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).

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. Done +9

Arrays

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].

Arrays

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.

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.

$$\begin{bmatrix} 6 & 8 & 11 \end{bmatrix}$$

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.

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).

a function to find the index of any peak elements. For example, for [5, 3, 2, 4, 1], you should return either 1 or 3 because the values at those indexes, 5 and 4, are both peak elements.

Array

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

Solution #9.8

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

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
```

since A is friends with B, B is friends with D, and D is friends with E.

String 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.

Array

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.

Array to Strings

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"]].

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
```

intervals overlap by checking the low pointer of the next interval versus the higher pointer of the current one.

In case intervals need to be merged (when one interval is completely within another), we set the previous low pointer value to be the current high pointer value. In this way, no intervals are deleted, but in the next iteration we need to compare against the current high interval's begin and end time:

```
def interval_removal(interval_list):
    if len(interval_list) == 0:
        return 0
    intervals = sorted(interval_list, key=lambda k: (k[0], k[1])) # sort
    res, low, count = 0, 0, 0
    for high in range(1, len(intervals)):
        if intervals[low][1] > intervals[high][0]: # overlap
            count += 1
        if not intervals[high][0] < intervals[low][1] < intervals[high][1]:
            low = high # merge
    return count
```

The time complexity is O(*N*) since there is just one for-loop. The space complexity is O(1) since there is no extra space being used.

Solution #9.14

We can use a helper function to identify anagrams, and then identify the strings that have the same composition. For the helper function that identifies anagrams, we can create a character-frequency map. As an example, for "abc" we can have the following dictionary: {"a": 1, "b": 1, "c": 1}. To compare anagram compositions among different strings, we will need to uniquely identify the maps: for example, if for "abc" we have the map: {"a": 1, "b": 1, "c": 1} and for "cab" we have: {"c": 1, "a": 1, "b": 1} then we want to make sure both correspond to the same anagram grouping. To do this, we can sort each dictionary in <key, value> format and use the string representation. Therefore, we simply keep a final dictionary whereby that anagram string is the key, and the value is the list of strings that fall under that anagram grouping:

```
def anagram_group(str_list):
 cfd_str_list = {} # {anagram_composition_string : {strings of common anagram}}
 for s in str_list:
   cfd = {} # char freq dict of s {character : frequency}
   for c in s:
     if c in cfd:
       cfd[c] += 1
     else:
       cfd[c] = 1
   # got sorted char freq dict of s
   cfd = dict(sorted(cfd.items()))
# flatten cfd to cfd_str as single string
```

```
cfd_str = ''.join('{}{}'.format(k, v)
                  for k, v in cfd.items()) # format


# anagrams will produce identical cfd_str
  if cfd_str not in cfd_str_list:
    cfd_str_list[cfd_str] = [s] # add anagram
  else:
    cfd_str_list[cfd_str].append(s) # not existing anagram yet
res = []
for cfd_str in cfd_str_list:
  res.append(cfd_str_list[cfd_str])
return res
```

The time complexity is O(NK(log K)), where N is the length of the string list input and K is the maximum length of any given string within the list, since the outer for-loop will take O(N) time for each string, the helper function takes O(K) time, and sorting the dictionary takes O(K log K). The space complexity is O(NK) since there are at most N results of size K.

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).

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.

```
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.

Arrays

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].

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.