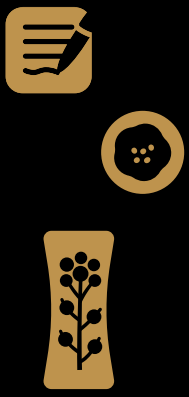
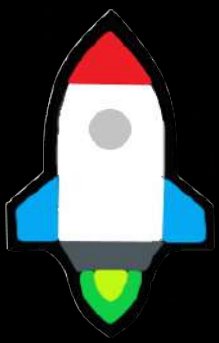


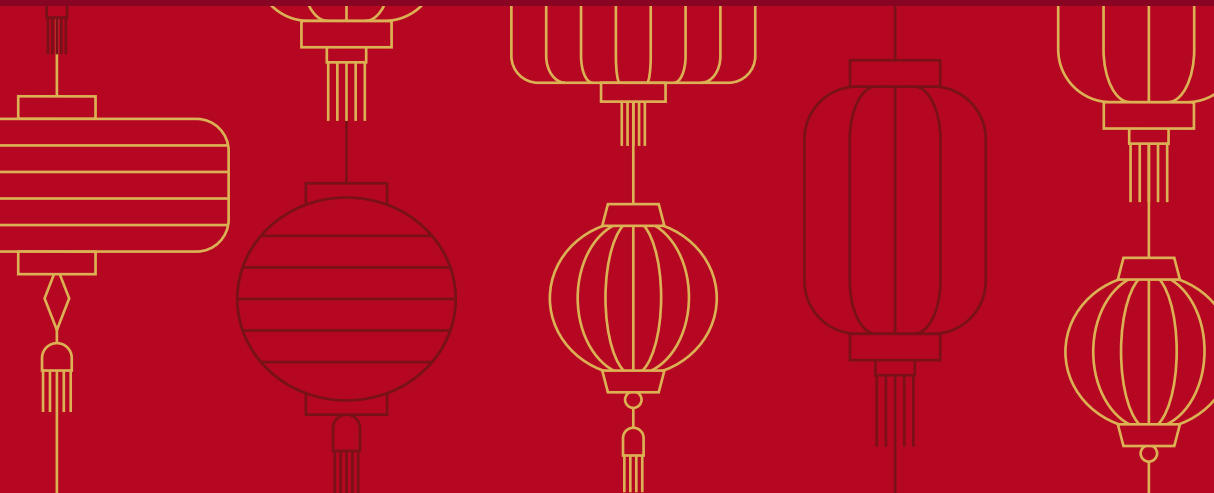
LeetCode in Python



(NeetCode ~150)



Arrays and Hashing



I) Contains Duplicate (LC 217)

Brute force - Iterate over the array for every element to check if there exists a duplicate. i.e. for A[i], iterate over A[i+1:] & return true if there's a duplicate.
Time - $O(n^2)$; Space - $O(1)$

Hashing - Create a dictionary/set by iterating over the array & only add a key if it doesn't exist, if an element in dict/set return True, else False.
Time - $O(n)$, space - $O(n)$

II) Valid Anagram (LC 242)

Brute force - Feels extremely silly to come up with brute force solns for such easy problems! Anyway, just manually go on checking for s' characters in t. While doing so, remove a character if it has been used in t, to avoid repitition.
Time - $O(n^3)$
- $O(n)$ for every character in s.
- $O(n)$ for searching for a character in t.
- $O(n)$ for removing a character in t.

Sorting - Sort both s & t. If they have the same characters, the sorted strings will be the same, i.e. sort(s) == sort(t).
Time - $O(n \log n)$, Space - $O(1)$
($O(n \log n)$ sorting)

Hashing - Create a dictionary of characters for one of the strings, say s. Such that, the keys correspond to unique characters & their values to the frequency of occurence. So, for a given string "anagram", the hash map/dictionary would look like -
{'a': 3, 'n': 1, 'g': 1, 'r': 1, 'm': 1}

Next, iterate over the other string (t). While doing so, each time a character in t is found in the dictionary subtract 1 from its frequency value. As soon as the frequency corresponding a character =0, delete it from the dictionary. However, if a character in t is not found in the dictionary, return False.
Time- O(n), Space-O(n)

III) Two sum (LC 1)

Brute force- For every element in the array, iterate over the subarray after it & check if there exists a solution.
Time- $O(n^2)$, space- O(1)

Hashing- Create a dictionary by iterating over the array & for every iteration check if the result i.e., target-num already exists in the dictionary. If it does return the values & if it doesn't add the element as numsDict[element]= index.
Time- O(n), space-O(n)

IV) Group Anagrams (LC 49)

pinda

Brute force- For every string in the array check with all the other strings for an anagram relationship. We'll have already discussed how to check for anagrams in LC242.

Time- $O(sn^2)$; space- O(ns)+O(s) → for every str, dict size = O(s). However, the dict is deleted & recreated for every str. O(ns) for o/p.

↳ for every string, create a hashmap O(s)
↳ then, iterate over the array to compare it with the others → O(sn).
so, O(sn) for every string & n such values, → $O(sn^2)$

Hashing + Sorting- Create a dictionary by iterating over the i/p array & store the keys as the sorted strings & the values as their corresponding indices in the i/p array. If the sorted string exists, append to the array

of indices, else create a new entry. Finally, output the strings corresponding to every unique key in the dictionary.

Time- O(n s log s) + O(n) ; space- O(n s) → o/p (if o/p is not included then O(1))

n sorting n s length strings.

V) Top K frequent Elements (LC347)

Brute force - Actually, with such problems, there are multiple levels of brute forcing. This was the case with the previous problem as well. So, let's take it step by step. -

1. Find the frequency → to do this we require a dictionary/hashmap. So, iterate over the array & store the frequency corresponding to every unique element.
2. Find the top K frequencies → Now, one way could be to sort the the frequency values & return the top K values. This would be the so called "BRUTE FORCE" Method.

Time- O(n) + O(n log n); space- O(n)

Track the top K values - use a max-heap & track the top K values by converting the list into a heap (using heapify) & then popping K times.

Time- O(n) + O(K log n) ; space- O(n)

Create a Bucket Array - This solution isn't the most obvious but is really effective. After creating the frequency dictionary, create an array of size len(nums)+1, initialized with empty arrays. Now, at every index of this array store elements with frequency = the index. This way, we can reverse iterate over the array & find the top K elements.

Time- O(n), space - O(n)

VI) Product of Array Except Self (LC 238)

Brute force - since the division operator is not allowed, for every element just multiply all other elements.
Time - $O(n^2)$, space - $O(n)$
↳ does not solve the problem.

$O(n)$ solution - For an element at the i^{th} index, store the pre-product i.e. product of all elements until the i^{th} index (excluding i) & the post-product of all elements after i. So, maintain 2 arrays - prefpro & postpro. Iterate over the array and incrementally compute the prefpro array. Subsequently iterate in reverse order to compute Postpro. Finally, the solution =

result[i] = prepro[i-1] x Postpro[i+1]

for edge cases, i.e. indices 0 & n-1; multiply by 1.

Time - $O(n)$; space - $O(n)$

Constant space - The question says that the o/p array is not included in the space complexity. Observe that while calculating the result values we needed both post & pre values, however the process seems kinda redundant. For Prefpro, we iterated in the correct order, while for Postpro, we iterated in reverse order. How about updating products for values at both ends. (not a sol^n that would immediately click). Keep 2 values Prefpro & Postpro & iterate over the array in the normal order.
Multiply the prefpro value to the result[i] & the postpro value to result[n-1-i]. Subsequently, update these 2 values. Have a look at the code for a better understanding & try to do a dry run with a small array.
Time - $O(n)$; space - $O(1)$; ∵ o/p is not included.

VII) Valid Sudoku (LC 36)

It's a simple if/else problem. 3 checks as suggested by the problem. For each row & each column, no repitition. & lastly, no repitition for any 3x3 grid.

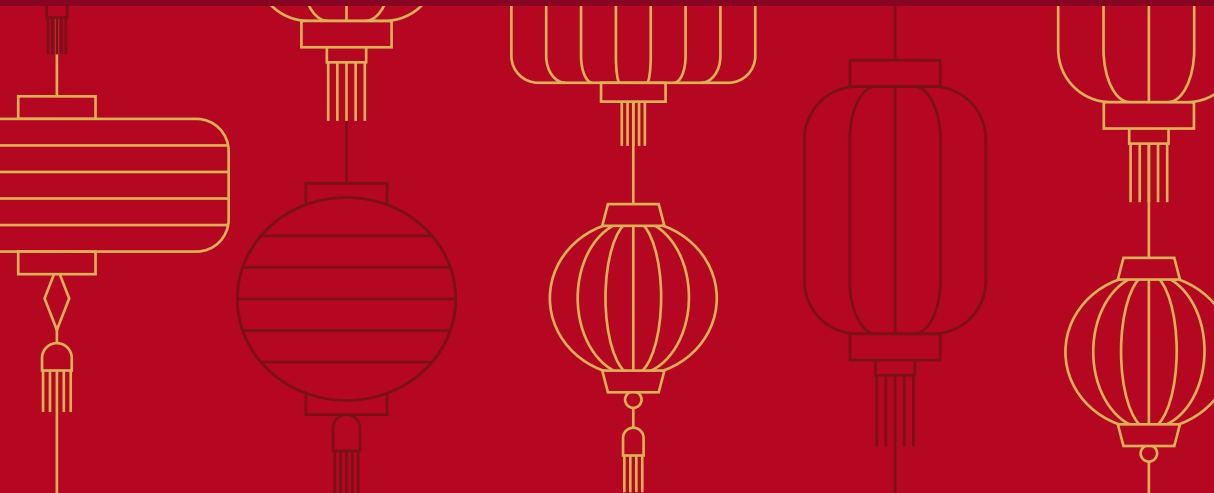
VIII) Longest Consecutive Subsequence (LC 128)

Brute force - Sort the array and find the ssq". However, sorting will take O(nlogn) which is not allowed. So, there has to be a better solution.

Hashing - Obviously xD. So, the same old trick, hash the nums away. Now, for every value, search for values immediately smaller than it, i.e., if num-1 is found, then num-2 ... & so on until they are found. When a num key is used in a sequence, update its value in the dictionary to avoid repitition. Similarly, do so for values, incrementally greater, i.e., num+1, then num+2, and so on. Finally after finding the length of the local sequence, compare it with the result until then & update it if required.

Time - O(n), space - O(n)

Two Pointers



IX) Valid Palindrome (LC125)

Brute force - The title of this section gives away the obvious solution to this problem. However, let's take it slow. First step is to transform the string, i.e., convert all characters to lower case & remove the alphanumeric characters. Next, by the "BRUTE FORCE" method, reverse the string & see if it matches with the original transformed string.
↳ .isalnum() to check for alphanumeric chars.

Time - O(n), space - O(n) ← to store the reverse string

Two Pointers - It's really simple; start by placing 2 pointers at both ends of the transformed string & check if the 2 values are equal. If they are, keep closing in, i.e. start+1 & end-1, this until start < end. Here, instead of storing the transformed string, we just iterate over the given string in a clever way.

Time - O(n); space - O(1)

X) Two Sum II - Input Array is Sorted (LC167)

Brute Force - Simple, don't care about the sorted bit, check the remnant array for every index and try to find the required number.

Time - $O(n^2)$; space - O(1)

Two Pointers - Use the fact that the array provided is sorted in a non-decreasing order. Place 2 pointers, start & end at the 2 ends. add the values at these 2 pointers & if the sum < target, increment the start pointer, elif sum > target, decrement the end pointer, & obviously if sum == target, return [start+1, end+1].

Time - O(n), space - O(1).

XI) 3Sum (LC15)

Brute Force - Back with the ugly stuff. The dumb answer is create a triple-nested for loop & check for the result required. This
Time - $O(n^3)$, space - $O(n)$ ← o/p.

Hashing - Create a double-nested for loop & then find the negative of that sum in the hashmap/dictionary/set of nums to make the 3sum = 0.
Time - $O(n^2)$, space - $\underbrace{O(n)}_{\text{o/p}} + O(n)$ ↳ besides the o/p for the set of vals.

Sorting + 2pointers - The best solution has to be a 2 pointer sol'n :D, (Name of the section). Actually, this problem is a simple combination of the normal 2sum & sorted 2sum problems. The idea is simple, sort the array & then for every element, treat the problem as a sorted 2 sum to find "-element" as the target. There are some nuances to these approaches, so take a look at the code if there are any doubts.
Time - $O(n^2)$, space - $O(n)$ ← o/p.

XII) Container with Most Water (LC11)

Brute Force - Try all combinations, ${}^nC_2$, 2 for loops, look over the remnant array following every element.
Time - $O(n^2)$, space - $O(1)$.

Two Pointers - This problem is probably the most intuitive example of the 2 pointers approach. Same old drill, initialize the 2 pointers as the 2 extremes of the height array. Observe, the capacity b/w 2 pointers = min(height[start], height[end]) × (end - start). Since we started at the extremes, it only makes sense to make the length (end - start) smaller, if we are able to improve on the min of height[start] & height[end].

Thus, we have our solution. Try to improve on the min pointer & then compare the new area & retain the max(old_area, new_area).

XIII) Trapping Rain Water (LC42)

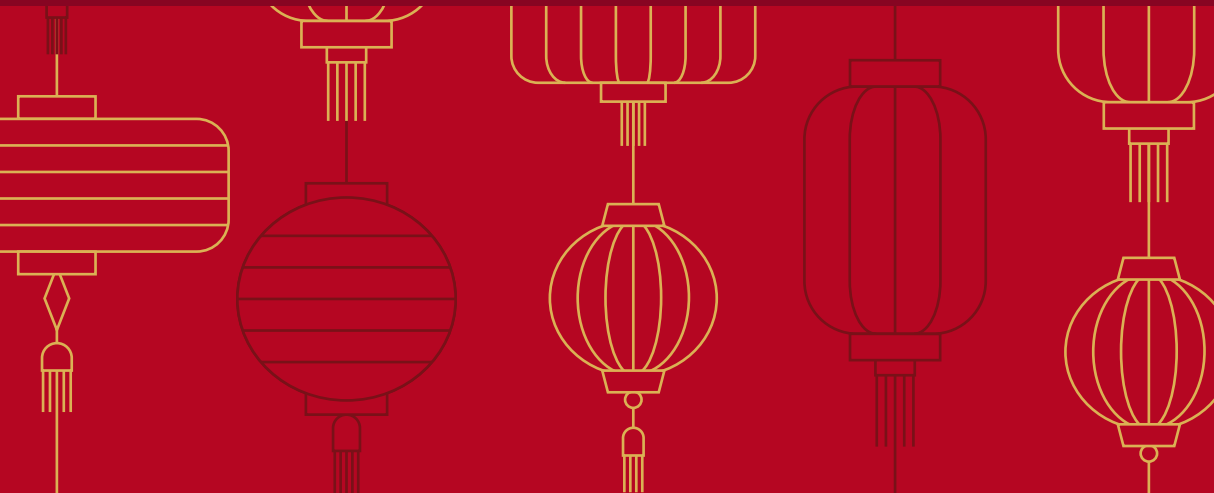
At this point, I am so used to writing Brute Force first & thinking second. For this problem however, the play is also about understanding & analysing the intricacies of the problem. So, let's get on it. There 2 questions that must be answered before moving on to the sol^n. – 1. What is the condition to trap water b/w any 2 bars?
2. How to calculate the amount of water trapped?
Pondering upon the problem will highlight a key aspect of the problem, 2 POINTERS! We must deal with 2 points / bars b/w which we can find the water trapped. The question now is how to place these pointers.
While figuring out the sol^n, I initially went slightly wrong & although my code was functional it was exceeding the time limit (cleared 320/322 test cases). So, what was the sol^n?

2 pointers 1 (Time Limit Exceeded) – You can find this method as commented code in the file. The idea is – to trap water, we must have "$\cup$" kinda shape – i.e., up, down, up. Vague, right? Let's make it more quantitative. Recall, we had to answer 2 questions – How to trap & How to calculate. The calculation bit depends on the smaller value of the 2 pointers. So, for the tedious approach – start pt = 0, end pt = 1. Now, we want to find an interval where height[end] > height[start] because once we exceed the height of our starting pointer, we will have stored the maximum water possible b/w 2 confined bars. After this, we can reposition the start pointer to the end pointer & try to find such an interval again. Once, we get an index interval satisfying the above condition, calculating the amt of water is easy – just add the difference b/w the min(height[start], height[end]) &

all bar heights b/w a given set of start & end indices.
Time - O(n), space - O(1)
However, there's more to time complexity than the Big O notation.

2 pointer 2 - Notice while calculating the water stored, we first tried to find an interval which met the criteria of height[start] < height[end], if this wasn't met throughout the array, we calculated the amount w.r.t. max end pointer value. Now, is this really necessary? Come to think of it, why can't we initialize our pointers to the extremes & then continue adding the water capacity as we converge in? Just one condition, we must see a "∪"; this shape inherently involves keeping track of the left & right max values. So, here's the idea -
One key takeaway from the initial idea was that water stored will always be w.r.t. the min pointer (start/end). So, for every iteration let's converge in, towards the middle from the smaller side. This way, we'll be narrowing our interval. So, track the start max & end Max values, and go on adding water from the side with the lower value, while updating the max terms. Observe that, the startMax & endMax values are updated first & the water is added later. This is done to ensure that if the current bar is the start or end max, no water is added, and effectively a new interval begins.
To understand these approaches better, do a dry run of the code using one of the test cases.
Time - O(n), space - O(1).

Sliding Window



XIV) Best Time to Buy & Sell Stock (LC 121)

Brute Force - Technically, the question wants us to find the best buy sell combo to maximise the profit. Well, we've been given an array & by the unsaid law of "BRUTE FORCE", don't overthink it, we shall make every possible buy sell combo & return the max-profit that can be made amongst these combos. In other words, pick a day to buy & choose to sell on a day in the remnant array that can maximize (sell-buy) & finally return the max value of the max profit that can be made from each of the "buy days." (The optimal O(n) sol^n will be a more mature version of this realization)
Time - $O(n^2)$; space - O(1)

Two Pointers (WRONG APPROACH) - Now when I say, wrong approach, I don't mean it any harm. There are perfectly valid ways to solve this problem using the 2 pointer method. It's just that the approach which I initially thought of was horribly wrong & I wanted to take a moment to analyse it. The approach was, set 2 pointers, buy & sell to 0 & len(prices)-1 respectively. Iterate over the array from 0 to len(prices) while comparing prices[buy] with prices[i] & updating buy if prices[i] < prices[buy]. Similarly for sell, compare prices[sell] with prices[len-1-i] & if prices[sell] is smaller, update sell = len-i-1.
Now, this approach kinda seems right; however, there is something intrinsically wrong about updating both buy & sell pointers together. The problem is you might skip a few cases! FYI - this cleared 186/211 test cases on leetcode so, trial & error is not really the way to go about things. ex - [2,1,2,1,0,0,1]. Fails.
WHY? because while the optimal points are indices (1,2) or (5,6). This method makes it so that neither of these is ever considered.
learning - I wish I could say why generally approaching such problems (if there's a broad generalization) with this kind of a closing from both ends two-pointers approach is wrong. Especially when the problem boils down to the following optimization -
max(prices[sell] - prices[buy]) ; s.t. buy < sell.

The buy < sell bit is what made me think of the wrong approach. A general advice would be to spend more time before coming up with greedy approaches such as this. The answer is also a kinda greedy approach but it's more of an obvious observation.

Track the Smallest Value: What's the best price to BUY at? the lowest, right? so, with this in mind, let's enforce the cond^{n} of buying first & selling later. Initialize the smallest value to the first value of the prices array, then update smallest value by iterating over the array, if it's not smaller than the current smallest value, then compare to the current profit. Seems too easy, this approach, right? Well, the idea is simple...
we update the smallest ptr (i) such that until the next smallest ptr (j) we compare all results with i, so prices(i) is the smallest price until the i^{th} index & until j^{th} index price(i) is the smallest price, $\Rightarrow$ for every value in (i, j), the profit will be maximised if buy is i.
Time - O(n) ; space - O(1).

XV) Longest Substring Without Repeating Characters (LC 3)

Brute Force - see the word "substring", that effectively sums up the brute force method. Do exactly what the question says, find the longest substring without repeating characters. Iterate over all the substrings & return the max. length found.
Time - $O(n^2)$; space - O(1)

Hashing + Sliding Window - Observe, the # of unique values possible in our string = 26 (capital English Alphabets). So, our hashmap will have a maximum of 26 entries. Effectively, constant space. Coming to the algo. Initialize the start pointer (winStart) to 0. Now, iterate over the string, s whenever a character in s, already exists in our set, we must have reached the end of the substring in consideration, with length = len(set). At this pt., we must start pruning values from the start pt.

or, in simpler words, shorten the window from the start pointer to make sure that there is no repitition. In this process, continue comparing the lengths of the substrings formed. There are multiple different versions/flavors of this approach, however, the idea remains the same.
Time - O(n), Space - O(26) ≈ O(1)

XVI) Longest Repeating Character Replacement (LC 424)

Brute Force - Back with the grind. Let's first understand, how to identify a substring compatible with the question. If a substring has n repeating & k non-repeating characters, the required length will be n + k. So, how to find the length of a substring here? It's easy take a substring, find the most commonly occuring character, if its frequency (n) when subtracted from the length of the substring (l) is smaller than or equal to k, we're good, i.e; l - n ≤ k. All such lengths are acceptable & thus at the end we can return the maximum value amongst these lengths. Even though, this approach involves an element of brute force, we can't do away with the hashing aspect, which is required to determine whether a string is even allowed.
Time - O(26 n²); Space - O(26) ≈ O(1)
(fetch the max char ← all substrings. → alphabets)

Hashing + Cond. sorting - This approach primarily abuses the fact that the number of unique elements in our string can be max. 26. So, here's the idea, in the brute force method, we have already established that we must find the most frequent alphabet in a substring. So, how do we keep track of the most frequent alphabet? Simple! In our sliding window, let's track the alphabet with the max. frequency in the corresponding hashmap. To do so, we'll continuously sort / or find the element with the max. frequency. In short, start with winStart = 0 & iterate over winEnd [0, len(s)). Update the hashmap with the character at winEnd & then fetch the new max. At every iteration, we'll check if the cond^n of l - n ≤ k is still valid. If it isn't, we can start

looking at the substring with winstart shifted by 1, else, we update the result. This might seem slightly confusing, so, let me break this down:

```
if l - n > k:
    char[s[winstart]] -= 1
    winStart += 1
else:
    result = max(result, winEnd - winstart + 1)
```

← we need not update the result here, because we know that this value cannot be greater than the prev. result as we are increasing winStart by 1. Also, the substring starting at winstart+1 might not even be valid, if winStart was pointing at the most frequent character because over l & n both reduce by 1 & thus, the relationship $l - 1 - (n-1) > k$, still holds. So, the role of this if cond" is just to redefine the scope of our length & not update the result. Hope this adds up. Try an example like "AABCD..." with k = 2. Here, the cond" will become true when winEnd points at 'D', however, shifting winStart by 1, won't help because the excess chars will still be 3 in the substring 'ABCD'.

* The explanation for these supposedly simple questions are getting slightly deep, but this exercise is very important to highlight the assumptions made in the process to ensure no edge cases are left out.

Time - O(26n) ; space - O(26) ≈ O(1)
↳ fetch the max everytime

Max. Frequency + Hashing - This approach is essentially a more mature version of the previous solution. Instead of fetching the alphabet with the maximum frequency every time, we update the max. frequency each time we encounter a character. So, if the character at winEnd is 'A', we shall update the hashmap for 'A' & then compare its value with the running max-frequency (initialised to 0). Rest everything remains the same.

Time - O(n); space - O(26) ≈ O(1)

XVII) Permutation in String (LC 567)

Brute Force - By now, we know that the brute force approach requires to glance over all substrings & perform the required comparisons. The only thing extra here is that, we need to check if a PERMUTATION of s1 exists in s2. So, just hash s1 & compare the hashmap of s1 with hashmaps of all substrings of s2. We can use the Counter type from collections for such questions.

Time - $O(s2^2)$; Space - $O(26)$

The time complexity is not exactly $O(s2^2)$ here because, we are also comparing hashmaps in $O(26)$ time when the length of the hashmap of a substring of s2 = len(hash(s1)).

Naive sliding Window: There's no point in checking substrings of size other than s1. So, let's keep it that way. Create a window of size s1 & drag it over s2 to check if the hashmap/counter of the substring equals that of s1.

Time - $O(26 \cdot s2)$; space - $O(26)$

↳ again an overestimate but better to mention it.

Optimal sliding Window: Honestly, at times with these "optimal" approaches, the advantages are outweighed by the messy code. The idea is to track remove the hashmap comparison aspect. This can be done by tracking the differences b/w the substring hashmap & that of s1 iteratively, instead of doing it all at once. The code is pretty self explanatory, however, it's imperative to mention that I did not arrive at this code all at once, took a few wrong submissions to identify all edge cases. Such is the situation with messy if-else codes! The point is, if a character in s2 is in s1, we check if its frequency is aligned with s1, if so, we reduce the difference, if it has exceeded the required frequency, then we must prune the substring until our current character is balanced while also updating the frequencies & the difference for characters encountered while pruning. Lastly, if a character is not in s1, we start over.

Time - $O(s2)$; space - $O(26)$

XVIII) Minimum Window Substring (LC 76)

Brute Force - It's honestly boring now! So, I'll keep it short - compare all substrings of s with t and amongst the substrings that satisfy the condition, return the one with the minimum length. If there's no such substring, just return "".

Time - $O(26n^2)$; space - $O(26)$

comparing the ↵ ↳ # substrings ↳ # capital alphabets.

hashmaps of s's substrings & t.

Sliding Window - The approach to this question, is similar to the one used in previous problem, i.e., keep a track of the number of unique characters required. We'll run an unconditional while loop. The beauty of this solution lies in the systematic classification of the conditions that might occur. We'll initialize 2 variables l, r = 0, 0 & winStart, winEnd = -1, len(s). l, r will be used to track the on-going windows & winStart, winEnd will be used to track the min. until a point. So, coming back to the conditions -

1) Let's find a solution that might work, so, go on increasing the window's length towards the RHS until required == 0.
2) Once we have a substring that satisfies the cond", we shall compare it w the then min., and make the necessary updates, if any. Lastly, we will increment l to reduce the size of the window & update ctrS & "required" if s[l] was in ctrT.
3) If r or l have exceeded len(s) - 1, we shall break the loop.

Time - $O(n)$; space - $O(26)$

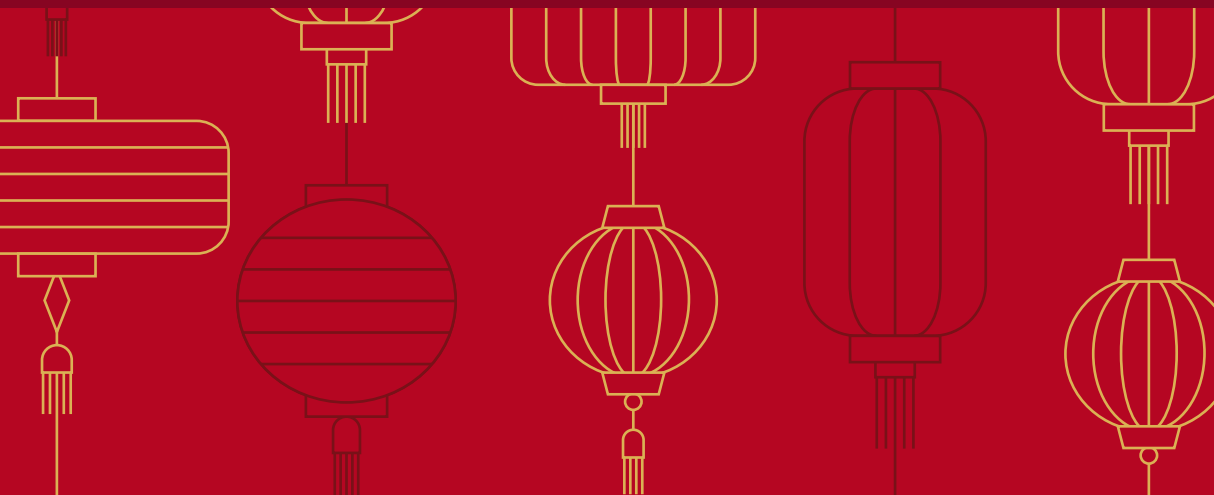
XIX) Sliding Window Maximum (LC 239)

Brute Force - It's really simple, just do as the question says. Drag a window of size k across the array & return the maximum element

in every such window.
Time - $O((n-k+1)k)) \approx O(nk)$; space - $O(1)$

Deque - This problem is actually not really about using the sliding window approach as the solution, rather it's just a part of the question. So, let's start with the linear time approach. Observe with every sliding window, we only find ourselves interested in the max. value. Consider a data structure storing values in a non-ascending order capable of popping values from both ends in constant time. This data structure will be used to track the values that MIGHT BE RELEVANT for finding the max. value for the CURRENT & THE SUBSEQUENT windows. The plan is simple: old values (values at lower indices) are only relevant if they are higher than the new values (values at higher indices) because if even within the current window an old value is being bested by a new one, we can be sure in saying that we won't need it now or for the subsequent windows. Thus, the idea of maintaing a non-ascending order. So, the algo would be to append a new value directly if it's smaller than the smallest value in the data structure, i.e. smaller than the last value; else prune the data structure from the RHS (pop) until this cond" is met or the data structure is empty. Lastly, just popleft, everytime you've hit the k size for a window.
*FYI - this data structure is what we call a double-ended queue, or deque.
Time - $O(n)$; space - $O(k)$
↳ max.

Stack/Queue



XX) Valid Parenthesis (LC20)

Stack - Notice - "NO BRUTE FORCE" :). I'll explain why? How do we ensure valid parenthesis? ① What opens last, should be closed first (RING - LIFO: Last In First Out). ② whatever opens should be closed. Everything else is just wrong. So, it's essentially screaming stack! Append all opening brackets & whenever you encounter a closing bracket, compare it with (last element in the stack & pop if it's a match. SIMPLE.
Time - O(n); Space - O(n).

XXI) Min Stack (LC155)

2 Stacks - Again, no brute force :), there's just no way to brute force this problem & even every operation in constant time. The solution is quite simple really. Maintain a normal stack with the elements being pushed in, while maintaing an array storing the min values until the current index. This way we can make sure that both stacks are always in sync.
NOTE - It's quite luring to maintain a single minStack at the first glance, i.e. storing elements in a non-descending fashion; however, doing so, will not be possible in constant time and also prevent us from tracking the indices properly.
Time - O(1) per operation; Space - O(n). to store the stacks

XXII) Evaluate Reverse Polish Notation (LC150)

Stack - I really like these kinda questions because there's just no other way xD. I mean with pure stack/queue questions such as this, we cannot avoid using stack/queue, Honestly, knowing a question is around stack/queue is knowledge enough to get it done (words of inexperience? maybe :p) Let's get to the sol^n. It's fairly simple, append the integer values to the stack &

whenever you encounter any operator, evaluate the expression with the last 2 values in the stack by popping them & finally appending the result for further calculations.
Time - O(n); Space - O(n)

XXIII) Generate Parantheses (LC 22)

Brute Force - Hehe, finally! The idea is to create as many combination as possible (2^{2n}) & then run the valid parenthesis function for each one. If the combination is valid, append the result else just let it be. → has to be the worst one until now :).
Time - $O(2n \cdot 2^{2n})$; Space - $O(2n)$ ↳ besides the output.

DFS - While creating a valid combination; we must follow 2 rules-
① # opening brackets & # closing brackets should always be <= n.
② # closing brackets <= # opening brackets. That's it.
Almost all generative type questions are recursively solved. So, let's just apply these 2 rules in if statements & let recursion take the wheel. Once you see how the results are created, you'll realise a DFS based approach, where a comb^n is explored fully before branching onto others.
My solution uses a string parameter in the recursive f^n. However, this can be done using a stack as well. The Neetcode guy did it using a stack & so this question was included in stacks. I for one, don't like that method.
Time - $O\left(\frac{4^n}{n\sqrt{n}}\right)$ → $\frac{1}{n+1}\,{}^{2n}C_n$; Space - $O(2n)$ ↳ besides the o/p
↳ Catalan number

XXIV) Daily Temperatures (LC 739)

Brute Force - For every element, iterate over the remnant array until a temperature greater than the element is found.

Time - $O(n^2)$; space - $O(1)$

Stack - The approach is really simple, just maintain a monotonically decreasing stack. It's easy to try the sorting method as well, however, in order to keep track of the relative positions, the time complexity can actually be worse than brute-force! $O(n^2 \log n)$ so, let's come back to stack. your hint for stack/queue based questions should be the necessity of a relative order. Back to the monotonically decreasing stack. Storing a value in our stack is only worthwhile until we have not found a warmer temp. for it. Thus, the monotonically decreasing stack. So, as soon as you find a temperature warmer than the last value in our stack, start popping the values & store the index difference between them. Keep popping values until a warmer temp is found in the stack or the stack is empty. Finally append the new value.

Time - $\sim O(n)$, space - $O(n)$.

XXV) Car Fleet (LC 853)

Sorting - The problem is slightly tricky to understand, but it's really just about middle school math. Firstly, sort the cars based on their gaps from the destination (target - position). Then, compute the time taken by each car to reach the target if it were to go at it's speed the whole time. Now, to evaluate the number of fleets, we can simply say that a car will join an existing fleet only it's time is lower than the first car in that fleet, else, it will create it's own fleet. That's basically the idea. This can be achieved by creating a time array & then iterating over it to return the number of fleets or, we can incrementally create a stack of fleet leaders based on the time each car takes. Using the stack approach is more memory-efficient.

Time - $O(n \log n)$; space - $O(n)$

XXVI) Asteroid Collision (LC 735)

Stack - Again, we see no brute force for this problem. The stack approach generally comes easier than "dumb" brute forcing. What essentially led me to consider a stack for this problem was the idea of collision and subsequent destruction/popping. The idea for this question is really simple. The asteroids are given in a row, & in this row we are interested in identifying the collisions to find out the remnant asteroids. Thus, we would work with the negative asteroids (moving leftwards). In the row, if we have the initial objects moving leftwards, we need not bother as they shall never interfere with any subsequent right moving (positive asteroids) to cause collisions. However, once we have a positive element, we must look out for the negative ones as they will start the collisions. Since, we are maintaining a stack, if the collision causes destruction of a positive element we can pop it. This can be seen in the while loop & the sol^n. As per the collision possibilities, there are 3 conditional statements used. Finally, the remnant asteroids, i.e. elements in the stack will be returned as the answer.

Time - $O(n)$; Space - $O(n)$ → maintaining the stack.

↳ the while loop in every for statement does not lead to a quadratic time complexity, because the max. elements that can be popped = len(stack). Assume if every element is positive & at the end, we add the biggest magn negative asteroid that destroys everyone, there will be $n-1$ collisions ⇒ # runs = $n + n - 1 = 2n - 1 \approx O(n)$

XXVII) Online Stock Span (LC 901)

Brute force - Need I even write this :p. It's really simple, do as asked & for every element go back in the array until you find a bigger entry.

Time - $O(n)$, Space - $O(n)$ ← to store the input

* Note - it's not a good idea to assess the efficiency of these algos just based on the big O notation, as there is more to efficiency. The following algos will always outperform brute-force but the big O notation will remain the same.

Only Check where it might matter (Array) - Definitely the most wordy sub-heading until now :). The idea here is pretty simple, there is an obvious relation b/w the days corresponding to the last price & the current price & we intend to use it.

1. If the current price < last price. - days = 1. (duh)
2. Elif the current price == last price:- days = days of last price
3. Else, lastly the most interesting case, wherein we need to look back & find a price > the current price. This shall be done carefully. Observe - if the current price > last price, the closest price that can be bigger is the price at array[len(array) - days of the last price]. Take a moment to absorb this as this is the only relevant point making this approach different from brute-force. Now, if the current price exceeds the value at array[len(array) - days], we need to add the days of this price to days & check again. Essentially what we have is a while loop which will run until a bigger price is found or days exceed the len(array).

Stack - Essentially, the same idea, however, instead of storing the unnecessary days in between, we shall pop them & there shall be no need for hopping days. In simpler words, maintain a monotonically decreasing stack. Compare the present price to the last price in the stack & use the same 3 conditions mentioned above. The only difference is in the 3rd condition, wherein we must not only continue adding days but also pop the last price until a price bigger than the current price is found or the stack is empty.

XXVIII) Decode String (LC 394)

Recursion - Now, this is one of those questions wherein the ability to translate an intuitive yet hard to define mental algo into code is tested. It's not hard to decode a small string manually, however, generalizing it is notoriously tricky for first-timers. Anyway, the idea is, go diving into every bracket until you are only left with a string of alphabetic characters. In this process continue creating the decoded string. To code this up, I have created a helper function recurseDecode(t), t here is the string that needs to be decoded. We shall put values in recursively. The point is wherever we encounter a number, we shall repeat the string enclosed in brackets that follow. So, we are on the look for numbers, as & when we find numbers, we recurse over the string enclosed. The code is self-explanatory; the recursion statement is → strvalue += recurseDecode(nextIter) * num.

Time - O(n); space - O(n) → to store the o/p.
↳ optimal time to decode the string
as we must go over every element

stack - Using a stack for a problem like this seems counter-intuitive, really! However, instead of creating a stack of recursion, we can also optimally address this problem by storing the relevant bits of the i/p in a stack. This solution offers a way out w/o worrying about the piling stack of recursion but in terms of Time complexity might be slightly worse. Observe how for recursion, we were on the look for numbers & '[' (opening braces). Here, however, we shall track the closing brackets, i.e. ']' because closing brackets occur in the right order for decoding purposes. While decoding a string nested with brackets, the ideal flow of decoding is to go in a descending order of bracket openings, i.e., decode the nested strings in the reverse order of their opening. Another way to say that is, decode the strings in the order of their closing brackets, as the bracket that was opened last, is closed first. It seems like I have

gone overboard with this rather simple explanation, but I hope this adds up. So, the goal is to continue appending chars to the stack until we encounter a ']'. Once we encounter a ']', our goal is to first pop values until we reach the the '[' to extract the alphabets enclosed & then pop until we extract our number to repeat this sequence of alphabets. Finally, once we have repeated string, we append it back to our stack for later use. A word of caution - since with stack, we pop values in a LIFO fashion, observe, how the presentString value is updated, instead of presentString += stack.pop(), we write. presentString = stack.pop() + presentString. Coming back to our method, we do not have to worry about there being any more nested brackets within the current set that we are assessing because we start with the first ']' or the last set of nested brackets in our encoded string.

Time - O(n); space - O(n)

XXIX) Remove K Digits (LC 402)

Brute Force - Try removing any k digits & see what happens. Trying all possible combinations with k digits removed, gives a factorial vibe :O. So, don't even try this method with big cases unless you wish to see time limit exceeded :).

Time - $O(^nC_k)$; Space - O(n) ← to maintain the min. value

Stack - What property must a digit have for us to remove it & inch toward the min numeric value? It's not so hard to reach to the idea that a digit might be worth getting rid off if it exceeds the digit after it. Now, to convert this "might" into a "must", we need to delve deeper. Let's take a number → abcd. Assume, a > b is the only information I have with k = 1, can I say that bcd is the smallest number possible? Yes. assume a = 8, b = 7, c & d = 9, you see unless I remove a, I will always start with a, which is sitting at the most dominant

position, so, since all 3 digit numbers starting with a exceed all 3 digit numbers starting with b, since $a > b$, I can comfortably say that removing a will ensure the lowest 3 digit number in this example. Thus, our concern of removing digits only arises if they exceed the digit that follows. Another thing, say if $k = 2$, the problem effectively becomes remove 1 digit from bcd to minimize the value. Basically, removing the $n+1^{th}$ digit will not meddle with our choice of removing the n^{th} digit. So, it's really simple now, init) Maintain a stack which will be monotonically increasing until k values have been popped. If the present value is smaller than the last value in the stack, pop it. The code is very simple to understand.

Time - O(n); space - O(n).

XXX) Remove All Adjacent Duplicates in String II (LC 1209)

stack [O(nk)] - Pretty simple really, without thinking much, continue appending characters to a stack & upon appending every character when len(stack) >= k, check if the last k characters are the same. If so, pop all of them, if not, move on.

Time - O(nk); space - O(n) ← stack

↳ iterating over the array + checking for the last k vals each time.

stack [O(n)] - Although this solution is quite simple, I wanted to mention this question to emphasize on storing tuples in a stack. All we need to do is, keep a track of the count of chars until a point, i.e. if the n^{th} character is 'a', I would store this char 'a' in my stack as ('a', # a's in succession including the n^{th} char). So, when the 2nd part of this tuple equals k, we must pop the value. Finally, once we've iterated over the string, we'll be left with char & frequency tuples in our stack which we'll convert into our result. Again, this might seem weird, but just try to do a dry run & it'll add

up.

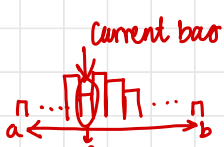
Time - O(n); space - O(n) → stack + result.
↳ iterating over the string.

XXXI) Largest Rectangle in Histogram (LC 84)

Brute Force - The brute force approach in this case isn't actually that bad :p. We can solve the problem in quadratic time because this is effectively like looking up all substrings. So, yeah, look up all the substrings & store the max area.

Time - O(n²); space - O(1) → we only need to track the min. value in ↳ all substrings every substring & the max area until a pt.

Stack - I'd be dishonest in saying that this problem wasn't a pain in the ass. So, this explanation isn't going to short by any measure :). So, as is it goes with difficult problems, let's make some keen observations:- ① for any substring, the area only depends on the min height & the length of the substring. ② This one might be obvious, but we can only increase the area by adding taller bars or by increasing the # of bars in our substring. The solution really is just a combination of these 2 observations moulded into a monontic stack. You see, if we encounter a bar smaller than our current bar, the maximum area that we can get out of keeping that bar height as our min will be from a bar height smaller than it to a bar smaller than it. Sorry if it sounds confusing, lemme illustrate this idea -



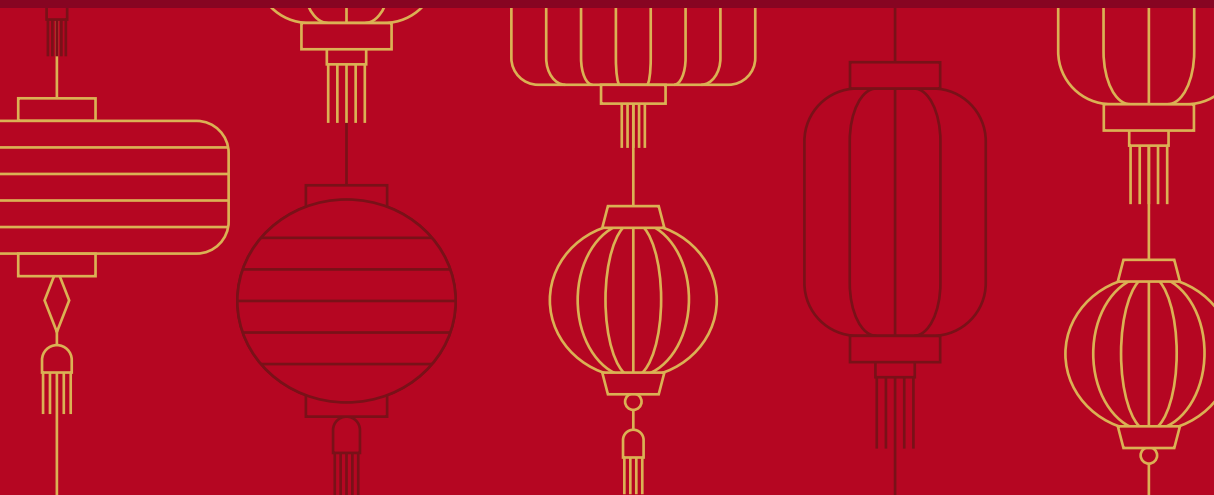
height(c) >= height(a) & height(b).

You see, in our result, we will have an area for which the height of one of these bars will be considered minimum, so if we're able to compare the areas spanned by keeping each bar as the min height

we can decisively choose the max-area, without worrying about any edge case lost. This realization is half the battle won. Now's let's focus on coding this up in linear time. This part is not so intuitive to arrive at but surely makes a lot of sense. The idea is to maintain a monotic non-decreasing (or increasing stack). In this stack, we shall store just the indices of bar heights in the heights array. The point of maintaining a monotonically increasing stack is that, at any point in the stack, all values to the right can be included in the area with the current point serving as the min. height. Storing of indices helps us determine the value of the width are dealing with. So, if you look at the code, it exudes simplicity, all we have is a stack & a simple while loop checking if the present height is smaller than the last entry in the stack. Like I said previously, the maximum area considering a particular bar as the min. height spans from a bar smaller than it to a bar smaller than it. (refer the abc bar illustration). So, in maintaining a monotonic stack, we are effectively keeping a track of the left boundary for every element (the immediate left value) & seeking the right boundary in every element as we iterate over the array. So, when we find a value smaller than the last value in our stack, we fix a pointer there to analyze the width of our area & continue popping values until the cond^n holds. Effectively every element popped has its max area with its height as the min. height considered. A small hack to make this work seamlessly is that we append a 0 to heights to ensure that the area corresponding to all natural numbers in the array is evaluated.

Time - O(n) ; space - O(n)

Binary Search



XXXII) Search a 2D Matrix (LC 74)

Transform 2D indices into a 1D representation - It's really simple, we've been given a 2D matrix, wherein the rows are in ascending order as are the elements in each row. Effectively, if I were to append all of these rows into one big array, I could straight away use binary search. So, let's just convert the 2D rep into a 1D rep and act as if we were dealing with one big array. assuming the length of each row / no. of columns to be m. to convert/read a 1D value from our input matrix, we can just look up → `[i//m, i%m]`. That's it :)

Time - O(log(n*m)) ; space - O(1).

XXXIII) Koko Eating Bananas (LC 875)

Binary search over the possible speeds - Actually, this subheading summarises the solution. Can we define a range of values that could result in the min. suffiient speed of eating for Koko? Of course we can! The range for $k \in [1, \max(piles)]$. We can check for all values in this range whether Koko can finish all bananas at a speed k in h hours. While binary searching over this range, each time we encounter a solution, we shall restrict the upper limit else we shall keep on increasing the lower limit.

Time - O(len(piles) x log(max(piles)))

len(piles) * check whether a speed is possible

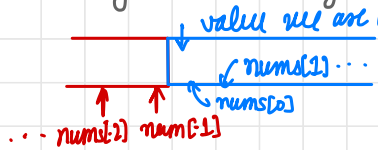
max(piles) → range of values we need to binary search over.

XXXIV) Find Minimum in Rotated Sorted Array (LC 153)

Brute Force - min(array) :), total B.S. not even the question

Time - O(n); space - O(1).

Binary Search for the inflection point - We have been given a Rotated sorted array, let's try to use that! The resultant rotated sorted array essentially has 2 sorted arrays →



Observe that the value we are chasing after is the og nums[0] that acts like an inflection pt. because, the values to its left & right are both greater than it. This is the only pt. in the rotated array where this property holds. In the code, we've chosen a slightly different point of comparison, however the logic remains the same. The reason behind involving nums[0] is to avoid the index range issues brought up while using nums[mid-1] & nums[mid+1] together. So, we're saying if index mid has the min-value, mid-1 will have the biggest value. Thus, if we are looking at the correct index, mid, nums[mid] < nums[0] & nums[mid-1] >= nums[0]. It's fairly straightforward.

Time - O(log n); space - O(1)

XXXV) Search in Rotated Sorted Array (LC 33)

Find the min & Binary Search over the shifted indices - We've just discussed the algorithm for finding the min value in a rotated sorted array. So, once we know the index of the min value, we can transform every index as if it were in the og array to the now rotated array. The function to do that is really simple (refer, reverseIndex(ind)) in the code. Finally, act as if we it's a normally sorted array & make the required comparisons by transforming the index with reverseIndex().

Time - O(2 log(n)) = O(log(n)); space - O(1).

finding min ↲ finding target

XXXVI) Time Based Key-Value Store (LC 981)

Binary Search to get the max value compliant with the requirement - That's really it, it's pretty simple :)

XXXVII) Search Suggestions System (LC 1268)

Sort + Two Pointers - This doesn't read binary search :O. Well, with binary search the code gets dirty, but I'll explain. The use case for this question needs absolutely no explanation! So, imagine the list of products, now, why sort them? The reason is fairly straightforward, it's only with a sorted array that we can optimally provide suggestions, else we will have to look up each substring again & again & that would make for a terrible algorithm. With a sorted array we can decisively create bounds with successive characters in our search word. So, upon sorting the array we just iterate over the products array to find the bounds for the substring seen until then. I have used l to represent the lower bound & h to denote the upper bound. Initialize l to 0 & h to len(products) - 1 & let's converge to our search word. This is where binary search could have been used to find l & h but since the range of values is continuously reducing binary search's code filth outweighed its speed for me. Basically, we want to pass over l's & h's corresponding to which either the i^{th} index does not exist (length too short) or the i^{th} index != i^{th} index of our search word. If products at both l & h are able to avoid these conditions, so will every product in b/w.

Sorting + Binary Search (Bisect) - I know I said that the binary search code was filthy but then I saw the solutions & I was introduced to bisect. Bisect is this really cool library to identify insertion points for new elements in a sorted array. You can read more in the documentation. The point is that

using a function, bisect bisectleft( ) we can identify the left most insertion point in a sorted array for any value, and internally this library uses binary search :), so a load off my shoulders! Anyway, if we maintain a prefix, i.e. a substring of all characters now we can use bisectleft to find the leftmost insertion spot for our prefix which effectively serves as our lower bound (l). While using this function we can also set an initial lower bound for our insertion point (i), which also fulfills our requirement of successive conversion. So, all in all a great discovery.

The big O notation of time complexity will look the same for both these methods, because we have the nlogn sorting beam which outweighs any subsequent minor linear to log time.
Time ~ O(nlogn); space - O(len(searchword)).

XXXVIII) Split Array Largest Sum (LC 410)

Brute Force - Try all combinations to get k subarrays & return the minimum sum. If this doesn't already sound ridiculous, lemme mention the time complexity associated - for creating k subarrays we have to choose k-1 points of division in our array, i.e. a time complexity of $^{n}C_{k-1}$!
Time - $O(^{n}C_{k-1})$; space - O(1).

Binary Search over the Possible Sums - I guess what made this problem relatively easy to solve was the fact that I already knew this to be a binary search problem. Otherwise, it wouldn't have been so easy. Despite knowing the problem to be a binary search one, it's not that simple to converge on binary searching over the possible sums, but I am sure it makes sense upon reading. Problems like this & Koko Eating Bananas provide viable alternatives to otherwise complex problems by binary searching for the answer itself. So, create a range for our answer

using l & h, with l = max(nums) & h = sum(nums). Then we shall perform a binary search over this range looking for our minSum. So, for each sum, we can check if we can get away with just k partitions, if yes; reduce the upper limit else, increase the lower limit.

Time - O(n + n(log n)) — finding max & sum ↙ ; n for checking the validity of each sum & log n for the number of sums we check for.

~~XXXIX~~) Median of Two Sorted Arrays (LC 4)

Brute Force - Brute forcing just takes the sting out of every problem now doesn't it xD. Just merge the 2 sorted arrays & return the median. However, the question clearly mentions logarithmic time, so no bs :)

Time - O(n); space - O(n)

Binary search for the Median with a weird twist :p - This is easily the hardest problem until now because both the logic & the code are hard to arrive at. Although I was able to arrive at the logic on my own, the code I wrote was not passing a few test cases due to index issues. Eventually I kinda lost my patience & saw the solution. Anyway enough with the rant, let's discuss the solution now.

After contemplating various failed ideas, it made sense to try creating 2 pointers but with a binary search twist. As I said, it's not very easy to arrive at this solution, but yes after multiple failed ideas, this one made sense. The idea is actually fairly simple. We know the position of our median value in the sorted array. So, let's try fixing a pointer (p1) for nums1 & p2 for nums2. Now, in a merged sorted array, I can say that p1 + p2 = median index. So, let's stick to that → we'll binary search over p1 & find p2 as the median index - p1. Now, comes the validation part. How do we know when we found our median? The answer is actually not that tricky. Here are 2 ans

we know that the arrays are sorted, so, right_m ≥ left_m & & right_n ≥ left_n. However, what makes this interesting is if we can say that right_n ≥ left_m & right_m ≥ left_n. Why is this so interesting?

*Honestly, this space was only left because I zoomed in too much. Anyway, any imp. point → we have to make a comparison b/w the ends of p1 & p2 because we do not know the exact order b/w em upon finding the sol.

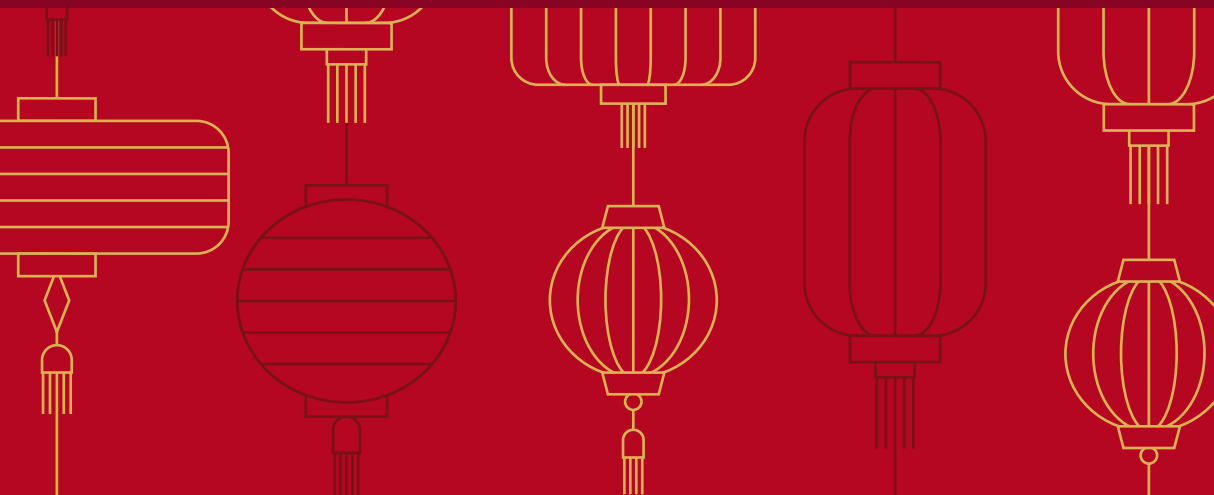
If we are able to arrive at pointers following this property, we've found our median! How so? Well by ensuring this condition we've effectively said that everything to the left of p1 & to the left of p2 occurs before everything to the right of p1 & to the right of p2 in the merged sorted array.

merged sorted array - left(p1) & left(p2) || right(p1) & right(p2)
w median

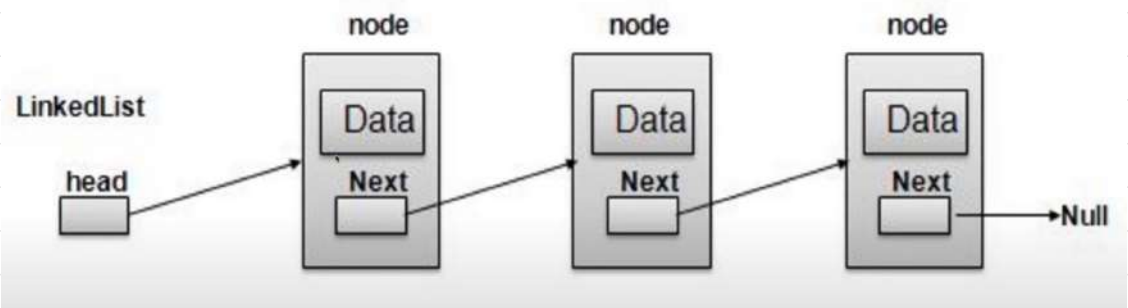
Read the starred note. See in our quest for finding the required value of p1 & thus p2, we always have to compare the values around p1 & p2 to converge to our answer. The important observation is that we only want to find the correct p1 thus, unless we have our og condition met we just continue our binary search over p1. To find the median, we require different formulae for even & odd m+n. That's about it. PS - In my initial attempts I was trying to make things work without using the inf conditions

Time - O(log(m+n)); Space - O(1).

Linked List



Before Leetcode, let's get the basics straight—



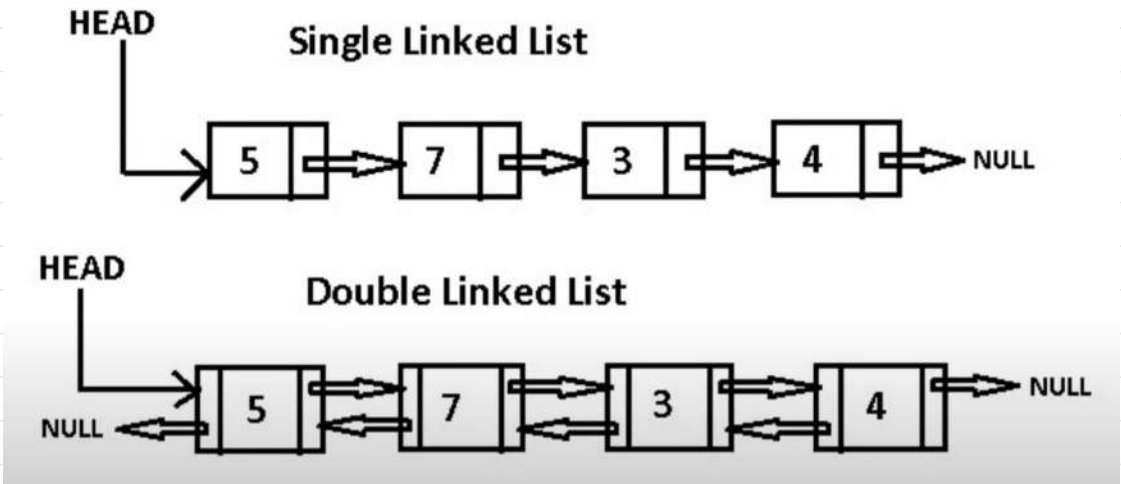


Table 1 Efficiency of Linked List Operations

Operation	Singly-Linked List	Doubly-Linked List
Access an element.	$O(n)$	$O(n)$
Add/remove at an iterator position.	$O(1)$	$O(1)$
Add/remove first element.	$O(1)$	$O(1)$
Add last element.	$O(1)$	$O(1)$
Remove last element.	$O(n)$	$O(1)$

The data structures from this point on are going to be pointer-centric, i.e., a node pointing to another node/nodes. Their implementation actually isn't all that tricky. However, it's easy to mess up the pointers. So, let's carefully understand how pointers work.

It's also worth mentioning that pointers in python do not quite share the implementation of other languages. Anyway, let's get started -

There's a big version of the explanation starting from the basics of mutable & immutable objects, however, it's really not all that important. So let's cut to the chase. For most people including myself a big disconnect with pointers in linked lists was the following code action -

curr = head ← head of the linked list
↑ pointer

altering curr & then returning head.

The disconnect here was with how come changing the values associated with curr can make changes to head. From a purely variable POV, that's what makes sense. If I were to write x = y = 5 & then alter y & return x. I'll get x = 5, no matter what! Here, however, head isn't an immutable object, rather it's an instance of the Node class used to implement the linked list node. Head in other words points to this object in memory which has certain mutable attributes like val & next.

This is where things get interesting! When you code, curr = head; curr also points to the same location in memory as head. REMEMBER, it's not just a value, it's an object with mutable value/s. So, when you change curr.next, it is the value corresponding to the next attribute of the object in memory that curr points to that you're changing. Thus, in the first iteration, since curr & head point to the same location in memory, you're also altering head.next.

XL) Reorder List (LC 143)

Slow-Fast Pointers to find the midpoint - Find the midpoint of the given linked list & then reverse the second half. After doing so, make sure that the last node in the first half points to None, because that node will be the last one. Finally, do as you were told, re-order! Initialize to pointers to the heads of the 2 halves, and then go on changing the next attribute of the nodes encountered. The code for doing so, might seem a little off at first, but just remember we are merging & re-ordering. Thus, in each step, we just want the current node, i.e. represented by head1 & the node next to head1, i.e. head2.

Time - O(n), space - O(1)

XLI) Remove Nth Node From End of list (LC 19)

Find the Nth Node from the end - Just one sentence to sum up this explanation :). Well, it's more about how you find this nth node from the end. We do not have the have the concept of reverse iteration over singly linked lists without actually reversing them iterating thereafter. So, how to do this efficiently instead? It's actually quite simple. Have 2 pointers at the head node & then shift one of them by n. Now, start shifting both these pointers until the node after the previously n-shifted pointer is None. Thus, our trailing pointers will now be at the n+1th node from the end, just skip over the nth node by connecting the n+1th & the n-1th node from the end. (left.next = left.next.next).

Time - O(n); space - O(1).

XLII) Copy List with Random Pointer (LC 138)

Hash Table / Dictionary - Create a dictionary with a None:None

key-value pair. Then iterate over the given linked list & create a copy node for every node encountered, in this copy node we are just concerned with the value, not the next/random values. Each time you create a copy, add an entry in the dict with the og_node as the key & the copy_node as its value to ensure that we can track the copy node for each og node. Finally, just rewire the copies by matching next & random attributes with the original linked list using the dict, we just created.

Time - O(n) ; space - O(n)

Interweave the Og & the Copied Nodes - This one sentence effectively sums up this approach, but I didn't find it very intuitive to arrive at it. Coming from an array mind-set, interweaving elements, isn't really the first thought! Anyway, the approach is really simple. say, you have the og list - (A)→(B)→(C). The interwoven list would look like -

(A)→(A')→(B)→(B')→(C)→(C')

In the first pass when you're creating copies & interweaving stuff, the next & random pointers of the copied notes will point to the ogs. After the first pass, we know that the value next to every og is its copied value. Use this to create another list with just the copies re-wired.

Time - O(n) ; space - O(1).

XLIII) **Linked List Cycle (LC 141)**

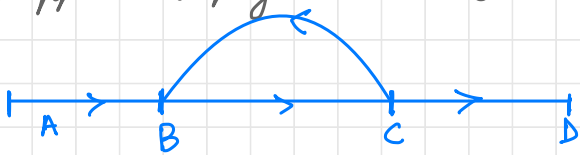
Hash Map/Dictionary - This is the most obvious & easy approach, just keep a track of all nodes visited. If before the traversal of the linked list is completed, a previously visited node is found, a cycle exists.

Time - O(n) ; space - O(n)

Floyd's Hare & Tortoise Algorithm – This is an existing algorithm that you're expected to know in an interview. The idea is actually quite neat. This algorithm entails the slow & fast pointer approach (previously used in re-order list). The takeaway is that if the slow & fast ptrs ever point to the same node then it can be said that there exists a cycle.

The algorithm is quite simple really, what's tricky is understanding why it works. It's fairly obvious to realize why slow & fast would never meet in a list where there are no cycles. However, proving that they always will in the case of a cycle is the main question. Here's a thought that could be leading you to believe that this won't work –

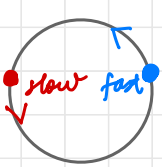
The fast pointer might skip the looping/cyclic node. This can never happen simply because this is how the progression looks –



A → head, B → C (loop), D → end.

Now, A can be = B & C can be equal to D, but, here's what matters, fast·next·next can never go beyond C because at some point either, fast or fast·next will be at C. So, if, fast·next is C, then the next update for fast, i.e. fast·next·next will be C·next = B. Else if, fast = C, then, fast·next i.e. B & fast·next·next lies between B & C. Again, trapped between B & C. So, the first wrong hunch has been resolved.

Having established that fast will always get stuck in the loop, if there is one, we now have to realise/prove that slow & fast will surely meet. Their meeting point will obviously be between B & C. It's easier to show this in a diagram –



Now, irrespective of where slow & fast start within this loop, knowing that fast is moving twice as quick as slow, it's pretty easy to understand that they must intersect.

Thus, this slow-fast / Floyd's hare & tortoise algorithm works because if there exists a loop/cycle -
(1) fast will always get trapped in it.
(2) fast will meet slow after slow enters the cyclic nodes (B-C) because it's twice as quick as slow.

Time - O(n); space - O(1)

XLIV) Find the Duplicate Number (LC 287)

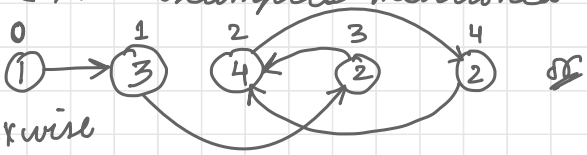
The solution to this problem is not at all intuitive to arrive at. So, this is a solution worth remembering. The algorithm is quite simple, understanding why it works is what matters

Floyd's Cycle Detection Algorithm - This algorithm has already been described in the previous question. However, the astounding bit is that, there we had a linked list & here we have an array. like I said, not an intuitive approach. This question is like an edge-case discovery which presumably would have been made unintentionally. Anyway, here goes -

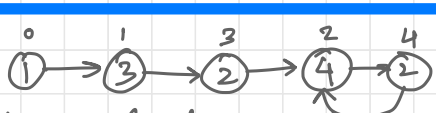
↳ carefully look at the framing of this question, an array of length $n+1$ with elements from 1-n with one of them being repeated.

↳ every element in the array points to a valid index. Moreover, every element points to a different index except 2 elements which point to the same index. This element/index is what we want to find. Since, 2 different elements point to the same index, there must be a cycle, while traversing the array by visiting the index of each element.

Take the examples mentioned -

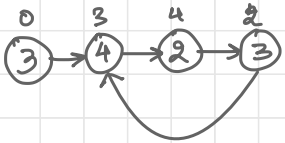


indexwise



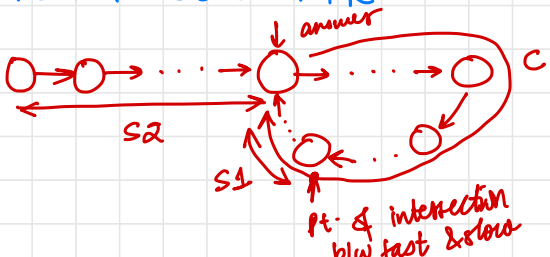
traversal order proposed.

similarly, (not important that we visit all elements / but we'll surely not miss out on any duplicates).



in each case, we want to find the index at which 2 arrows are directed so, we'll take the fast/slow pointer approach, & ensure that fast moves x2 as quick as slow and as we know, they will meet because, there's a cycle.
Upon identifying the point at which they meet, we head to the second part of this algorithm, where we look into converging at the desired index. We basically, initialize another slow2 ptr to 0, & then the midpoint in the traversal order b/w slow & slow2 is our desired value.
The question is WHY? Let's break the algo down into 2 phases -

FAST & SLOW PTR



S2: distance b/w the starting pt & our answer
S1: distance b/w answer & pt of intersection of the fast & the og slow ptr.
C: cycle length.

fast overlaps slow bc intersect it ⇒ distance covered by fast = S2 + C + C − S1 ; & distance covered by slow = C − S1 + S2
⇒ ∵ fast = 2x slow
∴ S2 + 2C − S1 = 2(C − S1 + S2)
∴ S2 − S1 = 2S2 − 2S1 ⇒ **S2 = S1** ✱

2 SLOW PTRs.

Having established that the answer, is equidistant from the og slow (pt of intersection b/w slow & fast) and the start. We can move another slow ptr from the start & the pt of intersection of the og slow & slow2 will be our answer
Time − O(n); space − O(1)

XLV) LRU Cache (LC 146)

Firstly, it's very important to understand the question. This is a good design problem that requires a decent grasp over the utilities of various data structures. The problem wants us to maintain a cache of size capacity, based on the stream of inputs. "Using a key-value pair" can happen in 2 ways - ① creating a new one (put); ② fetching the value of an existing key value pair (get). Based on this def^n of use, we can define the Least Recently Used Cache. Now, both put & get are expected to be O(1) operations. So, let's list down the requirements —

① Getting an existing element in const. time based on the key.
② Maintaining a relative order amongst elements in the cache.
③ Pushing an element to the end without disturbing the relative order in O(1)
④ Deleting the first element in constant time.

Having listed these requirements, it's evident that we need something like an Ordered Dictionary. Lucky for us, there already exists a data structure like this in python.

Ordered Dictionary - whenever a key is called (get), pop it & re-add it to the ordered dictionary, so that it comes last in order. Similarly, if a key is to be added, check whether it exists or not, check for the capacity & just add the key-value pair by either removing the LRU (first element in ordered Dict) if the capacity has been exhausted or simply add

Double Linked List + Dictionary - Actually, the data structure we used before is made from a double linked list & dictionary. The use of a dictionary is obvious, because it's the only way we know how to fetch an element by its name in const. time. So, we use a dictionary to store keys of the elements stored along with their corresponding node addresses/ids, to locate the value of the key in question. A question you might want answered would be why the double linked list?

Well the reason behind using a linked list is that we require a data structure that can hold a relative order & have const. time complexities for pushing an element to the end without disturbing the relative order of the other elements in O(1). So, we need something that stores relative order & not absolute order, thus linked lists. Now, why double linked lists? If you look at the implementation, we're gone for 2 pointers, i.e we have a head & a tail. The reason is quite simple → we need to add an element to the end (before tail) & pop an element from the beginning (after head). In order to do this we must use a double linked list.

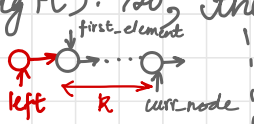
XLVI) Merge K Sorted Lists (LC 23) (not hard).

Brute Force - Preface - STUPID! I mean, just continue iterating over k sorted lists & keep creating the sorted list.
Time - O(nk) ; space - O(nk) ← the o/p.

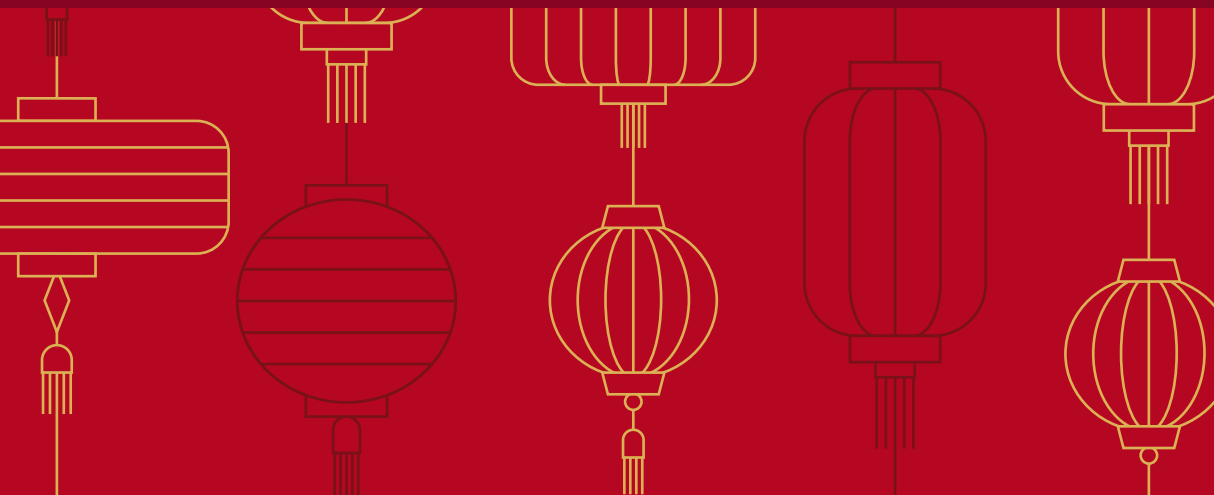
Merge sort - Use the merge sort idea & simply combine lists 2 at a time. This way the effort of k reduces to $\log_2 k$. So, you start with k lists → k/2 lists → k/4 lists & so on.
Time - O(n $\log_2$k) ; space - O(nk) ← the o/p.

XLVII) Reverse Nodes in k-Group (LC 25)

Maintain the Index & the first element of every k-group - The demand of the question is rather simple. The solution is also simple. We will iterate over the linked list & each time our index (1-indexed) is a factor of k, we shall reverse the sub-linked list of k nodes preceding the current node (including it). So, the k nodes following left need to be reversed. This is exactly what the code does. Time - O(n), space - O(1).



Trees



Reviewing Trees-

To solve tree problems we end up using some version of the following algorithms:-

1. Breadth-First-Search (BFS): scanning trees across levels-

code:-

```
queue = deque([root])
while queue:
    node = queue.popleft()  ← our current node.
    if node:
        queue.append(node.left)
        queue.append(node.right).
```

2. Depth First Search (DFS): scanning trees from root to leaves.

(i) Top to Bottom (iterative)

code:-

```
stack = [root]
while stack:
    node = stack.pop()  ← current node
    if node:
        stack.extend(node.right, node.left)
```

(ii) Bottom to Top (recursive)

code:-

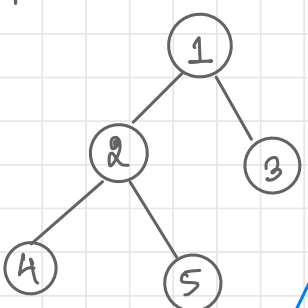
```
def dfs(node):
    if not node:
        return
    left = dfs(node.left)
    right = dfs(node.right)
    # any operation on the node.
    return
```

③ Recursion on subtrees - If you find yourself requiring repeated work for all children of a node, write recursive calls in the fn. over the left & right children depending on the requirement.

★ It's important to come up with algos & relate them to one of these fundamental approaches to proceed further.

XLVIII) Diameter of Binary Tree (LC 543)

It's important to visualise the algorithm before coding anything so, how do find the diameter, i.e. the length of the longest path between 2 nodes in a tree.



To solve this question, we just need 1 key observation & that is → the max. path through any node as the root = the sum of it's max root2leaf path of the left & right subtrees.

⇒ here, through (1) → 1 + (2,4,5) + (3) = 4.
(2,4,5) → 2, (3) → 1

through (2) → 1 + (4) + (5) = 3
(4) → 1, (5) → 1

and so on.

⇒ Clearly this question can be answered with a recursive dfs method.

Recursive DFS - We shall initialise our diameter to 0, and code up a dfs to return the max node2leaf path for every node.

```
def dfs(node):
    if not node: # 0 length
        return 0
    leftpath, rightpath = dfs(node.left), dfs(node.right)
    self.diameter = max(self.diameter, leftpath + rightpath)
    return 1 + max(leftpath, rightpath)
```

→ to calculate the max node2leaf length of the left & right subtrees.
→ updating the diameter
→ returning the height of the node, i.e. max node2leaf path length

XLIX) Subtree of Another Tree (LC 572)

Naive / Brute Force – Checking if 2 trees are same is an easy job, iterate over every item & compare. This check can be made for every node in the parent tree against the intended subtree.

Time – $O(n \cdot m)$; space – $O(1)$.
(n → parent, m → check for subtree)

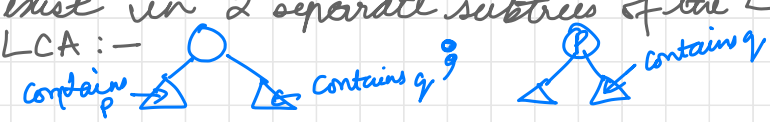
Merkle Tree – There is a lot of redundant & repetitive work going on in the naive approach. With Merkle trees we can create a unique representation of each node in a tree purely based on its subtree. Thus, we can bring the comparison of 2 nodes for a possible match down to constant time, instead of iterating over both of them each time. Take a look at the merkle fn defined in the code, we shall add a property merkle to every node in both our trees. This merkle value is written as a hash function of the merkle values of left & right nodes

⇒ `node.merkle = hash(str(merkle(node.left)) + str(node.val) + str(merkle(node.right)))`

Time – $O(n+m)$; space – $O(n+m)$

L) Lowest Common Ancestor of a Binary Tree (LC 236)

This question begs for just 1 observation & 1 optimization strategy. First the observation –
Observe, if the nodes we need to find the LCA for exist in the tree provided, they must have the root as a common ancestor. However, we must find the LCA. Now, there's something about the LCA which isn't really hard to arrive at. The LCA will either be one of the 2 nodes mentioned or the nodes will exist in 2 separate subtrees of the LCA node. i.e. → LCA :–



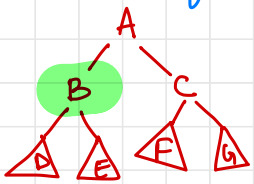
That's the observation. Now with this in mind we can move toward the optimization.
See, we can craft a code which looks something like the following —

```
def lca (root, p, q):                    # which checks if a node exists in the given tree
                                         # (here p)   (here, root.left)
    left_p = check_node(root.left, p)
    left_q = check_node(root.left, q)
    right_p = check_node(root.right, p)
    right_q = check_node(root.right, q)

    if (left_p and right_q) or (left_q and right_p):
        return root

    elif (left_p and left_q):
        return lca(root.left, p, q)

    else:
        return lca(root.right, p, q)
```

This seems redundant though, doesn't it? We are looking into the subtrees to find a node again & again. The idea makes sense but the execution requires an upgrade.

Create a common recursion function to check for both p & q (DFS).
It's actually extremely simple. Let's just try to find p & q in the left & right subtrees of the root & and the stopping clause for recursion is if we are able to get to p, q or the end (None). It's simple once you get it but easy to miss. We'll have a check for left & right each time. you see, if at any point we get non-null values for left & right, we've hit the LCA (i.e. p & q are both in separate subtrees (left & right) of the current node), if we do not get anything for either, it's the other one which will serve as the answer. Try a dry run to make sense of this.

LI) Delete Node in a BST (LC 450) (Learn this in context of AVL balancing too).



Delete B; ⇒ place D in place of B.; now you'll be left with E; E will be the smallest node in the subtree of C so the largest node in the subtree of D, so, either place it at the rightmost pt. of D or the leftmost point of E.

LII) Count Good Nodes in Binary Tree (LC 1448)

2D-DFS: It's quite obvious really, by the definition of a good node, we arrive at DFS. The question then comes down to keeping track of the max. value observed thus far down a particular path. Thus, while recursing we can simply check with each element if it's greater than or equal to the current max. It's not difficult. It's just about the craftsmanship that goes behind the recursive code. Usually looks like.

```
dfs( ):
    if (base cond^n (usually if not node)):
        return —

    dfs(node.left, ... any other param)
    dfs(node.right, ... any other param)

    operation you want to conduct for every node.

    return —
```

LIII) Validate Binary Search Tree (LC 98) ⭐⭐

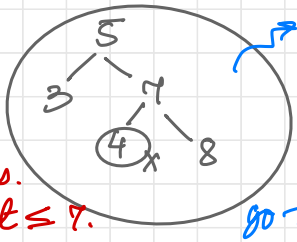
⭐ Remember a BST doesn't only adhere to node.val ∈ [node.left.val, node.right.val]. It's the entire subtree. So, while validating a BST. you need to keep track of the subtree limits instead of just

node limits. Let me explain -

⇒ we must create an interval for every node that is aligned with the previously visited nodes.

So, here, the limit on 4 is n't just ≤ 7.

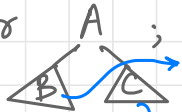
It is ≥5 & ≤7; which it isn't.



↑ not a valid BST.

See, I can't just go to 5 & see 3≤5≤7 & then go to 7 & say, 4≤7≤8; they aren't independent.

In general, for



A might be the right child of another node which acts a lower limit for A's subtree.

lower_limit of A ≤ B.val ≤ A.val.
(leftVal)

Again, A might be the left child of another node which fixes an upper limit for A's subtree.

A.val ≤ C.val ≤ upper_limit of A
(rightVal)

So, with this recursive relationship; we can easily create a recursive/dfs based solution with initial cond^n of root as lower_limit = $-\infty$ & upper_limit = $+\infty$; since, there is no limit on the root.

Time - O(n); space - O(1)

LTV) K^{th} Smallest Element in a BST (LC 230)

It's actually one of the simpler questions, all we need to do is perform an Inorder traversal and return the k^{th} element.

Now, inorder traversal is simple, I have gone with the iterative variant but there's obviously a recursive approach too. The idea is simple go as far left as possible & then explore the other right nodes & their subtrees on your way back up. It's really that simple.

LV) Construct Binary Tree from Preorder & Inorder Traversal (LC 105)

Again, a simple pattern spotting problem. This problem implores you to really understand what makes a tree different/unique. Using just one of the inorder or pre-order traversals, you cannot create a unique tree.

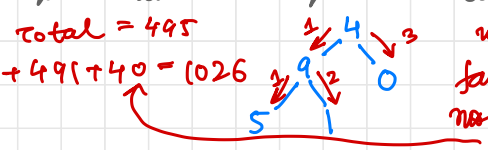
Recursion- With trees, it's usually about finding a pattern that scales so, just understand that the root of our tree is located at index 0 of our pre-order tree. Now, once we locate the root in our inorder traversal, we can objectively segment the tree into left & right. This right here is recursion. So, to rewind →

```
if not preorder or not inorder:
        return None
root = Node(preorder[0])
mid = inorder.index(preorder[0])
root.left = recurse(inorder[:mid], preorder[1:mid+1])
root.right = recurse(inorder[mid+1:], preorder[mid+1:])
return root
```

The reason why I am able to split the preorder array based on the root in the inorder array is simple; you see the no of elements in the left & right subtrees of every node will be the same in both inorder & preorder. We also know that preorder traversal goes down the left subtree first. Thus, we can make the split purely on the basis of the number of values ahead or behind mid & do this recursively to create our unique tree.

LVI) Sum Root to Leaf Numbers (LC 129)

The solution to this question is actually quite simple; the algorithm required is well explained in the question itself. Here's an example.



we go from 4 to 9 to 5 (leaf) => number so far = 495; then from 5 to 9; and 9 to 1 no. = 491; from 9 to 4 & 4 to 0 => no. = 40

DFS: I don't think there's any explanation required to support the reason behind choosing DFS. So, let's come to the construct. We'll have 2 global vars → sum - to track the sum so far & root2leaf to track the current root2leaf values. Everytime we hit a leaf we shall update the sum & everytime we intend to go up a node (i.e. coming back from recursion), we must remove the last digit in root2leaf. Simple:

```
dfs(node):                    ← initialized to 0.
    r2l = r2l*10 + node.val
    if node is leaf:
        sum += r2l
        return
    if node.left:
        dfs(node.left)
        r2l = r2l//10
    if node.right:
        dfs(node.right)
        r2l = r2l//10

    return
```

CRUX

LVII) House Robber III (LC 337)

The rule is summarized in one simple sentence - "CANNOT STEAL FROM 2 DIRECTLY LINKED HOUSES!" Which simply means that no parent child duo can be selected. This problem involves optimization & is related to trees, so it's going to have a touch of DP.

DFS- The choice is between stealing now or stealing later. B/w stealing the current node or heading into its children. This dilemma is held for every node involved. Thus, the construct is as follows.

```
dfs(node):
    if not node:
        return (0, 0)

    left = dfs(node.left)
    right = dfs(node.right)

    chooseNow = node.val + left[1] + right[1]

    chooseLater = max(left) + max(right)

    return (chooseNow, chooseLater)

return max(dfs(root))
```

The trick is to draw this recursive construct. It's a little tricky because of the 2D component involved.

present Node ← chooseNow → choose later for children

skipping the current node & choosing the best for children

answer

LVIII) Flip Equivalent Binary Trees (LC 951)

Recursion - It's a fairly simple question, but it's quite easy to get wrong. Ignore the base cases for the initial analysis. So, in simple terms what does it mean for 2 trees to be flip equivalent? Either the left & right subtrees are already aligned or they could use a switch. SIMPLE, just do so recursively!

The base cases you need to be wary of -

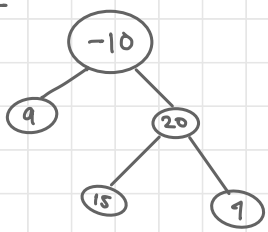
if both nodes being compared are None => True.

If either one is none or the node values are not equal => False

LIX) Binary Tree Maximum Path Sum (LC 124)

How do we arrive at the best/maximum sum path? The thing with non-linear data structures (trees, heaps, graphs...) is that it's not easy to come up with greedy algos. So, just take the simple approaches & you'll be good to go.

Consider the Maximum Path Sum with each node as the root:
This idea isn't really new. However, it is important to realise how universal its applications are. Consider the following example-



Here, if I want my path to go through (-10), I cannot go down the left & right subtrees of any children nodes. I can only choose 1. Like here, the max. path through (-10) is 9 + -10 + 20 + 15 = 34; CAN'T have both 15 & 9 if I want (-10) to be a part of my path.

Thus, back to the og idea. At every node, we shall consider the max. path that goes through it as the root & we shall return the value of the max. path with that node as a child, wherein we shall choose b/w left & right. Let's write the pseudo code to make this idea clearer. This idea obviously screams a recursive DFS-

```
dfs(node):
    if not node:
        return 0

    left = max(0, dfs(node.left))
    right = max(0, dfs(node.right))
    res = max(res, node.val + left + right)
    return node.val + max(left, right)

dfs(root)
```

left = max(0, dfs(node.left)) } max(0, path_val) to ensure non-negative vals.
right = max(0, dfs(node.right)) }
res = max(res, node.val + left + right) ← considering the current node as the root of the path.
return node.val + max(left, right) ← for paths through parent nodes.

IX) Serialize and Deserialize Binary Tree (LC 297)

You see this problem is composed of 2 parts →

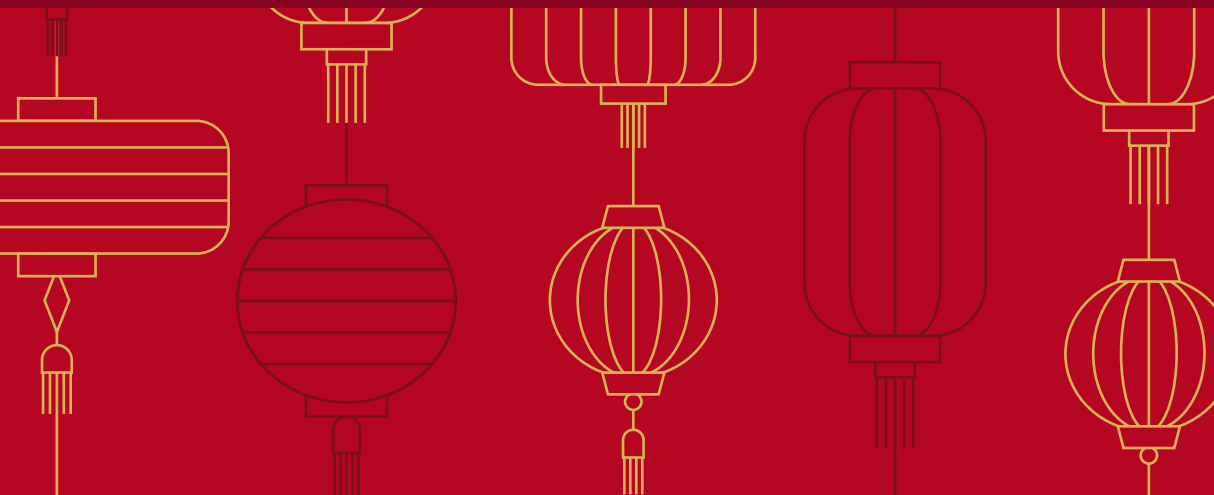
1) Serialization → storing the tree in an ordered manner to later de-serialize. Now, we are already aware of a couple of tree traversal orders. We've also solved a question wherein we re-created a tree based on its Inorder & Pre-order traversal. So, clearly the question can be easily solved with this understanding. The only point is that we don't really need 2 traversal orders to uniquely create a tree as long as we clearly mark the leaf nodes in the traversal order.

2) De-serialization - This step is all about unpacking the serialized tree data.

For the purpose of this question, I have used the pre-order traversal, clearly marking the leaf nodes with '#'.

It's a regular recursive dfs.

Backtracking



some important basics -

Backtracking like DP is what we can call, "Local Brute Forcing". So, to solve any backtracking problem you can use the following parameters to craft the code / recursive function -

* Define the Choice → the recursive fn must have a set of parameters which define a space of choices. While recursing over this space, we must narrow down (for convergence).

* Define Constraints → What causes a choice or a potential answer to be right / wrong / valid / invalid. Constraints can be included in the recursive function as base cases or out of range conditions.

↳ edit → I wrote rather unsatisfactory explanations for 3-4 problems that I have now erased. Backtracking can be tricky but I'll try to make things clear.
(Considering the following explanations are revised versions, I believe the expectations would be higher :'))

---

some general advice →

* Draw out trees to understand the algorithm you want. Might sound ridiculous but it's important. You see, most backtracking problems are intuitive to solve manually. Remember, it's local brute-forcing, it's not supposed to be very intricate (atleast in its initial versions). Drawing out trees corresponding to a problem will help you cover the cases & get a better grasp over the time complexity you might be looking at.

* Understand the General Flow of Backtracking →

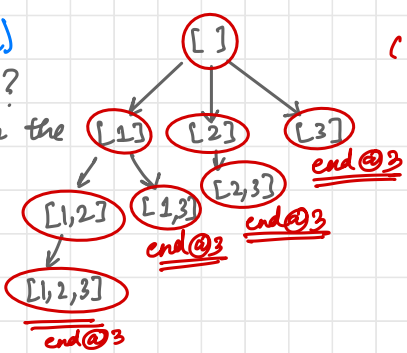
Backtracking problems are usually associated with trying a path until it's valid & then tracing our way back to the previously unexplored paths.

* You'll see patterns like append, backtrack, pop in list related problems or add, backtrack, remove in set related problems. This follows our pattern of trying, tracing our way back & trying again.

* Carefully craft the if conditions which serve as the required constraints to ensure validity. Crafting constraints intricately is what usually allows for optimization in backtracking. Don't lead with optimization though, narrow things down once you have a working solution.

LXI) Subsets (LC 78)

↳ manual method - Tree → (1)



(subsets of - [1, 2, 3]).

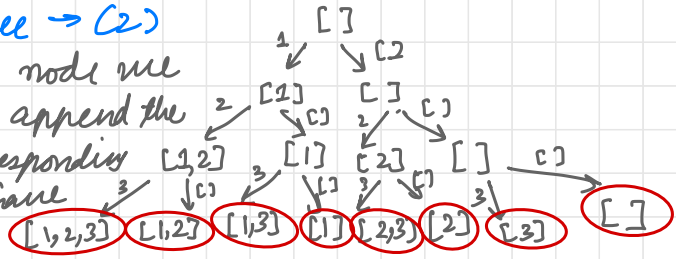
So, how can this tree be explained? Ignore the values, pay attention to the indices. Notice how each time a subset includes the last index we do not go any further. It's needless to say that the method works. In a nutshell here's what's happening → if the last element in our current subset corresponds to the i^{th} index, then the subsequent subsets can only have values corresponding to indices $> i$. So, when we hit the last index we're done. SIMPLE :).

The code is very simple, the main thing is to come up with this algo.

Time → $O(2^n)$ ← number of subsets, since each recursion yields a subset.

There can be numerous other algos to create all subsets. Here's another →

↳ manual method - Tree → (2)



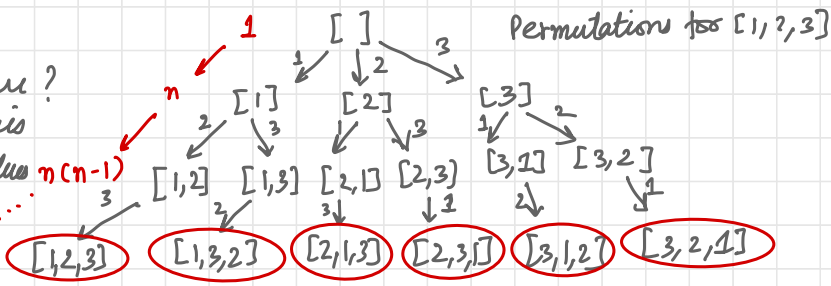
At each level for each node we either append or do not append the value at the index corresponding to the level. Once we have explored all elements we can end the tree. The desired subsets reside as the leaves of our tree. Again coding this up is very simple.

Time → $O(2^{n+1})$ ← look at the # nodes in the tree →

$1 + 2 + 2^2 + \cdots 2^n \rightarrow = 2^{n+1}$.

LXII) Permutations (LC 46)

↳ Tree →



So, what's happening here? along each branch of this tree we try adding values that haven't been used before. That's it. Once we have used all values, we have a valid permutation. In other words, at each level of the tree we decide for the value at the level & index in our current permutation.

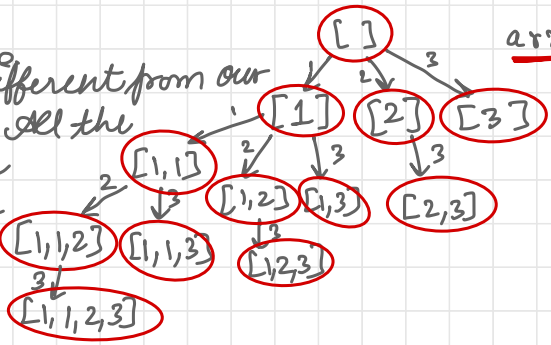
Time → $O(n!)$ → $1 + n + n(n-1) + n(n-1)(n-2) \cdots + n! \approx n!e$

$\approx O(n!)$.

LXIII) Subsets II (LC 90)

Subsets problem but with repeating integers. So, we'll just sort the array to ensure repeating values are adjacent to allow an easier handling.

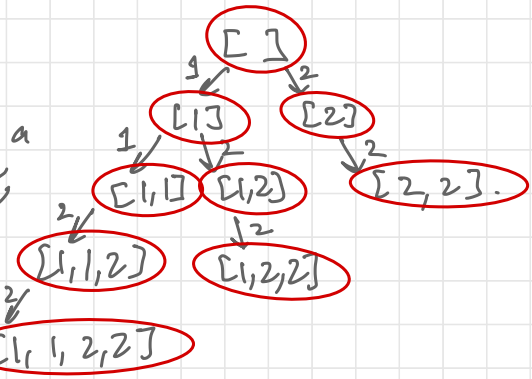
↳ Tree →



arr → [1, 1, 2, 3]

Notice anything different from our previous approach? All the backtracking algos we have used are modifications of dfs. Here, we do not give the second '1' an equal treatment. Notice how we did not create a separate branch for it. So, the algo is down to 1 simple addition, do not create a separate branch for a value if it's been dealt with before. Let's try another example to make this idea clearer.

arr = [1, 1, 2, 2]



Notice how we seem to follow a pattern of add first check later, i.e. we explore all remnant elements along a branch >= the previous element and create separate branches only if the next element is different. Implementing this in code is fairly simple. We'll just need a check. Time → same as subsets

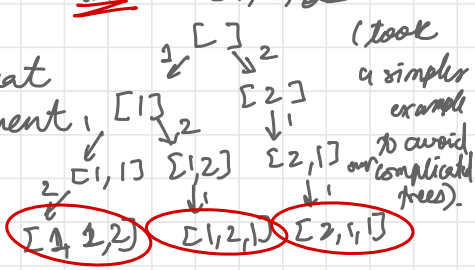
(XIV) Permutations II (LC 47)

Same idea but with Permutations.

↳ Tree → Quite simple really, the same concept we used for permutations, just that we will not branch over the same element (as we did with subsets).

Time → same as permutations

arr = [1, 1, 2]



(took a simpler example to avoid complicated trees).

XV) Word Search (LC 79)

A very simple backtracking problem with a few important takeaways. At what level must we check for our word on the grid? Well, we need to find a match of each alphabet in the right order. So, what all do we need → row, column (to locate where we currently are on the grid), index (to identify the alphabet we are looking for (ith index in the word)). We must also maintain a set of visited points on the grid along a path to ensure that we aren't looping.

That's basically it. Look at the code structure, that's essentially all there is to this problem. At each coherent element on the grid, we try L/T/R/B to find a path that can create the desired word. Ultimately, as the code is structured we also use a for loop to find the starting point & as soon as we hit a possible path, we return True.

Time Complexity → since the algorithm here has a couple of stopping points, the upper limit of the time complexity as denoted by Big O, isn't really good. Anyway, we have 4 possibilities from each value, one of which leads back to a loop for sure. So, 3 paths (again not always but a good upper bound). ⇒ $3^R \leftarrow$ (work (enqueue)) $\times$ $m \times n$ (m ↑ #rows, n ← #cols.)

for each node

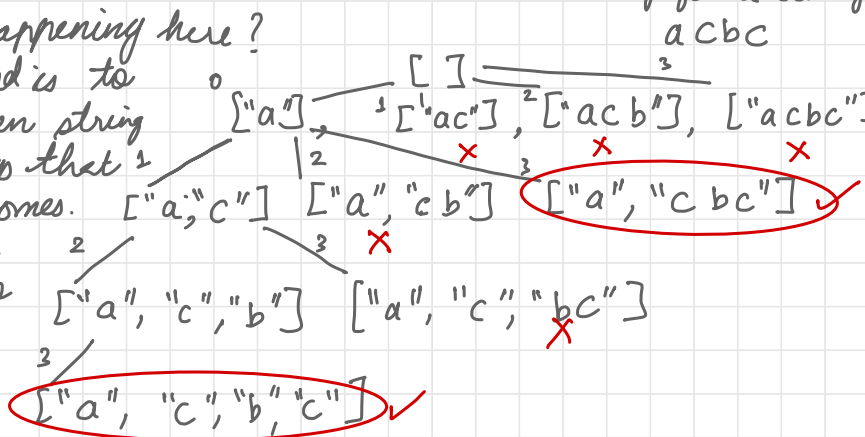
# of nodes to try as the first element.

XVI) Palindrome Partitioning (LC 131)

Again, let's start with a simple tree diagram understanding how this might be serviced.

↳ Tree →

say for a string - acbc



So what's happening here? The job at hand is to partition a given string into substrings that are all palindromes. The approach to do so is also quite simple. We start by creating the first partition; so, we choose b/w 0 & len(s) to find a partition point. Now, everytime we make a partition, we must ensure that the left half of the partition is a palindrome. If it is a palindrome, we explore further into that path, else we skip that path & move on to the other possible paths.

To do so in code → we have 2 params in our backtracking function → start & part. start represents the index until we which we have already explored the string for a path & part contains the palindromic substrings for a given path. It's quite simple really. We iterate over the remnant indices, i.e. [start, len(s)) in our function to find possible partition points. Each substring thus created is checked for being a palindrome. If it is a palindrome, we simply backtrack with the substring added to part & start → i+1.

Time → let's look at the tree → & say a branch operating with n elements in a string has T(n) nodes. We can see that →

$T(n) = T(n-1) + T(n-2) + T(n-3) \cdots T(1) + T(0) + 1$

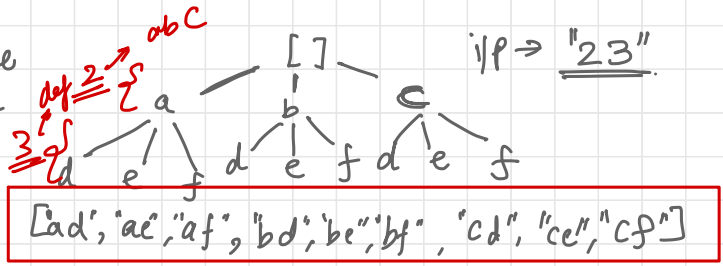
for $n \geq 2$ & $T(1) = T(0) = 1$ ⇒ $T(2) = 1+1+1 = 3$

in general → $T(n) = 2T(n-1) + 1$ for $n \geq 1$

⇒ $O(2^N)$

LXVII) Letter Combinations of a Phone Number (LC 17)

This is actually one of the easier problems. All you need to do is create a mapping of digits to their corresponding alphabets and brute force the life out of this. So, in the code, we'll recurse over the digits & iterate over the chars corresponding to each digit to get all possibilities.



Time = $O(4^N)$ — N = # of digits; # of possible chars mapping to a single digit.

LXVIII) Matchsticks to Square (LC 473)

There are a couple of observations that make this problem quite simple to solve:-

1) sum of all sticks should be divisible by 4 (# of sides in a square)
2) sum//4 = target length of every side.
3) The problem effectively boils down to creating 4 equally sized buckets of size = target.
4) This also means that we effectively need to choose which bucket to put a matchstick in (each matchstick has 4 choices).

So, what are going to be the params we parse into our backtracking function? Along every path we must track the value we want to find a bucket for & the current occupancy levels of the buckets along the path we are exploring. Thus, the code. It's fairly simple to understand.

You'll notice that there is no condition on sides[i] = 0, i.e., because we can be sure that if every element has been added & the overall sum % 4 = 0 & no bucket is overfilled the remnant capacity of each bucket will be zero. In other

words, the length of each side = target. ⭐ There's an important bit of optimization here → reverse sort the array. This ensures that buckets get filled faster & we're able eliminate quicker.

Time - $O(4^N)$ → # match sticks.
↳ # possible buckets

(Checkout LXX for answering this in 2^N time).

LXIX) Maximum Length of a Concatenated String with Unique Characters (LC 1239)

The question sounds fairly simple, doesn't it? We effectively need to find the longest subset of strings that do not share a common character. We know how to create subsets, how to ensure no overlap? Again, pretty straightforward, use Counters (frequency hashmaps). So, before adding element to our subset, we just need to check whether there is any repetition in the string or union of the existing string & the current string. Each time we go over the f^n we update the max length if required.

Time - $O(2^N)$ (subsets).

LXX) Partition to K Equal Sum Subsets (LC 698)

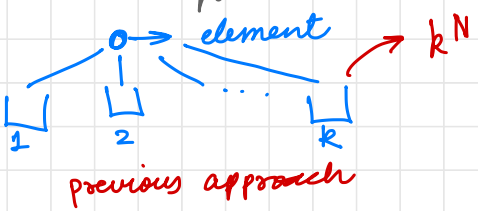
↳ observation → This question is actually just a generalization of the matchsticks to square problem. Here, instead of creating 4 equally sized buckets, we need k. Thus, the time complexity associated with that framework in mind = k^N.

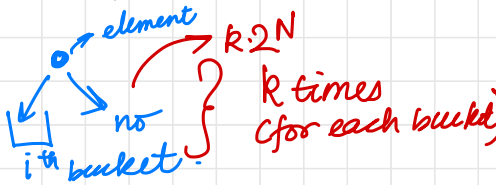
This is where framing the problem comes into play. Can we frame the problem differently? Right now, the problem is down to creating k buckets of equal weight using the elements provided. Can there be an alternate, possibly better

approach that can help us beat the time complexity of k^N? As it turns out, Yes! Moreover, it's simpler than you might think it is.

How about instead of allotting each element to one of the k buckets along each path, we just focus on creating k buckets with the elements provided; separately.

Might sound confusing, here's a simple illustration to drive this point home. →





↳ code → here's how we can code this up - along each path, we'll need to track → element being used (index), remaining empty buckets, currentBucketSum (subset sum), used elements. The rest is fairly simple, whenever subsetsum = target sum, move on to the next bucket, when remaing = 0; we can return True for the path. if subsetsum exceeds the target sum, abandon the path.

↳ again a couple of important optimizations for passing the time limit →

1) Reverse sort (to eliminate quicker)
2) If an element is not used for a bucket, then we can safely avoid trying the subsequent elements of the same value as well. (the first if cond^n in the for loop).

Time → $O(k \cdot 2^N)$

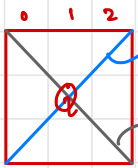
LXXI) N-Queens (LC51) → not really difficult though

So, we're given an $n \times n$ board & we must fit N queens on it such that none of them attack each other.

Here are a couple of obvious observations →

1. Each queen must occupy a separate row. → easy
2. Each queen must occupy a separate column. → easy
3. Each queen must occupy separate diagonals. → how to label diagonals?

Let's figure out how to label diagonals. Each queen owns 2 diagonals, let's call one positive & one negative.



→ positive diagonal $(r+c) = 2$.

→ negative diagonal $(r-c) = 0$

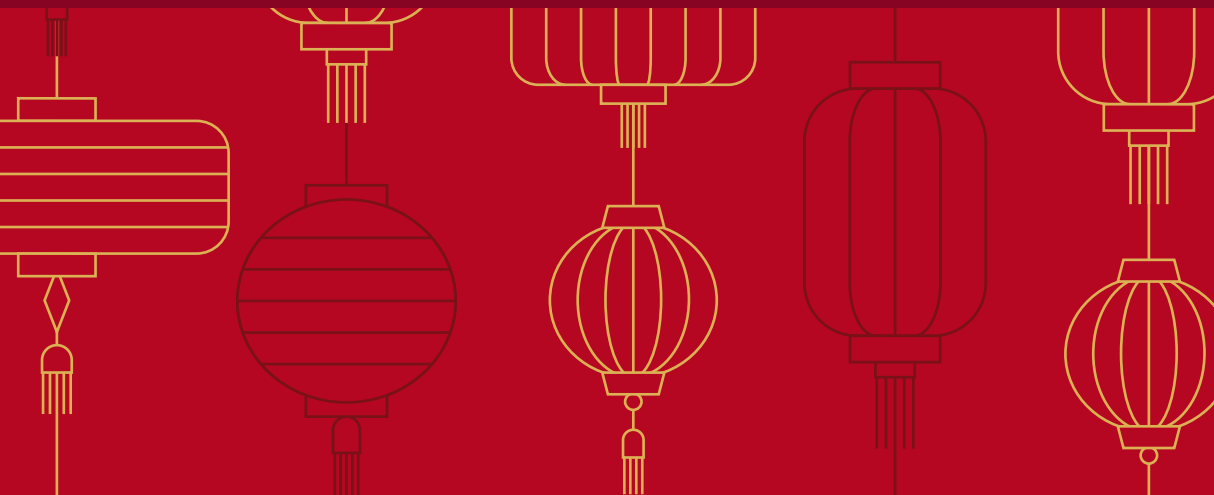
* Each positive diagonal (/) can be uniquely identified by $(r+c)$ & each negative diagonal (\) can be uniquely identified by $(r-c)$.

So, here's how we'll approach the solution → Each time we add a queen, we'll automatically add it to the next row & to make sure we have different columns & diagonals, we'll track the used cols & diagonals. So, in our backtracking f" we'll have the → row # (or the queen #), cols used (set), posdiagonals used (set), neg diagonals used (set), board (the current positioning of queens).

That's it :)

Time → $O(N^2)$. → N queens residing in N different cols.

1-D Dynamic Programming



Some important ideas surrounding DP →

↳ DP can be thought of as an optimized approach towards recursion / backtracking problems. The idea is very simple; do not explore the paths you've already explored; memorize the answers instead.

↳ so, after exploring our requirement/s along a particular path; we shall save its answer in a memo & use it later if we land on the same point through some other way, we can quickly plug-in the answer we know instead of re-doing the entire thing.

The SRTBOT approach to solve any DP problem →

S 1. **Subproblem** definition subproblem $x \in X$

- Describe the meaning of a subproblem **in words**, in terms of parameters
- Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
- Often multiply possible subsets across multiple inputs
- Often record partial state: add subproblems by incrementing some auxiliary variables

R 2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \ldots)$ for one or more $j < i$

- Identify a question about a subproblem solution that, if you knew the answer to, reduces the subproblem to smaller subproblem(s)
- Locally brute-force all possible answers to the question

T 3. **Topological order** to argue relation is acyclic and subproblems form a DAG

B 4. **Base** cases

- State solutions for all (reachable) independent subproblems where relation breaks down

O 5. **Original problem**

- Show how to compute solution to original problem from solutions to subproblem(s)
- Possibly use parent pointers to recover actual solution, not just objective function

T 6. **Time** analysis

- $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = O(W)$ for all $x \in X$, then $|X| \cdot O(W)$
- $\text{work}(x)$ measures **nonrecursive** work in relation; treat recursions as taking $O(1)$ time

* using @cache decorator for dp functions - In most of my code, I have used a hash map / dict (self-memo) for the purpose of memorization. However, instead of using this, @cache decorator can also be used. It is based on LRU cache & serves the same purpose as a memo while being more efficient on time and slightly worse on memory. So, for problems involving a clear topological order of computation it's better to use @cache.

LXXII) House Robber (LC 198)

Given an array of values, let's compute the max loot that can be collected beyond every index, that way we can compute the maximum robbery value from the first value (0 index). You see if we incrementally compute this value from every index starting from the last index, we can quickly arrive at the required value. This is exactly what you shall find in the SRTBOT approach for this problem.

S→ suffix, robMax(i): max loot starting from the ith index.
R→ robMax(i) = max(nums[i] + robMax(i+2), robMax(i+1))
↑ loot the ith value ↑ do not loot it & move on.
T→ descending (useful for non-recursive dp).
B→ robMax(i) ; i >= len(nums) = 0.
O→ robMax(0)
T→ 1 x N = O(n)
Time→ O(n); space→ O(n) (memo for every i in nums).

LXXIII) House Robber II (LC 213)

Just a simple modification of the first problem. There's one added constraint as a part of cyclic condition, i.e., the robber cannot have both the first & the last house. So, just find the max loot in nums[1:] & nums[:-1] & take a max of both of them.
Time→ O(n); space→ O(n)

LXXIV) Longest Palindromic Substring (LC 5) *

First let's examine the utility of DP in this problem. Atleast, I didn't come up with any DP related solution for this problem. It's related to 2 pointers.

There's a very obvious solution to this problem:- Test every substring for being a palindrome & if it is one, just compare its length to the existing max & make the necessary updates. This approach however, takes an $O(n3)$ time. n^2 substring & $O(n)$ to examine every substring for being a palindrome. Can we do better?
Of course we can! Instead of iterating in a nested fashion to create substrings & then checking them, let's only create substrings that are palindromes. How can that be done? Let's iterate the array considering every element as the center of a possible odd length palindrome & then continue adding vals. to its left & right if they are equal. This is obviously just for odd length palindromes, We can do the same thing for even length palindromes. Consider the centers as the i^{th} & $i+1^{th}$ indexed values of the even length palindrome & repeat the same left/right addition of values until they are equal. Once you understand this, the code is really easy to comprehend.
Time → $O(n^2)$; Space → $O(1)$

[XXV) Coin Change (LC 322)

Seems like a simple enough problem. Let's try to break it down into some tangible coding problems Let's store a memo containing the min. number of coins required to attain a specific target. For the denominations provided, this value = 1. Thus, we know have a workable definition for our subproblem.
S- dp(target) → min number of coins required to get to the input parameter target.
R- dp(target) = 1 + min(dp(target-coin) for coin in coins)
T- Ascending order of target, i.e. we'll build up from the targets being the given denominations, ultimately creating comb"s summing up to the required amount.
B- dp(0) = 0 & dp(coin) = 1.

O - dp(amount)
T - O(mn); m → # denominations, n → target.

This problem is quite simple once you are able to get to the subproblem. However, we vary of the edge-cases. What if it's not possible to create the amount required from the denominations provided? In that case, we have dealt with the problem by returning ∞ as the o/p of the dp function. All paths leading to the target being '-' are not valid, thus we shall return ∞ for them. Finally, we verify the o/p of the function as a finite value.

Time → O(m n) ; Space → O(n)
m → # coin denominations; n → target. ; n → target.

LXXVI) Maximum Product Array (LC 152) ⭐

This is not a dp problem. You see, dp is a fancy way to brute force. Thus, there might always be a fancier & more efficient greedy sol" to solve a problem. Brute forcing here is very simple, create all possible subarrays & compare their products to ultimately return the maximum value. This would be a quadratic solution. However, if we dig in deeper there are some beautiful patterns waiting to be observed.

So, the issue is that we have negative values. What makes negative values extremely important is, 2 negative values make a positive value. So, let's track 3 things -

1) maxprod 2) currMax 3) currMin.
- maxprod → the max. subarray product until the i^{th} element.
- currMax → the max. subarray product achievable using the i^{th} element.
- currMin → the min. subarray product achievable using the i^{th} element.

Let's start linking these 3 values.
What are the options for currMax as per the def"?
max(currMax * num, currMin * num, num)

i.e., continue the subarray of max, or min, or start a new one.
similarly for currMin = min(currMin * num, currMax * num, num)
Finally, the maxprod value until num = max(maxprod, currMax).

Seems very simple but it's trickier to arrive at than you might imagine. It's also very important to deal with 0s because they can ruin it! With a '0' in our subarray, we have effectively neutralized all other values. So, we can reset our currMax & currMin values, each time we encounter a '0'. If 0 is indeed the max subarray prod, it must be max. value in the array provided. Thus maxprod will have already been initialized to 0.

Time → O(n); space → O(1)

LXXVII) Word Break (LC 139)

what is this question really asking? One way of looking at it is to consider breaking the given string to valid substrings. This "validity" is determined by the presence of that substring in the dictionary provided. So, we're essentially backtracking. So, let's define the key params associated with this problem.

S — backtrack(i) → stores the answer to → is it possible to segment the string s[i:] into valid substrings?

R — backtrack(i) = backtrack[i:j] and backtrack[j:]
$i < j \leq len(s)$. We'll choose j such that s[i:j] is a valid word in the dictionary. There might be many such j's, thus many such paths (backtracking).

T — descending order of i.

B — backtrack(i) for $i >= len(s)$ = True.

O — backtrack(0).

T — O(m n); where m = # words in dict & n = len of string s.

Time → O(mn) ; space → O(n)

# of words in dict ↙ ↘ len(s)

LXXIX) Partition Equal Subset Sum (LC 416)

Again, a fairly straight forward problem to assess. We've solved similar problems in backtracking before. Here, we also get to leverage the memoization. So, since all elements must be involved, the array sum must be divisible by 2. If so, we are essentially down to looking for a subset that adds up to sum(arr)/2. So, our problem has boiled down to finding a subset that adds up to the required target of sum(arr)/2.

S- partition(i, target) → returns the possibility of finding target as a subset sum from the array nums[i:] (suffix).

R- partition(i, target) = partition(i+1, target) or partition(i+1, target - nums[i]).

T- i → decreasing; target → increasing.
B- partition(—, 0) = True; partition(>=len(nums), —) = False
O- partition(0, sum(nums)/2).
T- $O(n^2)$ → to be more precise → O(len(nums) * sum(nums))

time → O(len(nums) * sum(nums)); space → O(len(nums) * sum(nums)) or simply $O(n^2)$.

→ past trick :) (might be a bit rusty).

LXXX) Longest Increasing Subsequence (LC 300)

This problem can be boiled down to a suffix maximization problem. The relation is also extremely simple. Basically, each index can be thought of as the starting point of the longest increasing subsequence. So, at the end, in our memo, we'd be left with with the max increasing subsequence length starting from each index. & then we can just take a max over these values to arrive at our answer.

S- func(i) → length/max. subsequence starting from the element @ index i.

R- func(i) = max(1+ func(j) | for j > i & nums[j] > nums[i])
↳ done using a simple for loop.

T- descending (the final part of the code leverages this order to optimally compute the values & compare them → the reverse for loop).

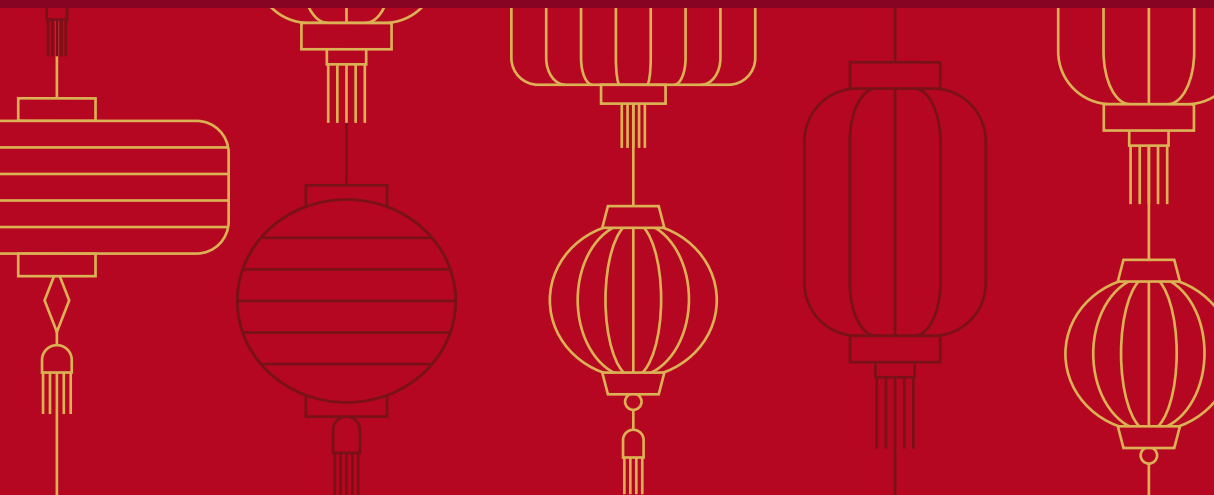
B- func(i) ; i >= len(nums) = 0.

O- max(func(i) ; for i in range(len(nums)).

T- N x N = O(n²).

time → O(n²) ; space - O(n).

2-D Dynamic Programming



2D dynamic programming simply means recursing & memoizing over 2 parameters as compared 1 param in 1D dp. As the number of params increase, the scope for optimization increases. So, we need to be wary of the parameters selected.

LXXXI) Unique Paths (LC 62)

This is a beginner friendly 2D dp question. The problem is self-explanatory & requires very little effort.

S- dp(i,j) → the number of distinct paths from (i,j) leading to (m-1, n-1).
R- dp(i,j) = dp(i+1,j) + dp(i,j+1) (bottom or right).
T- descending (m-1, n-1) → (0,0).
B- dp(m-1, n-1) = 1.
O- dp(0,0).
T- M x N = O(mn).
Time- O(mn) ; Space- O(mn).

LXXXII) Longest Common Subsequence (LC 1143)

This is a fairly simple problem once you understand that it's all just down to identifying the common characters in sequence. Take 2 pointers - i for text1 & j for text2. Compare the characters at these indices (i & j) in text1 & text2 respectively. If the characters match - search for the next match in text1[i+1:] & text2[j+1:] else fix either one of these indices & recurse, i.e. max(f(text1[i:], text2[j+1:]), f(text1[i+1:], text2[j:])).

S- dp(i,j) → the length of the longest common subsequence for text1[i:] & text2[j:]. [suffix]
R- dp(i,j) = 1 + dp(i+1, j+1) if text1[i] == text2[j] else max(dp(i+1, j), dp(i, j+1)).

T- descending for both i & j
B- dp(i, j) if i >= len(text1) or j >= len(text2) = 0.
O- dp(0, 0).
T- len(text1) x len(text2) = O(n²).
Time & Space - O(len(text1) x len(text2)) ≈ O(n²)

LXXXIII) Best Time to Buy and Sell Stock with Cooldown (LC 309)

Again, a fairly easy problem. Just figure out the modes you can operate in. In terms of actions you can choose amongst buying, selling or N/A. However, you can only buy the stock if you haven't already bought it. Similarly, you can only sell the stock if you own it. In all, there are 3 modes you can find yourself operating with →

	canBuy	canSell	
when you don't own the stock →	True	False	①
when you own the stock. →	False	True	②
cooldown after having sold the stock. →	False	False	③

} one of these 3 modes will be a provided as a part of each recursive function call.

S- dp(i, canBuy, canSell) → the max. profit that can be generated from the ith day given the mode.
R- if canBuy (mode 1)
dp(i, canBuy, canSell) = max(dp(i+1, False, True) - price[i], dp(i+1, True, False)) [between buying & N/A]
elif canSell (mode 2)
dp(i, canBuy, canSell) = max(dp(i+1, False, False) + price[i], dp(i+1, False, True)) [between selling+cooldown & N/A]
else (mode 3)
dp(i, canBuy, canSell) = dp(i+1, True, False) [cooldown]
T- descending for i & oscillating b/w True/False for canBuy & canSell. However, this variation only limited 3 modes per index.

B- dp(i, canBuy, canSell): i >= len(prices) ⇒ 0.
O- dp(0, True, False)
T- 3 x len(prices) = O(3n) ≈ O(n)
Time & space- O(3n). ≈ O(n)

LXXXIV) Coin Change II (LC 518)

This problem is slightly tricky to absorb because every path leading to the target must be unique. What does uniqueness mean in this context? Put simply, the frequency map of denominations used in each possible path leading to the target must be unique. The next question is how do we impose this uniqueness constraint? How do we ensure that 1, 2, 1; 1, 1, 2; 2, 1, 1 are all counted as 1 possible path. The answer is quite simple → allow an ordered choice! The only path allowed here would be 1, 1, 2. We shall impose an ascending order on the paths to screen out the unique ones. Mind you the ascending order is not required on the denomination value, just the index.

S- uniqueChillar(i, target): number of unique ways to create the target using coins[i:] subarray.
R- uniqueChillar(i, target) = uniqueChillar(i, target - coins[i]) + uniqueChillar(i+1, target)
T- descending for i & ascending for target.
B- uniqueChillar(i, 0) = 1. uniqueChillar(>= len(coins), target) = 0. uniqueChillar(i, < 0) = 0.
O- uniqueChiller(0, amount)
T- O(len(coins) x target) = O(mn).
Time & space- O(mn)

LXXXV) Interleaving String (LC 97) ⭐

Now this problem is easy to solve but hard to understand. The first time I read the 'abs(n-m)<=1' bit, I was thrown off. However, all it means is that the substrings must be interleaved. If you interleave 2 strings this cond" automatically holds true! so once you've made your peace with this idea, the problem just boils down to defining a straightforward dp function provisioning the required sol".

S- interleave(i,j) → is it possible to interleave s1[i:] & s2[j:] to create s3[i+j:]? (Bool)

R- if s1[i] == s3[i+j] and s2[j] != s3[i+j]:
interleave(i,j) = interleave[i+1,j]
elif s2[j] == s3[i+j] and s1[i] != s3[i+j]:
interleave(i,j) = interleave(i,j+1)
elif s1[i] == s3[i+j] and s2[j] == s3[i+j]:
interleave(i,j) = interleave(i+1,j) or interleave(i,j+1)
else
interleave(i,j) = False

(It's all down to matching the characters from s1 & s2 with s3).

T- descending for both i & j.

B- interleave(i, >=len(s2)) = s1[i:] == s3[i+j:].
interleave(>=len(s1), j) = s2[j:] == s3[i+j:].

O- interleave(0,0)

T- len(s1) x len(s2) = O(mn)

Time & space = O(mn).

LXXXVI) Longest Increasing Path in a Matrix (LC 329)

Doesn't seem like a hard problem for sure. The first thought that came to mind after reading the question was, how do I avoid wrap-arounds? The answer is quite obvious - there can never be a wrap-around if you only allow a strictly increasing path. So, that's a load off. Subsequently, the idea behind the sol" is down

a simple dp function which can compute the length of the longest increasing path from a pt. (r, c) in the matrix provided.

S- dp(r, c, prev) → length of the longest increasing path from pt. (r, c) given the previous value prev.
R- dp(r, c, prev) = max(r±1, c±1, matric[r][c]) (LRTB)
T- We'll compute it in the for loop (after the dp function) in ascending order for both rows & columns
B- dp(r, c, prev) = 0 if r<0 or r>=m or c<0 or c>=n or matrix[r][c] <= prev
O- max(dp(r, c, -1) for (r,c) in matrix)
T- m x n = O(mn)
Time & space = O(mn)

LXXXVII) Distinct subsequences (LC 115) (* @ cache used)

Again, not a particularly hard problem given that you know it requires the DP treatment. We'll just try to match the characters in s with t in different ways to compute the no. of possibilities.

S- dp(i, j) → # of distinct subsequences of s[i:] equalling t[j:].
R- dp(i, j) = dp(i+1, j) + [dp(i+1, j+1) if s[i] == t[j]]
T- decreasing order of i & j.
B- dp(i, >=len(t)) = 1. dp(>=len(s), j) = 0.
O- dp(0, 0)
T- len(s) x len(t) = O(mn).
Time & space - O(mn).

Elaborating on the R part → if the present indices match we can look for t[j+1:] in s[i+1:]. Irrespective, in our quest to explore all paths, we must try looking for t[j:] in s[i+1:]. Ex- s="rrabbit" & t = "rabbit". We can do it in 2 different ways as t[0:] exists in s[0:] & s[1:].

LXXXVIII) Edit Distance (LC 72)

you're got 4 options to explore each time - insert, delete, replace or let it be. These 4 options must be looked at for each char in word1. It's all about character matching (yet again :))

S- dp(i,j) → the min no. of operations required to convert word1[i:] to word2[j:].
R- dp(i,j) = dp(i+1,j+1) if word1[i] == word2[j] else
1 + min(dp(i+1,j), dp(i+1,j+1), dp(i,j+1))
choosing amongst delete, replace & insert if the chars don't match
T- decreasing for both i & j
B- dp(i, >=len(word2)) = len(word1)-i (delete the rem. chars)
dp(>=len(word1), j) = len(word2)-j (add the rem. chars).
O- dp(0,0)
T- len(word1) x len(word2) = O(mn)
Time & space = O(mn).

LXXXIX) Burst Balloons (LC 312)

This is a genuinely hard problem to solve optimally. Before explaining the optimal sol^n, let me first talk about a much more intuitive approach. This is the approach that struck me first. The idea is pretty simple, let's parse the entire array in our dp function & run through the possibilities of popping each balloon in the array parsed. Something like this →

```
def dp(arr):
    if not arr:
        return 0
    elif len(arr) == 1:
        return arr[0]
    else:
        res = 0
        for i in range(1, len(arr)-1):
```

redundant conds as they will never hold true as arr always atleast has the [-1]+[1]. Regardless :)

```
        res = max( res, dp(arr[:i] + arr[i+1:]) + arr[i-1] x
                        arr[i] * arr[i+1])
    return res
return dp([1] + nums + [1])
```

Now, this function when memoized will have a time complexity of 2^n, because we are effectively computing the max. amt. to gain from all possible subarrays with this code. We will be memoizing over all subarrays. So, a time complexity of 2^n & a ridiculous space complexity of $n \cdot 2^n$. (check the calc.)

$$n + {}^nC_1(n-1) + {}^nC_2(n-2) + {}^nC_3(n-3) + {}^nC_4(n-4) \cdots$$
$$= n\left(1 + n + {}^nC_2 + {}^nC_3 \cdots {}^nC_n\right) - \left(1 + n + 2\,{}^nC_2 + 3\,{}^nC_3 + 4\,{}^nC_4 \cdots\right)$$
$$= n \cdot 2^n - n \cdot 2^{n-1} = n \cdot 2^{n-1}$$

so, clearly we must aim for a better solution.

Optimal sol^n →

There are a couple of rather counter-intuitive steps required to optimally solve this problem. The first one is, while defining the subproblem (as part of our beloved SRTBOT setup) we shall consider using a range. Usually for one-dim objects we tend to use a single pointer, employing a prefix/suffix subproblem; however we require a range to get the job done. The second step is the real brain-teaser. While relating our subproblems, instead of choosing the balloon to pop first, we shall choose the balloon to pop last. This is imperative to help us create splits in the range without worrying about affecting the subsequent computations.

subproblem – 

(subarray defined by the range $[i, j]$).

Now, within this range we must choose an idx k, such that the balloon at the k^{th} idx must be popped last in the range defined by $[i, j]$.

So, 

: assuming that k is popped last, we can now write the rel^n in very simple terms

dp(i, j) = max(nums[i] x nums[k] x nums[j] + dp(i,k) + dp(k,j)) for k in range(i+1, j). ↳ ∵ k is popped last it shall be surrounded by i & j at the end

also, ∵ k is popped last, we can split [i:j] into [i:k] & [k:j] without worrying about the elements in b/w as their boundaries remain unaffected. You see this only holds true only if the k^{th} balloon is popped last in our range of [i, j]. Had we chosen the first balloon to pop, we would have never been able to split the array into 2 parts like we did here. Take some time to let this settle in. A simple change in the def^n, provides us such a simple path. Also, we'll modify the nums array slightly by adding 1s to both ends to ease our work for the edge balloons.

S - dp(i, j) → the max. amt possible from nums[i+1 : j]

R - dp(i, j) = max(nums[i] x nums[k] x nums[j] + dp(i,k) + dp(k,j)) for k in range(i+1, j).

T - increasing order of (j-i).

B - dp(i, i) = 0.

O - dp(0, len(nums)-1) (remind you this nums = [1] + nums + [1])

T - n x n x n = O(n³)

i, j, k.

Time - O(n³). Space - O(n²).

XC) Regular Expression Matching (LC 10) ★

Back to character matching! The '*' part of this problem, makes it hard in my opinion. So, let's cut to the chase & solve this problem! The subproblem definition shouldn't even be a point of concern at this point. We'll keep a suffix based def^n for both s & p. Now, there are 2 magic chars → '.' & '*'. '.' (period) is pretty simple to deal with. We can match it with the corresponding character in s. The asterisk wields all the power in this question. So, let's closely examine this '*'. It can do 2 things either skip/del

the character before it or repeat the character before it indefinitely. There are a couple of cases that should be clarified.

① what if '*' is the first char in p? → Then you can skip it.
② what if a period occurs before an asterisk, i.e '.*' → Then you can repeat any char you want indefinitely, ∵ '.' can be used to match any single char.

so, with this out of the way. Let's get back to SRTBOT-

S- dp(i, j) → Possibility of creating s[i:] using p[j:]
R- if j < len(p)-1 and p[j+1] == '*':
    ans = dp(i, j+2) # skip the preceding char & head on
    # to the char after the '*'.
    if i < len(s) and (p[j] == '.' or s[i] == p[j]):
        ans = ans or dp(i+1, j) match the char with s allowing the same char to be matched again
    dp(i,j) = ans.

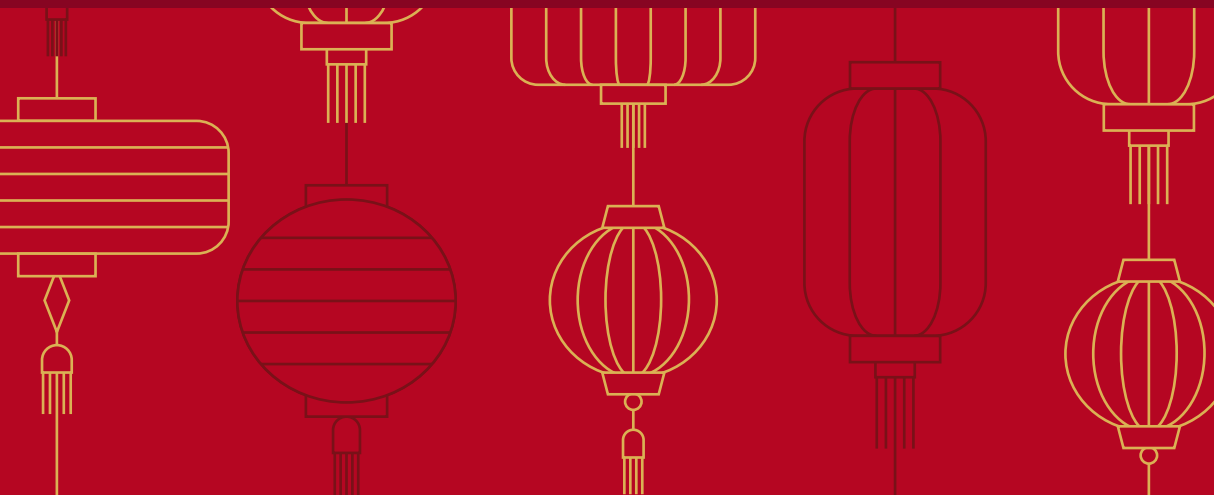
elif i < len(s) and (p[j] == '.' or s[i] == p[j]):
    dp(i,j) = dp(i+1, j+1) #match char & look fwd.

else:
    dp(i,j) = False

T- decreasing order for i & j.
B- dp(i, len(p)) = i == len(s). (if j has seen its course, i should have done so too)

O- dp(0,0)
T - len(s) x len(p) = O(mn).
Time & space → O(mn)

Graphs



While solving problems related to graphs, you'll find yourself using a lot of standard techniques. Here are a few of them →

1) DFS / BFS (for traversal)
2) Hashing (to keep track of the nodes you've visited)
3) *Union-Find (to work with disjoint sets).
4) *Topological sort (scheduling tasks where dependencies are involved)
5) *Dijkstra (for single source shortest path with '+' edge weights)
6) *Bellman Ford (for single source shortest path (general)).

Now, I could have solved questions for each of them under different sections but a piece of the puzzle is to figure out the category of the problem. So, let's shuffle things up.

XCI) Number of Islands (LC 200)

Seems like a straight forward problem. How would you do it manually? Well, you might start at a '1' most likely at positions close to grid's boundary and you will go LRTB from that point in search for 1s & if you encounter any 1s in the process you shall do the same with them. In order to ensure convergence you will also maintain a list of the values you've visited (to avoid looking at them again). So, you'll do this until you encounter new 1s within the grid & the moment you can't find anymore, you'll stop & have an island to yourself. This in a nutshell is how you will go about isolating islands. Do this for all the 1s in the grid & you can get a count! So, this problem is effectively around graph traversal + hashing.

DFS / BFS + Hashing → Maintain a set, visited to keep track of the (r,c) values explored in our grid. From each 1 you encounter go LRTB. I prefer a recursive DFS. The function doesn't need to return anything. Everytime it runs its course

the visited set gets updated so that we don't over count our islands by starting at a previously visited island point, and obviously each time we update the island count to +1. So, run a nested for loop over the grid (r, c) & each time you encounter a new 1, run the traversal function & increment the count.

Time - O(m*n); space - O(m*n)

XCII) Clone Graph (LC 133)

Creating deep copies of objects is an important concept to understand. The requirement of the problem is almost immediately obvious. The procedure to do so maybe not so obvious immediately. Usually while addressing the question of deep copies in mutually connected objects (here, nodes of our graph), we must create some sort of mapping b/w the old nodes & their new copies. This is the only idea required to solve this problem.

DFS / BFS (Graph traversal) + Dictionary (dict[oldNode] = newNode)
mapping

So, use a graph traversal algorithm. Again, I tend to prefer DFS. We have a class Node to initialize every node to its respective value and create an array of neighbours for it. So, each time you encounter a node not in your nodesDict, you create one. Initialize the copy to the node's value & an empty neighbours list. Subsequently, iterate over the neighbours of the og node & repeat the process for them, finally append the copies of the neighbours to the neighbours list for the og node.

↳ I usually prefer DFS because Graph problems tend to lend themselves to recursion quite easily.

Time → O(V+E); space → O(V+E)

XCIII) Pacific Atlantic Water Flow (LC 417)

So, you've been given a grid and 2 oceans. You must identify the points on that grid which can reach both oceans (the cond^n to do so is based on the relative height of a pt. wrt its neighbours). Now, there are multiple ways to solve this question. My approach for this question seems slightly unconventional but is very intuitive once stated. Instead of seeing for each point whether it can reach any ocean, let's see which points on the grid are accessible by the edge-points (which we know can reach their respective oceans).

DFS (Traversal) + Hash pts for Pacific & Atlantic:- One key observation for this question is that determining whether a pt. can reach an ocean is PATH-DEPENDENT. Since the parameter to determine the ocean accessibility of a pt is "relative height" it might be possible that even if one path doesn't work but another one might. Consider this →

1	5	7
4	3	2
8	6	4

assume, so far we have determined the red-dotted pts are accessible by an ocean.

Now, if I were to head down from 5 → 3. As per the algo.. 3 >= 5 is false. I can't reach 3 from 5. The imp. bit is that I cannot yet comment on whether 3 is accessible. I must try to reach 3 from all directions & that's when I'll have a conclusive remark on accessibility. Here, 3 is accessible via the neighbouring 2. Thus, if we had to track all unique ways, we need to store (r, c, prev), i.e., row, col. & prev. value. So that besides tracking pts visited (r, c), we also track the paths used to visit them, in this case the value of its immediate predecessor serves our purpose.

After having this epiphany of unique (r, c, path) values you realise that there is no sense in hashing these values, as they covers all the possibilities. So, you can just run the for loop (LRTB) from every accessible pt. :D. So, what's the plan? Create 2 sets for pacific & atlantic respectively. Store the pts accessible for both sets of oceans & finally take their intersection.

Time - O(m * n), space - O(m * n).

XCIV) Surrounded Regions (LC 130)

This question is kinda similar to the previous one (pacific-atlantic) you see even here a slightly skewed approach tends to make more sense. The definition of a surrounded island is very strict, i.e; 4 directionally surrounded by 'X's. However, let's look at the flip side → it's pretty easy to determine which 'O's are unsurrounded (not surrounded).
If you give it some thought, the unsurrounded 'O's must be part of islands with 'O's at the edge / boundary of the grid. That's essentially, the only way a 'O' will be unsurrounded.
So, let's use this!
DFS + Visited Hashing → Let's start at the boundary 'O's & traverse inwards to identify other 'O's which are a part of their island & keep track of the visited/unsurrounded 'O's. Lastly, iterate over the grid & convert all the not visited/surrounded 'O's to 'X's. Simple!
Time - O(m*n); space - O(m*n).

XCV) Rotting Oranges (LC 994)

Now, this question is interesting. The requirement of the question is simple - It's almost like a chain reaction. However, unless you've solved similar problems, it takes some time to arrive at the solution. Let's revisit the requirements. 0 → empty, 1 → fresh, 2 → rotten. So, the first thing that you'll notice is that multiple rotten oranges can spread their rottenness simultaneously. So, we have some rotten oranges, let's call them patients 0, and let's call their impacted oranges, patients 1, & so on... patients 2, patients 3.... Observe, how patients X, all shall operate simultaneously to spread their rottenness. This rottenness seems to be travelling in a BFS manner, i.e., degree by degree or layer by layer. Thus, the traversal algorithm will be BFS.

BFS encapsulated in a for loop (exhausting all elements in one layer (patients x) together). - So, we'll use the std BFS but the only twist would be that all elements in the queue at the beginning of every BFS iteration will be exhausted together. This is a technique which has already been used in Trees to measure the height.
We shall initialize our queue to the patients (i.e. the (r, c) values of '2's in the grid. Subsequently, we run a "multi-node BFS", each time we encounter a '1' that can be rotten, we append it to our queue and reduce the fresh count by 1. After 1 layer has been exhausted, increment the time taken. Finally, after running this BFS starting from a queue of the og 2s, we shall check if the fresh count is down to 0, if so, return time, else return -1 (∵ all fresh oranges could not be rotten).
Time - O(m*n); space → O(m*n) (queue for BFS)

XCVI) Open the Lock (LC 752)

So, 4 circular wheels spanning the 10 digits (0-9), a target to crack & a couple of deadends (comb"s which must not be dialed in). Our job here is to find the min. number of turns required to hit the target comb". If a wheel reads x, we can go to x-1 or x+1. Thus, with 4 wheels, we can arrive at 8 different combinations. I hope you can imagine the tree that gets created based on this idea. The question is which traversal algorithm to use? A rule of thumb, use BFS whenever you're asked for shortest paths. It's quite obvious why using BFS for such questions is always better than DFS. In a nutshell, DFS explores 1 path at a time, BFS on the other hand explores paths coming out of all nodes equidistant from the starting pt., thus, with BFS we can easily comment on the min./max. path lengths.

BFS + visited nodes hashing - This phrase would have served as the answer, had there not been as many nuances to this question. Firstly, how to get all the 8 permutations out of a particular node? The perms are easy. The issue is addressing the "circular wheel", where, 9→0 & 9←0 in a pretty way. The ans. is actually quite simple - $X_{n+1} = (X_n + 1) \% 10$. & $X_{n-1} = (X_n + 9) \% 10$. So, one problem out of the way. Finally, for deadends, we can simply hash the list of deadends & whenever we arrive at a comb", we can simply check if it's a deadend, only moving forward from that node, if it isn't one. There's a noteworthy point in this problem's code, besides storing the combination value in our BFS queue, we also store the number of turns to reach it. This approach serves as an alternative to the multi-node BFS approach, commonly used for finding the height of a tree or in questions like Rotting Oranges.

Time - $O(10^4)$, space - $O(10^4)$. 10^4 being the # of comb's possible

XCVII) Course Schedule II (LC 210) ★

This is a tricky question to solve correctly. The structure & ask of this question scream adjacency list! So, let's create an adjacency list, where corresponding to every course no; we shall store its list of dependencies. Now, there must be atleast 1 course in this adjacency list that has an empty list, because without an independent starting pt; we can't get the ball rolling. So, here's the first key idea for our solution→ courses with an empty list can always be taken up. So, as we begin to identify courses that can be taken up, we can update their dependency list to an empty list. This is not crucial to the solution, but just helps us be more memory-efficient. Next, as we begin identifying courses which can be taken up, we must hash them/store them in a set. So, that if they happen to be

dependencies of subsequent courses, we can easily add them to our list.

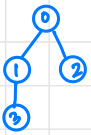
DFS + Included set (to keep track of courses that have been identified as "Possible") + Visiting set (to keep track of courses that we are visiting while exploring the dependencies of a course). - Pretty big header, eh? DFS ⇒ obvious. We must explore the dependencies of a node entirely. Included set ⇒ mentioned above. The second key idea for this question is ⇒ maintaining the visiting set. This set is important to guard us from endless loops or cycles. Imagine having the entries [1,3] & [3,1] in our i/p, i.e; 3 is a dependency of 1 & 1 is a dependency of 3. Such patterns make the question impossible to solve. So, we must maintain a set to ensure that we do not circle back to the initial courses while exploring their dependencies. The rest of the code is fairly straight-forward.

Time - O(V+E), space - O(V+E) (Basically, the size of our graph).

XCVIII) Redundant Connection (LC 684)

If you have n nodes, you need n-1 edges to connect all of them. Here, we have n edges, out of which we are supposed to return a redundant edge. The definition of a redundant edge is easy, an edge which if removed doesn't make the graph disjoint, thus, the redundancy. Now, there's a very popular algorithm to deal with problems involving disjoint sets, namely - Union-Find.

Union-Find - Let me first describe the algorithm & then explain how it relates to the problem at hand. So, you maintain a parent array & a rank array. The parent array stores the parent of each node, and the rank array stores the size of the sub-graph of every node.



Say, we have this graph. parent = [0, 0, 0, 1]

0 is 2's parent

1 is 3's parent

0 is 0's parent (top node)

0's is 1's parent

1's subgraph has 1,3

rank = [4, 2, 1, 1]

0's subgraph has 0,1,2,3

3's subgraph has 3.

2's subgraph has 2

As the name suggests, there are 2 steps to this algorithm, Union & Find. The find bit finds the foremost parent of every node. In the above graph, the foremost parent of every node is node 0. The code is fairly easy to understand, just a while loop. The union part of the algorithm on the other hand, updates the parent & rank arrays in order to establish an edge b/w 2 nodes. So, the union function helps us connect 2 nodes from disjoint sets (i.e., different foremost parents), and in the process re-assigns the parents of the 2 nodes in question, based on their ranks. Check the code to get a clear idea. The union function returns False if the 2 nodes are already connected (i.e., if they have the same foremost parent)

So, with the explanation of union-find out of the way, let's understand how it aligns with our problem of identifying redundant connections. Let's think of all the nodes as disjoint sets, i.e., each node is its own parent & each node has a rank 1. Now, iterate over the the edges given. An edge signifies a connection b/w 2 nodes. So, corresponding to each edge, run the union function, passing the 2 nodes as the parameters. If the union function returns False, we can return the corresponding edge as the redundant connection. Because a False o/p means that the nodes passed are already connected. Note - the question mentions that we must return the last redundant edge from the i/p list. This is automatically taken care of as there is only 1 excess connection. While there could be multiple possible answers, however, since we are iterating over the edges in order, we shall always return the last edge causing redundancy.

usually, the time & space complexities are mentioned like foot notes, but this one needs more explanation.

Union-Find $\approx O(\log n)$ (when coded up using rank & path compression) ↳ # of disjoint sets

However, since, we must iterate over the entire edges array. Our complexity is linear.

So, Time- $O(N)$, space- $O(n)$

XCIX) Accounts Merge (LC 721) ★

So, we have been given some accounts, and the job is to merge the same accounts. 2 accounts are the "same" if they share a common email address. In other words we have been given a few disjoint sets (denoted by the account index) and our job is to connect the disjoint sets referring to the same account. Seems like a job suited for union-find.

Union Find + dictionary to store the account indices corresponding to the emails — Let's keep this simple. We'll iterate over the accounts list and for each entry we shall iterate over its email ids (acc[1:], ∵ the first entry is the name). Now, each time we encounter a new email-id, we shall create a key-value entry in our email2Account dict, to store the account index corresponding to the email id. However, whenever we stumble upon pre-existing email ids, we must merge the 2 accounts. Thus, the union call b/w the previously listed account index & the new account ind. corresponding to the email-id in question. Ultimately, we must compile the list of merged accounts. So, create a dictionary with values initialized to empty lists, using defaultdict(list). Now, iterate over the email2Account dictionary & for each account index, find the parent/leader (result of taking unions in the previous step), and finally, corresponding to the keys for each of these leaders, append the

respective emails. The last bit is just to sort the emails & align the O/P with format asked.
Time ≈ O(# emails), space ≈ O(# emails).

C) Word Ladder (LC 127) ✳

A lot of starred questions in graphs, eh xD? Anyway, this one is not particularly difficult. We have been given a beginWord & we must chart the shortest path to arrive at the end word. This screams BFS. The question is how do we optimally find the neighbouring words (words which are 1 transformation away from one another).
BFS + Patterns (to track the transformations) + Visited - I don't think that the BFS bit needs explanation. We have a word, we can get to some words from that word & so on, until we arrive at the end word. Figuring out the neighbouring words is greatly aided by the idea of patterns! It's quite simple really. A word of length n can lend itself to n patterns. ex- hit → *it, h*t, and hi*. Now, all the words fitting into these patterns can be clubbed together. So, let's create a dictionary nei, wherein corresponding to each pattern key, we shall store a list the compliant words from our wordList (check the first for loop). Next, we come to the BFS + visited part of the program. Since, we are supposed to measure the path length, we shall iterate over the entire BFS queue at once (exhaust the whole layer in 1 go), like in tree height problems. Finally, as we go through our queue of words, starting with beginword, we shall find its neighbours using the nei dictionary, and append the words not yet visited in our queue. The BFS while loop can stop if we arrive at the end word. As always the path length is updated after a layer has been exhausted, i.e. after the for loop → for i in range(len(q)).
Time - O(wordlength * len(wordList)), space - < Time, the exact ans is weird.

CI) Reconstruct Itinerary (LC 332)

This is not a particularly difficult problem to solve (logically). The code is probably the reason why it's been marked hard. Anyway, you've been given a list of tickets from pt: A to pt: B. The ask of the problem is to return an itinerary of locations you'll have to visit (in order) to use all your tickets, while starting from JFK. Upon pondering over the question for a minute or two, the use of an adjacency list becomes intuitive. So, for each source in our tickets, we shall create an entry in our adjacency list storing the list of possible destinations. Each such list of destinations will be sorted to ensure that we check the possible paths in lexical order. The idea is simple, start from JFK & dfs over the possible destinations (recursively, ofc), and if you come across a path that exhausts all the tickets, you have your answer! The tricky part is that since only a few paths are possible, you will have to provision backtracking.

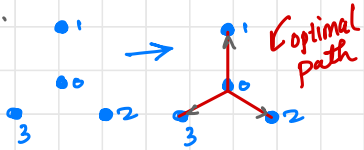
DFS + Backtracking - DFS + backtracking is essentially just Backtracking. The reason behind starring this question was to introduce backtracking in the context of graphs/adjacency lists. To come up with a possible itinerary, we must track the tickets we have exhausted. Since, each ticket is stored in our adjacency list as per the source location, while traversing down a path, we must remove the src → dst entries once used. In other words, we must remove dst from adjacencyList[src] while exploring the path involving the air ticket from src → dst. I feel like I have over-explained a simple idea, anyway xD. The point is that we have to modify our adjacency list according to the current dfs path. Thus, we'll have to insert & remove elements at specific indices from our adjacency list. Check out the code for this implementation.

Time - $O((V+E)^2)$ (Traversal + Backtracking) ; space - $O(V+E)$ ↳ adjacency list.

(11) Min Cost to Connect All Points (LC 1584) ★

This question falls under a general category of problems called "Minimum spanning Tree". Minimum spanning Tree in simpler words refers to the min cost / shortest total length / any proxy of distance b/w nodes of the graph required to connect all points. Here, this cost function is the Manhattan Distance. The solution to problems around min spanning trees is greedy, one such such commonly used greedy solution is Prim's Algorithm. Let's try to arrive at it!

Knowing that the solution is greedy, let's dive right into it with all our greed xD. In order to connect all points, we must have at least 1 edge connecting each point. How about we minimize this edge weight to ensure that each edge gets optimally connected? In other words, we require n-1 edges to make a simple path, how about choosing the smallest n-1 edges connecting all nodes? Take a moment to digest this. Initially, you might be tempted to think that the min path will only have one edge coming out of every node, but you'd be wrong. Consider 3 pt's arranged in a equilateral triangle & a 4th point placed at the centroid of that Δ. You see, the optimal path need not be like a line graph. The closest pt. form 1, 2, 3 is 0. That's what matters.



So, with this crude understanding, let's formalize things. We must store all the edge weights & to ensure that the path is simple & not cyclic, we cannot visit the same node again. Thus, besides tracking the edge weights, we must track the corresponding destination nodes. There is a sense of direction in our path, thus the use of "destination node".

Prim's Algorithm - (visited + minheap): We'll start at the first node. To store all edge weights and their corresponding destination nodes, we'll maintain a heap (because of logN insertions/deletion of elements stored according to relative value). We'll initialize the heap with [0, 0], i.e. cost to reach the 0th node (first node-0 index) is 0. Next, we shall maintain a set of the visited destinations to ensure that we don't re-visit them (to ensure a simple path). Lastly, put these 2 together in a while loop tracking the # of nodes visited. Each time, we'll pop the element with the min. cost from our heap. If the element popped corresponds to a destination we've already been to, we'll keep looking, else we'll add its cost to our result & the dest node to visited. Finally, we'll add the [edge wt, dst node] pairs from the current destination to our minHeap for further calculation. That's it. once, we visited all nodes, return result.

Time - $O(N^2)$ (N vertices => N^2 possible edges)
Prim's algo takes $O(N \log N)$ but since we must find all possible edge weights the $O(N^2)$ takes over.

Space - $O(N^2)$ (all edge wts).

(III) Network Delay Time (LC 743) ★

So, we been given a source node & we need to find the min time it takes for all nodes to receive the signal emitted by the source node. The signal travels in a BFS manner. Let me rephrase this question so that it makes perfect sense. You must find the longest path length from the single shortest path lengths to all nodes from the source node. This screams Dijkstra! The only twist is that we want to reach all nodes. Thus, the time it takes to reach the final node will be our answer.

Dijkstra - There's really not a lot to explain in the theory part of this question, other than explaining Dijkstra

itself. Watch a Youtube video explaining Dijkstra if your grip on it is loose. The point is that since this problem has positive edge weights & can be boiled down to find the longest single source shortest path in the graph from our source node, we use Dijkstra.

Time - $O(E \log(V))$, space - $O(V+E)$

(IV) Swim in Rising Water (LC 778)

The question has a tricky wording. So let me clarify the essentials before we take this question forward.

1. 0 time to swim. (∞ speed)
2. Height of water = time passed
3. to swim from pt. A to pt. B. water level ≥ max(elevation A, elevation B); i.e. time >= max(elevation A, elevation B).

With these things in mind, we have a simple SSSP problem wherein, we must go from pt. (0,0) to pt. (n-1, n-1) on the grid. All edge weights are positive, since, time can't be negative.

DIJKSTRA!! - We'll maintain a minHeap with each element = [time to reach (r, c), (r, c)]. The initial point is (0, 0) ⇒ minHeap will be initialized to [0, (0, 0)]. The only part which you should be wary of is pt. (3). We must make sure that we update time according to both the source & destination. It might seem quite obvious once you see it implemented, but it's actually easy to mess up. Notice the max(time, ...) snippets in the code.

Time - $O(n^2 \log n^2)$ or $O(n^2 \log n)$ & space - $O(n^2)$. (size of the grid is $n \times n$).

(V) Cheapest Flights Within K Stops (LC 787) ★

It's a fairly simple question. We have a source node & a destination node, with positive edge weights. TEXTBOOK

DIJKSTRA! The only twist is that we cannot allow more than K stops.

Dijkstra (with stop restriction) - You'd be surprised to realise how easy this condition is to impose on top of vanilla dijkstra. In our heap, besides storing the price & the dest node, we'll also store the no. of stops required to reach that node @ that price. Our answer will be the cost of the first element popped with the dest node = dst & stops <= K. Dijkstra guarantees popping in an ascending order, so whatever the cond^n might be, the first element following the cond^n/s is our optimal answer.

Time - O(ElogV), space - O(V+E) (TLE :'C)

Technically, dijkstra is O((E+V) log V) but ∵ O(E) ≥ O(V), we can simplify

Breadth First Search + Maintaining a MinCost Map/list to keep track of the Min. cost to reach all nodes (like in Bellman-Ford) - The approach is actually quite simple; however, Vanilla BFS isn't really the first thing that springs to mind when we see a SSSP problem.

The idea as advertised is simple. Create a minCost array and initialize all its indices to ∞, except the index corresponding to the source node. Next, let's run our layer-exhaustive version of BFS (the one with an extra for loop on the queue length) and add the stops cond^n on the while loop. As we pop nodes & their corresponding costs from our queue, update the cost to reach their immediate neighbours if required.

If you are updating the cost to reach a node, i.e., if it's lower than the current estimate of the minCost for that node's index, append the node & its new cost to the queue as it will affect the cost to reach the subsequent nodes.

You see, by constraining the while loop to only explore upto k+1 layers from the source node & updating the minCosts in every node traversal, we are assured of getting the minCost to reach 'dst' in upto k stops. BFS + Bellman Ford, ftw!

Time - O(V+E*k) or O(E*k) & space - O(V+E).

(VI) Alien Dictionary (LC 269) ★

This is a very interesting question. Based on the order of words provided; you must decipher the lexical order of the alien dictionary. First, try to develop some intuition by manually solving this problem.

ex- words- ["wrt", "wrf", "er", "ett", "rftt"]

∵ wrt is before wrf ; => t comes before f ; in other words, we can establish relative order b/w chars that serve as the first point of difference b/w 2 adjacent dictionary words.

w r (t)	t : [f]	so, we compare 2 adjacent words in the
(w) r (f)	w : [e]	alien dictionary & the first char that
(e) (r)	r : [t]	differs helps us establish the relative
(e) (t) t	e : [r]	lexical order; Here → wertf.
(r) f t t	{ char : [characters after char in the alien lexical order] }	

Let's understand how this order can be obtained more generally. We are maintaining an adjacency list of sets, storing char keys and the corresponding characters below them in lexical order as per the dictionary, by making adjacent pair-wise comparisons. There is a popular dfs based algorithm called Topological sort made especially for this purpose, i.e. establishing order b/w nodes; given their relative order (here, in the form of an adjacency list). This algorithm can be more easily understood as "post-order dfs".

Read the following code snippet, keeping in mind the adjacency list created above.

```
visited, result = [], set()
def dfs(char):
    if char in visited: return
    if char in adjList:
        for nei in adjList[char]:
            dfs(nei)
    visited.add(char)
    result.append(char)

for char in adjList:
    dfs(char)

print(result).
```

This snippet is → post-order dfs (because, we append the char to our result after completing its dfs path).
Consider using this algorithm on the Wertf adjacency list.

start @ t; t's list = [f]. ; visited = { }, result = [ ]
=> dfs on f; f's list = [ ]; visited = {f}, result = [f]
→ back to t; visited = {f, t}, result = [f, t].
start @ w; w's list = [e]; visited = {f, t}, result = [f, t]
dfs on e; e's list = [r]; visited = {f, t}, result = [f, t]
dfs on r; r's list = [t]; visited = {f, t}, result = [f, t]
dfs on t; t in visited ⇒ visited = {f, t}, result = [f, t]
→ back to r; visited = {f, t, r}, result = [f, t, r]
→ back to e; visited = {f, t, r, e}, result = [f, t, r, e]
→ back to w; visited = {f, t, r, e, w}; result = [f, t, r, e, w]
start @ r; r in visited; return
start @ e; e in visited; return.
result = [f, t, r, e, w] reverse of our actual order.

The result can be reversed and returned. To avoid this you can also just flip the def^n of the adjacency list. Instead of storing chars "after" in the lexical order; store chars 'before' in the lexical order. i.e.;

instead of t: [f], w: [e], e: [r], r: [t] store f: [t], e: [w], r: [e], t: [r]; this will create the result in the correct order.

Anyway, there are a couple of caveats about this approach that you should appreciate →

① The order of chars in our adjacency list does not affect the result, i.e. if, we had w: [e] as the first entry, instead of t: [f], the result won't change. Therefore, it's the relative order that matters, not the how we store this relative order, ∵ DFS will automatically converge on the leaf nodes.

② In such questions, you might end up with disjoint sets. Imagine had the last entry of "rftt" not been there; No connection b/w the sets of [w, e] & [r, t, f]

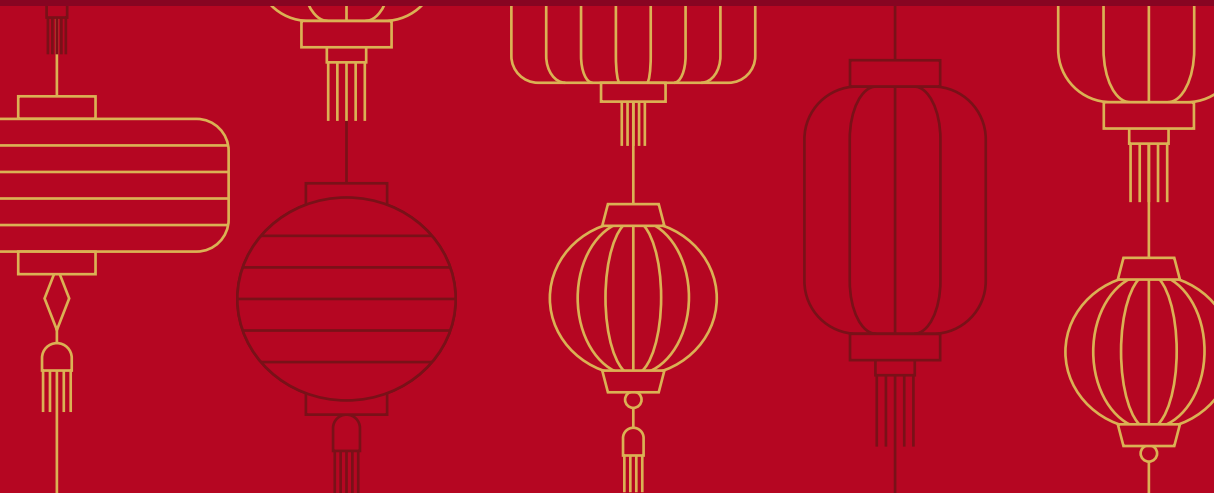
would have existed. In which case there could be multiple possible lexical orders. So, as long as, w appears before e & r, t, f appear in order. ex- wertf, rtfwe, wretf etc; we are good.

With these things out of the way, let's get to the actual code. The code snippet discussed previously is not the one you see in the actual code. The reason is simple, we haven't "edge-case proofed" our code yet. There is a possibility of cycles in our graph. To overcome this, we must modify the visited set. Instead of just storing the the visited chars, we'll use a dictionary to store {char: bool} key-value pairs. If a char is present in visited => it has been visited (obviously), the catch is that the bool value indicates if it's in the current dfs path. If we encounter the same char twice in one dfs path => cycle. To enforce this idea, we have slightly modified the code. I hope this looong explanation made sense :)

Time - O(len(words) x len(words[i])) → create the adj list
+ O(# chars) → size of the graph created

space - O(# chars) ↗

Heap/ Priority Queue



Heaps come in handy when we are tasked with maintaining elements in sorted order dynamically. By "dynamic", I am referring to operations that require adding new elements, removing the smallest/largest elements, etc., all this while maintaining the elements in a sorted fashion.

To implement heaps in python, we use the heapq library.
heapq.heapify(list). ← to make the list → heap.
heapq.heappop(heap) ← to pop the smallest element from our heap.
heapq.heappush(heap, item) ← to add an item to our heap.

The questions discussed below are all quite important. So, take some time to go through them.

(VII) Task scheduler (LC 621) ⭐

You see, heap is just a data structure, it's the idea around the algorithm that makes the use of heaps optimal. There's obviously a brute-force way to solve this question. It's almost too ridiculous to fathom xD. Let's be greedy. All it takes to solve this question is 1 realization - "EXECUTE THE TASKS WITH THE HIGHEST FREQUENCIES AS FREQUENTLY AS POSSIBLE!". It's not easy to arrive at this epiphany; however, running through a few test-cases will help you gravitate towards this idea.
Once you've made your peace with this realization, you can implement the idea. If you're not comfortable with this, here's something that might help:- In order to minimize the total time taken, we must minimize the the no. of cool-off times (idle times). To achieve this, we must execute tasks with higher frequencies as quickly as possible, or else we'll be left with just them at the end & that'll cause a lot of avoidable "idle times" for the CPU.

Think of using tasks with lower frequencies as fillers b/w the high frequency tasks. I hope this helps.
Anyway, to execute/schedule the tasks in decreasing order of their frequencies, we need a heap! Everytime we execute a task, it's frequency goes down by 1 & we must re-arrange the tasks in the desired order.
The code is a little tricky, so have look at the supporting comments.
Time - $O(n \log n)$; space - $O(n)$
↳ # of tasks.

(VIII) Find Median from Data Stream (LC295)

Another tricky question to solve optimally. The question asks us to find the median from a data stream. The very definition of a "median" relies on maintaining a sorted list. Thus, the use of heaps, isn't all that obscure. However, heaps aren't used here in the conventional way.
You see, heaps are only good for tracking the min./max values, not the medians. Median is the middle element.
How do we fetch that in less than linear time?
The idea is to maintain 2 heaps, one max heap & one min heap. Both these heaps will have ≈ half the elements in the stream. We'll have a smaller half (max. heap) & a bigger half (min. heap); since, for tracking the median, the only 2 values that might concern us are - the largest value of the smaller half & the smallest value of the larger half. The median will be the average of these 2 values, if the # of elements is even; else the median will correspond to the top element of the heap with more elements.
It's not really the easiest thing to arrive at, but it's hard to forget once you know. So, we must append/push new elements in the stream to maintain a balance b/w the 2 heaps, while ensuring the relative order doesn't get

twisted. Please refer the code for the details.

Time - AddNum → O(log n)
↓ current length of the data stream.

FindMedian → O(1).

Space - O(n) ← to maintain the 2 heaps.

CIX) Hand of Straights (LC 846) ★

Making groups of size, "groupsize" consisting of consecutive values. It's obviously in our interest maintain these the i/p values/ cards in a sorted order to check the possibility of consecutive values. The question requires each card to be in a group. Consecutive values for a card can be card-1 & card+1. Thus, it's always good to look at the extremes, i.e. the smallest & the largest cards in our sorted list. since they can only be a part of a group if it has (card+1 (for the smallest value) & card-1 (for the largest value).

So, come to think of it; we'd be better of clearing cards from 1 end while trying to create the respective groups. This process is easy to implement as we always try to create a group for the smallest value in the remaining list of cards. Corresponding to each such "smallest card", we need groupSize-1 subsequent values to create the group, and should that not be possible, return False.

It's quite easy to just implement this idea using a list. We can simply remove values using the remove() function. This, however will cause the solution to be of quadratic time complexity.

Let's try to optimize it further; instead of just sorting the list, let's first get a count on each of the cards because that's relevant to determine the possibility of grouping our cards in straights. You see, if the frequency of the smallest card (n) is f. Then n+1 ... n+groupSize-1 cards must all have a frequency of at least f (to ensure that all 'n's get

grouped. Next, let's create a minHeap to store the cards & their respective frequencies (minHeap over the card values, to fetch the smallest value). The rest is simple. Iterate over the minHeap until its existence, each time popping the smallest value in it and its corresponding frequency. Subsequently, pop the next groupSize-1 elements to check if they can form a group with the smallest card. Finally, push the remnant elements from the subsequent groupSize-1 elements popped.

Time - O(n + mlogm); space - O(n)

n → # of cards; m → # of unique cards

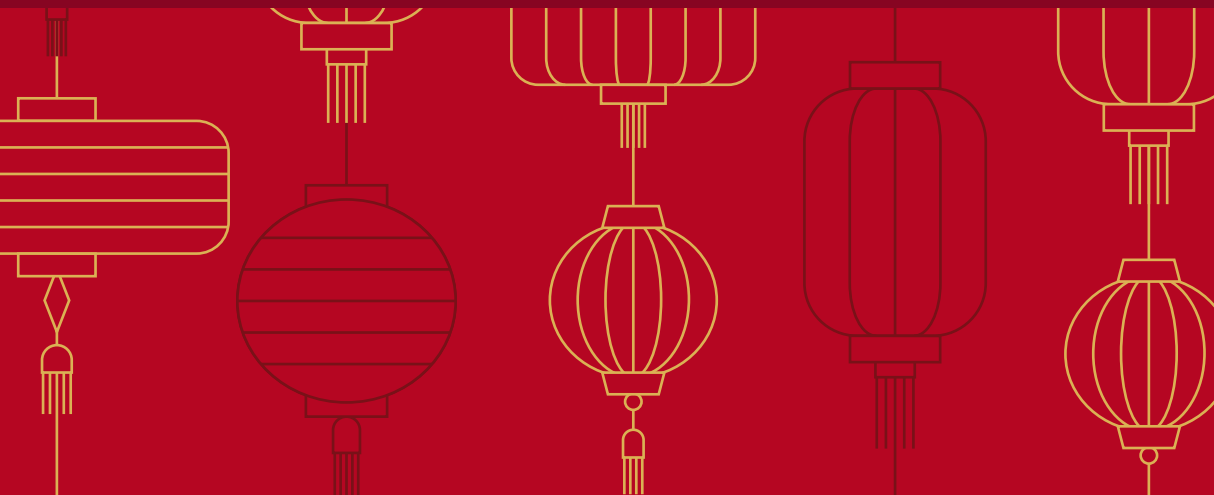
IX) Minimum Interval to Include Each Query (LC 1851)

There's obviously a simple brute force approach to this question but quadratic time complexities are seldom the optimal ones. Like while dealing with any intervals problem, sorting the intervals list is a good place to start.
Once you take some time & ponder over finding better solutions, you'll realise that we must somehow, find better ways to establish relationships b/w subsequent queries. Say if one query was over 2 & another was over 3, we need not go through the entire list of sorted intervals to find the shortest one for 3, if we're already done so for 2. This also leads us towards the importance of sorted queries to deliver an optimal time complexity.
You see, if an interval was rejected by query, i due to the interval's upper limit being smaller than i, it sure can't work for query, i+1 (in a sorted paradigm).
Now, we must find the shortest interval that fits the query. What better data structure than our good ol' heap to keep track of the possible intervals as per their sizes (minHeap).

So, the idea is to iterate over the sorted list of queries & first store the intervals whose lower limit is compliant with the current query. If the lower limit works for query i, it sure will for query, i+1. Thus, let's first push the intervals whose lower limits comply with our query. We'll push the intervals as $(o - l + 1, r)$ into our heap; we need not store l, because it has already been validated for query i, & thus for the subsequent queries. Next, we'll just pop the values from our heap until we find a value whose upper limit (o) is greater than or equal to the current query. Lastly, we can just store the interval length of the top-most interval in our heap (corresponding to the current query). In case the heap is empty $\Rightarrow$ there are no intervals for the current query. The heap retains the useful intervals from the sorted list of intervals, i.e. whose lower limits comply with the subsequent queries. Thus, instead of exploring the intervals list from the top, start iterating from where the lower limits were too big for the previous query i.e., where you left off (thus, the variable i in the code). So, this way, we only iterate over the intervals list once for all queries. Well not really, because we're maintaining a heap of the relevant prev. intervals, but yes, it's definitely better.

Time $\approx O(n \log n + q \log q)$; space - $O(n+q)$.

Miscellaneous



There is no end to questions around DSA. I have purposely not written elaborate explanations for the topics mentioned below as I feel that they should be used to test one's understanding of the basic algos & data structures covered thus far. Regardless, here are some short notes to help -

1) Tries → Prefix trees are useful for string related tasks & to work efficiently with trie problems, one must be good with the basic traversal algorithms for trees & graphs.

2) Intervals → It's barely a heading in and of itself XD. Anyway, just familiarize yourself with the basic code structure to deal with such problems & you'll be good.

3) Greedy → There's no particular concept here; it's just specific solutions to specific problems. In other words, not a lot of scope for generalization.. Practice aids intuition

4) Math & Geometry → Basically, it tests your comfort in dealing with matrices. Helps to familiarize yourself with performing basic matrix operations, optimally.

5) Bit Manipulation → This is just a re-branded version of digital design. Thankfully, just knowing the basics works.

Take some time, go through the questions for these topics & have a look at the code. I have added a lot of comments for these topics to aid the understanding.

I hope this helps :)