Coding

CHAPTER 9

Every Superman has his kryptonite, but as a Data Scientist, coding can't be yours. Between data munging, pulling in data from APIs, and setting up data processing pipelines, writing code is a near-universal part of a Data Scientist's job. This is especially true at smaller companies, where data scientists tend to wear multiple hats and are responsible for productionizing their analyses and models. Even if you are the rare Data Scientist that never has to write production code, consider the collaborative nature of the field — having strong computer science fundamentals will give you a leg up when working with software and data engineers.

To test your programming foundation, Data Science interviews often take you on a stroll down memory lane back to your Data Structures and Algorithms class ***(you did take one, right?)****. These coding questions test your ability to manipulate data structures like lists, trees, and graphs, along with your ability to implement algorithmic concepts such as recursion and dynamic programming. You're also expected to assess your solution's runtime and space efficiency using Big O notation.*

Approaching Coding Questions

Coding interviews typically last 30 to 45 minutes and come in a variety of formats. Early in the interview process, coding interviews are often conducted via remote coding assessment tools like HackerRank, Codility, or CoderPad. During final-round onsite interviews, it's typical to write code on a whiteboard. Regardless of the format, the approach outlined below to solve coding interview problems applies.

After receiving the problem: Don't jump right into coding. It's crucial first to make sure you are solving the correct problem. Due to language barriers, misplaced assumptions, and subtle nuances that are easy to miss, misunderstanding the problem is a frequent occurrence. To prevent this, make sure to repeat the question back to the interviewer so that the two of you are on the same page. Clarify any assumptions made, like the input format and range, and be sure to ask if the input can be assumed to be non-null or well formed. As a final test to see if you've understood the problem, work through an example input and see if you get the expected output. Only after you've done these steps are you ready to begin solving the problem.

When brainstorming a solution: First, explain at a high level how you could tackle the question. This usually means discussing the brute-force solution. Then, try to gain an intuition for why this brute-force solution might be inefficient, and how you could improve upon it. If you're able to land on a more optimal approach, articulate how and why this new solution is better than the first brute-force solution provided. Only after you've settled on a solution is it time to begin coding.

When coding the solution: Explain what you are coding. Don't just sit there typing away, leaving your interviewer in the dark. Because coding interviews often let you pick the language you write code in, you're expected to be proficient in the programming language you chose. As such, avoid pseudocode in favor of proper compilable code. While there is time pressure, don't take many shortcuts when coding. Use clear variable names and follow good code organization principles. Write well-styled code — for example, following PEP 8 guidelines when coding in Python. While you are allowed to cut some corners, like assuming a helper method exists, be explicit about it and offer to fix this later on.

After you're done coding: Make sure there are no mistakes or edge cases you didn't handle. Then write and execute test cases to prove you solved the problem.

At this point, the interviewer should dictate which direction the interview heads. They may ask about the time and space complexity of your code. Sometimes they may ask you to refactor and clean the code, especially if you cut some corners while coding the solution. They may also extend the problem, often with a new constraint. For example, they may ask you not to use recursion and instead tell you to solve the problem recursively. Or, they might ask you to not use surplus memory and instead solve the problem in place. Sometimes, they may pose a tougher variant of the problem as a follow-up, which might require starting the problem-solving process all over again.

Space & Time Complexity

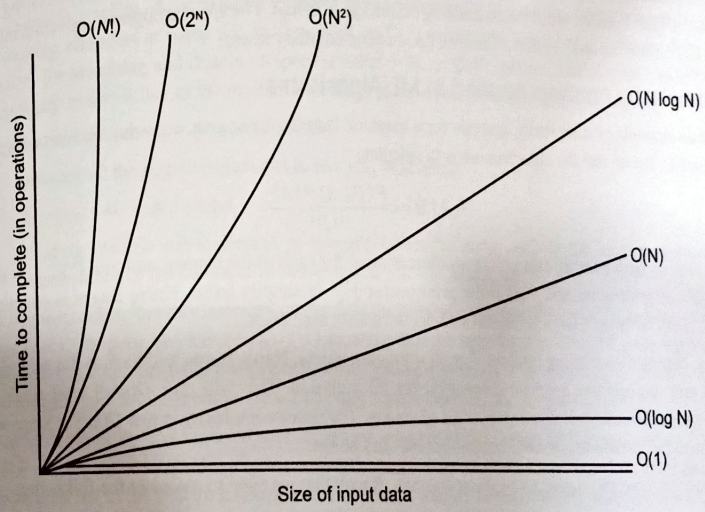
Determining the runtime and space usage (how much memory is utilized) of an algorithm is essential for coding interviews and real-world data science. Because compute and storage resources can be bottlenecks to machine learning model training and deployment, analyzing an algorithm's performance can affect what techniques you choose to implement. Consider OpenAI's GPT-3 language model, which contains over 175 billion parameters and took $12 million in compute resources to train. Much of the work bringing GPT-3 to the world involved optimizing resource usage to efficiently train such a large model.

Computer scientists analyze and classify the behavior of an algorithm's time and space usage via asymptotic complexity analysis. This technique considers how an algorithm performs when the input size goes toward infinity and characterizes the behavior of the runtime and space used as a function of n. In academic settings, we establish tight bounds on performance in terms of n using Big θ (big theta) notation. However, in industry, the technical definitions have been muddled, and we tend to denote these tight bounds on performance using Big O notation.

In the context of companies asking interview questions, we care about not just establishing tight bounds on performance but thinking about the worst-case scenario for this performance. As such, Big O notation often describes the "worst-case upper bound," or the longest an algorithm would run or the maximal amount of space it would need in the worst case.

For instance, consider an array of size N. Here are the following classes of runtime complexities, from fastest to slowest, using Big O notation:

- $O(1)$: Constant time. Example: getting a value at a particular index from an array
- $O(\log N)$: Logarithmic time. Example: binary search on a sorted array
- $O(N)$: Linear time. Example: using a for-loop to traverse through an array
- $O(N \log N)$: Log-linear time. Example: running mergesort on an array
- $O(N^2)$: Quadratic time. Example: iterating over every pair of elements in an array using a double for-loop
- $O(2^N)$: Exponential time. Example: recursively generating all binary numbers that are N digits long
- $O(N!)$: Factorial time. Example: generating all permutations of an array



The same Big-O runtime analysis concepts apply analogously to space complexity. For example, if we need to store a copy of an input array with N elements, that would be an additional $O(N)$ space. If we wanted to store an adjacency matrix among N nodes, we would need $O(N^2)$ space to keep the N-by-N sized matrix.

For a basic example for both runtime and space complexity analysis, we can look at binary search, where we are searching for a particular value within a sorted array. The code that implements this algorithm is below (with an extra set of conditions that returns the closest if the exact value is not found):

```
def binary_search(a, k):
  lo, hi = 0, len(a) - 1
  best = lo
  while lo <= hi:
    mid = lo + (hi - lo) // 2
    if a[mid] < k:
      lo = mid + 1
    elif a[mid] > k:
      hi = mid - 1
    else:
      best = mid
      break
    if abs(a[mid] - k) < abs(a[best] - k):
      best = mid
    return best
```

If we start the binary search with an input of N elements, then at the next iteration, we would only need to search through $N/2$ elements, and so on. The runtime complexity for binary search is $O(\log N)$ since at each iteration we cut the remaining search space in half. The space complexity would simply be $O(N)$ since the array is size N, and we do not need auxiliary space.

Complexity Analysis Applied to ML Algorithms

For an example of complexity analysis for a machine learning technique, consider the Naive Bayes classifier. Recall that the algorithm aims to calculate

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

for each of the n training data points (of dimension d) for each of the classes, and that $P(B \mid A)$ is the likelihood probability, and $P(A)$ is the prior probability. In simpler terms, Naive Bayes is counting how many times each of the d features co-occurs within each class.

Now, consider the training runtime. For all n training points, Naive Bayes will look at the posterior and prior probabilities over all d features, for all k classes. This will take $O(nkd)$ total runtime since the operations boil down to a series of counts. The space complexity is just $O(kd)$ to store the probabilities needed to compute results for new data points.

As another example, consider logistic regression. Recall that we need to calculate the following:

$$S(x) = \frac{1}{1 + e^{-X\beta}}$$

for any given x. There are n training data points (each with dimension d); hence, β is a d-by-1 vector of weights. Recall that the goal of logistic regression is to find the optimal decision boundary to split the data into two classes. This involves multiplying each of the n training points with β, which is d-by-1 vector, so the training runtime complexity is $O(d)$. The space complexity is just $O(d)$ to store the weights (β) to classify new data points.

Data Structures

Below is a brief overview of the most common data structures used for coding interviews. The best way to become familiar with each data structure is by implementing a basic version of it in your favorite language. Knowing the Big-O for common operations, like inserting an element or finding an element within the structure, is also essential. The table below can be used for reference:

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	0(1)	0(n)	0(n)	0(n)	0(1)	0(n)	0(n)	0(n)	0(n)
Stack	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Queue	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Linked List	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Hash Map	N/A	0(1)	0(1)	0(1)	N/A	0(n)	0(n)	0(n)	0(n)
Binary Search Tree	0(log (n))	0(log (n))	0(log (n))	0(log (n))	0(n)	0(n)	0(n)	0(n)	0(n)

Arrays

An array is a series of consecutive elements stored sequentially in memory. Arrays are optimal for accessing elements at particular indices, with an O(1) access and index time. However, they are slower for searching and deleting a specific value, with an O(N) runtime, unless sorted. An array's simplicity makes it one of the most commonly used data structures during coding interviews.

Common array interview questions include:

- Moving all the negative elements to one side of an array
- Merging two sorted arrays
- Finding specific sub-sequences of integers within the array, such as the longest consecutive subsequence or the consecutive subsequence with the largest sum

A frequent pattern for array interview questions is the existence of a straightforward brute-force solution that uses O(n) space, and a more clever solution that uses the array itself to lower the space complexity down to O(1). Another pattern we've seen when dealing with arrays is the prevalence of off-by-1 errors — it's easy to crash the program by accidentally reading past the last element of an array.

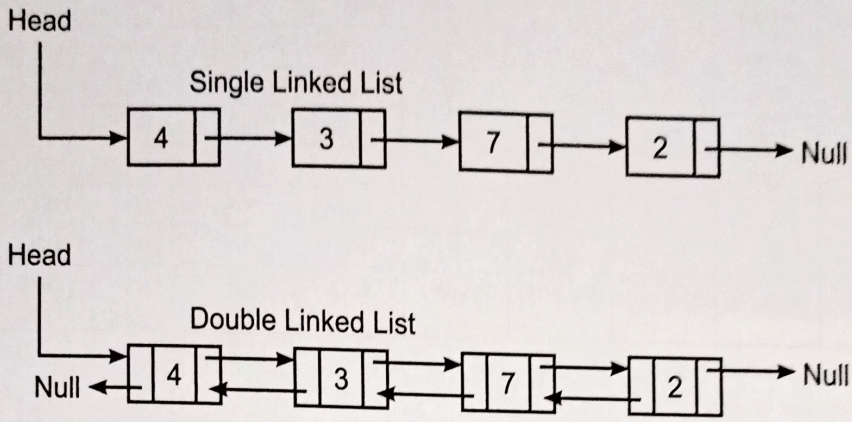
For jobs where Python knowledge is important, interviews may cover list comprehensions, due to their expressiveness and ubiquity in codebases. As an example, below, we use a list comprehension to create a list of the first 10 positive even numbers. Then, we use another list comprehension to find the cumulative sum of the first list:

```
a = [x*2 for x in range(1, 11)] # list creation
c = [sum(a[:x]) for x in range(len(a)+1)] # cumulative sum
```

Arrays are also at the core of linear algebra since vectors are represented as 1-D arrays, and matrices are represented by 2-D arrays. For example, in machine learning, the feature matrix X can be represented by a 2-D array, with one dimension as the number of data points (n) and the other as the number of features (d).

Linked Lists

A linked list is composed of nodes with data that have pointers to other nodes. The first node is called the head, and the last node is called the tail. Linked lists can be circular, where the tail points to the head. They can also be doubly linked, where each node has a reference to both the previous and next nodes. Linked lists are optimal for insertion and deletion, with O(1) insertion time at the head or tail, but are worse for indexing and searching, with a runtime complexity of O(N) for indexing and O(N) for search.



Common linked list questions include:

- Reversing a linked list
- Detecting a cycle in a linked list
- Removing duplicates from a sorted linked list
- Checking if a linked list represents a palindrome

As an example, below we reverse a linked list. Said another way, given the input linked list 4 → 1 → 3 → 2, we want to write a function which returns 2 → 3 → 1 → 4. To implement this, we first start with a basic node class:

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

Then we create the LinkedList class, along with the method to reverse its elements. The reverse function iterates through each node of the linked list. At each step, it does a series of swaps between the pointers of the current node and its neighbors.

```
class LinkedList:
  def __init__(self):
    self.head = None

  def reverse(self):
    prev = None
```

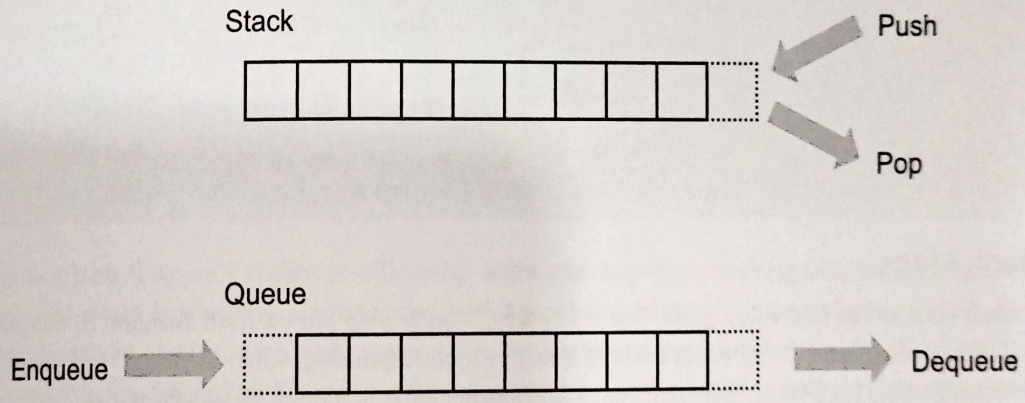
```
    curr = self.head
    while curr:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    self.head = prev
```

Like array interview questions, linked list problems often have an obvious brute-force solution that uses O(n) space, but then also a more clever solution that utilizes the existing list nodes to reduce the memory usage to O(1). Another commonality between array and linked list interview solutions is the prevalence of off-by-one errors. In the linked list case, it's easy to mishandle pointers for the head or tail nodes.

Stacks & Queues

A stack is a data structure that allows adding and removing elements in a last-in, first-out (LIFO) order. This means the element that is added last is the first element to be removed. Another name for adding and removing elements from a stack is pushing and popping. Stacks are often implemented using an array or linked list.

A queue is a data structure that allows adding and removing elements in a first-in, first-out (FIFO) order. Queues are also typically implemented using an array or linked list.



The main difference between a stack and a queue is the removal order: in the stack, there is a LIFO order, whereas in a queue it's a FIFO order. Stacks are generally used in recursive operations, whereas queues are used in more iterative processes.

Common stacks and queues interview questions include:

- Writing a parser to evaluate regular expressions (regex)
- Evaluating a math formula using order of operations rules
- Running a breadth-first or depth-first search through a graph

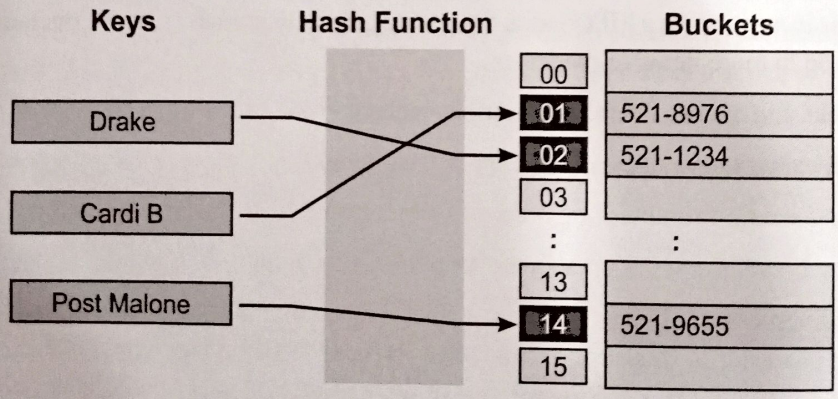
An example interview problem that uses a stack is determining whether a string has balanced parentheses. Balanced, in this case, means every type of left-side parentheses is accompanied by valid right-side parentheses. For instance, the string "({}(()){})" is correctly balanced, whereas the string "{}()" is not balanced, due to the last character, ')'. The algorithm steps are as follows:

1) Starting parentheses (left-sided ones) are pushed onto the stack.
2) Ending parentheses (right-sided ones) are verified to see if they are of the same type as the most recently seen left-side parentheses on the stack.
3) If the parentheses are of the same type, pop from the stack. If they don't match, return false since the parentheses are mismatched.
4) Continue parsing the input until it's completely processed, and the stack is empty (every pair of parentheses was correctly accounted for), in which case, return true. Or, if the stack is not empty, in which case, return false.

```
def check_balance(s):
  left_side = ["(", "{", "["] # left parentheses
  right_side = [")", "}", "]"] # right parentheses
  stack = [] # stack
  for i in s:
    if i in left_side:
      stack.append(i) # push onto stack
    elif i in right_side:
      pos = right_side.index(i) # get right char
      # check match
      if len(stack) == 0 or (left_side[pos] != stack[len(stack)-1]):
        return False
      else:
        stack.pop() # remove and continue
    if len(stack) == 0: # balanced
      return True
    else:
      return False
```

Hash Maps

A hash map stores key-value pairs. For every key, a hash map uses a hash function to compute an index, which locates the bucket where that key's corresponding value is stored. In Python, a dictionary offers support for key-value pairs and has the same functionality as a hash map.



While a hash function aims to map each key to a unique index, there will sometimes be "collisions" where different keys have the same index. In general, when you use a good hash function, expect the elements to be distributed evenly throughout the hash map. Hence, lookups, insertions, or deletions for a key take constant time.

Due to their optimal runtime properties, hash maps make a frequent appearance in coding interview questions.

Common hash map questions center around:

- Finding the unions or intersection of two lists
- Finding the frequency of each word in a piece of text
- Finding four elements a, b, c and d in a list such that $a + b = c + d$

An example interview question that uses a hash map is determining whether an array contains two elements that sum up to some value. For instance, say we have a list [3, 1, 4, 2, 6, 9] and k. In this case, we return true since 2 and 9 sum up to 11.

The brute-force method to solving this problem is to use a double for-loop and sum up every pair of numbers in the array, which provides an O(N^2) solution. But, by using a hash map, we only have to iterate through the array with a single for-loop. For each element in loop, we'd check whether the complement of the number (target - that number) exists in the hash map, achieving an O(N) solution:

```
def check_sum(a, target):
  d = {} # create dictionary
  for i in a:
    if (target - i) in d: # check hashmap
      return True
    else:
      d[i] = i # add to hashmap
  return False
```

Due to a hash function's ability to efficiently index and map data, hashing functions are used in many real-world applications (in particular, with regards to information retrieval and storage). For example, say we need to spread data across many databases to allow for data to be stored and queried efficiently while distributed. Sharding, covered in depth in the databases chapter, is one way to split the data. Sharding is commonly implemented by taking the given input data, and then applying a hash function to determine which specific database shard the data should reside on.

Trees

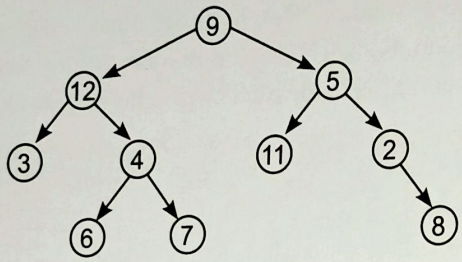
A tree is a basic data structure with a root node and subtrees of children nodes. The most basic type of tree is a binary tree, where each node has at most two children nodes. Binary trees can be implemented with a left and right child node, like below:

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

There are various types of traversals and basic operations that can be performed on trees. For example, in an in-order traversal, we first process the left subtree of a node, then process the current node, and, finally, process the right subtree:

```
def inorder(node):
    if node is None:
        return []
    else:
        return inorder(node.left) + [node.val] + inorder(node.right)
```

The two other closely related traversals are post-order traversal and pre-order traversal. A simple way to remember how these three algorithms work is by remembering that the "post/pre/in" refers to the placement of the processing of the root value. Hence, a post-order traversal processes the left child node first, then the right child node and, in the end, the root node. A pre-order traversal processes the root node first, then the left child node, and then, the right child node.



Level-order Tree Traversal:
9, 12, 5, 3, 4, 11, 2, 6, 7, 8

In-order Tree Traversal:
3, 12, 6, 4, 7, 9, 11, 5, 2, 8

Pre-order Tree Traversal:
9, 12, 3, 4, 6, 7, 5, 11, 2, 8

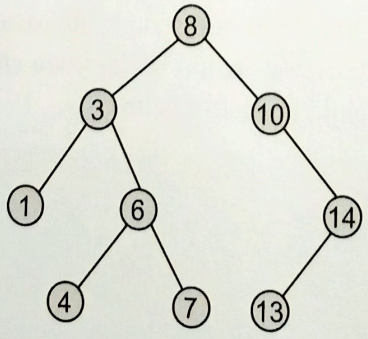
Post-order Tree Traversal:
3, 6,7, 4, 12, 11, 8, 2, 5, 9

For searching, insertion, and deletion, the worst-case runtime for a binary tree is O(N), where N is the number of nodes in the tree.

Common tree questions involve writing functions to get various properties of a tree, like the depth of a tree or the number of leaves in a tree. Oftentimes, tree questions boil down to traversing through the tree and recursively passing some data in a top-down or a bottom-up manner. Coding interview problems also often focus on two specific types of trees: Binary Search Trees and Heaps.

Binary Search Trees

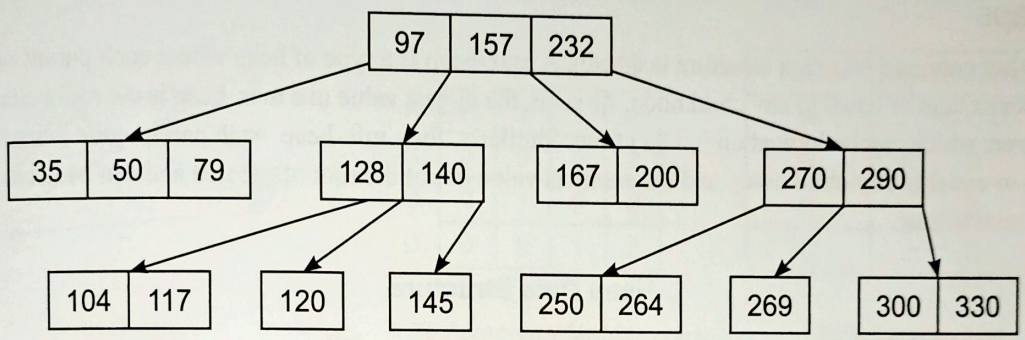
A binary search tree (BST) is composed of a series of nodes, where any node in a left subtree is smaller than or equal to the root, and any node in the right subtree is larger than or equal to the root. When BSTs are height balanced so no one leaf is much deeper than another leaf from the root, searching for elements becomes efficient. To demonstrate, consider searching for the value 9 in the balanced BST below:



Example of a Binary Search Tree

To find 9, we first examine the root value, 8. Since 9 is greater than 8, the node containing 9, if it exists, would have to be on the right side of the tree. Thus, we've cut the search space in half. Next, we compare against the node 10. Since 9 is less than 10, the node, should it exist, has to be on the left of 10. Again, we've cut the search space in half. In conclusion, since 10 doesn't have a left child, we know 9 doesn't occur in the tree. By cutting the search space in half at each iteration, BSTs support search, insertion, and deletion in O(log *N*) runtime.

Because of their lookup efficiency, BSTs show up frequently not just in coding interviews but in real-life applications. For instance, B-trees, which are used universally in database indexing, are a generalized version of BSTs. That is, they allow for more than 2 nodes (up to *M* children), but offer a searching and insertion process similar to that of BST. These properties allow B-trees to have O(log *N*) lookup and insertion runtimes similar to that of BSTs, where *N* is the total number of nodes in the B-tree. Because of the logarithmic growth of the tree depth, database indexes with millions of records often only have a B-tree depth of four or five layers.



Example of a B-Tree

Common BST questions cover:

- Testing if a binary tree has the BST property
- Finding the *k*-th largest element in a BST
- Finding the lowest common ancestor between two nodes (the closest common node to two input nodes such that both input nodes are descendants of that node)

An example implementation of a BST using the TreeNode class, with an insert function, is as follows:

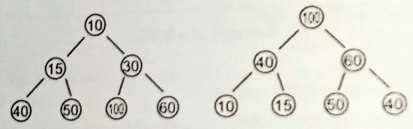
```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

class BST:
    def __init__(self, val):
        self.root = TreeNode(val)

    def insert(self, node, val):
        if node is not None:
            if val < node.val:
                if node.left is None:
```

```
            node.left = TreeNode(val)
        else:
            self.insert(node.left, val)
    else:
        if node.right is None:
            node.right = TreeNode(val)
        else:
            self.insert(node.right, val)
else:
    self.root = TreeNode(val)
return
```

Heaps

Another common tree data structure is a heap. A max-heap is a type of heap where each parent node is greater than or equal to any child node. As such, the largest value in a max-heap is the root value of the tree, which can be looked up in O(1) time. Similarly, for a min-heap, each parent node is smaller than or equal to any child node, and the smallest value lies at the root of the tree and can be accessed in constant time.

Heap Data Structure



Minimum Heap Maximum Heap

To maintain the heap property, there is a sequence of operations known as "heapify," whereby values are "bubbled up/down" within the tree based on what value is being inserted or deleted. For example, say we are inserting a new value into a min-heap. This value starts at the bottom of the heap and then is swapped with its parent node ("bubbled up") until it is no longer smaller than its parent (in the case of a min-heap). The runtime of this heapify operation is the height of the tree, O(log N).

In terms of runtime, inserting or deleting is O(log N), because the heapify operation runs to maintain the heap property. The search runtime is O(N) since every node may need to be checked in the worst-case scenario. As mentioned earlier, heaps are optimal for accessing the min or max value because they are at the root, i.e., O(1) lookup time. Thus, consider using heaps when you care mostly about finding the min or max value and don't need fast lookups or deletes of arbitrary elements.

Commonly asked heap interview questions include:

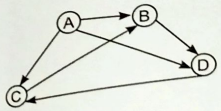
- finding the K largest or smallest elements within an array
- finding the current median value in a stream of numbers
- sorting an almost-sorted array (where elements are just a few places off from their correct spot)

To demonstrate the use of heaps, below we find the k-largest elements in a list, using the heapq package in Python:


```
import heapq
k = 5
a = [13, 5, 2, 6, 10, 9, 7, 4, 3]
heapq.heapify(a) # creates heap
heapq.nlargest(k, a) # finds k-largest
```

Graphs

Graphs are composed of nodes (vertices) and a set of edges that connect the nodes. Edges are often represented in two ways: an adjacency matrix or an adjacency list. In the case of an adjacency matrix, there is a n-by-n matrix for n-nodes where the entries are either Boolean values (denoting an edge is present or not between two nodes) or a weight. In contrast, an adjacency list stores a list of neighbors for each node.



To Graph

	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	1	0	0
D	0	0	1	0

To Adjacency Matrix

Node	Adjacent Node(s)
A	B C D
B	D
C	B
D	C

To Adjacency List

The lookup time to check whether two nodes are neighbors for an adjacency matrix is O(1). For an adjacency list, it could be O(N) in the worst case (if a node has edges to every other node). What an adjacency list lacks in time efficiency compared to an adjacency matrix, it makes up for in space efficiency. For a large graph with n nodes that are sparse (the N nodes don't have many connections to each other), an adjacency list offers a compact way to represent the edges, versus an adjacency matrix which requires you to store an N^2 sized matrix in memory. For instance, consider the Facebook friendship graph, which has 2 billion users (nodes), but a maximum of only 5,000 connections per node (the 5,000 friend limit). A full adjacency matrix would need to be 2 billion rows by 2 billion columns, whereas an adjacency list would require considerably less memory.

An example implementation of a graph is below. The Vertex class uses an adjacency list to keep track of its neighbors with weights:

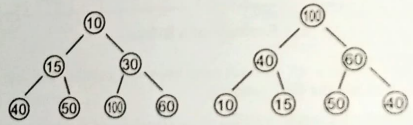
```
class Vertex:
    def __init__(self, val):
        self.val = val
        self.neighbors = {}

    def add_to_neighbors(self, neighbor, w): # add to neighbor dict with weight
        self.neighbors[neighbor] = w

    def get_neighbors(self):
        return self.neighbors.keys()
```

```
            node.left = TreeNode(val)
        else:
            self.insert(node.left, val)
    else:
        if node.right is None:
            node.right = TreeNode(val)
        else:
            self.insert(node.right, val)
else:
    self.root = TreeNode(val)
return
```

Heaps

Another common tree data structure is a heap. A max-heap is a type of heap where each parent node is greater than or equal to any child node. As such, the largest value in a max-heap is the root value of the tree, which can be looked up in O(1) time. Similarly, for a min-heap, each parent node is smaller than or equal to any child node, and the smallest value lies at the root of the tree and can be accessed in constant time.

Heap Data Structure



Minimum Heap Maximum Heap

To maintain the heap property, there is a sequence of operations known as "heapify," whereby values are "bubbled up/down" within the tree based on what value is being inserted or deleted. For example, say we are inserting a new value into a min-heap. This value starts at the bottom of the heap and then is swapped with its parent node ("bubbled up") until it is no longer smaller than its parent (in the case of a min-heap). The runtime of this heapify operation is the height of the tree, O(log *N*).

In terms of runtime, inserting or deleting is O(log *N*), because the heapify operation runs to maintain the heap property. The search runtime is O(*N*) since every node may need to be checked in the worst-case scenario. As mentioned earlier, heaps are optimal for accessing the min or max value because they are at the root, i.e., O(1) lookup time. Thus, consider using heaps when you care mostly about finding the min or max value and don't need fast lookups or deletes of arbitrary elements.

Commonly asked heap interview questions include:

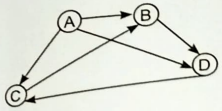
- finding the *K* largest or smallest elements within an array
- finding the current median value in a stream of numbers
- sorting an almost-sorted array (where elements are just a few places off from their correct spot)

To demonstrate the use of heaps, below we find the *k*-largest elements in a list, using the heapq package in Python:


```
import heapq
k = 5
a = [13, 5, 2, 6, 10, 9, 7, 4, 3]
heapq.heapify(a) # creates heap
heapq.nlargest(k, a) # finds k-largest
```

Graphs

Graphs are composed of nodes (vertices) and a set of edges that connect the nodes. Edges are often represented in two ways: an adjacency matrix or an adjacency list. In the case of an adjacency matrix, there is a *n*-by-*n* matrix for *n*-nodes where the entries are either Boolean values (denoting an edge is present or not between two nodes) or a weight. In contrast, an adjacency list stores a list of neighbors for each node.



To Graph

	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	1	0	0
D	0	0	1	0

To Adjacency Matrix

Node	Adjacent Node(s)
A	B C D
B	D
C	B
D	C

To Adjacency List

The lookup time to check whether two nodes are neighbors for an adjacency matrix is O(1). For an adjacency list, it could be O(*N*) in the worst case (if a node has edges to every other node). What an adjacency list lacks in time efficiency compared to an adjacency matrix, it makes up for in space efficiency. For a large graph with *n* nodes that are sparse (the *N* nodes don't have many connections to each other), an adjacency list offers a compact way to represent the edges, versus an adjacency matrix which requires you to store an *N*^2 sized matrix in memory. For instance, consider the Facebook friendship graph, which has 2 billion users (nodes), but a maximum of only 5,000 connections per node (the 5,000 friend limit). A full adjacency matrix would need to be 2 billion rows by 2 billion columns, whereas an adjacency list would require considerably less memory.

An example implementation of a graph is below. The Vertex class uses an adjacency list to keep track of its neighbors with weights:

```
class Vertex:
    def __init__(self, val):
        self.val = val
        self.neighbors = {}

    def add_to_neighbors(self, neighbor, w): # add to neighbor dict with weight
        self.neighbors[neighbor] = w

    def get_neighbors(self):
        return self.neighbors.keys()
```

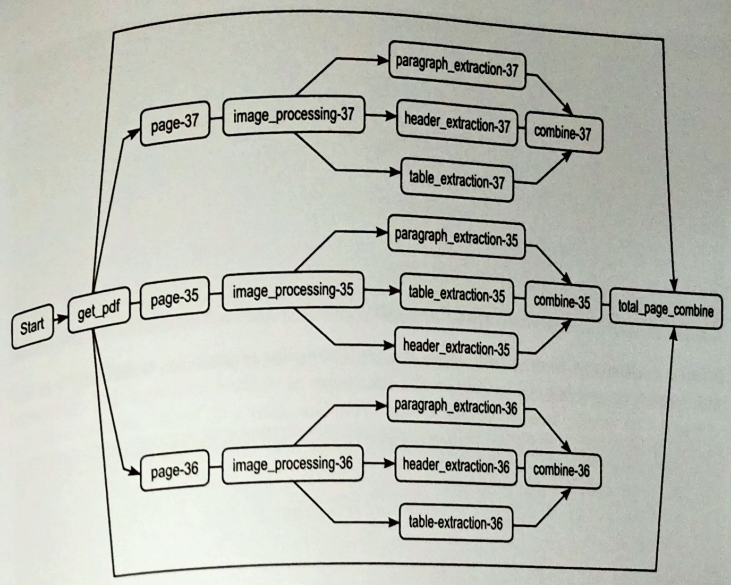
```
class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, val):
        new_vertex = Vertex(val)
        self.vertices[val] = new_vertex
        return new_vertex

    def add_edge(self, u, v, weight): # add edge from u to v
        if u not in self.vertices:
            self.add_vertex(Vertex(u))
        if v not in self.vertices:
            self.add_vertex(Vertex(v))
        self.vertices[u].add_to_neighbors(self.vertices[v], weight)

    def get_vertex(self, val):
        return self.vertices[val]

    def get_vertices(self):
        return self.vertices.keys()

    def __iter__(self):
        return iter(self.vertices)
```

Real-World Applications of Graphs

Graphs serve as a prevalent data structure for representing many real-world problems. For example, PageRank, developed by Google co-founders Larry Page and Sergey Brin, is an algorithm that measures how important various web pages are. PageRank represents each web page as a node, and each hyperlink from one page to another represents an edge. The underlying assumption is that important web pages are more likely to have been linked to by other influential pages on the web. In the context of graphs, this means that nodes with a high number of edges from other high-quality nodes are likely to have a better PageRank score.

Another real-world use of graphs is for computational graphs used within scheduling tools like Airflow or large-scale data processing frameworks like Spark or Tensorflow. These frameworks represent the series of computations and data flows that need to be performed as a directed acyclic graph (DAG), which is a directed graph (meaning edges are unidirectional) that has no cycles. Nodes represent operations that create or manipulate data, and edges represent how data (typically multidimensional arrays also known as tensors) must flow from one operation (node) to another.

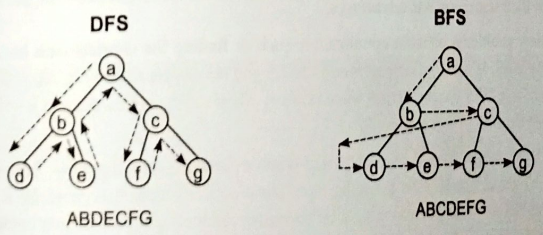
These computational graphs enable parallelism, since it's easy to order the operations in such a way that certain operations that do not depend on one another can be run concurrently. Another advantage is that the computational graph offers a language-agnostic representation of the computations we want, which can easily be ported between underlying frameworks.



An example Airflow execution graph to extract a data table from a large PDF file using image processing

Breadth-First and Depth-First Search

Interview problems related to graphs often involve traversing the graph, either in a breadth-first-search (BFS) or in a depth-first-search (DFS). In BFS, start with one node and add it to a queue. For each node in the queue, process the node and add its neighbors to the queue. In DFS, you also start with one node, but instead of processing all the neighbors next, you recursively process each neighbor one by one.



Below is an example of BFS using a queue for iterative processing:

```
def bfs(graph, v):
    n = len(graph.vertices)
    visited = [False] * (n+1)
    queue = []
```

```
    queue.append(v)
    visited[v] = True

    while queue:
        curr = queue.pop(0)
        for i in graph.get_vertex(curr):
            if visited[i.val] == False:
                queue.append(i.val)
                visited[i.val] = True
    return visited
```

Below is a primitive example of DFS, where *visited* tracks the set of processed nodes, and *v* is the node currently being processed:

```
def dfs_helper(graph, v, visited):
    visited.add(v)
    for neighbor in graph.get_vertex(v).get_neighbors():
        if neighbor.val not in visited:
            dfs_helper(graph, neighbor.val, visited)
    return visited


def dfs(graph, v):
    visited = set()
    return dfs_helper(graph, v, visited)
```

The runtime for both is O(*E* + *V*), where E is the number of edges in the graph and V is the number of vertices. For most basic use cases, DFS and BFS are interchangeable. However, there are some differences in their use cases and advantages.

When interview problems concern optimization, such as finding the shortest path between nodes, consider using BFS. It can find solutions definitively and never gets trapped in recursive sub-calls. Some drawbacks to BFS are that it uses more memory when storing nodes and can take a lot of time if solutions are far away from the root.

When interview problems concern analyzing a graph's structure, such as when finding cycles or orderings in directed graphs (for example, topological sorting), consider using DFS. Because it exhausts paths before trying others, it may be trapped recursively and cannot guarantee a solution. However, DFS has fewer memory requirements and can find long-distance elements in a shorter amount of time compared to BFS.

Algorithms

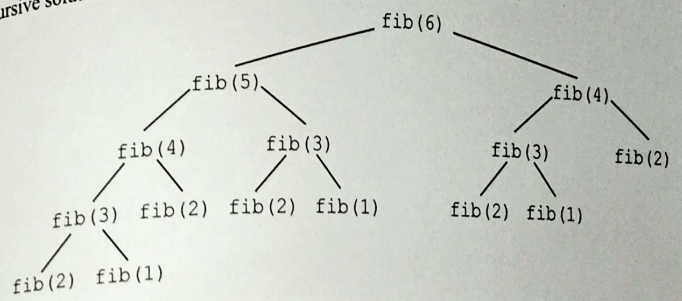
Recursion

A recursive function is one that calls itself directly or indirectly. It can be broken down into two parts: the recursive case (the part that calls itself) and the base case (which terminates the recursion).

For an example of a recursive algorithm, consider the classic problem of producing Fibonacci numbers (0, 1, 1, 2, 3, 5, 8, etc.):

```
def fib(n):
    if n == 0:
        return 0
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

If you draw the tree of recursive calls, you'll see that many times fib(*n*) for a particular *n* is called repeatedly. This repetition leads to an undesirable exponential runtime and memory usage. Since recursive calls are stored on a stack, and often, there is limited allocated memory for a program, recursive solutions can also lead to stack overflows which crash the program.



In contrast, iterative algorithms do not overwhelm the call stack. Such algorithms rely on using a for-loop or while-loop, rather than calling themselves repeatedly like in recursion. An iterative example of the Fibonacci sequence is as follows: we use a for-loop to iterate up to *n*, set the current result to be the sum of the previous Fibonacci number at indexes *n*-2 and *n*-1, and continue this until we reach the desired *n*:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1 or n == 2:
        return 1
    else:
        prev2 = 0 # fib(n-2)
        prev = 1 # fib(n-1)
        res = 0 # filb(n)
        for i in range(1, n): # process iteratively
            res = prev2 + prev
            prev2 = prev
            prev = res
        return res
```

Compared to the recursive solution, the iterative approach is less expressive code-wise but more memory efficient since there aren't $O(N)$ recursive calls, but instead just two variables to keep track of (the most recent two Fibonacci numbers).

Greedy Algorithms

Greedy algorithms choose at each step what seems to be the best option. In other words, greedy problems reduce problems into smaller ones by making the locally optimal choice at each step.

A classic example where a greedy algorithm works well is when making change with the minimum number of coins. Say, for instance, you wanted to make change for 67 cents, and all you had were pennies, nickels, dimes, and quarters. The greedy approach is to use as many coins of the highest denomination as possible (two quarters), before continuing to the next denomination (one dime). The code is as follows:

```
def minCoins(k):
    coins = [1, 5, 10, 25]
    n = len(coins)
    res = []
    i = n-1
    while(i >= 0 and k >= 0):
        if k >= coins[i]:
            k -= coins[i]
            res.append(coins[i])
        else:
            i -= 1
    return res
```

In the real world, greedy algorithms show up in various areas of machine learning. For example, decision trees are split in a greedy manner to maximize information gain (reduction in entropy based on the feature chosen) at each split. For every feature, we evaluate all possible features, then, in a greedy fashion, choose the feature with the best information gain to split on next. This process takes place recursively until we end up with leaf nodes. The pseudocode for getting the best feature is as follows:

```
info_gains = [getInfoGain(feature) for feature in features]
best_feature_index = np.argmax(info_gains)
best_feature = features[best_feature_index]
```

For interview questions, keep greedy algorithms in mind for optimization problems, where there's an obvious set of choices to select from, and it's easy to know what the appropriate choice is. Keep in mind that it's often easier to reason about a greedy algorithm recursively, but then implement it later iteratively for better memory performance.

Dynamic Programming

Earlier in the recursion section, we saw how an iterative solution to generating the n-th Fibonacci number had its advantage over a recursive version since it used less memory and didn't recompute

work. That's where dynamic programming, the art of storing results of subproblems to speed up recursion, comes in. Although, technically, the iterative solution serves as a form of dynamic programming, to be more explicit, for pedagogical purposes, below we show how an array can be used to cache existing results. Now, instead of making the previous recursive calls all the way down, we get the stored sub-result from the cache to prevent values from being recomputed.

```
dp = [0] * 1000

def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        dp[n] = fib(n-1) + fib(n-2)
        return dp[n]
```

Two properties are needed for dynamic programming (DP) to be applicable:

1) optimal substructure
2) overlapping subproblems

An optimal substructure means the problem can be solved optimally by breaking it down into subproblems and solving those subproblems. Overlapping subproblems indicates the problem can be broken down into subproblems, that are then reusable in other subproblems. If both are present, then calculating and storing the solutions to subproblems in a recursive manner will solve the overall problem. Note that the Fibonacci satisfies the two requirements because:

1) Fib(n) is found by solving the subproblems fib(n-1) and fib(n-2)
2) Subproblems are overlapping: both fib(n) and fib(n-1) require having solved fib(n-2)

Dynamic programming can be implemented with a top-down or bottom-up approach. In a top-down approach, we start with the top and break the problem into smaller chunks (as in the Fibonacci example). In practice, the top-down approaches often are implemented with recursion, and the intermediate results are cached in a hash table.

In contrast, in a bottom-up manner, we start with the smaller problems and continue to the top. These solutions are often implemented iteratively, where an n-dimensional array is used to cache previous results. A bottom-up Fibonacci example would be:

```
def fib(n):
    dp = [0 for _ in range(n+1)]
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Dynamic programming isn't just an academic exercise or coding interview favorite — it often shows up in the real world, too. For instance, in reinforcement learning, the goal is to understand what

actions to take given a particular state of the universe to maximize an expected eventual payoff. The famous Bellman equations are a core part of the reinforcement learning process and utilize the dynamic programming approach. The Bellman equations break down the total eventual payoff into a series of smaller subproblems that can each be optimized, where the best eventual payoff comes from combining different subproblem payoffs.

Greedy Algorithms Vs. Dynamic Programming

As explained earlier, a greedy algorithm does whatever is locally optimal and hence always chooses the option that leads to the best result for the next step. This contrasts with dynamic programming, which will exhaust the search space and is guaranteed to find the globally optimal solution, not just the locally optimal solution. Therefore, you can think of greedy algorithms as getting the "local maximum minimum," but not necessarily the "global maximum/minimum" that dynamic programming achieves.

For a concrete example of where greedy algorithms can fall short, consider the classic 0/1 knapsack problem with the values below. In this problem, we are choosing between N items, each with some positive value and positive weight. We want to maximize the amount of value obtained by selecting items with a total weight of no more than W.

Example Knapsack

Item	Value	Weight	Value/Weight
A	3	4	3/4
B	1.2	2	1.2/2
C	2	3	2/3

The greedy implementation is to just sort the elements that maximize value per unit weight and choose the most efficient items.

```
def knapsackGreedy(values, weights, max_weight):
    curr_weight = 0
    value_list, weight_list = ([], []) # values and weights
    total_sorted = sorted(zip(values, weights),
                          key=lambda x: x[0]/x[1], reverse=True)
    values_sorted, weights_sorted = zip(*total_sorted)
    # iterate in sorted fashion
    for value, weight in zip(values_sorted, weights_sorted):
        if weight + curr_weight <= max_weight: # meets weight requirement
            value_list.append(value)
            weight_list.append(value)
            curr_weight += weight
    return sum(value_list)
```

However, this does not maximize the amount of value obtained overall. Take, for example, the table of weights and values provided earlier, along with a total weight limit of 5.

Greedily, we would select item A (the highest value/weight ratio) and end up with a total value of 3, since the weight constraint does not allow for extra items. However, the best possible method would be to take B and C for a total value of 3.2, which is larger than 3.

In contrast, a dynamic programming approach achieves the global maximum. An example of a bottom-up approach is below, where the core logic is in deciding whether to take an item or not. If an item meets the weight requirement, then we either

- Take the item, and get a new optimal maximum value, which now gives us a new weight constraint (current weight minus item's weight), or
- Do not take the item, and continue with the current optimal maximum value and the current weight constraint.

```
def knapsackDP(values, weights, max_weight):
    n = len(values)
    dp = [[0 for x in range(max_weight+1)] for x in range(n+1)] # 2d-array

    for i in range(n+1):
        for w in range(max_weight+1):
            if weights[i-1] <= w: # meets weight requirement
                dp[i][w] = max(values[i-1]+dp[i-1][w-weights[i-1]],
                               dp[i-1][w]) # either take value or not

            else:
                dp[i][w] = dp[i-1][w] # not taking value
    return dp[n][max_weight]
```

The dynamic programming approach coded above leads to the optimal solution of taking items B and C for a total value of 3.2. When solving coding interview problems, it's crucial to not fall for a more apparent greedy approach when the correct answer is found through dynamic programming.

Sorting

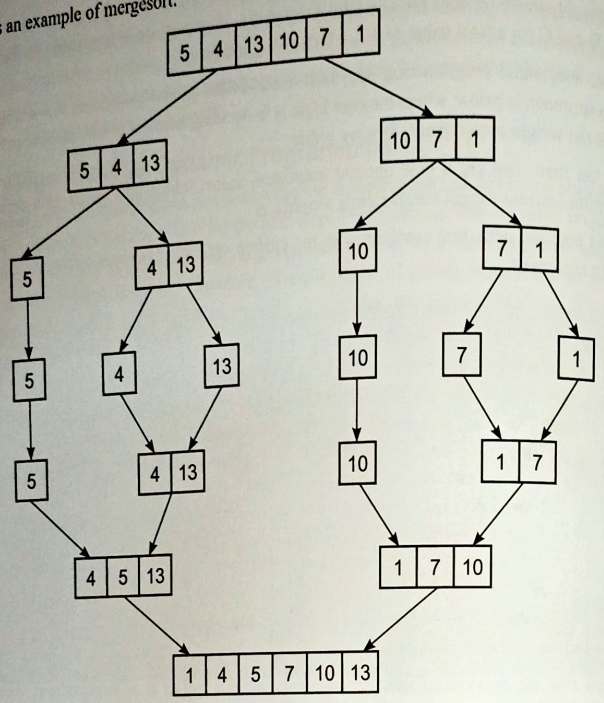
Two unique sorting algorithms — mergesort and quicksort — arise in coding interviews from time to time. While it's rare to be asked to code up these algorithms, both sorts are often modified to solve a problem or used as an intermediate step within a larger coding interview solution. This is because sorting the input can expose some structure, which makes the problem simpler.

Mergesort

Mergesort uses a "divide-and-conquer" approach as follows:

1. Repeatedly divide the input into smaller subarrays, until a base case of a single element is reached (this single element subarray is considered sorted).
2. Repeatedly merge the smaller sorted subarrays into bigger sorted subarrays, until the entire input is merged back together.

Below is an example of mergesort:



Overall, mergesort has a runtime complexity of O(N log N) and also requires O(N log N) space to support the auxiliary merging steps. An example implementation of mergesort that utilizes a helper function for merging subarrays is as follows:

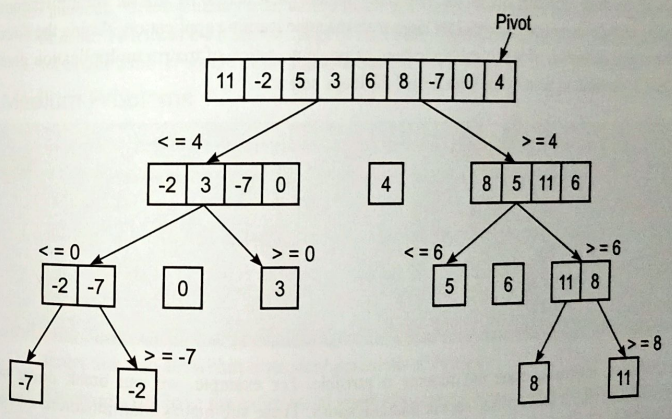
```
def merge_helper(a, low, high, mid):
    if len(a) == 1:
        return a
    i = j = k = 0 # k is for merged array
    left = a[:mid] # copy of left side
    right = a[mid:] # copy of right side
    while i < len(left) and j < len(right): # compare left and right sides, add
        if left[i] < right[j]:
            a[k] = left[i]
            i += 1
        else:
            a[k] = right[j]
            j += 1
        k += 1
    while i < len(left): # remaining on left
        a[k] = left[i]
```



```
        i += 1
        k += 1
    while j < len(right): # remaining on right
        a[k] = right[j]
        j += 1
        k += 1
    return a

def mergesort(a, low, high):
    if low >= high:
        return a
    mid = (low + high-1) // 2
    mergesort(a, low, mid)
    mergesort(a, mid+1, high)
    merge_helper(a, low, high, len(a) // 2)
    return a
```

Quicksort

Quicksort selects an arbitrary pivot element and puts all elements smaller than the pivot to the left of the pivot, and larger elements to the right of the pivot. The exchanging of elements occurs through swaps. This process of selecting a pivot and swapping the left and right elements ensues recursively, until the base case is hit when just two elements are swapped into their correct relative order.



The worst-case runtime for quicksort is O(N^2), although, in practice the expected runtime is closer to O(N log N) through picking a "smart" pivot whereby the elements are roughly divided into equal halves upon each iteration.

Below is an example implementation of quicksort that utilizes a helper function to partition the array:

```
def helper(a, low, high):
    i = low-1 # smaller element index
    pivot = a[high] # pivot
    for j in range(low, high):
        if a[j] <= pivot:
            i += 1 # increment
            a[i], a[j] = a[j], a[i] # swap
    a[i+1], a[high] = a[high], a[i+1]
    return i+1

def quicksort(a, low, high):
    if len(a) == 1:
        return a
    if low < high:
        pivot = helper(a, low, high) # place pivot
        quicksort(a, low, pivot-1) # recurse left side
        quicksort(a, pivot+1, high) # recurse right side
    return a
```

Matrix Multiplication

Since all machine learning algorithms eventually reduce to a series of matrix multiplications, speeding up these operations is crucial for large-scale machine learning applications. Using the direct mathematical definition of matrix multiplication, an implementation of matrix multiplication gives an O(*N*^3) runtime, as seen by the three nested for-loops below:

```
def matrix_multiply(A, B):
    m = [[0 for row in range(len(B[0]))] for col in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                m[i][j] = m[i][j] + A[i][k] * B[k][j] # add
    return m
```

Nevertheless, improving upon this runtime is possible. For example, we can break down the multiplication into a series of sub-matrix multiplications. These sub-matrix multiplications can be calculated in parallel, which enables the task to be accomplished in a distributed manner.

Finance companies like to ask candidates about coding up matrix multiplication or implementing other common linear algebra topics such as singular-value decomposition because it allows firms to test your mathematical understanding and programming ability at once. These questions are also picked because optimizing numerical calculations done in linear algebra is a relevant concept when trying to reduce the latency of trading systems.

Interview Questions

Easy Problems

9.1. Amazon: Given two arrays, write a function to get the intersection of the two. For example, if A = [1, 2, 3, 4, 5], and B = [0, 1, 3, 7] then you should return [1, 3].

9.2. D.E. Shaw: Given an integer array, return the maximum product of any three numbers in the array. For example, for A = [1, 3, 4, 5], you should return 60, while for B = [−2, −4, 5, 3] you should return 40.

9.3. Facebook: Given a list of coordinates, write a function to find the *k* closest points (measured by Euclidean distance) to the origin. For example, if *k* = 3, and the points are: [[2, −1], [3, 2], [4, 1], [−1, −1], [−2, 2]], then return [[−1, −1], [2, −1], [−2, 2]].

9.4. Google: Say you have an *n*-by-*n* matrix of elements that are sorted in ascending order both in the columns and rows of the matrix. Return the *k*-th smallest element of the matrix. For example, consider the matrix below:

$$\begin{bmatrix} 1 & 4 & 7 \\ 3 & 5 & 9 \\ 6 & 8 & 11 \end{bmatrix}$$

If *k* = 4, then return 5.

9.5. Akuna Capital: Given an integer array, find the sum of the largest contiguous subarray within the array. For example, if the input is [−1, −3, 5, −4, 3, −6, 9, 2], then return 11 (because of [9, 2]). Note that if all the elements are negative, you should return 0.

9.6. Facebook: Given a binary tree, write a function to determine whether the tree is a mirror image of itself. Two trees are a mirror image of each other if their root values are the same and the left subtree is a mirror image of the right subtree.

Medium Problems

9.7. Google: Given an array of positive integers, a peak element is greater than its neighbors. Write a function to find the index of any peak elements. For example, for [3, 5, 2, 4, 1], you should return either 1 or 3 because the values at those indexes, 5 and 4, are both peak elements.

9.8. AQR: Given two lists X and Y, return their correlation.

9.9. Amazon: Given a binary tree, write a function to determine the diameter of the tree, which is the longest path between any two nodes.

9.10. D.E. Shaw: Given a target number, generate a random sample of *n* integers that sum to that target that also are within σ standard deviations of the mean.

9.11. Facebook: You have the entire social graph of Facebook users, with nodes representing users and edges representing friendships between users. Given a social graph and two users as input, write a function to return the smallest number of friendships between the two users. For example, take the graph that consists of 5 users A, B, C, D, E, and the friendship edges are: (A, B), (A, C), (B, D), (D, E). If the two input users are A and E, then the function should return 3 since A is friends with B, B is friends with D, and D is friends with E.

9.12. LinkedIn: Given two strings A and B, write a function to return a list of all the start indices within A where the substring of A is an anagram of B. For example, if A = "abcdcbac" and

B = "abc," then you want to return [0, 4, 5] since those are the starting indices of substrings of A that are anagrams of B.

9.13. Yelp: You are given an array of intervals, where each interval is represented by a start time and an end time, such as [1, 3]. Determine the smallest number of intervals to remove from the list, such that the rest of the intervals do not overlap. Intervals can "touch," such as [1, 3] and [3, 5], but are not allowed to overlap, such as [1, 3] and [2, 5]). For example, if the input interval list given is: [[1, 3], [3, 5], [2, 4], [6, 8]], then return 1, since the interval [2, 4] should be removed.

9.14. Goldman Sachs: Given an array of strings, return a list of lists where each list contains the strings that are anagrams of one another. For example, if the input is ["abc", "abd", "cab", "bad", "bca", "acd"] then return: [["abc", "cab", "bca"], ["abd", "bad"], ["acd"]].

9.15. Two Sigma: Say that there are *n* people. If person A is friends with person B, and person B is friends with person C, then person A is considered an indirect friend of person C.

Define a friend group to be any group that is either directly or indirectly friends. Given an *n*-by-*n* adjacency matrix *N*, where *N*[i][j] is one if person *i* and person *j* are friends, and zero otherwise, write a function to determine how many friend groups exist.

9.16. Workday: Given a linked list, return the head of the same linked list but with *k*-th node from the end of a linked list removed. For example, given the linked list $3 \rightarrow 2 \rightarrow 5 \rightarrow 1 \rightarrow 4$ and $k = 3$, remove the 5 node and, thus, return the linked list $3 \rightarrow 2 \rightarrow 1 \rightarrow 4$.

9.17. Goldman Sachs: Estimate π using a Monte Carlo method. Hint: think about throwing darts on a square and seeing where they land within a circle.

9.18. Palantir: Given a string with lowercase letters and left and right parentheses, remove the minimum number of parentheses so the string is valid (every left parenthesis is correctly matched by a corresponding right parenthesis). For example, if the string is "))a(b((cd)e(f)g)" then return "ab((cd)e(f)g)".

9.19. Citadel: Given a list of one or more distinct integers, write a function to generate all permutations of those integers. For example, given the input [2, 3, 4], return the following 6 permutations: [2, 3, 4], [2, 4, 3], [3, 2, 4], [3, 4, 2], [4, 2, 3], [4, 3, 2].

9.20. Two Sigma: Given a list of several categories (for example, the strings A, B, C, and D), sample from the list of categories according to a particular relative weighting scheme. For example, say we give A a relative weight of 5, B a weight of 10, C a weight of 15, and D a weight of 20. How do we construct this sampling? How do you extend the solution to an arbitrarily large number of *k* categories?

9.21. Amazon: Given two arrays with integers, return the maximum length of a common subarray within both arrays. For example, if the two arrays are [1, 3, 5, 6, 7] and [2, 4, 3, 5, 6] then return 3, since the length of the maximum common subarray, [3, 5, 6], is 3.

9.22. Uber: Given a list of positive integers, return the maximum increasing subsequence sum. In other words, return the sum of the largest increasing subsequence within the input array. For example, if the input is [3, 2, 5, 7, 6], return 15 because it's the sum of 3, 5, 7. If the input is [5, 4, 3, 2, 1], return 5 (since no subsequence is increasing).

9.23. Palantir: Given a positive integer *n*, find the smallest number of perfect squares that sum up to *n*. For example, for $n = 7$, you should return 4, since $7 = 4 + 1 + 1 + 1$. For $n = 13$, you should return 2, since $13 = 9 + 4$.

9.24. Facebook: Given an integer *n* and an integer *k*, output a list of all of the combinations of *k* numbers chosen from 1 to *n*. For example, if $n = 3$ and $k = 2$, return [1, 2], [1, 3], [2, 3].

Hard Problems

9.25. Citadel: Given a string with left and right parentheses, write a function to determine the length of the longest well-formed substring. For example, if the input string is ")()()(," then return 4, since the longest well-formed substring is "()()."

9.26. Bloomberg: Given an *m*-by-*n* matrix with positive integers, determine the length of the longest path of increasing integers within the matrix. For example, consider the input matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

In this case, return 5, since one of the longest paths would be 1-2-5-6-9.

9.27. Google: Given a number *n*, return the number of lists of consecutive positive integers that sum up to *n*. For example, for $n = 9$, return 3, since the consecutive positive integer lists are: [2, 3, 4], [4, 5], and [9]. Can you solve this in linear time?

9.28. Citadel: Given a continuous stream of integers, write a class with functions to add new integers to the stream, and a function to calculate the median at any time.

9.29. Two Sigma: Given an input string and a regex, write a function that checks whether the regex matches the input string. The input string is composed of the lowercase letters a-z. The regular expression contains lowercase a-z, '?', or '*', where the '?' matches any one character, and the '*' matches an arbitrary number of characters (empty as well). For example, if the input string is "abcdba" and the regex is "a*c?*", return true. However, if the regex was instead "b*c?*" return false.

9.30. Citadel: A fire department wants to build a new fire station in a location where the total distance from the station to all houses in the town (in Euclidean terms) is minimized. Given a list of coordinates for the n houses, return the coordinates of the optimal location for the new fire station.

Solutions

Solution #9.1

The simplest way to check for intersecting elements of two lists is to use a doubly-nested for-loop to iterate over one array and check against every element in the other array. However, this leads to a time complexity of $O(N*M)$ where N is the length of A, and M is the length of B.

A better approach is to use sets (which utilizes a hash map implementation underneath) since the runtime time is O(1) for each lookup operation. Then we can do the series of lookups over the larger set (resulting in a $O(\min(N, M))$ total runtime):

```
def intersection(a, b):
    set_a = set(a)
    set_b = set(b)
```

```
    if len(set_a) < len(set_b):
        return [x for x in set_a if x in set_b]
    else:
        return [x for x in set_b if x in set_a]
```

The time complexity is O($N + M$) due to the creation of the sets, and the space complexity is also O($N + M$) since the arrays might contain all distinct elements.

Solution #9.2

If all of the numbers were positive, then the maximum product of three numbers is a matter of finding the largest three numbers in the array and multiplying them. Be sure to clarify with the interviewer what's in the integer array — don't just surmise they are all positive integers. However, with negative integers, the largest product could arise if we take the two smallest numbers (both could be negative) and multiply that result by the largest positive number. We'd need to compare this potential product to the number involving just the largest three numbers.

By first sorting the array, you can get the largest three numbers and the smallest two numbers. Alternatively, we can use a heap (finding the largest three numbers using a max-heap, and the smallest two numbers using a min-heap), rather than sorting. An implementation involving heaps is below:

```
import heapq

def max_three(arr):
    a = heapq.nlargest(3, arr) # largest 3 numbers
    b = heapq.nsmallest(2, arr) # smallest 2 (for negatives case)
    return max(a[2]*a[1]*a[0], b[1]*b[0]*a[0]) # compare
```

The time complexity is where O(N) is the size of the input array and the space complexity is O(1) for the heap, since the number of elements is fixed.

Solution #9.3

The brute-force solution to finding the k-closest coordinates to the origin would be to iterate over all coordinates, compute the distance from each coordinate to the origin, and then sort by distance to find the k-closest ones. This would yield a runtime of O(N log N) because of sorting N coordinates. A more optimal way is to instead utilize a min-heap whereby we add coordinates based on the Euclidean distance. Then, at the end, we can just take the first k elements, which will already be the smallest due to the min-heap:

```
from heapq import heappush, heappop

def closest(points, k):
    def get_dist(x, y): # distance helper function
        return x**2 + y**2
    min_heap = [] # heap
```



```
    n = len(points)
    for i in range(n):
        x = points[i][0]
        y = points[i][1]
        heappush(min_heap, (get_dist(x, y), points[i])) # add to heap
    res = []
    for i in range(k):
        res.append(heappop(min_heap)[1]) # get top k
    return res
```

The time complexity is O(N*log(K)), since we have O(N) additions to the heap, each of which takes O(log K) time. The space complexity is O(K) to accommodate the min-heap.

Solution #9.4

The simple approach to finding the k-th smallest element is to iterate over all elements in the matrix, add each element to a min-heap, and then pop from the heap k times. However, because the matrix is sorted, we know that the k-th smallest element must be within row k and column k. Therefore, we can use a double for-loop with k as the max value for each of the two indices. Once all elements are added, then we can pop the top-k smallest elements from the stack and return the last element.

```
from heapq import heappush, heappop

def smallest(m, k):
    n = len(m)
    heap = [] # heap
    res = -1
    for i in range(min(k, n)):
        for j in range(min(k, n)):
            heappush(heap, m[i][j]) # add to heap
    for _ in range(k):
        res = heappop(heap) # pop k times
    return res
```

We will presume, that, in general $K << N$ (although the code handles any event in which this is not the case. Then the time complexity is O(K^2 log K) because each heap push operation takes O(log K) time, and this operation is within a doubly-nested for-loop. The space complexity is O(K) to support the heap.

Solution #9.5

A brute-force way solution to finding the sum of the largest contiguous subarray is to compute the sum over all possible contiguous subarrays, and then return the biggest sum found. This would have a runtime complexity of O(N^2) due to the doubly-nested for-loops. However, there is no need to do multiple passes: in a single iteration of the array, if we keep track of the current sum and it ever goes below 0, there is no need to include elements in that previous subarray, and we can set the current sum to 0. Note that if the array is all negatives, then we get a final result of 0 (by taking no elements).

While doing the single iteration through the array, we keep track of the maximum seen at every point and return that at the end:

```
def max_subarray(arr):
    n = len(arr)
    max_sum = arr[0] # max
    curr_sum = 0 # current sum
    for i in range(n):
        curr_sum += arr[i]
        max_sum = max(max_sum, curr_sum) # get max
        if curr_sum < 0: # reset
            curr_sum = 0
    return max_sum
```

The time complexity is O(N), where N is the size of the input array, and the space complexity is O(1).

Solution #9.6

Finding out if a tree is a mirror image of itself can be solved recursively in a straightforward manner. We will use the TreeNode class implementation below. The helper function checks that for two subtrees:

a) The root values match

b) The first subtree's left side is a mirror of the second subtree's right side

c) The first subtree's right side is a mirror of the second subtree's left side

This helper function is then called on the root's left and right subtrees:

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

def is_mirror(root):
    if root is None:
        return True
    return helper(root.left, root.right)
def helper(x, y):
    if (x is None and y is None):
        return True
    elif (x is None or y is None):
        return False
    return x.val == y.val and
        helper(x.left, y.right) and
        helper(x.right, y.left)
```


The time complexity and space complexity are both O(N), where N is the size of the tree since the entire tree is traversed once.

Solution #9.7

The brute-force way to check for peak elements is to iterate through the array and check, for each element, whether the left and right neighbor is less than the element's value. That would yield an O(N) runtime for one linear scan. However, we can improve upon this by doing a modified binary search. Take the middle value and check the number to the left of it (say it is smaller). If the middle value's right element is also smaller, then the middle value is a local peak. Alternatively, if the right value is greater than the middle value, then there must be a local peak on the right-hand side (numbers ascending and then a descent, or it keeps ascending till it reaches the end of the array, which is also a local peak). We do this in an iterative manner until we reach the condition that the element's value is larger than both its left and right neighbor values.

```
def get_peak(arr):
    start = 0
    end = len(arr) - 1
    while True:
        mid = (start + end) // 2 # get midddle element and check bounds
        left = arr[mid-1] if mid - 1 >= 0 else float('-inf')
        right = arr[mid+1] if mid + 1 < len(arr) else float('-inf')
        if left < arr[mid] and right < arr[mid]: # left < mid > right
            return mid
        elif arr[mid] < right: # change bounds
            start = mid + 1
        else:
            end = mid - 1
```

The time complexity is O(log(N)) since it is a variant of binary search and the space complexity of O(1).

Solution #9.8

We know that correlation is given by: $\rho_{x,y} = \dfrac{Cov(X,Y)}{\sigma_X * \sigma_Y}$

where the numerator is the covariance of X and Y, and the denominator is the product of the standard deviation of X and the standard deviation of Y. Recall the definition of covariance:

$$Cov(X, Y) = E[(X - \mu_X)(Y - \mu_Y)]$$

Therefore, a simple implementation is to have helper functions for calculating both the mean and standard deviation. Note that calculating both the mean and standard deviation is O(N) runtime (since both take a single sum across all N elements). Therefore, the correlation runtime is O(N), since we calculate a few means and standard deviations, as well as iterate over all N elements once through. The space complexity is O(N) to keep track of N elements in the correlation to be processed.

```
import math

def mean(x):
    return sum(x)/len(x)

def sd(x):
    m = mean(x)
    ss = sum((i-m) ** 2 for i in x)
    return math.sqrt(ss / len(x))

def corr(x, y):
    x_m = mean(x)
    y_m = mean(y)
    xy_d = [] # product of de-meaned X and Y, for covariance calc
    for i in range(len(x)):
        x_d = x[i] - x_m
        y_d = y[i] - y_m
        xy_d.append(x_d * y_d) # add product of X_i and Y_i
    return mean(xy_d) / (sd(x) * sd(y)) # from formula above
```

Solution #9.9

Finding the diameter of a tree can be solved recursively, since at any given root node, the diameter is just one more than the maximum of the depths from its left and right subtrees. Depth, in this context, is defined as the number of nodes from the root to the farthest leaf node. We can utilize a helper function called depth() to calculate tree depth, and have its base case to return 0 if the root is None. Inside the diameter function, we want to recursively call depth(root.left) and depth(root.right) and return 1 + max(left result, right result).

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

def calc_diameter(root):
    def depth(root, diameter): # helper
        if root is None:
            return 0, diameter # base case
        left, diameter = depth(root.left, diameter)
        right, diameter = depth(root.right, diameter)
        diameter = max(diameter, left + right) # update diameter
        return max(left, right) + 1, diameter # return root's longest path
    depth, diameter = depth(root, 0) # call helper
    return diameter
```


The time complexity is O(N), where N is the size of the tree, since the entire tree is traversed once. The space complexity is also O(N), since there will be N recursive calls on the stack in the worst case.

Solution #9.10

To generate the numbers at random, we can start by calculating the mean and defining upper and lower bounds on any given integer based on the standard deviation. Let the target be represented by T, the number of elements by N, and the sigma by σ. The mean μ will be T/N, and the allowable standard deviation is given by σ * μ. That means any given element must be within σ * μ of μ, which establishes our lower and upper bounds.

Once we have the upper and lower bounds, we can take all n integers, starting each at the lower bound, and increment them at random in an iterative manner until the sum of the numbers is the given target value. For random incrementing, we can randomly sample one of the N elements and increment it by 1 if it is not beyond the upper bound. If it is, we skip it for that iteration. An example implementation is as follows:

```
import random

def generate_nums(target, n, sigma):
    mean = target / n
    sd = int(sigma * mean)
    max_val, min_val = mean + sd, mean - sd # bounds
    results = [min_val] * n
    remaining = target - n * min_val

    while remaining > 0:
        a = random.randint(0, n-1) # choose random index
        if results[a] >= max_val: # skip if above bound
            continue
        results[a] += min(remaining, 1)
        remaining -= 1
    return results
```

After creating the initial results array of n elements that are each *min_val*, let R be the remaining amount results need to be incremented to hit the target sum.

The while loop has a time complexity of O(R), which is a function of O(T*N) for the following reasons:

$$R = T - n\left(\mu - \sigma * \frac{T}{n}\right)$$

based on the above formulation. Simplifying yields: $R = T - n(\mu) + \sigma * T$

Recall that by definition $T \approx n * \mu$ therefore, we can simplify the above to be:

$$R = T - n\left(\frac{T}{n}\right) + \sigma * T = \sigma * T$$

Thus, the runtime is a function of O(max(σ * T), N) since the bottleneck is either the while loop, or in generating the results of size N initially. The space complexity will be O(N) to store the results.

Solution #9.11

To find the shortest distance in a graph between two nodes (in this case, the number of friends between two individuals), we can use a breadth-first-search (BFS) and maintain a list of distances from one of the input users. Each distance in this list starts at 0. In typical BFS fashion, we utilize a queue and add the initial node to the queue, processing the queue in an iterative fashion as long as it is not empty. At every step, for any friend node we see, we update the distance value for that friend node. Since the distance array will track all of the distances from the starting user x, we return the distance of the other user y:

```
import queue

def friendship_distance(n, edges, x, y):
    distance = [0] * n # distances from x
    processed = [False] * n # visited or not already
    q = queue.Queue()
    q.put(x)
    processed[x] = True
    while (not q.empty()):
        curr = q.get() # current node
        if curr not in edges:
            continue
        for neighbor in range(len(edges[curr])):
            if not processed[edges[curr][neighbor]]: # process if not visited
                distance[edges[curr][neighbor]] = distance[curr] + 1
                q.put(edges[curr][neighbor]) # add neighbor
                processed[edges[curr][neighbor]] = True
    return distance[y]
```

The time complexity is O(N*M), where N is the number of users, and M is the number of friendships. The space complexity is O(N) to maintain the queue.

Solution #9.12

The brute-force way to find all the anagrams of B within A is to find all substrings of A and then check if each is an anagram of B. This is inefficient because it leads to an O($N^{2*}K$) runtime since there are N^2 substrings of A, and the anagram check takes O(K) time.

A better way is to have a rolling window approach: say the length of B is k. We can iterate over A and check a window of size k and compare that window to B to check for an anagram match.

A basic way to confirm if two strings are anagrams is to check if the sorted versions of the two strings are equal. However, that is O(N log N) due to sorting. To speed this up to an O(N) check, we can utilize two dictionaries. For each window in A, keep a dictionary of character counts, and compare it to B's dictionary (for an O(N) comparison rather than an O(N log N) via sorting). After setting up each initial dictionary for A and B (via utilizing helper function below for adding), we can then run through all valid windows and check at each step whether the two dictionaries have valid anagrams (and, if so, add to result). We can check for anagrams, and then keep sliding the window within A.


```
def total_anagrams(A, B):
    n, k = len(A), len(B)
    if n < k:
        return []

    def is_anagram(d_a, d_b):
        for x in d_b:
            if x not in d_a: # key doesn't exist
                return False
            if d_b[x] != d_a[x]: # count doesn't match
                return False
        return True

    def add(char, d): # tracking char freq
        if char in d:
            d[char] += 1
        else:
            d[char] = 1
        return d

    d_a, d_b = {}, {}
    start, res = 0, []
    for i in range(k):
        d_a = add(A[i], d_a) # set up dictionary for A
        d_b = add(B[i], d_b) # set up dictionary for B

    for i in range(k, n+1):
        if is_anagram(d_a, d_b):
            res.append(start)
        if i < n:
            d_a = add(A[i], d_a) # add next char of window to dict
        d_a[A[start]] -= 1 # remove leftmost char of window from dict
        start += 1
    return res
```

The time complexity is O(NK), since the anagram comparison is O(K) and there are O(N) windows (assuming $N >> K$). The space complexity is O(N) to support the resulting list and dictionaries.

Solution #9.13

How do we know when two intervals overlap? First, sort all intervals by start time, and then by end time. Two intervals overlap when the ending time of an interval is larger than the start time of the next interval. Therefore, we want to iterate over the intervals and increment a count whenever two intervals overlap.

To keep track of the current and the previous interval, we create two pointers called low and high. Let low point to the interval with the smaller start time. At each iteration, we can detect when the

The space complexity is O(N) since there are N elements in the set, and the time complexity is O(N^2) since there are O(N^2) edges to traverse in the adjacency matrix.

Solution #9.16

To remove the k-th element from the back, we can create a count helper function to get the ordering of the nodes, and then take the k-th last node and make sure the previous node's next value is the node $k + 1$. Along the way, we need to address some corner cases (for example, if k is the last element in the linked list).

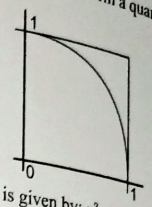
We want to avoid recalculating node counts, so we cache the ordering of the nodes in a dictionary. Thus, our count() helper function below will store the ordering of the nodes, with the order number as the key. Then, when looking for nodes $k + 1$ and $k - 1$ for editing the linked list, we can query the dictionary for those two nodes. Finally, we return the head:

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

def remove(head, k):
    def count(node, d): # helper function
        if node.next is None:
            d[1] = node # tail
            return 1
        res = count(node.next, d) # recursive call
        d[1+res] = node # store res in dictionary
        return 1 + res
    d = {}
    n = count(head, d) # get counts
    if k == len(d): # tail case
        if len(d) == 1:
            return None
        head.next = None
        return d[k-1]

    prev_node = d[k+1] # get prev node
    if k == 1:
        next_node = None
    else:
        next_node = d[k-1] # get next node

    prev_node.next = next_node # set next node
    return head
```

The time complexity is O(N) since the count() function will visit every linked list node once, and the remaining operations are constant time. The space complexity is O(N) to accommodate the dictionary that contains all the input nodes.

Solution #9.17

To estimate π, start by imagining a unit square and choosing a random pair of coordinates (x, y) inside it. Within the unit square, we can also form a quarter of a circle.



For reference, the equation for a circle is given by: $x^2 + y^2 = r^2$

Therefore, we can sample random (x, y) pairs and verify if it falls within the unit circle (if $x^2 + y^2 <= 1$). Then, we can count the resulting number of points inside the circle; when divided by the total number of points, this yields the proportion of points within the quarter circle:

$$\frac{1}{4}\pi r^2 = \frac{\pi}{4}$$

Therefore, π should be 4 times the resulting proportion of points. The time complexity is O(N), where N is the number of iterations, due to the for-loop. The space complexity is O(1) since only a fixed number of values is looked at per iteration.

```
import random
import math

count = 0 # num points inside quarter circle
n = 10000000 # iterations

for i in range(0, n):
    x_sq = random.random()**2
    y_sq = random.random()**2
    if math.sqrt(x_sq + y_sq) < 1.0: # check if inside circle
        count += 1

pi = (float(count) / n) * 4 # accounts for quarter circle
return pi
```

Solution #9.18

Note that an invalid string has more left or right parentheses than its respective counterpart. To keep track of parentheses, we can use a stack and push to it when encountering a left parenthesis, and pop from it when encountering a right parenthesis. Note that for a valid string, every right parentheses must be matched by the latest (according to the stack) left parentheses. Therefore, when we encounter a right parentheses, we must set the valid left and right parentheses accordingly. If we iterate over a letter, then we use it in the result. A temporary array can be used to track the valid letters and parentheses:

```
def splitParen(s):
    stk = [] # stack
    res = [''] * len(s) # chars to return

    for index, val in enumerate(s): # index and value of each char
        if val == '(':
            stk.append(index) # push index of left paren
        elif val == ')':
            if stk:
                latest = stk.pop() # pop the latest corresponding left paren
                res[latest] = s[latest] # or '('
                res[index] = val # or ')'
        else:
            res[index] = val
    return ''.join(res)
```

The space complexity is O(*N*) due to the temporary array, and the time complexity is O(*N*) since there is one scan through all of the characters in the string.

Solution #9.19

We can generate permutations in a recursive manner as follows: start with an empty array of results. For the base case, note that if the length of the input is 1, we just return an array of that number. Then, for every element in the input array, we make a recursive call to the subarray with all elements except the current element. Remember that each recursive call returns a list of permutations. Therefore, while iterating over all elements, in each recursive call take the results and add the element to each list that is returned. At the end we return that array of results:

```
def permute(nums):
    n = len(nums)
    res = [] # store results
    if n <= 1:
        return [nums]
    else:
        for i in range(n): # this is the element E
            # recurse on previous combos
            for combo in permute(nums[:i] + nums[i+1:]):
                res.append([nums[i]] + combo)
    return res
```

The time complexity is O(*N*!) since there are *N*! permutations being generated in the recursive call, and the space complexity is O(*N***N*!) since each of the *N*! calls uses O(*N*) space.

Solution #9.20

Let *N* be the sum of the weights. In the example, *N* would be 5 + 10 + 15 + 20 = 50. A basic way to sample from a fixed number of categories would be to generate a list of size *N* and create list elements proportional to their weights as follows (using the example):


```
import random
w_a, w_b, w_c, w_d = 5, 10, 15, 20
l = ['A'] * w_a + ['B'] * w_b + ['C'] * w_c + ['D'] * w_d
return(random.choice(l))
```

However, this solution is not optimal because the space usage here is O(*N*) since we store the number of elements according to sum of the weights. If we wanted to do this more generally, keeping space usage in mind, we can do the following:

1. Calculate the cumulative sum of weights.
2. Choose a random number *k* between 0 and the sum of weights.
3. Assign *k* the corresponding category where the cumulative sum is above *k*.

In the example below, we use the weights [5, 10, 15, 20] to create a list of cumulative sums: [5, 15, 30, 50]. Then, we choose a random number between 1 and the total weight (50, in this case) and use a modified binary search to look for that number in the list of cumulative sums. That modified binary search will return the closest corresponding index to the random number, and we can return the category label associated with that index.

```
def binary_search(a, k):
    lo, hi = 0, len(a) - 1
    best = lo
    while lo <= hi:
        mid = lo + (hi - lo) // 2
        if a[mid] < k:
            lo = mid + 1
        elif a[mid] > k:
            hi = mid - 1
        else:
            best = mid
            break
        if a[mid] - k > 0:
            best = mid
    return best

import random
categories = ['A', 'B', 'C', 'D']
weights = [5, 10, 15, 20] # list of weights
# cumulative sum [5, 15, 30, 50]
cum_sum = [sum(weights[:i]) for i in range(1, len(weights) + 1)]
k = random.randrange(cum_sum[-1]) # choose randomly in range (0, total sum)
i = binary_search(cum_sum, k) # binary search for k
return categories[i]
```

If there are K categories and the total sum of weights is N, where $N >> K$, then this method uses $O(K)$ space and has $O(K)$ runtime (since the binary search part is $O(\log(K)) < O(K)$), whereas the first method uses $O(N)$ space and $O(N)$ time to create the full list.

Solution #9.21

The brute-force solution to find the maximum common subarray would be to iterate over all possible subarrays for each input and compare the subarrays to verify if they match. However, this takes $O(N^5)$ time, since there are N^2 subarrays for both arrays, and there is an $O(N)$ check to compare any two subarrays. It is inefficient because it doesn't exploit the overlapping subproblems. Specifically, once you have a smaller common array, you can see if the next character between the two arrays is the same, which extends that common subarray by one.

Due to these overlapping subproblems, we can utilize dynamic programming: say that dp[i][j] denotes the length of the longest common subarray for a[:i] and b[:j], where a and b are the two arrays. If the i-th character of a and j-th character of b match (a[i-1] = b[j-1]), then we can extend the common subarray by one (dp[i][j] = 1 + dp[i-1][j-1]). Therefore, we can iterate over all relevant (i, j), where i ranges from 1 to the length of a, and j ranges from 1 to the length of b, and update dp[i][j] within each iteration to keep track of the maximum value seen:

```
def longest_common(a, b):
    m = len(a)
    n = len(b)

    dp = [[0 for i in range(n+1)] for j in range(m+1)] # setup dp

    max_val = 0
    for i in range(1, m+1):
        for j in range(1, n+1):
            if a[i-1] == b[j-1]:
                dp[i][j] = 1 + dp[i-1][j-1] # update dp[i][j]
                max_val = max(max_val, dp[i][j]) # keep track of max
    return max_val
```

The time complexity is $O(MN)$, where M is the length of a and N is the length of b since, in the worst case, we will fill out every cell accordingly in a bottom-up manner. The space complexity is $O(MN)$ as well, to accommodate the 2d-array of cached results.

Solution #9.22

The brute-force method to find the maximum increasing subsequence sum (MISS) would be to examine all possible subsequences (checking if it's increasing) and then calculate the sums of each to compare. Since there are N^2 possible subsequences (each of which takes $O(N)$ time to sum/verify it's increasing), this brute-force method takes $O(N^3)$ to calculate the MISS. However, this is inefficient because it doesn't take advantage of the overlapping subproblems: if you have an existing best MISS so far, and if we encounter a new element larger than the current maximum of the MISS, we can extend that MISS by that element. As such, we can use dynamic programming to store previous MISS values.

To be more precise, say we have an array tracking the max increasing subsequence sum (MISS) up to index i of the array. We can solve for that index by looking over all previous indices j (each with the

MISS up to index $j < i$). We know that we have a new MISS if the current element and the previous MISS sum up to more than the current MISS.

```
def max_subseq_sum(arr):
    n = len(arr)
    res = [0 for x in range(n)] # store results
    for a in range(n):
        res[a] = arr[a]
    for i in range(1, n):
        for j in range(i):
            if arr[j] < arr[i] and res[i] < res[j] + arr[i]: # extend the MISS
                res[i] = res[j] + arr[i] # add incremental element
    return max(res)
```

The time complexity is $O(N^2)$ due to the doubly-nested for-loops, and the space complexity is $O(N)$ to store the array of MISS results.

Solution #9.23

At first, finding the smallest number of perfect squares that sum up to n may look like an application of a greedy algorithm, where you try to find the biggest square less than the value. However, consider $4^2 + 5^2 = 16 + 25 = 41$ as an example. If you used $6^2 = 36$, you'd get a non-optimal result of $36 + 4 + 1$ rather than $25 + 16$.

As such, to get the correct answer for 41, check the smallest number of perfect squares for 41-1, 41-4, 41-9, etc. for all squares up to 41. There are overlapping subproblems, so we can use dynamic programming to store sub-results. Let res[i] represent the smallest number of perfect squares that sum to n. Our recursive step is:

```
res(i) = min(res(i), res(i-j^2) + 1)
```

where j is an integer less than or equal to the square root of i. This is because either the current value is minimal, or we can subtract off a square and yield a smaller number of squares to add up to the remaining value. We can solve the subproblems in a bottom-up manner:

```
def square_count(n):
    res = [x for x in range(n+1)] # store results
    for i in range(2, n+1):
        for j in range(1, int(i ** 0.5) + 1):
            res[i] = min(res[i], res[i - j ** 2] + 1)
    return res[n]
```

The time complexity is $O(N^2)$ due to the doubly-nested for-loops, and the space complexity is $O(N)$ to store the array of results.

Solution #9.24

To find all combinations, we can use a method called backtracking, which builds upon a solution set according to some constraints and abandons paths that cannot lead to a valid solution. To get all combinations with k elements, first, pick an initial element from the set of existing numbers, then concatenate that element with all other possible combinations with k-1 elements produced so far, which occurs in a recursive call.

For instance, assume we have a backtrack() helper function that takes in the following parameters: n, k, *res* (which is the resulting list of lists that we can return eventually), *combo* (any list of elements that is a candidate combination), *num* (the number of elements in the list) and *start* (where we start within a list in generating combinations).

Then the logic is as follows: anytime *num* is equal to k, append the *combo* list to *res*. If *start* is beyond n or *num* is beyond k, we can return early. Otherwise, going from *start* until n, we can do the following:

1. Add the current element to *combo*.
2. Recursively call backtrack() on the existing version of a *combo* with updated *num* and *start* parameters.
3. Remove that element from *combo*.

Step #3 is essential, because further combinations generated do not always include that element from Step #1. In the end, we return the result we obtain from calling backtrack():

```
def combos(n, k):
    def backtrack(n, k, res, combo, num, start):
        if num == k:
            res.append(list(combo)) # add to result
        if start > n or num >= k:
            return
        for i in range(start, n+1): # iterate over every element
            combo.append(i)
            backtrack(n, k, res, combo, num+1, i+1) # recurse
            combo.remove(i)
    res = []
    backtrack(n, k, res, [], 0, 1)
    return res
```

The time complexity is $O(\binom{N}{k})$, and there is no way to circumvent it since that is the number of returned combinations. This also holds true for the space complexity.

Solution #9.25

The brute-force way is to check every possible substring and see if it's well-formed. A more optimal way is to incrementally grow the substring, and use a stack to make sure parentheses are balanced and the string is well formed. For every left parenthesis, push its index within the string onto the stack. For every right parenthesis, pop the topmost element and then calculate the length based on the current index versus the index of the first leftmost parentheses of the current valid substring (the

top element of the stack). Hence, the difference between the current index and that top element in the stack is potentially the longest length of a well-formed string seen so far. In the case where the stack is empty after the pop, we can add the current index to the stack. This occurs when you have right parentheses before left parentheses.

To address the case where the first character is a right parentheses, we can add an initial dummy value onto the stack. This dummy value needs to be –1 for the following reason: we would give the correct length of the substring when popping the first '(' at matching the corresponding ')' at stack[i] as i + 1, only if the initial dummy value is –1.

```
def longest_parens(s):
    stack = []
    longest = 0
    n = len(s)
    stack.append(-1) # initial dummy value
    for i in range(n):
        if s[i] == '(':
            stack.append(i) # append index
        else:
            stack.pop()
            if len(stack) == 0:
                stack.append(i)
            else:
                longest = max(longest, i - stack[-1]) # get length
    return longest
```

The time complexity and space complexity are both O(N), since the stack goes through every element one time.

Solution #9.26

To find the length of the longest path of increasing integers, first, consider a single element in the matrix. Presume that we have found the longest path of increasing integers for all other paths not involving that number. When evaluating the element, we know that the length of the new longest path must be 1 plus the maximum length of the longest paths of its four neighbors. To be specific, for any given indices (i, j), there are up to 4 potential neighboring indices, of which the longest increasing path from (i, j) is 1 plus the max of the longest increasing path from its four neighbors, if the element is larger than its neighbor.

This structure lends itself well to dynamic programming, and as such, we can keep track of a max length and a table (to serve as a cache) with indices (i, j), which tracks the length of the longest increasing path starting at index i, j of the matrix. To calculate this max, we can utilize depth-first-search (DFS), where we recurse on the 4 neighboring elements. For each DFS, we also need to keep track of a previous max element (since the path needs to be increasing — so if the current value at i, j is less than that value, we can just return 0). To start the algorithm, we can instantiate the max to be -∞ and start with an empty dictionary for the table cache, and then run DFS for each i, j:

```
def longest_increasing_path(mx):
    table = {} # cache
    if len(mx) == 0:
        return 0
    m = len(mx)
    n = len(mx[0])

    def dfs(i, j, prev, table): # DFS helper
        # edge case
        if(i < 0 or i >= len(mx) or j < 0 or j >= len(mx[0]) or mx[i][j] <= prev):
            return 0
        if (i, j) in table:
            return table[(i, j)] # get cached value
        curr_len = 1 + max(dfs(i-1, j, mx[i][j], table),
                           dfs(i+1, j, mx[i][j], table),
                           dfs(i, j-1, mx[i][j], table),
                           dfs(i, j+1, mx[i][j], table))
        table[(i, j)] = curr_len # get max
        return curr_len

    for i in range(m): # call DFS from each i,j
        for j in range(n):
            dfs(i, j, -float("inf"), table) # set initial max and table

    return max(table.values())
```

The time complexity is O(MN), since in the worst case, the DFS will visit every cell within the matrix. The space complexity is O(MN) as well to accommodate the table cache.

Solution #9.27

The brute force method to find consecutive integers that sum to n is to start creating lists from 1 to n, and keep adding consecutive integers to each list until they equal n or go over. However, this solution has a runtime of O(N^2) and is inefficient. Here is that code:

```
def consecutive_sum(n):
    num = 0
    for i in range(1, n+1):
        total = 0
        while total < n:
            total = total + i
            i += 1
        if total == n:
            num += 1
    return num
```


Due to the structure of this problem, we can look for some properties that valid solutions follow, so more intelligently create candidate lists rather than brute-force generating all of them. Let's reframe the problem in a mathematical way. Say we start at some number k, and go up to some number $k + m$, so we have a sequence of $m + 1$ terms. Note that:

$$k + (k + 1) + \ldots + k + m - 1 = (2k + m)\left(\frac{m+1}{2}\right)$$

Now, if we set this to n, we have the following: $(2k + m)\left(\frac{m+1}{2}\right) = n$

Solving for k in terms of m and n, we get: $2k = \frac{2n}{m+1} - m$

Since k needs to be an integer for a valid solution, we can check whether the right-hand side satisfies two conditions:

1. $2n$ is divisible by $m + 1$ (otherwise the right-hand side is not an integer).
2. The right-hand side is divisible by 2 (i.e., it is an even number, since $2k$ is even).

Note that for the right-hand side we have: $\frac{2n}{m+1} - m$

We do not need to check all values of m since we need k to be a positive integer. Thus, to find the bounds on m, we take the above expression and find when it is greater than 0:

$$\frac{2n}{m+1} - m > 0$$

$$\frac{2n}{m+1} > m$$

$$2n > m(m+1) = m^2 + m$$

Since $m^2 + m > m^2$, we have:

$$2n > m^2$$

$$\sqrt{2n} > m$$

As such, we only need to iterate m from 1 to $\sqrt{2n}$:

```
import math

def consecutive_sum(n):
    upper_limit = int(math.sqrt(2*n))
    num = 0
    for m in range(upper_limit):
        if (2*n) % (m+1) == 0 and (2*n/(m+1)-m) % 2 == 0:
            num += 1
    return num
```

The runtime is O(N), and the space complexity is O(1).

Solution #9.28

The brute-force way to find the median of a continuous stream of elements is to store all seen elements in an array and then find the median there. However, this is inefficient in the find operation — it is generally O(N log N), since a sort is needed.

A more efficient way is to keep track of two heaps: a max-heap to store the smaller half of numbers, and a min-heap to store the larger half of numbers. The intuition is that the median will always be either the largest value of the max-heap, the smallest value of the min-heap, or the average of the two (try some examples out on paper). The question, then, is how to maintain the balance, since this approach only works if both elements contain about half the elements (otherwise, the median would be deep inside one of the two heaps if the sizes were skewed). To handle this, simply check the heap size when adding elements. If the size of one heap is larger than the other, add an element from one heap to another to balance it out.

Below, we use the heapq library for a priority queue implementation and use the negatives of values (since by default the priority queue implementation in Python weights elements by their value from smallest to largest):

```
import heapq

class MedianFinder(object):
    def __init__(self):
        self.min_heap = []
        self.max_heap = []

    def add_num(self, num):
        heapq.heappush(self.max_heap, -num) # add negative for max heap
        heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))
        if(len(self.min_heap) > len(self.max_heap)):
            # balance heaps
            heapq.heappush(self.max_heap, -heapq.heappop(self.min_heap))

    def find_median(self):
        if len(self.max_heap) > len(self.min_heap): # return max heap value
            return 1.0 * -self.max_heap[0]
        else: # return average of two
            return 1.0 * (self.min_heap[0] - self.max_heap[0]) / 2
```

The runtime of add_num() is O(log(N)) because each heap insertion (up to 3) takes O(log(N)) time. The runtime of find_median() is O(1). Therefore, the overall procedure takes O(log N). The space complexity is O(N), since N elements are stored among the heaps.

Solution #9.29

Instead of checking whether the whole input string matches the regex, let's start by breaking down the problem and considering just one character at a time. We can start from the back of both the regex and the input strings. If both are regular 'a-z' characters and do not match, we can immediately return false. Otherwise, check whether the remaining regex string and input strings match without those two ending characters.

Now, say we see that the last character of the regex string is '?'. Because '?' matches any single character, we can check if the regex without the '?" matches the rest of the input string without the most recent character. Here we begin to see the recursive nature of the problem.

If we see a '*', then there are two cases. The first is that '*' could represent an arbitrary number of characters. In this case, its behavior is similar to '?' - for any one character, we just recursively check the rest of the input minus that one character with the same regex string. On the other hand, '*' could have the behavior of an empty string, in which case check the rest of the input with the same regex string without the '*'.

Note the overlapping subproblems: if we find that a part of the regex matches a substring of the input, we can determine whether another character added onto the regex ('?', '*', or character) matches the input substring with another character added on to it.

In particular, say we have inputs s (string) and r (regex). We can keep track of a table that stores Booleans for the function output. In this table, indices (i,j) will store the function result of checking whether or not the substring of the input up to character i matches that of the regex substring up to character j (whether s[:i] and r[:j] match). Thus, after this table is filled, we want to check the index $i = -1, j = -1$ (the last element of the table) to confirm whether the full regex string matches the full input string.

We can start by instantiating the dp matrix to be false for all indices. Then, for the value at dp[0][0], we can set it to true, since it's trivially true that an empty regex string matches an empty string. In addition, since we know that any regex substrings with just '*' characters matches any substring, we can set dp[0][j] to be 0 as long as the regex contains '*' characters up to index j.

Summarizing the above, our recurrence relations are as follows:

1. If r[j-1] is '?' or matches s[i-1], then we set dp[i][j] = dp[i-1][j-1] (recursive call where we check everything before the current last character in both the regex and input string).
2. If r[j-1] is '*', then the result is the logical OR of the following 3 sub-cases:
   a) dp[i-1][j-1] (recursive call as if the '*' functioned as a ?).
   b) dp[i][j-1] (recursive call where the '*' matches the current last character and will continue from there); or
   c) dp[i-1][j] (recursive call where the '*' matches an empty string).

The time complexity is O(MN), where M is the length of the first string, and N is the length of the second string, since, in the worst case, we will fill out every cell accordingly in a bottom-up manner. The space complexity is O(MN) as well, to accommodate the 2-d array.

If neither of the prior cases applies, then we ignore changing dp[i][j] since the entry will be false (and we can set it to that by default). Finally, we return the last indices for the string and regex within dp, i.e. dp[-1][-1]:

```
def wildcard_match(string, regex):
    m = len(string)
    n = len(regex)
    dp = [[False for _ in range(n+1)] for _ in range(m+1)] # initialization
    dp[0][0] = True
    for j in range(1, n+1): # filling in Trues for any *s
        if regex[j-1] != "*":
            break
        dp[0][j] = True
```

```
    for i in range(1, m+1):
        for j in range(1, n+1):
            if regex[j-1] == '?' or regex[j-1] == string[i-1]: # case 1
                dp[i][j] = dp[i-1][j-1]
            elif regex[j-1] == "*": # case 2
                dp[i][j] = dp[i-1][j-1] or dp[i][j-1] or dp[i-1][j]
    return dp[-1][-1]
```

Solution #9.30

To find the optimal location of the new fire station, minimize the following cost function, where (x_c, y_c) represents the coordinates of the new fire station, and there are n houses each with coordinates (x_i, y_i):

$$L(x, y) = \sum_{i=1}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}$$

One possible method to solve this optimization problem is through grid search, where you manually sweep through all possible locations. Our solution, however, will focus on gradient descent. Recall that gradient descent allows you to find a local minimum for a convex function by taking steps in the opposite direction of the gradient of the function. Note that because there is no other local minimum other than the global minimum, and the sum of squares function above is convex, gradient descent leads us to an optimal result. For each iteration of gradient descent, as we refine the current location and get closer to the optimal location, we decay the learning rate and take smaller and smaller steps until we arrive at the solution.

Before we can implement gradient descent in code, we need to calculate the partial derivatives of the sum of squares function for both x and y. We get the following for the partial with respect to x:

$$\frac{\partial L(x,y)}{\partial x} = \frac{(x_c - x_i)}{\sum_{i=1}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}}$$

And an analogous partial with respect to y: $\frac{\partial L(x,y)}{\partial y} = \frac{(y_c - y_i)}{\sum_{i=1}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}}$

To control the rate of gradient descent, we initialize the following hyperparameters:

1. Initial learning rate: the starting value for the learning rate. The learning rate is used to scale the size of the moves we take in the opposite direction of the gradient.
2. Rate of decay: used to decrease the learning rate. We make the learning rate progressively smaller to converge upon an eventual answer.
3. Terminal rate: the stopping condition. When the learning rate becomes smaller than the terminal rate, we can exit the gradient descent process.
4. Damping factor: constant that helps with numerical stability. During gradient descent there are always oscillations back and forth around a range of values — by applying a damping factor to reduce the magnitude of step sizes, there is a better chance of a smooth convergence.

To implement gradient descent, we begin with the mathematical centroid of all the coordinates as the starting estimate for the optimal location (x_c, y_c). Then, we can set the hyperparameters defined above and update the values of (x_c, y_c) iteratively until the terminal condition is hit and the estimate for the optimal location is reached:


```
import math

def get_optimal_location(coords):
    def partialX(x_c, y_c): # partial with respect to x
        return sum(((x_c-x_i)/math.sqrt((x_c-x_i)**2 +
                    (y_c-y_i)**2) if x_c != x_i else 0) for x_i, y_i in coords)

    def partialY(x_c, y_c): # partial with respect to y
        return sum(((y_c-y_i)/math.sqrt((x_c-x_i)**2 +
                    (y_c-y_i)**2) if y_c != y_i else 0) for x_i, y_i in coords)

    rate = 1 # initial learning rate
    rate_decay = 0.99 # rate of decay
    terminal_rate = 1e-8 # 1e-8 # terminal rate (stop condition)
    damping_factor = 0.7

    l = len(coords)
    x_c = sum(x for x, y in coords) / l # centroid x
    y_c = sum(y for x, y in coords) / l # centroid y

    dx = 0
    dy = 0

    while rate > terminal_rate:
        dx = partialX(x_c, y_c) + damping_factor * dx
        dy = partialY(x_c, y_c) + damping_factor * dy
        x_c = x_c - rate * dx # update coordinates
        y_c = y_c - rate * dy
        rate = rate * rate_decay # update learning rate
    return (x_c, y_c)
```

To assess the runtime complexity, assume that the minimum values of (x_i, y_i) are given by (x_{min}, y_{min}) and the maximum values are given by: (x_{max}, y_{max}). Let $x_d = x_{max} - x_{min}$ and $y_d = y_{max} - y_{min}$. Then, we know the upper bound, in terms of points tested, is $x_d * y_d$. Since calculating the partial derivatives for x and y takes $O(N)$ each, then the time complexity should be $O(N * x_d * y_d)$. The space complexity is $O(N)$ since, during the partial derivatives calculating, we are summing over N elements in a temporary list.

The algorithm's runtime is also dependent on the hyperparameters that control the gradient descent. Here is how the runtime is affected by each:

1. Initial learning rate: this is often set at 1. The higher it is, the longer the runtime, since the initial learning rate needs to converge to the terminal rate.
2. The rate of decay: this is set between 0 and 1. The higher it is, the faster the runtime, since a larger rate of decay means reaching the terminal learning rate faster.

```
for i in range(1, m+1):
    for j in range(1, n+1):
        if regex[j-1] == '?' or regex[j-1] == string[i-1]: # case 1
            dp[i][j] = dp[i-1][j-1]
        elif regex[j-1] == "*": # case 2
            dp[i][j] = dp[i-1][j-1] or dp[i][j-1] or dp[i-1][j]
return dp[-1][-1]
```

Solution #9.30

To find the optimal location of the new fire station, minimize the following cost function, where (x_c, y_c) represents the coordinates of the new fire station, and there are n houses each with coordinates (x_i, y_i):

$$L(x, y) = \sum_{i=1}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}$$

One possible method to solve this optimization problem is through grid search, where you manually sweep through all possible locations. Our solution, however, will focus on gradient descent. Recall that gradient descent allows you to find a local minimum for a convex function by taking steps in the opposite direction of the gradient of the function. Note that because there is no other local minimum other than the global minimum, and the sum of squares function above is convex, gradient descent leads us to an optimal result. For each iteration of gradient descent, as we refine the current location and get closer to the optimal location, we decay the learning rate and take smaller and smaller steps until we arrive at the solution.

Before we can implement gradient descent in code, we need to calculate the partial derivatives of the sum of squares function for both x and y. We get the following for the partial with respect to x:

$$\frac{\partial L(x, y)}{\partial x} = \frac{(x_c - x_i)}{\sum_{i=1}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}}$$

And an analogous partial with respect to y: $\frac{\partial L(x, y)}{\partial y} = \frac{(y_c - y_i)}{\sum_{i=1}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}}$

To control the rate of gradient descent, we initialize the following hyperparameters:

1. Initial learning rate: the starting value for the learning rate. The learning rate is used to scale the size of the moves we take in the opposite direction of the gradient.
2. Rate of decay: used to decrease the learning rate. We make the learning rate progressively smaller to converge upon an eventual answer.
3. Terminal rate: the stopping condition. When the learning rate becomes smaller than the terminal rate, we can exit the gradient descent process.
4. Damping factor: constant that helps with numerical stability. During gradient descent there are always oscillations back and forth around a range of values — by applying a damping factor to reduce the magnitude of step sizes, there is a better chance of a smooth convergence.

To implement gradient descent, we begin with the mathematical centroid of all the coordinates as the starting estimate for the optimal location (x_c, y_c). Then, we can set the hyperparameters defined above and update the values of (x_c, y_c) iteratively until the terminal condition is hit and the estimate for the optimal location is reached:


```
import math

def get_optimal_location(coords):
    def partialX(x_c, y_c): # partial with respect to x
        return sum(((x_c-x_i)/math.sqrt((x_c-x_i)**2 +
                    (y_c-y_i)**2) if x_c != x_i else 0) for x_i, y_i in coords)

    def partialY(x_c, y_c): # partial with respect to y
        return sum(((y_c-y_i)/math.sqrt((x_c-x_i)**2 +
                    (y_c-y_i)**2) if y_c != y_i else 0) for x_i, y_i in coords)

    rate = 1 # initial learning rate
    rate_decay = 0.99 # rate of decay
    terminal_rate = 1e-8 # 1e-8 # terminal rate (stop condition)
    damping_factor = 0.7

    l = len(coords)
    x_c = sum(x for x, y in coords) / l # centroid x
    y_c = sum(y for x, y in coords) / l # centroid y

    dx = 0
    dy = 0

    while rate > terminal_rate:
        dx = partialX(x_c, y_c) + damping_factor * dx
        dy = partialY(x_c, y_c) + damping_factor * dy
        x_c = x_c - rate * dx # update coordinates
        y_c = y_c - rate * dy
        rate = rate * rate_decay # update learning rate
    return (x_c, y_c)
```

To assess the runtime complexity, assume that the minimum values of (x_i, y_i) are given by (x_{min}, y_{min}) and the maximum values are given by: (x_{max}, y_{max}). Let $x_d = x_{max} - x_{min}$ and $y_d = y_{max} - y_{min}$. Then, we know the upper bound, in terms of points tested, is $x_d * y_d$. Since calculating the partial derivatives for x and y takes $O(N)$ each, then the time complexity should be $O(N * x_d * y_d)$. The space complexity is $O(N)$ since, during the partial derivatives calculating, we are summing over N elements in a temporary list.

The algorithm's runtime is also dependent on the hyperparameters that control the gradient descent. Here is how the runtime is affected by each:

1. Initial learning rate: this is often set at 1. The higher it is, the longer the runtime, since the initial learning rate needs to converge to the terminal rate.
2. The rate of decay: this is set between 0 and 1. The higher it is, the faster the runtime, since a larger rate of decay means reaching the terminal learning rate faster.

3. The terminal rate: the higher it is, the faster the runtime, since convergence conditions are reached more quickly.
4. The damping factor: set between 0 and 1. The smaller the value, the faster the runtime, since there is less damping; hence, larger moves due to the gradient.