if vertex not in
seen]
    return False
```

Breadth First Search

Definition:

- An algorithm that searches a tree (or graph) by searching levels of the tree first, starting at the root.
  - It finds every node on the same level, most often moving left to right.
  - While doing this it tracks the children nodes of the nodes on the current level.
  - When finished examining a level it moves to the left most node on the next level.
  - The bottom-right most node is evaluated last (the node that is deepest and is farthest right of it's level).

What you need to know:

- Optimal for searching a tree that is wider than it is deep.
- Uses a queue to store information about the tree while it traverses a tree.
  - Because it uses a queue it is more memory intensive than **depth first search**.
  - The queue uses more memory because it needs to stores pointers

Time Complexity:
- Search: Breadth First Search: O(V + E)
- E is number of edges
- V is number of vertices

Code sample:

```
def bfs(graph, start_vertex, target_value):
  path = [start_vertex]
  vertex_and_path = [start_vertex, path]
  bfs_queue = [vertex_and_path]
  visited = set()
  while bfs_queue:
    current_vertex, path = bfs_queue.pop(0)
    visited.add(current_vertex)
    for neighbor in graph[current_vertex]:
      if neighbor not in visited:
        if neighbor is target_value:
          return path + [neighbor]
        else:
          bfs_queue.append([neighbor, path + [neighbor]])
```

Depth First Search :

Definition:
- An algorithm that searches a tree (or graph) by searching depth of the tree first, starting at the root.
  - It traverses left down a tree until it cannot go further.
  - Once it reaches the end of a branch it traverses back up trying the right child of nodes on that branch, and if possible left from the right children.
  - When finished examining a branch it moves to the node right of the root then tries to go left on all it's children until it reaches the bottom.
  - The right most node is evaluated last (the node that is right of all it's ancestors).

What you need to know:
- Optimal for searching a tree that is deeper than it is wide.
- Uses a stack to push nodes onto.
  - Because a stack is LIFO it does not need to keep track of the nodes pointers and is therefore less memory intensive than breadth first search.
  - Once it cannot go further left it begins evaluating the stack.

Time Complexity:
- Search: Depth First Search: O(|E| + |V|)
- E is number of edges
- V is number of vertices

Breadth First Search Vs. Depth First Search
- The simple answer to this question is that it depends on the size and shape of the tree.
  - For wide, shallow trees use Breadth First Search
  - For deep, narrow trees use Depth First Search

Nuances:
  - Because BFS uses queues to store information about the nodes and its children, it could use more memory than is available on your computer. (But you probably won't have to worry about this.)
  - If using a DFS on a tree that is very deep you might go unnecessarily deep in the search. See [xkcd](http://xkcd.com/761/) for more information.
  - Breadth First Search tends to be a looping algorithm.
  - Depth First Search tends to be a recursive algorithm.

Code sample:
```
def dfs(graph, current_vertex, target_value, visited = None):
  if visited is None:
    visited = []
  visited.append(current_vertex)
  if current_vertex is target_value:
    return visited

  for neighbor in graph[current_vertex]:
    if neighbor not in visited:
      path = dfs(graph, neighbor, target_value, visited)
      if path:
        return path
```

Dijkstras :

Definition:
- Dijkstra’s algorithm is an algorithm to find all of the shortest distances between a start vertex and the rest of the vertices in a graph.
- The algorithm works by keeping track of all the distances and updating the distances as it conducts a breadth-first search.
- Dijkstra’s algorithm runs in O((E+V)log V).

What you need to know:

Dijkstra’s Algorithm works as following:
1. Instantiate a dictionary that will eventually map vertices to their distance from the start vertex
2. Assign the start vertex a distance of 0 in a min heap
3. Assign every other vertex a distance of infinity in a min heap
4. Remove the vertex with the smallest distance from the min heap and set that to the current vertex
5. For the current vertex, consider all of its adjacent vertices and calculate the distance to them as (distance to the current vertex) + (edge weight of current vertex to adjacent vertex).
6. If this new distance is less than the current distance, replace the current distance.
7. Repeat 4 and 5 until the heap is empty
8. After the heap is empty, return the distances

Within the while loop, create a variable called mid and set it to the average of first and last.

Code sample:

```
def dijkstras(graph, start):
```

```
  distances = {}

  for vertex in graph:
    distances[vertex] = inf

  distances[start] = 0
  vertices_to_explore = [(0, start)]

  while vertices_to_explore:
    current_distance, current_vertex = heappop(vertices_to_explore)

    for neighbor, edge_weight in graph[current_vertex]:
      new_distance = current_distance + edge_weight

      if new_distance < distances[neighbor]:
        distances[neighbor] = new_distance
        heappush(vertices_to_explore, (new_distance, neighbor))

  return distances
```

** A* algorithm ** :

Definition:

- The A* algorithm is a greedy graph search algorithm that optimizes looking for a target vertex.
- A* is a modification of Dijkstra’s done by adding the estimated distance of each vertex to the goal vertex when searching.
- We can modify Dijkstra’s and turn it into A* by changing the following:
  - Adding a target for the search.
  - Gathering possible optimal paths and identify a single shortest path.
  - Implementing a heuristic that determines the likely distance remaining.
- The runtime of A* is $O(b^d)$ where b is the branching factor of the graph and d is the depth of the goal vertex from the start vertex.

Code sample:

```
# Manhattan Heuristic:
def heuristic(start, target):
  x_distance = abs(start.position[0] - target.position[0])
  y_distance = abs(start.position[1] - target.position[1])
  return x_distance + y_distance

# Euclidean Heuristic:
#def heuristic(start, target):
#  x_distance = abs(start.position[0] - target.position[0])
#  y_distance = abs(start.position[1] - target.position[1])
#  return sqrt(x_distance * x_distance + y_distance * y_distance)

def a_star(graph, start, target):
  print("Starting A* algorithm!")
  count = 0
  paths_and_distances = {}
  for vertex in graph:
    paths_and_distances[vertex] = [inf, [start.name]]

  paths_and_distances[start][0] = 0
```

```
  vertices_to_explore = [(0, start)]
  while vertices_to_explore and paths_and_distances[target][0] == inf:
    current_distance, current_vertex = heappop(vertices_to_explore)
    for neighbor, edge_weight in graph[current_vertex]:
      new_distance = current_distance + edge_weight + heuristic(neighbor,
target)
      new_path = paths_and_distances[current_vertex][1] + [neighbor.name]

      if new_distance < paths_and_distances[neighbor][0]:
        paths_and_distances[neighbor][0] = new_distance
        paths_and_distances[neighbor][1] = new_path
        heappush(vertices_to_explore, (new_distance, neighbor))
        count += 1
        print("\nAt " + vertices_to_explore[0][1].name)

  print("Found a path from {0} to {1} in {2} steps: ".format(start.name,
target.name, count), paths_and_distances[target][1])

  return paths_and_distances[target][1]
```

Practice:

Memoization – Fibonacci :

```
memo = {}

def fibonacci(num):
  answer = None
  # Write your code here
  if num in memo:
    answer = memo[num]
  elif num == 0 or num == 1:
    answer = num
  else:
    answer = fibonacci(num - 1) + fibonacci(num - 2)
    memo[num] = answer
  return answer
```

The Two Pointer Approach - Capturing Rainwater :

The previous solution had a quadratic runtime, but it’s possible to solve this problem in O(n) time by using two pointers. The pointers will start at each end of the array and move towards each other. The two-pointer approach is a common approach for problems that require working with arrays, as it allows you to go through the array in a single loop and without needing to create copy arrays.

The two-pointer approach is one that you can, and should, use in many interview questions. When you see a problem that requires you to iterate through an array (or string), take a moment and think if it would be possible to iterate through the array in sections at the same time instead of in separate loops. Common problems that can be solved using the two-pointer technique are the two sum problem (finding two numbers in an array that sum to a specified number) and reversing the characters in a string.

```
def efficient_solution(heights):
  total_water = 0
  left_pointer = 0
  right_pointer = len(heights) - 1
  left_bound = 0
  right_bound = 0

  # Write your code here
  while left_pointer < right_pointer:
    if heights[left_pointer] <= heights[right_pointer]:
      left_bound = max(heights[left_pointer], left_bound)
      total_water += left_bound - heights[left_pointer]
      left_pointer += 1
    else:
      right_bound = max(heights[right_pointer], right_bound)
      total_water += right_bound - heights[right_pointer]
      right_pointer -= 1

  return total_water
```

The Knapsack Problem:

While this recursive solution works, it has a big O runtime of O(2^n). In the worst case, each step would require us to evaluate two subproblems, sometimes repeatedly, as there's overlap between subproblems. We can drastically improve on this runtime by using dynamic programming.

```
def recursive_knapsack(weight_cap, weights, values, i):
  if weight_cap == 0 or i == 0:
    return 0
  elif weights[i - 1] > weight_cap:
    return recursive_knapsack(weight_cap, weights, values, i - 1)
  else:
    include_item = values[i - 1] + recursive_knapsack(weight_cap - 
weights[i - 1], weights, values, i - 1);

    exclude_item = recursive_knapsack(weight_cap, weights, values, i - 1);

    return max(include_item, exclude_item)
```

This version has a big O runtime of O(n * weight_cap) compared to the recursive implementation's runtime of O(2^n). While this optimized runtime might seem worse using small cases, it is much more efficient as the parameters grow.

```
def dynamic_knapsack(weight_cap, weights, values):
  rows = len(weights) + 1
  cols = weight_cap + 1
  # Set up 2D array
  matrix = [ [] for x in range(rows) ]

  # Iterate through every row
  for index in range(rows):
    # Initialize columns for this row
    matrix[index] = [ -1 for y in range(cols) ]

    # Iterate through every column
    for weight in range(cols):
      # Write your code here
      if index == 0 or weight == 0:
        matrix[index][weight] = 0
      # If weight at previous row is less than or equal to current weight
      elif weights[index - 1] <= weight:
        # Calculate item to include
        include_item = values[index - 1] + matrix[index - 1][weight - 
weights[index - 1]]

        # Calculate item to exclude
        exclude_item = matrix[index - 1][weight]

        # Calculate the value of current cell
        matrix[index][weight] = max(include_item, exclude_item)
      else:
        # Calculate the value of current cell
        matrix[index][weight] = matrix[index - 1][weight]
  # Return the value of the bottom right of matrix
  return matrix[rows-1][weight_cap]
```

Sieve of Eratosthenes:

Basic version :
```
def sieve_of_eratosthenes(limit):
  true_indices = []

  # handle edge cases
  if (limit <= 1):
    return true_indices

  # create the output list
  output = [True] * (limit+1)

  # mark 0 and 1 as non-prime
  output[0] = False
  output[1] = False

  # iterate up to the square root of the limit
  for i in range(2, limit+1):
    if (output[i] == True):
      j = i*2
      # mark all multiples of i as non-prime
      while j <= limit:
        output[j] = False
        j += i

  # remove non-prime numbers
  output_with_indices = list(enumerate(output))
  true_indices = [index for (index,value) in output_with_indices if value
== True]
  return true_indices
```

Optimized version:

The complexity of the Sieve of Eratosthenes with optimizations is O(n log (log n)). There are two operations to take into account: the creation of the array and the incrementing and multiple-marking loops. Creation happens in O(n) time, since it creates an element for each number from 2 to n.Multiple marking happens in O(n log (log n)) time.

```
# import math library
import math

def sieve_of_eratosthenes (limit):
  # handle edge cases
  if (limit <= 1):
    return []

  # create the output list
  output = [True] * (limit+1)

  # mark 0 and 1 as non-prime
  output[0] = False
  output[1] = False

  # iterate up to the square root of the limit
  for i in range(2, math.floor(math.sqrt(limit))):
    if (output[i] == True):
      j = i ** 2    # initialize j to square of i
```

```
    # mark all multiples of i as non-prime
    while j <= limit:
      output[j] = False
      j += i

# remove non-prime numbers
output_with_indices = list(enumerate(output))
trues = [index for (index,value) in output_with_indices if value == True]
return trues
```

TECHNICAL INTERVIEW PROBLEMS IN PYTHON: LISTS

Palindrome:

```
# what if we wanted to use recursion?
def palindrome_1(str):
  if len(str) <= 1:
    return True
  if str[0] != str[-1]:
    return False
  return palindrome_1(str[1:-1])

# what if we didn't care about case?
def palindrome_2(str):
  lower = str.lower()
  for i in range(len(str) // 2):
    if lower[i] != lower[-i - 1]:
      return False
  return True

# what if we wanted to ignore punctuation?
def palindrome_3(str):
  punctuation = [',', '!', '?', '.']
  no_punc_str = str[:]
  for punc in punctuation:
    no_punc_str = no_punc_str.replace(punc, '')
  for i in range(len(no_punc_str) // 2):
    if no_punc_str[i] != no_punc_str[-i - 1]:
      return False
  return True

# what if we wanted to ignore space?
def palindrome_4(str):
  no_space_str = str.replace(' ', '')
  for i in range(len(no_space_str) // 2):
    if no_space_str[i] != no_space_str[-i - 1]:
      return False
  return True
```

rotate list

```
# no time/space requirements
# return "rotated" version of input list

def rotate(l, k):
  for i in range(k):
    l.insert(0, l.pop())
  return l

def rotate_alternative(lst, degree):
  rotation = degree % len(lst)
  return lst[-rotation:] + lst[:-rotation]
```

List Rotation: Indices

```
def rev(lst, low, high):
  while low < high:
    lst[low], lst[high] = lst[high], lst[low]
    high -= 1
    low += 1
  return lst

def rotate(my_list, num_rotations):
```

```
  # first half
  rev(my_list, 0, num_rotations - 1)

  # second half
  rev(my_list, num_rotations, len(my_list) - 1)

  # whole list
  rev(my_list, 0, len(my_list) - 1)
  return my_list
```

Rotation Point: Linear Search

```
def rotation_point(rotated_list):
  rotation_idx = 0
  for i in range(len(rotated_list)):
    if rotated_list[i] < rotated_list[rotation_idx]:
      rotation_idx = i
  return rotation_idx
```

Rotation Point: Binary Search

```
def rotation_point(rotated_list):
  low = 0
  high = len(rotated_list) - 1
  while low <= high:
    mid = (low + high) // 2
    mid_next = (mid + 1) % len(rotated_list)
    mid_previous = (mid - 1) % len(rotated_list)

    if (rotated_list[mid] < rotated_list[mid_previous]) and
(rotated_list[mid] < rotated_list[mid_next]):
      return mid
    elif rotated_list[mid] < rotated_list[high]:
      high = mid - 1
    else:
      low = mid + 1
```

Remove Duplicates: Naive

```
def remove_duplicates(dupe_list):
  unique_values = []
  for el in dupe_list:
    if el not in unique_values:
      unique_values.append(el)
  return unique_values
```

Remove Duplicates: Optimized

```
def move_duplicates(dupe_list):
  unique_idx = 0
  for i in range(len(dupe_list) - 1):
    if dupe_list[i] != dupe_list[i + 1]:
      dupe_list[i], dupe_list[unique_idx] = dupe_list[unique_idx],
dupe_list[i]
      unique_idx += 1
  dupe_list[unique_idx], dupe_list[len(dupe_list) - 1] =
dupe_list[len(dupe_list) - 1], dupe_list[unique_idx]
  return unique_idx
```

Max list sub-sum: Naive

```
def maximum_sub_sum(my_list):
  max_sum = my_list[0]
  for i in range(len(my_list)):
    for j in range(i, len(my_list)):
      sub_sum = sum(my_list[i:j + 1])
      if sub_sum > max_sum:
        max_sum = sub_sum
  return max_sum
```

Max List Sub-Sum: Optimized

```
def maximum_sub_sum(my_list):
  if max(my_list) < 0:
    return max(my_list)

  max_sum = 0
  max_sum_tracker = 0
  for i in range(len(my_list)):
    max_sum_tracker += my_list[i]
    if max_sum_tracker < 0:
      max_sum_tracker = 0
    if max_sum_tracker > max_sum:
      max_sum = max_sum_tracker

  return max_sum
```

Pair Sum: Naive

```
def pair_sum(nums, target):
  for i in range(len(nums)):
    for j in range(i, len(nums)):
      if nums[i] + nums[j] == target:
        return [i, j]
  return None
```

Pair Sum: Optimized

```
def pair_sum(nums, target):
  complements = {}
  indices = {}
  for i in range(len(nums)):
    x = complements.get(nums[i], None)
    if x is not None:
      return [indices[x], i]
    complements[target - nums[i]] = nums[i]
    indices[nums[i]] = i
```

TECHNICAL INTERVIEW PROBLEMS IN PYTHON: LINKED LISTS

Insert at Point

```
def insert(self, node_value, location):
    if not location:
      new_head = Node(node_value)
      new_head.next = self.head
      self.head = new_head
      return self

    prev = self.head
    node = Node(node_value)
    current_node = self.head.next

    while location > 1:
      prev = current_node
      current_node = current_node.next
      location -= 1

    prev.next = node
    node.next = current_node

    return self
```

Nth From Last

```
  def n_from_last(self, n):
    nodes_remaining = self.size() - 1 - n
    result = self.head

    while nodes_remaining:
      result = result.next
      nodes_remaining -= 1

    return result
```

Remove Duplicates

```
  def remove_duplicates(self):
    current_node = self.head

    while current_node:
      while current_node.next and current_node.next.val ==
current_node.val:
        current_node.next = current_node.next.next
      current_node = current_node.next
    return self
```

Merge Sorted Linked Lists

```
def merge(linked_list_a, linked_list_b):

  current_node_a = linked_list_a.head
  current_node_b = linked_list_b.head

  if current_node_a.val < current_node_b.val:
    start_node = current_node_a
    current_node_a = current_node_a.next
  else:
    start_node = current_node_b
```

```
    current_node_b = current_node_b.next

  head = start_node

  while current_node_a or current_node_b:
    if not current_node_a:
      start_node.next = current_node_b
      current_node_b = current_node_b.next
    elif not current_node_b:
      start_node.next = current_node_a
      current_node_a = current_node_a.next
    elif current_node_a.val < current_node_b.val:
      start_node.next = current_node_a
      current_node_a = current_node_a.next
    else:
      start_node.next = current_node_b
      current_node_b = current_node_b.next
    start_node = start_node.next

  return LinkedList(head)
```

Find Merge Point

```
def merge_point(linked_list_a, linked_list_b):
  size_of_a = linked_list_a.size()
  size_of_b = linked_list_b.size()

  diff = abs(size_of_a - size_of_b)

  if size_of_a > size_of_b:
    bigger = linked_list_a.head
    smaller = linked_list_b.head
  else:
    bigger = linked_list_b.head
    smaller = linked_list_a.head

  for i in range(diff):
    bigger = bigger.next

  while bigger and smaller:
    if bigger == smaller:
      return bigger
    bigger = bigger.next
    smaller = smaller.next

  return None
```

Reverse a Linked List

```
def reverse(linked_list):
    prev = None
    current_node = linked_list.head
    while current_node:
      tmp = current_node.next
      current_node.next = prev
      prev = current_node
      current_node = tmp
    return LinkedList(prev)
```

Detect Cycle in a Linked List

```
def has_cycle(linked_list):
  slow, fast = linked_list.head, linked_list.head
  while slow and fast:
    slow = slow.next
    fast = fast.next
    if fast:
      fast = fast.next
    else:
      return False
    if fast == slow:
      return True
  return False
```

<u>### Add Two Numbers :</u>

```
def add_two(linked_list_a, linked_list_b):

  result = LinkedList()
  carry = 0

  a_node = linked_list_a.head
  b_node = linked_list_b.head

  while a_node or b_node:

    if b_node:
      b_val = b_node.val
      b_node = b_node.next
    else:
      b_val = 0

    if a_node:
      a_val = a_node.val
      a_node = a_node.next
    else:
      a_val = 0

    to_sum = a_val + b_val + carry

    if to_sum > 9:
      carry = 1
      to_sum %= 10
    else:
      carry = 0


    if not result.head:
      result.head = Node(to_sum)
      tmp = result.head
    else:
      tmp.next = Node(to_sum)
      tmp = tmp.next

  if carry:
    tmp.next = Node(carry)

  return result
```