OUTCO

A note from the Lead Author

Sergey, former Outcoder now Engineer at Google

After working at Outco for the past two years, I've noticed some patterns on how engineers learn technical material and what kinds of roadblocks they regularly run into.
The breadth of material covered in Outco is roughly equivalent to two semesters of computer science topics, condensed and delivered in 5-weeks instead of a year.

We cover a lot during the program, from how to traverse trees and graphs, to sorting lists efficiently, to reducing the time complexity of recursive functions. Engineers in our program will learn a lot by sticking to the material we provide, however the ones that learn the most excel at another, more "meta skill" that is harder to teach explicitly.

That skill is how to approach problem-solving in general.

Because ultimately there's a limit to how many algorithm questions you can memorize and learn for an interview.

The risk of sticking to a rigid curriculum or set of problems is called **overfitting**, which means that your way of thinking becomes too tailored towards solving only specific types problems.

It can create a vicious cycle where engineers think they are learning by practicing very similar problems, but will fail miserably when attempting problems that require them apply what they've learned in a different context.

The end result is an engineer being unable to solve new problems they encounter during the interview process, even if they might be similar to ones they've solved in the past. This makes them rely more on the chance that the interviewer asks a problem they've seen before, which ultimately translates to a longer job search.

This guide is meant to lay out a general framework for how to approach problems in a methodical way, and provide actionable insights for handling tech interviews.

OUTCO

STEP 1

Always Utilize the Power of Asking Questions

OUTCO

STEP ONE

A lot of engineers have the tendency to freeze after being given a particularly hard problem on an interview. The reason for this is something called an Amygdala Hijack, which is many of the "non technical" roadblocks we help cover during our course.

In short, this is when the fear center of the brain takes over higher functions of the brain and basically prevents people from being able to think clearly and effectively problem-solve.

Most engineers have experienced this at some point during the interview process and we've found that **having a set starting point of what to do during an interview is crucial to prevent this from happening.**

Whenever you have been given the prompt to a problem, start by asking questions about it.

It doesn't matter if you're just working on the problem by yourself or if an interviewer is asking you.

The first stage of this process is all about establishing an understanding of the problem, and to constrain your thinking about it.

- Does this look like a problem you've seen before?
- What do you remember about similar problems?
- What immediately stands out about it to you?

You can start with time and space constraints. Knowing this will help you rule out solutions that run too inefficiently, or it will give you a hint of how many auxiliary data structures or loops you need.

If you have an array, for example, you could ask:

- Is the array sorted?
- Does it contain only numbers?
- Are those numbers integers?
- Can they be negative, or just positive?
- Is there a max length or a range of input sizes?

For strings you could ask:

- Is there a distinction between upper/lower case?
- Are there spaces?
- Is there any punctuation?

Be sure to ask about edge cases, like empty strings, empty arrays, very large or very small inputs etc.

Ask about language specific questions:

- Can you use certain libraries?
- How should you handle errors?
- What kinds of inputs should your function handle?

If it's a pair programming interview:

- Can you google syntax and docs?
- Can you run your code with examples?
- Will the interviewer be able to answer certain questions?

Try to figure out what guarantees there are in the problem

It's all about putting constraints on your thinking and finding parameters that will focus your attention towards a smaller solution space.

Keep in mind time and space complexities might not always be given. They might just say 'do the best you can.'

This shouldn't deter you, but it also means there could be several ways to solve this problem, and some are better than others.

The point is that there are a ton of conditions and constraints that will help focus your thinking and can hint at the direction you need to go in. At the most basic level you are gathering data.

With all this said, keep this portion of the interview short, maybe just a couple minutes of discussion or thinking. And don't frame it as you looking for hints. Frame it as you trying to understand the problem better.

In summary, asking questions is all about gathering more information making sure you are solving the correct problem.

It would shock you how often people start solving different problems than the one I asked, all because they didn't clarify what they were supposed to be doing. For example, the Tower of Hanoi Problem.

PRO TIP

Remember you can ask questions later on as they come up, so don't stress about trying to nail EVERY constraint now.

OUTCO

Apply the Concept

Problem: Tower of Hanoi

The Tower of Hanoi is a puzzle with three pegs, and a variable number of discs.

The discs are stacked with the largest ones on the bottom and the smallest on top.

Each time a disc is moved from one peg to another peg, counts as one step, regardless of if the new peg is adjacent to the old one.
The goal of the puzzle is to move the whole tower from one peg to another peg in the smallest number of steps.

Your task is to write a function that takes an integer as the input and outputs another integer that represents the minimum number of steps necessary to move a tower of that size from one peg to another.

Input: Integer (discs)

Output: Integer

By Trixx (I designed this using http://thewalnut.io/) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

In this example the questions you should be asking are about the nature of the discs, and the answers will provide you with constraints you can use later on:

- Can you move more than one at a time?
- Can you put larger ones on smaller ones?
- Is there a maximum number of discs for this problem to be solvable?
- What should the time complexity of the solution be?

You should also be asking if you actually have to **simulate** the motion of the discs themselves.

This isn't an immediately obvious question to ask, but it's a very important one. A lot of students start devising data structures and disc swapping algorithms they'll need to solve this problem, when in reality they've already failed the interview.

The solution only asks for the **NUMBER OF MOVES**, not for you to actually create the **SEQUENCE OF MOVES**. This is a BIG difference, one which simplifies the problem tremendously. This is something all interviewees should be on the lookout for when they are asking questions.

Often times the problem description will appear deceptively difficult, or conversely, deceptively simple. Asking more questions about it will help shed light on whether the problem is straight forward, or if there are hidden subtleties.

All the answers to these questions should go somewhere, whether that's on a whiteboard, or on a shared text editor, or even a piece of paper.

You should also write down everything that comes to mind: patterns you notice, constraints you are given or things you observe. Any hunches, ideas or thoughts should all go somewhere.

This might be like identifying important features of the problem.

- Is the max/min element relevant?
- Does there seem to be some kind of pattern between input size and output size?
- Can you rule out certain approaches based on constraints, like using built in methods or certain libraries?

These are the building blocks of your solution. Think of this as a brain dump.

You want to get everything down so you can go back to it later. You can only hold so much in your head at a time, and this will help lighten some of that cognitive load.

Additionally, the process of writing things down and thinking of how to express them in another way is what helps your brain move closer to a solution. It's the same as how the simple act of talking out loud about a problem can help you solve it. There's something that happens when your brain translates ideas from one format to another, that makes you gain a deeper understanding of those ideas.

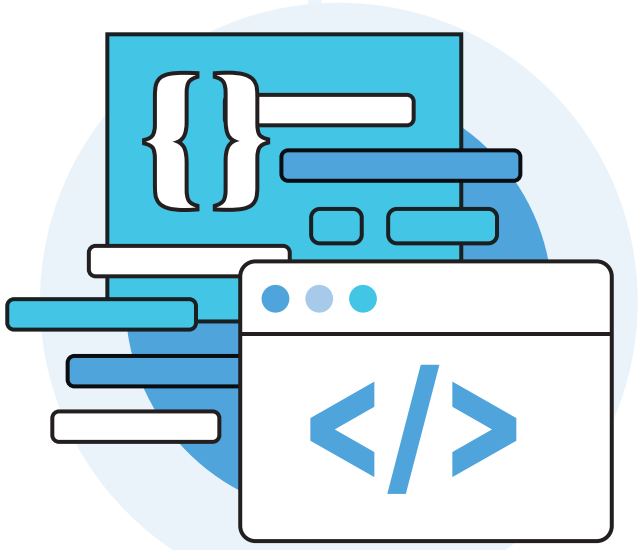
Again, the goal here is to simply get everything written down somewhere. It doesn't need to be super organized, but it should be legible and you should be able to make sense of it. This step is meant to be a tool for you to use later one when you are solving your problem.

```
/*
Here are the constraints:
Only move one disc at a time
Only smaller discs on larger discs
No max number of discs
Number of discso won't be negative
Do the best you can on space and time
No need to simulate moves, just output total number
*/
```

OUTCO

STEP 2

Try more
Examples



OUTCO

STEP TWO_

One of the most common places we see engineers fall short in the interview process is in their use of examples. More specifically, they simply don't try enough of them.

The interviewer may prompt you with an example or two, but remember it's up to you to try out more. The objective here is to gain even more data points that will serve you down the line.

An analogy for this is an imagine in which you must connect the dots. As you add more dots and make more connections, the pattern starts to emerge.

OUTCO

Apply the Concept

Example: Sequence of Numbers

If I give you a few numbers and ask you what sequence is this a part of, it would be impossible for you to tell.

[1, 2]

That isn't enough information to say what the pattern will be. Depending on the sequence, the next number could be 3 if its the linear sequence, it could be 4 if it's the exponential sequence and it could be 6 if its the factorial sequence

Now if I give you the next number 3, you know a little more but you still can't be certain what sequence it is exactly.

[1, 2, 3]

If the next number is 5, you have a good indication that we're looking at the Fibonacci sequence, where any given number is the sum of the two previous numbers in the sequence.

[1, 2, 3, 5]

But it's still a good idea to find more, just to be certain.

[1, 2, 3, 5, 8, 13]

Going back to the tower of Hanoi example, it's a good idea to see what the output for a given number of discs would.

Do this by manually moving discs and tallying up how many moves it takes to get them from one peg to another.

```
/* Javascript
Hanoi(0) => 0
Hanoi(1) => 1 (trivial cases)
Hanoi(2) => 3 (move top one off, move bottom disc, and then top
back on)
Hanoi(3) => 7
Hanoi(4) => 15
*/
// Runs in Exponential Time O(2^N) Linear Call Stack Space O(N)
function Hanoi(discs) {
if(discs === 0) {
return 0;
}
return Hanoi(discs - 1) + 1 + Hanoi(discs - 1)
}
// Runs in Linear Time O(N) & Linear Cal Stack Space O(N)
function Hanoi(discs) {
if(discs === 0) {
return 0;
}
return 2 * Hanoi(discs - 1) + 1
}
// One Line Recursive Solution
const Hanoi = (discs) => discs === 0 ? 0 : 2 * Hanoi(discs - 1)
+ 1
//Iterative Solution Runs in Linear Time O(N) & Constant Space
O(N)
function Hanoi(discs) {
let result = 0
for(let i = 0; i <= discs; i++) {
 results = 2 * result + 1
}
return result
}
//Formula Solution
//Log Time O(log N)
function Hanoi(discs) {
return Math.pow(2, discs) - 1
}
```

This process of trying different examples tests your understanding of the problem. Given a certain input, can you correctly predict the output of the problem?

This can be something you confirm with the interviewer, and it won't come across as you asking for hints, but solidifying your understanding.
Sometimes the output is obvious from the input, like sorting a list, but often it isn't.

What you are doing is establishing the endpoints of the problem. The start and the finish. Your algorithm is the route the data takes between those two points. And oftentimes there are many ways to get there.

Your function at this point is a black box. You don't know how it works on the inside, or what routes the data takes as it moves through the algorithm, but you know what the inputs and outputs SHOULD look like.

It's like knowing you are starting a road trip in San Francisco and ending up in New York. There are different routes you can take; some are faster, safer or more scenic. Which you choose depends on your needs, but ultimately, all routes need to lead you to the same destination.

This is also the foundation of Test Driven Development. You have some expectations for outputs given some inputs, and now your job is to write a function that fulfills those different test cases.

Approach this step with a scientific mindset. Create a hypothesis, and test that hypothesis.

> *"If I give my function this input, I should get that output."*

Be rigorous and if your hypothesis is wrong, change it and try again. This is what scientists do. They sample data, they survey the space and they try to draw conclusions.

PRO TIP

Trying a lot of examples also **reduces the risk of overfitting**

This is where a solution you try only works for a more limited set of inputs than required.

Maybe it only works for sorted inputs, or positive inputs, or lowercase letters. In some cases, like creating a binary search algorithm in a sorted array, these are constraints on the problem itself. But other times it means you've overlooked an edge case or a set of inputs that would break your reasoning.

We see this all the time with candidates who haven't solved a lot of problems.

In the extreme case, they'll literally use the one example they are given and come up with a solution that ONLY works for that specific example and maybe similar inputs.

Sometimes they hard code in their solutions things related to that specific example. Like they think solving that example is the whole challenge.

Luckily in interviews the solution space tends to be small. Often you'll only have to try a handful of examples to get the gist, or even one might suffice. It can be a bit of an art to know how many or what kinds of inputs to try.

One thing you can do is be random with your inputs.

Given enough different inputs this will likely give you a good sense of the different possible cases your function will have to handle. This is kind of how machine learning in your head works; by trying random permutations and shuffling things around until an optimal solution just kind of "clicks".

Another thing you can do is to be methodical with the examples you try.

Try to break your assumptions about them. Randomness can help it be robust, but also try to design situations that break your algorithm.

Figure out the edge cases, and test them.
If you can't engineer an example to break your reasoning, you are on a good track.

Start specific and then slowly generalize your function. Make sure it works in a specific case and then slowly expand it to handle more and more cases.

The bottom line is that trying examples will really help you understand the problem better, and there's a whole layer of unconscious activity that is working in your mind under the hood that is getting you closer to the solution.

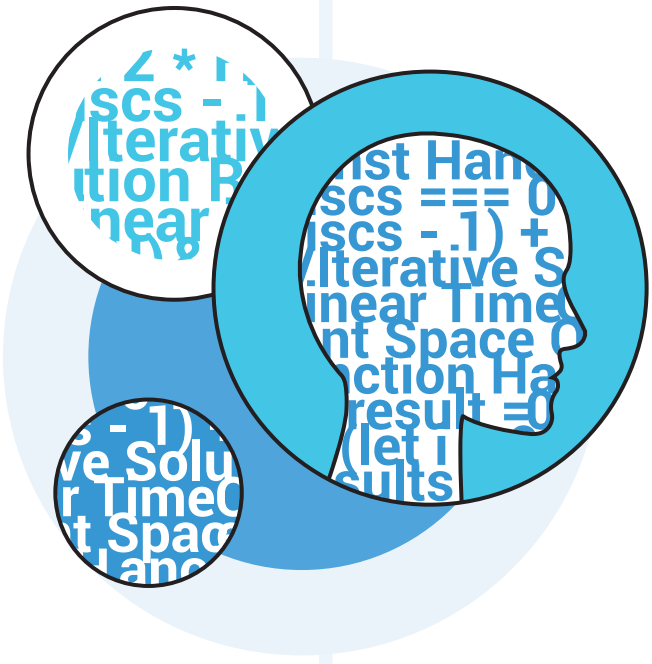
PRO TIP

The harder the problem, the more examples you should play around with.

OUTCO

STEP 3

Become The Algorithm



OUTCO

STEP THREE

If you were living during a time before personal computers and had to solve this problem with a pen and paper how would you do it?

Forget about figuring out the concrete steps you would have to follow, or how you would implement your solution.

Just focus on trying to solve ONE EXAMPLE yourself.

This might feel weird at first, but it involves exercising something called **lateral thinking**.

It's synonymous to "thinking outside of the box" and it involves looking at a problem from different angles and playing around with it. We all know how to do as children but forget as we go through our formal education and our thinking becomes more rigid.

Lateral thinking is difficult to teach directly, but on a high level it involves playing around with assumptions; relaxing and tightening constraints and trying out and testing new things.

For example:

- What if your algorithm "knew" more or less about the input data?
- What would happen if you focused on solving only a part of the problem?
- What if you assumed you already had something that could solve part of problem?

It's called lateral because it's non-linear. There isn't necessarily a logical progression; it's more organic and exploratory.

Which leads to the next important piece here: **adopting a mindset of exploration and play.**

These are things present in children when they are young and learning about the world that we as adults tend to forget. Tap into that innate sense of discovery and lose yourself a little in the algorithm.

Treat this like a fun puzzle or brain-teaser and remind yourself that the enjoyment from **challenging yourself** and **solving interesting problems** is why you got into programming in the first place.

Now obviously this is easier to do in a low-stress environment where you are practicing and learning new problems.
It's more difficult to do this in an actual interview setting with time constraints and added pressure. But being able to enter this "zone" or mental space comes with time and practice.

We can't stress enough how much solving algorithm problems comes down to having the correct mindset and focus, and being able to adopt them in a controlled setting will increase your likelihood of doing so when it matters in an interview.

Josh Waitzkin talks a lot about this in a book called "The Art of Learning", where he chronicles his journey of how he became a chess and martial arts champion. It's a fascinating read where he goes deep into the inner-workings of the learning process.

The Art of Learning: An Inner Journey to Optimal Performance

This is also where diagramming comes in. Our monkey brains are designed for visual input so this is another step to help bridge the gap between what we understand intuitively and what machines understand programmatically.

If you have arrays, for example, draw them out with pointers:

- What could you do if you had one pointer?
- What about 2? Or 3? Or more?
- What if you don't start both pointers at the beginning?

One could go to the end, another might be in the middle, or could serve as some kind of tally.

These are the common patterns for when you have O(1) (constant) space complexity constraints.

What if you had a hash table or some other auxiliary data structure to help you?

Think about what the algorithm knows and when. If you only have two pointers, looping from the front and back, the algorithm won't have any sense of what's in the middle. If you have a dictionary of key-value pairs with all your values, you have O(1) lookup for everything. This is a common pattern for when you have O(N) space.

For trees, graphs, matrices etc. draw out an example with all the connections. You can think of this being the "View", you are the "Controller" and the input data/ auxiliary data structures would be the "Model" in an MVC framework.

Updating the view (changing the drawing), updates the model (or data) and you are the controller (the one making it all happen).

Trees are also a great way of visualizing multiple recursion functions, where the recursive cases are represented in the branchings.

Sometimes you get problems that produce trees with unusual branching factors, like for balanced parens permutations or for string permutations. This will give you some insight into how you should structure a variable number of recursive calls depending on different input cases.

PRO TIP

The code for multiple recursion and depth-first binary tree traversals often looks very similar.

Apply the Concept

1. Print All Combinations of Balanced Parentheses Exercise on GeeksforGeeks.com
   https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/
2. Write a Program to Print All Permutations of a Given String Exercise on GeeksforGeeks.com
   https://www.geeksforgeeks.org/print-all-combinations-of-balanced-parentheses/
3. Then think about how the call stack would treat the problem.

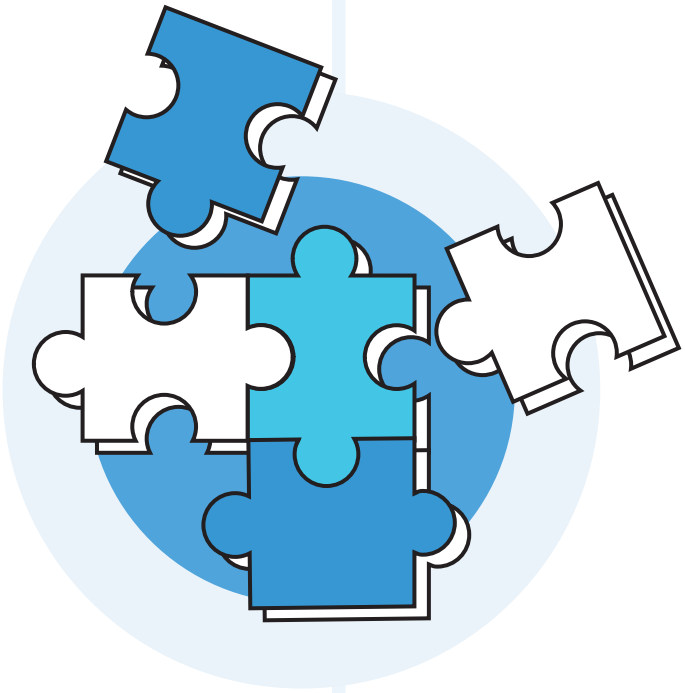
   Because at the end of the day computers follow instructions in a linear manner even if it makes more sense for us to conceptualize things in branched data structures.

   Be the algorithm. Behave how it would behave. Think how it would 'think'.

OUTCO

STEP 4

Distill the Problem To its Essence



OUTCO

STEP FOUR_

The goal of all this playing around, writing observations and acting as the algorithm is to ultimately boil down the solution into one or a few sentences. This represents the most compressed way of expressing your solution, and being able to find the hidden gem that holds the key to solving the problem.

The best way to start to understand and practice this part of the process is to do it with really simple algorithms.

Take finding the max element in an array of numbers for example.
The one sentence solution for that would be:

"Iterate over the array, keeping track of the largest number you've seen so far, updating it when you see a larger one."

That's it.

That sentence contains everything you need for solving that problem, and being able to come up with it is the true indicator of whether you understand how to solve the problem.

I can't tell you how many times I've asked people whether they know how to solve some algorithm they are given, and they have been unable to express it in one sentence. Or when they've tried, it turned into a long, garbled run-on sentence.

You really can't trust your intuition here, you have to rely on your ability to produce this sentence to check yourself.

One big aspect of this distilling process is also about breaking larger more complex problems into smaller, distinct pieces pieces.

The way you determine whether something is a distinct piece is if you could write a distinct helper function that produced the output for that piece.

For example, for the problem of Anagram Palindrome, the two pieces would be:

1. Count the number of occurrences of each letter

2. Make sure there's at most one letter that occurs an odd number of times

You could write a distinct function that takes in a string of letters and then outputs a dictionary that just displayed the count of each letter. That would solve the first piece.

The second piece would take that dictionary as an input and have some kind of tally of the number of letters the occur an odd number of times, and return whether that tally was less than 2.

But all this relies on knowing some kind of **keystone piece of information**. In the anagram palindrome example, that's the observation that all *palindromes have at most one letter that occurs an odd number of times.*

If you think of the problem solving process as an hourglass, this would be the midpoint.

This one sentence, or insight is the most condensed piece to solving this problem. Before this point, you are dealing with examples, and diagrams, and test cases, and auxiliary variables, and writing notes/observations.

And after this you will expand your few sentences out into many lines of code that handle looping, reassigning variables, mutating data…

But at this point, your goal is to express things as simply as possible.

And then to write this down somewhere so you can refer back to it if you ever get lost. I've seen people come up with the key insight and then forget it as they got into implementing their solution.

And this can be applied to all kinds of complex problems:

Take heapsort, for instance.

The first step is to create a max heap from the unsorted array. Then you remove elements from the peak of the heap and placing them at the end, while maintaining the max heap condition. That's it. Two steps.

The main trick here is trying to answer "what" your function is going to do, and to resist the urge to express "how" it's going to do it. The how comes right

after, but it's good to know your goals before trying to implement them. Otherwise you might commit to using some kind of data structure or approach that isn't optimal and then your solution starts to try to make things work with that data structure, rather than figuring out what the best tool for the job is by knowing what you are trying to accomplish.

As the saying goes, "when you're a hammer, every problem starts to look like a nail."

A specific coding example that we cover in the course is that of an LRU cache. You can replicate its functionality and behavior with a simple array, but a more efficient implementation involves a queue and a dictionary.

But if you pre-commit to using an array, all your LRU cache methods will be seen through the lens of making them work with an array.

Apply the Concept

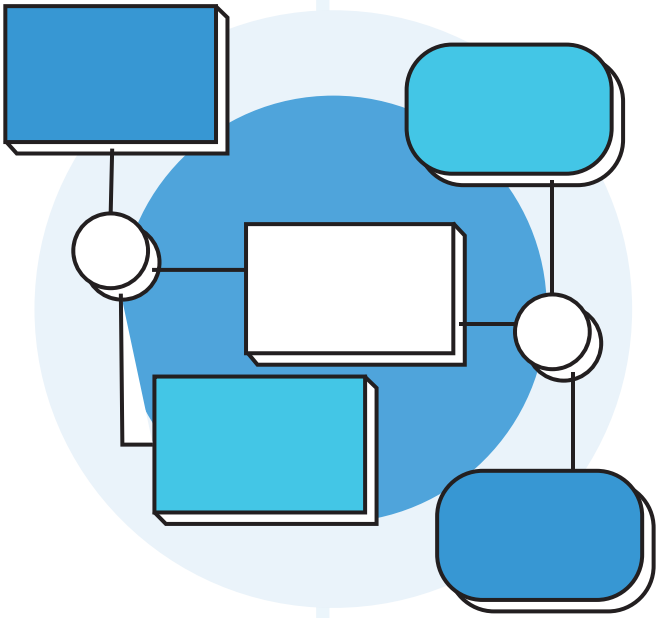
How to Implement LRU Caching Sheme on GeeksforGeeks.com
https://www.geeksforgeeks.org/lru-cache-implementation/

To summarize, the goal of this step is to condense all your observations and work into only a few sentences in english, that express WHAT your algorithm will do, not HOW it will do it. For that, we will need the next step.

OUTCO

STEP 5

Pseudocoding



STEP FIVE.

A lot of people skip the pseudocode step, which in our opinion is a big mistake.

The main reason you may do this is that you think it saves time, but that's not how things tend to play out.

Let's say you skip writing your pseudocode. Best case scenario, you've saved yourself 5 minutes or so on your solution and you get done with the problem a little faster.

It's unlikely that amount of time saved will actually increase your chances of landing the job, though saving time is always beneficial.

Worst case scenario though, you'll end up with a botched solution.

I've seen a number of people on the right track to solving a problem end up spending 4 times longer than necessary because they got confused and didn't have a "source of truth" to refer back to.

They end up erasing and rewriting a lot of code, and forgetting what their original plan was in the first place. Sometimes they end up not even solving the problem at all when I know they were capable.

So clearly there's a huge potential downside to skipping this step, and only a small potential upside.

Doing this step will in the worst case cost you a few minutes, which in all fairness you may need. But there's also a good chance that you will actually save yourself a huge amount of time and effort. Your first implementation of a solution will usually have some bugs in it, or the problem might be complex and you'll want to have something to refer back to while you're coding out your final answer.

So having discussed the merits of pseudocoding, let's talk about how to actually go about it.

Two Types of Pseudocode

The first is high-level and in plain or abbreviated English. This pseudocode answers the question of "WHAT" you are going to do.

That's what you created in the last step. If this is a pair-programming interview this can be as a comment, and if it's on a whiteboard it can be off to the side somewhere as long as it's easy to refer back to.

Going back to Anagram Palindrome as the example:

1. Count each character in the string

2. Determine whether there are two or more that occur an odd number of times

From that you will derive the second type of pseudocode, which is probably closer to what most people think of when they imagine pseudocoding.

This is the "HOW" of the problem. It's far more granular and it actually lays out the steps needed to be taken to implement a solution.

Typically there's a one-to-one correspondence between a line of low-level pseudocode and a line of actual code.

Here you'll actually be spelling out any additional variables, auxiliary data structures or actions your algorithm will perform, like adding, swapping, traversing, instantiating etc.

In some languages with very minimal syntax like Python, this pseudocode will look almost like code, and often interviewers won't even ask you to code out your solution if you do this part well. That's because these step-by-step instructions should be sufficient to implement a solution in any language, which is why many tech interviews are language-agnostic: if you can get to this point, implementing the next part should relatively trivial.

Remember that this part is still meant to be a valuable tool for you. It should be written in such a way that you can code out each line one at a time and not have to hold too much information about the rest of your solution in your head.

Sometimes it can be difficult to move from your high level pseudocode to your low-level code, in which case, try thinking about your problem in terms of its different cases.

For example, if you are iterating through an array of numbers, where you are removing repeats, you can think of any given number falling into one of two categories: either you've seen it before, or you haven't.

For something like the rainwater problem, you have to figure out, does a given space contain water above it, or not. And if it doesn't how much?

Those are all things you can do at the diagramming stage. Once you know those pieces, you can go about figuring out how you to systematize your solution.

Apply the Concept

Trapping Rain Water on GeeksforGeeks.com
https://www.geeksforgeeks.org/trapping-rain-water/

Let's look at a problem like sorting an array of 0s and 1s in linear time and constant space.

You can do this with two pointers starting from the front and end. If you do that there are only four possibilities for what the value of the array is at those pointers:

Both point to 0s

Both point to 1s.

The first points to 0, and the second points to 1

The first points to 1, and the second points to 0

That's it, there are not other cases to handle. In the first one you want to increment increment the lower pointer, and leave the upper pointer where it is. In the second case you want to decrement the second pointer, and leave the first.

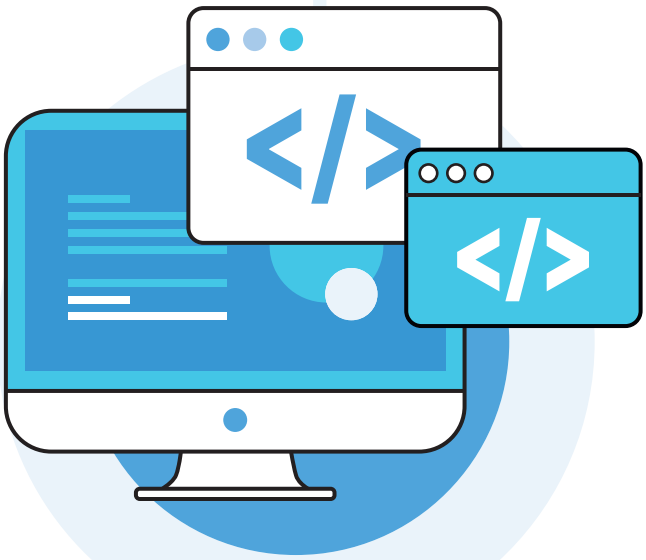
In the third case you'll increment the first and decrement the second, and in the final case you want to swap the two values.

So think about the cases you can encounter and try to capture all of them. The goal is to not have to think about syntax while you are solving your problem and writing out the steps. It also lets you be less precise and able to back track if you make a mistake. Locking yourself in with code can mean you end up erasing a lot of time consuming work, which you want to avoid at all costs.

OUTCO

STEP 6

Time To Do Some Coding



OUTCO

STEP SIX

If you've done everything right up to this point, this section shouldn't be super difficult. In fact, it should almost be trivial.

If your pseudocode is good, you should have a step-by-step formula for implementing the solution to this problem laid out basically in English.

That said, it pays to be clear but concise when you're writing your pseudocode so that when you or your interviewer refers back to it, it's legible and understandable. And make sure that when you start writing code that it's clean and DRY. Style points are a big part of this section.

Here is the solution to Anagram Palindrome with what some notes, examples and pseudocode to mimic what real world interview notes might look like:

```
// Anagram Palindrome
// Given a word, return whether some anagram exists that is a
palindrome
// Palindromes: mom, racecar, dad, bbaabb, asdffdsa
// Anagrams: abc, cba, acb,
// "mmo" => true
// "carrace" => true
// "cutoo" (outco) => false
// Brute force: Create all anagrams, see if any are palindromes
// a
// ab, ba
// abc, acb, cab, bac, bca, cba
// Runs in O(N!) which is very bad
// Key insight all palindromes have at most one character that
occurs an odd number of times
/*
Diagram:
             i
Input: "carrace"
{
c:2,
a:2,
r:2,
e:1
}
*/
// 1) Count the frequencies of each letter
// 2) Make sure that no more than one letter occurs an odd
number of times
```

```
// 1)
// Create a hash table
// Loop through the string
//   if the current character is not a key in the hash table
//        then create a new entry with that character as the
key, and 1 as the value
//   else (otherwise)
//       Increment the value by one
function AnagramPalindrome(str) {
// 1) Count each letter in the string
 let counts = {}
 for(let i = 0; i < str.length; i++) {
   if(counts[str[i]] === undefined) {
     counts[str[i]] = 1
   } else {
     counts[str[i]] += 1
   }
 }

// 2) Make sure no more than 1 letter occurs an odd number of
times
 let numOdds = 0
 for(let key in counts) {
  if(counts[key] % 2 === 1) {
   numOdds += 1
  }
 }
 return numOdds < 2
}
console.log(AP("carrace"))
```

PRO TIP

On code DRYness - it's okay to start with a solution that doesn't use the fewest number of if/else statements or the most clever optimizations

Code clarity here is more important than saving a few lines on the whiteboard. Below are two different implementations of the same solution to the above problem of sorting an array of 0s and 1s. One uses three "if" statements while the other just uses one.

They both work, but it's not immediately clear why the more "clever" solution works. It makes it harder to understand and judge for yourself whether you have the right answer, and it also introduces the possibility of bugs and other mistakes.

```
// Very Clear implementation, handling all 4 cases, but verbose
function sortBits(arr) {
 let start = 0;
 let end = arr.length - 1;
 while(start < end) {
   if(arr[start] === 0 && arr[end] === 0) {
     start++;
   }
   else if(arr[start] === 0 && arr[end] === 1) {

    start++;
     end--;
   }
   else if(arr[start] === 1 && arr[end] === 0) {
     [arr[start], arr[end]] = [arr[end], arr[start]];
   }
   else if(arr[start] === 1 && arr[end] === 1) {
     end--;
   }
 }
 return arr;
}

//Slightly condensed, cleaner solution
function sortBits(arr) {
 let back = 0;
 let front = arr.length - 1;
 while(back < front) {
   if(arr[back] === 1 && arr[front] === 0){
     [arr[back], arr[front]] = [arr[front], arr[back]];
   }
   if(arr[back] === 0){
     back++;
   }
   if(arr[front] === 1){
     front--
   }

}
 return arr;
}
// VERY clever solution, with only one if case
function sortBits(arr) {
```

```
 let slow = 0;
 for(let fast = 0; fast < arr.length; fast++) {
   if(arr[fast] === 0) {
     [arr[fast], arr[slow]] = [1, 0]
     slow++
   }
 }
 return arr
}
// Slightly easier to follow with while loop
function sortBits(arr) {
 let slow = 0;
 let fast = 0;
 while(fast < arr.length) {
   if(arr[fast] === 0) {
     [arr[fast], arr[slow]] = [1, 0]
     slow++
   }
   fast++
 }
 return arr
}
sortBits([1, 0, 0, 1, 1, 0]) //outputs: [0, 0, 0, 1, 1, 1]
```

One important thing you should think about at this point is syntax.

In fact, syntax is the main reason for separating the pseudocode from the code in the first place: so you don't have to worry about syntax while you're reasoning through the solution.

Every bit of your brain's CPU is valuable and while it's focused on solving the problem it shouldn't have to be worrying about language-specific idiosyncrasies.

So make sure you know your syntax, especially for strictly typed languages like Java.

We regularly see engineers who rely on editors like Eclipse to autocomplete syntax and it costs them interviews where they don't have those tools available. It ends up being a crutch.

Debugging your code also becomes important here. Sometimes, even if your process was super tight up until this point, there still might be off-by-one errors, or other easy-to-miss bugs that can cause havoc in your final solution.

And if there is a bug, test the different components of your solution with strategic print statements. The point is to minimize the surface area the bug might be in, but checking to see if the output of each part of the algorithm is correct. If you structured your solution correctly, this should be easy to do. It's kind of like performing a binary search for finding the bug: successively cutting your search space in half is the fastest way to find it.

The rules are simple, if you see a function, add a block the stack and the state of all the variables at that call-time. If you see a return statement, pop off a block, replacing the function call in the previous block on the stack with the value you are returning, if you are returning one at all.

Here's a visualization of it.

And then try edge cases to make sure your final solution is robust. What happens if your input is 0? Or negative? Or an empty array/object? What if every value is the same, or if they are very large etc. Stress test your solution.

This part is all about being precise and getting into the nuts and bolts of your solution to make sure everything is working properly.

In addition to these 6 main steps to solving algorithms, Outco will provide you with personal guidance on how to structure your studying ie what problems to practice and how often to do them etc.

Try solving similar problems to build flexibility and comfort.

If you wrote a function that produces all the combinations of strings of 0s and 1s of a certain length, write a function that only outputs the combinations that don't have consecutive ones.

Or if you wrote the code for a min heap, try writing the code for a max heap. If you know quicksort, try mergesort. It's kind of like "muscle confusion" training in the gym. You're trying to reduce overfitting to a particular problem.

Also look at other solutions to the same problem. Studying other people's code will help you progress faster than if you just stick to solving problems yourself. Everyone has a different way of thinking, and you'd be surprised how much you can learn and optimize your own code by looking at other people's.

PRO TIP

You can also visualize the call stack, which is especially important for recursive problems.

Start to look for patterns of space and time complexity

If you see a problem with constant space, that usually means only a few auxiliary variables. Linear space means a dictionary or array. If you see linear time, that means one iteration through the array/string or up to the input number, but it could also be some fixed number that doesn't scale with input size (I've seen problems with up to 4 passes!). If you see log(N) time, that implies some kind of binary search.

There are a limited number of patterns that cover a good chunk of all the problems that get asked. If you internalize those, you'll be able to decide what tools are best for the job.

This will do far more good than a few days of cramming every once in a while, for the same reason that one really hard workout won't get you in shape as well as a lot of consistent sustainable workouts. But don't spend more than 30min to an hour spinning your wheels.

If you are completely stuck, look at the solution, learn it and then come back to the problem a few days later. Over time, you will learn. And often you'll be 95% of the way there, and there is a small subtle change that needs fixing. So don't stress; this is a marathon, not a sprint and burning out will not get you anywhere. Step away from the problem and come back, you might be surprised how much your unconscious does; it really does work, try it.

Confidence is also a huge part of doing well on interviews, but that is tied to how competent you feel. And this just gets built over time with practice. As Ben Horowitz said in his book, 'The Hard thing about Hard Things': "sometimes there are no silver bullets, only lead bullets".

PRO TIP

A good habit to also get into is solving a daily algorithm.

Conclusion

And finally, going back to "The Art of Learning", you are trying to learn "form to leave form."

This is a martial arts term and it means learning a structure to eventually transcend that structure to have a more fluid form.

We do not expect you to think about each of these steps exactly when you see a new algorithm, you should naturally internalize this line of thinking.

Test things, move forward and if you hit a dead end you can move back to a previous step, write pseudocode, then diagram, then ask questions and so fourth.

Instead of a rigid structure, it should become more organic with time and practice, personalized to your style of thinking.

On a higher level, you can think about coding out algorithm problems as a sort of transformation from human thinking into computer thinking.

Under the hood everything gets compiled down to zeroes and ones eventually, but if we were to look at that series of digits it would seem like random noise.

Similarly, a computer (and even humans) can't really understand the inner workings of our brains. It just kind of pattern matches and comes up with a solution.

The amazing thing is that we are able to bridge that gap, by constructing programming languages and specific ways of thinking and interacting with computers that allow us to encode the solutions to abstract problems in a step-by-step format.

Solving algorithm problems is both a science and an art that indirectly measures a person's ability to interface with computers and allow themselves to think the way they "think." And while it's not a perfect way of evaluating someone's coding skills, mastering the kind of thinking it requires is both essential for getting a job and a valuable skill on its own.