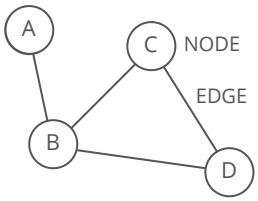
Interview Cake

Given an undirected graph

A **graph** organizes items in an interconnected network.

Each item is a **node** (or **vertex**). Nodes are connected by **edges**



Strengths:

- **Representing links**. Graphs are ideal for cases where you're working with *things that connect to other things*. Nodes and edges could, for example, respectively represent cities and highways, routers and ethernet cables, or Facebook users and their friendships.

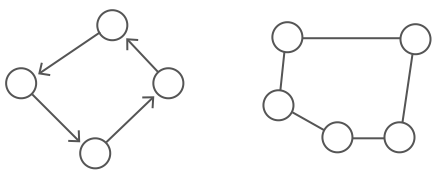
Weaknesses:

- **Scaling challenges**. Most graph algorithms are $O(n * lg(n))$ or even slower. Depending on the size of your graph, running algorithms across your nodes may not be feasible.

Terminology

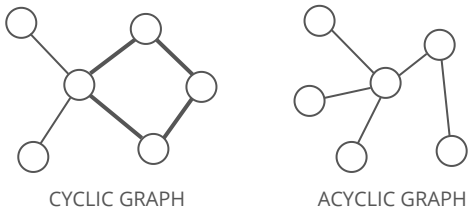
Directed or undirected

In **directed** graphs, edges point from the node at one end to the node at the other end. In **undirected** graphs, the edges simply connect the nodes at each end.



Cyclic or acyclic

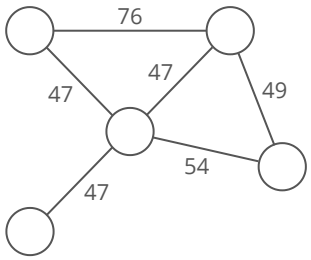
A graph is **cyclic** if it has a cycle—an unbroken series of nodes with no repeating nodes or edges that connects back to itself. Graphs without cycles are **acyclic**.



CYCLIC GRAPH ACYCLIC GRAPH

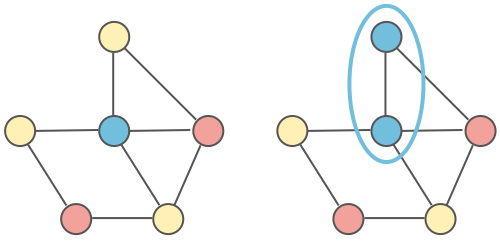
Weighted or unweighted

If a graph is **weighted**, each edge has a "weight." The weight could, for example, represent the distance between two locations, or the cost or time it takes to travel between the locations.



Legal coloring

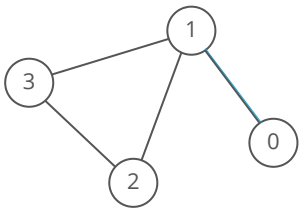
A **graph coloring** is when you assign colors to each node in a graph. A **legal coloring** means no adjacent nodes have the same color:



LEGAL COLORING ILLEGAL COLORING

Representations

There are a few different ways to store graphs. Let's take this graph as an example:



Edge list

A list of all the edges in the graph:

```
graph = [[0, 1], [1, 2], [1, 3], [2, 3]]
```
Python 3.6

Since node `3` has edges to nodes `1` and `2`, `[1, 3]` and `[2, 3]` are in the edge list.

> Sometimes it's helpful to pair our edge list with a list of all the *nodes*. For example, what if a node doesn't have *any* edges connected to it? It wouldn't show up in our edge list at all!

Adjacency list

A list where the index represents the node and the value at that index is a list of the node's neighbors:

```
graph = [
    [1],
    [0, 2, 3],
    [1, 3],
    [1, 2],
]
```
Python 3.6

Since node `3` has edges to nodes `1` and `2`, `graph[3]` has the adjacency list `[1, 2]`.

We could also use a dictionary (/concept/hash-map) where the keys represent the node and the values are the lists of neighbors.

```
graph = {
    0: [1],
    1: [0, 2, 3],
    2: [1, 3],
    3: [1, 2],
}
```

Python 3.6

This would be useful if the nodes were represented by strings, objects, or otherwise didn't map cleanly to list indices.

Adjacency matrix

A matrix of `0`s and `1`s indicating whether node `x` connects to node `y` (`0` means no, `1` means yes).

```
graph = [
    [0, 1, 0, 0],
    [1, 0, 1, 1],
    [0, 1, 0, 1],
    [0, 1, 1, 0],
]
```

Python 3.6

Since node `3` has edges to nodes `1` and `2`, `graph[3][1]` and `graph[3][2]` have value `1`.

Algorithms

BFS and DFS

You should know breadth-first search (BFS) (/concept/bfs) and depth-first search (DFS) (/concept/dfs) down pat so you can code them up quickly.

Lots of graph problems can be solved using just these traversals:

Is there a path between two nodes in this undirected graph? Run DFS or BFS from one node and see if you reach the other one.

What's the shortest path between two nodes in this undirected, unweighted graph? Run BFS from one node and backtrack once you reach the second. *Note:* BFS always finds the shortest path, assuming the graph is undirected and unweighted. DFS does *not* always find the shortest path.

Can this undirected graph be colored with two colors? Run BFS, assigning colors as nodes are visited. Abort if we ever try to assign a node a color different from the one it was assigned earlier.

Does this undirected graph have a cycle? Run BFS, keeping track of the number of times we're visiting each node. If we ever visit a node twice, then we have a cycle.

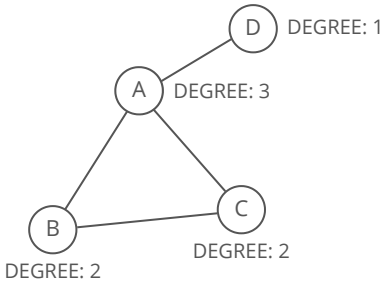
Advanced graph algorithms

If you have lots of time before your interview, these advanced graph algorithms pop up occasionally:

- **Dijkstra's Algorithm:** Finds the shortest path from one node to all other nodes in a *weighted* graph.
- **Topological Sort:** Arranges the nodes in a *directed*, *acyclic* graph in a special order based on incoming edges.
- **Minimum Spanning Tree:** Finds the cheapest set of edges needed to reach all nodes in a *weighted* graph.

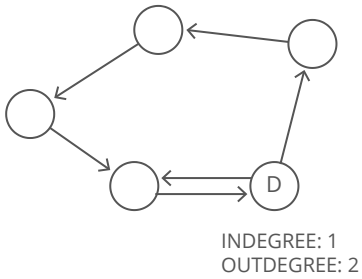
with maximum degree

The **degree** of a node is the number of edges connected to the node.



The **maximum degree** of a graph is the highest degree of all the nodes. The graph above has a maximum degree of 3.

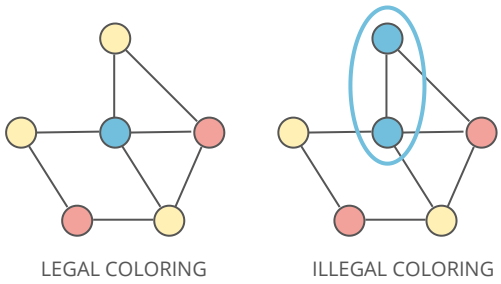
In a directed graph, nodes have an **indegree** and an **outdegree**.



D, find a graph coloring

Graph coloring is when you assign colors to the nodes in a graph.

A **legal coloring** means no adjacent nodes have the same color:



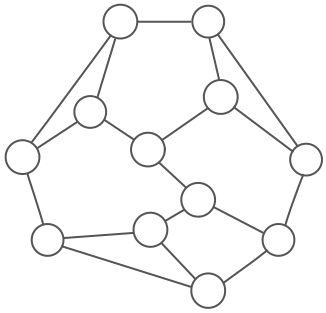
The color could be used literally. Say we're coloring a map. We'll want to color adjacent states or countries differently so their borders are clear.

The color could also represent some concept or property. For example, say we have a graph where nodes represent university classes and edges connect classes with overlapping students. We could use colors to represent the scheduled class exam time. In an illegal coloring, a student could be booked for multiple exams at once!

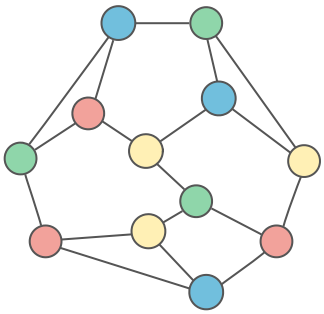
***Edge* coloring** is less common, but it's also a thing. A legal *edge* coloring means no nodes have two edges with the same color.

using at most $D + 1$ colors.

For example:



This graph's maximum degree (D) is 3, so we have 4 colors ($D + 1$). Here's one possible coloring:



Graphs are represented by a list of N node objects, each with a label, a set of neighbors, and a color:

Python 3.6

```
class GraphNode:

    def __init__(self, label):
        self.label = label
        self.neighbors = set()
        self.color = None


a = GraphNode('a')
b = GraphNode('b')
c = GraphNode('c')

a.neighbors.add(b)
b.neighbors.add(a)
b.neighbors.add(c)
c.neighbors.add(b)

graph = [a, b, c]
```

Gotchas

$D + 1$ colors is always enough. Does your function ever need more colors than that?

Does your function go through *every* color for *every* node? You can do better. You don't want $N * D$ in your final runtime.

We can color a graph in **linear time and space** (on the number of nodes, edges and/or the maximum degree).

What if the input graph has a **loop**? Does your function handle that reasonably?

Breakdown

Let's take a step back. Is it always *possible* to find a legal coloring with $D + 1$ colors?

Let's think about it. Each node has at most D neighbors, and we have $D + 1$ colors. So, if we look at any node, there's always at least one color that's not taken by its neighbors.

So yes—$D + 1$ is always enough colors for a legal coloring.

Still not convinced? We can prove this more formally using induction.

In general, an **inductive proof** uses 2 steps to prove a claim is true for all (usually positive) integers:

1. A **base case** showing the claim is true for the first number (1 or 0)
2. An **inductive step** showing that if we *assume* the claim is true for a number n, then the claim is *also* true for $n + 1$

If the claim is true for the first number, *and* for any *next* number, then it must be true for *all numbers*.

So let's prove this claim:

A legal coloring with $D + 1$ colors is always possible for a graph of N nodes with maximum degree D.

For our **base case**, we need to show this claim holds for a graph with 1 node.

A graph with 1 node has 0 edges, so the maximum degree D is 0. That means we have 1 color ($D + 1 = 1$).

Can we color that node with the one available color? Definitely, since it doesn't have any adjacent nodes that could make the coloring illegal.

> What about if the graph has a loop! We're assuming our graph doesn't have these because otherwise, there's no possible legal coloring. Keep this edge case in mind though: we'll need to check for loops in our input graph and throw an error if we find any.

So we've proven our base case. Now for the **inductive step**.

This'll be our assumption:

A $D + 1$ coloring is possible for a graph with N nodes.

Can we show that *if* our assumption is true, it *must also be true* for a graph with $N + 1$ nodes?

Let's say we have a graph with $N+1$ nodes and maximum degree D. We're not sure yet if we can color it with $D+1$ colors, right?

Ok, so let's remove a node and its edges from the graph. Any node. Now we have a graph with N nodes.

What happened to D by removing a node?

D either stayed the same or went down. We *removed* edges, so there's no way D went up.

So now we have a graph with N nodes and maximum degree at most D. Can we color this graph with $D+1$ colors?

Yup! That's exactly our assumption! As part of the inductive step, we've assumed that we can color this graph with $D+1$ colors. So let's go ahead and color the graph.

Now all we have to do is add back in the node we removed (so we have $N+1$ nodes again) and show we can find a valid color for that node.

When we add the node we removed back in, what's the most neighbors it can have?

D. We started with a graph with $N+1$ nodes and maximum degree D, and we just rebuilt that graph.

In the worst case, the node we add back in *will* have D neighbors, and they'll all have different colors. Not a problem. We have $D+1$ colors to choose from, so at least one color is still free. We'll use that one for this node. Bam.

Okay, so there is always a legal coloring. Now, how can we find it?

A brute force⤵

A **brute force** algorithm finds a solution by trying *all* possible answers and picking the best one.

Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?

You could try running through all potential coin combinations and pick the one that adds to 67 cents using the fewest coins. That's a *brute force* algorithm, since you're trying *all* possible ways to make change.

Here are a few other brute force algorithms:

- Trying to fit as many overlapping meetings as possible in a conference room? Run through all possible schedules, and pick the schedule that fits the most meetings in the room.
- Trying to find the cheapest route through a set of cities? Try all possible routes and pick the cheapest one.
- Looking for a minimum spanning tree in a graph (/concept/graph)? Try all possible sets of edges, and pick the cheapest set that's also a tree.

Brute force solutions are usually *very slow* since they involve testing a huge number of possible answers.

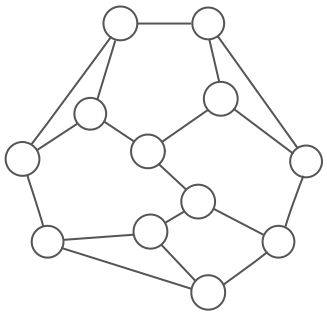
Brute force approaches are rarely the most efficient. Other approaches, like greedy algorithms (/concept/greedy) or dynamic programming (/concept/bottom-up) tend to be faster.

Even so, talking through a brute force solution can be a good first step in a coding interview. It's usually pretty easy to derive, so it allows you to quickly make progress and come up with *something* that works. From there, you have some helpful boundaries for refining your algorithm—you're only interested in solutions that are faster (and/or more space efficient) than the brute force solution you've already come up with.

approach would be to try *every possible combination of colors* until we find a legal coloring. Our steps would be:

1. For each possible graph coloring,
2. If the coloring is legal, then return it
3. Otherwise, move on to the next coloring

For example, looking back at our sample graph:



D is 3, so we can use 4 colors. The combinations of 4 colors for all 12 nodes are:

```
red, red, red, red, red, red, red, red, red, red, red, red
red, red, red, red, red, red, red, red, red, red, red, yellow
red, red, red, red, red, red, red, red, red, red, red, green
red, red, red, red, red, red, red, red, red, red, red, blue
red, red, red, red, red, red, red, red, red, red, yellow, red
...
blue, blue, blue, blue, blue, blue, blue, blue, blue, blue, blue, green
blue, blue, blue, blue, blue, blue, blue, blue, blue, blue, blue, blue
```

And we'd keep trying combinations until we reach one that legally colors the graph.

This would work. But what's the complexity?

Here we'd try 4^{12} combinations (every combination of 4 colors for 12 nodes). In general, we'll have to check $O(D^N)$ colorings. And that's not all—*each* time we try a coloring, we have to check *all* M *edges* to see if the vertices at both ends have different colors. So, our runtime is $O(M * D^N)$. That's **exponential time** since N is in an exponent.

> Since this algorithm is so inefficient, it's probably not what the interviewer is looking for. With practice, it gets easier to quickly judge if an approach will be inefficient. Still, sometimes it's a good idea in an interview to *briefly* explain inefficient ideas and *why* you think they're inefficient. It shows rigorous thinking.

How can we color the graph more efficiently?

Well, we're wasting a lot of time trying color combinations that don't work. If the first 2 nodes are neighbors, we shouldn't try any combinations where the first 2 colors are the same.

Instead of assigning all the colors at once, what if we colored the nodes *one by one*?

We could assign a color to the first node, then find a *legal* color for the second node, then for the third node, and keep going node by node.

Python 3.6

```
def color_graph(graph, colors):
    for node in graph:
        # Get the node's neighbors' colors, as a set so we
        # can check if a color is illegal in constant time
        illegal_colors = set([
            neighbor.color
            for neighbor in node.neighbors
            if neighbor.color
        ])
        legal_colors = [
            color
            for color in colors
            if color not in illegal_colors
        ]

        # Assign the first legal color
        node.color = legal_colors[0]
```

> Is it possible we'll back ourselves into a corner somehow and run out of colors for some nodes?
>
> Let's think back to our earlier argument about whether a coloring always exists:
>
> > *"Each node has at most D neighbors, and we have $D + 1$ colors. So, if we look at any node, there's always at least one color that's not taken by its neighbors."*
>
> That reasoning works here, too! So no—we'll never back ourselves into a corner.

Ok, what's our runtime?

We're iterating through each node in the graph, so the loop body executes N times. In each iteration of the loop:

1. We look at the current node's neighbors to figure out what colors are already taken. That's $O(D)$, since any given node can have up to D neighbors.
2. Then, we look at all the colors (there are $O(D)$ of them) to see which ones are available.
3. Finally, we pick the first color that's free and assign it to the node ($O(1)$).

So our runtime is $N * (D + D + 1)$, which is $O(N * D)$.

Can we tighten our analysis a bit here? Take a look at step 1, where we collect the neighbors' colors:

We said looking at each node's neighbors was $O(D)$ since each node can have *at most* D neighbors . . . but each node might have way fewer neighbors than that.

Can we say anything about the *total* number of neighbors we'll look at across *all* of the loop iterations? How many neighbors are there in the entire graph?

Each *edge* creates two neighbors: one for each node on either end. So when our code looks at *every* neighbor for *every* node, it looks at $2 * M$ neighbors in all. With $O(M)$ neighbors, collecting all the colors over the entire for loop takes $O(M)$ time.

Using this tighter analysis, we've taken our runtime from $N * (D + D + 1)$ down to $N * (D + 1) + M$. That's $O((N * D) + M)$ time.

Of course, that complexity doesn't look any faster, at least not asymptotically. But in the underlying expression, we've gotten rid of one of the two $N * D$ factors.

Can we get rid of the other one to bring our complexity down to linear

> In general, **linear time** means an algorithm runs in $O(n)$ time, where n is the size of the input.
>
> In graph problems like this, we usually express complexity in terms of the number of nodes (N) and edges (M) in the input graph. So, when we say linear time here, we mean $O(N + M)$.

time?

The remaining $N * D$ factor comes from step 2: looking at every color for every node to populate `legal_colors`.

Do we *have* to look at *every* color for every node?

When we're coloring a node, we just need *one* color that hasn't been taken by any of the node's neighbors. We can stop looking at colors as soon as we find one:

Python 3.6

```
def color_graph(graph, colors):
    for node in graph:
        # Get the node's neighbors' colors, as a set so we
        # can check if a color is illegal in constant time
        illegal_colors = set([
            neighbor.color
            for neighbor in node.neighbors
            if neighbor.color
        ])

        # Assign the first legal color
        for color in colors:
            if color not in illegal_colors:
                node.color = color
                break
```

Okay, now what's the time cost of assigning the first legal color to every node (the whole last block)?

We'll try at most `len(illegal_colors) + 1` colors in total. That's how many we'd need if we happen to test all the colors in `illegal_colors` first, before finally testing the one *legal* color last.

> Remember the "+1" we get from testing the one *legal* color last! It's going to be important in a second.

How many colors are in `illegal_colors`? It's *at most* the number of neighbors, if each neighbor has a different color.

Let's use that trick of looking at all of the loop iterations together. In *total*, over the course of *the entire loop*, how many neighbors are there?

Well, each of our M *edges* adds two neighbors to the graph: one for each node on either end. So that's $2 * M$ neighbors in total. Which means $2 * M$ illegal colors in total.

But remember: we said we'd try as many as `len(illegal_colors) + 1` colors per node. We still have to factor in that "+1"! Across all N of our nodes, that's an additional N colors. So we try $2 * M + N$ colors in total across all of our nodes.

That's $O(M + N)$ time for assigning the first legal color to every node. Add that to the $O(M)$ for finding all the illegal colors, and we get $O(M + N)$ time in total for our graph coloring function.

Is this the fastest runtime we can get? We'll *have* to look at every node ($O(N)$) and every edge ($O(M)$) at least once, so yeah, we can't get any better than $O(N + M)$.

How about our space cost?

The only data structure we allocate with non-constant space is the set of illegal colors. What's the most space that ever takes up?

In the worst case, the neighbors of the node with the maximum degree will all have different colors, so our space cost is $O(D)$.

Before we're done, what about edge cases?

For graph problems in general, edge cases are:

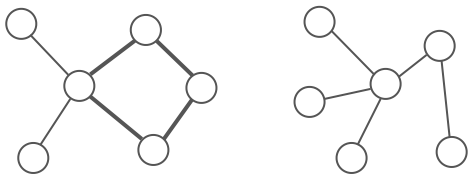
- nodes with no edges
- cycles
- loops

What if there are nodes with no edges? Will our function still color every node?

Yup, no problem. Isolated nodes tend to cause problems when we're *traversing* a graph (starting from one node and "walking along" edges to other nodes, like we do in a depth-first or breadth-first search). We're not doing that here—instead, we're *iterating over a list of all the nodes*.

What if the graph has a cycle?

A graph has a **cycle** if there is a path from any node to itself via other nodes.



Will our function still work?

Yes, it will. Cycles also tend to cause problems with graph *traversal*, because we can end up in infinite loops (going around and around the cycle). But we're not actually traversing our graph here.

What if the graph has a loop?

A **loop** in a graph is an edge where both ends connect to the same node:

That's a problem. A node with a loop is adjacent to itself, so it can't have the same color as . . . itself. So it's impossible to "legally color" a node with a loop. So we should throw an error.

How can we detect loops?

We know a node has a loop if the node is in its own set of neighbors.

Solution

We go through the nodes in one pass, assigning each node the first legal color we find.

How can we be sure we'll always have at least one legal color for every node? In a graph with maximum degree D, each node has at most D neighbors. That means there are at most D colors taken by a node's neighbors. And we have $D + 1$ colors, so there's always at least one color left to use.

When we color each node, we're careful to stop iterating over colors as soon as we find a legal color.

Python 3.6

```
def color_graph(graph, colors):
    for node in graph:
        if node in node.neighbors:
            raise Exception('Legal coloring impossible for node with loop: %s' %
                            node.label)

        # Get the node's neighbors' colors, as a set so we
        # can check if a color is illegal in constant time
        illegal_colors = set([
            neighbor.color
            for neighbor in node.neighbors
            if neighbor.color
        ])

        # Assign the first legal color
        for color in colors:
            if color not in illegal_colors:
                node.color = color
                break
```

Complexity

$O(N + M)$ time where N is the number of nodes and M is the number of edges.

The runtime might not *look* linear because we have outer and inner loops. The trick is to look at each step and think of things in terms of the *total number of edges* (M) wherever we can:

- We check if each node appears in its own set of neighbors. Checking if something is in a set is $O(1)$, so doing it for all N nodes is $O(N)$.
- When we get the illegal colors for each node, we iterate through that node's neighbors. So *in total*, we cross each of the graphs M edges twice: once for the node on either end of each edge. $O(M)$ time.
- When we assign a color to each node, we're careful to stop checking colors as soon as we find one that works. In the worst case, we'll have to check one more color than the total number of neighbors. Again, each edge in the graph adds two neighbors—one for the node on either end—so there are $2 * M$ neighbors. So, *in total*, we'll have to try $O(N + M)$ colors.

Putting all the steps together, our complexity is $O(N + M)$.

What about space complexity? The only thing we're storing is the `illegal_colors` set. In the worst case, all the neighbors of a node with the maximum degree (D) have different colors, so our set takes up $O(D)$ space.

Bonus

1. Our solution runs in $O(N + M)$ time but takes $O(D)$ space. Can we get down to $O(1)$ space?
2. Our solution finds a legal coloring, but there are usually *many* legal colorings. What if we wanted to optimize a coloring to use *as few colors as possible*?

> The lowest number of colors we can use to legally color a graph is called the **chromatic number**.
>
> There's no known polynomial time solution for finding a graph's chromatic number. It might be impossible, or maybe we just haven't figured out a solution yet.
>
> We can't even determine in polynomial time if a graph can be colored using a given k colors. Even if k is as low as 3.
>
> We care about **polynomial time** solutions (n raised to a constant power, like $O(n^2)$) because for large ns, polynomial time algorithms are more practical to actually use than higher runtimes like **exponential time** (a constant raised to the power of n, like $O(2^n)$). Computer scientists usually call algorithms with polynomial time solutions **feasible**, and problems with worse runtimes **intractable**.
>
> The problem of determining if a graph can be colored with k colors is in the class of problems called **NP** (nondeterministic polynomial time). This means that in polynomial time, we can *verify a solution is correct* but we can't *come up with a solution*. In this case, if we have a graph that's already colored with k colors we verify the coloring uses k colors and is legal, but we can't take a graph and a number k and determine if the graph can be colored with k colors.
>
> If you can find a solution or prove a solution doesn't exist, you'll win a $1,000,000 Millennium Problem Prize (http://www.claymath.org/millennium-problems/p-vs-np-problem).
>
> For coloring a graph using as few colors as possible, we don't have a feasible solution. For real-world problems, we'd often need to check so many possibilities that we'll never be able to use brute-force no matter how advanced our computers become.
>
> One way to reliably reduce the number of colors we use is to use the greedy algorithm but **carefully order the nodes**. For example, we can prioritize nodes based on their degree, the number of colored neighbors they have, or the number of uniquely colored neighbors they have.

What We Learned

We used a greedy

> A **greedy** algorithm builds up a solution by choosing the option that looks the best at every step.
>
> Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?
>
> Whenever picking which coin to use, you'd take the highest-value coin you could. A quarter, another quarter, then a dime, a nickel, and finally two pennies. That's a *greedy* algorithm, because you're always *greedily* choosing the coin that covers the biggest portion of the remaining amount.
>
> Some other places where a greedy algorithm gets you the best solution:
>
> - Trying to fit as many overlapping meetings as possible in a conference room? At each step, schedule the meeting that *ends* earliest.
> - Looking for a minimum spanning tree in a graph (/concept/graph)? At each step, greedily pick the cheapest edge that reaches a new vertex.
>
> **Careful: sometimes a greedy algorithm *doesn't* give you an optimal solution:**
>
> - When filling a duffel bag with cakes of different weights and values (/question/cake-thief), choosing the cake with the highest value per pound doesn't always produce the best haul.
> - To find the cheapest route visiting a set of cities, choosing to visit the cheapest city you haven't been to yet doesn't produce the cheapest overall itinerary.
>
> Validating that a greedy strategy always gets the best answer is tricky. Either prove that the answer produced by the greedy algorithm is as good as an optimal answer, or run through a rigorous set of test cases to convince your interviewer (and yourself) that its correct.

approach to build up a correct solution in one pass through the nodes.

This brought us *close* to the optimal runtime, but we also had to take that last step of iterating over the colors *only until we find a legal color*. Sometimes stopping a loop like that is just a premature optimization that doesn't bring down the final runtime, but here it actually made our

runtime linear!

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.