UMPIRE (/umpire/)

For any problem — whether in interviews, or studying — one must always create a plan of attack. The UMPIRE method is one way to create such a plan, and find your solution. It's especially helpful when you're completely lost as to how to solve a given problem.

The UMPIRE Method:

1. **Understand**what the interviewer is asking for by using test cases and questions about the problem.
2. **Match**what this problem looks like to known categories of problems, e.g. Linked List or Dynamic Programming and strategies or patterns in those categories.
3. **Plan**the solution with appropriate visualizations and pseudocode.
4. **Implement**the code to solve the algorithm.
5. **Review**the code by running specific example(s) and recording values (watchlist) of your code's variables along the way.
6. **Evaluate**the performance of your algorithm and state any strong/weak or future potential work.

UMPIRE helps you organize your thoughts to avoid getting stuck in an interview. Also, even though it might seem like a drawn out process, it will help solve problems *faster* than jumping in without a plan. Also, when you dive in to coding the answer to the problem right away, the consequences are as follows:

- The interviewer will think you isolate yourself when you solve problems. Translation: you won't be able to work on a team effectively.
- You will miss out on valuable hints. Part of UMPIRE is opening dialogue with the interviewer. The interviewer cannot help you if they don't know your thought process **ABS (ALWAYS BE SPEAKING)**
- You may find yourself at a dead end with 10 minutes remaining, and you will then scremble to reorganize your approach. If you tell the interviewer your plan

beforehand, as is done in UMPIRE, the interviewer can guide you in the right direction, and prevent you from going down that path.

UNDERSTAND:

One of the best ways to understand a problem is to come up with test cases. Coming up with good test cases can be an art, but I typically like to cover one happy path or an average case. And any interesting edge cases. This can be tricky as it depends on the problem. Remember that the interview is a dialogue, so understanding the problem is best done as a dialog as well. Asking questions, like Should this be done in place?… Does the ordering beyond the left/right of the partition matter? Create simple test cases and try to talk with your interviewer to see if they believe you get the idea. NEVER leave any stones unturned, keep asking until you perfectly understand the questions!!!

MATCH:

Once you understand, try to match algorithms/data-structures to the question. For example, if the answer involves a sort maybe try using a priority queue. If the answer needs a certain order, try a stack or a list. Told to search in a sorted list, BINARY SEARCH should be the first thing that hits your head, a two-pointer approach should be second. You get the idea.

PLAN:

This is where you try your ideas. I cannot stress enough to write out your logic in sentences and then pseudo code it. Interviewers need at least a pseudo code form just in case you can’t code the question for real. Sometimes its ok if you can’t code it completely, if they can understand your logic and know in general you can code with more resources… its good enough. I like to write a complete pseudo code in leetcode on top of the given function. Many times this has led me to writing the real code in 2-3 minutes! DO NOT FORGET EDGE CASES, what if your input is null or size of 1? The better you plan the more your interviewer will be impressed. You can also discuss the run/space complexity with the interviewer if your **100%**sure it’s right. Up to this point try to aim to leave the plan stage within 8-10 minutes. The next stage should take up most of the rest of the time.

IMPLEMENT:

If our plan stage was successful, we should be able to implement the code by simply gluing the plan together while filling out the details of our language specific implementation. If your pseudo code is strong, this part should be a breeze. If you ever get stuck, your plan is already made so there's A LOT less thinking and MORE doing! Remember the CPU reference before? If you didn't plan and got stuck implementing, now your brain is processing and retrieving memory at the same time. This is how you burn out quickly! Use your pseudo code as a CPU CACHE!!! Quick small retrieval if necessary, to process that plan efficiently! If you still get stuck this is where you ask for hints or stop completely for 1-2 minutes to think.

EVALUATE:

The last step of the UMPIRE strategy is to discuss the pros and cons of your algorithm with your interviewer. An algorithm much like a good interview is not amenable to correct/incorrect label, instead there are often things to discuss about what you like about your algorithm or code and what you would like to change depending on conditions, inputs, expectations, the business, etc. A great place to start this conversation and is almost always required is to discuss the asymptotic performance of the algorithm.

Always remember, everyone always starts at the bottom of the ladder. Its tough at first but eventually you'll notice patterns. When you do, that's when you activate god mod. If a problem is tough, spend 1 hour thinking the best solution or part of it. Then check the answer and spend the time looking at different solutions/understanding it line by line! Remember treat your brain only as a CPU AND ABS!!!