Randomized Algorithms

Textbook:

Randomized Algorithms, by Rajeev Motwani and Prabhakar Raghavan.

Randomized Algorithms

- **A Randomized Algorithm** uses a random number generator.
  - its behavior is determined not only by its input but also by the values chosen by RNG.
  - It is impossible to predict the output of the algorithm.
  - Two executions can produce different outputs.

Why Randomized Algorithms?

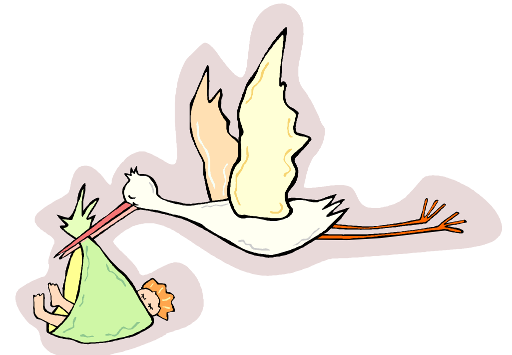
- Efficiency
- Simplicity
- Reduction of the impact of bad cases!
- Fighting an adversary.

Types of Randomized Algorithms

- **Las Vegas** algorithms
  - Answers are always correct, running time is random
  - In analysis: bound *expected* running time
- **Monte Carlo** algorithms
  - Running time is fixed, answers may be incorrect
  - In analysis: bound error probabilities

Randomized Algorithms

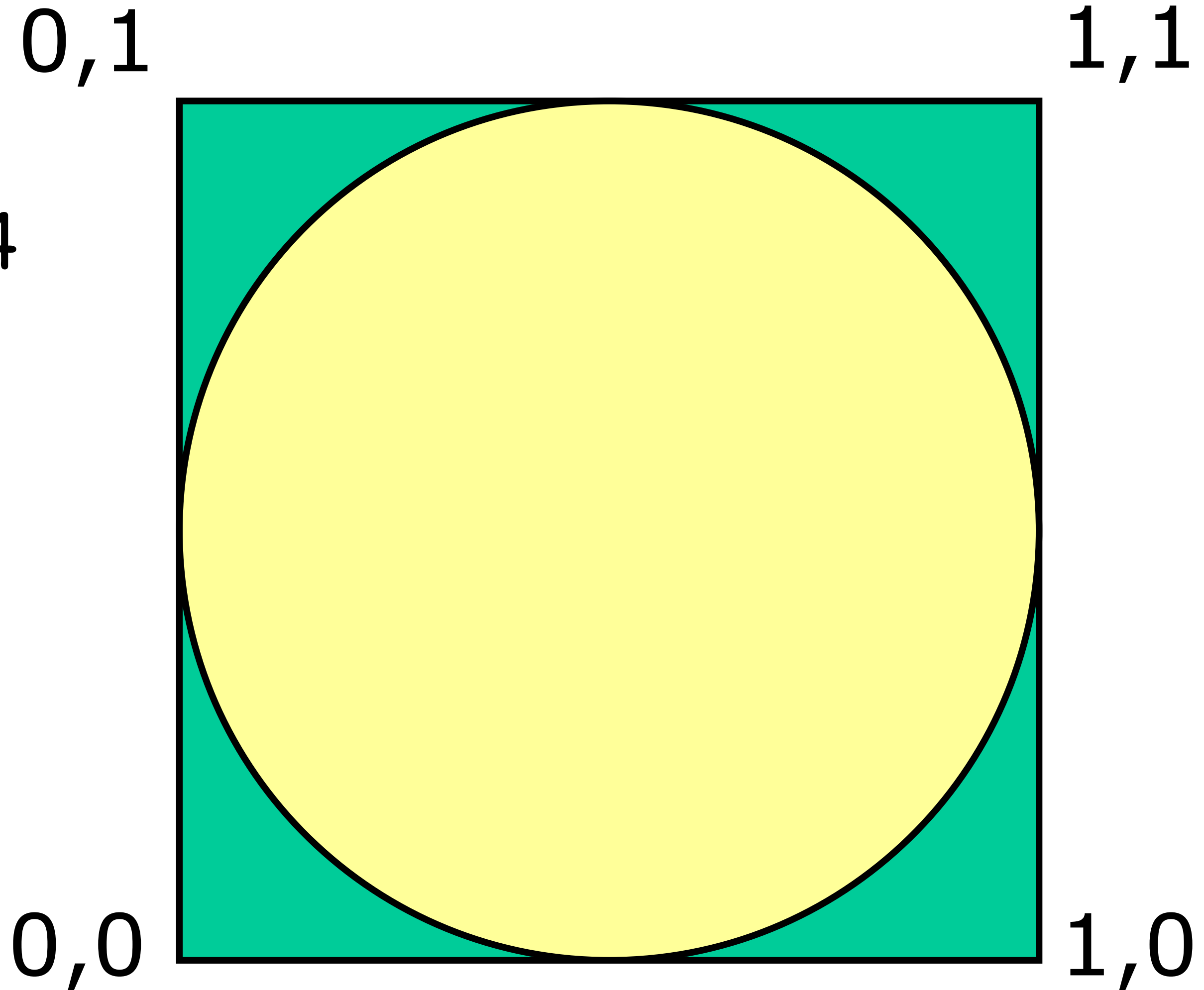
- Where do random numbers come from?
  - Sources of Entropy: physical phenomena, user's mouse movements, keystrokes, atmospheric noise, lava lamps.
- Pseudo-random generators: take a few "good" random bits and generate a lot of "fake" random bits.
  - Most often used in practice
  - Output of pseudorandom generator should be "indistinguishable" from true random



We will see (up to random decisions):

1. A randomized approximation Algorithm for determining the value of Π.
2. A Randomized algorithm for the selection problem.
3. A randomized data structure.
4. Analysis of random walk on a graph.
5. A randomized graph algorithm.

Determining π



Square area = 1
Circle area = $\pi/4$

The probability that a random point in the square is in the circle = $\pi/4$

π = 4 * points in circle/points

Determining π

```
def findPi (points):
    incircle = 0

    for i in 1 to points:
        x = random() // float in [0,1]
        y = random()
        if (x - ½)² + (y - ½)² < 0.25)
                incircle = incircle + 1

    return 4.0 * incircle / points
```

Note : a point is in the circle if its distance from (½, ½) < r

Determining π - Results

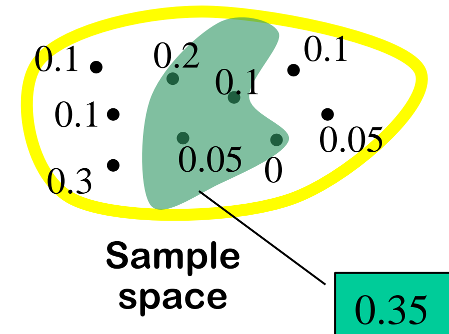
n	Output π
1	0.0
2	4.0
4	3.0
64	3.0625
1024	3.1640625
16384	3.1279296
131072	3.1376647
1048576	3.1411247

Real π = 3.14159265

If we wait long enough will it produce an arbitrarily accurate value?

Basic Probability Theory (a short recap)

- Sample Space Ω:
  - Set of possible outcome points
- Event $A \subseteq \Omega$:
  - A subset of outcomes
- $\Pr[A]$: probability of an event
  - For every event A: $\Pr[A] \in [0,1]$
  - If $A \cap B = \emptyset$ then $\Pr[A \cup B] = \Pr[A] + \Pr[B]$
  - $\Pr[\Omega] = 1$



Basic Probability Theory (a short recap)

- Random Variable X:
  - Function from sample space to $\mathbb{R}$
- Example:
  - $\Omega=\{v=(g_1,g_2,\ldots g_n) \mid g_i \in [0,100]\}$
  - Events:
    - $A=\{v=(g_1,g_2,\ldots g_n) \mid \forall i: g_i \in [60,100]\}$
    - $B=\{v=(g_1,g_2,\ldots g_n) \mid \exists i,j,k : g_i,g_j,g_k \in [60,100]\}$
  - Random variables
    - X_i – 1 if student i passed, 0 if not
    - $X=X_1+\ldots+X_n$ – number of passing students
    - Y – Average grade

Possible grades for entire class

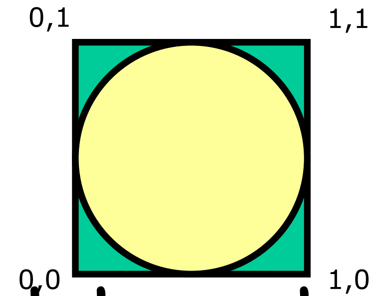
Everyone passed

At least three passed

Basic Probability Theory (a short recap)

- Expectation of a Random Variable X:
  - $E[X]=\sum_x x \cdot Pr[X=x]$ - the "average value"
- Expectation of Indicator Variables:
  - $E[X_i]=Pr[X_i=1]$ - for an indicator r.v.
- Linearity of Expectation:
  - $E[a_1X_1+\ldots+a_nX_n]=a_1E[X_1]+\ldots+a_nE[X_n]$
  - Works for arbitrary random variables!

Back to π



- How many points must we test to get accuracy of 0.01? How about 0.0001?

- X_i=1 iff the ith point is in the circle
  - $E[X_i]=Pr[X_i=1]=\pi/4$
  - $E[X_1+...+X_n]=E[X_1]+...+E[X_n]=n\pi/4$
- Accuracy: distance of X from E[X]

- We want to bound $Pr[|X-E[X]|>\delta]$

A Simple Calculation

True or False:

If the average income of people is $100 then more than 50% of the people can be earning more than $200 each

False! else the average would be higher!!!

Markov's Inequality

- If X is a non-negative r.v. with mean E[X] then
  - $\Pr[\, X > 2E[X] \,] \leq \frac{1}{2}$
- More generally, for all a>0:
  - $\Pr[\, X > a \,] \leq E[X]/a$

Andrei A. Markov

Markov's Inequality

Non-neg random variable X has expectation E[X]

$$\mathrm{E[X]} = \sum_x x \cdot \Pr(X = x)$$

$$\geq \sum_{x \geq a} x \cdot \Pr(X = x) \qquad \text{(since X,a non-neg)}$$

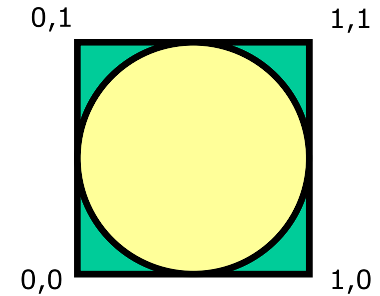
$$\geq \sum_{x \geq a} a \cdot \Pr(X = x)$$

$$= a \sum_{x \geq a} \Pr(X = x)$$

$$= a \Pr(X \geq a)$$

$$\Rightarrow \quad \Pr(X \geq a) \leq \mathrm{E[X]}/a$$

Back to π again



- X is the number of points in the circle
  - X is non-negative
  - E[X]=nπ/4
- What does Markov's inequality give us?
  - $\Pr[X - E[X] > \delta] = \Pr\left[X > \delta + E[X]\right] \leq \frac{E[X]}{E[X]+\delta}$
  - However, $\frac{E[X]}{E[X]+\delta} \rightarrow 1$ when $n \rightarrow \infty$
- Not good enough!

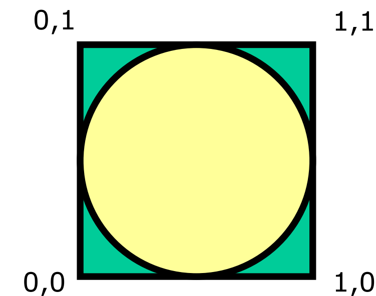
Chernoff-Hoeffding Bound

- Used to bound convergence of sums of random variables: $X=X_1+\ldots+X_n$
  - X_i are zero/one variables
  - All variables are independent
    - For all i,j: $\Pr[X_i=1|X_j=1]=\Pr[X_i=1]$
  - Denote $\mu = E[X]/n$
- When satisfied, sum converges *exponentially fast*!

$$\Pr\left[\left|\frac{X}{n}-\mu\right| \geq \delta\mu\right] \leq 2e^{-\frac{n\delta^2\mu}{3}}$$

Using Chernoff to estimate accuracy

- Recall:
  - X_i=1 iff the ith point is in the circle
  - $E[X_i]=Pr[X_i=1]=\pi/4$
  - $E[X_1+\ldots+X_n]=E[X_1]+\ldots+E[X_n]=n\pi/4$
  - $\mu = \pi/4$



Using Chernoff to estimate accuracy

- Using Chernoff:

$$\Pr\left[\left|\frac{X}{n} - \mu\right| \geq \delta\mu\right] \leq 2e^{-\frac{n\delta^2\mu}{3}} \Leftrightarrow \Pr\left[\left|\frac{X}{n} - \mu\right| \geq \delta\right] \leq 2e^{-\frac{n\delta^2}{3\mu}}$$

- To get close within δ with error prob. ϵ:

$$2e^{-\frac{n\delta^2}{3\mu}} < \epsilon \Leftrightarrow n > \frac{3\mu \ln 2/\epsilon}{\delta^2} = \frac{3\pi \ln 2/\epsilon}{4\delta^2}$$

- E.g.: to get 0.01 accuracy with 95% success:

$$\epsilon = 0.05, \delta = 0.01 \Rightarrow n > \frac{3\pi \ln 40}{4 \cdot 0.0001} \approx 86917$$

The Selection Problem

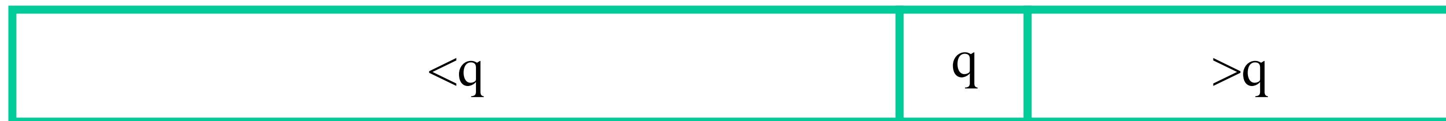
- **Input:** Array $A[1...n]$ of elements in an arbitrary order, and an index i.
- **Output:** the i-th smallest element in $A[1..n]$.
- If $i = 1$, we are looking for the smallest element.
- If $i = n$, we are looking for the largest element.
- If $i = n/2$, we are looking for the median.

- **Possible algorithm:** Sort and return the i-th element.
  Complexity: At least $\Omega(n \log n)$
- We will see a practical randomized algorithm with $O(n)$ expected running time

```
In class example:  i=4
A = {7, 3, 2, 1, 5, 9, 8};
```

Randomized Selection

- Key idea: similar to quicksort
- Choose a random element q
- partition the array into small (<q) and big (>q) elements.
- Keep searching in the appropriate side.



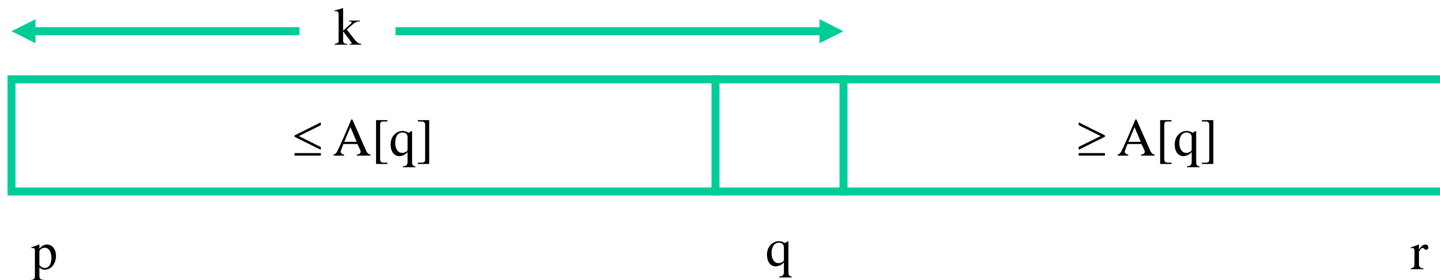
Randomized Selection

Select(A,i)

- If array A is of size 1 – trivial
- Randomly partition A with pivot q.
- Suppose q was the k'th smallest element
- If i=k, answer is q
- If i<k, recursively select i'th element in left subarray
- If i>k, recursively find (i-k)'th element in right subarray

Randomized Selection

```
RandSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandSelect(A,p,q-1,i);
    else
        return RandSelect(A,q+1,r,i-k);
```



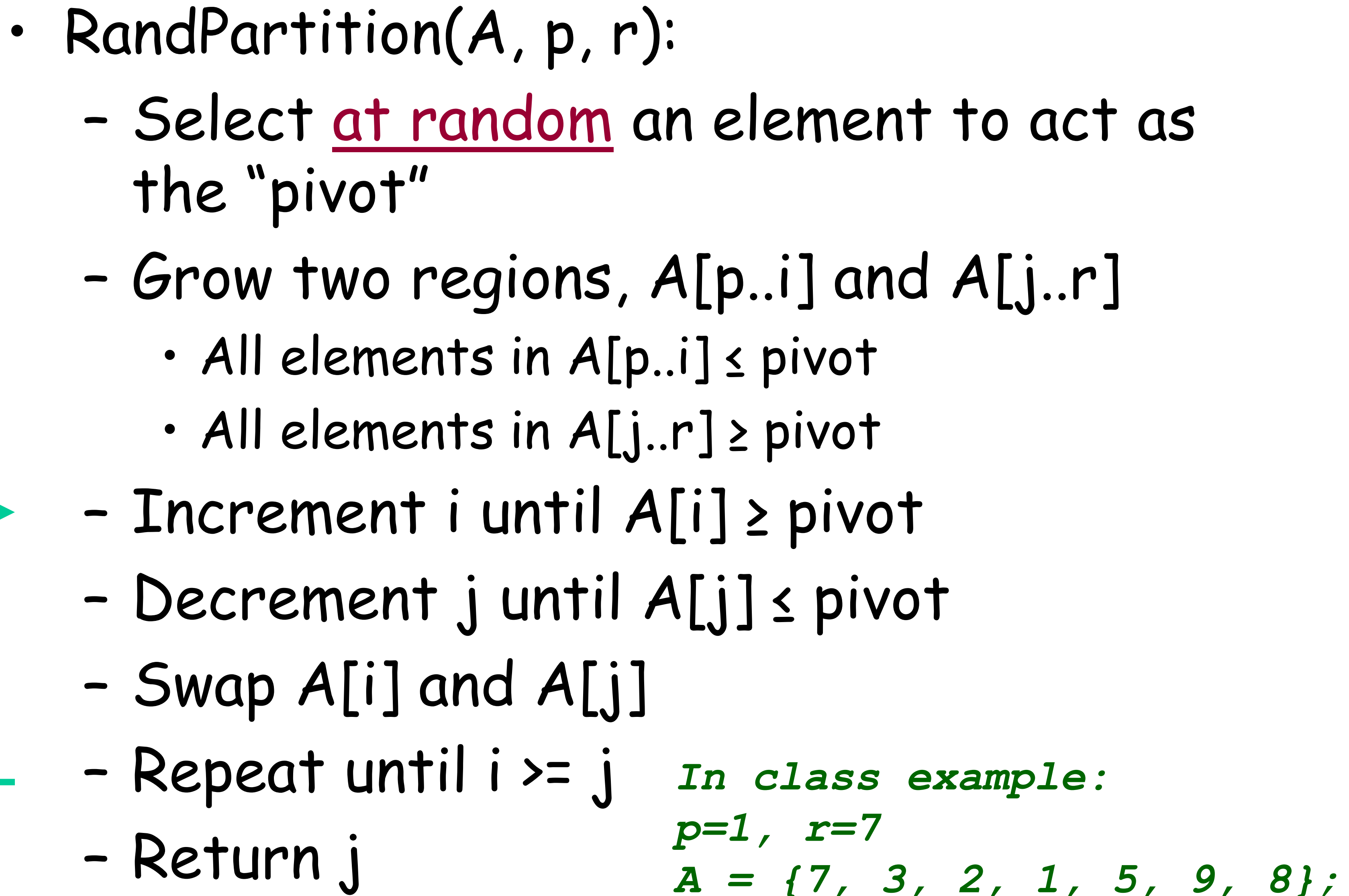
How to partition?

We choose a random element q and want to move all elements smaller than q before q. and all elements greater than q after q.

- Keep two indices, one running from left and the other from right.
- Move each index until points to an element that belongs to the other side
- swap the two elements and repeat.
- Stop when indices cross each other.

Running time? O(n)

RandPartition In Words



Partition Code

```
RandPartition(A, p, r)
    x = A[random integer in p..r];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

What is the running time of ***RandPartition()?***

Randomized Selection

- Analysis:
  - Worst case: partition always 0:n-1

    $T(n) = T(n-1) + O(n)$

    $= O(n^2)$ (arithmetic series)

    - Worse than sorting!
  - "Best" case: suppose an $\alpha n : (1-\alpha)n$ partition for some fraction α.

    $T(n) = T(\alpha n) + O(n)$

    $= O(n)$ (Master Theorem, case 3)

    - Better than sorting!

$a=1$, $b=1/\alpha$
$f(n)=\Omega(n^{\log_b(a)+\varepsilon})$

Randomized Selection

- Average case
  - For upper bound, assume *i*-th element always falls in larger side of partition:

$$E[T(n)] \leq \frac{1}{n}\sum_{k=0}^{n-1} E[T(\max(k, n-k-1))] + \Theta(n)$$

Pr[pivot is kth element]

$$\leq \frac{2}{n}\sum_{k=n/2}^{n-1} E[T(k)] + \Theta(n)$$

  - We show that $E[T(n)] = O(n)$ by induction

Randomized Selection

- Assume E[T(k)] ≤ ck for some const. c *(and k<n)*:

$$E[T(n)] \le \frac{2}{n}\sum_{k=\frac{n}{2}}^{n-1} E[T(k)] + \theta(n)$$

The recurrence we start with

$$\le \frac{2}{n}\sum_{k=\frac{n}{2}}^{n-1} ck + \theta(n)$$

Inductive assumption

$$\le c\frac{2}{n}\cdot\frac{1}{2}\left(\frac{n}{2}+n\right)\frac{n}{2} + \theta(n)$$

Arithmetic series

$$\le \frac{3}{4}cn + \theta(n)$$

$$\le cn$$

For large enough c

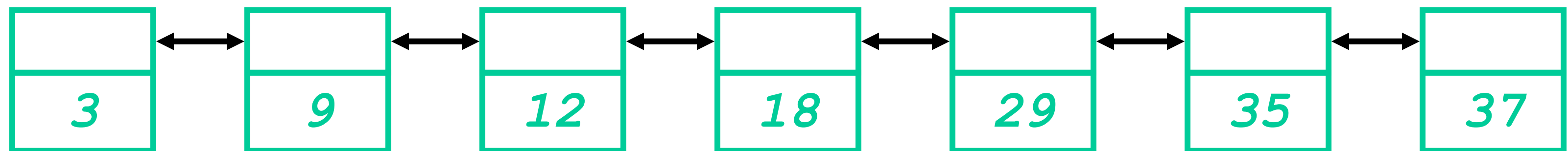
Randomized Selection

- We got a randomized selectio algorithm that runs in expected linear time.
- Is this Monte Carlo or Las vegas?

Skip lists - a randomized data structure

Review:

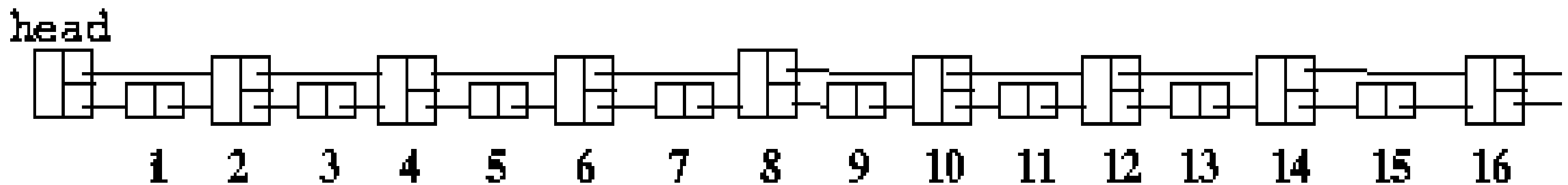
- **Sorted Linked List**
  What is the worst case performance of find( ), insert( )?



- **Sorted Array**
  What is the worst case performance of find( ), insert( )?

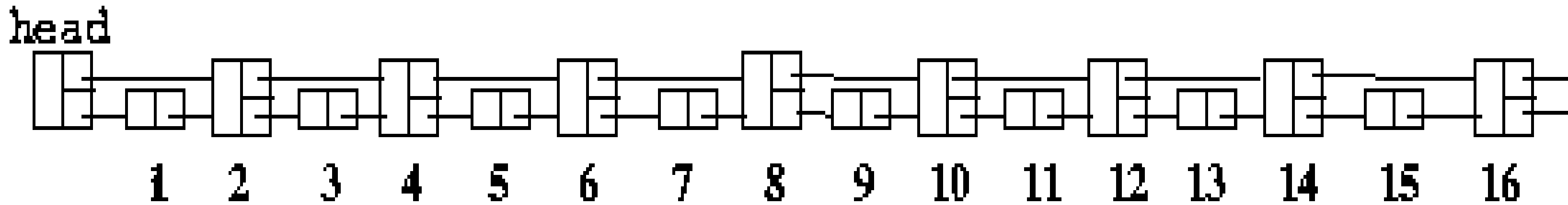
An Alternative Sorted Linked List

- What if you skip every other node?
  - Every other node has a pointer to the next and the one after that



- Find :
  - follow "skip" pointer until pass target
- Resources
  - Additional storage
- Performance of find( )?

Skipping every 2nd node

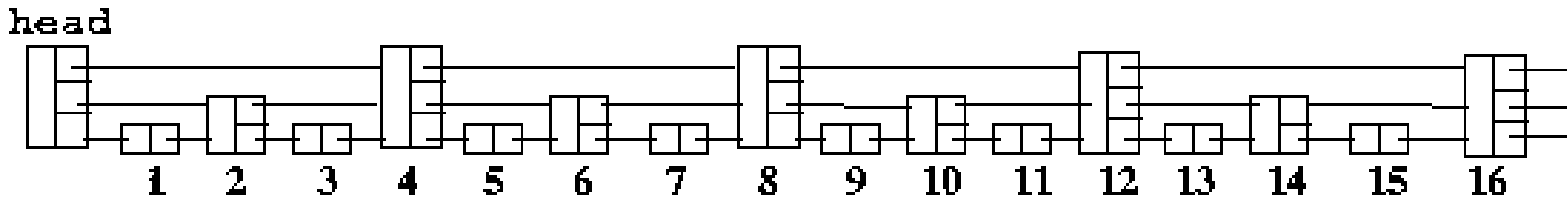


The value stored in each node is shown below the node and corresponds to the the position of the node in the list.

It's clear that find( ) does not need to examine every node. It can skip over every other node, then do a final examination at the end. The number of nodes examined is no more than $\lceil n/2 \rceil + 1$.

For example the nodes examined finding the value 15 would be
2, 4, 6, 8, 10, 12, 14, 16, 15 -- a total of $\lceil 16/2 \rceil + 1 = 9$.

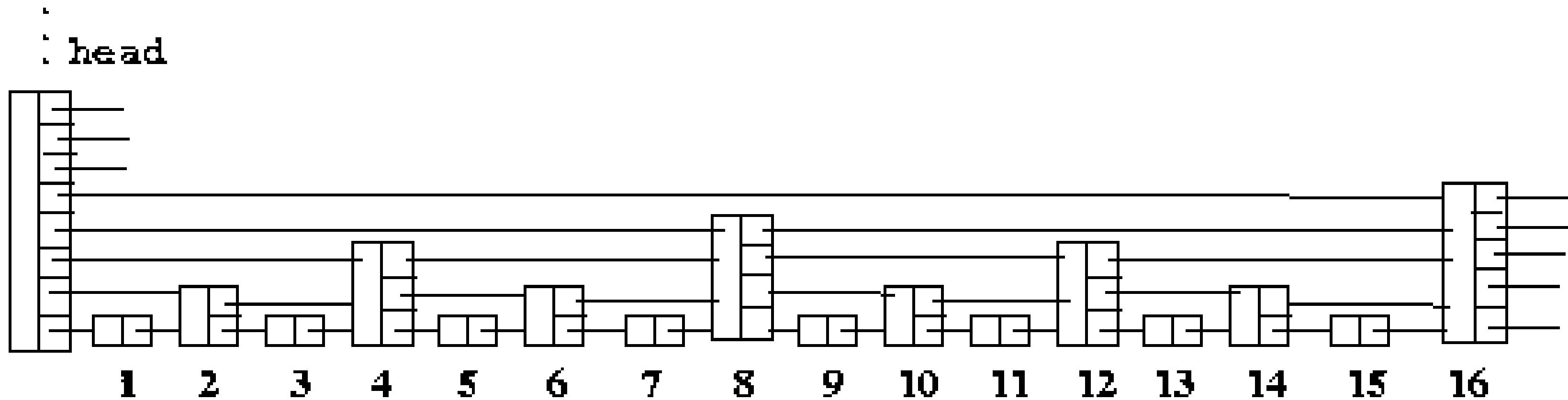
Skipping every 2nd and 4th node



The find operation can now make bigger skips than the previous example. Every 4th node is skipped until the search is confined between two nodes. At this point as many as three nodes may need to be scanned. The number of nodes examined is no more than $\lceil n / 4 \rceil + 3$.

Again, look at the nodes examined when searching for 15

Skipping every 2^i-th node



Add hierarchy of skip pointers

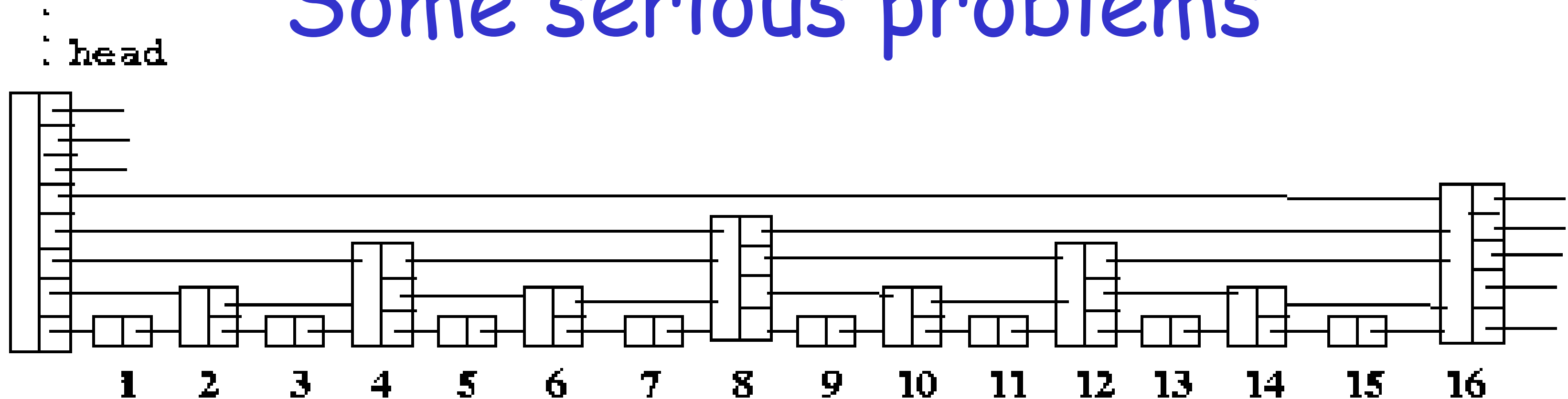
every 2^i-th node points 2^i nodes ahead

For example, every 2nd node has a reference 2 nodes ahead; every 8th node has a reference 8 nodes ahead

We can now search just as binary search in an array.

The max number of nodes examined is O(log n).

Some serious problems

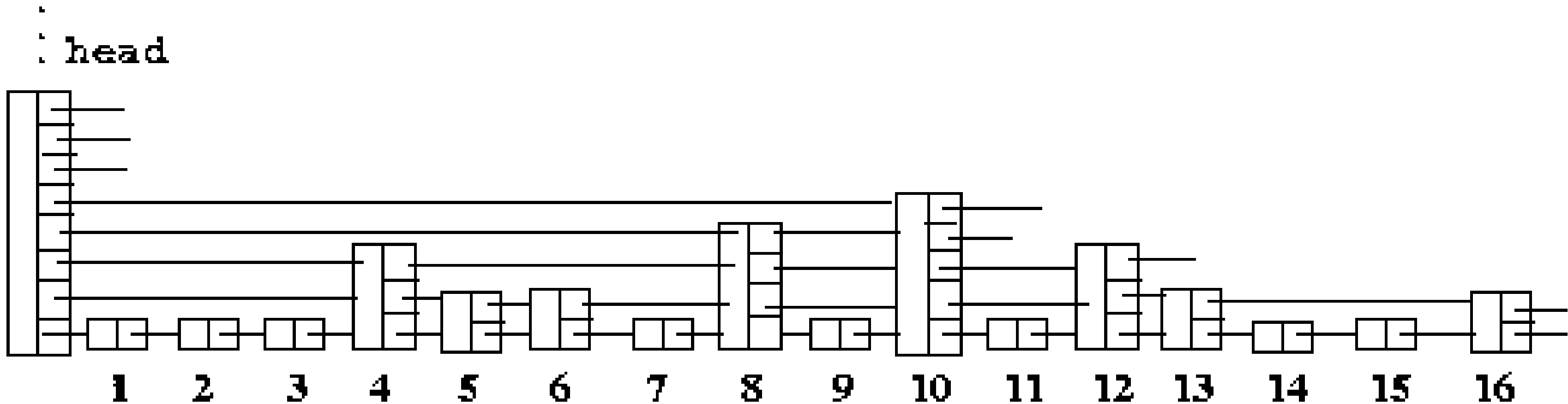


- what happens when we insert or remove a value from the list?
  Reorganizing the list is θ(n).
- E.g., removing the first element in the list elements that were in odd positions (only at lowest level) are now at even positions and vice versa.

A Probabilistic Skip Lists

- Concept: A skip list that maintains the same **distribution** of nodes, but without the requirement for the rigid pattern of node sizes
  - On average, 1/2 have 1 pointer
  - On average, 1/4 have 2 pointers
  - …
  - On average, $1/2^i$ have i pointers
- It's no longer necessary to maintain the rigid pattern by moving values around for insert and remove. This gives us a high probability of still having O(log n) performance. The probability that a skip list will behave badly is very small.

A Probabilistic Skip List



The number of forward reference pointers a node has is its "size"

The distribution of node sizes is exactly the same as the previous figure, the nodes just occur in a different pattern.

Skip List Search

- To search for an element with a given key:
  - Find location in top list
    - Top list has $O(1)$ elements (with high probability).
    - Location in this list defines a range of items in next list
  - Drop down a level and recurse
- $O(1)$ time per level on average
- $O(\log n)$ levels with high probability
- Total *expected* time: $O(\log n)$

Skip List Insert

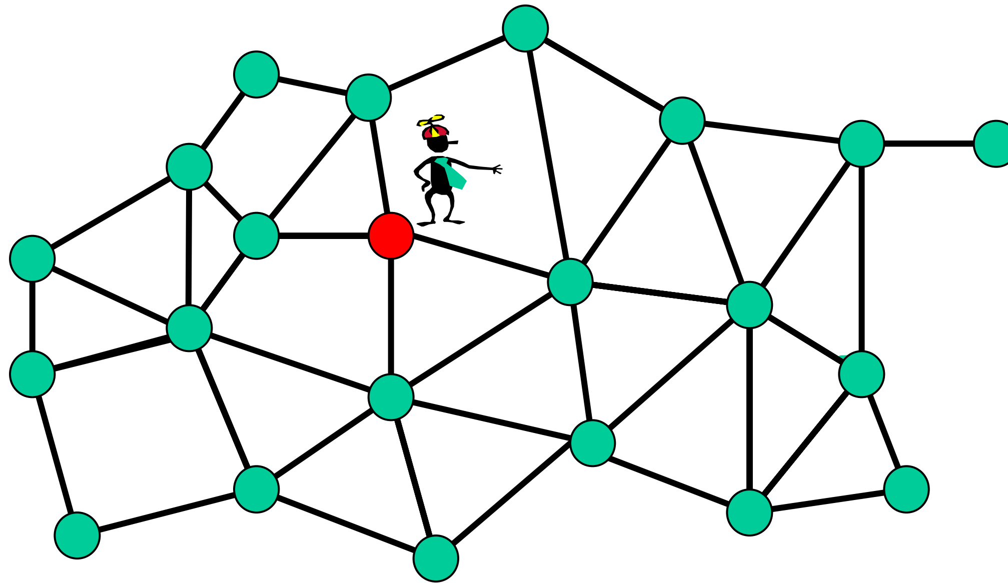
1. Perform a search for that key
2. Insert element in bottom-level list
3. With probability *p*, recurse to insert in upper level (note: p does not have to be 1/2)
   - Expected number of occurances = $1 + p + p^2 + \ldots =$
     $= 1/(1-p) = O(1)$ if *p* is constant
   - Total time = Search + O(1) = $O(\log n)$ expected

Skip list delete: O(1) (assuming a pointer to the element is given)

Probabilistic Skip Lists - summary

- O(log *n*) expected time for find, insert
- O(n) time worst case (*Why?*)
  - But random, and probability of getting worst-case behavior is extremely small
- O(*n*) expected storage requirements (*Why?*)
- Easy to code

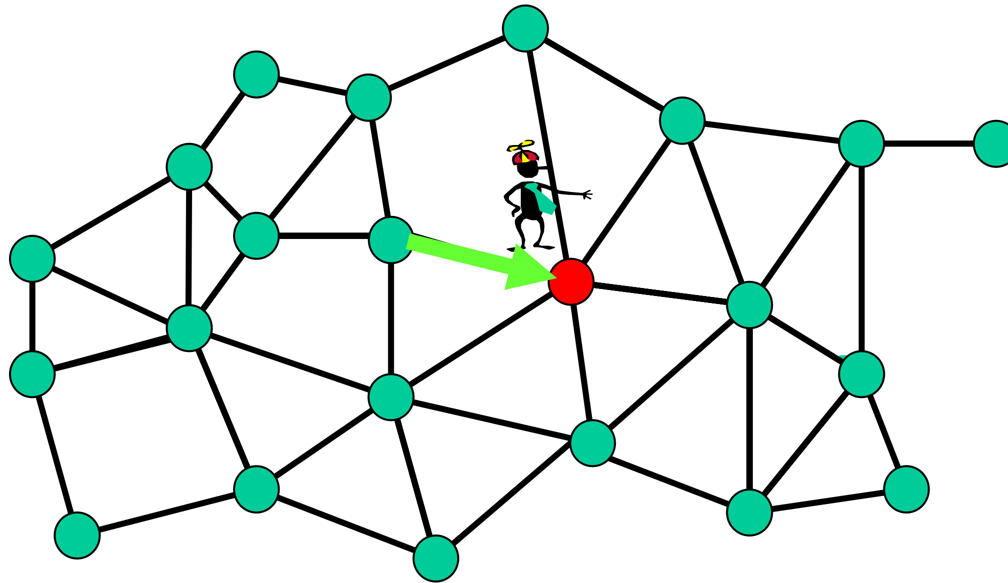
Random Walks on Graphs



At any node, go to one of the neighbors of the node with equal probability

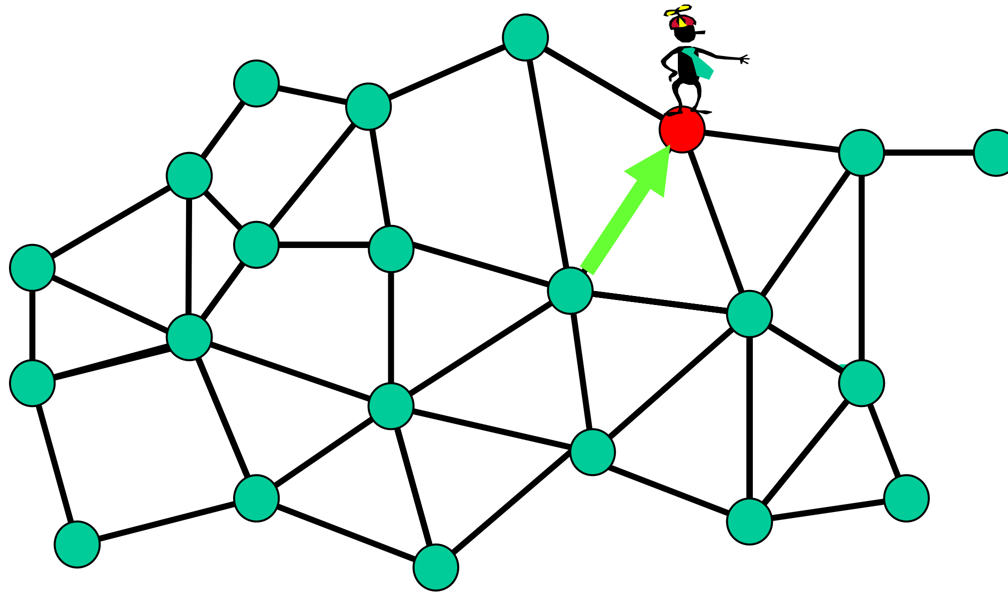
Slides on Random Walks are based on slides by Prof. Steven Rudich (CMU)

Random Walks on Graphs



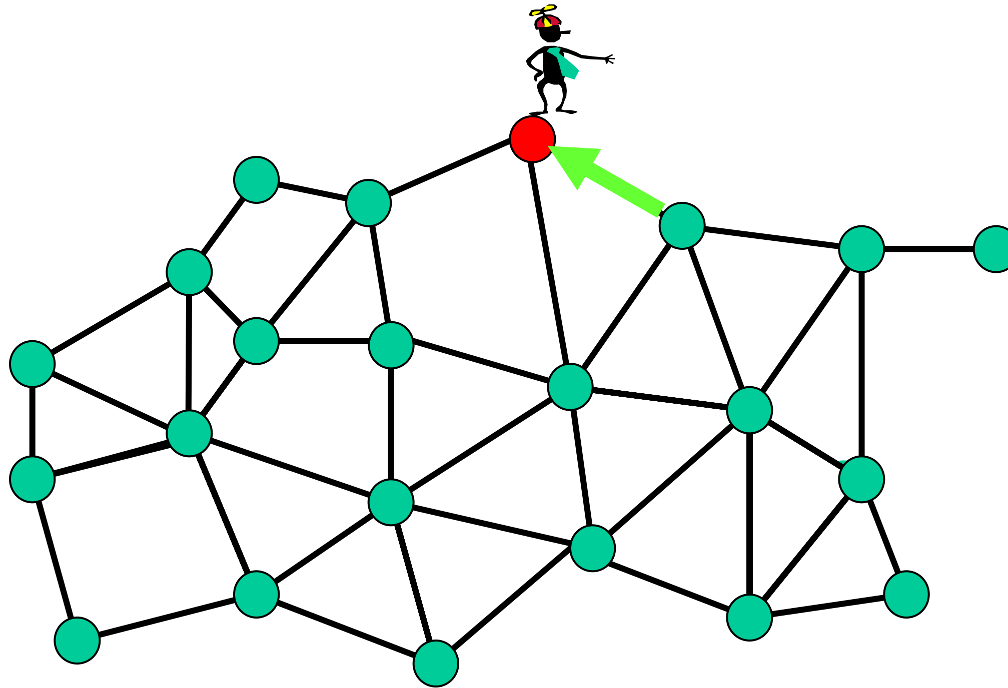
At any node, go to one of the neighbors of the node with equal probability

Random Walks on Graphs



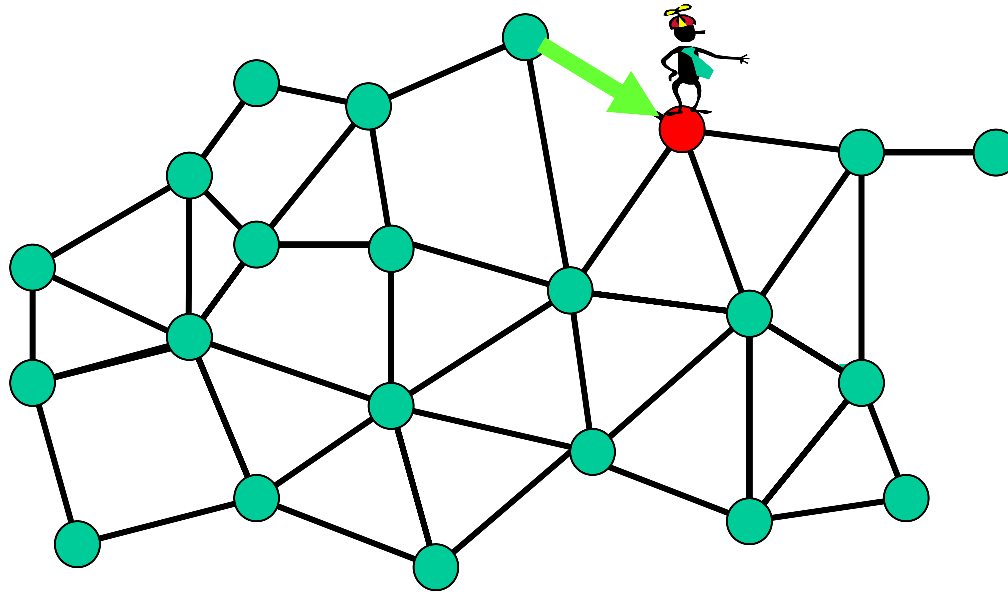
At any node, go to one of the neighbors of the node with equal probability

Random Walks on Graphs



At any node, go to one of the neighbors of the node with equal probability

Random Walks on Graphs

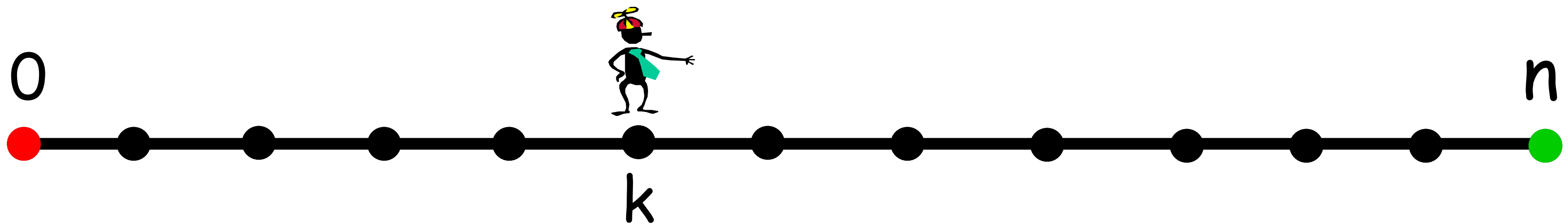


At any node, go to one of the neighbors of the node with equal probability

Random Walk on a Line

You go into a casino with $\$k$, and at each time step, you bet $\$1$ on a fair game

You leave when you are broke or have $\$n$



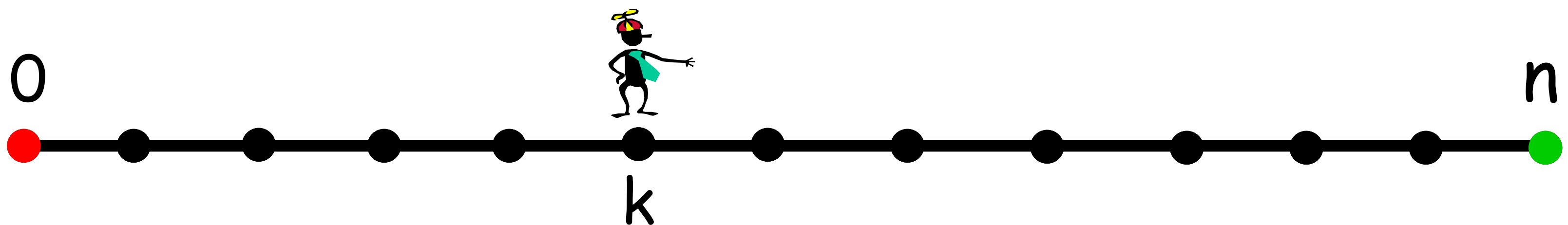
Question 1: what is your expected amount of money at time t?

Let X_t be a R.V. for the amount of $\$$ at time t

Random Walk on a Line

You go into a casino with $k, and at each time step, you bet $1 on a fair game

You leave when you are broke or have $n



$X_t = k + d_1 + d_2 + ... + d_t,$

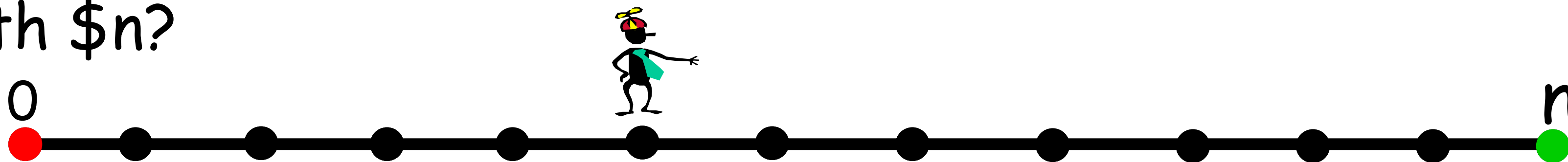
(d_i is RV for change in your money at time i)

$E[d_i] = 0$

So, $E[X_t] = k$

Random Walk on a Line

Question 2: what is the probability that you leave with $n?



$E[X_t] = k$

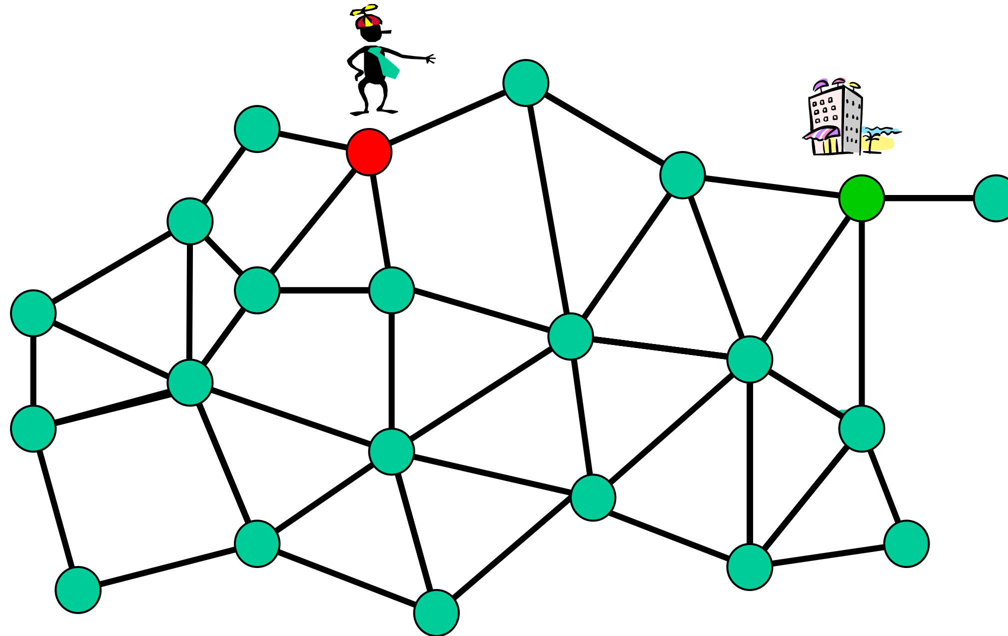
$E[X_t] = E[X_t | X_t = 0] \times Pr(X_t = 0)$
$+ E[X_t | X_t = n] \times Pr(X_t = n)$
$+ E[X_t | \text{neither}] \times Pr(\text{neither})$

probability that you hit green before you hit red

$k = n \times Pr(X_t = n)$
$+ (\text{something}_t) \times Pr(\text{neither})$

As $t \to \infty$, $Pr(\text{neither}) \to 0$, also $\text{something}_t < n$
Hence $Pr(X_t = n) \to k/n$

Getting Back Home



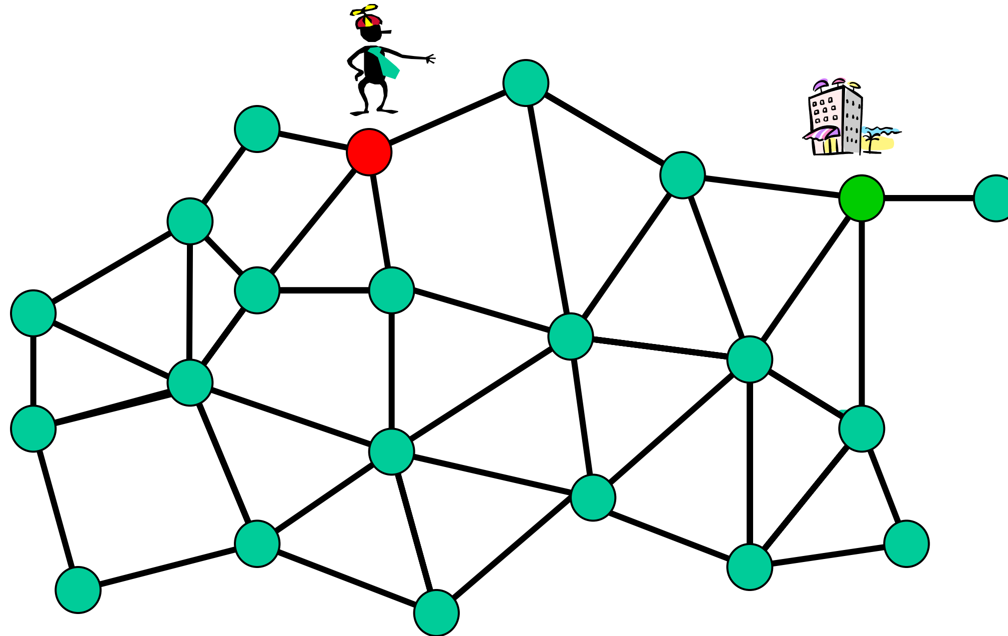
Lost in a city, you want to get back to your hotel

How should you do this?

Depth First Search!

Requires a good memory or a piece of chalk

Getting Back Home



How about walking randomly?

Is Pr[reach home] = 1?

When will I get home?

What is
E[time to reach home]?

We Will Eventually Get Home

Look at the first n steps

There is a non-zero chance p_1 that we get home

Also, $p_1 \geq (1/n)^n$

Suppose we fail

Then, wherever we are, there is a chance $p_2 \geq (1/n)^n$ that we hit home in the next n steps from there

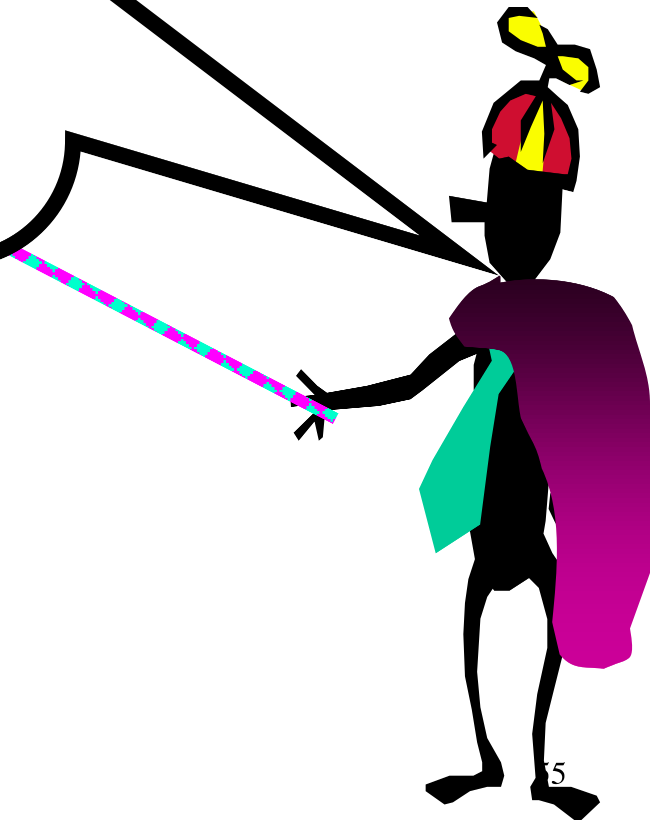
Probability of failing to reach home by time kn

$$= (1 - p_1)(1 - p_2) \ldots (1 - p_k) \to 0 \text{ as } k \to \infty$$

Furthermore:

If the graph has
n nodes and m edges, then

E[time to visit all nodes]
$\leq 2m \times (n-1)$



Cover Times

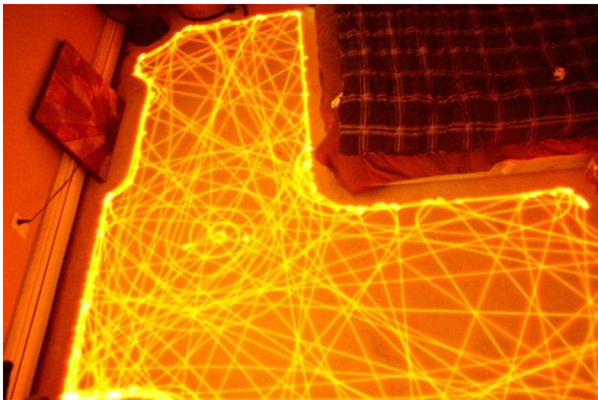
Cover time (from u)

C_u = E [time to visit all vertices | start at u]

Cover time of the graph

$C(G) = \max_u \{ C_u \}$

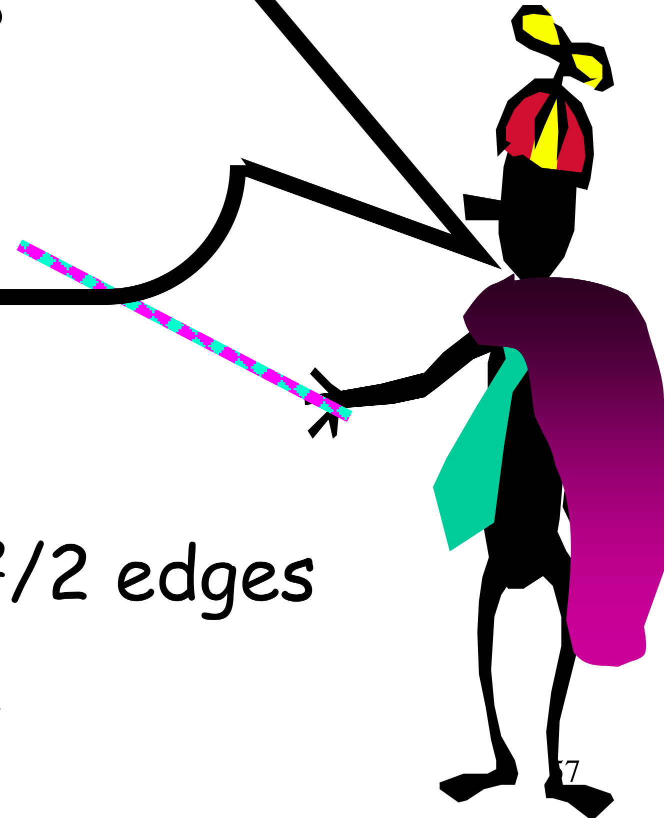
(worst case expected time to see all vertices)



Cover Time Theorem

If the graph G has
n nodes and m edges, then
the cover time of G is

$$C(G) \leq 2m(n-1)$$



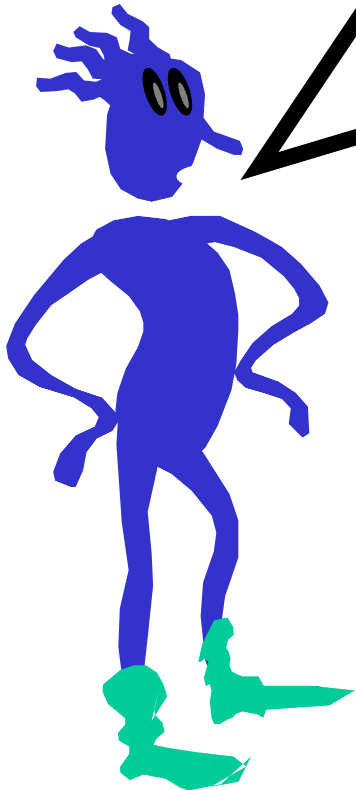
Any graph on n vertices has $< n^2/2$ edges

Hence $C(G) < n^3$ for all graphs G

Actually, we get home pretty fast…

Chance that we don't hit home by $(2k)2m(n-1)$ steps is $(\frac{1}{2})^k$

Why?

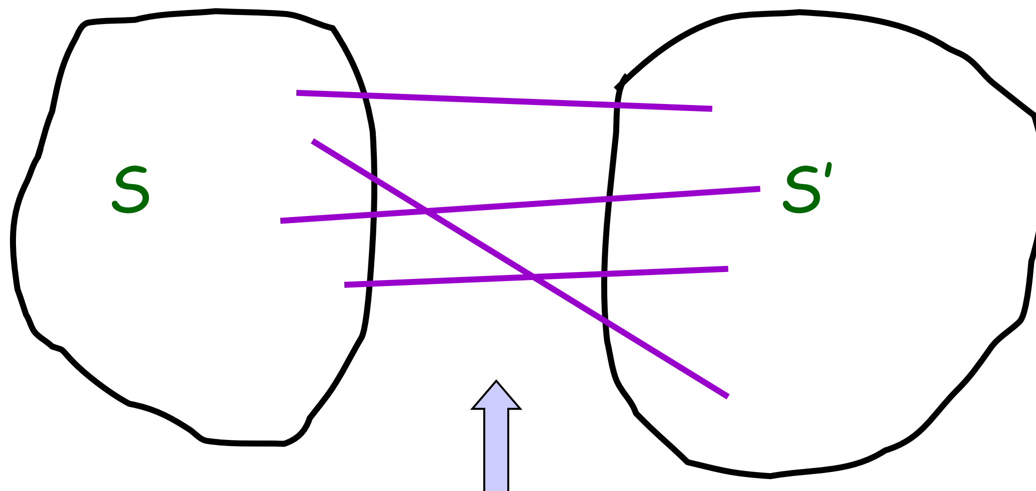


Random Walks are useful

- Efficient sampling from complicated distributions
- Ranking (search results, web pages, friend suggestions, spam detection)
- Clustering
- Design and analysis of random data structures and algorithms
- Applications in physics, economics, biology…
- Much more…..

Cut Problems

- Definition: A Cut S in the graph G(V; E) is a vertex partitioning into two sets: S and S′ such that $S \cup S' = V$.
- The edges in the cut (S;S′) are those connecting a vertex in S and a vertex in S′.



The cut defined by S

Minimum Cut Problem

- Input: an undirected, connected graph $G = (V,E)$
- Output: A cut (V_1, V_2 where $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$) such that number of edges between V_1 and V_2 is the fewest possible.

Equivalent problem: find the fewest edges that can be removed to disconnect G.

A deterministic algorithm: Run max-flow algorithm n^2 times (for all possible s-t pairs).

Resulting complexity: $O(n^3m^2)$

Maximum Cut Problem

- Input: an undirected, connected graph $G = (V,E)$
- Output: A cut (V_1, V_2 where $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$) such that number of edges between V_1 and V_2 is the largest possible.

Equivalent problem: find the largest set of edges that can be removed without disconnecting G.

Min cut is polynomially solvable

Max Cut is NP-hard.

A randomized 2-approx algorithm for max-cut

- Weights may be assigned to the edges (and the goal is to find the heaviest cut).

- Algorithm: Independently for every vertex v, chose randomly with probability ½ whether v belongs to S.

- Analysis: Consider an edge e. The probability that it is in the cut is ½ (why?). So the expected weight of the cut is $\frac{1}{2}\Sigma_e w(e) \geq \frac{1}{2}$ OPTCUT.

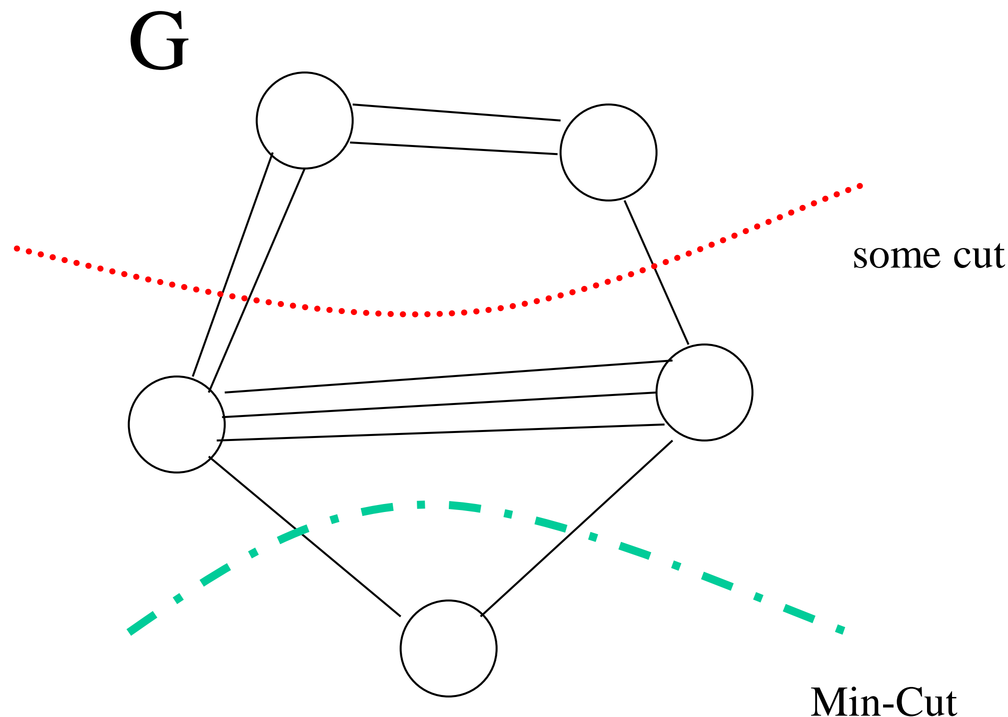
A randomized 2-approx algorithm for max-cut

Remarks:

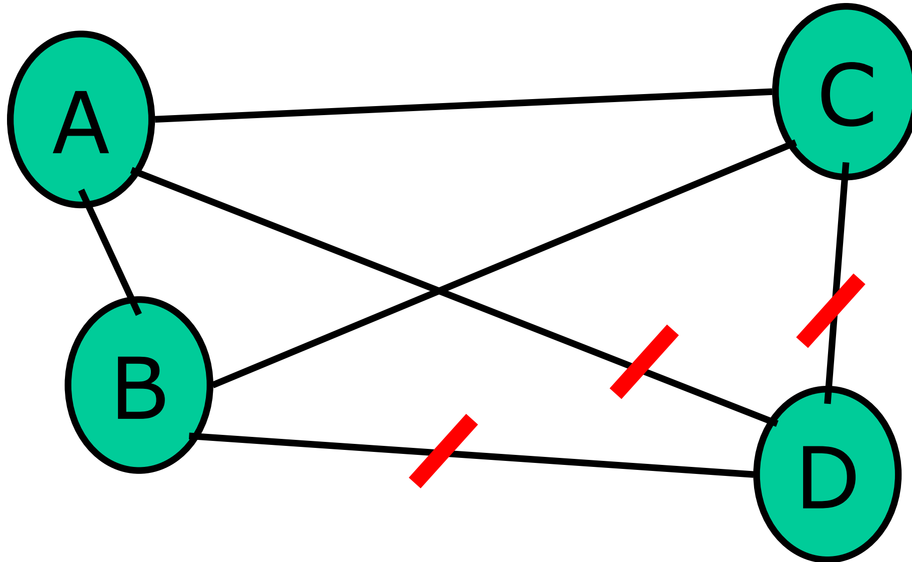
- We can run the algorithm a few times to improve its performance – getting closer to guaranteeing 2-approx.
- Corollary: There exists a cut having weight at least $\frac{1}{2}\Sigma_e w(e)$.

Minimum Cut Problem in a multigraph

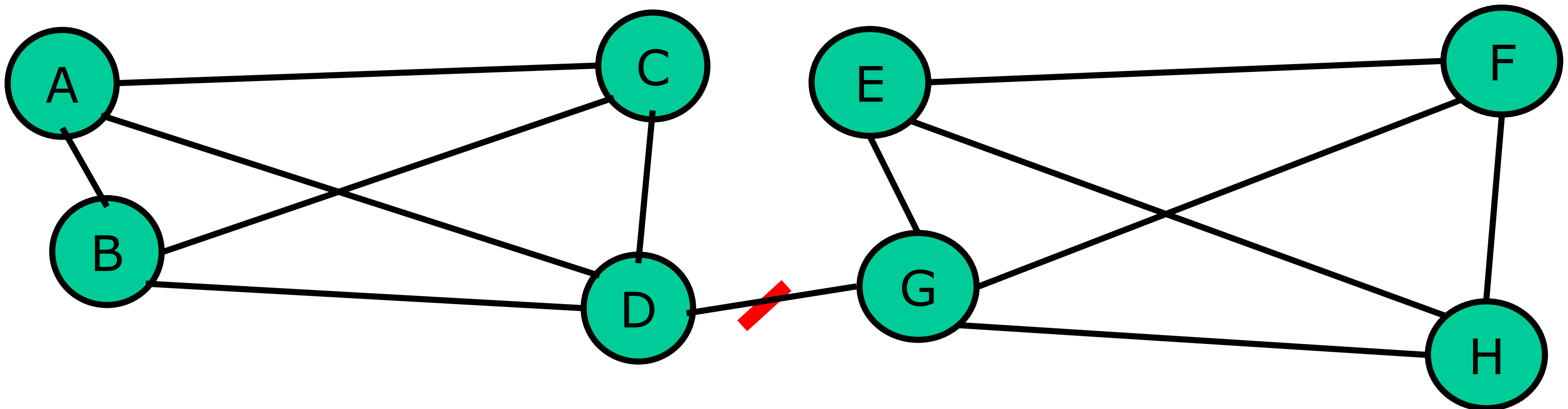
<u>MultiGraph:</u> A graph that may contain multiple edges between any pair of vertices.



Minimum Cut



Size of the min cut is not larger than the smallest node degree in graph



But can be much smaller…

An Algorithm for min-cut

1. While the number of nodes in G is more than 2
   1.1. Choose an edge e not in the min-cut
   1.2. contract e
2. MinCut ← edges connecting the last two remaining vertices.

Contract e = merge the two nodes connected by e

An Algorithm for min-cut

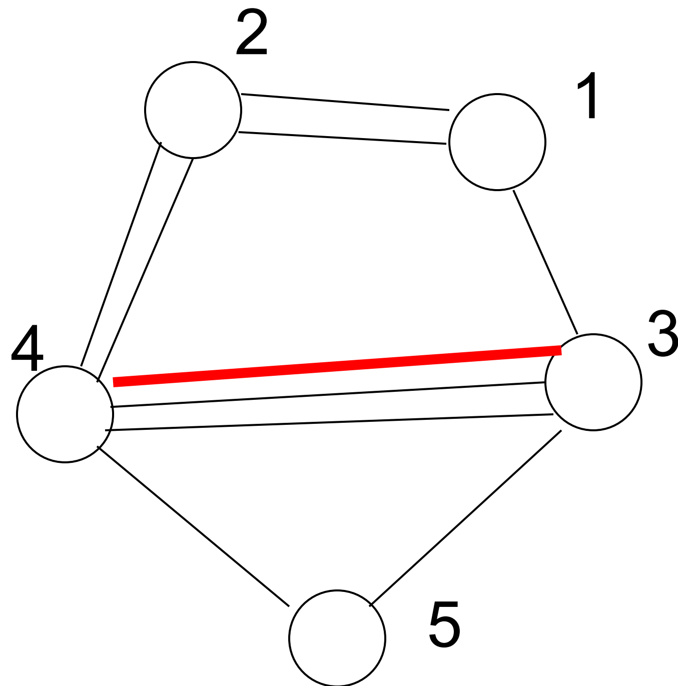
1. While the number of nodes in G is more than 2
   1.1. Choose an edge ~~e not in the min cut~~ randomly
   1.2. contract e
2. MinCut ← edges connecting the last two remaining vertices.

A Randomized Algorithm for min-cut

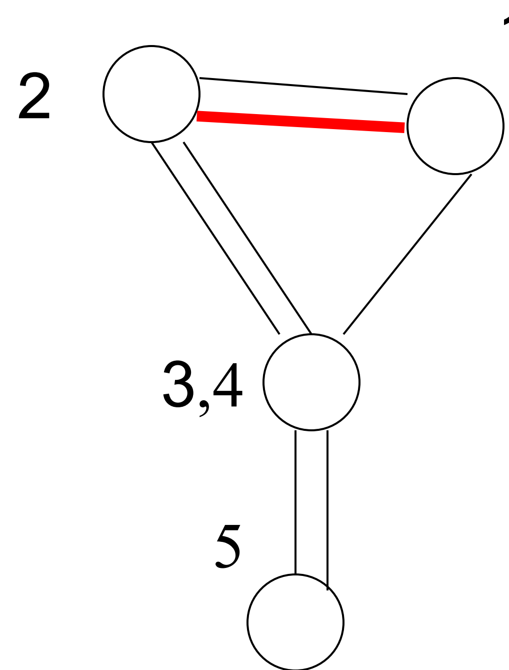
1. While the number of nodes in G is more than 2
   1.1. Choose a random edge.
   1.2. Merge the two nodes connected by that edge.
2. MinCut ← edges connecting the last two remaining vertices. Notice that these edges are actually all the edges connecting nodes contracted to one of the remaining nodes with nodes contracted to the other remaining node.

Running example

Iteration 1:

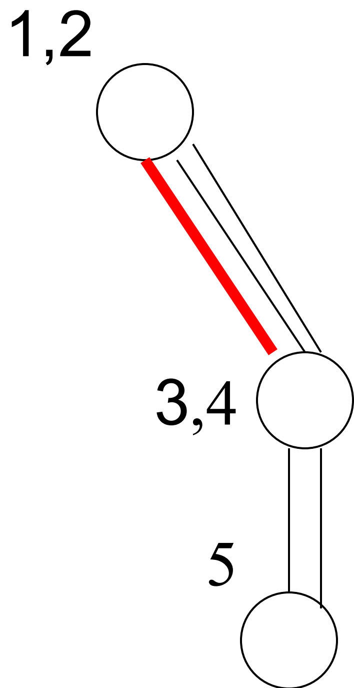


Iteration 2:

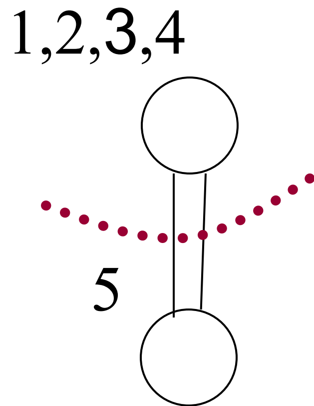


Running example (cont')

Iteration 3:

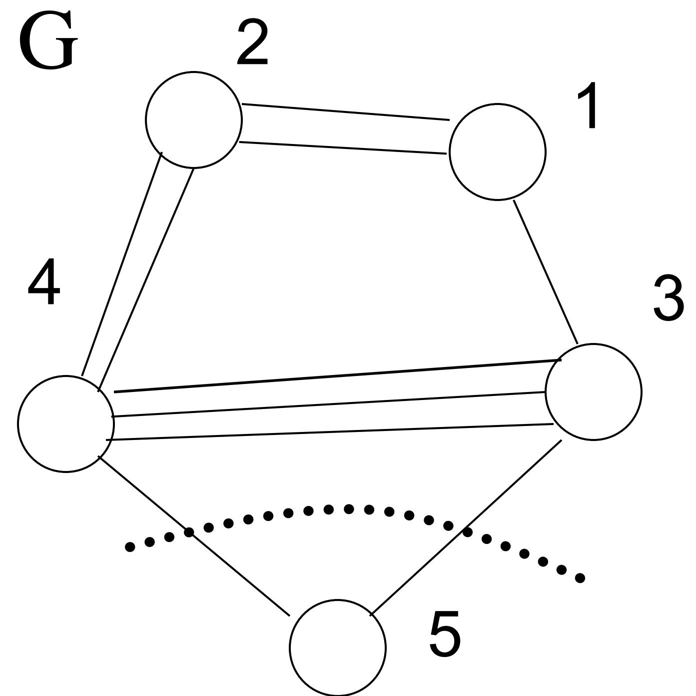


Iteration 4:



stop: the cut separates the vertices sets {5} and {1,2,3,4}

Output:



Min-Cut algorithm analysis

Will this algorithm return the Min-Cut?

Let C be a particular Min-Cut. Let k be size of C.

The algorithm will return C if it never contracts any edge of C.

We need to estimate the probability that in each iteration it chooses an edge not from C.

Min-Cut algorithm analysis (cont')

- The probability of picking at each iteration an edge from C is $k/|E|$, where $|E|$ is the number of edges still in G.
- There is no vertex with degree less than k (we already mentioned that $k \leq$ minimal degree)
- So $|E|$ = (sum of degrees)/2 $\geq kn/2$
- Therefore, the probability of choosing an edge from C in iteration i is at most

Note: In iteration i there are n-i+1 nodes.

$$\frac{k}{(n-i+1)\cdot k/2} = \frac{2}{n-i+1}$$

Min-Cut algorithm analysis (cont')

- Define ε_i to be the event of *not* choosing an edge from C in iteration i.

$$\Pr[\varepsilon_1] \geq 1 - \frac{2}{n}$$

- The probability of *not* choosing an edge from C in iteration *i* (assuming not chosen earlier) is at least

$$1 - \frac{2}{n-i+1}$$

- The probability of not choosing an edge from C at all is:

$$\Pr\left[\bigcap_{i=1}^{n-2} \varepsilon_i\right] \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \frac{\prod_{j=3}^{n}(n-j+1)}{\prod_{i=1}^{n-2}(n-i+1)} = \frac{2}{n(n-1)}$$

Is this good enough? How can we improve it?

- The probability that the algorithm will not find the Min-Cut is:

$$1-\frac{2}{n\cdot(n-1)}<1-\frac{2}{n^2}$$

- To improve the performance we repeat the algorithm $n^2/2$ times, making independent random choices each time.

- This way, the probability that the algorithm will not find the Min-Cut is reduced to

$$\left(1-\frac{2}{n^2}\right)^{n^2/2}\leq\frac{1}{e}$$

Recall: $\forall x \geq 1$: $(1 - 1/x)^x \leq 1/e$

Primality Testing

- Public-Key Cryptography needs large prime numbers
- How can you tell if p is prime?
  - Try dividing p by all smaller integers
    - Exponential in |p|
      (number of bits to represent p)
  - Improvement: try only smaller primes
    - Still exponential
- Until 2002 no deterministic Poly-time test was known!
  - Agrawal-Kayal-Saxena showed (inefficient) test
  - Much more efficient randomized tests exist.

Fermat's (little) Theorem

- If p is a prime then for all a,
  $$a^{p-1} \equiv 1 \pmod{p}$$
- Randomized test for a number n:
  - Choose a random a between 1 and n-1
  - Check if $a^{n-1} \equiv 1 \pmod{n}$
  - If not, n is definitely not prime
- What is the probability of error?

Fermat Primality Test

- If $a^{n-1} \not\equiv 1 \pmod{n}$ we call a a "Fermat Witness" for the compositeness of n
- For composite n, if $b^{n-1} \equiv 1 \pmod{n}$ (b≠1) we call b a "Fermat Liar"

- **Claim:** if a is a Fermat witness for n, then at least half the a's are Fermat witnesses
- **Proof:** Let $b_1, \ldots, b_k$ be Fermat liars. Then $(a \cdot b_i)^{n-1} \equiv a^{n-1} b_i^{n-1} \equiv a^{n-1} \cdot 1 \not\equiv 1 \pmod{n}$
  - So $ab_1, \ldots, ab_k$ are all Fermat witnesses!

Fermat Primality Test

- So, error probability at most $\frac{1}{2}$?

- No: we assumed there *exists* a Fermat witness.
  - There is an infinite sequence of composites that have no Fermat witnesses at all: the *Carmichael Numbers*
- Fermat primality test will always fail for Carmichael numbers.
  - They are rarer than primes, but still a problem.
- Slightly more complex test (Rabin-Miller) always works