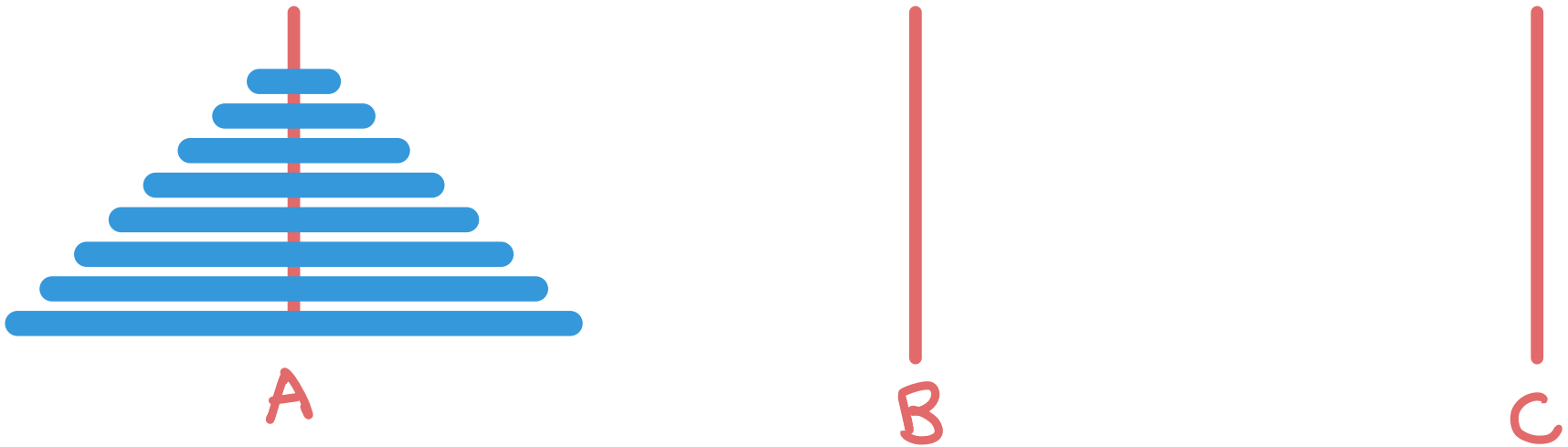
Towers of Hanoi

3 posts A, B, C. n rings of n incr. sizes (top-to-bottom) on A

can move ring from top of one post to top of another

rule: a ring cannot be placed on a smaller ring



Goal: move all n rings from A to B

Induction & Recursion

welcome to Graduate Algorithms

fundamentalalgorithms.com

Graduate Algorithms, CS580, Spring 2024

Lectures: 11:30AM to 12:20 PM; on Monday, Wednesday and Friday; in WALC 3087. *First two lectures are over Zoom.*

Staff:	Email (@purdue.edu)	Office hours
Kent Quanrud	krq	TBD at LWSN 1211[1]
Tiantian Gong (TA)	gong146	Tuesday, 11AM–12PM, at HAAS 143.
Haoyu Song (TA)	song522	Monday, 3–4PM, at HAAS 143.

Please see the syllabus (page 549) for more information.

Homework

- Homework 0 due January 17.
- Homework 1 due January 24.
- Homework 2 due January 31.
- Homework 3 due February 7.
- Homework 4 due February 21.
- Homework 5 due February 28.
- Homework 6 due March 6.
- Homework 7 due March 20.
- Homework 8 due April 3.
- Homework 9 due April 10.
- Homework 10 due April 17.

Class schedule[2]

1. *January 8.* Searching and sorting, including binary search, merge sort, and lower bounds for sorting (chapter 1). *On zoom:* https://purdue-edu.zoom.us/j/98283084796?pwd=MEE1RlhSdi9BUksyY3lCZWhzcXpNZz09

Please read the syllabus

- will ask for questions next class

Highlights

Awesome TA's

2 midterms, final

weekly homework

drop lowest 20% of HW scores

late/resubmit HW policy

Lecture 1

Concepts & techniques

$\log(x)$ vs 2^x

recursion

divide & conquer

recursion trees

big-O & big-Ω

Algorithms & applications

Binary search

Merge sort

Sorting lower bounds

Designing a recursive algorithm

Divide-and-conquer

1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

① Recursive specification

`merge-sort`$(A[1..n])$: Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

② Implement spec

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

`merge-sort`$(A[1..n])$

```
/* In the base case, the list is so short there is nothing to do. */
1. If n ≤ 1 then return A.                                   ← Base case
/* In the general case, we divide the input in half and recursively sort each half. */
2. Let m = ⌊n/2⌋.
3. B1[1..m] ← merge-sort(A[1..m]).                           ← satisfies recursive spec by induction on n
4. B2[1..n − m] ← merge-sort(A[m + 1..n]).                   ← satisfies recursive spec by induction on n
/* Now we merge the two sorted halves in linear time. */
5. return merge(B1, B2).
```

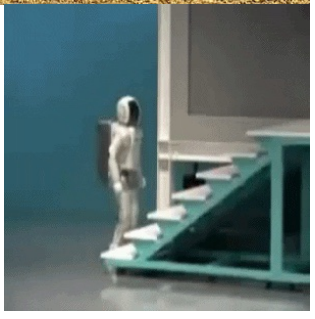
③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

Induction

Climbing stairs

how would you explain to someone how to climb a long staircase?



recognize that each step is identical

Let's take it one step at a time. Say you're at a particular step. Now put one foot on the next step, and use that leg to push your body up and put the other foot on that step, so now your entire mass is on the next step. You can use either foot for the first step. Now if you keep repeating this for each step you will eventually get to the top.

how can we prove this?

"Proof" by induction

Let's take it one step at a time. Say you're at a particular step. Now put one foot on the next step, and use that leg to push your body up and put the other foot on that step, so now your entire mass is on the next step. You can use either foot for the first step. Now if you keep repeating this for each step you will eventually get to the top.

(assume instructions work for 1 step)

Base case: $n = 1$ stair.

General case: $n > 1$ stairs

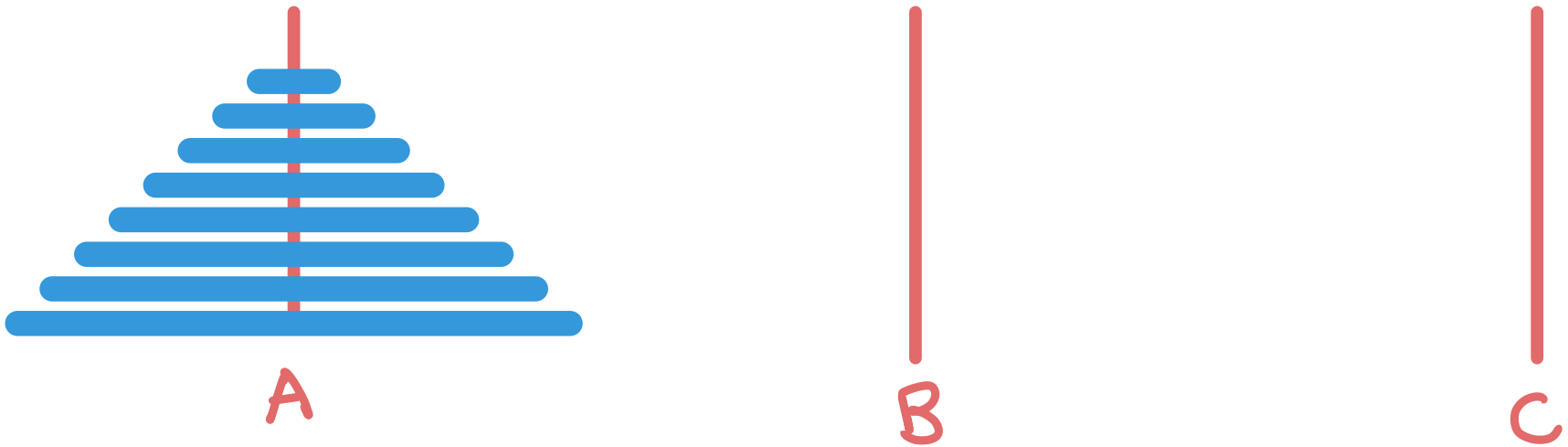
- instructions work for 1 step
- $n-1$ steps remain
- By induction on n, instructions work for staircase of $n-1$ remaining steps

Towers of Hanoi

3 posts A, B, C. n rings of n incr. sizes (top-to-bottom) on A

can move ring from top of one post to top of another

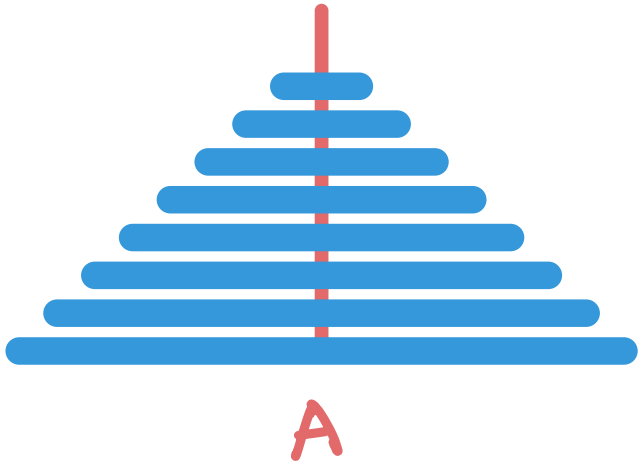
rule: a ring cannot be placed on a smaller ring



Goal: move all n rings from A to B

① Recursive specification

ToH(n, A, B, C): assuming the n smallest rings are on A, moves top n rings from post A to post B



Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

ToH(n, A, B, C): assuming the n smallest rings are on A, moves top n rings from post A to post B

② Implement spec

TOH(n, A, B, C):

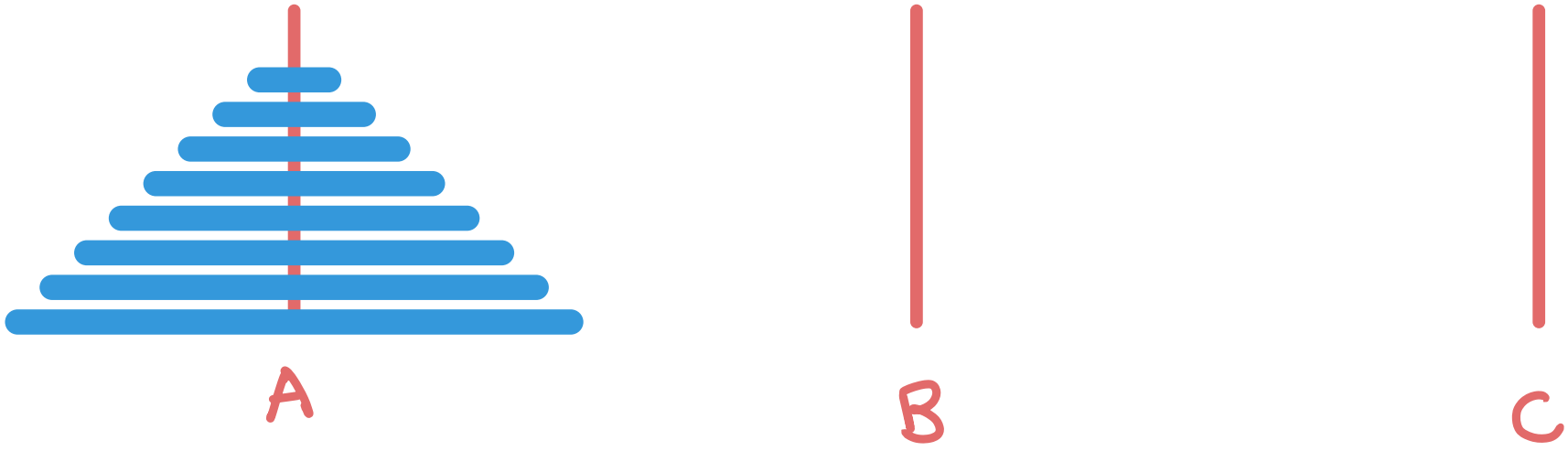
1. if n > 0:
   A. ToH(n-1, A, C, B)
   B. move ring from A to B
   C. ToH(n-1, C, B, A)

Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n



ToH(n, A, B, C): assuming the n smallest rings are on A, moves top n rings from post A to post B

Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

③ Prove correctness

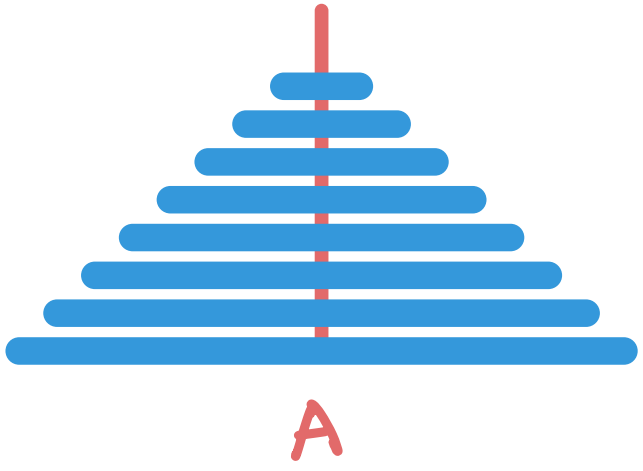
Base case n=0. ✓

Case n>0:

recursive calls correct by induction

⇒ correct for n

```
TOH(n, A, B, C):
1. if n>0:
   A. TOH(n-1, A, C, B)
   B. move ring from A to B
   C. TOH(n-1, C, B, A)
```

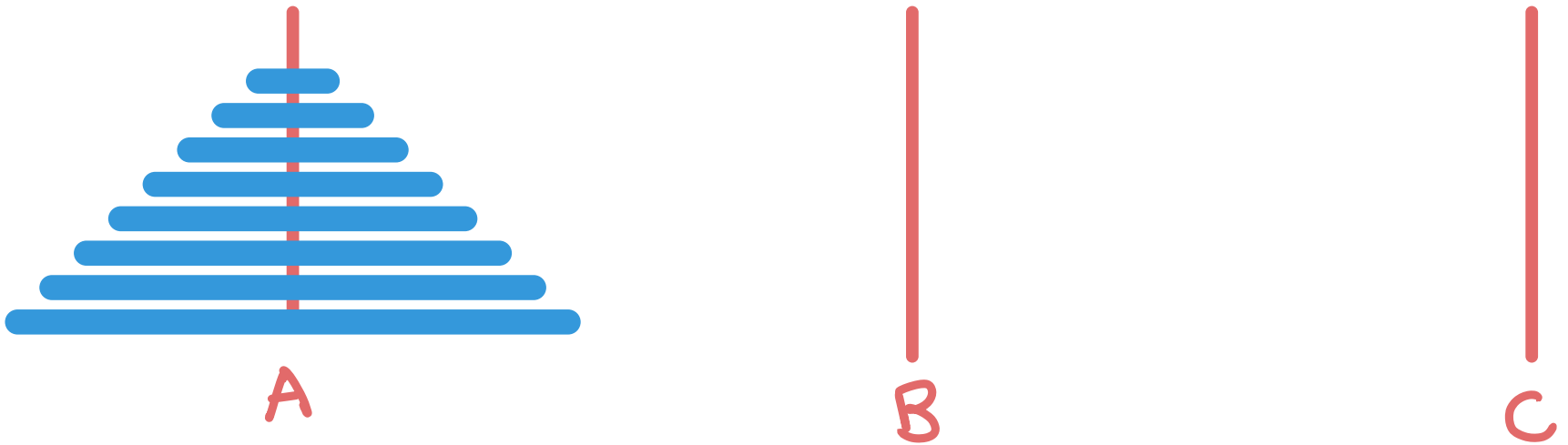


Towers of Hanoi

3 posts A, B, C. n rings of n incr. sizes (top-to-bottom) on A

can move ring from top of one post to top of another

rule: a ring cannot be placed on a smaller ring



Goal: move all n rings from A to B

w/ minimum # of moves

① Recursive specification

ToH(n, A, B, C): assuming the n smallest rings are on A, moves top n rings from post A to post B

w/ the minimum # of moves

everything else stays the same!

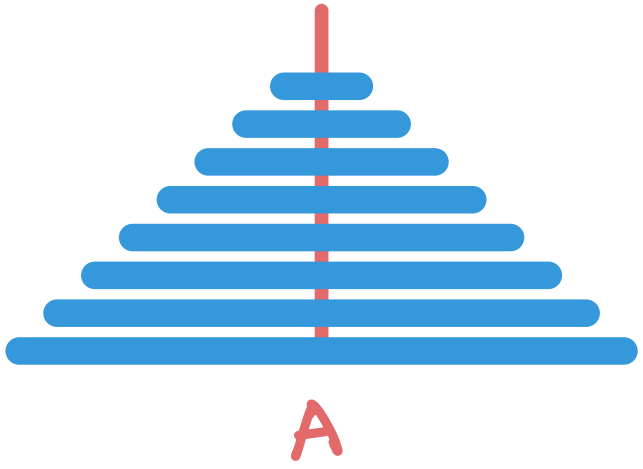
Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

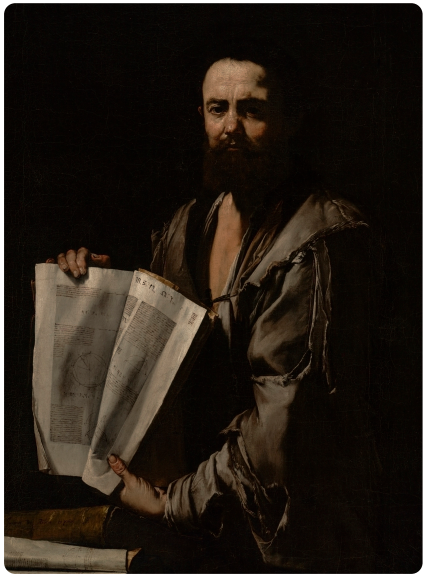
TOH(n, A, B, C):
1. if n>0:
   A. ToH(n-1, A, C, B)
   B. move ring from A to B
   C. ToH(n-1, C, B, A)



Greatest common divisor

given $x, y \in \mathbb{N}$, compute largest integer q that divides x and y

e.g. $GCD(146740, 12180) = 580$



Euclid

let $x \geq y$.

q divides x and y $\iff$ q divides y and $x-y$

① Recursive specification

Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

let $x \geq y$.

q divides x and y <=> q divides y and x-y

GCD(x, y): Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

GCD(x, y): Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

② Implement spec

GCD(x, y)

/ Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y. */*

1. If $y = 0$ then return x.
2. Return GCD$(\max\{x - y, y\}, \min\{x - y, y\})$.

Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

let $x \geq y$.

q divides x and y <=> q divides y and x-y

GCD(x, y): Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

GCD(x, y)

/ Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y. */*

1. If $y = 0$ then return x.
2. Return GCD$(\max\{x - y, y\}, \min\{x - y, y\})$.

③ Prove correctness

Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

let $x \geq y$.

q divides x and y <=> q divides y and x-y

GCD(x,y): Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

GCD(x,y)

/ Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y. */*

1. If $y = 0$ then return x.
2. Return GCD$(\max\{x-y, y\}, \min\{x-y, y\})$.

Designing a recursive algorithm

① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

let $x \geq y$.

q divides x and y <=> q divides y and x-y

Running time

each iteration decreases $x+y$ by 1

$O(x+y)$ time (assuming $O(1)$-time arithmetic)

Is this "linear time"?

x, y take $\log(x)$, $\log(y)$ bits to represent

$O(x+y)$ is *exponential* in the input size

Bad case

x	y
101	2
99	2
97	2
95	
93	6
6	
5	2
3	2
2	1

$\texttt{GCD}(x,y)$: Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

$\underline{\texttt{GCD}(x,y)}$

/ Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y. */*

1. If $y = 0$ then return x.
2. Return $\texttt{GCD}(\max\{x-y, y\}, \min\{x-y, y\})$.

let $x \geq y$ and $r = x \bmod y$

$x = ky + r$ w/ $k \in \mathbb{N}$, $ky \leq x$

q divides x and y $\Leftrightarrow$ q divides y and r

GCD(x,y): Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

$x \geq y$ and $r = x \bmod y$
$x = ky + r$ w/ $k \in \mathbb{N}$, $ky \leq x$
q divides x and y $\Leftrightarrow$ q divides y and r

GCD(x,y)

/ Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y. */*

1. If $y = 0$ then return x.
2. Return GCD$(y, x \bmod y)$.

Proof by induction on y

Base case $y=0$ ✓

General case $y>0$:

GCD(x,y): Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

$x \geq y$ and $r = x \bmod y$
$x = ky + r$ w/ $k \in \mathbb{N}$, $ky \leq x$
q divides x and y $\Leftrightarrow$ q divides y and r

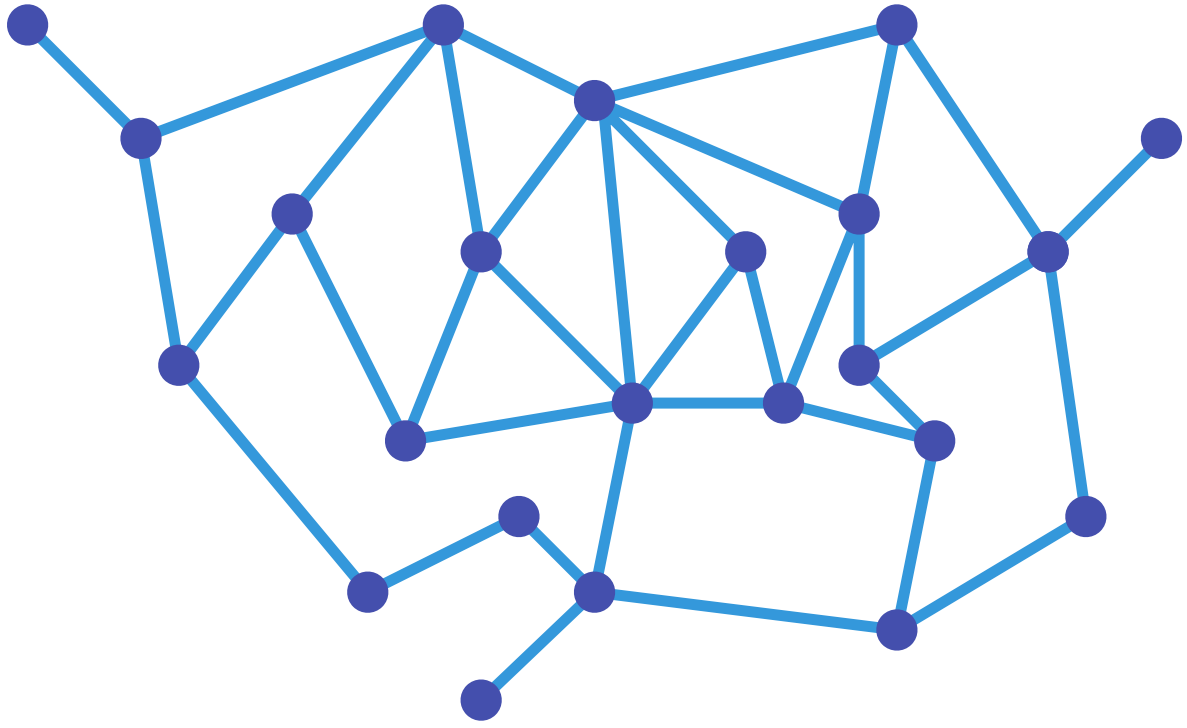
GCD(x,y)

/ Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y. */*

1. If $y = 0$ then return x.
2. Return GCD$(y, x \mod y)$.

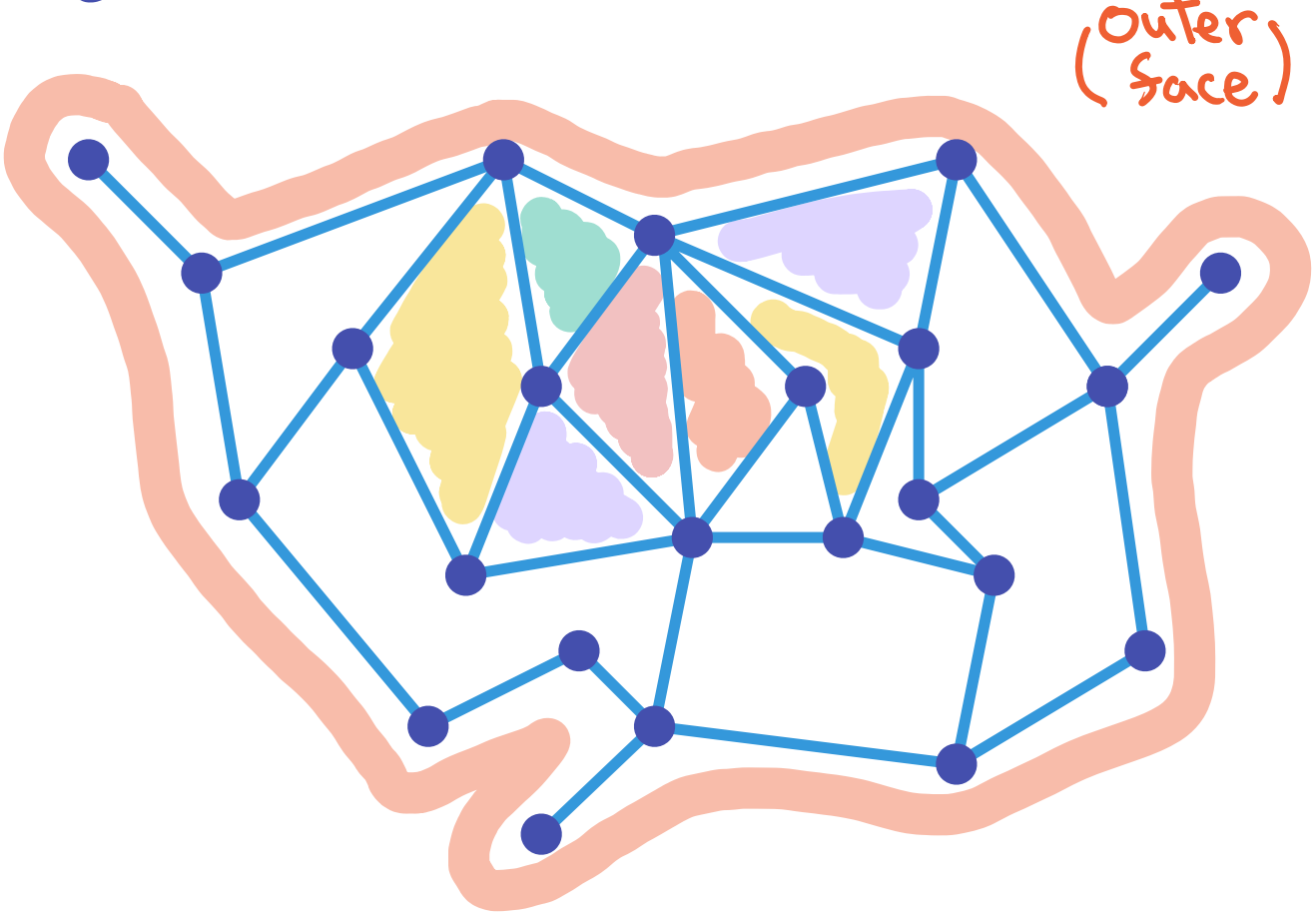
Running time

Planar graphs



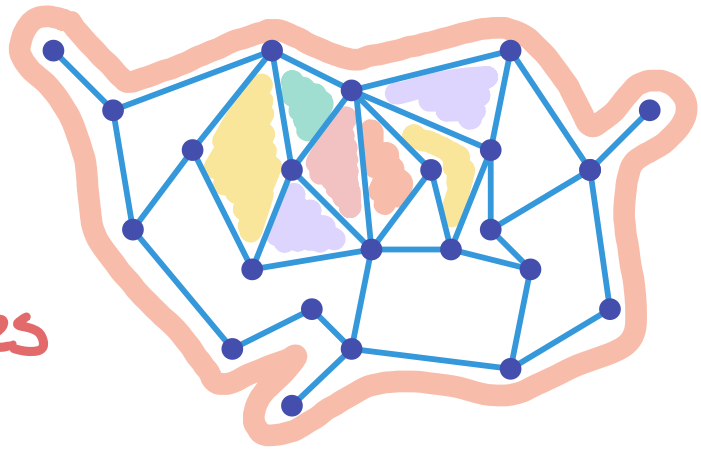
graph can be drawn so that edges only interect at their endpoints (aka embedding in the plane)

Planar graphs



for fixed embedding, graph divides plane into regions called **faces**.

Euler's Formula



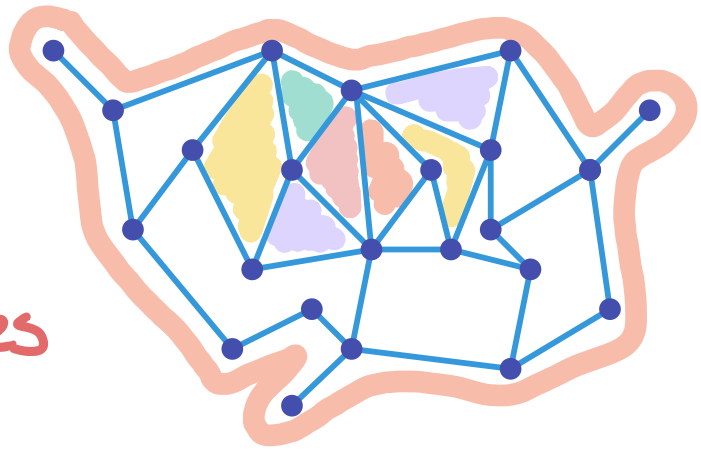
$G=(V,E)$ connected planar graph.

$m=\#$ edges, $n=\#$ vertices, $p=\#$ faces

then $n-m+p=2$

Proof

Euler's Formula



$G=(V,E)$ connected planar graph.

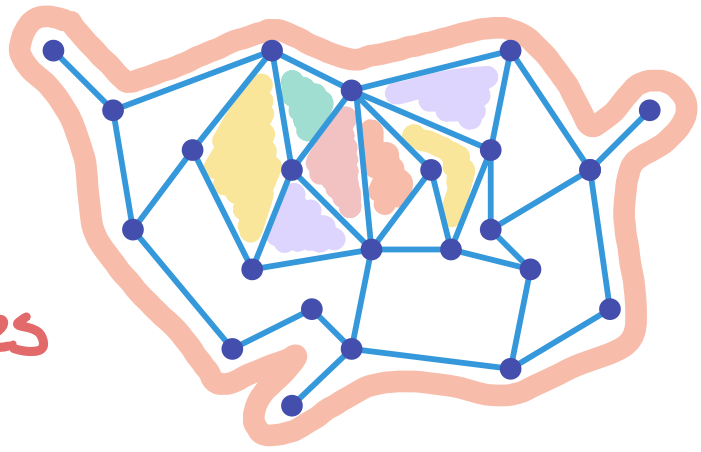
$m=\#$ edges, $n=\#$ vertices, $p=\#$ faces

then $n-m+p=2$

Proof by induction on n

Base case $n=1$

Euler's Formula



$G=(V,E)$ connected planar graph.

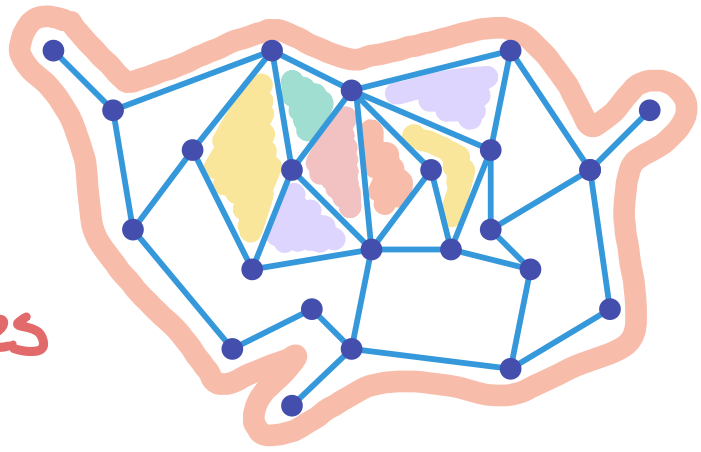
$m=\#$ edges, $n=\#$ vertices, $p=\#$ faces

then $n-m+p=2$

Proof by induction on n

General case $n>1$.

Euler's Formula



$G=(V,E)$ connected planar graph.

$m=\#$ edges, $n=\#$ vertices, $p=\#$ faces

then $n-m+p=2$

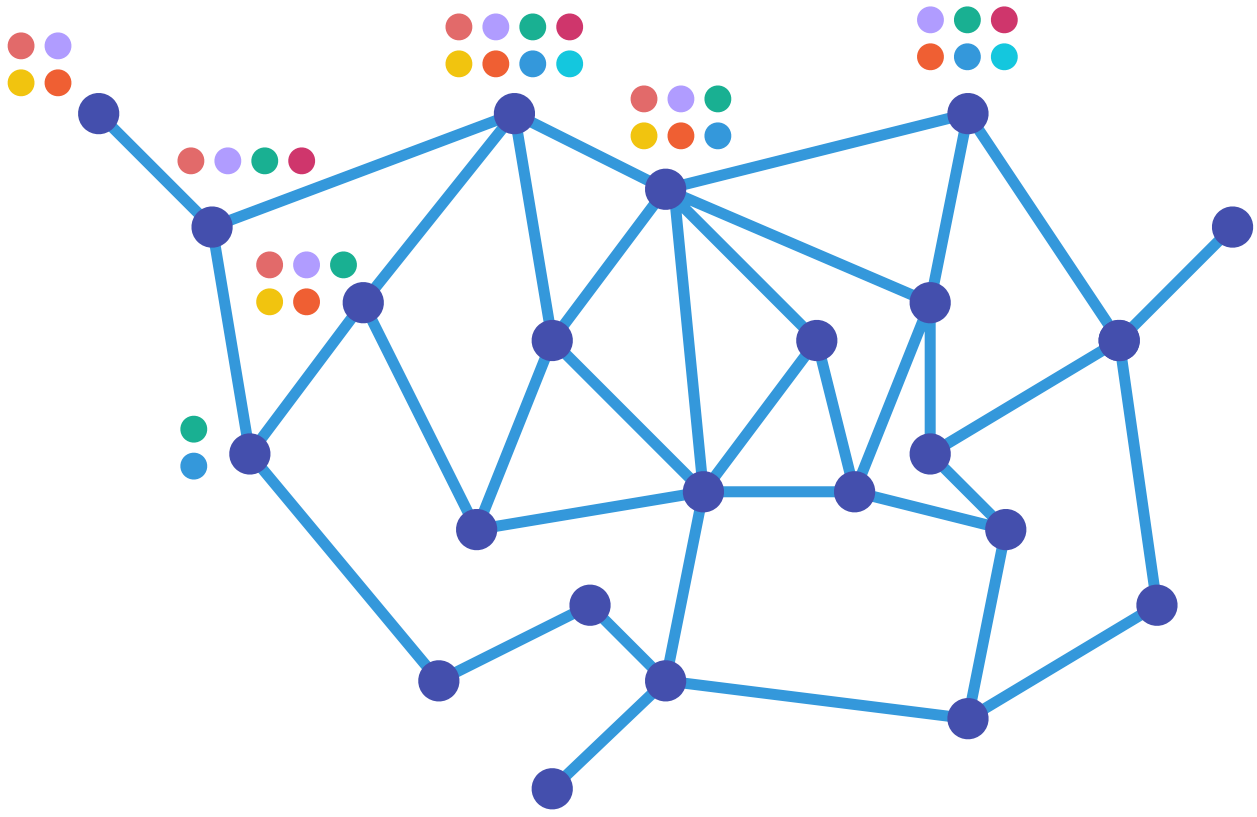
Corollary for simple (no parallel edges or self-loops) planar G, and $n \geq 3$,

$$m \leq 3n-6$$

proof

List coloring

each vertex v has set of colors $C(v)$



Goal: choose color from $C(v)$ for all v so that each edge has 2 diff. colors

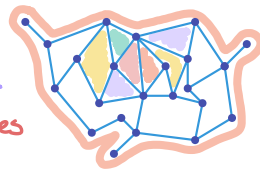
Theorem: all planar graphs are 6-list colorable

Euler's Formula

$G=(V,E)$ connected planar graph.

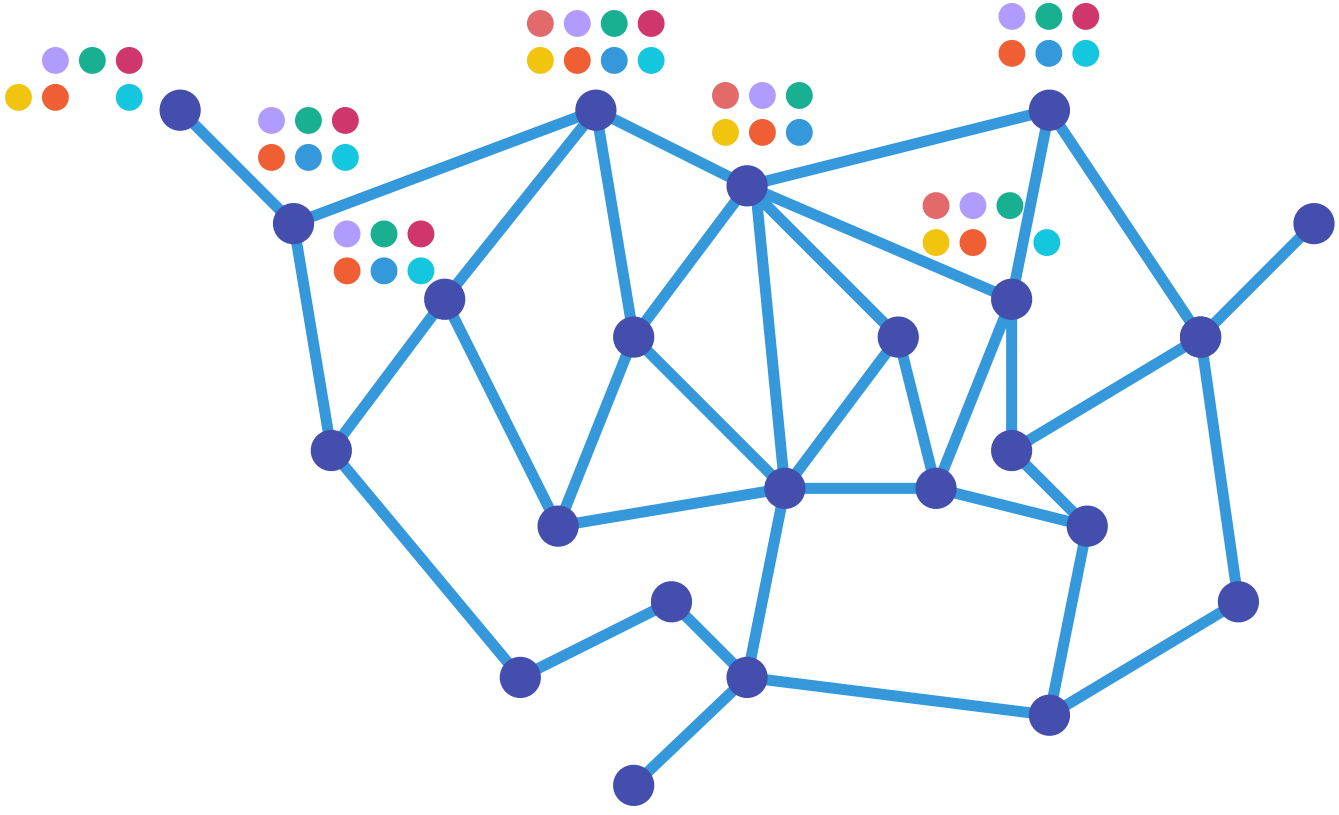
m = # edges, n = # vertices, p = # faces

then $n-m+p=2$



Corollary for (no parallel edges or self-loops) simple planar G, and $n \geq 3$,

$m \leq 3n-6$



Theorem: all planar graphs are 6-list colorable

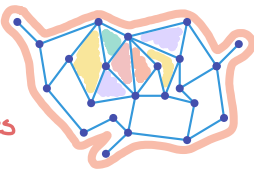
proof

Euler's Formula

$G=(V,E)$ connected planar graph.

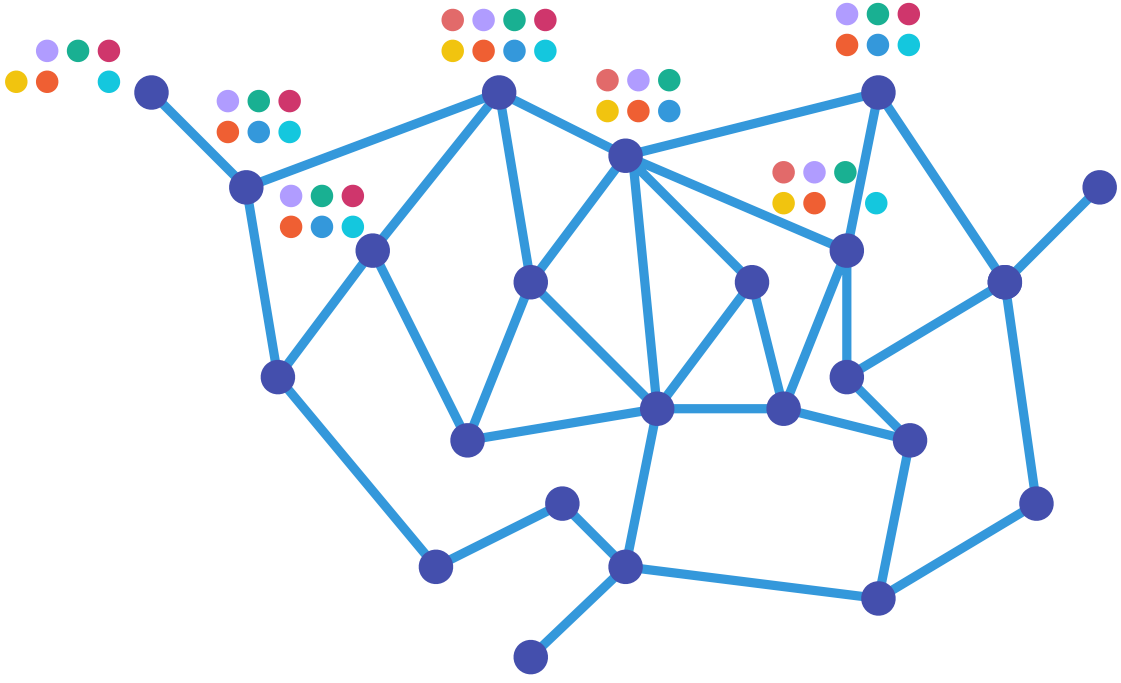
m = # edges, n = # vertices, p = # faces

then $n-m+p=2$

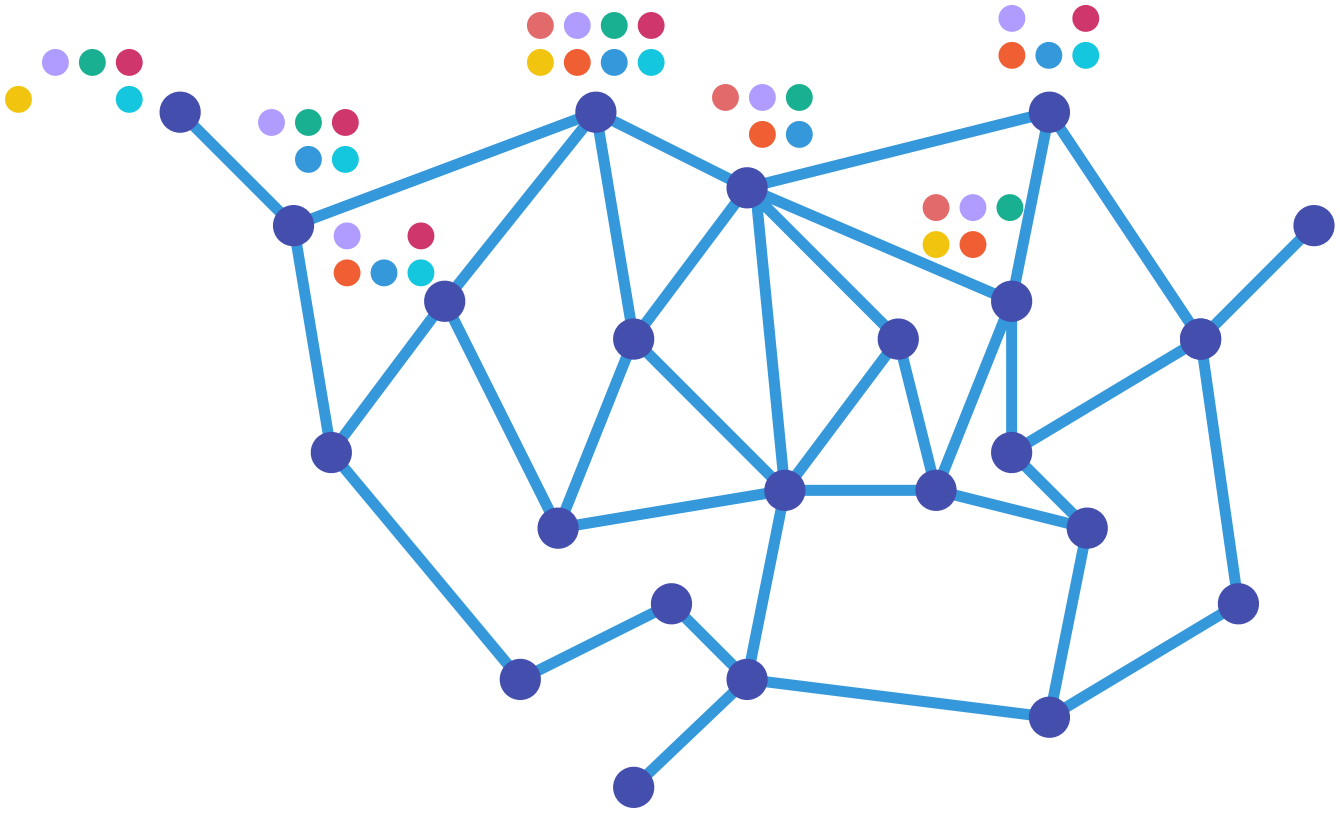


Corollary for (no parallel edges or self-loops) simple planar G, and $n \geq 3$,

$m \leq 3n-6$



Theorem: all planar graphs are 5-list colorable



Theorem: all planar graphs are 5-list colorable

Proof: use stronger induction hypothesis:

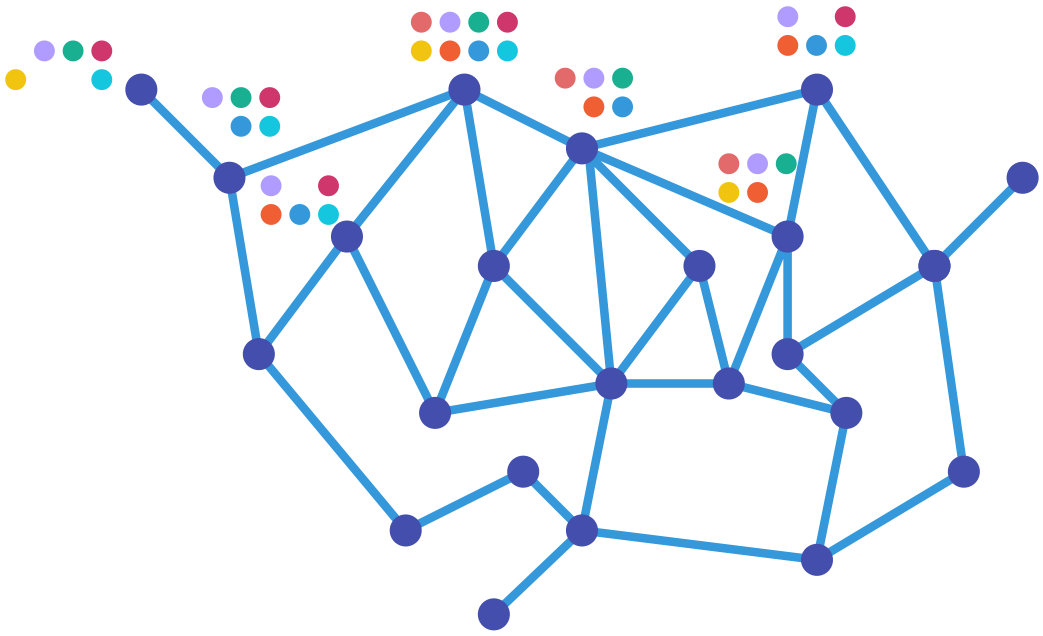
Lemma 2.7. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

(b) $|C(v)| \geq 3$ for all $v \in V_B$.

(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

Then the coloring of x and y can be extended to a proper list coloring of G.



Lemma 2.7. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

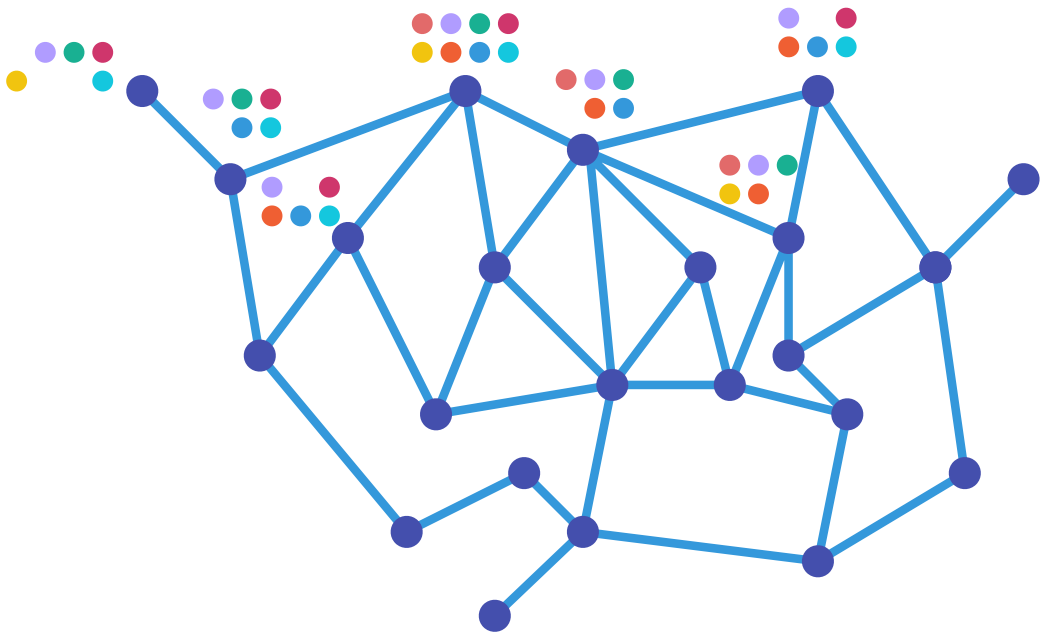
(b) $|C(v)| \geq 3$ for all $v \in V_B$.

(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

Then the coloring of x and y can be extended to a proper list coloring of G.

proof by induction on n.

base case: n=3



Lemma 2.7. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

(b) $|C(v)| \geq 3$ for all $v \in V_B$.

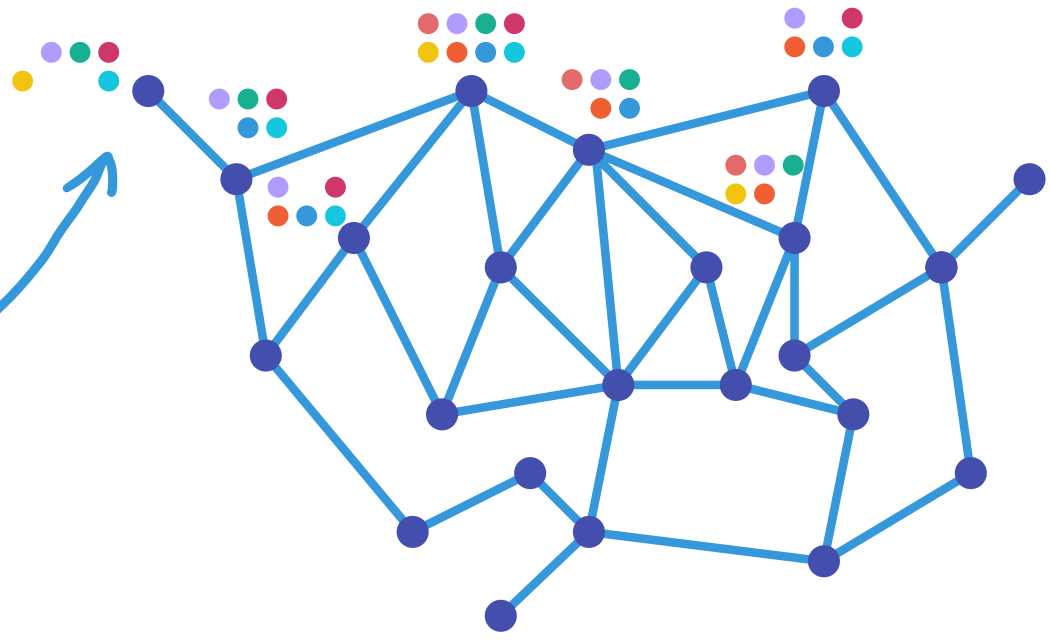
(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

Then the coloring of x and y can be extended to a proper list coloring of G.

proof by induction on n.

let n>3.

Case 1: B has a leaf



let $n > 3$.

Case 2: B is a cycle

Case 2.a: B has a chord

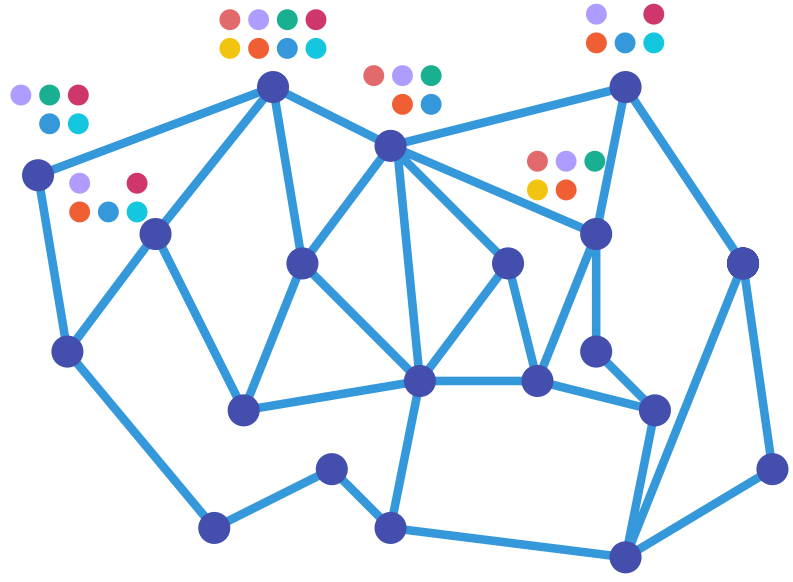
Lemma 2.7. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

(b) $|C(v)| \geq 3$ for all $v \in V_B$.

(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

Then the coloring of x and y can be extended to a proper list coloring of G.



let $n > 3$.

Case 2: B is a cycle

Case 2.b: B has no chord

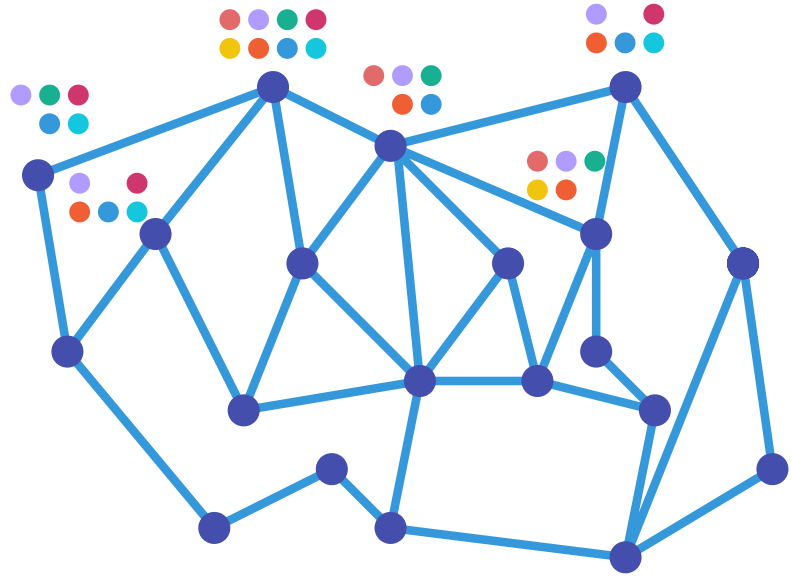
Lemma 2.7. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

(b) $|C(v)| \geq 3$ for all $v \in V_B$.

(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

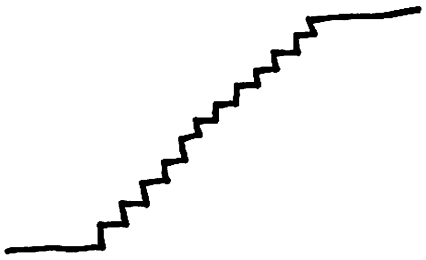
Then the coloring of x and y can be extended to a proper list coloring of G.



Chapter 2

Induction and recursion

2.1 One step at a time



Imagine a long staircase with, say, 1000 stairs. Someone asks you, how do I get from the bottom to the top? And you might respond:

> *Let's take it one step at a time. Say you're at a particular step. Now put one foot on the next step, and use that leg to push your body up and put the other foot on that step, so now your entire mass is on the next step. You can use either foot for the first step. Now if you keep repeating this for each step you will eventually get to the top.*

The key part is the beginning: *Let's take it one step at a time. Say you're at a particular step.* Immediately you recognize that each step is identical, and it suffices to give instructions for a generic step. It would be excessive to say, "For the first step, do bla bla bla. Then for the second step, do bla bla bla. For the third step, do bla bla bla, ..." and so forth, giving detailed instructions for every step.

Let us try to *prove by induction* that the instructions will scale to any staircase. Of course there is something a little awkward about proving something like this that's not as formal as a purely mathematical claim. This is meant as an introductory example to develop intuition.

Claim: *For any $n \in \mathbb{N}$, the instructions above will scale a staircase of n steps.*

In this proof we take for granted that the instruction of using one foot at a time works to go up one step. The only thing to prove is that repeating this instruction will extend to long staircases, *no matter how long.*

The subtle challenge is that our argument needs to be independent of n. We don't want to write a separate proof for every value of $n \in \mathbb{N}$, or proofs that get longer and longer for larger values of n.

The motivation behind the principle of induction is as follows. Let n be large. Directly verifying that the instructions work for each of n steps, one by one, is tedious and boring. However, if we assumed the claim holds for $n-1$ steps, then the extension to n steps is much easier. *Induction* allows us to assume that the claim holds for $n-1$, for any generic n.

The only exception is $n = 1$, since we have no "$n-1$" to build on. So this is a *base case* that we prove from scratch. Although we do not leverage an induction hypothesis, $n = 1$ is obviously very simple. It is typically the case that base cases are simpler.

Base case: suppose $n = 1$. If you follow the instructions once, then you climb one stair. Since the staircase has one stair, this one step completes the staircase.

General case: Suppose $n > 1$. Note that we do not know the value of n, although in our minds, we may sense that it shouldn't really matter. We probably sense that it should behave just like $n-1$ steps, with one extra step. Here's how we put that intuition into words:

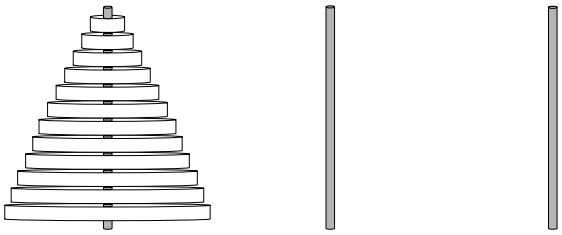
We follow the instructions and take one step onto the next stair. The remaining staircase is a staircase with $n-1$ steps. *By induction on n, the instructions will work for any $(n-1)$-step staircase.* We apply this to the $n-1$ remaining steps, and conclude that the instructions work on any n step staircase as well.

That completes the proof. The principle of induction assures that by addressing the base cases, and explaining how to transition from one proven case to the next unproven case, we automatically address all cases. This might seem obvious for staircases but the point is that the principle applies generally to much more complicated and abstract situations.

2.2 Towers of Hanoi

So much for stairs. Hopefully it is a simple enough example that the point is clear, but my only concern is that it is *too* simple. Instead of understanding staircases by induction / recursion, one can also easily "see through" the induction and conceptualize

the instructions iteratively as a loop, so to speak. So here is a famous puzzle that is too complicated to easily understand and solve without recursion and induction.



In the legend of the towers of Hanoi, there are three posts, which we call post A, post B, and post C. On post A, there is a stack of n rings of increasing radii from top to bottom. (In the story, $n = 64$.) We are allowed to move one ring at a time from the top of one post to the top of another post. Our only constraint is that each post must remain in order: a ring cannot be placed on top of another ring with smaller radius.

The challenge is as follows: is it possible to move all the rings from post A to post B, subject to these rules? We encourage the reader to pause and try to solve this puzzle.

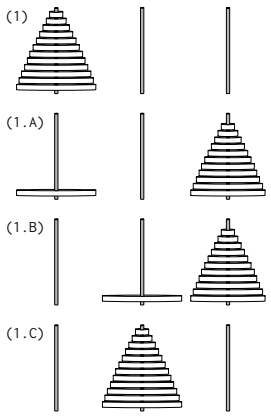
Algorithm. We present a *recursive* algorithm to solve the Towers of Hanoi problem. We first define a recursive spec for our algorithm:

> `Towers-of-Hanoi(`n`,`A`,`B`,`C`)`: Assuming the n smallest rings are on post A, moves the top n rings from post A to post B.

We now implement `Towers-of-Hanoi` (abbr. `ToH`). In the base case, $n = 0$, and there are no rings to move. So `ToH(`0`,`A`,`B`,`C`)` does not do anything.

```
Towers-of-Hanoi(n, A, B, C)
/* Assuming the n smallest rings are on post A, move the top n rings from post A to post B. */
1. If n > 0:
   A. Towers-of-Hanoi(n − 1, A, C, B).
   B. Move the top ring from A to B.
   C. Towers-of-Hanoi(n − 1, C, B, A).
```

Figure 2.1: Recursively solving the towers of Hanoi problem.



Let $n > 0$. When implementing the algorithm recursively, we assume `ToH` fulfills its specification for any smaller value of n.

Now, to move the top n rings from A to B, we will need to move the nth largest ring from A to B. To do that we need to move the top $n-1$ rings out of the way, from A to C. How do we do that? Easy: `ToH`$(n-1, A, C, B)$!

We call `ToH`$(n-1, A, C, B)$ to move the top $n-1$ rings from post A to post C. Then we move the top ring from A to B. Then we move the $n-1$ rings from post C to post B by calling `ToH`$(n-1, C, B, A)$.

That's the entire algorithm. The general case is essentially three lines. See fig. 2.1 above for pseudocode.

Proof of correctness. The proof of correctness is essentially built into the recursive specification. More precisely, the recursive specification is *exactly* the inductive hypothesis. We prove `Towers-of-Hanoi` fulfills the recursive spec by induction on n.

In the base case, $n = 0$, is immediate, as there are no rings to move, and the algorithm does nothing.

Let $n > 0$. When we first call `ToH`$(n-1, A, C, B)$, the $n-1$ smallest rings are on post A, as required. By induction on n, `ToH`$(n-1, A, C, B)$ moves the top $n-1$ rings from A to C. Then we move the top ring from A to B, putting the nth smallest ring on B. Before we call `ToH`$(n-1, C, B, A)$, the $n-1$ smallest rings are on post C, as required. By induction on n, `ToH`$(n-1, C, B, A)$ moves the $n-1$ smallest rings from C to B, where they sit on top of the nth smallest ring. At termination, the n smallest rings are now on post B, as desired. This completes the proof.

Optimal towers of Hanoi. We now ask for algorithm the solves the towers of Hanoi problem with the minimum number of ring moves. Again we pause the reader

attempt the problem.

This turns out to be not very difficult: we argue that the `Towers-of-Hanoi` algorithm above also makes the minimum number of ring moves. How do we prove this? By *strengthening* the recursive spec.

> `Towers-of-Hanoi(`n`,`A`,`B`,`C`)`: Assuming the smallest n rings are on post A, moves the top n rings from post A to post B *with the minimum number of ring moves.*

Let T_n be the number of ring moves made by `ToH` on n rings. We claim that T_n is optimal for all n, by induction on n.

The base case is $n = 0$. Here $T_0 = 0$ which is clearly optimal.

Suppose $n > 0$, and we are moving n rings from post A to post B. `ToH` makes $T_n = 2T_{n-1} + 1$ ring moves; we want to show that this is optimal.

The nth ring can be moved from A to B only if the top $n-1$ rings are on post C. Thus any algorithm must (at least) move the top $n-1$ rings from A to C, move the nth ring from A to B, and move the $n-1$ rings back from C to A. By induction on n, T_{n-1} equals the minimum number of ring moves to move $n-1$ rings from one post to another. So any algorithm must make at least $2T_{n-1} + 1 = T_n$ ring moves. Thus T_n is the minimum number of ring moves.

2.3 Greatest common divisor

For two integers $x, y > 0$, the *greatest common divisor* of x and y is the largest positive integer that divides both x and y. We ask the reader to try to think of an algorithm to compute the greatest common divisor of x and y.

Suppose $x \geq y$. If q divides x and y, then q also divides $x - y$. Conversely if q divides y and $x - y$, then q also divides $x = y + (x - y)$. Thus the greatest common divisor of x and y is the greatest common divisor of y and $x - y$.

Now how do we compute the greatest common divisor of y and $x - y$? Recurse!

As always, before implementing the recursive algorithm we define the recursive spec. Here we add the condition that $x \geq y$ for the sake of convenience. (It allows the code to be a little more compact.)

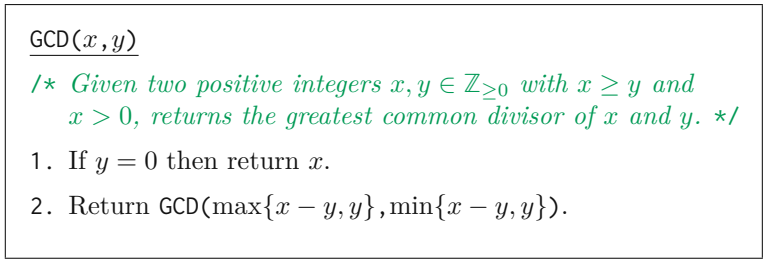


Figure 2.2: Euclid's algorithm for computing the greatest common divisor.

`GCD(`x`,`y`)`: Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

Lemma 2.1. *Given two nonnegative integers $x, y \geq 0$ with $x \geq y$ and $x > 0$, `GCD(`x`,`y`)` of fig. 2.2 computes the greatest common divisor of x and y.*

Proof. We prove the claim by induction on $x + y \in \mathbb{N}$.

In the base case $x + y = 1$. Then we must have $x = 1$ and $y = 0$, and the greatest common divisor is 1. This matches the output of `GCD(1,0)`.

Suppose $x + y > 1$. If If $y = 0$ then x is the greatest common divisor of x and y, matching the output of `GCD(`x`,`y`)`.

Now suppose $y > 0$. Let $x' = \max\{x - y, y\}$ and $y' = \min\{x - y, y\}$. The algorithm returns `GCD(`x'`,`y'`)`. As argued above, the greatest common divisor of x and y is the greatest common divisor of x' and y'. Since $x' + y' < x + y$, by induction, `GCD(`x'`,`y'`)` returns the greatest common divisor of x' and y', which is also the greatest common divisor of x and y, as desired. □

Running time analysis. What is the running time of `GCD(`x`,`y`)`? Each recursive call decreases $x + y$ by at least 1, and the minimum value of $x + y$ is 1. So the running time is $O(x + y)$ (treating subtracting as a constant time operation).

Is $O(x + y)$ a good running time? It looks like a linear running time. However, it takes $O(\log n)$ bits to express x and $O(\log y)$ bits to express y, so the input size is really $O(\log(x) + \log(y)) = O(\log(x + y))$. Thus $O(x + y)$ is in fact *exponential* in the bit complexity of x and y.

A faster algorithm. To speed up `GCD(`x`,`y`)`, let us consider a bad case that realizes the $O(x + y)$ running time. For example, suppose $y = 2$, and x is a very large odd number. The algorithm repeatedly subtracts 2 until x becomes 1, taking $O(x/2)$ iterations in all.

In that example, $1 = x \mod y$. For general $x \geq y > 0$, the algorithm repeatedly subtracts y from x we reach $x \mod y$.

Observe that if $r = x \mod y$, then q divides both x and y if and only if (abbr. iff) q divides r and y. This follows from the fact that $x = ky + r$ for some integer k such that $ky \leq y$. Thus r and y have the same greatest common divisor as x and y. This gives the following faster implementation for `GCD(`x`,`y`)` (as specified above):

`GCD(`x`,`y`)`

`/*` *Given two positive integers* $x, y \in \mathbb{Z}_{\geq 0}$ *with* $x \geq y$ *and* $x > 0$*, returns the greatest common divisor of* x *and* y*.* `*/`

1. If $y = 0$ then return x.
2. Return `GCD(`y`,`$x \mod y$`)`.

Figure 2.3: A faster algorithm for computing the greatest common divisor.

Lemma 2.2. *Given two nonnegative integers* $x, y \geq 0$ *with* $x \geq y$ *and* $x > 0$, `GCD(`x`,`y`)` *of fig. 2.3 computes the greatest common divisor of* x *and* y*.*

Proof. We prove the claim by induction on $y \in \mathbb{N}$.

If If $y = 0$ then x is the greatest common divisor of x and y, matching the output of `GCD(`x`,`y`)`.

In the general case $y > 0$. Let $r = x \mod y$. The algorithm returns `GCD(`y`,`r`)`. As argued above, the greatest common divisor of x and y is the greatest common divisor of y and y. We also have $r < y$. By induction on y, `GCD(`y`,`r`)` returns the greatest common divisor of y and r, hence the greatest common divisor of x and y. □

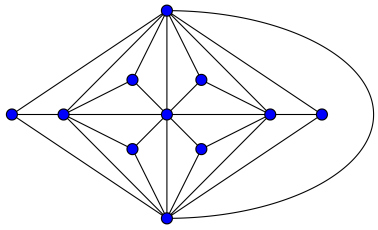
Now for the running time analysis. If $y = 0$ then of course the running time is constant. Let $x \geq y > 0$, and let $r = x \mod y$. If $r \leq y/2$, then this iteration divides y in half. If $r \geq y/2$, then in the next iteration, $y \mod r \leq y/2$. Thus every two iterations divides the y parameter by a factor of 2. Thus the algorithm runs in $O(\log y)$ iterations.

If we treat "$x \mod y$" as a constant time operation, then the algorithm takes $O(\log y)$ time. Otherwise, it $x \mod y$ can be computed in $O(\log(x)\log(y))$ time by long division. This gives a $O\Big(\log(x)\log^2(y)\Big)$ running time overall. Either way, we now have a running time that is polynomial in the input, rather than exponential.

2.4 Induction on planar graphs

Recall that an *undirected graph* $G = (V, E)$ is a discrete structure consisting of a set of elements V, called *vertices*, and a collection of unordered pairs of vertices E, called *edges*. (A *directed graph* is similar except the edges are *ordered* pairs of vertices). The two vertices u, v of an edge $e = \{u, v\} \in E$ are called the *endpoints* of e. The endpoints of e are allowed to be equal; then e is called a *self-loop* or a *loop*. Graphs are typically visualized by drawing each vertex as a dot and each edge as a line connecting the dots.

A *planar graph* is a graph that can be drawn in the plane so that the edges intersect only at their endpoints. Such a drawing is called an *embedding* of the graph in the plane. The graph on the right is an example of a planar graph.



Not all graphs are planar. For example, the complete graph on 5 vertices (with all $\binom{5}{2}$ distinct edges) is not a planar graph.

Let G be a planar graph and fix an embedding of G in the plane. The vertices and edges of a planar graph divide the plane into regions, called *faces*. The unbounded region is called the *outer face*; the vertices and edges bounding the outer face is called the *outer boundary* of G.

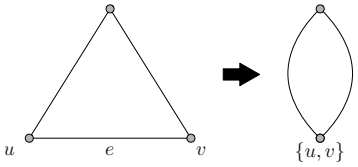
Euler's formula. Planar graphs have several beautiful properties that do not hold for graphs in general. A striking one is *Euler's formula*.

Theorem 2.3 (Euler's formula)**.** *For any connected planar graph G with m edges, n vertices, and p faces,*

$$n - m + p = 2.$$

Proof. We prove the claim by induction on the number of vertices, n. In the base case, $n = 1$. Then every edge is a self-loop, enscribing an interior face, so $p = m + 1$. Then $n - m + p = n + 1 = 2$, as desired.

Suppose $n > 1$. Since the graph is connected, G has at least one edge $e = \{u, v\}$. Let H be the graph obtained by contracting e to a single vertex. This has the effect of decreasing the number of edges and the number of vertices by 1. The number of faces remains the same. By induction on the number of vertices, Euler's formula holds for H, which gives



$$(n - 1) - (m - 1) + p = 2.$$

The LHS simplifies to $n - m + p$, as desired. □

Corollary 2.4. *For any simple planar graph (with no self-loops or parallel edges), with m edges and $n \geq 3$ vertices, $m \leq 3n - 6$.*

Proof. Let p be the number of faces. Every face is bounded by at least three edges, and each edge is on the boundary of at most 2 faces. Thus $3p \leq 2m$. Plugging this into Euler's formula gives

$$n - m + (2/3)m \geq 2.$$

Rearranging, we have $m \leq 3n - 6$ □

List-coloring planar graphs with 6 colors. Fix a graph $G = (V, E)$. For $k \in \mathbb{N}$, a *k-coloring* is an assignment $\pi : V \to [k]$ such that for any edge $e = \{u, v\} \in E$, $C(u) \neq C(v)$.[1] We can think of $[k]$ as representing k different colors, and $\pi : V \to [k]$ as "coloring" each vertex one of the k colors. Then π is a feasible k-coloring if there is no monochromatic edge; i.e., the endpoints of every edge has different colors.

A fundamental problem is as follows: given a graph $G = (V, E)$ and $k \in \mathbb{N}$, is there a k-coloring of G?

The famous *four color theorem* states that any planar graph can be colored with 4 colors. The theorem is notorious for its long history of failed proofs. The theorem was finally proven by [AH77], but with a catch: Appel and Haken needed the help of a computer to enumerate a large number of special cases. Consequently the proof was largely incomprehensible. It remains open to obtain a proof without the help of a computer.

We will study a related problem, called *list coloring*. Here each vertex v is associated with a set of colors $C(v) \subset \mathbb{N}$. The goal is to color the vertices $\pi : V \to \mathbb{N}$ so that (a) $\pi(v) \in C(v)$ for all v, and (b) there is no monochromatic edge. A graph $G = (V, E)$ is said to be *k-list colorable* if for any family of lists $\{C(v) : v \in V\}$ such that $|C(v)| \geq k$ for all v, there exists a feasible list-coloring of G.

Theorem 2.5. *Every planar graph is 6-list colorable.*

Proof. We may assume the graph is simple, since self-loops and parallel edges have no effect on coloring.

In any planar graph with at least 3 vertices, by Euler's formula, the average degree is bounded above by

$$\frac{2m}{n} \leq \frac{6n - 12}{n} < 6.$$

[1] Here $[k] \stackrel{\text{def}}{=} \{1, \ldots, k\}$.

Since the average degree is less than 6, the minimum degree is less than 6; that is, at most 5.

Now, consider any planar graph G. If G has less than 3 vertices than certainly it can be colored with 6 colors. Suppose G has at least 3 vertices. Then there is a vertex v with degree ≤ 5. Let H be the graph obtained by removing v and all incident vertices. H has one less vertex than G. By induction on the number of vertices, there is a 6-coloring of H. At most 5 of the 6 colors are taken by neighbors of v. Thus the 6-coloring on H can be extended to a 6-coloring on G by assigning v an available color. □

One can covert the proof into a linear time algorithm; see exercise 2.5.

List-coloring planar graphs with 5 colors. Now we prove a stronger result: that every planar graph is *5-list colorable*. The following theorem and proof is due to [Tho94].

Theorem 2.6. *All planar graphs are 5-list colorable.*

The proof is elegant example of identifying a *stronger* induction hypothesis that is easier to prove. Theorem 2.6 follows from the following lemma by arbitrarily coloring any two consecutive vertices along the outer boundary.

Lemma 2.7. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

(b) $|C(v)| \geq 3$ for all $v \in V_B$.

(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

Then the coloring of x and y can be extended to a proper list coloring of G.

Proof. We prove the claim by induction on $|V|$.

In the base case, $|V| = 3$. The remaining uncolored vertex v has $|C(v)| \geq 3$, so there is a color available for v.

Consider the case $|V| > 3$. We assume in induction on $|V|$ that the claim holds for $|V| - 1$ (or fewer) vertices. Let B be the boundary of G. We have a few cases.

Case 1. Suppose G has a leaf v along its outer boundary. Let H be the graph obtained by removing v. H has one fewer vertex, and the outer boundary of H is the outer boundary of G except without v.

Suppose first that v is not either x or y. H now meets the hypothesis of the claim with one fewer vertex. By induction on the number of vertices, there is a proper list-coloring of H extending the coloring on x and y. Since $|C(v)| \geq 3$, and v has only one neighbor, there is an available color to extend the coloring from v to G.

Now suppose $v = x$ or $v = y$. Without loss of generality we assume $v = x$. Choose another vertex x' adjacent to y, and assign it an available coloring α'. H, x', and y now meet the hypothesis of the claim with one fewer vertex. By induction, there is a proper list coloring of H extending the colors assigned to x' and y. Again, since $|C(v)| \geq 3$, and v has only one neighbor, there is an available color to extend the coloring from v to G.

Case 2. Suppose G has no leaf along its outer boundary B. Then B is a cycle. Here we have two sub-cases.

Case 2.a. Suppose B has a chord.[2] Split B into two sides B_1 and B_2 meeting at u and v. Let G_1 be the planar graph bounded by $B_1 \cup \{u, v\}$ and let G_2 be the planar graph bounded by $B_2 \cup \{u, v\}$. G_1 has fewer vertices than G, so by induction, there is a proper list coloring of G_1.

Now consider G_2, and assign u and v the two colors given by the list coloring G_1. By induction, the colors on u and v can be extended to a proper list-coloring of G_2.

Finally, combining the list-colorings of G_1 and G_2 give a list-coloring on G.

Case 2.b. Now suppose B does not have any chords. Let v be the vertex on the other side of x from y. Since $|C(v)| \geq 3$, $C(v)$ contains at least two additional colors γ, δ distinct from α.

The neighbors of v consist of x and another neighbor u on B, and a number of neighbors $t_1, \ldots, t_k$ in the interior of B. Let H be the graph obtained by removing v and all incident edges from the graph. Observe that $t_1, \ldots, t_k$ are all exterior vertices of H. Consider the list-coloring problem on H where we remove γ and δ from $C(t_i)$ for $i = 1, \ldots, k$ Then all vertices on the boundary of H have colors lists of size 3 or greater, and all interior vertices have color lists of size 5 or greater. By induction on the number vertices, we can extend the coloring on x and y to a proper list coloring on H.

To then color v, observe that none of the vertices $t_1, \ldots, t_k$ are assigned γ or δ. The only other neighbors of v are u and x, where x is already assigned α. So either γ or δ (or both) is available to color v, which extends the list coloring to all of G. □

[2] Given a cycle C, a chord is an edge between two non-adjacent vertices of the cycle.

2.5 Do I really need induction and recursion?

This style of thinking is abstract and naturally met with resistance. Why adopt new styles of thinking when you have already come so far?

Some of our problems are simple enough that one can sort of "see the whole thing" without taking a top-down perspective. However the solutions that come out of this, even when correct, tend to be harder to understand and verify because the thinking process is not very structured.

A bigger issue is that at some point, problems are sufficiently complicated that it's harder to see everything all at once. It is helpful to break down the problem into smaller steps, having faith that repeating these steps eventually solves the entire problem. Moreover it is much easier to analyze and prove the correctness of a single step satisfying declared invariants, and then rely on induction to automatically scale the argument to the entire algorithm, then it is to analyze the whole algorithm all at once.

This is what induction really does for us. It is a sound principle for breaking down big problems into locally verifiable subproblems. It allows us to not only see solutions to harder problems, but also feel more confident that they are correct.

2.6 Additional notes and materials

See [Che15], [Eri19, Chapter 0], or [GKP94, Chapter 0] for more background on induction.

There are many different ways to prove Euler's formula; see [Epp]. Thomassen's proof [Tho94] is also described in [AZ18].

2.7 Exercises

Exercise 2.1. For each of the following variations of the Tower of Hanoi problem, give an algorithm that solves the problem with the minimum number of ring moves. Recursive algorithms should have an explicit recursive spec. We will treat the recursive spec as an induction hypothesis to justify the correctness of the algorithm, and no explicit proof is otherwise required.[3] Non-recursive algorithms require a proof (which should be short).

You may assume as fact that the `Towers-of-Hanoi` algorithm (fig. 2.1, on page 35) moves n rings from one post to another in the minimum number of steps.

[3]You may want to work out a full proof for yourself anyway for extra practice. Additional justification or comments might help the grader's understanding in the event of partial credit.

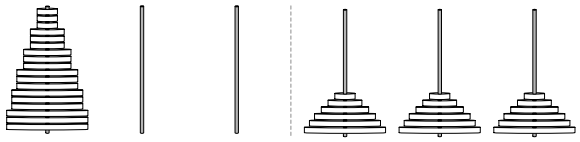
1. In the *easy towers of Hanoi* problem, we remove the restriction that the rings always have to be in sorted order on each post. The goal is still to move n sorted rings from post A to post B, in sorted order on post B.

2. In the *cyclic towers of Hanoi* problem, a ring can only be moved in "cyclic" order from post A to post B, from post B to post C, and from post C to post A. The goal is to move n sorted rings from post A to post B.

3. The *double-cyclic towers of Hanoi* is like the cyclic towers of Hanoi problem except now the goal is to the n sorted rings from post A to post C.

4. In the *thick towers of Hanoi* problem, we have $3n$ rings of n distinct sizes with three copies of each ring. (Rings of the same size can stack on top of each other.) The goal is to move all $3n$ rings from post A to post B.

5. In the *triple towers of Hanoi* problem, we have $3n$ rings of n distinct sizes with three copies of each ring. The goal is to distribute the rings so that each post has n rings, one of each size, in sorted order.



6. In the *American towers of Hanoi* problem, we have n rings with each ring colored red, white or blue. The three posts are also colored red, white, and blue. Initially all the rings are stacked (in order) on the red post. The goal is to stack all the red rings on the red post, the blue rings on the blue post, and the white rings on the white post, in order.

7. In the *messy towers of Hanoi* problem, the n rings are initially distributed arbitrarily over the three posts. (Each post is still in order.) The goal is to move all the rings to a single designated post.[4]

8. (Extra credit) Invent a fun and interesting variant of the towers of Hanoi problem.

[4]You are allowed to look at the current configuration and identify which post has the nth smallest ring, for any given n. You may have different cases depending on which post has the nth smallest ring.

Exercise 2.2. For each of the recursive specifications below, design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[5]

1. `subset-sum`$(x_1, \dots, x_n,\ T)$: Given n integers $x_1, \dots, x_n \in \mathbb{Z}$, and an additional integer T, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

2. `min-vertex-cover`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of G.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

3. `partition`$(x_1, \dots, x_n)$: Given n integers $x_1, \dots, x_n \in \mathbb{Z}$, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = \sum_{i \in [n] \setminus S} x_i$, and false otherwise.

4. `max-matching`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the maximum size of any matching.

   (A matching is a set of edges $M \subseteq E$ with disjoint endpoints.)

5. `max-independent-set`$(G = (V, E)$: Given an undirected graph $G = (V, E)$, return the maximum size of any independent set of vertices.

   (Given an undirected graph $G = (V, E)$, an *independent set* is a set of vertices $S \subseteq V$ such that no two vertices $u, v \in S$ are connected by an edge.)

6. `longest-increasing-subsequence-including-first`$(x_1, \dots, x_n)$: Given a sequence of numbers $x_1, \dots, x_n$, return the length of the longest (strictly) increasing subsequence of $x_1, \dots, x_n$ over all subsequences that include x_1. (You may assume $n \geq 1$.)

   (A subsequence of $x_1, \dots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \dots, x_{i_k}$, where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. A sequence of numbers $y_1, \dots, y_\ell$ is strictly increasing if $y_i < y_{i+1}$ for $i = 1, \dots, \ell - 1$.)

[5] Your algorithms will not be very fast, but their correctness should follow from an induction argument using the recursive spec as the induction hypothesis.

You are welcome to describe your algorithm using high level steps as long as they are clearly easy to implement and would take linear time; e.g., "let G_1 be the graph obtained by removing such and such vertices and such and such edges."

We recognize that some of these problems have well-known efficient algorithms. That is besides the point; the goal here is to rapidly get some practice thinking recursively.

7. `reachable`$(G = (V, E), s, t)$: Given a directed graph G and two vertices $s, t \in V$, return `true` if s can reach t in G, and `false` otherwise.

   (s can reach t if either $s = t$ or there is a sequence of (directed) edges of the form $(s, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, t)$. Note that the endpoint of one edge is the initial point of the next edge. This is also called a *(directed) walk* in G from s to t.)

Exercise 2.3. Recall that a tree is an undirected graph with no cycles.

1. Prove by induction that every tree has list-coloring number at most 2.
2. Design and analyze am algorithm that, given a tree $T = (V, E)$, and color lists $C(v)$ of size ≥ 2 for each vertex v, computes a proper list coloring of T in linear time.

Exercise 2.4. Let $T = (V, E)$ by a rooted binary tree. Recall that a *full* binary tree is a binary tree where every non-leaf node has exactly. The goal is to compute the maximum size of any full binary tree that is a subgraph of T.

Exercise 2.5. Design and analyze a linear time algorithm for six-coloring a planar graph.