S2

Perspective; changes in perspective, identifying useful perspectives, when to switch perspectives

problems to cover: ECNA ICPC 2024 A, B, G, D, L

A: goals : cover basic input/output, understanding problem, understanding solution

- ask if everyone has had the chance to read through the problem
- the problem is slightly confusing, but as long as you read through it carefully, and you understand what it is asking of you, then you will be able to almost immediately solve it. so let's look at the problem and try and find what exactly it is asking for
- if you'll notice, the problem actually has a lot of filler, and it can be hard to immediately identify which pieces of information are important. this is a quirk of the ICPC contests, in other online judges, this will usually not happen. but when you practice, that is a specific skill that you should aim to learn, which is to identify the useful information from the problem statement
- in this problem, there are actually only two useful pieces of information. the first is in the second sentence of the first paragraph, where it basically tells you what problem you are trying to solve. the second useful piece of information, is in the second paragraph, or actually is the entire second paragraph. not only does it specify the problem that you are trying to solve, it also tells you how you are supposed to solve it.
- so if I were to summarize it, the problem is essentially asking you, if you had N stacks of coins, where the first stack has one coin, the second has two coins, and so on, which of the stacks has the gold coins? and you also know that, of these N stacks, there is one stack with the material gold, and all of the others have the material tungsten. and each material has their own specific weight.
- ask if anybody knows how to solve this
- go through different approaches
- first approach: note that weight of gold is more than weight of tungsten, so calculate total if all were tungsten, subtract all tungsten weight from real weight, then divide by one gold coin weight
- second approach: note that if you can calculate weight if each specific stack from 1 to N was the gold stack, and check if that is the same as real weight

B: goals -> introduce perspectives
- B's problem statement is much more concise, and easier to understand than problem A
- ask if anybody has any trouble understanding
- explain substring: sub-arrays are contiguous (a subarray contains all indices [l, r), while sub-sequences are not, can contain any index, such as 1, 2, 6, 10
- now, if you only wanted to find the answer, it's simple to construct an algorithm using brute force
- as long as you iterated through all possible subarrays, or substrings, you would be able to find an answer
- how many subarrays are there? ask (if nobody answers, it's around N^2, since there are n subarrays, since there are n subarrays of length n, n - 1 subarrays of length 2, etc)
- so just going through all subarrays would take n^2 time, but you'd also have to find the maximum digit in the subarray
- doing this naively, you could iterate from the left index of the subarray to the right index, and update the maximum iteratively
- we know that n^2 is already too much to pass, so n^3 is definitely impossible, we must find a faster solution
- so in informatics style problems, the core is to identify useful information, information that we can use to solve the problem
- let's take a look back at this code, can we all agree on something obvious?
- each subarray has a left index, and a right index
- on its own, this is not a useful piece of information. but what is the problem that we are trying to solve? basically, we're trying to find the sum of all of the max digits in each sub array. we can find the average by dividing by the amount of subarrays, which is an easy calculation
- now, if you notice, we're basically going through each subarray by fixing the left index and incrementing the right index, one by one. what if there was a better way to increment the right index than just by manually increasing it by one every time?
- if you read the problem statement again, you'll notice that we want to find the max digit in a substring. but there are only ten digits in total. so what if we calculated, for each substring starting on some index L, which substrings have the maximum value of digit 0, which substrings have the maximum value of digit 1, and so on?
- so now, we've basically shifted our perspective from considering each individual substrings, to a group of substrings that share a common maximum value
- but how will we find the maximum value for a substring?
- basically, we can go back to the problem and discover that if a substring contains a 9, then it doesn't matter if the substring contains an 8, a 7, or anything else
- and the same goes for a substring that contains an 8 (if we know that it doesn't have a 9). then the max value will be 8, regardless of how many 7s or 6s there are
- so as long as we found the smallest index (larger than our current index) that had a 9, we know that for all of the substrings from this index to the last index, the maximum value is 9.

- so if we then found the smallest index (still larger than our current index) that had an 8, there are two cases: either this index is less than 9's index, or the index is to the right of 9's index. since it doesn't matter when the index is to the right of 9's index, we can ignore this case.
- so if this index is smaller than the other index, which substrings will have 8 as its max value?
- you're basically asking, which substrings have 8 as its max value? well, all of the substrings that have an 8 but not a 9
- so it'll be the substrings from 8's index to 9's index, including 8's index, excluding 9's index
- you can extend this logic all the way down to 0.
- finding the minimum index is also simple, you can binary search for the first index (of whichever digit) greater than the current left index
- we have here another shift in perspective: we solved this problem by separating it into sub-problems that we are familiar with and have solved before
- this is the core of the technique: you want to shift your perspective to look at the problem as one that you can solve

problem G
- relatively simple problem
- problem statement is a little confusing - how can we simplify the problem statement
- each team has a set of letters that they need to solve; we have to choose the largest subset of teams such that there is no overlap between the problems that each team solves
- we can write this formally if anybody wishes to
- why don't we try to brute force this problem first?
- why do we brute force? look at constraints, teams <= 20, and length of string of p <= 80. but most importantly, we know that there are only 26 letters in the alphabet, which reduces the search space by a lot.
- so how do we brute force it? well, brute force means that we search through the entire search space, this means that all we have to do is try every single subset of teams and check if they meet the required conditions. and we will take the max size of all of the subsets that pass this check
- any ideas on how we can achieve this?
- one method that makes the code relatively clean is to use recursion. let's say that we have a current subset of teams that pass the conditions, meaning they do not share any letters. we want to check, for the list of teams that we haven't gone through yet, if we were to add them to the current subset if they would violate the conditions.
- to do this, we would need to keep two arrays: one to store which teams are in the current subset, one to store which letters are currently being used.
- now, all we have to do to get the answer is to update the maximum when we find that the current subset size is larger than the previously found maximum
- time complexity? well, we want to check every single subset with brute force. there are 2^n subsets. we have to check all available teams when choosing the next team, so for each subset we check, it's around n teams average. so the total time complexity is 2^n * n, times a constant depending on your implementation. this is around 10^8, so with a little optimizations, it will pass.
- recursion is not the only way to count subsets, it's just one of the easier ones. another method is using bitmasks. if you'll notice, a binary integer with n bits has 2^n possible states, you can choose 1's to correspond to sets included, and 0 to be excluded from subset considered

problem D
- brownian bears
- solution is quite simple, but is one of the harder questions (this year)
- I'd like to talk about forcing perspectives
- what is forcing perspectives?
- the idea behind it is to select some algorithm or data structure or problem solving technique, and see how you could use this algorithm or data structure or problem solving technique, to solve this problem
- for example, suppose you have some question that requires you to answer some queries
- there are multiple different methods to choose from when answering queries. first of all, you can choose whether to answer queries online or offline, then you can choose which data structures to perform update queries, i.e. segment trees or sqrt decomposition
- this is obviously not applicable to everything. i.e., if you were asked to calculate the sum of a list, you would obviously not need a fenwick tree. likewise, for some heavy mathematical problems, the core of the problem will need some mathematical analysis; trying to use some complicated data structure would likely not help
- now, there are some downsides to using this technique. as the famous saying goes, when all you have is a hammer, everything starts to look like a nail. similarly, if you only think in certain ways to solve problems, you may develop an over reliance on these specific techniques, and fail to recognize other obvious solutions. as a semi related note, if you only practice problems, knowing the technique or topic beforehand, you will also find it harder to solve problems in contests. why? since usually identifying the main idea, or the topic, is one of the hardest parts of the problem
- what are the benefits of using this technique then? why do you use it?
- it limits the space where you can find a technique. for example, if you decided to use a segment tree to solve a problem, there are only so many things a segment tree can be used for. if you identify one useful place where you can use this segment tree, perhaps it will lead to you finding other observations, or even finding the solution to the problem
- so immediately, when you see this problem, one of the first thoughts that comes to mind is dynamic programming. this is because, there are many problems of this type where I've solved them using DP before. so it basically comes down to intuition. intuition is a very interesting and nuanced topic that I won't cover today, but just know that intuition comes with practice
- how do we apply DP here?
- well, what is DP exactly? dynamic programming is basically just storing previous states that can be used in calculating other states
- one sort of common method, that spoiler alert, will not work, is to record all states.
- we can do this by starting on day 1, and calculating on day 2, how many states will bear 1 end up in location 1 to location N
- on day 1, there is only one possible square he can be in

- on day 2, there are two possible squares he will be in: either square minus one, or square plus one
- on day 3, let's say he ends up in the original square minus one. which squares can he be in? square minus two and square original
- let's now look at the case where he ends up in the plus one square. which squares can he be in? square original and square plus two
- this can be easily extrapolated: let's say that on day A, there are X possible paths for the bear to end up in square B. now, for day A + 1, there will be X possible ways for the bear to end up in the left square, and another X possible ways to be in the right square
- we can also calculate this for bear number 2.
- on each day, we now have the amount of ways that bear 1 ends up in some square, and same for bear 2.
- can we sum up the amount of ways that the bears meet on each square for each day? is that our answer?
- no, because we didn't consider the fact that if the bears meet on one day, then the bears still might meet on the next day, meaning that we are overcounting
- this solution basically leads to a dead end.
- does this mean that DP doesn't work?
- no, it doesn't; we once again have to shift perspective
- if we can't simply calculate all the locations of the bears, why don't we just do what the problem asks for?
- let's look at the problem in a more general way
- let's say that the bears are on day X, at locations A and B
- what is the chance that they'll meet on day X?
- well, it's clear that if location A equals B, the chance is 100%
- otherwise, how can the bears move?
- 0 0, 0 1, 1 0, 1 1
- all of these are equally likely
- so the probability is one fourth of the probability that they meet on day X + 1, A - 1, B - 1, etcetera
- solution ends up being incredibly simple! but if you got stuck in other perspectives, then who's to say when you'll end up solving a problem?
- lesson to be learned: switch perspectives or solutions when you find yourself stuck