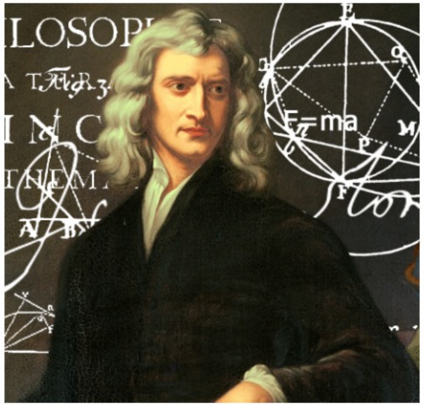
Randomized Algorithms

Slides credits: Ariel Procaccia, Anil Ada

Randomized and the universe



Newtonian physics suggests that the universe evolves deterministically.

Quantum physics says otherwise.

Randomized and the universe

Does the universe have true randomness?

Even if it doesn't, we can still model our uncertainty about things using probability.

Randomness is an essential tool in modeling and analyzing nature.

It also plays a key role in computer science.

Randomized in Computer Science

Randomized algorithms (our focus today)

Does randomness speed up computation?

Statistics via sampling

e.g. election polls

Nash equilibrium in Game Theory

Nash equilibrium always exists if players can have probabilistic strategies.

Cryptography

A secret is only as good as the entropy/uncertainty in it.

Randomized in Computer Science

Randomized models for deterministic objects

e.g. the www graph

Quantum computing

Randomness is inherent in quantum mechanics.

Machine learning theory

Data is generated by some probability distribution.

Coding Theory

Encode data to be able to deal with random noise.

…

Randomized Algorithms

How can randomness be used in computation?

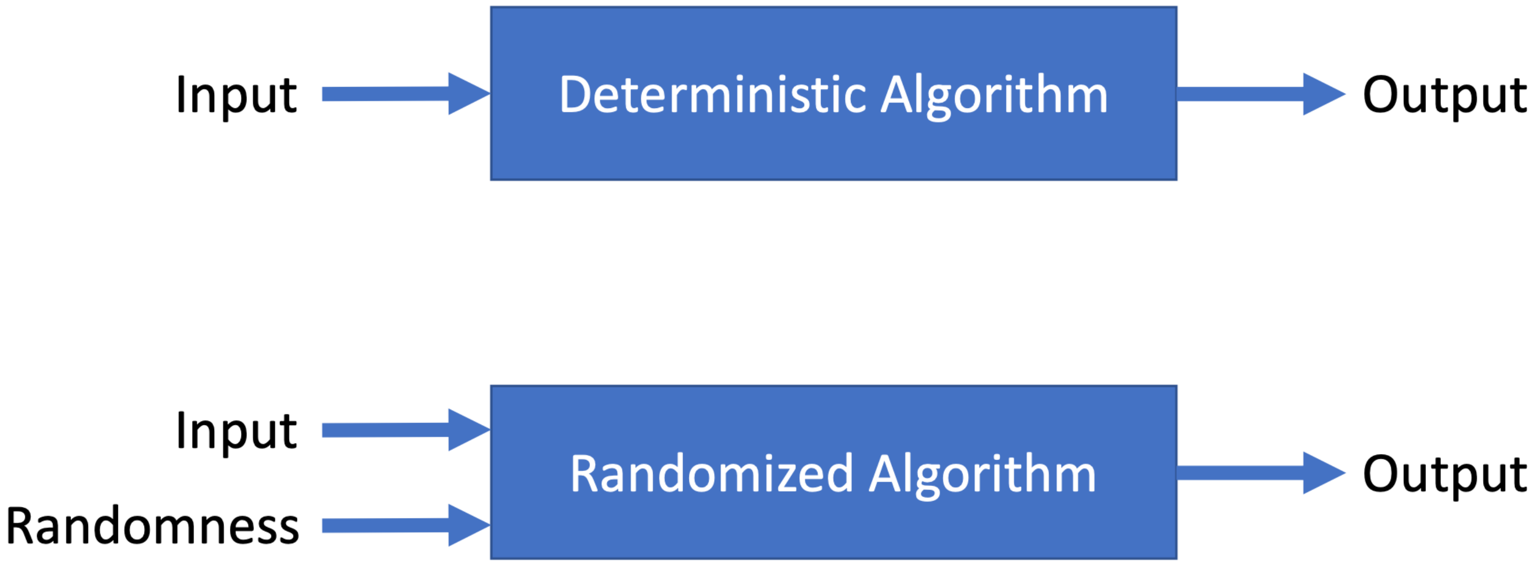
Where can it come into the picture?

Given some algorithm that solves a problem...

~~- What if the input is chosen randomly?~~

- What if the algorithm can make random choices?

Randomized Algorithms



Randomized Algorithms

A randomized algorithm is an algorithm that is allowed to flip a coin.

(it can make decisions based on the output of the coin flip.)

Today's lecture:

A randomized algorithm is an algorithm that is allowed to call:

- RandInt(n)
- Bernoulli(p)

(we'll assume these take O(1) time)

Randomized Algorithms

An Example

```
def f(x):
    y = Bernoulli(0.5)
    if(y == 0):
        while(x > 0):
            print("What up?")
            x = x - 1
    return x+y
```

For a fixed input (e.g. x = 10)

- the output can vary
- the running time can vary

Randomized Algorithms

For a randomized algorithm, how should we:

- measure its correctness?
- measure its running time?

If we require it to be

- always correct, and
- always runs in time $O\big(T(n)\big)$

then we have a deterministic algorithm running in this time.

(Why?)

Randomized Algorithms

So for a randomized algorithm to be interesting:

- it is not correct all the time, **or**
- it doesn't always run in time $O(T(n))$,

(It either gambles with correctness or running time.)

Types of randomized algorithms

Given an array with n elements (n even). A[1 ... n].
Half of the array contains 0s, the other half contains 1s.

Goal: Find an index that contains a 1.

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

Doesn't gamble with correctness
Gambles with run-time

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

Gambles with correctness
Doesn't gamble with run-time

Types of randomized algorithms

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

$$\mathbf{Pr}[\text{failure}] = \frac{1}{2^{300}}$$

Worst-case running time: $O(1)$

This is called a Monte Carlo algorithm.

(gambles with correctness but not time)

Types of randomized algorithms

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

$$\mathbf{Pr}[\text{failure}] = \ 0$$

Worst-case running time: can't bound
(could get super unlucky)

Expected running time: $O(1)$
(2 iterations)

This is called a Las Vegas algorithm.

(gambles with time but not correctness)

Types of randomized algorithms

Given an array with n elements (n even). A[1 ... n].
Half of the array contains 0s, the other half contains 1s.

Goal: Find an index that contains a 1.

	Correctness	Run-time
Deterministic	always	$\Omega(n)$
Monte Carlo	w.h.p.	$O(1)$
Las Vegas	always	$O(1)$ w.h.p.

w.h.p. = with high probability

Formal definition: Monte Carlo algorithms

Let f be a computational problem.

Suppose A is a randomized algorithm such that

$\forall$ input x: $\Pr[A(x) \neq f(x)] \leq \epsilon$

$\forall$ input x: # Steps $A(x)$ takes is $\leq T(|x|)$
(no matter what random choices are)

Then we say A is a $T(n)$-time Monte Carlo algorithm for problem f with ϵ probability of error.

Formal definition: Las Vegas algorithms

Let f be a computational problem.

Suppose A is a randomized algorithm such that

$\forall$ input x: $\quad A(x) = f(x)$

$\forall$ input x: $\quad E[\#\text{ steps } A(x) \text{ takes}] \leq T(|x|)$.

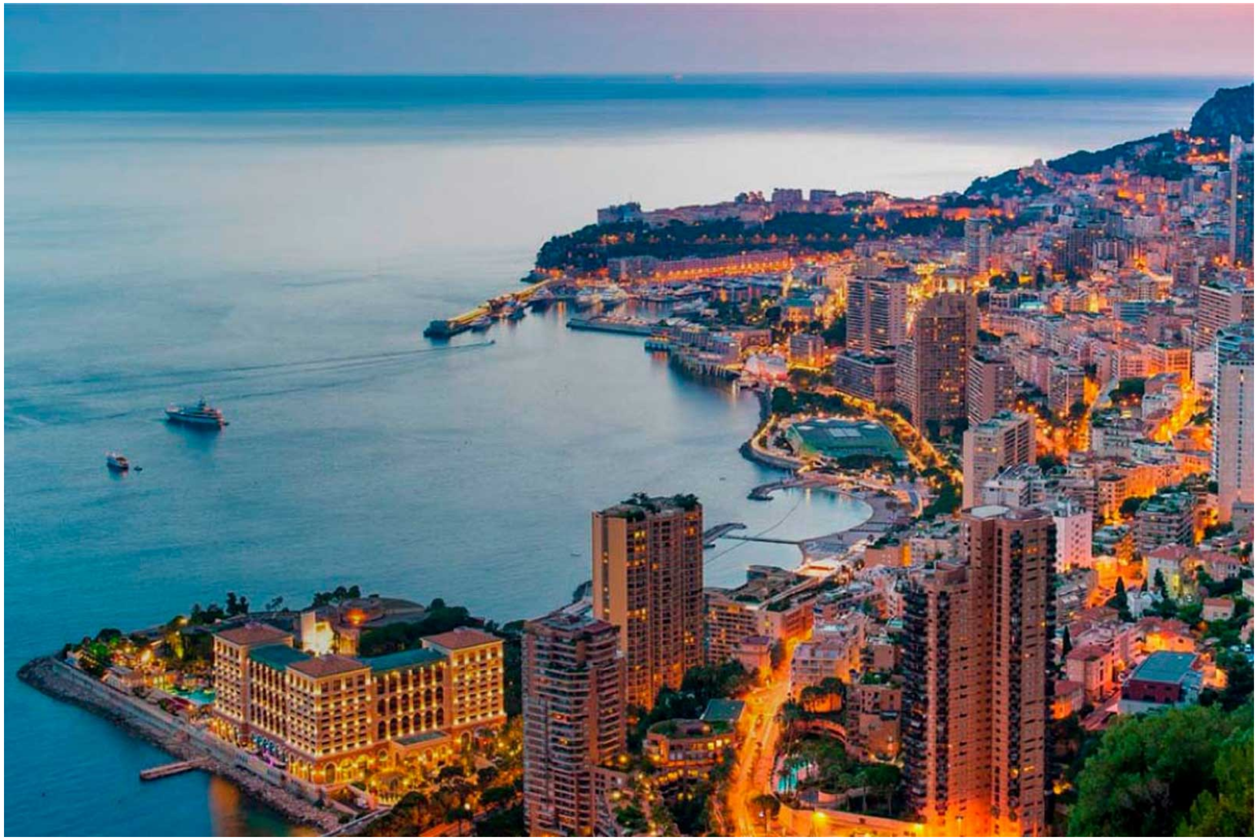
Then we say A is a $T(n)$-time Las Vegas algorithm for problem f.

We will discuss two randomized algorithms:

Example of a Monte Carlo algorithm:
Min Cut

Example of a Las Vegas algorithm:
QuickSort

Example of a Monte Carlo Algorithm (Min Cut)

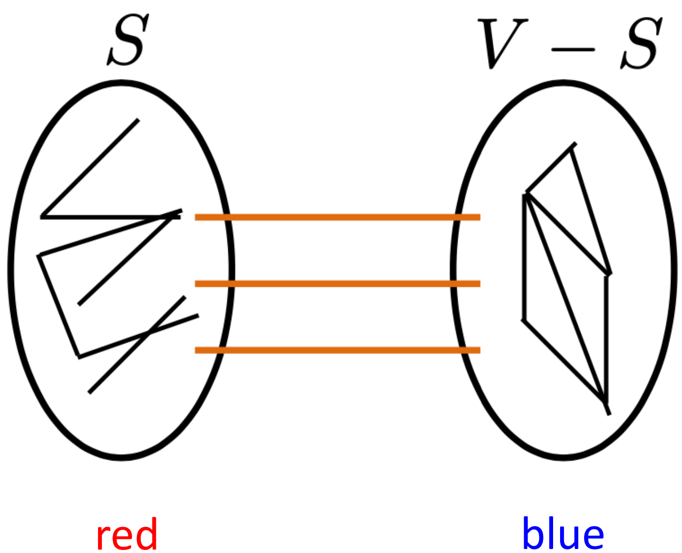


Gambles with correctness.
Doesn't gamble with running time.

Cut Problems

Max Cut Problem

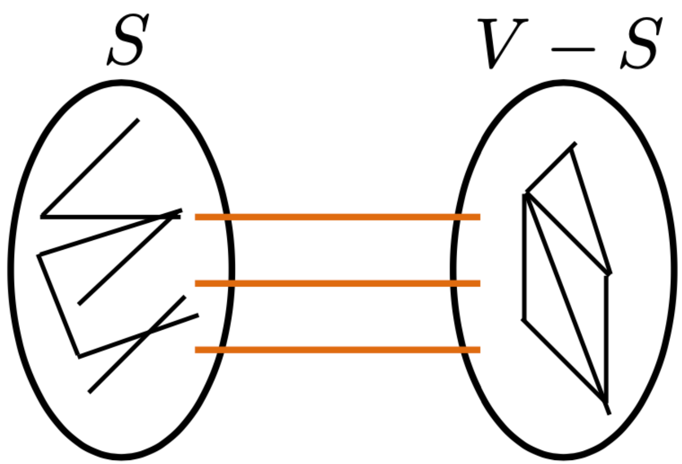
Given a graph $G = (V, E)$,
color the vertices red and blue so that the number of edges with two colors (e = {u,v}) is maximized.



Cut Problems

Max Cut Problem

Given a graph $G = (V, E)$,
find a non-empty subset $S \subset V$ such that
number of edges from S to $V - S$ is maximized.

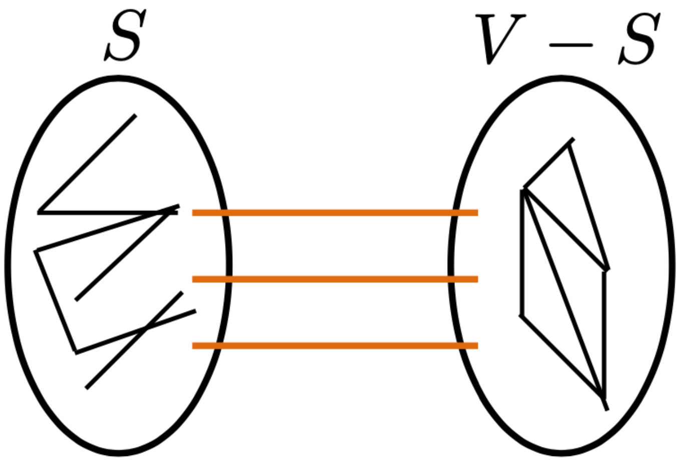


Size of the cut = # edges from S to $V - S$.

Cut Problems

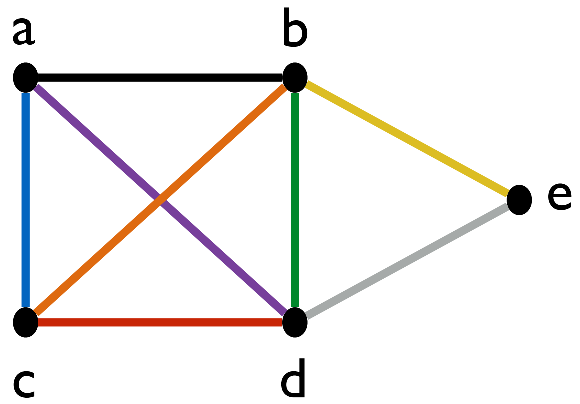
Min Cut Problem

Given a graph $G = (V, E)$,
find a non-empty subset $S \subset V$ such that
number of edges from S to $V - S$ is minimized.



Size of the cut = # edges from S to $V - S$.

A Monte Carlo Min Cut Algorithm



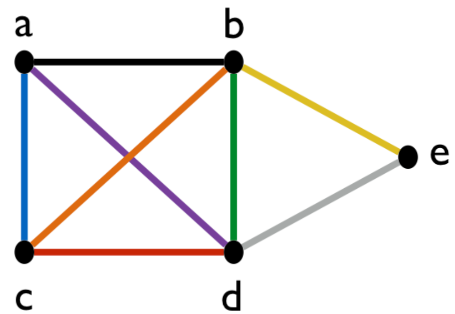
Select an edge randomly:

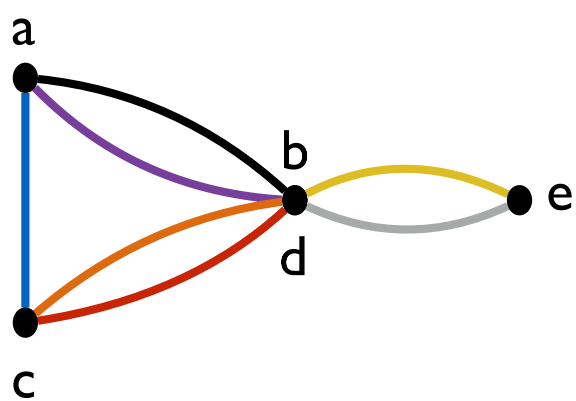
Size of min-cut: 2

Green edge selected.

Contract that edge.

A Monte Carlo Min Cut Algorithm





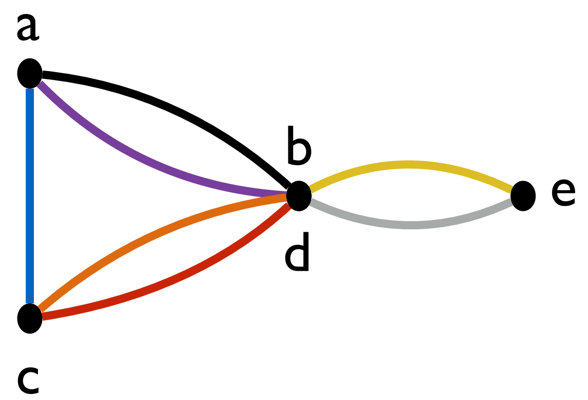
Select an edge randomly:

Size of min-cut: 2

Green edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm



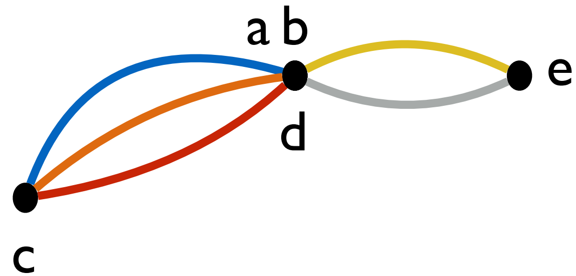
Select an edge randomly:

Size of min-cut: 2

Purple edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm

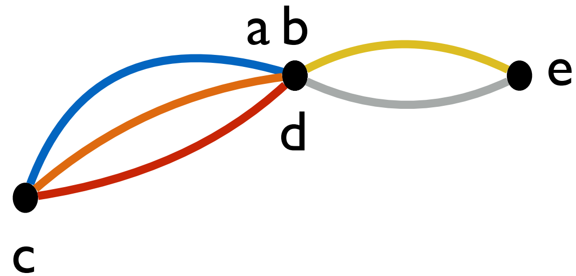


Select an edge randomly: *Size of min-cut: 2*

Purple edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm



Select an edge randomly: *Size of min-cut: 2*

Blue edge selected.

Contract that edge. (delete self loops)

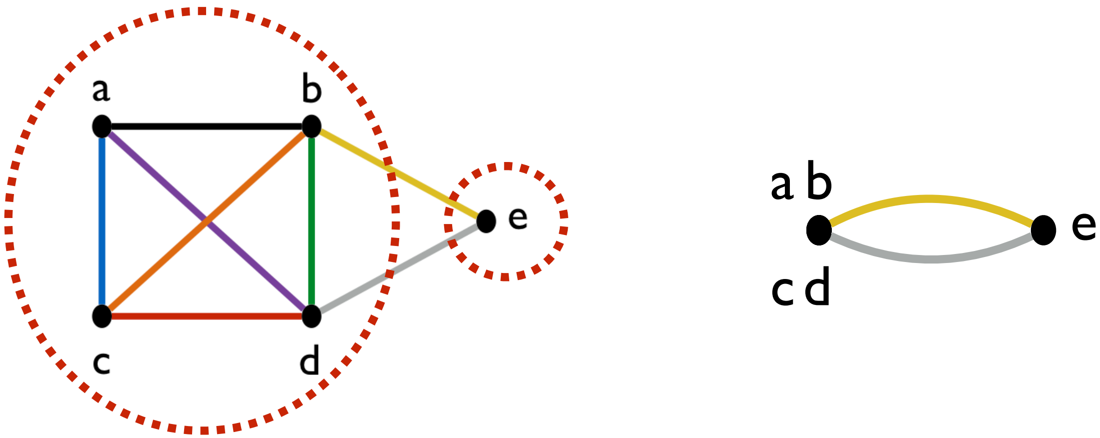
A Monte Carlo Min Cut Algorithm

Select an edge randomly: *Size of min-cut: 2*

Blue edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm



Select an edge randomly: *Size of min-cut: 2*

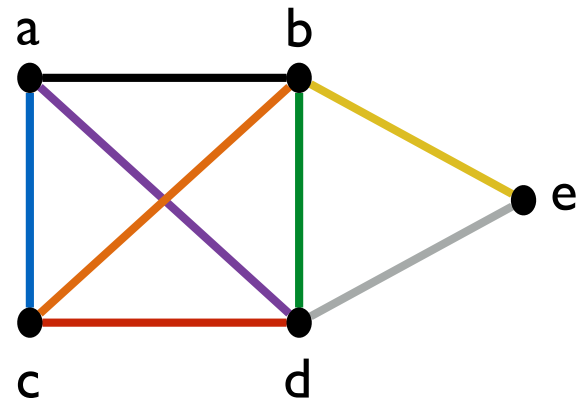
Blue edge selected.

Contract that edge. (delete self loops)

When two vertices remain, you have your cut:

{a, b, c, d} {e} *size: 2*

A Monte Carlo Min Cut Algorithm



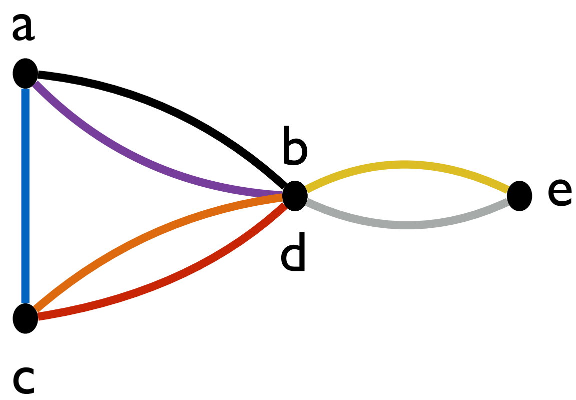
Select an edge randomly:

Size of min-cut: 2

Green edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm

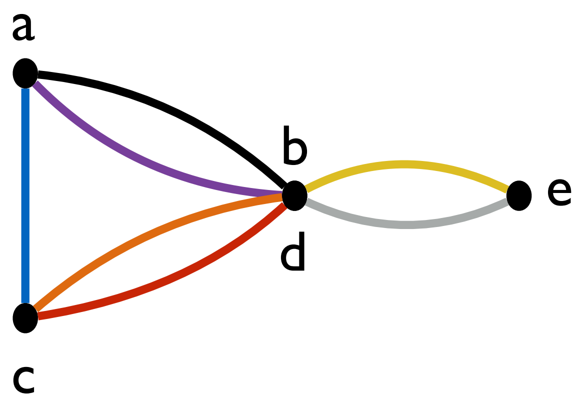


Select an edge randomly: *Size of min-cut: 2*

Green edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm

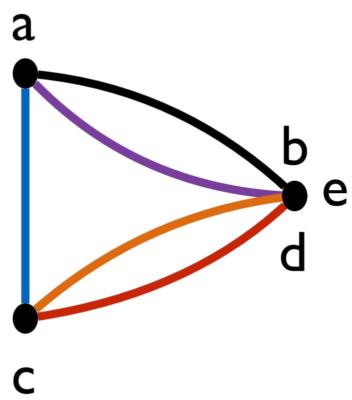


Select an edge randomly: *Size of min-cut: 2*

Yellow edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm

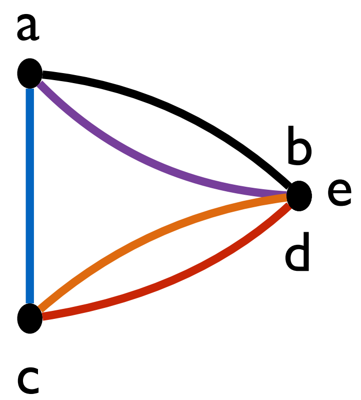


Select an edge randomly: *Size of min-cut: 2*

Yellow edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm

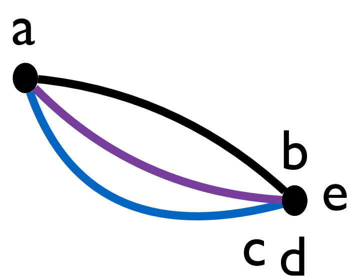


Select an edge randomly: *Size of min-cut: 2*

Red edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm

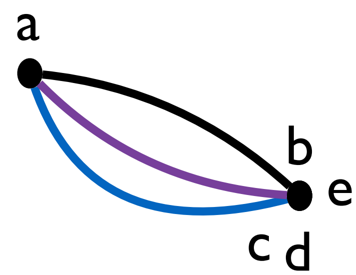


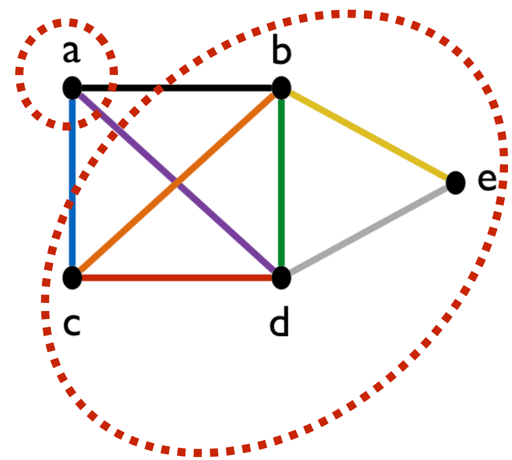
Select an edge randomly: *Size of min-cut: 2*

Red edge selected.

Contract that edge. (delete self loops)

A Monte Carlo Min Cut Algorithm





Select an edge randomly:

Size of min-cut: 2

Red edge selected.

Contract that edge. (delete self loops)

When two vertices remain, you have your cut:

{a} {b,c,d,e} *size: 3*

$$G = G_0 \xrightarrow{\text{contract}} G_1 \xrightarrow{\text{contract}} G_2 \xrightarrow{\text{contract}} \cdots \xrightarrow{\text{contract}} G_{n-2}$$

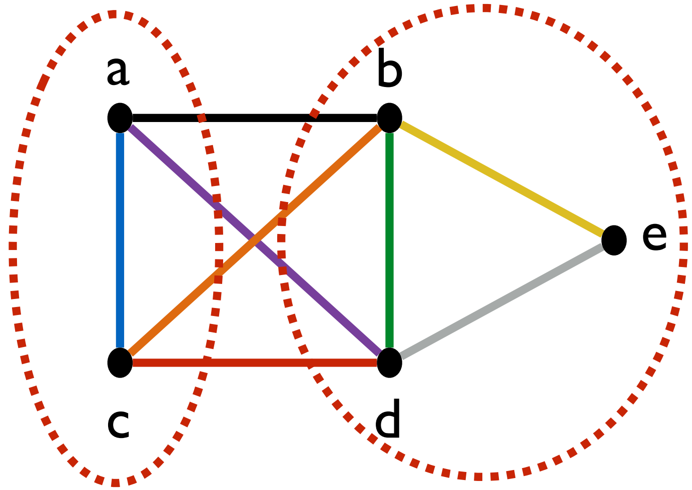
$\boxed{n \text{ vertices}}$ $\boxed{2 \text{ vertices}}$

$n-2$ iterations

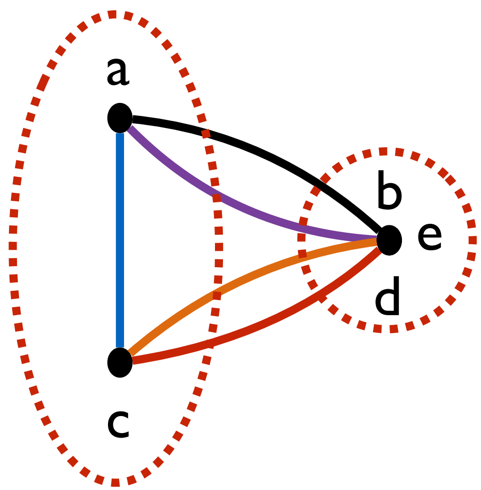
Observation:

For any i: A cut in G_i of size k corresponds exactly to a cut in G of size k.

G



G_i



Let k be the size of a minimum cut.

Which of the following are true (can select more than one):

For $G = G_0$, $\qquad k \leq \min_v \deg_G(v)$

For every G_i, $\qquad k \leq \min_v \deg_{G_i}(v)$

For every G_i, $\qquad k \geq \min_v \deg_{G_i}(v)$

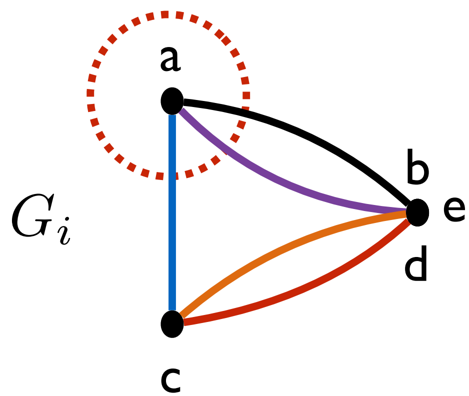
For $G = G_0$, $\qquad k \geq \min_v \deg_G(v)$

For every G_i , $k \le \min_v \deg_{G_i}(v)$

i.e., for every G_i and every $v \in G_i$, $k \le \deg_{G_i}(v)$

Why?

A single vertex v forms a cut of size $\deg(v)$.



This cut has size $\deg(a) = 3$.

Same cut exists in original graph.

So $k \le 3$.

Contraction algorithm for min cut

Theorem:

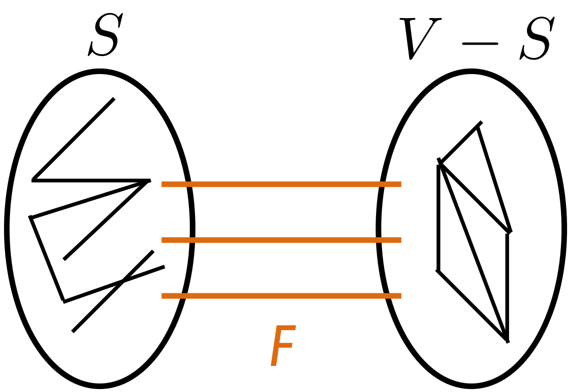
Let $G = (V, E)$ be a graph with *n* vertices.
The probability that the contraction algorithm will output a min-cut is $\geq 1/n^2$.

Should we be impressed?

- The algorithm runs in polynomial time.
- There are exponentially many cuts. (≈ 2^n)
- There is a way to boost the probability of success to $1 - \frac{1}{e^n}$ (and still remain in polynomial time)

Proof of theorem

Fix some minimum cut.



$|F| = k$
$|V| = n$
$|E| = m$

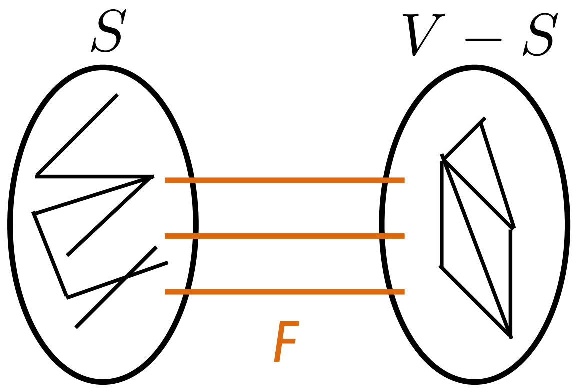
Will show $\Pr[\text{algorithm outputs } F] \geq 1/n^2$

(Note $\Pr[\text{success}] \geq \Pr[\text{algorithm outputs } F]$)

Proof of theorem

Fix some minimum cut.

$|F| = k$
$|V| = n$
$|E| = m$



When does the algorithm output F ?

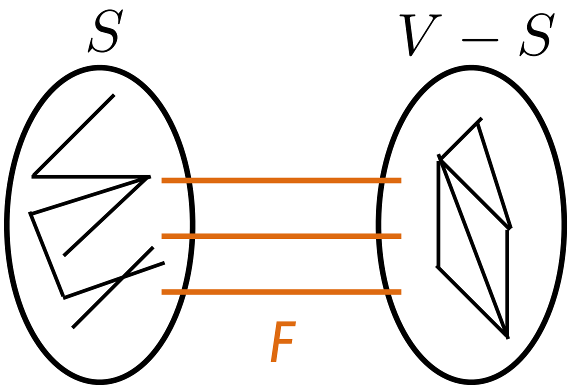
What if it never picks an edge in F to contract?
Then it will output F.

What if the algorithm picks an edge in F to contract?
Then it cannot output F.

Proof of theorem

Fix some minimum cut.

$|F| = k$
$|V| = n$
$|E| = m$



$\Pr[\text{algorithm outputs } F] =$

$\Pr[\text{algorithm never contracts an edge in } F] =$

$\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}]$

E_i = an edge in F is contracted in iteration i.

Proof of theorem

Let E_i = an edge in *F* is contracted in iteration i.

Goal: $\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \geq 1/n^2.$

$$\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}]$$

$$\overset{\text{chain rule}}{=} \Pr[\overline{E_1}] \cdot \Pr[\overline{E_2} | \overline{E_1}] \cdot \Pr[\overline{E_3} | \overline{E_1} \cap \overline{E_2}] \cdots$$

$$\Pr[\overline{E_{n-2}} | \overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-3}}]$$

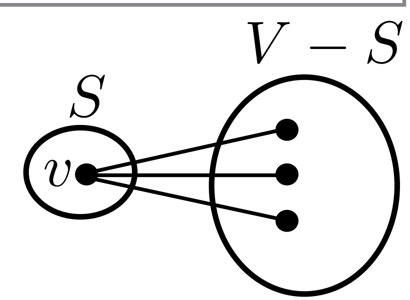
$$\Pr[\overline{E_1}] = 1 - \Pr[E_1] = 1 - \frac{\#\text{ edges in } F}{\text{total } \#\text{ edges}} = 1 - \frac{k}{m}$$

want to write in terms of *k* and *n*

Proof of theorem

Let E_i = an edge in *F* is contracted in iteration i.

<u>Goal</u>: $\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \geq 1/n^2.$



Observation: $\forall v \in V \ : \ k \leq \deg(v)$

Recall: $\underbrace{\sum_{v \in V} \deg(v)}_{\geq kn} = 2m \implies 2m \geq kn$

$$\implies m \geq \frac{kn}{2}$$

$$\Pr[\overline{E_1}] = 1 - \frac{k}{m} \geq 1 - \frac{k}{kn/2} = \left(1 - \frac{2}{n}\right)$$

Proof of theorem

Let E_i = an edge in *F* is contracted in iteration i.

Goal: $\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \geq 1/n^2.$

$$\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}]$$
$$\geq \left(1 - \frac{2}{n}\right) \cdot \Pr[\overline{E_2} | \overline{E_1}] \cdot \Pr[\overline{E_3} | \overline{E_1} \cap \overline{E_2}] \cdots$$
$$\Pr[\overline{E_{n-2}} | \overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-3}}]$$

$$\Pr[\overline{E_2} | \overline{E_1}] = 1 - \Pr[E_2 | \overline{E_1}] = 1 - \frac{k}{\boxed{\#\text{ remaining edges}}}$$

want to write in terms of *k* and *n*

Proof of theorem

Let E_i = an edge in *F* is contracted in iteration i.

Goal: $\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \geq 1/n^2$.

Let $G' = (V', E')$ be the graph after iteration 1.

Observation: $\forall v \in V' \; : \; k \leq \deg_{G'}(v)$

$$\underbrace{\sum_{v \in V'} \deg_{G'}(v)}_{\geq k(n-1)} = 2|E'| \qquad \Longrightarrow \; 2|E'| \geq k(n-1)$$

$$\Longrightarrow \; |E'| \geq \frac{k(n-1)}{2}$$

$$\Pr[\overline{E_2}|\overline{E_1}] = 1 - \frac{k}{|E'|} \geq 1 - \frac{k}{k(n-1)/2} = \left(1 - \frac{2}{n-1}\right)$$

Proof of theorem

Let E_i = an edge in *F* is contracted in iteration i.

Goal: $\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \geq 1/n^2.$

$$\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}]$$

$$\geq \left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \Pr[\overline{E_3} | \overline{E_1} \cap \overline{E_2}] \cdots$$

$$\Pr[\overline{E_{n-2}} | \overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-3}}]$$

Proof of theorem

Let E_i = an edge in *F* is contracted in iteration i.

Goal: $\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \geq 1/n^2.$

$$\Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}]$$

$$\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\left(1 - \frac{2}{n-2}\right)\cdots\left(1 - \frac{2}{n-(n-4)}\right)\left(1 - \frac{2}{n-(n-3)}\right)$$

$$= \left(\frac{\cancel{n-2}}{n}\right)\left(\frac{\cancel{n-3}}{n-1}\right)\left(\frac{\cancel{n-4}}{\cancel{n-2}}\right)\left(\frac{\cancel{n-5}}{\cancel{n-3}}\right)\cdots\left(\frac{2}{\cancel{4}}\right)\left(\frac{1}{\cancel{3}}\right)$$

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

$\square$

Contraction algorithm for min cut

Theorem:

Let $G = (V, E)$ be a graph with *n* vertices.
The probability that the contraction algorithm will output a min-cut is $\geq 1/n^2$.

Should we be impressed?

- The algorithm runs in polynomial time.
- There are exponentially many cuts. ($\approx 2^n$)
- There is a way to boost the probability of success to $1 - \frac{1}{e^n}$ (and still remain in polynomial time)

Contraction algorithm for min cut

Theorem:

Let $G = (V, E)$ be a graph with *n* vertices.
The probability that the contraction algorithm will output a min-cut is $\geq 1/n^2$.

Should we be impressed?

- The algorithm runs in polynomial time.

- There are exponentially many cuts. (≈ 2^n)

- There is a way to boost the probability of success to
$1 - \dfrac{1}{e^n}$ (and still remain in polynomial time)

Boosting phase

Run the algorithm ***t*** times using fresh random bits.

Output the smallest cut among the ones you find.

$$
\begin{tikzcd}
G \arrow[d, "\text{Contraction Algorithm}"] & G \arrow[d, "\text{Contraction Algorithm}"] & G \arrow[d, "\text{Contraction Algorithm}"] & \cdots & G \arrow[d, "\text{Contraction Algorithm}"] \\
F_1 & F_2 & F_3 & \cdots & F_t
\end{tikzcd}
$$

Output the minimum among F_i's.

larger $t \implies$ better success probability

What is the relation between t and success probability?

Boosting phase

What is the relation between t and success probability?

Let A_i = in the i'th repetition, we **don't** find a min cut.

$$\Pr[\text{error}] = \Pr[\text{don't find a min cut}]$$

$$= \Pr[A_1 \cap A_2 \cap \cdots \cap A_t]$$

ind. events

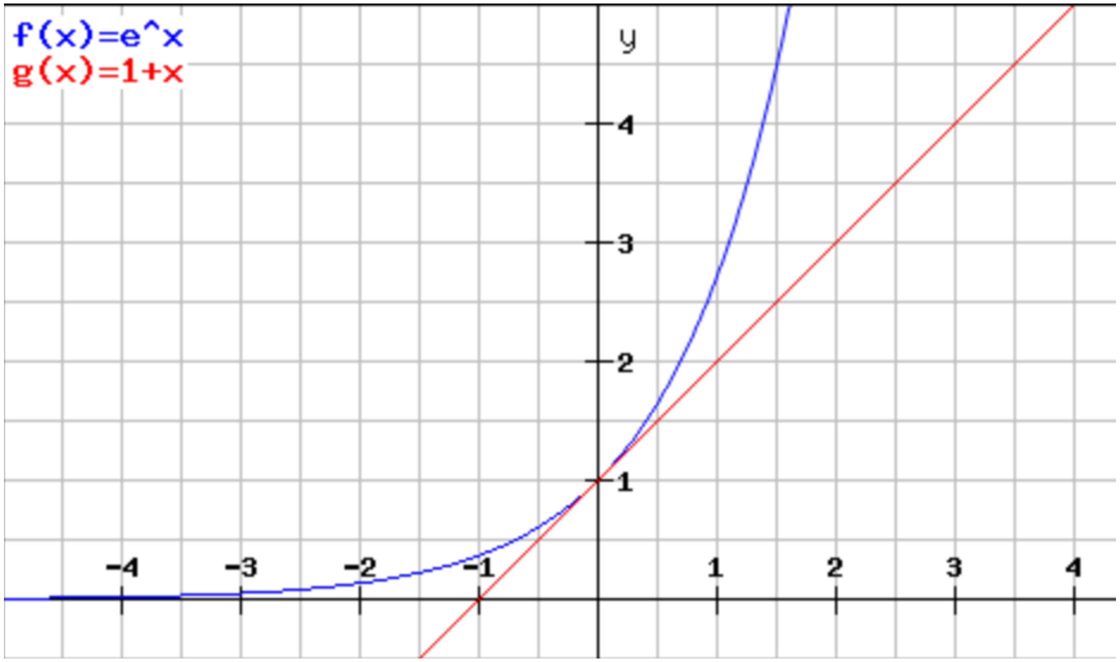
$$= \Pr[A_1] \Pr[A_2] \cdots \Pr[A_t]$$

$$= \Pr[A_1]^t \leq \left(1 - \frac{1}{n^2}\right)^t$$

Boosting phase

$$\Pr[\text{error}] \leq \left(1 - \frac{1}{n^2}\right)^t$$

Extremely useful inequality: $\forall x \in \mathbb{R} : \; 1 + x \leq e^x$



Boosting phase

$$\Pr[\text{error}] \leq \left(1 - \frac{1}{n^2}\right)^t$$

Extremely useful inequality: $\forall x \in \mathbb{R} : \ 1 + x \leq e^x$

Let $x = -1/n^2$

$$\Pr[\text{error}] \leq (1+x)^t \leq (e^x)^t \quad = e^{xt} \quad = e^{-t/n^2}$$

$$t = n^3 \implies \Pr[\text{error}] \leq e^{-n^3/n^2} = 1/e^n \implies$$

$$\Pr[\text{success}] \geq 1 - \frac{1}{e^n}$$

Conclusion for min cut

We have a polynomial time algorithm that solves the min cut problem with probability $1 - 1/e^n$.

Theoretically, not equal to 1.
Practically, equal to 1.

Important Note

Boosting is not specific to Min-cut algorithm.

We can boost the success probability of
Monte Carlo algorithms via repeated trials.

Example of a Las Vegas Algorithm (QuickSort)

Always correct.
Gambles with running time.

QuickSort

4	8	2	7	99	5	0

On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S

QuickSort

4	8	2	7	99	5	0

On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m

QuickSort

8	2	7	99	5	0

$\boxed{4}$

On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m

QuickSort

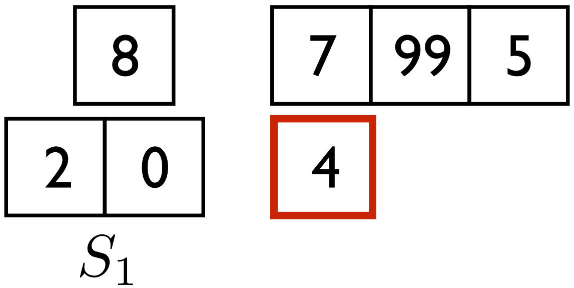
8	2	7	99	5	0

$\boxed{4}$

On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m
- Compare x_m to all other x's
- Let $S_1 = \{x_i : x_i < x_m\}$, $S_2 = \{x_i : x_i > x_m\}$

QuickSort



On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m
- Compare x_m to all other x's
- Let $S_1 = \{x_i : x_i < x_m\}$, $S_2 = \{x_i : x_i > x_m\}$

QuickSort

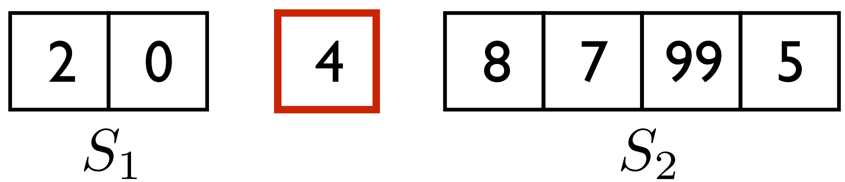
2	0		$\boxed{4}$		8	7	99	5

S_1 $\qquad$ S_2

On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m
- Compare x_m to all other x's
- Let $S_1 = \{x_i : x_i < x_m\}$, $S_2 = \{x_i : x_i > x_m\}$

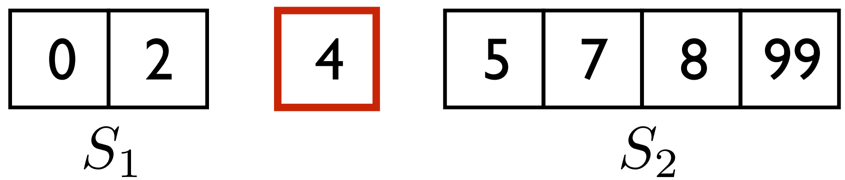
QuickSort



On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m
- Compare x_m to all other x's
- Let $S_1 = \{x_i : x_i < x_m\}$, $S_2 = \{x_i : x_i > x_m\}$
- Recursively sort S_1 and S_2 .

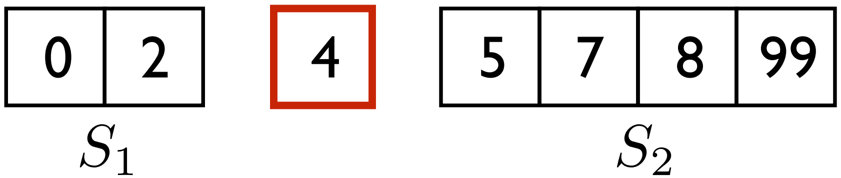
QuickSort



On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m
- Compare x_m to all other x's
- Let $S_1 = \{x_i : x_i < x_m\}$, $S_2 = \{x_i : x_i > x_m\}$
- Recursively sort S_1 and S_2.

QuickSort



On input $S = (x_1, x_2, \ldots, x_n)$

- If $n \leq 1$, return S
- Pick uniformly at random a "pivot" x_m
- Compare x_m to all other x's
- Let $S_1 = \{x_i : x_i < x_m\}$, $S_2 = \{x_i : x_i > x_m\}$
- Recursively sort S_1 and S_2 .
- Return $[S_1, x_m, S_2]$

QuickSort

This is a Las Vegas algorithm:

- always gives the correct answer
- running time can vary depending on our luck

It is not too difficult to show that the expected run-time is

$$\leq 2n \ln n = O(n \log n).$$

In practice, it is basically the fastest sorting algorithm!

Final Remarks

Randomness adds an interesting dimension to computation.

Randomized algorithms can be faster and much more elegant than their deterministic counterparts.

There are some interesting problems for which:
- there is a poly-time randomized algorithm,
- we can't find a poly-time deterministic algorithm.

Another (morally) million dollar question:

Does every efficient randomized algorithm have an efficient deterministic counterpart?

Is $\mathsf{P} = \mathsf{BPP}$?