6/29 Lecture Agenda

- Announcements
- Part 2-3: Randomized Algorithms and QuickSort
- 10 minute break!
- Part 2-4: Karger's Algorithm

Announcements!

- HW1 due Thursday, 11:59 PM
  - See Problem 6 (Coding) for a bit more detail. Also, no late days will be assessed on Problem 6 specifically (still a limit of 2 though)
- Quick feedback request coming soon
- HW2 out tonight
- Deadline to add the course is July 1, 5 PM.
  - If you added this week, please reach out to me.
- Remember: **no class on July 4!** (due times for Pre-HW2 and HW2 are longer to reflect the vacation)

Announcements!

- HW1 due Thursday, 11:59 PM
  - See Problem 6 (Coding) for a bit more detail. Also, no late days will be assessed on Problem 6 specifically (still a limit of 2 though)

- Quick feedback request coming soon

- HW2 out tonight

- Deadline to add the course is July 1, 5 PM.
  - If you added this week, please reach out to me.

- Remember: **no class on July 4!** (due times for Pre-HW2 and HW2 are longer to reflect the vacation)

is July 4! = July 24?

6/29 Lecture Agenda

- Announcements
- Part 2-3: Randomized Algorithms and QuickSort
- 10 minute break!
- Part 2-4: Karger's Algorithm

WORLD 2-3

Randomized Algorithms and QuickSort

Divide and Conquer
Sorting & Randomization
Data Structures
Graph Search
Dynamic Programming
Greed & Flow

Special Topics

LOL I'm so random

- A deterministic algorithm behaves the same way every time it is run on the same input.
- A randomized algorithm has access to – and uses – some source of randomness. It may produce different results each time it is run, even on the same input.

What is a "source of randomness"?

- Computers are good at doing exactly what you tell them! But how do you tell them to create *randomness*?
- There are deterministic algorithms to create "pseudorandom" sequences of bits that *appear* to be random.
- These still have to be *seeded* (kind of like a base case!) The seed can come from the current time (bad idea), or from, say, temperature fluctuation in the CPU... should be something intractable to reverse-engineer.
    - *Ian's apocryphal? Keno story*

The random patterns of lava lamps have been used as one such source!

Four "random" patterns ought to be enough, right?

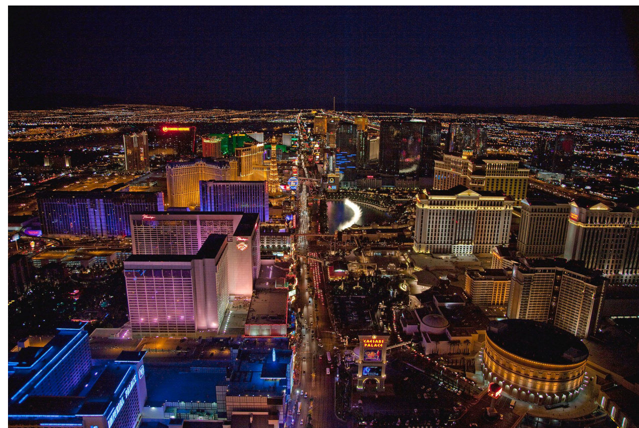
This guy won $100K in 1984 on the show Press Your Luck by memorizing the "random" patterns

Two ways to use randomness

- A **Monte Carlo** algorithm is always fast, but may be wrong some of the time.
  - *side note: non-apocryphal roulette story*

- A **Las Vegas** algorithm is always correct, but may be slow some of the time.

  It's hard to remember which of these is which. Don't worry about memorizing the names.



What good is an algorithm that can be wrong?

- Say we have a randomized Monte Carlo algorithm to solve a problem with a yes-or-no answer, and it is correct 51% of the time.

- But that's not good enough! We want to be **99%** sure that the answer is correct!

- Can we still get what we want?

Solution: Run the algorithm a lot!

- Run it k times, record whether each result is "Yes" or "No", and take the majority answer.

- The probability that we get more of the wrong answer than the right answer can be calculated with a normal approximation to the binomial distribution (out of scope for this class).
  - see https://web.stanford.edu/class/cs109a/109asp22notes6.pdf if curious

- It turns out that if we run this 13527 times, we have less than a 1% probability of getting the wrong overall answer.
  - If we want more confidence, we can run more times.

Who wants to run an algorithm 13527 times?

- Computers are fast!
    - I run some brute-force combinatorics stuff in O(vernight) time

- Some problems may lack obvious deterministic solutions, but have easy randomized ones.

Who wants to run an algorithm 13527 times?

- Computers are fast!
  - I run some brute-force combinatorics stuff in O(vernight) time

- Some problems may lack obvious deterministic solutions, but have easy randomized ones.

- Real-world example: Primality testing!

  - Some forms of cryptography depend on products of two huge primes being very hard to factor.

  - But to use this, we need a source of huge primes.

  - A randomized algorithm makes this easy! Coming on HW2...

Why not just derandomize randomized algorithms?

- Consider some randomized algorithm. It is fed some list of random bits.

- What if we always give it the **same** fixed list of bits, making it deterministic? Won't it still work?
  - Yes, but there might be some inputs where it always fails because the fixed bit list just happens to be unlucky for those.

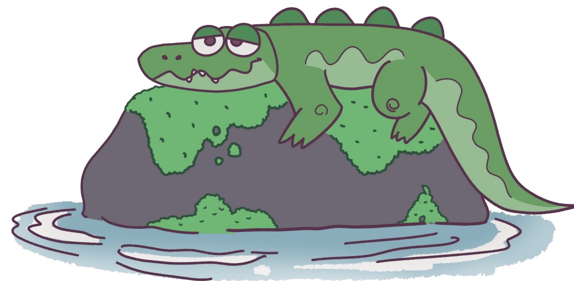
- There seem to be some problems that *require* randomness to be solved (see "Ramsey theory")

Let's see an example!

- On Monday we used the idea of *partitioning*:
  - We have some (unsorted) list.
  - We find some value in the list to use as a "pivot". In the context of k-Select, this was an estimate of the median.
  - We go through the list moving values smaller than the pivot to one side, and values larger than the pivot to the other side.

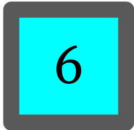
Finding a good pivot is a lot of work!

So what if we don't even try?



Here's a strategy to sort a list.

- Pick an element of the list uniformly at random.
- Use that element as a pivot.
- Then recursively sort the left and right halves in the same way.



Pick a pivot at random

Partition

Two sublists of
about half size!
Then keep going!

Pick a pivot at random

Partition

One empty sublist, one of almost full size! Then keep going!

The worst case is really bad

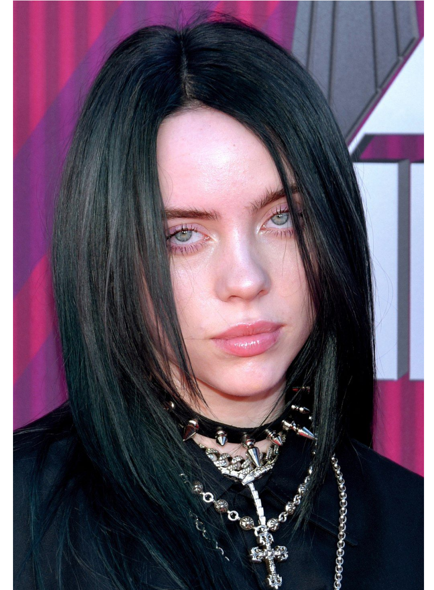
- The first call creates sublists of size $n-1$ and 0.
- The recursive call on the sunlist of size $n-1$ creates sublists of size $n-2$ and 0.
- So our total work is: $n + (n-1) + (n-2) + \ldots + 1$, which we know is $O(n^2)$.

…and this algorithm is ironically called QuickSort!

What does "worst case" mean here?

- In deterministic algorithms, by "worst case" we meant: use the input that makes the algorithm run the slowest.

- In randomized algorithms, by "worst case" we mean the worst of two worlds:
  - the input is as bad as possible, *and*
  - a "bad guy" has seen the input and then chosen our algorithm's randomness for us.



Some good news...

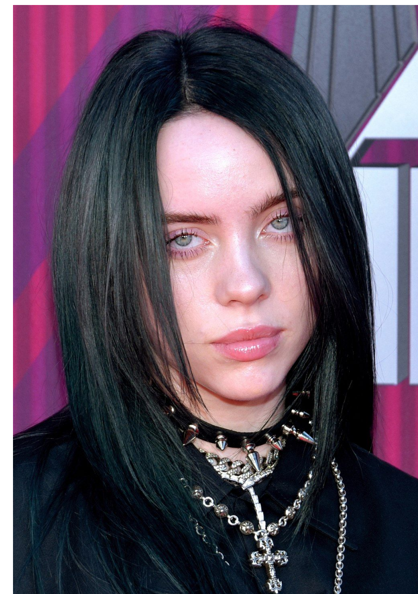
- Many randomized algorithms essentially do not care about how "bad" the input is!
    - All lists are equally good/bad for QuickSort, since only the relative order of the elements matters.
    - What about ties? These only help. When we're partitioning, we can just slurp up any values that exactly match the pivot.

But the "bad guy" can doom us

- **Us:** We'd like to QuickSort [4, 2, 1, 3, 5].
- **Bad Guy:** Cool. OK, your pivot is 1.

(we do the first step to get [1] and [4, 2, 3, 5])

- **Us:** We'd like to QuickSort [4, 2, 3, 5].
- **Bad Guy:** Heh, suckers. OK, your pivot is 2...



What about the average case?

- Here average means over all possible ways the randomized algorithm might make its choices, **not** over all possible inputs.
  - What would that even mean? There are ∞

- We still use a worst-case input. (But for QuickSort, any list of distinct elements is equally bad)

- But now the bad guy has no power! We consider *all* ways that the dice of our algorithm might fall, and take the *average* of those running times.

An incorrect argument

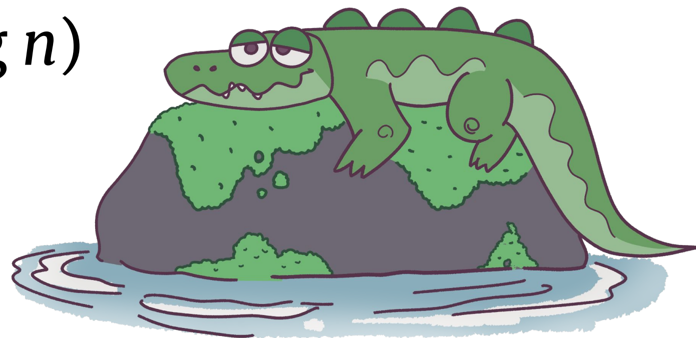
- On average, QuickSort should choose a pivot sort of close to the median of the list, right?

- So the list roughly breaks into two halves of about the same size, right?

it takes a linear pass through the list to partition

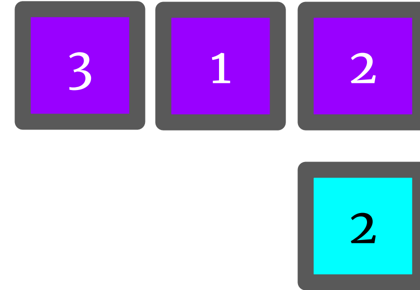
- $T(n) = 2T(n/2) + O(n)$
- so, by the Master Theorem, $O(n \log n)$

Waverly's reasoning is not sound. We'll see why in Problem Session 2. But her answer turns out to be right!



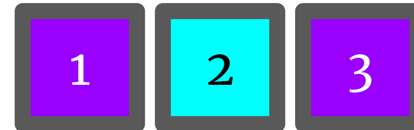
To argue correctly, we'll need some probability

- The running time is determined by **how many pairs of elements we compare.**
- In QuickSort, whenever we compare two elements, one of them is always the current pivot.
- Any pair is compared at most once, and it's possible to never compare some pairs.



Partition:

- compare 3 to 2
- compare 1 to 2

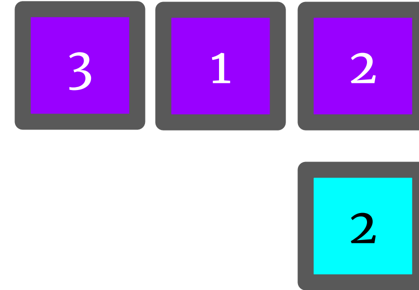


We never compared 1 and 3!

If two non-pivot elements end up on opposite sides of a pivot:

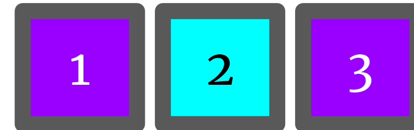
- They **will never be** compared to each other.
  - The recursive QuickSort calls for the two sides are totally separate!

- They **have never been** compared to each other.
  - Otherwise one of them would have been a pivot earlier, and we wouldn't be looking at it now.



Partition:

- compare 3 to 2
- compare 1 to 2



We never compared 1 and 3!

Arguing over the average case

- Consider the j-th and k-th **largest** elements of the list, with $j < k$. (Call them E_j and E_k.) **(They may not be in sorted order in the list.)**
- What is the probability that they get compared?
  - If E_j or E_k is chosen as the pivot in this round, the two surely get compared (since the pivot is compared with everything else).
  - If the pivot is **smaller than or larger than** both, then they might still get compared in the next round.
  - If the pivot is between the two (**larger than E_j, smaller than E_k**) they definitely never get compared.

Arguing over the average case

- Consider the j-th and k-th **largest** elements of the list, with $j < k$. (Call them E_j and E_k.) **(They may not be in sorted order in the list.)**
- What is the probability that they get compared?
  - If E_j or E_k is chosen as the pivot in this round, the two surely get compared (since the pivot is compared with everything else).
  - If the pivot is **smaller than or larger than** both, then they might still get compared in the next round. ***This part is hard to handle!***
  - If the pivot is between the two (**larger than E_j, smaller than E_k**) they definitely never get compared.

Arguing over the average case

- Consider the j-th and k-th **largest** elements of the list, with $j < k$. (Call them E_j and E_k.)
- What is the probability that they get compared?
  - It's the probability that in the sublist of elements $[E_j, E_{j+1}, \ldots, E_{k-1}, E_k]$, **i.e. those that are no less than E_j and no greater than E_k**, either E_j or E_k is chosen as a pivot before any other element in the range is.
  - Since the total size of the range is $k - j + 1$, this probability is $\mathbf{2 / (k - j + 1)}$.

Expectation and indicator random variables

- An indicator random variable is associated with some event, and it is 1 if the event happens and 0 if not.
- Here, let I_{jk} be the indicator for whether elements E_j and E_k ever get compared (at any point in the algorithm).
- The total number of comparisons is the sum over all $j < k$ of I_{jk}.
- We want the expectation (average) of that sum, over all the ways that QuickSort could randomly choose its pivots.
- (See Prereq Review #3 on the site!)

$$\mathbf{E}_{\text{Quicksort's randomness}}[\sum_{j=1}^{n}\sum_{k=j+1}^{n} I_{jk}]$$

This is the number of comparisons.

$$= \sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathbf{E}_{\text{Quicksort's randomness}}[I_{jk}]$$

Linearity of expectation (see Prereq Review #3)

$$= \sum_{j=1}^{n-1}\sum_{k=j+1}^{n} P(j, k\text{-th elements are compared})$$

Expectation of an indicator is the probability of its event (see Prereq Review #3)

$$= \sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \frac{2}{k-j+1}$$

This is the probability we found earlier.

$$\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{2}{k-j+1} = \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l+1}$$

Simplify the term in the summation by defining l = k – j (and changing the bounds accordingly)

$$\leq \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l} \leq \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{2}{l} = 2 \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{1}{l}$$

Use inequalities to make the term in the summation easier to work with

$$\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{2}{k-j+1} = \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l+1}$$

Simplify the term in the summation by defining l = k – j (and changing the bounds accordingly)

$$\leq \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l} \leq \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{2}{l} = 2 \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{1}{l}$$

Use inequalities to make the term in the summation easier to work with

Now we use another math fact (and I do want you to remember this one): the sum 1/1 + 1/2 + … + 1/l is the l-th harmonic number, H_l. And these grow slowly: $H_l = O(\log l)$.

$$\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{2}{k-j+1} = \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l+1}$$

Simplify the term in the summation by defining l = k – j (and changing the bounds accordingly)

$$\leq \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l} \leq \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{2}{l} = 2 \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{1}{l}$$

Use inequalities to make the term in the summation easier to work with

Now we use another math fact (and I do want you to remember this one): the sum 1/1 + 1/2 + ... + 1/l is the l-th harmonic number, H_l. And these grow slowly: $H_l = O(\log l)$.

$$= 2 \sum_{j=1}^{n-1} H_n = 2(n-1)H_n = O(n \log n)$$

$$\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{2}{k-j+1} = \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l+1}$$

$$\leq \sum_{j=1}^{n-1} \sum_{l=1}^{n-j} \frac{2}{l} \leq \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{2}{l} = 2 \sum_{j=1}^{n-1} \sum_{l=1}^{n} \frac{1}{l}$$

That was a lot

- You would not have been expected to come up with all of that on your own!

- But I do want you to understand the details, since many of the "tricks" used there are useful again and again.

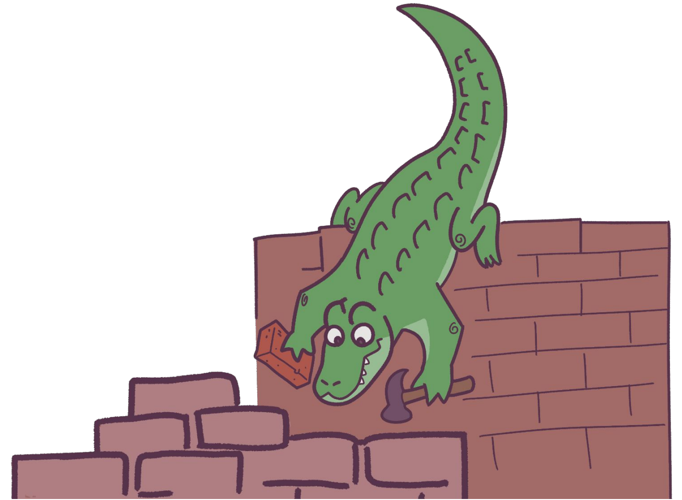
    - Indicator random variables
    - Linearity of expectation
    - Changing variables to simplify an expression
    - Using inequalities to simplify an expression
    - Harmonic numbers

So, wait what were we doing again?

- QuickSort is O(n log n) in the average case!
- (Side note: it's also O(n log n) in the best case. Then the recursion tree does look just like MergeSort.)
- In practice, it's really fast!
- BTW, QuickSort is a "Las Vegas" algorithm: it's always correct, but might be slow.

We never said anything about implementation

- Sorry, Sisi. Today is pretty math/theory-heavy.
- One thing to consider: when we partition a list, we shouldn't actually create a bunch of separate lists in memory. We can cleverly swap stuff around in the existing list. (Coming in Problem Session 2)



What do major languages actually use to sort?

C++: Introsort (a hybrid of Quicksort, Heapsort, and Insertion Sort)

Java: Quicksort (for primitives), a modified MergeSort (for objects)

JavaScript: Implementation-dependent

Python: Timsort (a hybrid of Insertion Sort and MergeSort)

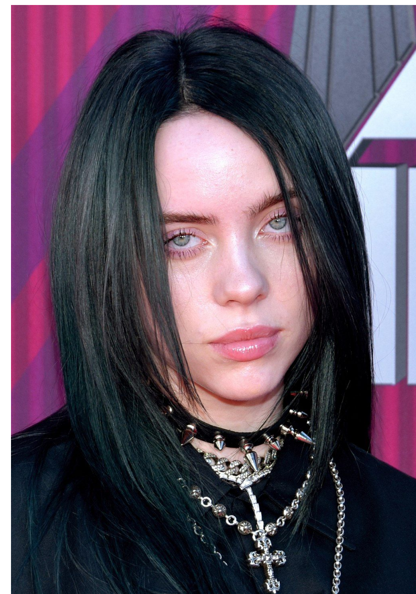
One takeaway from this: maybe there is no universally best sort?

But remember the bad guy

- If a bad guy knows we're using QuickSort and has the ability to choose our randomness... or knows the random generator and can choose our input...

- In, say, a coding contest, this is annoying (my solution got hacked by the right test case!)

- In a real application, this could be serious.



Randomized Algorithms Takeaway

- If you want **speed** and are willing to tolerate a chance we might not be correct, a **Monte Carlo** algorithm may be the way to go.
  - and you can push that failure chance arbitrarily low by running the algorithm more and more times.

- If you want **correctness** and can tolerate occasional slowness, a **Las Vegas** algorithm may be for you!

Randomized Algorithms Takeaway

- If you want **speed** and are willing to tolerate a chance we might not be correct, a **Monte Carlo** algorithm may be the way to go.
  - and you can push that failure chance arbitrarily low by running the algorithm more and more times.

- If you want **correctness** and can tolerate occasional slowness, a **Las Vegas** algorithm may be for you!

- Any randomized algorithm can be made deterministic (say, for gov't work) but at the cost of average-case goodness. (There will probably be some pathological inputs that do really badly)

- We thought outside the narrow box of "always has to be correct and fast", and got more real-world flexibility.

6/29 Lecture Agenda

- Announcements
- Part 2-3: Randomized Algorithms and QuickSort
- 10 minute break!
- Part 2-4: Karger's Algorithm

6/29 Lecture Agenda

- Announcements
- Part 2-3: Randomized Algorithms and QuickSort
- 10 minute break!
- Part 2-4: Karger's Algorithm

WORLD 2-4

Karger's Amazing Algorithm

Divide and Conquer

Sorting & Randomization

Data Structures

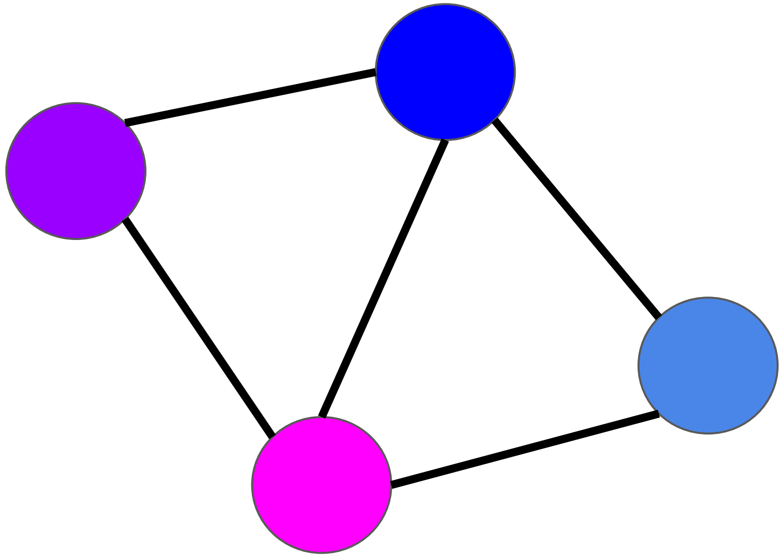
Graph Search

Dynamic Programming

Greed & Flow

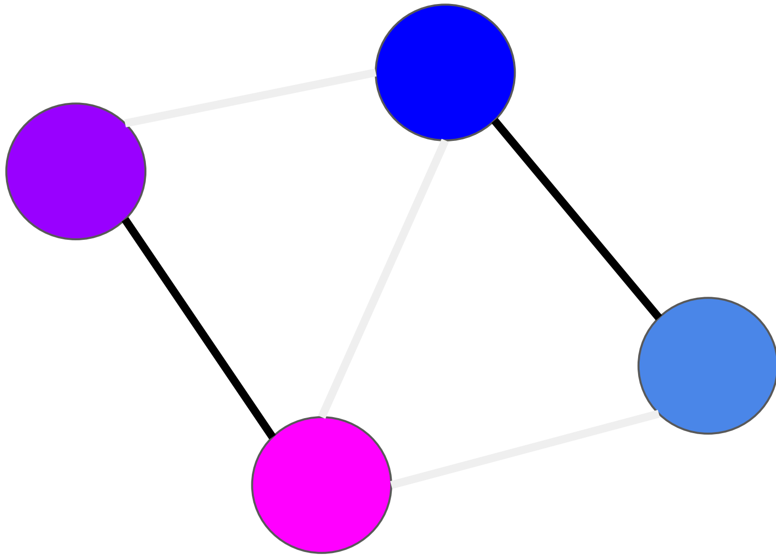
Special Topics

An Early Taste of Graphs



- Graphs consist of **vertices** (sometimes called nodes) connected by **edges**.
- Here, the edges are **undirected** (two-way) and the graph is **connected** (every vertex is reachable from every other)
- We'll talk about how these are actually represented in memory later in the course.

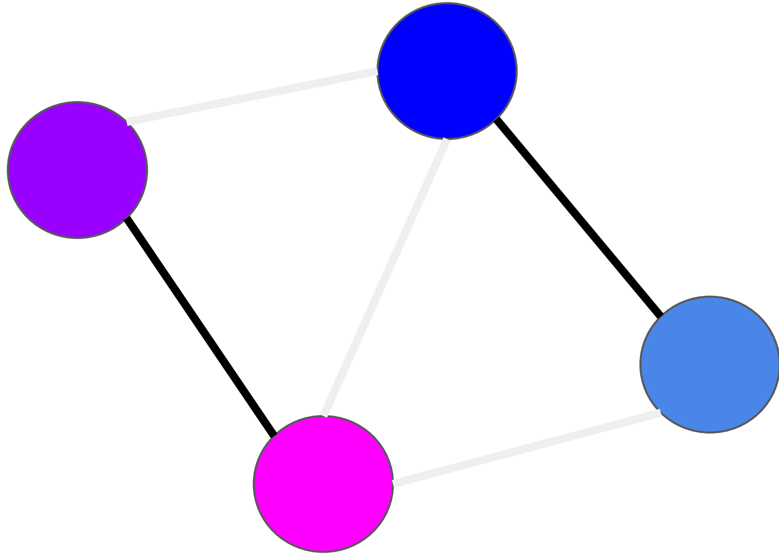
Graph Cuts



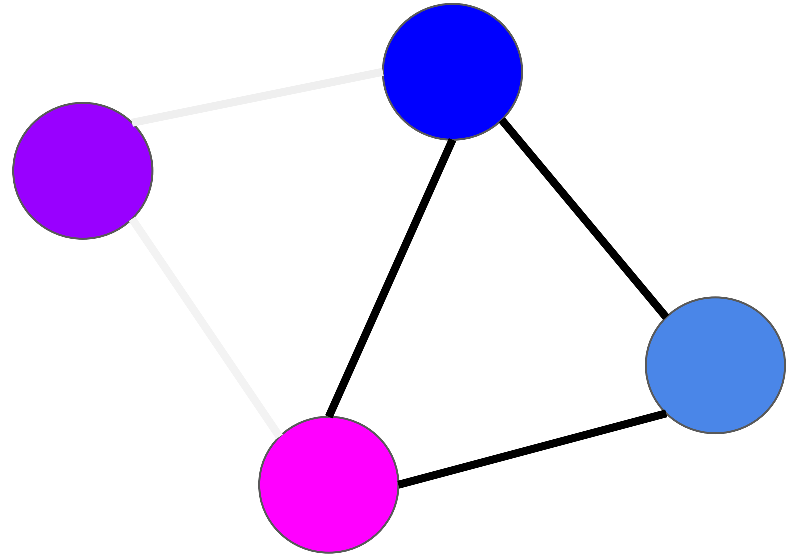
- A **cut** is a set of edges that, when removed, break the graph into two separate **connected components**.

- What does "connected" mean here? It means that within a component, you can get from every vertex to every other. But you can't get from one component to another!

Finding *a* Minimum Cut

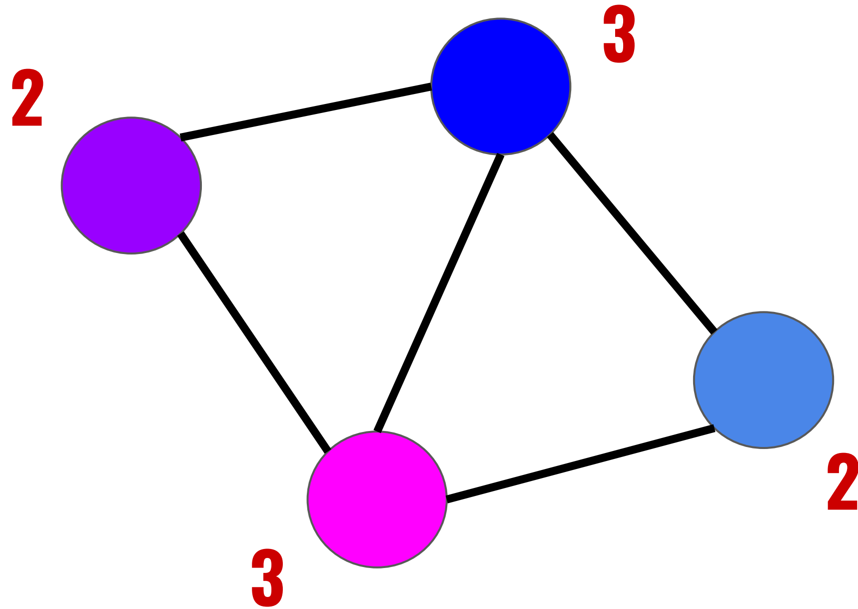


Here we had to remove three edges to disconnect the graph.



But we could do it with only two! Also notice that this isn't the only min cut of size 2.

A Useful Observation

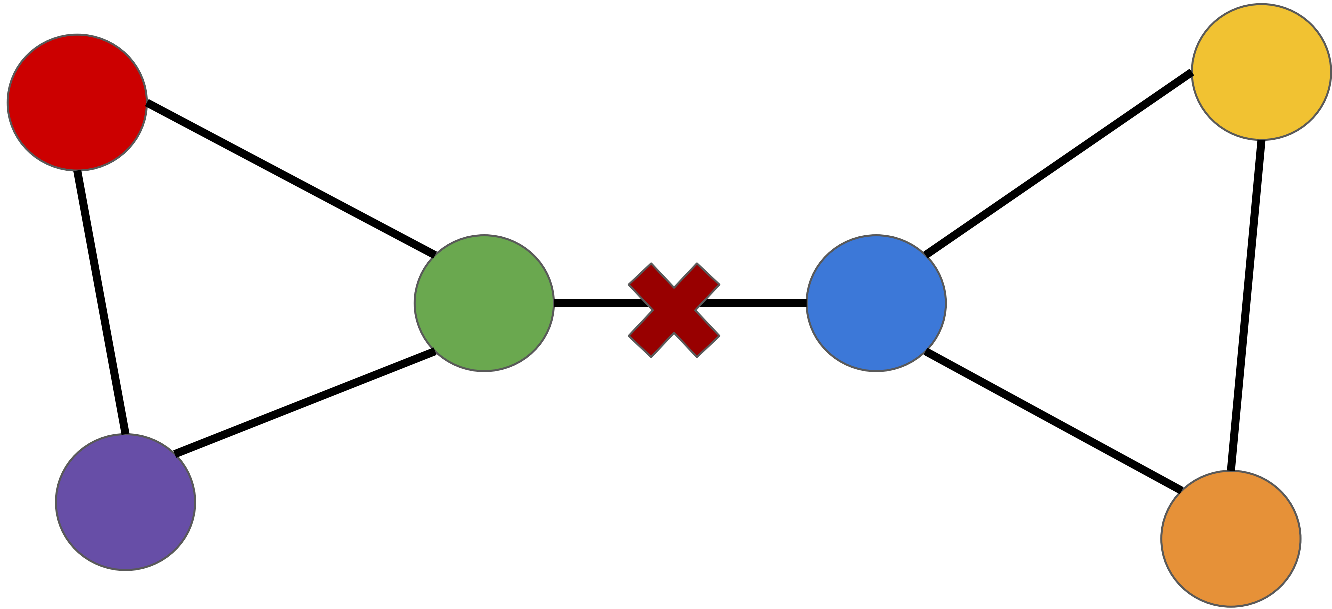


- The **degree** of a vertex is the number of edges touching it.
- The size of a minimum cut of a graph is surely no greater than the smallest degree of any of its vertices.
- (Why? Because we can just remove all of that vertex's edges! Then the vertex is its own connected component.)

Something to Ponder

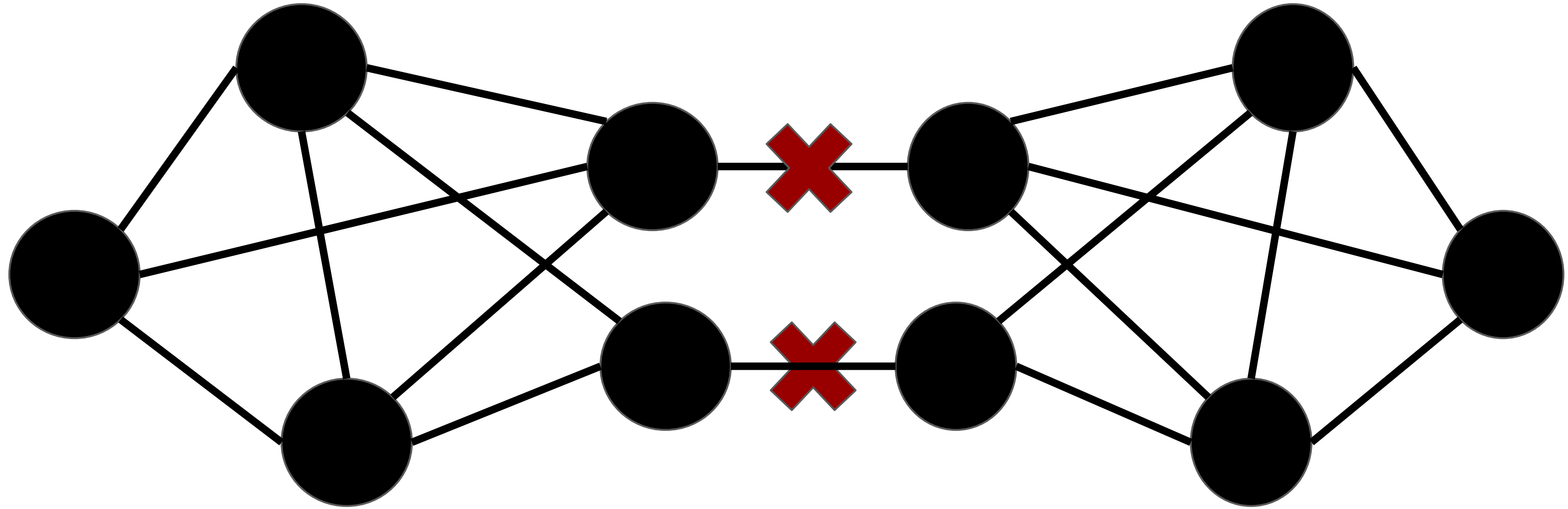
Can the size of a graph's minimum cut possibly be smaller than the degrees of any of its vertices?

Yes!



Here every vertex has degree at least 2, but we only need to remove the middle edge to disconnect the graph.

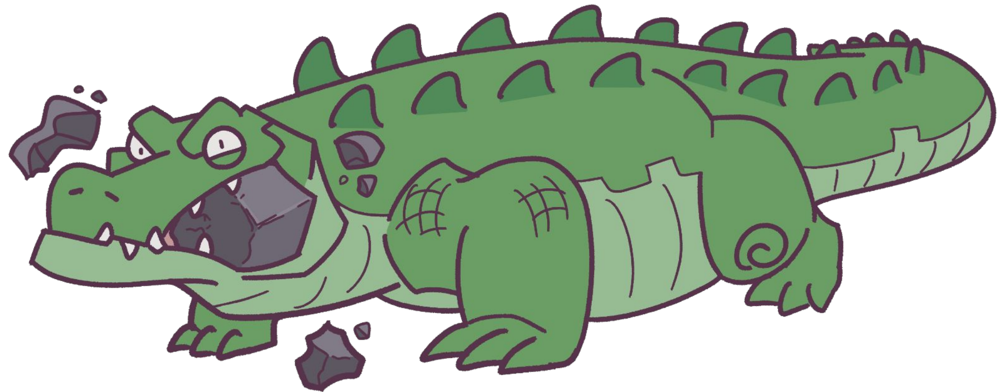
One more thing about min cuts...



A min cut might be a set of edges that do not share a vertex.

Min Cut is a hard problem!

- What would a **brute force** solution look like?
  - Check all possible subsets of edges.
  - For each such subset, see if removing it disconnects the graph.
    - Take any smallest subset of edges like this.



This is maybe not a job for Brutus

- In a graph with n vertices, there could be an edge between any two of them!
  - This is n choose 2, which is $(n)(n-1) / 2 = O(n^2)$.

- In a graph with E edges, there are 2^E ways to select a subset of them. (Each one either is or isn't included.)

- So there are $O(2^{(n^2)})$ possible cuts to check. Yeowwwch!
  - And we haven't even learned yet how much time it takes to count the number of connected components! (Coming in Unit 4)

So what do we do?

- Min Cut can be solved deterministically using a technique called Max Flow, which is coming in Unit 6.
- However, there is a beautiful randomized approach as well, called Karger's algorithm.
- We're going to focus on the *idea* and the *analysis*, not the implementation.

Haystack Grou
Research on In
Access, Analys
Management, a
Distribution

People

David Karger
PhD Candidate

Normally...

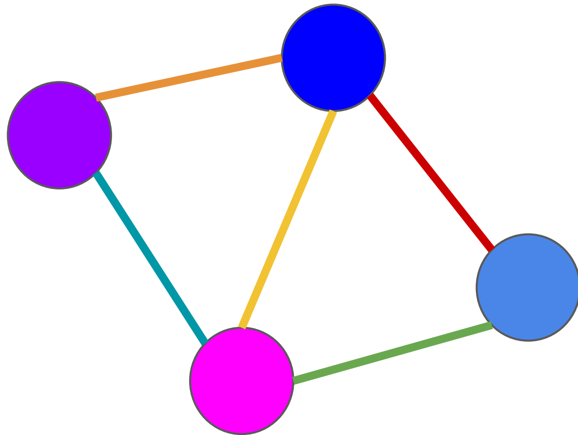
- I like to give some intuition on how one might come up with an idea.
- But this case is more like Strassen's matrix multiplication algorithm in Problem 4 of Homework 1, where it's hard to imagine anyone coming up with it.
- This one does something 👻*weird*👻 with graphs that we won't see anywhere else in the course.

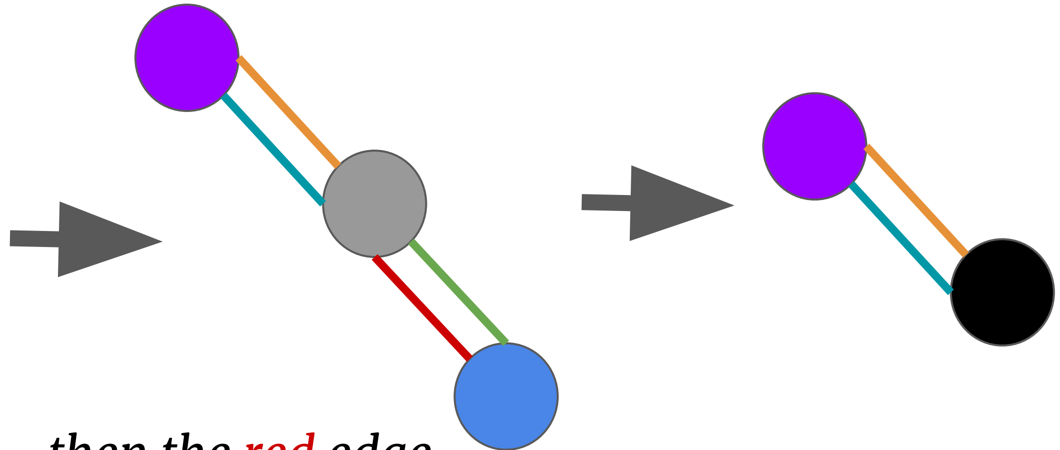
Edge Contractions

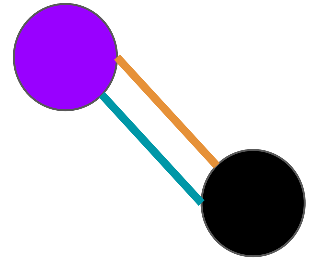
To *contract* an edge of a graph:

- merge its two vertices
- all their edges come along for the ride
- delete any edge(s) between the two vertices

suppose we contract the yellow edge



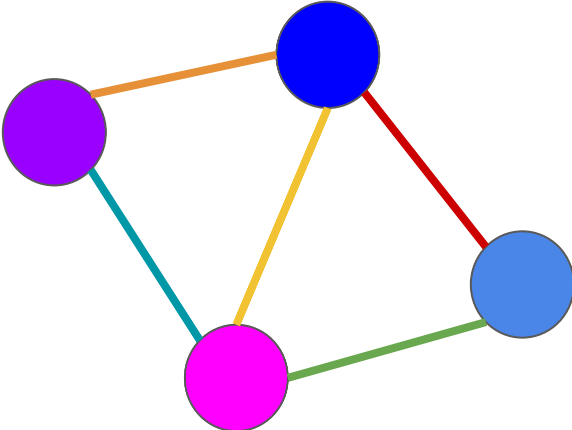


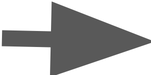


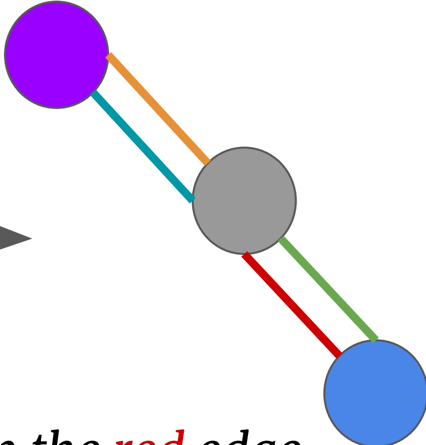
then the red edge

Here's something interesting...

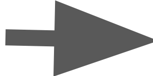
suppose we contract the yellow edge

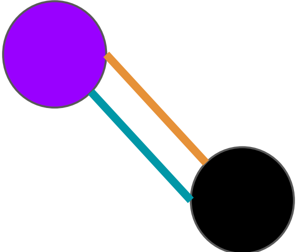






then the red edge





By doing this, we happened to find a min cut (the orange and teal edges).

Karger's Algorithm

- Repeat the following until there are only two vertices left in the graph:
    - Select an edge uniformly at random from all remaining edges. *(not from all remaining pairs of vertices)*
    - Contract that edge.
- Return the set of edges between the vertices as the min cut.

What if that fails?

- OK, fine. Do this a lot:

  - Repeat the following until there are only two vertices left in the graph:
    - Select an edge uniformly at random from all remaining edges.
    - Contract that edge.
  - Return the set of edges between the vertices.

- Return the smallest cut found as the min cut.

What if that fails?

Side note: this is a Monte Carlo algorithm!

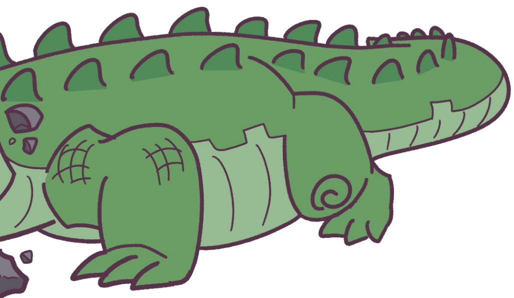
- OK, fine. Do this a lot:

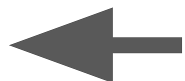
  - Repeat the following until there are only two vertices left in the graph:
    - Select an edge uniformly at random from all remaining edges.
    - Contract that edge.
  - Return the set of edges between the vertices.

- Return the smallest cut found as the min cut.

How much is "a lot"? Why would this even work?

- It's time for some more **MATH**





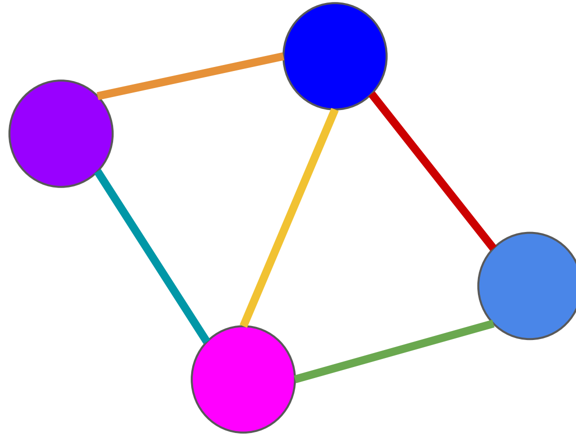
Brutus is leaving

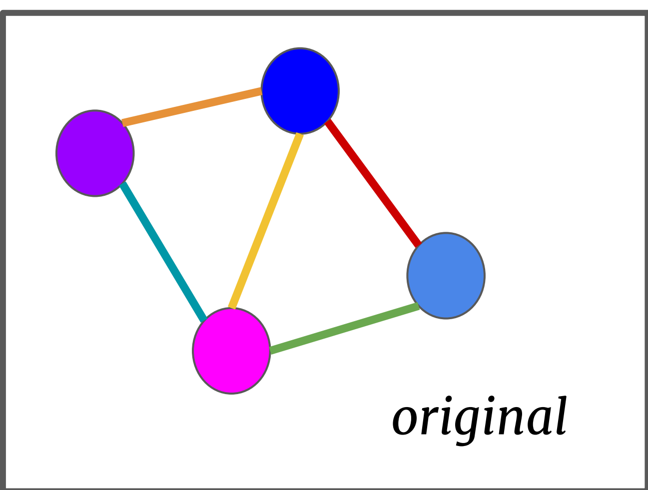
- Suppose we have an arbitrary graph. Consider any single min cut M in the graph.
  - *There may be more than one, but we won't need to think about them to make our point.*

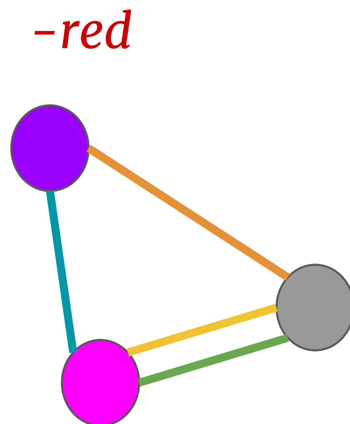
- Let's find the probability that this cut survives one instance of Karger's procedure.

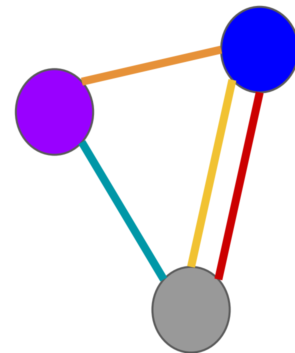
- If we ever lose *even one* of those edges, we fail!

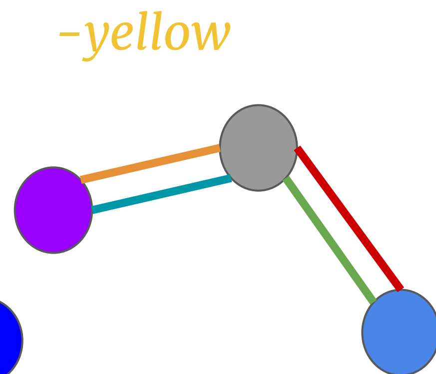
- Let M be the orange and teal edges.
- What's the probability that we are still OK after our first edge contraction?





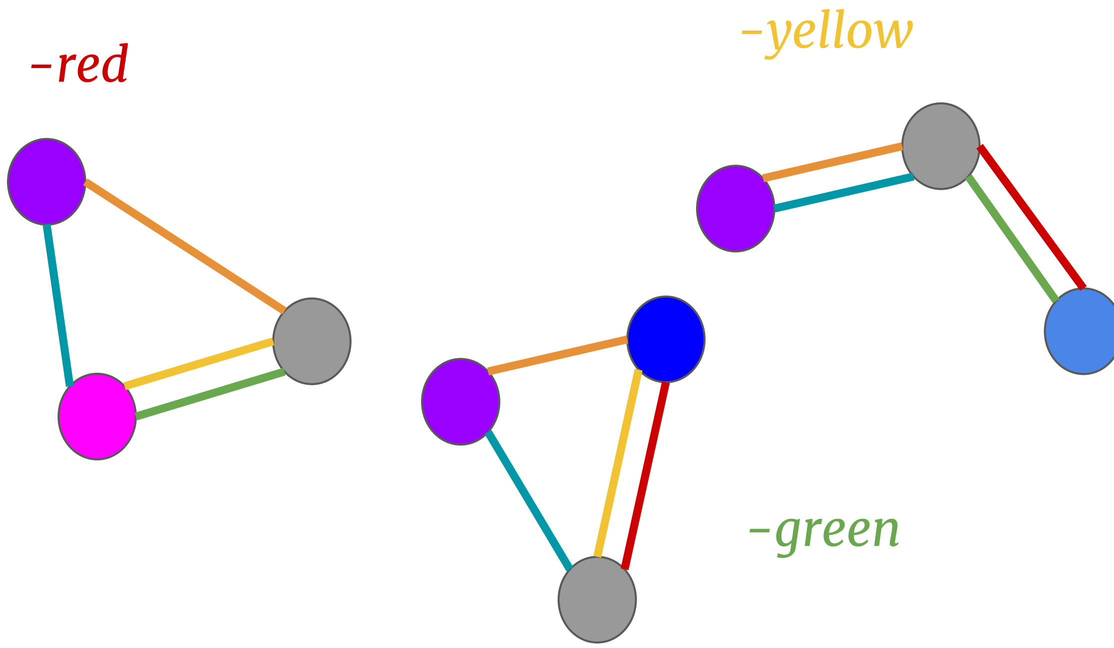






With 2/5 probability, we contract orange or teal and we lose.

With 3/5 probability, we're still OK.

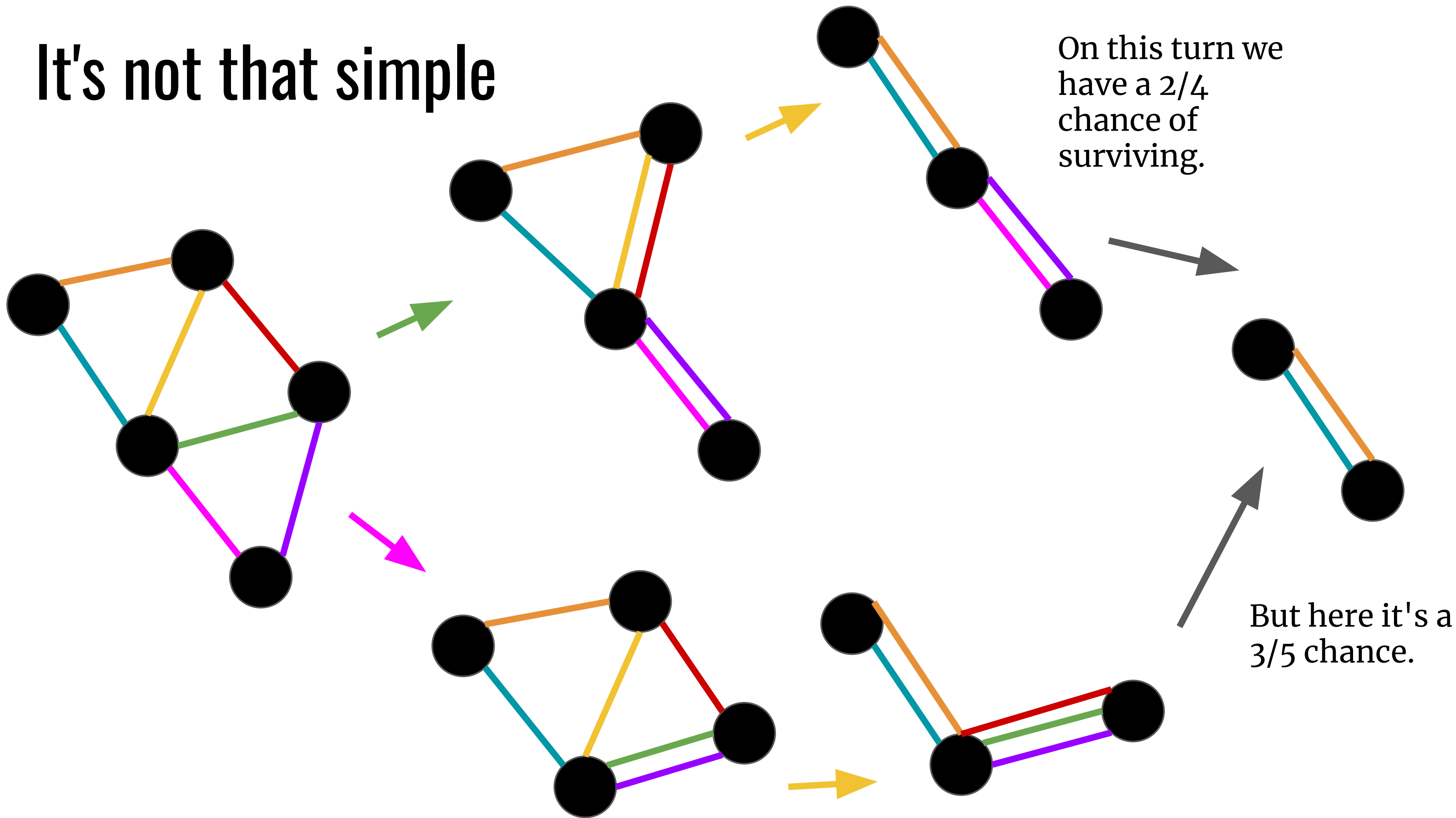


Then, in any of these scenarios, we lose on the next turn with probability 2/4. Otherwise, with probability 2/4, we win immediately! *(This takes some checking.)*

So: success probability is 3/5 * 2/4 = 3/10.

Pattern: Is the answer always just $(n-|M|)/n * (n-|M|-1)/(n-1) * \ldots$, where $|M|$ is the size of the min cut?

It's not that simple

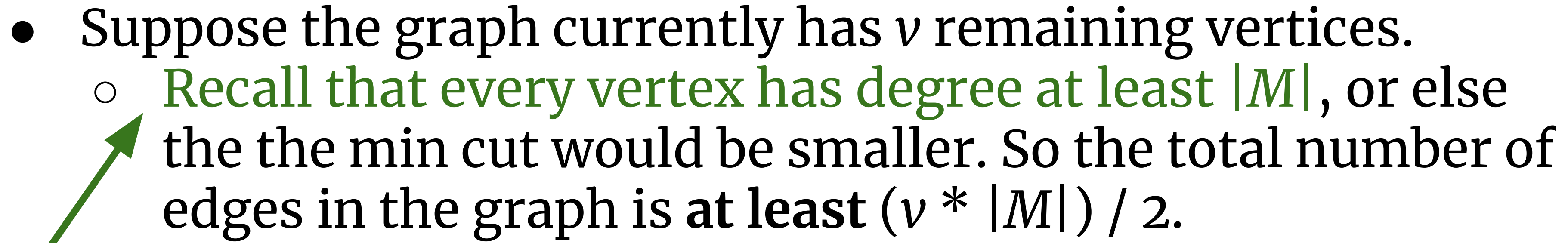


- Suppose we are at some stage of the algorithm, and we have not lost yet.

- Let $|M|$ be the size of our min cut M.

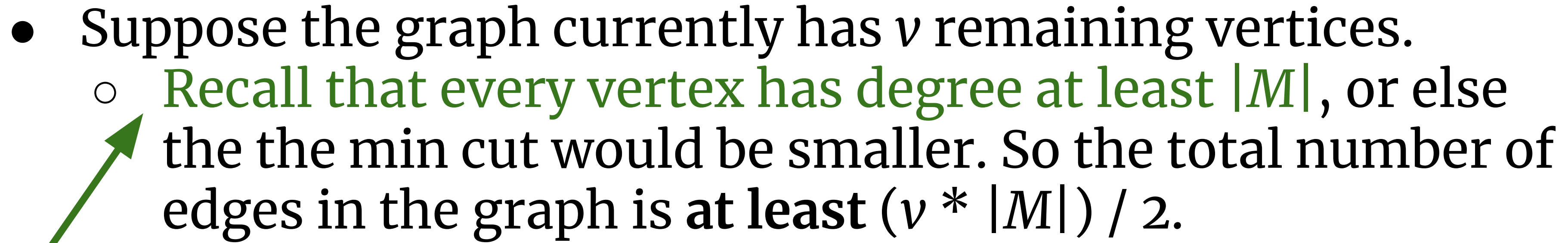
- Suppose the graph currently has v remaining vertices.
  - Recall that every vertex has degree at least $|M|$, or else the the min cut would be smaller. So the total number of edges in the graph is **at least** $(v * |M|) / 2$. *the /2 is since each edge is counted twice*

- We just need to not pick one of the $|M|$ edges of M. So we lose with probability **at most** $|M| / ((v * |M|) / 2) = 2 / v$.
  - and survive with probability at least $(v-2) / v$.



- Suppose the graph currently has v remaining vertices.
  - Recall that every vertex has degree at least $|M|$, or else the the min cut would be smaller. So the total number of edges in the graph is **at least** $(v * |M|) / 2$.

But how do we know this is still true at any stage of the algorithm?



- Suppose the graph currently has v remaining vertices.
  - Recall that every vertex has degree at least $|M|$, or else the the min cut would be smaller. So the total number of edges in the graph is **at least** $(v * |M|) / 2$.

But how do we know this is still true at any stage of the algorithm?

Suppose that at some point, we had a vertex with fewer than $|M|$ edges. But then those edges would be an even better min cut in the original graph.

Putting it together

- In a round in which we have *v* remaining vertices, we survive with probability at least (*v*-2) / *v*.
- We start with *n* vertices and each round eliminates one. We stop when we're down to 2.
- So our overall survival chance is

$$P(\text{survival}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \dots \cdot \frac{2}{4} \cdot \frac{1}{3}$$

Putting it together

Note that this doesn't depend on the number of edges! This sidesteps the earlier problem.

- In a round in which we have v remaining vertices, we survive with probability at least $(v-2) / v$.
- We start with n vertices and each round eliminates one. We stop when we're down to 2.
- So our overall survival chance is

$$P(\text{survival}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \ldots \cdot \frac{2}{4} \cdot \frac{1}{3}$$

Putting it together

- In a round in which we have v remaining vertices, we survive with probability at least $(v-2) / v$.

- We start with n vertices and each round eliminates one. We stop when we're down to 2.

- So our overall survival chance is

$$P(\text{survival}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \frac{2}{4} \cdot \frac{1}{3}$$

Putting it together

- In a round in which we have *v* remaining vertices, we survive with probability at least (*v*-2) / *v*.

- We start with *n* vertices and each round eliminates one. We stop when we're down to 2.

- So our overall survival chance is

$$P(\text{survival}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \ldots \cdot \frac{2}{4} \cdot \frac{1}{3}$$

Putting it together

- In a round in which we have v remaining vertices, we survive with probability at least $(v-2) / v$.
- We start with n vertices and each round eliminates one. We stop when we're down to 2.
- So our overall survival chance is

$$P(\text{survival}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \cdot \frac{2}{4} \cdot \frac{1}{3} = 2 / (n(n-1))$$

This is huge! (figuratively)

- How many times do we need to try, in expectation, to get our first success when the probability of succeeding is $\mathbf{2 / (n(n-1))}$?
- This is a *geometric distribution*. The expectation is the inverse of the success probability.

like how you need to roll a 6-sided die 6 times on average to see a 1

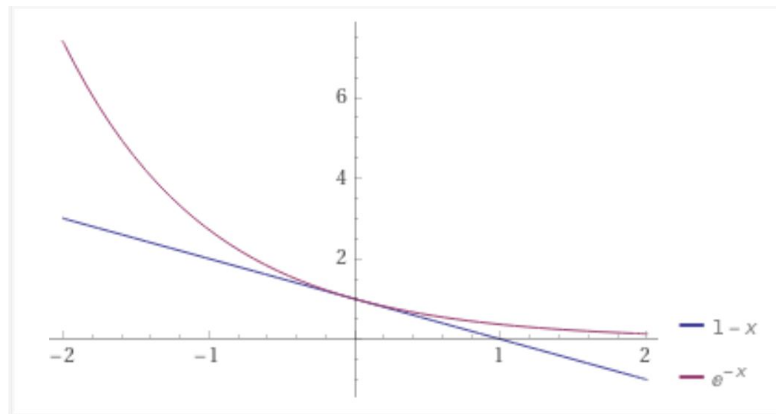
- So **in expectation**, it takes us only $n(n-1)/2 = O(n^2)$ tries to get our min cut.
    - *but this doesn't guarantee it...*

But how many times do we actually need to try

- Say we want to find the number k of trials needed to be 99% sure we find our min cut M.
- Probability of failing every time: $(1 - \frac{2}{n(n-1)})^k$
- ..., so, of succeeding at least once: $1 - (1 - \frac{2}{n(n-1)})^k$
- Set that last quantity equal to 0.99, and we can solve for k!

Let ε be the error probability we're willing to tolerate.

$$(1 - \frac{2}{n(n-1)})^k = \epsilon$$



A useful fact: $(1 - x) \leq e^{-x}$ ← *this trick shows up all over CS theory!*

So $(1 - \frac{2}{n(n-1)})^k \leq (e^{-\frac{2}{n(n-1)}})^k = e^{-\frac{2k}{n(n-1)}}$

Therefore: $\epsilon \leq e^{-\frac{2k}{n(n-1)}}$. $\frac{2k}{n(n-1)} \geq -\log_e \epsilon$. $\boxed{k \geq \frac{n(n-1)}{2} \log_e \frac{1}{\epsilon}}$.

$$(1 - \frac{2}{n(n-1)})^k = \epsilon$$

A useful fact: $(1 - x) \leq e^{-x}$

So $(1 - \frac{2}{n(n-1)})^k \leq (e^{-\frac{2}{n(n-1)}})^k = e^{-\frac{2k}{n(n-1)}}$

Therefore: $\epsilon \leq e^{-\frac{2k}{n(n-1)}}$. $\frac{2k}{n(n-1)} \geq -\log_e \epsilon$. $\boxed{k \geq \frac{n(n-1)}{2} \log_e \frac{1}{\epsilon}}$.

The upshot: We need O(n^2 log (1/ε)) trials.

Some details

- The algorithm never has to do any checking to see that the min cut actually disconnects the graph!!

- The algorithm's success guarantees only depend on the size of the graph, not on the size of the min cut!!

- Why is it enough to succeed just once?
  - Because we take the smallest cut we find. All the failures don't really matter.

- What if there are multiple min cuts?
  - This only helps us! We might find one of those instead.

So... why does this work?!?!?!

- The min cut has to be small relative to the total number of edges in the graph. (It's at most the minimum degree!)
- As long as we don't remove the min cut edges, contracting helps us get rid of all the other unwanted edges faster? (HW2 will have you see what happens if you *don't* contract edges.)
- But I wasn't satisfied... so...

Ian Tullis <tigupine@gmail.com> Tue, Jun 28, 10:58 PM (13 hours ago)

to karger

Prof. Karger:

Hi! I'm a master's student in CS at Stanford, studying theory. This summer I'm the primary instructor for Stanford's core algorithms course, and tomorrow I'm teaching my students about randomized algorithms. I'm including your 1993 min-cut algorithm and telling my students (truthfully!) that it is my favorite algorithm.

I like to give my students some sense of how one might conceivably come up with each algorithm we study, but sometimes (as with Strassen's) it is hard to imagine how anyone had that particular flash of inspiration. Your algorithm is one of those; I've since seen edge contractions in other contexts, but it's really hard to imagine the intuition for why they seemed promising here. I was wondering if you have any quick words about what inspired the idea.

(Apologies if you're tired of people writing to you about this now three-decades-old work, like fans of a band always requesting a particular classic!)

Thanks,
Ian

PS - I noticed that your https://haystack.csail.mit.edu/#projects site lists you as a "Ph.D. Candidate". Then there's truly no hope for the rest of us :D

Ian Tullis <tigupine@gmail.com>
to karger

Tue, Jun 28, 10:58 PM (13 hours ago)

Prof. Karger:

Hi! I'm a master's student in CS at Stanford, studying theory. This summer I'm the primary instructor for Stanford's core algorithms course, and tomorrow I'm teaching my students about randomized algorithms. I'm including your 1993 min-cut algorithm and telling my students (truthfully!) that it is my favorite algorithm.

I like to give my students some sense of how one might conceivably come up with each algorithm we study, but sometimes (as with Strassen's) it is hard to imagine how anyone had that particular flash of inspiration. Your algorithm is one of those; I've since seen edge contractions in other contexts, but it's really hard to imagine the intuition for why they seemed promising here. I was wondering if you have any quick words about what inspired the idea.

(Apologies if you're tired of people writing to you about this now three-decades-old work, like fans of a band always requesting a particular classic!)

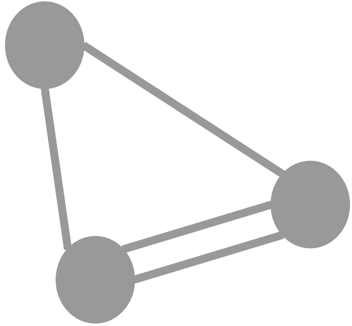
Thanks,
Ian

PS - I noticed that your https://haystack.csail.mit.edu/#projects site lists you as a "Ph.D. Candidate". Then there's truly no hope for the rest of us :D

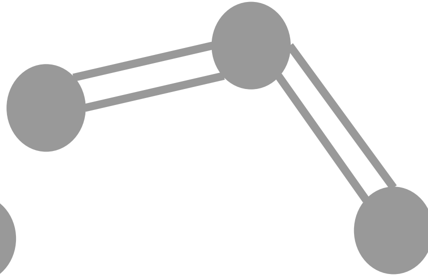
This morning: He sent back an amazingly detailed and thoughtful response!!!!!

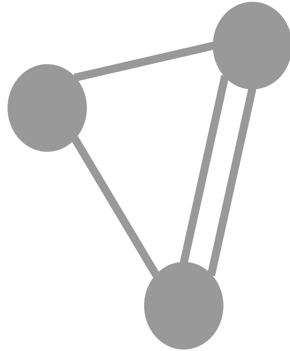
I'll post it in Ed (he has given permission)

–red



–yellow





–green

Then, in any of these scenarios, we lose on the next turn with probability 2/4. Otherwise, with probability 2/4, we win immediately! *(This takes some checking.)*

So: success probability is 3/5 * 2/4 = 3/10.

This was sort of correct, but not... you'll see on HW2.

Pattern: Is the answer always just (n-k)/n * (n-k-1)/(n-1) * ...?

This is my favorite algorithm!

- What about the rest of the staff's favorite algorithms/ data structures?
    - **Goli:** Fast Fourier Transform, Segment Trees, Shor's (quantum) Algorithm
    - **Ivan:** Camerini's Algorithm, Gradient Descent
    - **Lucas:** Binary Indexed Trees
    - **Ricky:** Euclid's (GCD) Algorithm
    - **Rishu:** Rainbow coloring of graph edges
    - **Ziang:** Rapidly-Exploring Random Tree

Next Week!

- No class on Monday!
- Lectures are on Wednesday and Friday. We'll study some great data structures! Including hash tables!
    - (Sisi and Indy are happy.)