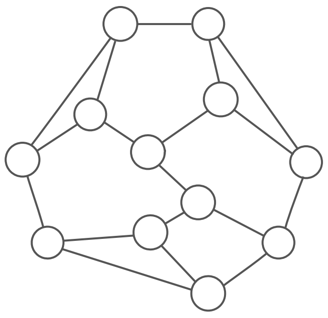
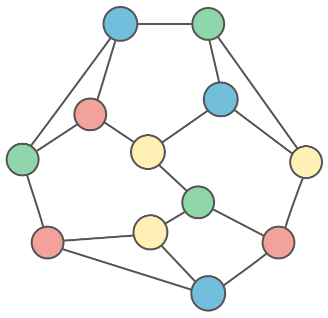
Interview Cake

Given an undirected graph⮧ with maximum degree⮧ D, find a graph coloring⮧ 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 vector of N node objects, each with a label, an unordered set of neighbors, and a color:

C++

```
class GraphNode
{
private:
    string label_;
    unordered_set<GraphNode*> neighbors_;
    string color_;

public:
    GraphNode(const string& label) :
        label_(label),
        neighbors_(),
        color_()
    {
    }

    const string& getLabel() const
    {
        return label_;
    }

    const unordered_set<GraphNode*> getNeighbors() const
    {
        return neighbors_;
    }

    void addNeighbor(GraphNode& neighbor)
    {
        neighbors_.insert(&neighbor);
    }

    bool hasColor() const
    {
        return !color_.empty();
    }

    const string& getColor() const
    {
        if (hasColor()) {
            return color_;
        }
        else {
            throw logic_error("GraphNode is not marked with color");
        }
    }

    void setColor(const string& color)
    {
        color_ = color;
    }
};

GraphNode a("a");
GraphNode b("b");
GraphNode c("c");

a.addNeighbor(b);
b.addNeighbor(a);
b.addNeighbor(c);
c.addNeighbor(b);

vector<GraphNode*> 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.

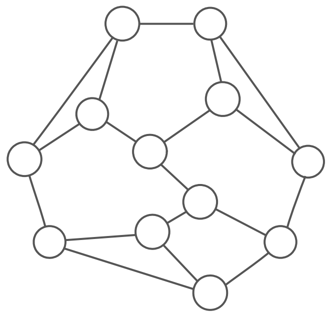
Still not convinced? We can prove this more formally using induction.↓

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

A brute force↓ 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.

C++ ▾

```cpp
void colorGraph(const vector<GraphNode*>& graph, const vector<string>& colors)
{
    for (auto node : graph) {

        // get the node's neighbors' colors, as a set so we
        // can check if a color is illegal in constant time
        unordered_set<string> illegalColors;
        for (const auto neighbor : node->getNeighbors()) {
            if (neighbor->hasColor()) {
                illegalColors.insert(neighbor->getColor());
            }
        }

        unordered_set<string> legalColors;
        for (const auto& color : colors) {
            if (illegalColors.find(color) == illegalColors.cend()) {
                legalColors.insert(color);
            }
        }

        // assign the first legal color
        node->setColor(*legalColors.begin());
    }
}
```

> 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 each 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 the two $N * D$ factors.

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

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

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:

C++ ▾

```
void colorGraph(const vector<GraphNode*>& graph, const vector<string>& colors)
{
    for (auto node : graph) {

        // get the node's neighbors' colors, as a set so we
        // can check if a color is illegal in constant time
        unordered_set<string> illegalColors;
        for (const auto neighbor : node->getNeighbors()) {
            if (neighbor->hasColor()) {
                illegalColors.insert(neighbor->getColor());
            }
        }

        // assign the first legal color
        for (const auto& color : colors) {
            if (illegalColors.find(color) == illegalColors.cend()) {
                node->setColor(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 `illegalColors.size() + 1` colors in total. That's how many we'd need if we happen to test all the colors in `illegalColors` 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 `illegalColors`? 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* add 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 `illegalColors.size() + 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 unordered 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 vector of all the nodes.*

What if the graph has a cycle?↓ 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?↓

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 unordered 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.

C++ ▼

```cpp
void colorGraph(const vector<GraphNode*>& graph, const vector<string>& colors)
{
    for (auto node : graph) {
        const auto& neighbors = node->getNeighbors();

        if (neighbors.find(node) != neighbors.end()) {
            ostringstream errorMessage;
            errorMessage << "Legal coloring impossible for node with loop: "
                    << node->getLabel();
            throw invalid_argument(errorMessage.str());
        }

        // get the node's neighbors' colors, as a set so we
        // can check if a color is illegal in constant time
        unordered_set<string> illegalColors;
        for (const auto neighbor : neighbors) {
            if (neighbor->hasColor()) {
                illegalColors.insert(neighbor->getColor());
            }
        }

        // assign the first legal color
        for (const auto& color : colors) {
            if (illegalColors.find(color) == illegalColors.cend()) {
                node->setColor(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 unordered set of neighbors. Checking if something is in an unordered 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 `illegalColors` unordered set. In the worst case, all the neighbors of a node with the maximum degree (D) have different colors, so our unordered 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 and 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↓ 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!

Want more coding interview help?

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