

# **Randomized Algorithms**

**Part-II**

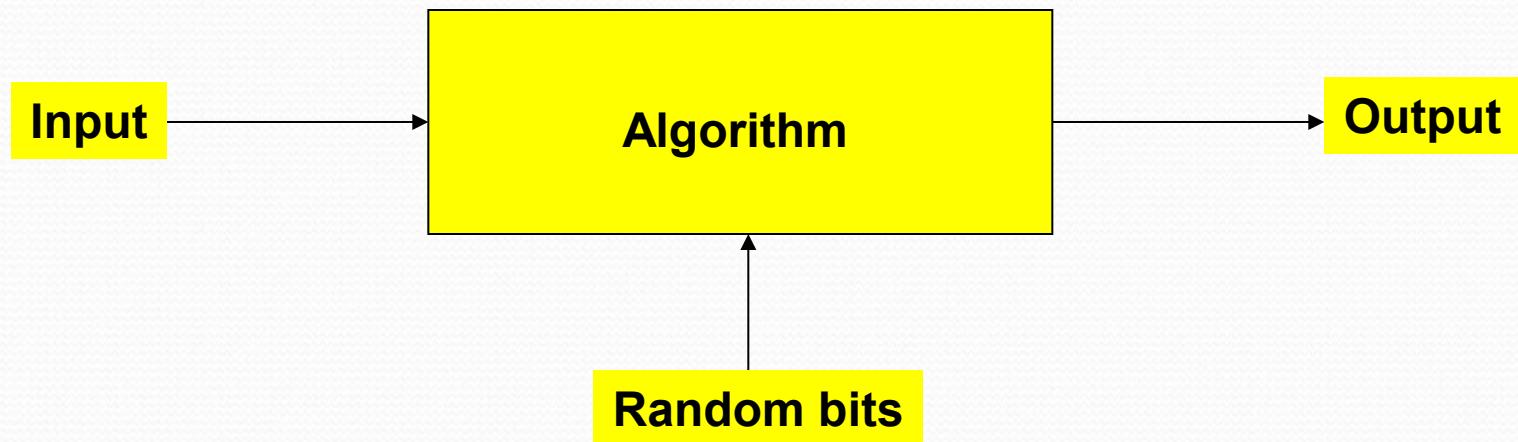
*Bibhudatta Sahoo*  
*National Institute of Technology Rourkela*

# Course Structure : Randomized Algorithm

- Introduction
- Some complexity Classes
- Computing  $\pi$
- Numerical Integration
- Primality Testing
- Randomized Algorithm for Majority Element
- Randomized Min-Cut
- Randomized Algorithm for the n-queen Problem
- Randomized Selection
- Randomized Quick-Sort

# Randomized Algorithms

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.



# Probabilistic or Randomized algorithm

- **At least once** during the algorithm, a random number is used to make a decision instead of spending time to work out which alternative is best.
- The worst-case running time of a randomized algorithm is almost always the same as the **worst-case running time** of the **non-randomized algorithm**.
- A good randomized algorithm has no bad input, but only bad random numbers.
- The random numbers are important, and we can get an **expected running time**, where we now average over all possible random numbers instead of over all possible inputs, or the mean time that it would take to solve the same instance over and over again

# Probabilistic or Randomized algorithm

- A randomized algorithm runs quickly but occasionally makes an error. The probability of error can, however, be made negligibly small. Any purported solution can be verified efficiently for correctness.
- A randomized algorithm may give probabilistic answers which are not necessarily exact.
- An expected running time bound is somewhat stronger than an **average-case bound**, but is weaker than the corresponding worst-case bound.

# Probabilistic or Randomized algorithm

- The same algorithm may behave differently when it is applied twice to the **same instance**. Its **execution time**, and even the result obtained, may vary considerable from one use to the next. If the algorithm gets stuck (e.g., core dump), simply restart it on the same instance for a fresh chance of success. If there is more than **one correct answer**, several different ones may be obtained by running the probabilistic algorithm more than once.

# Non-deterministic sorting algorithm

- Let  $A[i]$ ,  $1 < i < n$ , be an un-sorted array of positive integers. The non-deterministic algorithm **NSort(A, n)** sorts the numbers into non-decreasing order and then outputs them in this order.
- An auxiliary array  $B[1 : n]$  is used for convenience.
- The assignment statement  $j := \text{Choice}(1, n)$  could result in  $j$  being assigned any one of the integers in the range  $[1, n]$ .
- Its complexity is  $O(n)$ , whereas all deterministic sorting algorithms must have a complexity  $O(n \log n)$

# Replace Choice(1,n) with Uniform(1,n)

```
1  Algorithm NSort(A, n)
2    // Sort n positive integers.
3    {
4      for i := 1 to n do B[i] := 0; // Initialize B[ ].
5      for i := 1 to n do
6        {
7          j := Choice(1, n);
8          if B[j] ≠ 0 then Failure();
9          B[j] := A[i];
10         }
11        for i := 1 to n – 1 do // Verify order.
12          if B[i] > B[i + 1] then Failure();
13        write (B[1 : n]);
14      Success();
15    }
```

# Classification of Randomized Algorithm

1. Numerical Probabilistic Algorithm
2. Monte-carlo Algorithm
3. Las-Vegas Algorithm
4. Sherwood Algorithm



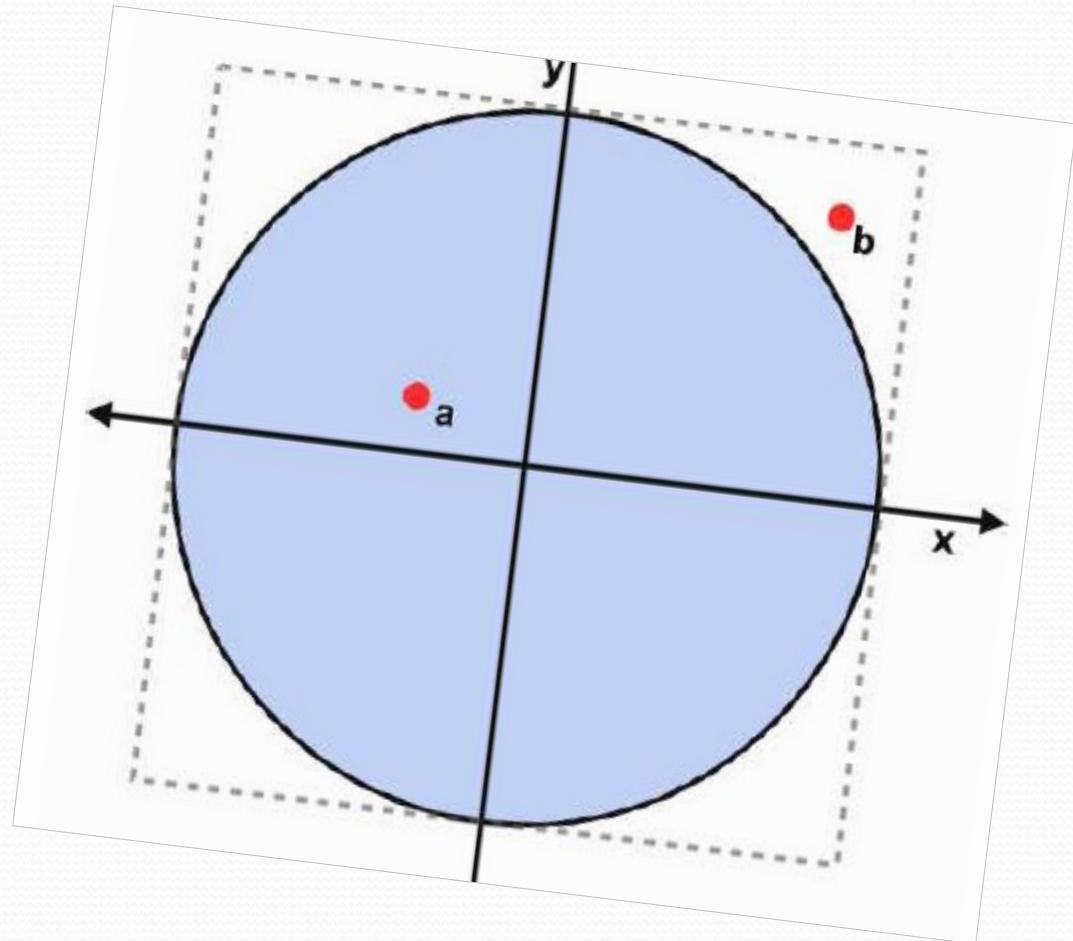
# Numerical Probabilistic Algorithm

Here the random element allows us to get approximate numerical results, often much faster than direct methods, and multiple runs will provide increasing approximation.

# Numerical Probabilistic Algorithm

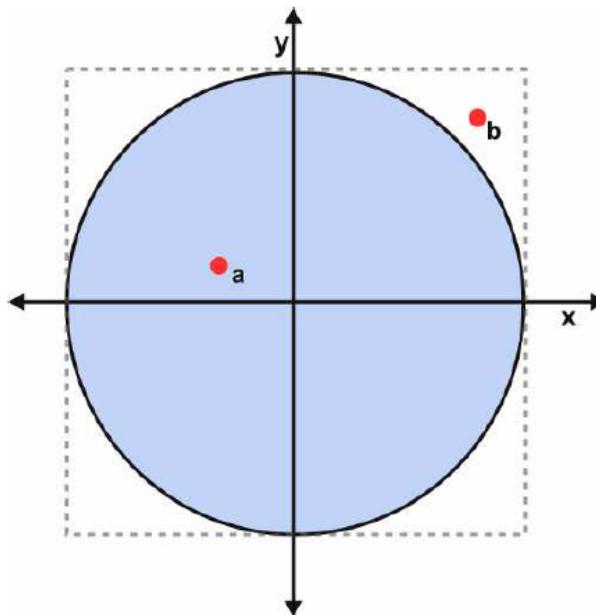
- For certain real-life problems, computation of an exact solution is not possible even in principle, e.g., uncertainties in the experimental data, digital computers handle only binary values, etc.
- For other problems, a precise answer exists but it would take too long to figure it out exactly.
- Numerical algorithms yield a confidence interval, and the expected precision improves as the time available to the algorithm increase.
- The error is usually inversely proportional to the square root of the amount of work performed

# Computing $\Pi$



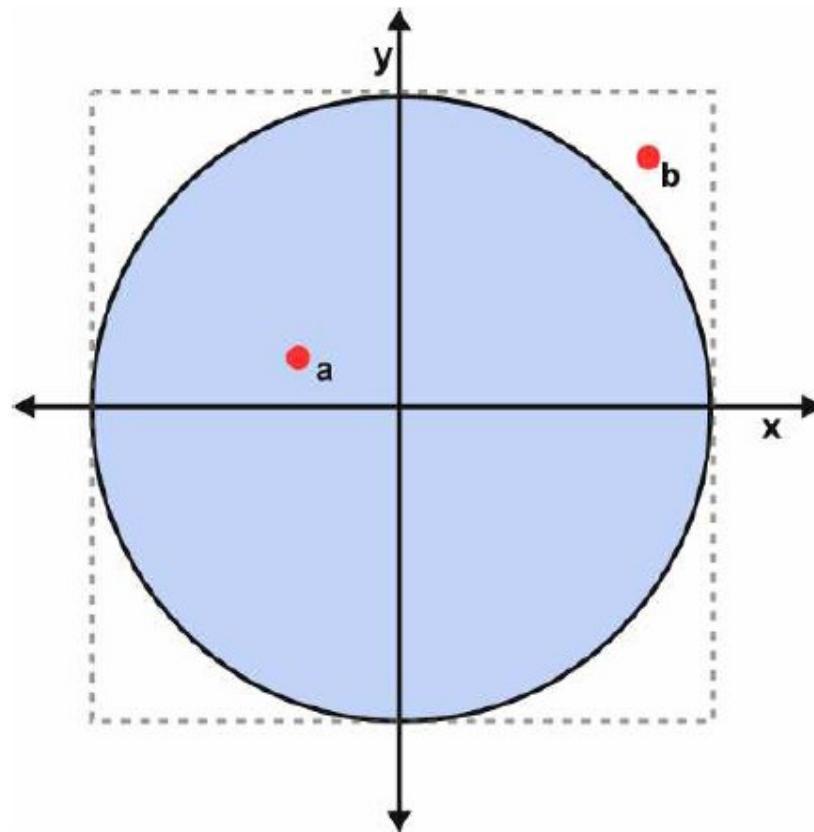
# Computing II

- One way to estimate the value of PI is to randomly throw darts at a unit circle (with radius of 1). Count how many of the darts fall inside of the circle.
- For example, in the figure below, point **a** is inside the circle and point **b** is outside the circle. Let **n** be the total number of darts thrown and let **k** be the number of darts falling inside the circle.



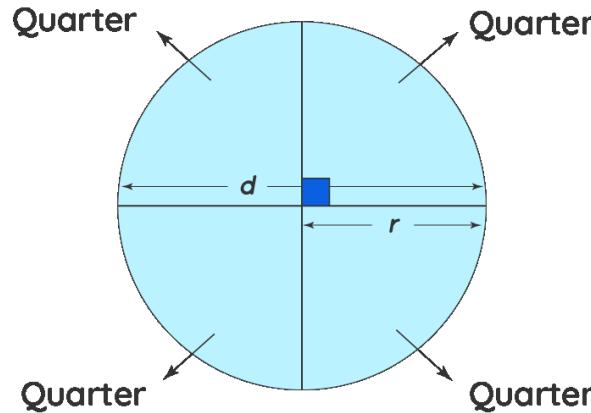
## Basis for estimating a value of $\pi$

- With number of trials  $n/k$  will be approaching the ratio of the area of the square and area of the circle.
- With sufficiently large  $n$ ,  $n/k = (4r^2/\pi r^2)$ , implies  $\pi = 4k/n$



# The area of the quarter circle

## Area of a Quarter Circle



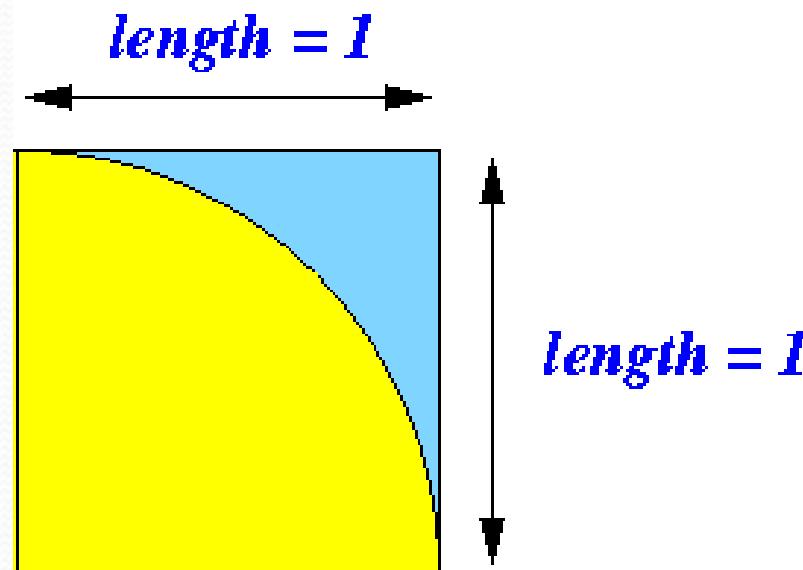
### Area of a quarter circle (Quadrant)

$$= \frac{\pi r^2}{4} \quad (\text{or}) \quad \frac{\pi d^2}{16}$$

- The **area of the full circle** =  $\pi \times 1^2 = \pi$
- The **area of the quarter circle** =  $\pi / 4$

## Computing the probability that a dart hits the yellow portion of the dart board

- If You throw  $n$  darts and count the number of



# Computing II

```
1. Algorithm Phi
2. // Numerical probabilistic algorithm to compute  $\Pi$ 
3. {
4.   k:=0;
5.   for j := 1 to n do
6.   {
7.     x := Uniform(0,1);
8.     // Uniform(0,1) returns a uniformly random number
     between 0 and 1
9.     y:= Uniform(0,1);
10.    If (  $x^2 + y^2 \leq 1$  ) then k:= k+1;
11.  }
12.  write ("Value of  $\Pi =$ ", 4k/n)
13. }
```

# Program design to compute $\pi$

1. Simulate the throw of a dart by generating random numbers between 0 and 1 for the x and y coordinates of a point. Note that the `rand()` function generates integer numbers. So you'll have to find a way to map those numbers to the range of 0 and 1. Since these are real numbers, you'll have to declare both x and y as float and do floating arithmetic. One way to do this is **x = rand()%1000/1000.0**
2. if the point (dart) falls inside or outside the circle. This is done by calculating the distance d from the origin of the circle to the point by using the Pythagoras Theorem which is `sqrt(x * x + y * y)`, where `sqrt` is the square root function.

# Program design to compute $\pi$

3. If the distance d calculated in step 2 is less than or equal to 1 then it is inside the circle and so you want to count it by executing inside++;
4. 4) Repeat steps 1 to 3 for n = 100 times.
5. 5) At the end of the loop, calculate PI as  $4.0 * \text{inside} / n$ . We need to multiply by 4 because we are generating the coordinates between 0 and 1. So our points are only in the first quadrant (or  $\frac{1}{4}$ ) of the circle as shown in the figure below

# Matlab code

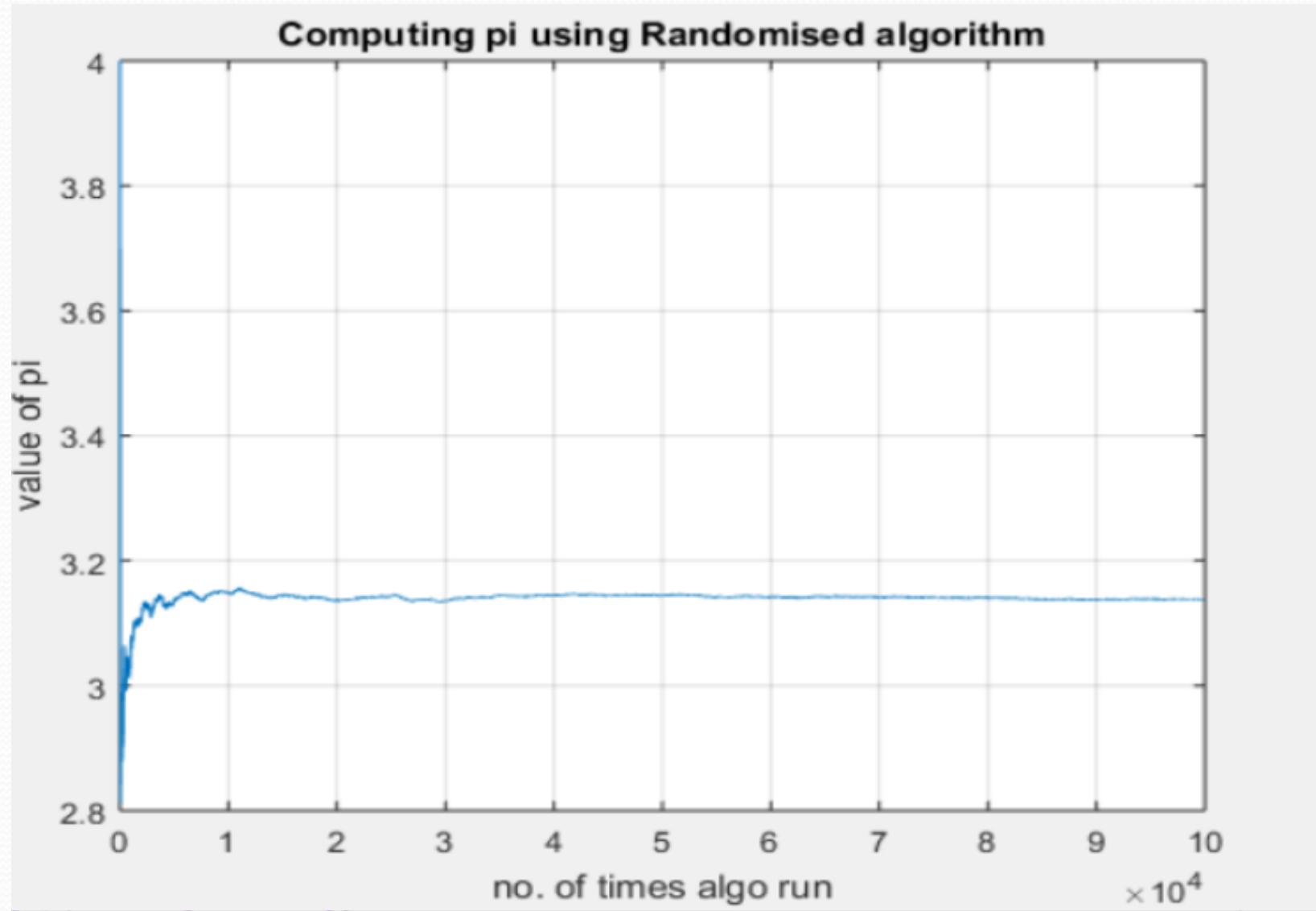
## Source Code:

```
times = zeros(1,100000);
pival = zeros(1,100000);
c = 1;
k = 0;
for i = 1:100000
    times(c) = i;
    x = rand();
    y = rand();
    if (x*x + y*y) <= 1
        k = k+1;
    end
    pival(c) = 4*(k/i);
    c = c+1;
end
disp(pival(100000));
plot(times,pival);
title('Computing pi using Randomised algorithm');
xlabel('no. of times algo run');
ylabel('value of pi');
grid on
```

## Result:

Value of pi=3.1452

# Simulation results



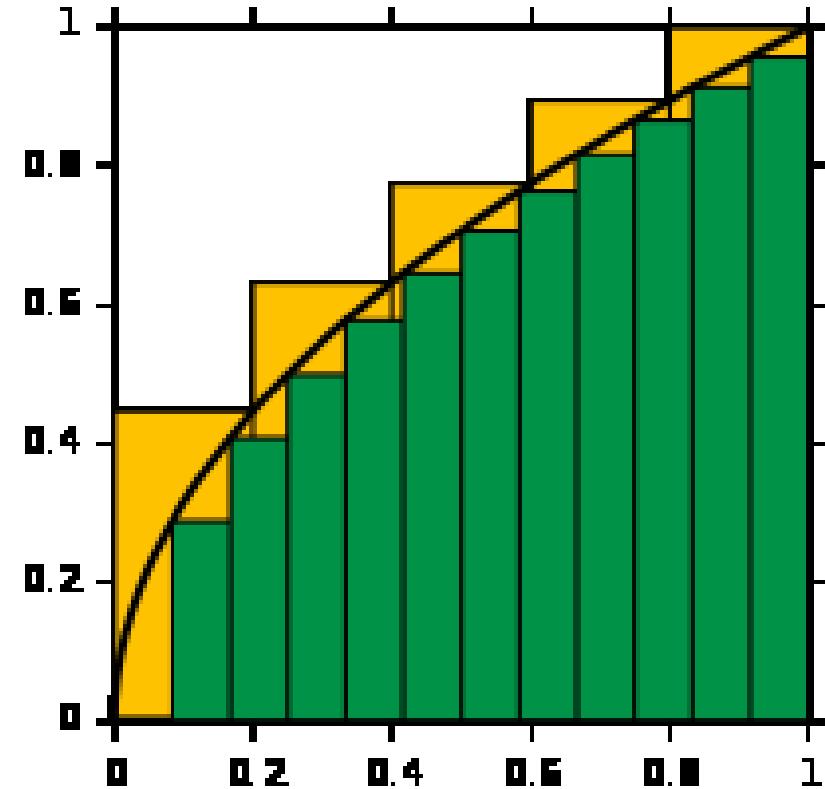
# Questions on simulation experiments

1. How accurate is your value of PI?
2. Do you get the same value when you run your program again?
3. Is your PI more accurate if you make n larger in step 4?
4. In step 1, what happens if you use integer calculations?
5. In step 1, you can also make PI more accurate if you increase the number.
6. How accurate can you make PI to be?

## Prime numbers can be obtained

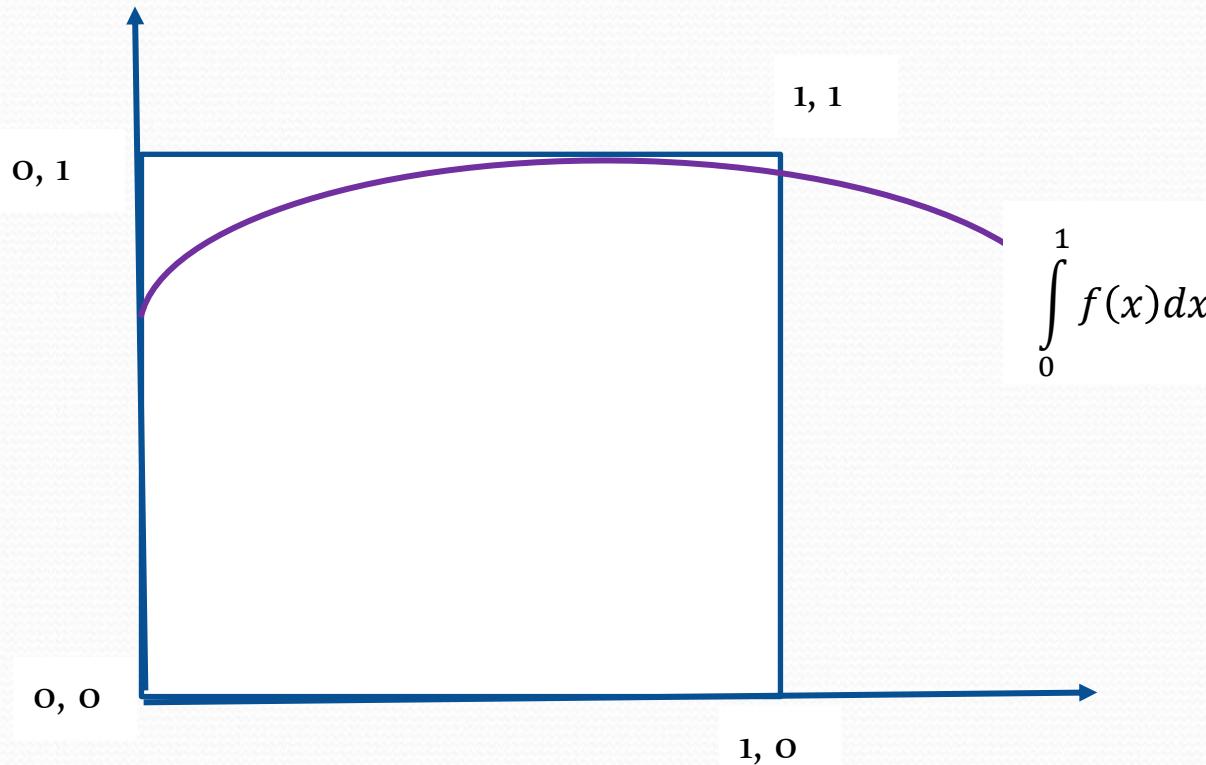
- <https://www.mathsisfun.com/numbers/prime-numbers-to-10k.html>

# Numerical Integration



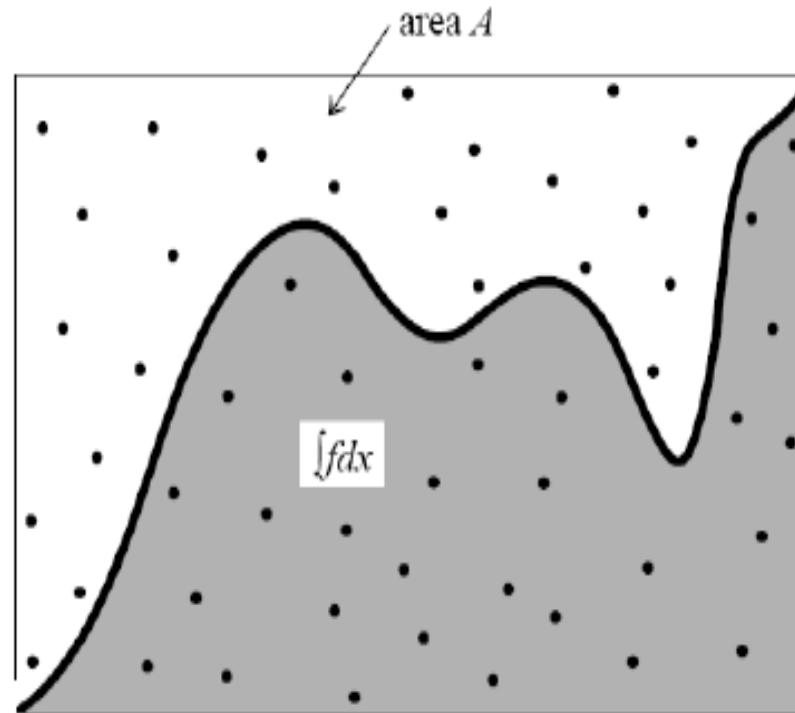
# Finding integral of a continuous function

- Let  $f$  be continuous function from  $[0, 1]$  to  $[0, 1]$ .
- The size of the area bounded by the curve  $y = f(x)$  in Cartesian coordinate is given by  $\int_0^1 f(x)dx$



# Finding integral of a continuous function

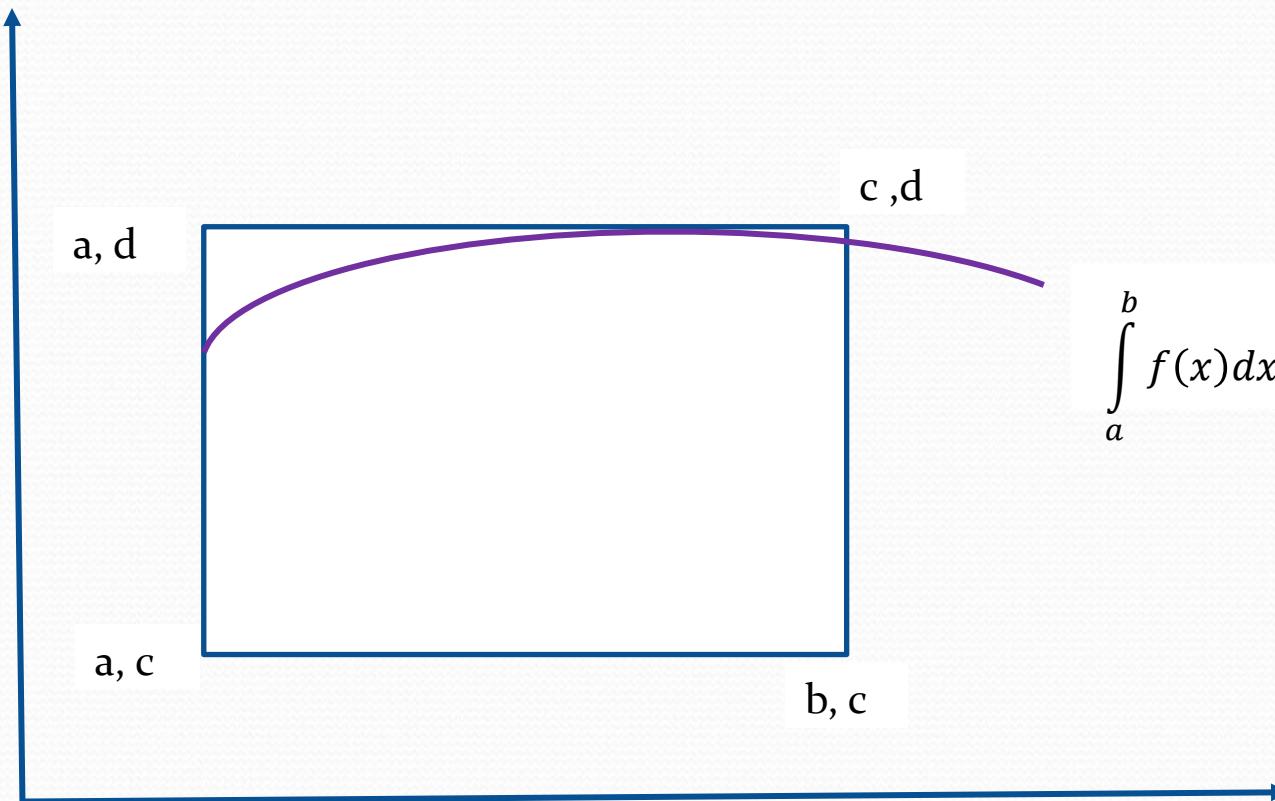
- The value of the integral can be estimated by throwing large number of darts( $n$ ) at the unit square. Let  $k$  be the darts that hits the square below the curve  $f(x)$ .
- The ratio  $k/n$  gives the value of the integral when  $n$  is sufficiently large.



# Numerical Integration

```
1. Algorithm Integral1
2. // Numerical probabilistic algorithm for Integral f(x)
3. {
4.   k:=0;
5.   for j := 1 to n do
6.   {
7.     x := Uniform(0,1);
8.     y := Uniform(0,1);
9.     If ( y <= f(x)) then k:= k+1;
10.    }
11.   write ("Value of the Integral = ", k/n)
12. }
```

# Numerical Integration $f(x)$ in $[a,b]$ to $[c,d]$



# Numerical Integration f(x) in [a,b] to [c,d]

```
1. Algorithm Integral2
2. // Numerical probabilistic algorithm for Integral f(x)
3. {
4.   k:=0;
5.   for j := 1 to n do
6.   {
7.     x := Uniform(a, b);
8.     y := Uniform(c, d);
9.     If ( y <= f(x)) then k:= k+1;
10.   }
11.   result := (b-a)(c+(d-c)k/n);
12.   write ("value of the integral = ", result)
13. }
```

Write a program that computes the value of the following integral using randomized algorithm.

$$\int_0^2 \sqrt{4 - x^2} dx$$

***Estimated Integral Value = 3.1353***

# Matlab code

```
function [ I ] = estimateIntegral ( num_pts_total )

    % Generate Points in interval [0,2]
    x = rand(1, 200) * 2;

    % Find maximum value of f(x) possible
    f_max = ceil(max(sqrt(4 .- (x).^2)));

    num_pts_under = 0;

    for i = 1 : num_pts_total

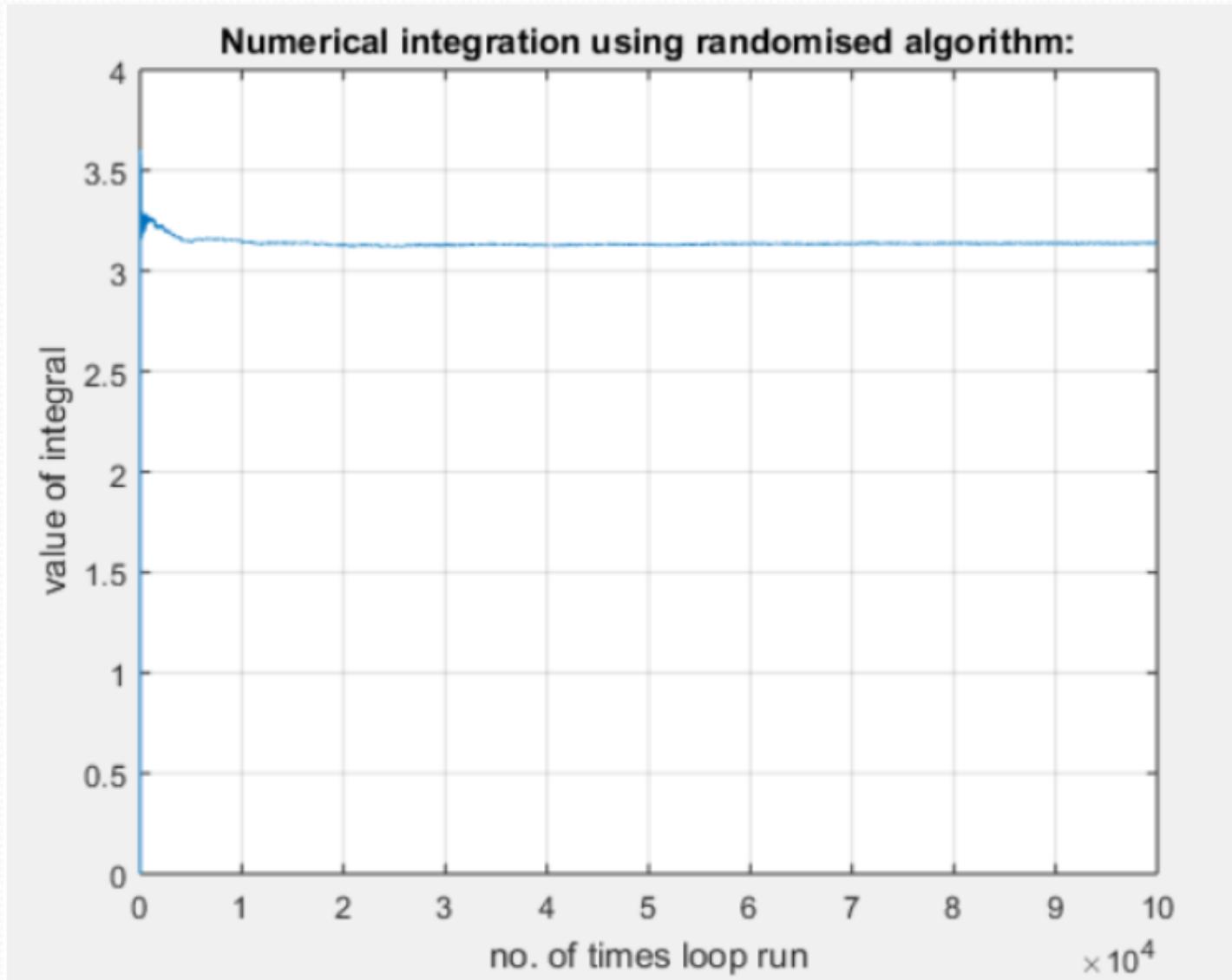
        % Generate x randomly in interval [0,2]
        x = rand(1,1) * 2;

        % Generate y randomly in interval [0, f_max]
        y = rand(1,1) * f_max;

        % Drop this point and check if it falls under the curve
        if sqrt(4 .- (x.^ 2)) > y
            num_pts_under = num_pts_under + 1;
        end
    end

    % Estimate Integral
    % (f_max * 2) is area of bounding box for the curve
    I = (num_pts_under / num_pts_total) * (f_max * 2);
end
```

# Simulation results

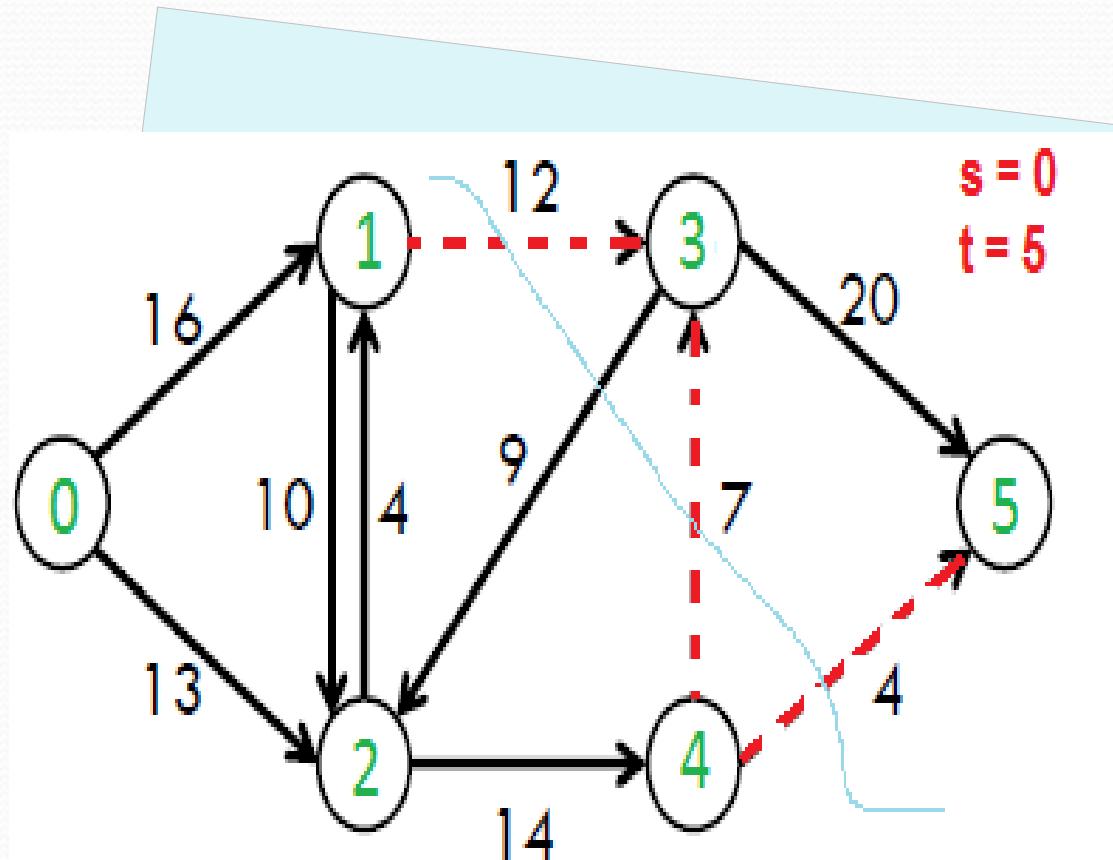


# Monte-carlo Algorithm

Algorithms which always return a result, but the result may not always be correct. We attempt to minimise the probability of an incorrect result, and using the random element, multiple runs of the algorithm will reduce the probability of incorrect results.

- **Randomized Min-Cut**
- **Primality Testing**
- **Finding Majority Element**

## Randomized MIN\_CUT

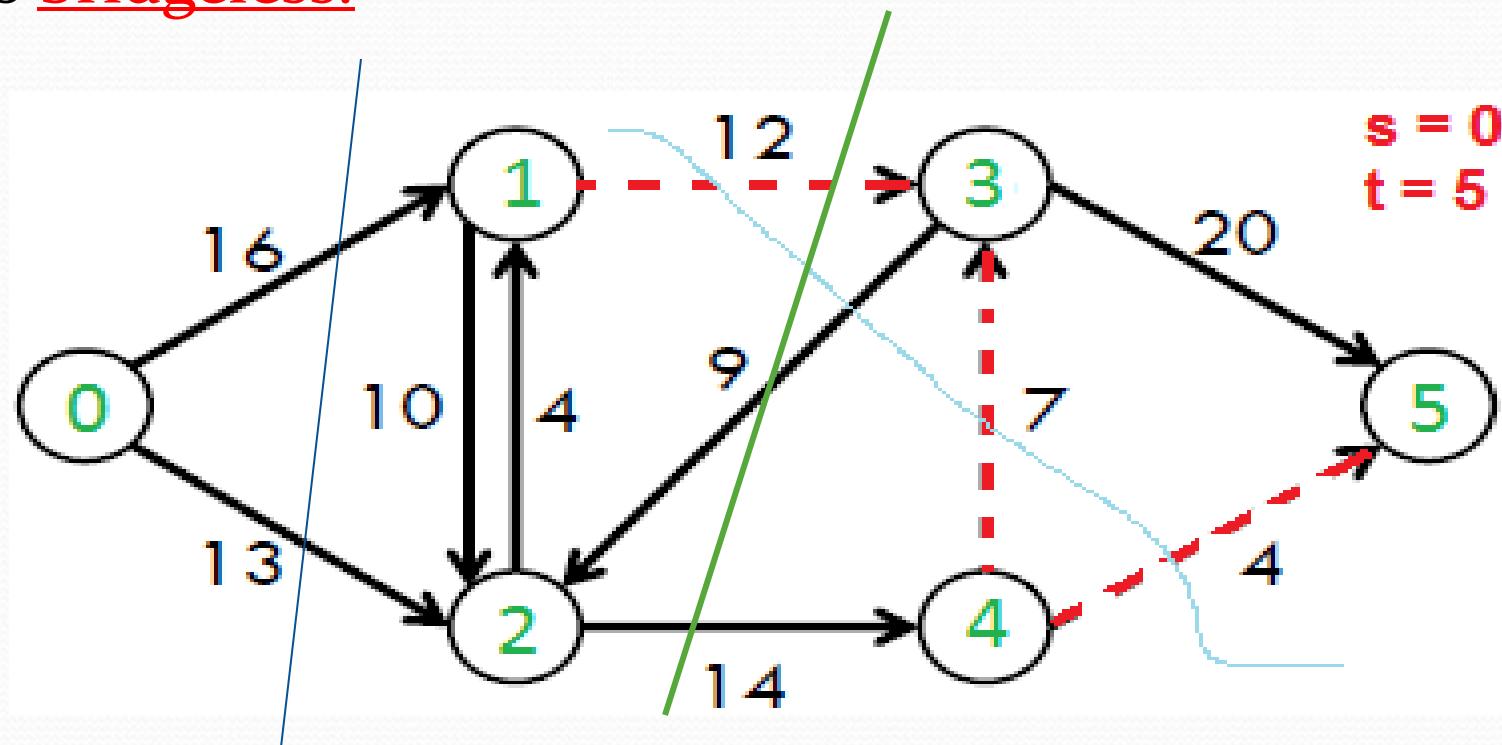


# Cut and cut-set

- In graph theory, a **cut** is a partition of the vertices of a graph into two disjoint subsets.
- Any cut determines a **cut-set**, the set of edges that have one endpoint in each subset of the partition.
- These edges are said to **cross** the cut. In a connected graph, each cut-set determines a unique cut, and in some cases cuts are identified with their cut-sets rather than with their vertex partitions.
- In a flow network, an **s-t cut** is a cut that requires the *source* and the *sink* to be in different subsets, and its *cut-set* only consists of edges going from the source's side to the sink's side.
- The *capacity* of an s-t cut is defined as the sum of **capacity** of each edge in the *cut-set*.

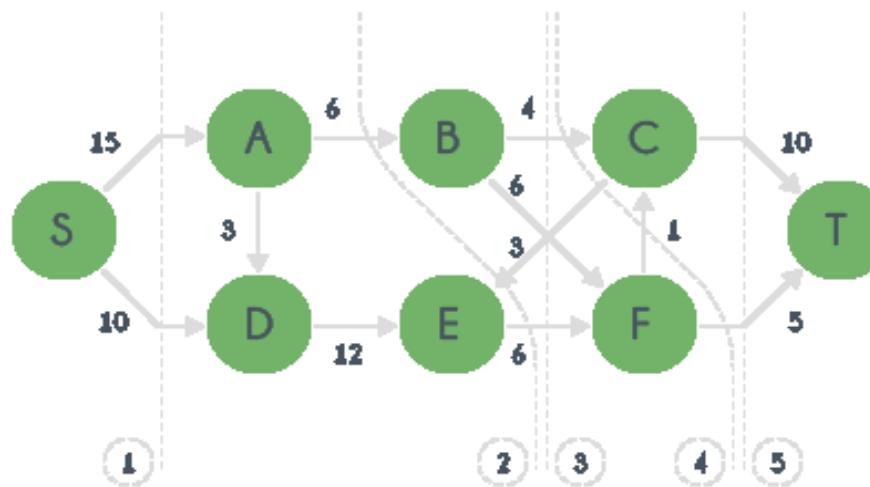
# Minimum cut

- A cut is *minimum* if the size or weight of the cut is not larger than the size of any other cut. The illustration on the right shows a minimum cut: the size of this cut is 2, and there is no cut of size 1 because the graph is bridgeless.



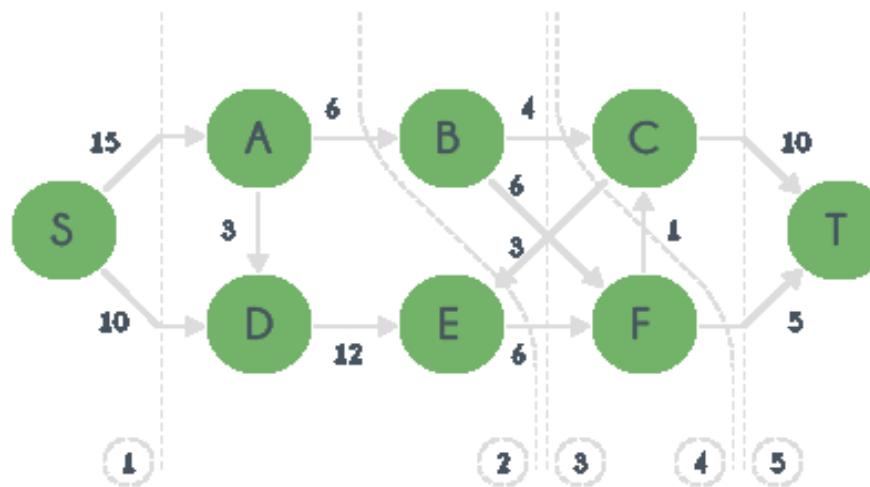
# Min-Cut of a weighted graph

- Min-Cut of a weighted graph is defined as the minimum sum of weights of (at least one)edges that when removed from the graph divides the graph into two groups.



# Min-Cut of a weighted graph

- Few possible cuts in the graph are shown in the graph and weights of each cut are as follows: Cut1:25, Cut2:12, Cut3:16, Cut4:10, Cut5:15Cut1:25, Cut2:12, Cut3:16, Cut4:10, Cut5:15.

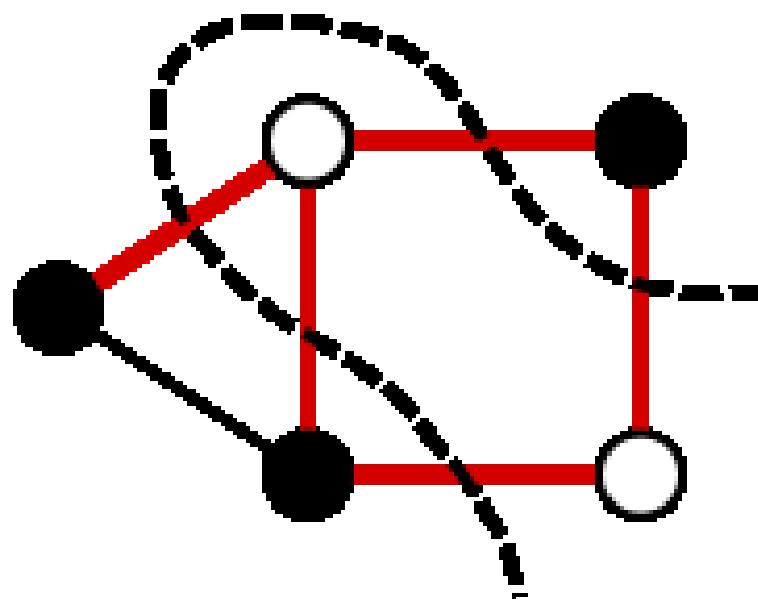


# Graph division

- You are given a graph with the following information:
- The graph contains  $N$  bidirectional edges. Each edge is denoted as  $(U, V)$ , where  $U$  and  $V$  represent nodes of the graph. Each edge  $(U, V)$  has a Value  $Val$  associated with it.
- You are also given two nodes  $X$  and  $Y$ . You have to divide the graph by deleting some edges such that:
  - Each part of the divided graph contains either  $X$  or  $Y$ .
  - The loss of the values from the edges is minimal.
  - **Write an algorithm to find the value that is lost while dividing this graph.**

# The Problem

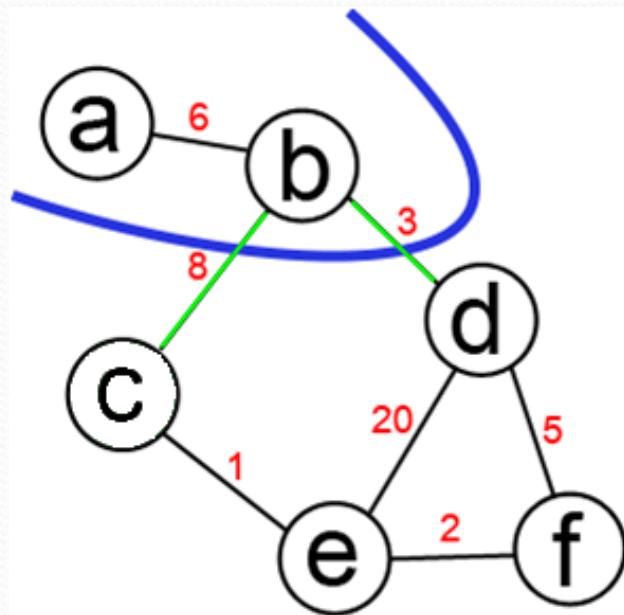
- Input: Undirected graph  $G=(V,E)$   
Edges have non-negative weights
- Output: A minimum cut of  $G$



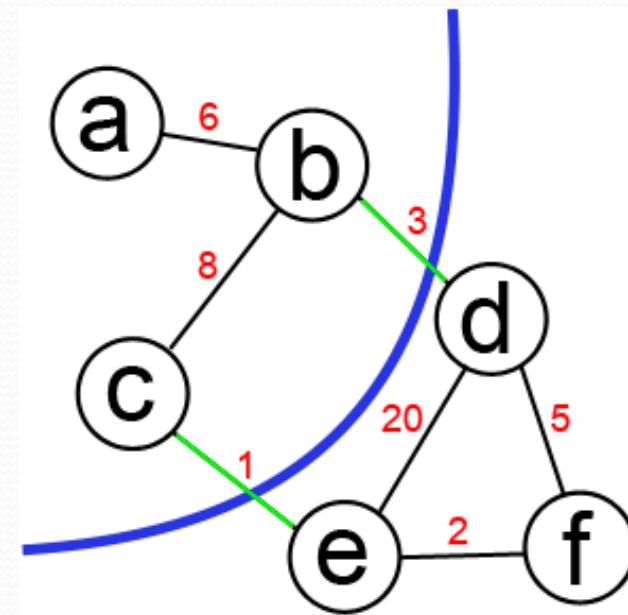
# Cut Example

Cut: set of edges whose removal disconnects  $G$

Min-Cut: a cut in  $G$  of minimum cost



Weight of this cut: 11

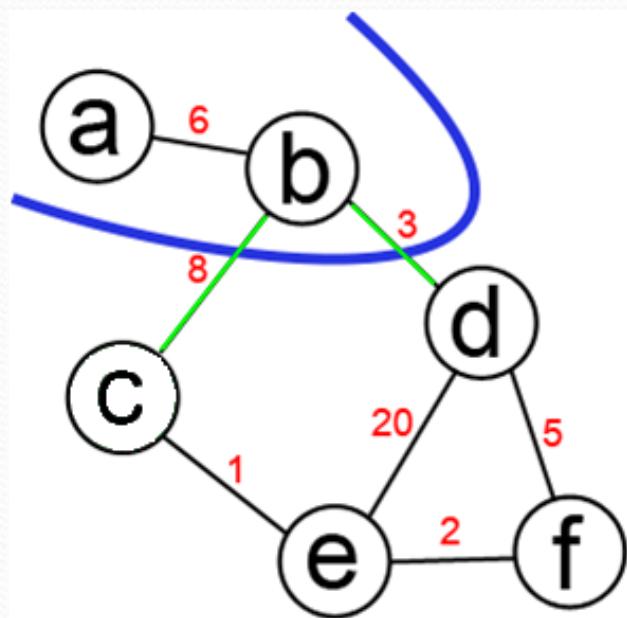


Weight of min cut: 4

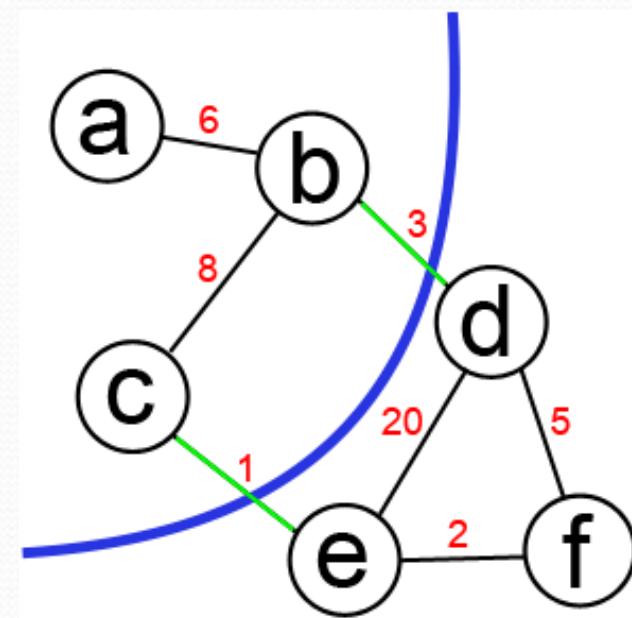
## *s-t* Cut Example

*s-t* cut: cut with  $s$  and  $t$  in different partitions

$$s = a \text{ and } t = d$$



Weight of this  $a-d$  cut: 11



Weight of min  $a-d$  cut: 4

# Naive Solution

- Check every possible cut
- Take the minimum
- A complete graph has an edge between any two vertices. You can get an edge by picking any two vertices. So if there are  $n$  vertices, there are  $n(n-1)/2$  edges.
- Running time:  $O(2^n)$

# Randomized Min-cut

- Given an undirected, connected multi-graph  $G(V,E)$  , we want to find a cut  $(V_1, V_2)$  such that the number of edges between  $V_1$  and  $V_2$  is minimum.
- This problem can be solved optimally by applying the max-flow min-cut algorithm  $O(n^2)$  time by trying all pairs of source and destination.

# Karger's algorithm: 1/3

- Karger's algorithm is a randomized algorithm to compute a minimum cut of a connected **Graph**.
- It was invented by **David Karger** and first published in 1993.
- The algorithm is a **Monte carlo algorithm**, i.e. its running time is deterministic, but it isn't guaranteed that at every iteration the best solution will be found.

## Karger algorithm: 2/3

- Karger algorithm builds a cut of the graph by randomly creating this partitions, and in particular by choosing at each iteration a random edge, and contracting the graph around it:
- basically, merging its two endpoints in a single vertex, and updating the remaining edges, such that the self-loops introduced (like the chosen edge itself) are removed from the new Graph
- storing parallel-edges (if the algorithm chooses an edge  $(u,v)$  and both  $u$  and  $v$  have edges to a third vertex  $w$ , then the new Graph will have two edges between the new vertex  $z$  and  $w$ )
- After  $n-2$  iterations, only two macro-vertex will be left, and the parallel edges between them will form the cut.

## Karger's algorithm:3/3

- Actually the probability of finding the minimum cut in one run of the algorithm is pretty low, with an upper bound of  $1/(n^2)$ , where  $n$  is the number of vertices in the Graph.
- Nonetheless, by running the algorithm multiple times and storing the best result found, the probability that none of the runs finds the minimum cut becomes very small:  $1/e$ (Neper) for  $n$  squared runs, and  $1/n$  for  $n^2 * \log(n)$  runs - for large values of  $n$ , i.e. for large Graphs, that's a negligible probability.

# Simple randomized algorithm: D. Karger

- The simple algorithm (view #1)
  1. Pick an edge  $(x, y)$  at random in  $G$ .
  2. CONTRACT the edge, keeping multi-edges, but removing self-loops. (i.e., if there were edges  $(x, v)$  and  $(y, v)$ , we keep both of them.)
  3. If there are more than 2 nodes, go back to 1. Else, output the edges remaining as your cut.

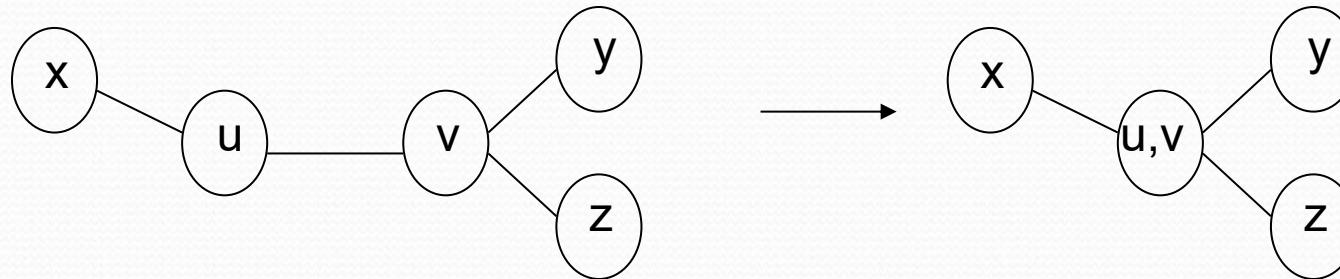
# Randomized algorithm for min-cut

1. Algorithm Min-cut( $G = (V, E)$ )
2. {
3.   While ( $|V| > 2$ ) do
4.   {
5.     Select an edge  $(x, y) \in E$  uniformly at random;
6.     Contract the edge  $(x, y)$ ;
7.   }
8.   return ( $|E|$ );
9. }

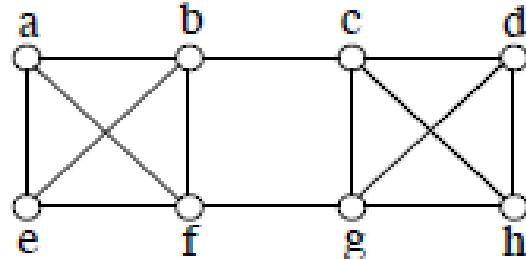
# Randomized Min-cut

In randomized Min-cut, we repeatedly do the following:

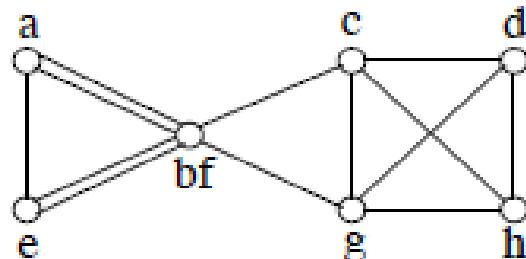
- Pick randomly an edge  $e(u,v)$ . Merge  $u$  and  $v$ , and remove all the edges between  $u$  and  $v$ . For example:
- until there are only 2 vertices left. We will report the cut between these 2 vertices as the min-cut.



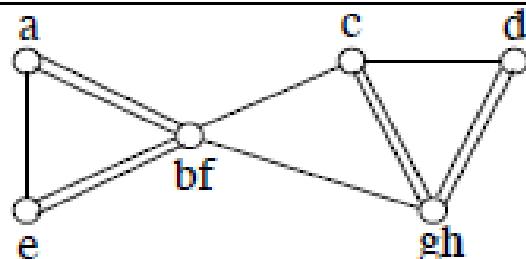
# Working of Karger's algorithm for Min-cut



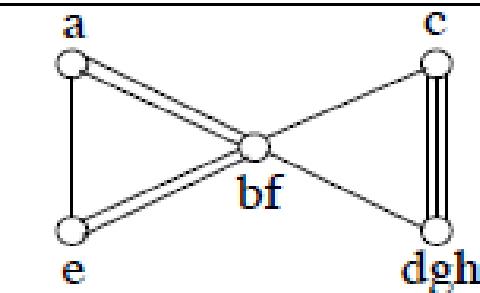
14 edges to choose from  
Pick  $b - f$  (probability  $1/14$ )



13 edges to choose from  
Pick  $g - h$  (probability  $1/13$ )

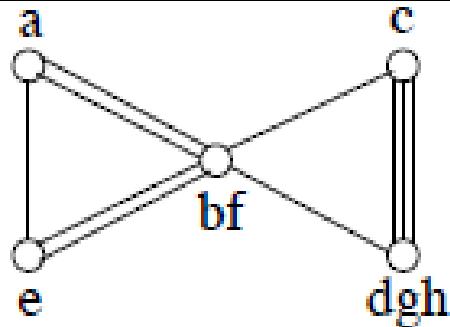


12 edges to choose from  
Pick  $d - gh$  (probability  $1/6$ )

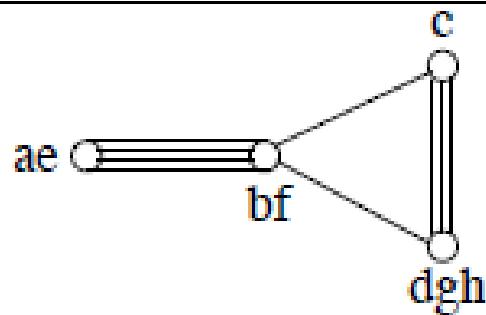


10 edges to choose from  
Pick  $a - e$  (probability  $1/10$ )

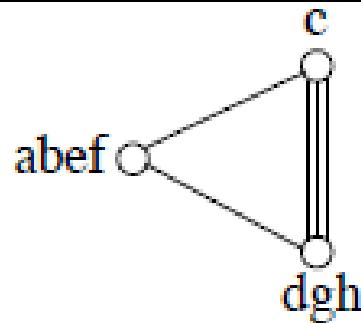
# Working of Karger's algorithm for Min-cut



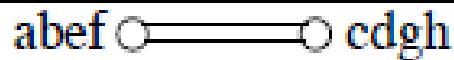
10 edges to choose from  
Pick  $a - e$  (probability  $1/10$ )



9 edges to choose from  
Pick  $ab - ef$  (probability  $4/9$ )



5 edges to choose from  
Pick  $c - dgh$  (probability  $3/5$ )



Done: just two nodes remain

# Analysis of Randomized Min-cut

- Let  $k$  be the min-cut of the given graph  $G(E,V)$  where  $|V|=n$ .
- Then  $|E| \geq kn/2$ .
- The probability  $q_1$  of picking one of those  $k$  edges in the first merging step  $\leq 2/n$
- The probability  $p_1$  of not picking any of those  $k$  edges in the first merging step  $\geq (1-2/n)$
- Repeat the same argument for the first  $n-2$  merging steps.
- Probability  $p$  of not picking any of those  $k$  edges in all the merging steps  $\geq (1-2/n)(1-2/(n-1))(1-2/(n-2))...(1-2/3)$

# Analysis of Randomized Min-cut

- Therefore, the probability of finding the min-cut:

$$\begin{aligned} P &\geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \dots \left(1 - \frac{2}{3}\right) \\ &= \frac{(n-2)(n-3)\dots(1)}{n(n-1)\dots3} \\ &= \frac{2}{n(n-1)} \end{aligned}$$

- If we repeat the whole procedure  $n^2/2$  times, the probability of **not finding** the min-cut is at most

$$(1 - 2/n^2)^{n^2/2} \cong 1/e$$

- This expression is less than  $1/e$ , where  $e = 1 + 1 + 1/2! + 1/3! + \dots$  which is 2.71828 approximately.

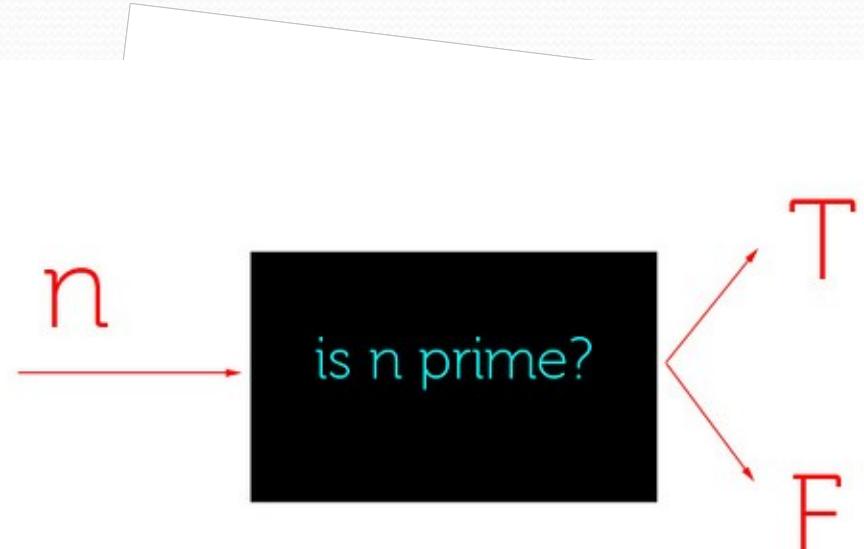
## Analysis of Randomized Min-cut

Hence the probability of failure has been reduced from

$$(1 - 2/n^2)^{n^2/2} \text{ to } 1/e$$

through repetition process.

# Primality Testing



# Primality Testing

- Primality testing is an important algorithmic problem.
- In addition to being a fundamental mathematical question, the problem of how to determine whether a given number is prime has tremendous practical importance.
- Every time someone uses the **RSA public-key cryptosystem**, they need to generate a private key consisting of **two large prime** numbers and a public key consisting of their product.
- To do this, one needs to be able to **check rapidly** whether a number is prime.

# Primality Testing

- In 1980, Michael Rabin discovered a **randomized polynomial-time** algorithm to test whether a number is prime.
- It is called the Miller-Rabin primality test because it is closely related to a deterministic algorithm studied by Gary Miller in 1976.
- This is still the most practical known primality testing algorithm, and is widely used in software libraries that rely on RSA encryption

# Primality test

**Definition:**

An integer  $p \geq 2$  is prime **iff** ( $a \mid p \rightarrow a = 1$  **or**  $a = p$ ).

**Algorithm:** deterministic primality test (naive)

**Input:** integer  $n \geq 2$

**Output:** answer to the question: Is  $n$  prime?

**if**  $n = 2$  **then return** true

**if**  $n$  even **then return** false

**for**  $i = 1$  **to**  $\sqrt{n}/2$  **do**

**if**  $2i + 1$  divides  $n$

**then return** false

**return** true

**Complexity:**  $\Theta(\sqrt{n})$

# Primality test

**Goal:**

## Randomized method

- Polynomial time complexity (in the length of the input)
- If answer is “not prime”, then  $n$  is not prime
- If answer is “prime”, then the probability that  $n$  is not prime is at most  $p > 0$

$k$  iterations: probability that  $n$  is not prime is at most  $p^k$

# Primality test

## Observation:

Each odd prime number  $p$  divides  $2^{p-1} - 1$ .

Examples:  $p = 17, 2^{16} - 1 = 65535 = 17 * 3855$

$p = 23, 2^{22} - 1 = 4194303 = 23 * 182361$

## Simple primality test:

- 1 Calculate  $z = 2^{n-1} \bmod n$
- 2 if  $z = 1$
- 3 then  $n$  is possibly prime
- 4 else  $n$  is definitely not prime

Advantage: This only takes polynomial time

# Simple primality test

**Definition:**

$n$  is called pseudoprime to base 2, if  $n$  is not prime and

$$2^{n-1} \bmod n = 1.$$

**Example:**  $n = 11 * 31 = 341$

$$2^{340} \bmod 341 = 1$$

# Randomized primality test

**Theorem:** (Fermat's little theorem)

If  $p$  prime and  $0 < a < p$ , then

$$a^{p-1} \bmod p = 1.$$

**Definition:**

$n$  is pseudoprime to base  $a$ , if  $n$  not prime and

$$a^{n-1} \bmod n = 1.$$

**Example:**  $n = 341$ ,  $a = 3$

$$3^{340} \bmod 341 = 56 \neq 1$$

# Randomized primality test

**Algorithm:** 1 Randomized primality test

- 1 **Randomly choose**  $a \in [2, n-1]$
- 2 Calculate  $a^{n-1} \bmod n$
- 3 **if**  $a^{n-1} \bmod n = 1$
- 4   **then**  $n$  is possibly prime
- 5   **else**  $n$  is definitely not prime

$\text{Prob}(n \text{ is not prim, but } a^{n-1} \bmod n = 1) \ ?$

# Monte-Carlo algorithm for primality test

```
1. Algorithm Ptesting(N, k)
2. // k is the number of attempts to test N for primality
3. {
4.   i:=1;
5.   r := Uniform(1, N-1);
6.   Remainder := MOD(N, r);
7.   If (Remainder = 0 ) Goto 12;
8.   i :=i+1;
9.   if(i ≤ k) then Goto 5;
10.  Print('N is prime');
11.  Exit;
12.  Print ('N is Composite');
13.  Exit;
14. }
```

# Monte-Carlo algorithm for primality test

- The parameter k controls number of repetitions.
- By increasing the value of k, the probability of the error that the number is composite but is reported as prime by the algorithm may be reduced.

## Example

Write a program that test a number to be prime or not. Perform an analysis to compute the correctness?

<https://www.rsok.com/~jrm/first100primes.html>

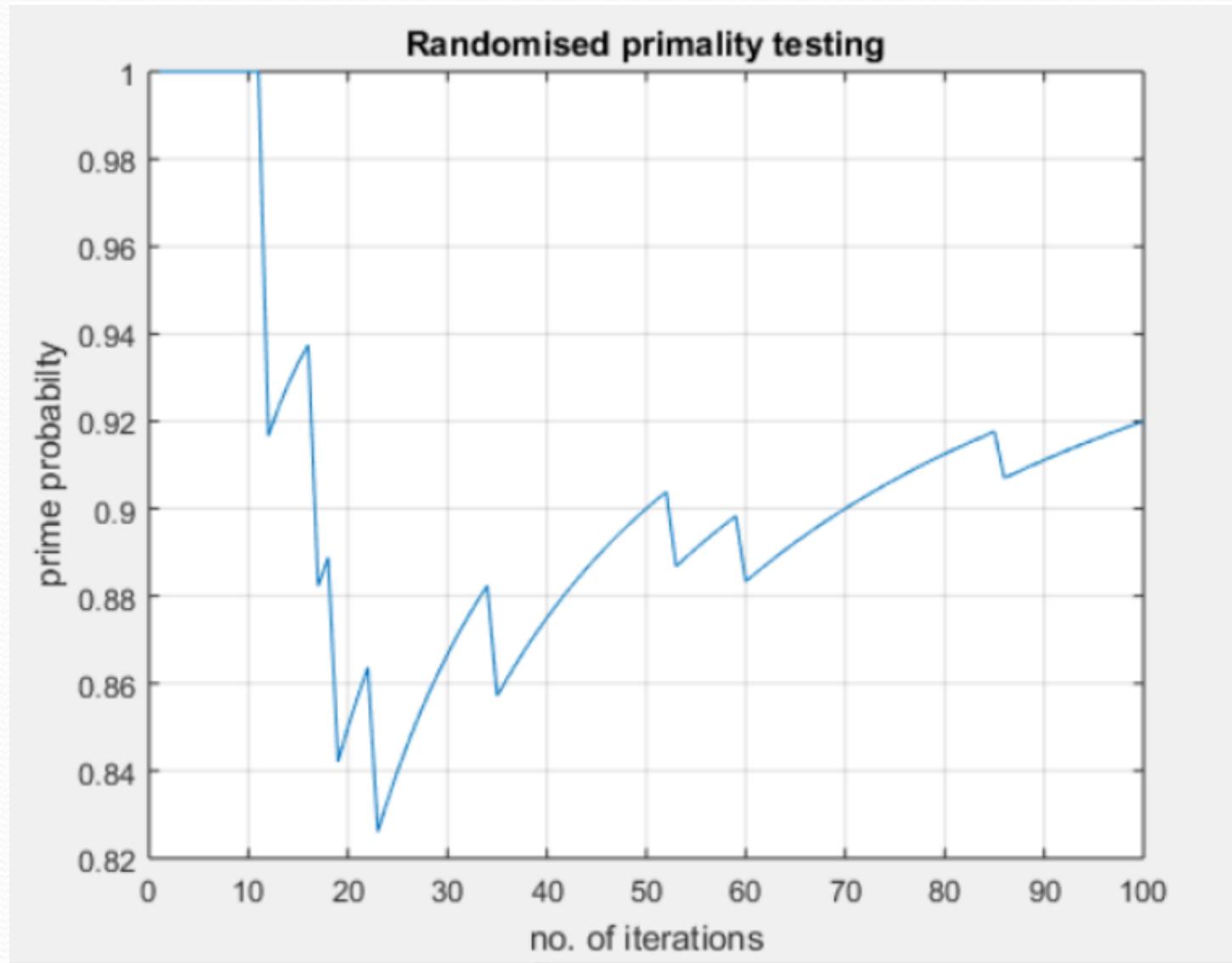
# Source code in Matlab

## Source code:

```
n = 52;
k = 0;
itr = [];
prob = [];
flag = false;
for i=1:100
    itr = [itr i];
    fact = randi([1, n-1]);
    rem = mod(n, fact);
    if rem ~= 0
        k = k+1;
    end
    prob = [prob k/i];
end

fprintf('prime probability:%f\n', k/100);
plot(itr,prob);
xlabel('no. of iterations');
ylabel('prime probability');
title('Randomised primality testing');
grid on;
```

# Simulation result



# Majority element

a list of numbers



3 → appears 4 times

©w3resource.com

# Problem Description

- **Problem Description:** You are given an array A[] consisting of n elements. You need to find and return the number which appears more than  $n/2$  times.

Approach	Time Complexity	Space Complexity
Brute Force: Using nested loops	$O(n^2)$	$O(1)$
Using Sorting	$O(n \log n)$	$O(1)$ Heap Sort $O(n)$ Merge Sort
Using Hashing	$O(n)$	$O(n)$
Bit Manipulation	$O(n)$	$O(1)$
Boyer-Moore Majority Vote Algorithm	$O(n)$	$O(1)$

# Majority element

- An array A of N natural numbers has a majority element n, if the number of occurrences of n in A is greater than  $N/2$ .

**For example :**

- Input:  $A[] = \{3, 9, 1, 3, 5, 3, 3\}$
- Output: 3
- Input:  $A[] = \{8, 8, 8, 8, 8, 10, 10\}$
- Output: 8

- Is an element a majority element if it appears exactly  $n/2$  times? (Ans: No, the majority element needs to appear more than  $n/2$  times)
- What should I return if there is no majority element? (Ans: In that case, return -1, assume that “-1” will never be an element in the input array)

# Randomized :Majority element

- **Question:** How would you decide whether an array has a majority element using a deterministic algorithm, and what is its complexity?
- By choosing an element at random and testing to see if it is a majority element we get a probabilistic algorithm.
- The complexity of the algorithm is  $O(Nk)$ , where  $k$  is the number of times the algorithm **Majority()** is invoked.

# Majority element

```
1. Algorithm Main(A, N)
2. {
3.   Counter :=0;
4.   For i:= 1 to k do
5.   {
6.     Call Majority(A, N, R, E);
7.     If R =‘True’ then Counter = Counter + 1;
8.     If Counter > o then goto 10;
9.   }
10.  If Counter > o then
11.  {
12.    Print(‘Majority Element =‘ E);
13.  Else
14.    Print(‘No Majority Element “);
15.  }
16. }
```

# Algorithm Majority(A, N, R, E)

```
1. Algorithm Majority(A, N, R, E)
2. // A is an array of N element
3. {
4.   R := 'False';
5.   choice := Uniform(1,N);
6.   count := 0;
7.   for i:= 1 to N do
8.   {
9.     If A(i) = A(choice) then count := count +1;
10.    }
11.   If (count > N/2) then
12.   {
13.     R := 'True';
14.     E := A(choice);
15.   }
16. }
```

# Algorithm Majority()

- If the algorithm **Majority()** makes R= ‘True’, it means the array has a **majority element**.
- The answer is correct if the algorithm makes R= ‘False’, we are not sure whether the array has got a majority element or not. We might have picked up wrong element for the checking the presence of majority element in the array.
- If there is a majority element picking up a **minority** element is less than 0.5.
- If we use **Majority ()** ten times , the probability of missing the majority element ,if there is a majority element is **less than**  $(0.5)^{10} = 0.00097$
- The complexity of the algorithm is  $O(Nk)$ , where k is the number of times the algorithm **Majority()** is invoked.

# Five possible solutions for majority element

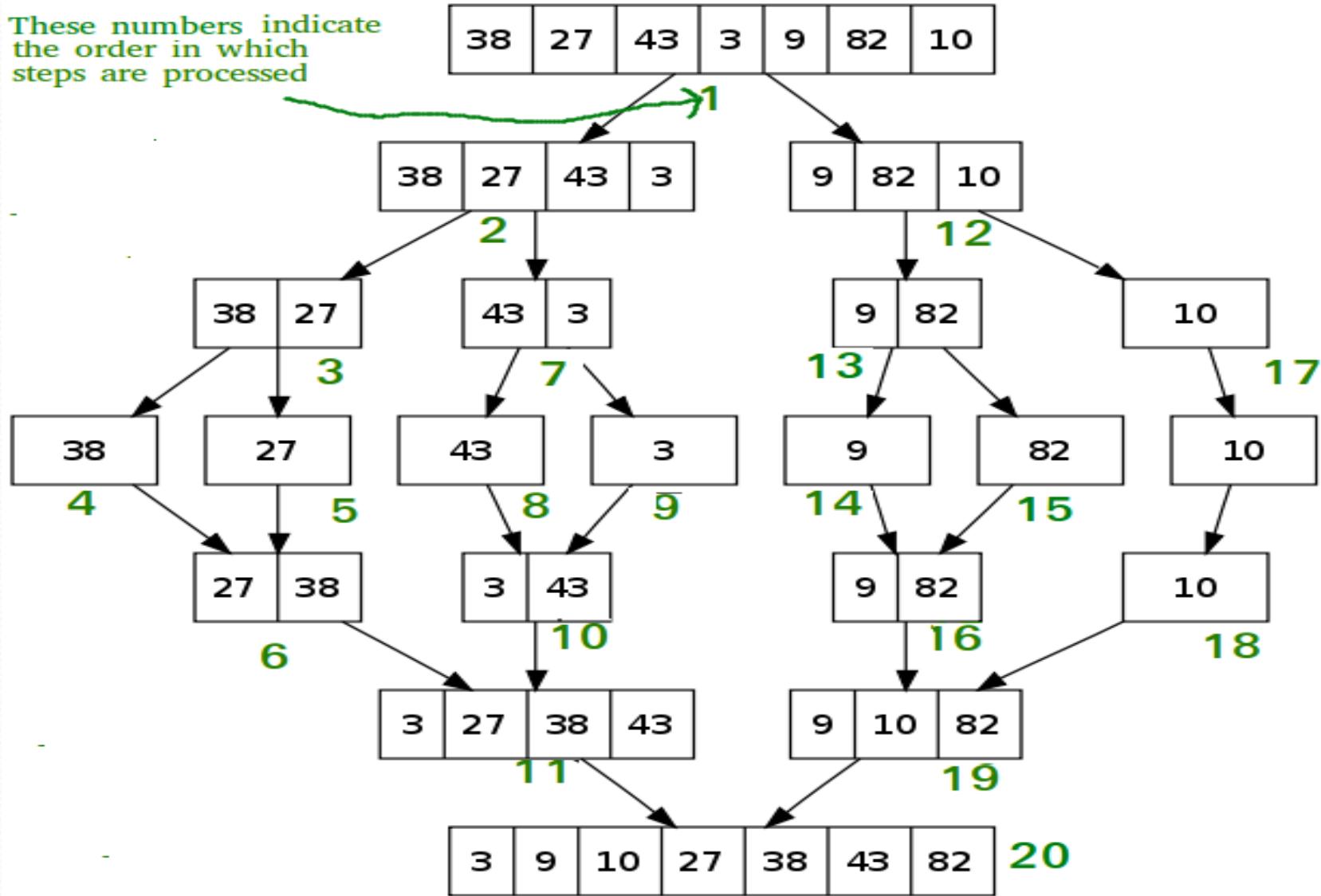
- Brute Force Approach: Using two nested loops
- Using Sorting
- Using Hash Table
- Bit Manipulation algorithm
- Boyer-Moore Majority Vote Algorithm

# Divide and Conquer algorithm for majority element

- A constraint on this problem is that only **equality** is defined on the objects of the array.
- You can check if the array elements are equal but there is no  $>$  or  $<$  relation defined on the elements
- (a) We split the array  $A$  into 2 subarrays  $A_1$  and  $A_2$  of half the size. We choose the majority element of  $A_1$  and  $A_2$ .
- After that we do a linear time equality operation to decide whether it is possible to find a majority element.
- The recurrence therefore is given by  $T(n) = 2T( n/ 2 ) + O(n)$
- The complexity of algorithm comes to  **$O(n \log n)$**

# Divide and Conquer algorithm for majority element

These numbers indicate the order in which steps are processed



# Performance majority element algorithm:

Method	Time Complexity	Space Complexity	Running time (ms)
Hash Table (unordered_map)	$O(n)$	$O(n)$	23
BST (map)	$O(n \log k)$	$O(n)$	19
Randomization	$O(n)$	$O(1)$	19 - 26
Bit vote	$O(32 * n)$	$O(1)$	26 ~ 29
Boyer–Moore vote	$O(n)$	$O(1)$	19 ~ 23
Full sorting	$O(n \log n)$	$O(1)$	26
Partial sorting / partition	$O(n)$ on average	$O(1)$	19
Divide and conquer	$O(n) \sim O(n \log n)$	$O(\log n)$	19

## Suggested Problems to solve

1. Find if any element repeats more than  $N/3$  times
2. Check if an array has a majority element
3. Find majority element in a circular array of 0's and 1's
4. Sort elements based on their frequency
5. Find the element with the second-largest frequency

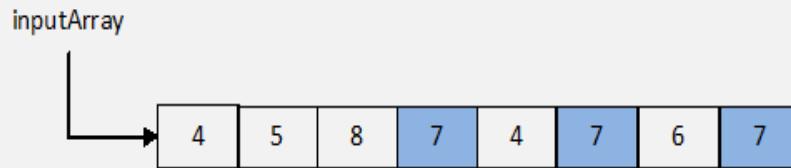
# Las-Vegas Algorithm

- These are randomized algorithms which never produce incorrect results, but whose execution time may vary from one run to another .
- Random choices made within the algorithm are used to establish an expected running time for the algorithm that is, essentially, independent of the input.

Identifying the Repeated Element  
8 Queen Problem

# Identifying the Repeated Element

## The Most Frequent Element In Array



The most frequent element : 7

It's Frequency : 3

# Identifying the Repeated Element

- Consider an array  $a[ ]$  of  $n$  numbers that has  $n/2$  distinct elements and  **$n/2$  copies of another element**. The problem is to identify the repeated element.
- Any deterministic algorithm for solving this problem will need at least  **$(n/2) + 2$**  steps in the worst case.

# Identifying the repeated array number

- a simple and elegant randomized Las Vegas algorithm that takes only **O(log n)** time.
- The algorithm returns the array index of one of the copies of the repeated element

```
1  RepeatedElement(a, n)
2  // Finds the repeated element from a[1 : n].
3  {
4      while (true) do
5      {
6          i := Random() mod n + 1; j := Random() mod n + 1;
7          // i and j are random numbers in the range [1, n].
8          if ((i ≠ j) and (a[i] = a[j])) then return i;
9      }
10 }
```

# Proving the run time Repeated Element

- Any iteration of the **while loop will be successful** in identifying the repeated number if **i** is any one the **n/2** array indices corresponding to the repeated element and **j** is any one of the same **n/2** indices other than **i**.
- The probability that the algorithm quits in any given iteration of the while loop is  $P = \{ (n/2(n/2-1))/n^2\}$ , which is  $\geq 1/5$  for all  $n \geq 10$ .
- Hence the probability that the algorithm does not quit in a given iteration is  $< 4/5$ .
- There fore the probability that the algorithm does not quit in 10 iterations is  $< (4/5)^{10} < 0.1074$ .
- So, Algorithm will terminate in 10 iterations or less with probability  **$\geq 0.8926$**

# Proving the run time RepeatedElement

- The probability that the algorithm does not terminate in 100 iterations is  $< (4/5)^{100} < 2.04 * 10^{-10}$ . That is, almost Certainly the algorithm will quit in 100 iterations or less.
- If  $n$  equals  $2 * 10^6$ , for example, any deterministic algorithm will have to spend at least one million time steps as opposed to the 100 iterations.
- In general, the probability that the algorithm does not quit in the first **c  $\alpha \log n$**  ( $c$  is a constant to be fixed) iterations is
$$< (4/5)^{c \alpha \log n} = n^{c \alpha \log(5/4)}$$
which will be  $< n^{-\alpha}$  if we pick  $c \geq (1/(\log 5/4))$
- Thus the algorithm terminates in  $(1/(\log 5/4)) \alpha \log n$  iterations or less with probability  $\geq (1 - n^{-\alpha})$ .

# RepeatedElement

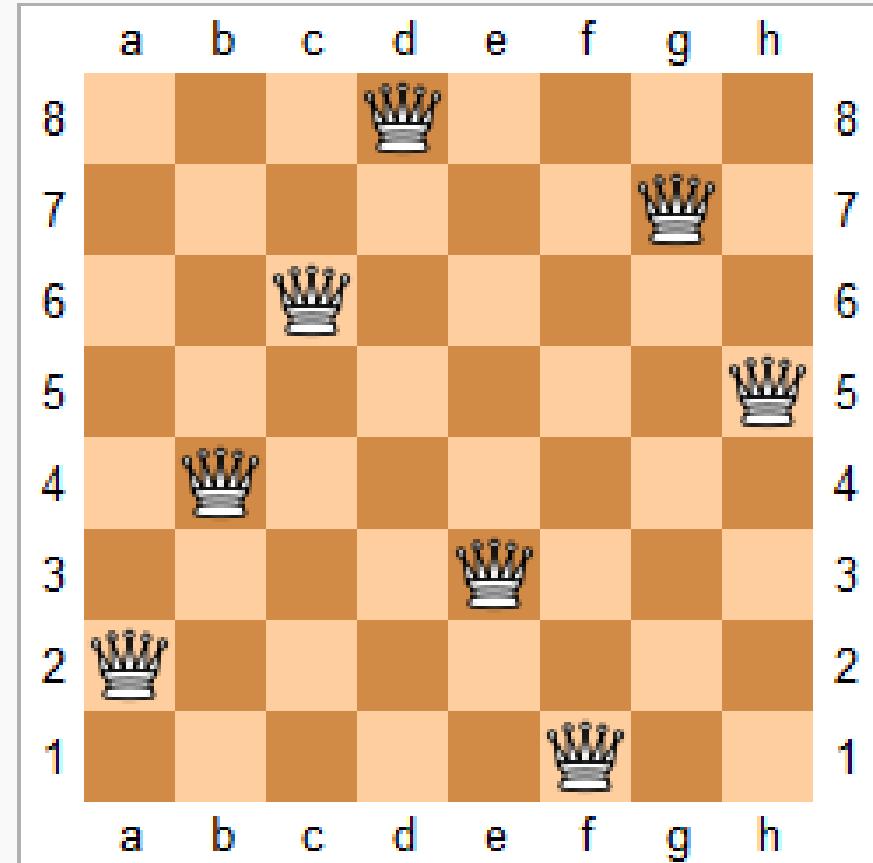
- Since each iteration of the **while loop takes  $o(1)$**  time, the run time of the algorithm is  $O(\log n)$
- Note that this algorithm, if it **terminates**, will always output the correct answer and hence is of the Las Vegas type.
- The ~~following analysis~~ shows that the algorithm will terminate quickly with high probability

Write a program that FINDS majority element from a linear array using randomized algorithm. Show that probability of missing majority element is 0.00097.

## Use randomized algorithm to efficiently find the repeated element

```
1 Question: Given an array with n/2 distinct elements and n/2 repeated elements;  
2     input: [1,2,4,3,7,7,7,7]  
3     output: 7  
4  
5 Non-efficient way O(n^2):  
6  
7     for i in (0...array.length)  
8         j = i+1  
9         while j < array.length  
10            j += 1  
11            return array[i] if array[i] == array[j]  
12        end  
13    end  
14  
15 More efficient way:  
16  
17     until true do  
18         i = random(array.length)  
19         j = random(array.length)  
20  
21         return array[i] if i != j && array[i] == array[j]
```

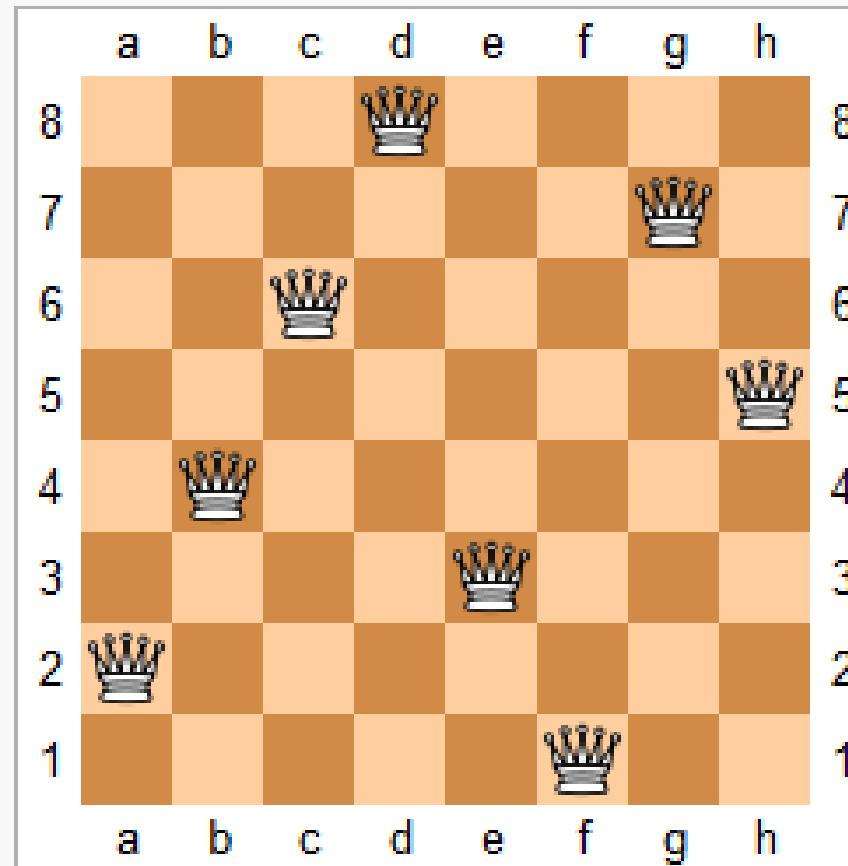
## 8 Queen Problem



One solution to the eight queens puzzle

# The eight queens puzzle

- The **eight queens puzzle** is the **problem** of placing **eight** chess **queens** on an  $8 \times 8$  chessboard so that no two **queens** threaten each other. Thus, a **solution** requires that no two **queens** share the same **row**, **column**, or **diagonal**.



One solution to the eight queens puzzle

# The eight queens puzzle

- The classic solution is via **backtracking**. Place the first queen in the top-left corner.
- Now with some non-attacking queens in place, place a queen in the next row, if it attacks previous queens move it across the row, square by square.
- If there is no suitable position in this row then move the queen in the immediately previous row on one space if possible.
- If neither row leads to a required arrangement, move the queen in the previous row by one space, etc. In fact this returns a solution after examining only 114 of the 2,057 states.

# 8-Queens brute force

## Brute force

- Try all positions
- Mark illegal squares
- Backtrack if dead-end
- 114 setups in total

## Random

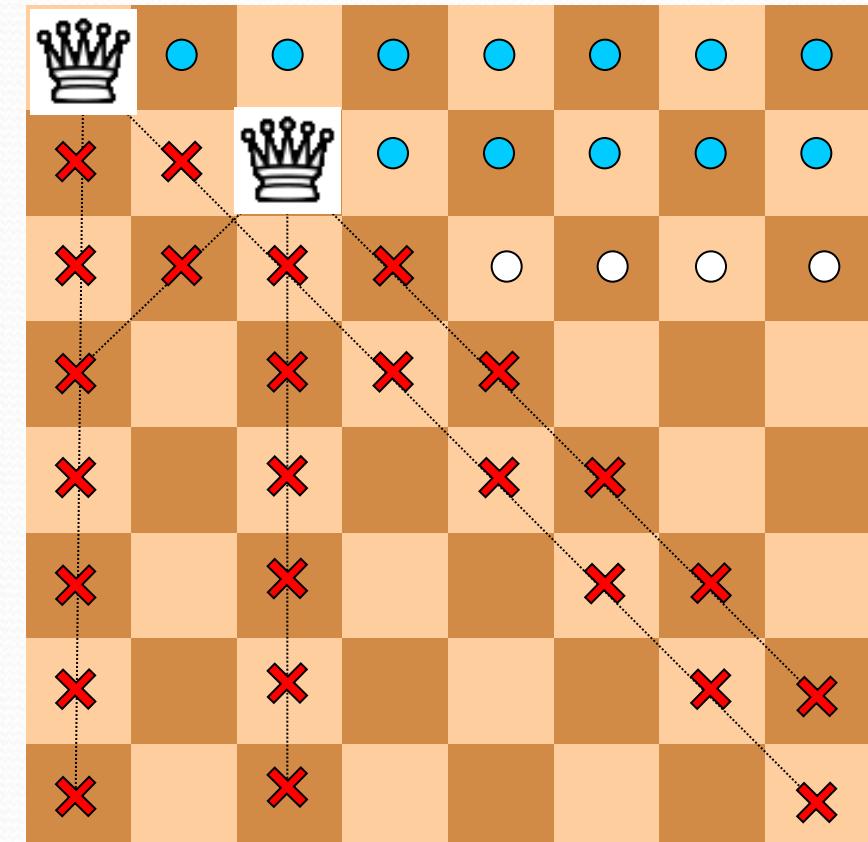
- Select positions randomly
- If dead-end, start over

## Randomized

- Select  $k$  rows randomly
- Rest rows by Brute Force



Where next...?





# Las Vegas Algorithm for n-queen problem

```
repeat
    set valid to true
    shuffle the current permutation vector
    set the two diagonal vectors to all false
    mark the row 0 queen in the two diagonal
        vectors
    for row = 1 to n-1
        set test = row+1
        while this queen is on an in-use diagonal
            if test = n
                set valid to false
                break out of the while loop
            else
                swap positions row and test
                increment test
            end if/else
        end while
        if not valid
            break out of the for loop
        mark this row's queen in the diagonal
            vectors
    end for
loop until valid
```

# Pseudo code

## 8-Queens(k)

FOR i=1 TO k DO // k Queens randomly

    r ← Random[1,8];

    IF Board[i,r]=TAKEN THEN RETURN Fail;

    ELSE ConquerSquare(i,r);

FOR i=k+1 TO 8 DO // Rest by Brute Force

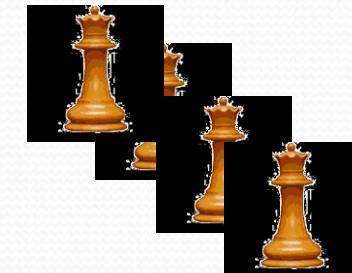
    r←1; found←NO;

    WHILE (r≤8) AND (NOT found) DO

        IF Board[i,r] NOT TAKEN THEN

            ConquerSquare(i,r); found←YES;

    IF NOT found THEN RETURN Fail;



### ConquerSquare(i,j)

Board[i,j] ← QUEEN;

FOR z=i+1 TO 8 DO

    Board[z,j] ← TAKEN;

    Board[z,j-(z-i)] ← TAKEN;

    Board[z,j+(z-i)] ← TAKEN;

# Probability of success

s = processing time in case of success

e = processing time in case of failure

p = probability of success

q = 1-p = probability of failure

$$t = ps + q(e + t) = ps + qe + qt$$

$$\Leftrightarrow t - qt = ps + qe$$

$$\Leftrightarrow t - t + pt = ps + qe$$

$$\Leftrightarrow t = s + \frac{q}{p} \cdot e$$

**Special case:**

$$s=e=1$$

$\Rightarrow$

$$\begin{aligned} t &= 1 + (1-p)/p \\ &= 1/p \end{aligned}$$

## A Las Vegas solution to the 8 Queen problem:

- Suppose that  $k$  rows ( $0 \leq k \leq 8$ ) have been successfully occupied (initially  $k = 0$ ).
- If  $k = 8$  then stop with success. If not then we wish to occupy row  $k + 1$ .
- Calculate all the possible positions on this row i.e. those which are not attacked by already placed queens.
- If there are no such places, then fail. Otherwise choose one such place at random, increment  $k$ , and repeat

## A Las Vegas solution to the 8 Queen problem:

- It can be calculated that the probability of success of this random placement is 0.1293...
- This may seem low, but it means that a solution is obtained 1 time out of 8 simply by guessing! .
- The expected number of states explored is approximately 56, compared to 114 for the deterministic backtracking solution.

	<b>Q1</b>		
			<b>Q2</b>
<b>Q3</b>			
		<b>Q4</b>	

Solution 1

		<b>Q1</b>	
<b>Q2</b>			
			<b>Q3</b>
		<b>Q4</b>	

Solution 2

*Example*

## Finding a Closest Pair



# A Randomized Algorithm for Finding a Closest Pair

- **Algorithm**
- **Input:** A set  $S$  consisting of  $n$  elements  $x_1, x_2, \dots, x_n$ , where  $S \subseteq \mathbb{R}^2$ .
- **Output:** The closest pair in  $S$ .
- **Step 1.** Randomly choose a set  $S_1 = \{ \}$  where  $m = n^{2/3}$ . Find the closest pair of  $S_1$  and let the distance between this pair of points be denoted as .
- **Step 2.** Construct a set of squares  $T$  with mesh-size .

# A Randomized Algorithm for Finding a Closest Pair

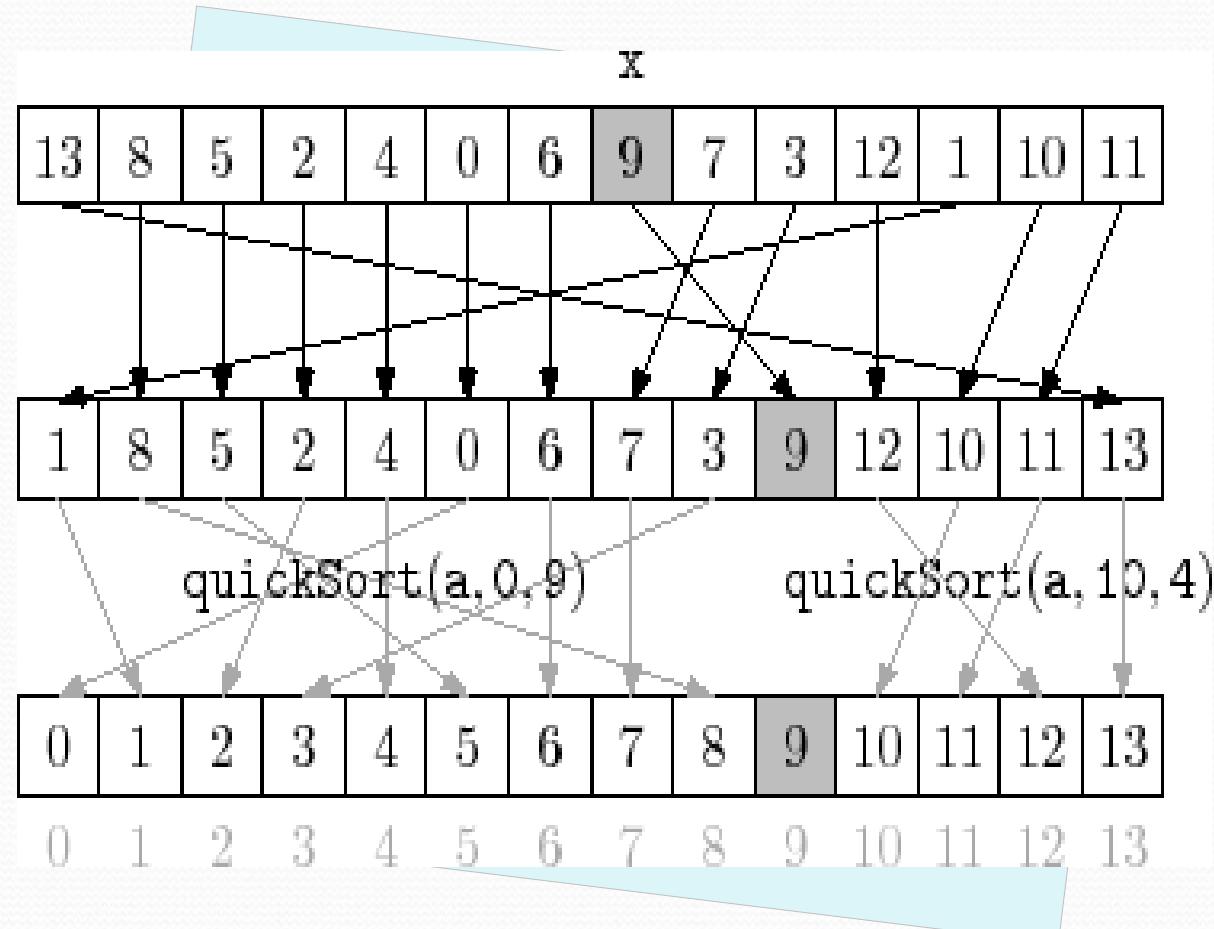
- **Step 3.** Construct four sets of squares  $T_1, T_2, T_3$  and  $T_4$  derived from  $T$  by doubling the mesh-size to 2.
- **Step 4.** For each  $T_i$ , find the induced decomposition  $S = S_1^{(i)} \cup S_2^{(i)} \cup \dots \cup S_{j(i)}^{(i)}$ , where  $S_j^{(i)}$  is a non-empty intersection of  $S$  with a square of  $T_i$ .
- **Step 5.** For each  $x_p, x_q \in S_j^{(i)}$ , compute  $d(x_p, x_q)$ . Let  $x_a$  and  $x_b$  be the pair of points with the shortest distance among these pairs. Return  $x_a$  and  $x_b$  as the closest pair.

# Sherwood Algorithm

Algorithms which always return a result and the correct result, but where a random element increases the efficiency, by avoiding or reducing the probability of worst-case behaviour.

This is useful for algorithms which have a poor worst-case behaviour but a good average-case behaviour, and in particular can be used where embedding an algorithm in an application may lead to increased worst-case behaviour.

## Randomized Quick Sort



# Randomized Quick Sort

- In traditional Quick Sort, we will always pick the first element as the pivot for partitioning.
- The worst case runtime is  $O(n^2)$  while the expected runtime is  $O(n \log n)$  over the set of all input.
- Therefore, some inputs are born to have long runtime, e.g., an inversely sorted list.

# Randomized Quicksort

Given a set  $S$  of  $n$  numbers.

1. If  $|S| \leq 1$ , output the elements of  $S$  and stop.
2. Choose a pivot element  $y$  uniformly at random from  $S$ .
3. Determine the set  $S_1$  of elements  $\leq y$ , and the set  $S_2$  of elements  $> y$ .
4. Recursively apply to  $S_1$ , output the pivot element  $y$ , then recursively apply to  $S_2$ .

# Quick-Sort

- Quick-Sort( $A, 1, n$ )
  - **Divide**
    - $q \leftarrow RANDOM(1, n)$
    - $t = \text{Partition}(A, 1, n, q)$
  - **Conquer**
    - Quick-Sort( $A, 1, t-1$ )
    - Quick-Sort( $A, t+1, n$ )

# Quick-Sort( $A, p, r$ )

- Quick-Sort( $A, p, r$ )
  - if  $p < r$  then
    - $q \leftarrow RANDOM(p, r)$
    - $t = \text{Partition}(A, p, r, q)$
    - Quick-Sort( $A, p, t-1$ )
    - Quick-Sort( $A, t+1, r$ )
- Theorem: Worst Case:  $\Omega(n^2)$
- Theorem: Expected Case:  $O(n \lg n)$

## Partition( $A, p, r, q$ )

Suppose there are  $t-1$  elements in  $A$  that is smaller than  $s = A[q]$ .

Then return  $t$  and reorder  $A$  so that

$A[p..p+t-1] < A[t] = s \leq A[t+1..r]$

- Partition( $A, p, r, q$ )
  - $x = A[q]; A[q] \leftarrow A[r]; A[r] \leftarrow x$       SWAP( $A[q], A[r]$ )
  - $i \leftarrow p - 1$
  - for  $j$   $p$  to  $r-1$ 
    - do if
      - then  $A[j] \leq x$ 
        - $i \leftarrow i + 1; SWAP(A[i], A[j])$
    - $SWAP(A[i + 1], A[r])$
  - return  $i+1$

## Informal analysis

Randomized quicksort has **expected time** (averaged over all choices of pivots) of  $O(n \log n)$ .

Assume we sort the set and divide into four parts, the middle two contain the best pivots.

Each is larger than at least  $\frac{1}{4}$  of the elements and smaller than at least  $\frac{1}{4}$  of the elements.

Choosing an element from the middle two means we split at most  $2 \log 2n$  times.

A random choice will only choose from these middle parts half the time, but this is good enough to give an average call depth of  $2(2 \log 2n)$ .

And hence the expected time of  $O(n \log n)$ .

# Complexity of Quick-Sort

- The Element chosen by RANDOM is called **the pivot element**
- Each number can be a pivot element at most once
- So totally, at most  $n$  calls to Partition procedure
- So the total steps is bounded by a constant factor of the number of comparisons in Partition.

## Compute the total number of comparison in calls to Partition

- When does the algorithm compare two elements?
- When does not the algorithm compare two elements?
- Suppose  $(Z[1], Z[2], \dots, Z[n])$  are the sorted array of elements in  $A$
- that is,  $Z[k]$  is the  $k^{\text{th}}$  smallest element of  $A$ .

## Compute the total number of comparison in calls to Partition

- The reason to use  $Z$  rather than  $A$  directly is that it is hard to locate elements in  $A$  during Quick-Sort because elements are moving around.
- But it is easier to identify  $Z[k]$  because they are in a sorted order.
- We call this type of the analysis scheme: **the backward analysis**.

# Compute the total number of comparisons in calls to Partition

- Under what condition does Quick-Sort compare  $Z[i]$  and  $Z[j]$ ?
- What is the probability of comparison?
- First:  $Z[i]$  and  $Z[j]$  are compared at most once!!!
- Let  $E_{ij}$  be the random event that  $Z[i]$  is compared to  $Z[j]$ .
- Let  $X_{ij}$  be the indicator random variable of  $E_{ij}$ .
  - $X_{ij} = I\{Z[i] \text{ is compared to } Z[j]\}$

# Compute the total number of comparisons in calls to Partition

- So the total number of comparisons is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

- We are interested in

$$E(X) = E\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right)$$

# Compute the total number of comparisons in calls to Partition

- By linearity of expectation, we have

$$\begin{aligned} E(X) &= E\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E(X_{ij}) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr(Z[i] \text{ is compared to } Z[j]) \end{aligned}$$

- So what is  $\Pr(Z[i] \text{ is compared to } Z[j])$ ?

# Compute the total number of comparisons in calls to Partition

- So what is  $\Pr(Z[i] \text{ is compared to } Z[j])$ ?
- What is the condition that
  - $Z[i]$  is compared to  $Z[j]$ ?
- What is the condition that
  - $Z[i]$  is not compared to  $Z[j]$ ?
  - **Answer:** no element is chosen from  $Z[i+1] \dots Z[j-1]$  before  $Z[i]$  or  $Z[j]$  is chosen as a pivot in Quick-Sort
- therefore ...

# Compute the total number of comparisons in calls to Partition

- Therefore

$$\begin{aligned} & \Pr\{Z[i] \text{ is compared to } Z[j]\} \\ &= \Pr\{Z[i] \text{ or } Z[j] \text{ is first pivot from } Z[i] \dots Z[j]\} \\ &= \Pr\{Z[i] \text{ is first pivot from } Z[i] \dots Z[j]\} \\ &\quad + \Pr\{Z[j] \text{ is first pivot from } Z[i] \dots Z[j]\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \end{aligned}$$

# Compute the total number of comparisons in calls to Partition

- By linearity of expectation, we have

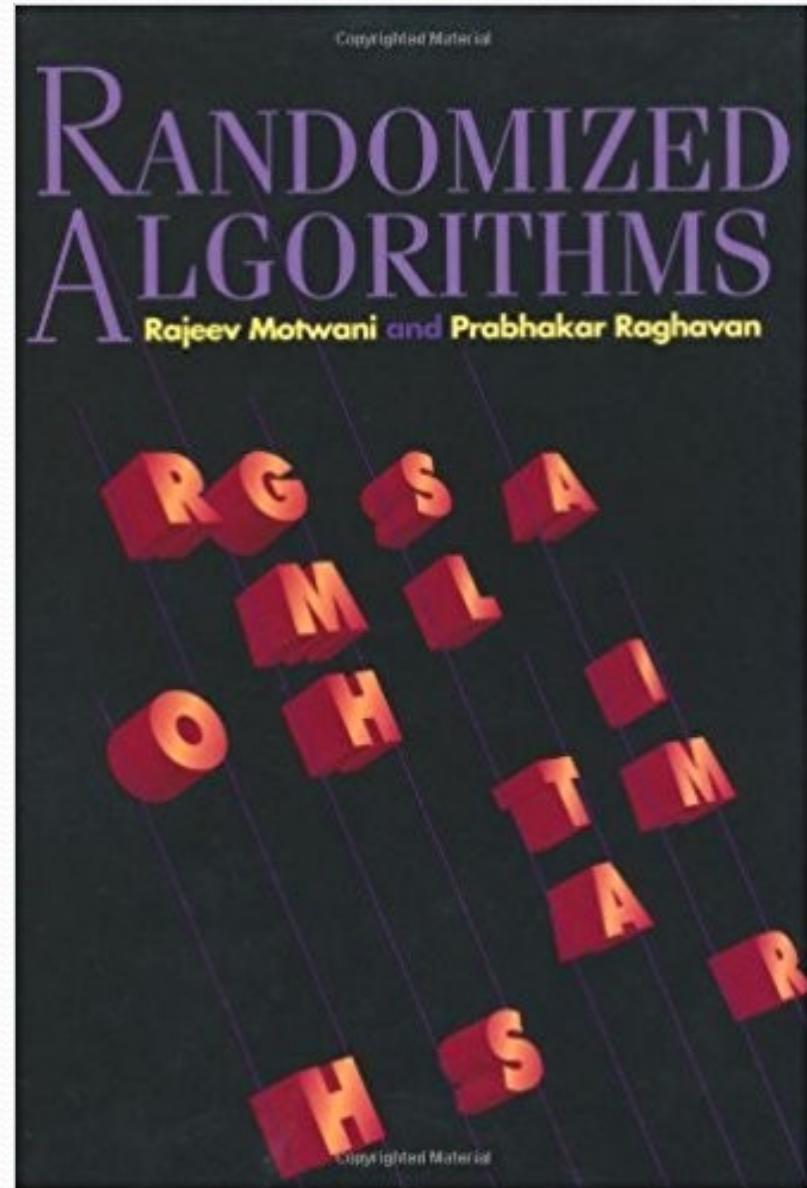
$$\begin{aligned} E(X) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr(Z[i] \text{ is compared to } Z[j]) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n) \end{aligned}$$



# Original Quick-Sort (Tony Hoare)

- Partition with the first element
- **Average-Case Complexity:**
  - Assume inputs come from uniform permutations.
- Our analysis of the Expected time analysis of Random Quick-Sort extends directly.
- Notice the difference of randomized algorithm and average-case complexity of a deterministic algorithm

# Randomized Algorithm Classes



# Randomized Algorithm Classes

- **RP (Randomized Polynomial time)**
  - Monte Carlo algorithm running in worst-case polynomial time
  - If input part of language,  $P(\text{accept}) \geq 0.5$
  - If input *not* part of language,  $P(\text{accept}) = 0$
  - “one-sided error” – could err on ‘YES’
- **co-RP (complement of Randomized Polynomial time)**
  - Monte Carlo algorithm running in worst-case polynomial time
  - If input part of language,  $P(\text{accept}) = 1$
  - If input *not* part of language,  $P(\text{accept}) \leq 0.5$
  - “one-sided error” – could err on ‘NO’
- **ZPP (Zero-error Probabilistic Polynomial time)**
  - Las Vegas algorithm running in expected polynomial time
  - If input part of language,  $P(\text{accept}) = 1$
  - If input *not* part of language,  $P(\text{accept}) = 0$
  - “neither-sided error”

# Randomized Algorithm Classes

- **PP (Probabilistic Polynomial time)**
  - Monte Carlo algorithm running in worst-case polynomial time
  - If input part of language,  $P(\text{accept}) > 0.5$
  - If input *not* part of language,  $P(\text{accept}) < 0.5$
  - “two-sided error” – could err on ‘YES’ and ‘NO’ – not so reliable
- **BPP (Bounded-error Probabilistic Polynomial time)**
  - Monte Carlo algorithm running in worst-case polynomial time
  - If input part of language,  $P(\text{accept}) \geq 0.75$
  - If input *not* part of language,  $P(\text{accept}) \leq 0.25$
  - “two-sided error” – could err on ‘YES’ and ‘NO’, but more useful than PP

# Randomized Algorithms: Recap

- **Faster and simpler:** Even when a fast deterministic algorithm exists, it is more complex
- **Lots of theory** usually needed in the specific application area to prove either:
  - a reasonable one-time successful probability, such that repeating would lead to good overall probability (**Monte Carlo**)
  - a reasonable expected running time (**Las Vegas**)
- **Just as good:** High probability of success just as good as deterministic success:  $P = 0.5^{100}$  is smaller than probability of your hard drive being hit by a meteor...while running the algorithm
- **Problem classes:** Another set to consider

## Applications and Scope:

- Consider a tool that basically does sorting. Let the tool be used by many users and there are few users who always use tool for already sorted array. If the tool uses simple (not randomized) QuickSort, then those few users are always going to face worst case situation. On the other hand if the tool uses Randomized QuickSort, then there is no user that always gets worst case. Everybody gets expected  $O(n \log n)$  time.
- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.
- Number-Theoretic Applications: Primality Testing

# Applications and Scope:

- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: Matrix permanent Counting combinatorial structures.
- .

# Applications and Scope

- Parallel and distributed computing: Deadlock avoidance distributed consensus.
- Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
- Derandomization: First devise a randomized algorithm then argue that it can be derandomized to yield a deterministic algorithm

# Conclusions

- Sometimes you want random numbers for their own sake—for example, you want to shuffle a virtual deck of cards to play a card game
- Sometimes you can use random numbers in a simulation (such as the queue example) to avoid difficult mathematical problems, or when there is no known feasible algorithm (such as the 0-1 knapsack problem)
- Randomized algorithms are basically *experimental*—you almost certainly won't get perfect or optimal results, but you can get “pretty good” results
- Typically, the longer you allow a randomized algorithm to run, the better your results
- A randomized algorithm is what you do when you don't know what else to do
  - As such, it should be in every programmer's toolbox!

# Conclusions

- Randomized algorithms are often easier to design than deterministic algorithms, though often the analysis requires some manipulations of random events or random variables.

## References

Algorithmics: Theory and Practice, G. Brassard and P. Bratley. Prentice-Hall. 1988. Introductory chapter on the topic.

Randomized Algorithms, R. Motwani and P. Raghavan. Cambridge University Press, 1995. Very good and detailed development of the subject.

Algorithms in Java, R. Sedgewick, Addison Wesley. Chapter on random number generators.

## Further Reading on randomized algorithms

- **Algorithmics: Theory and Practice.** G. Brassard and P. Bratley, Prentice-Hall, 1988.  
*Introductory chapter on the topic.*
- **Randomized Algorithms.** R. Motwani and P. Raghavan, Cambridge University Press, 1995.  
*An excellent comprehensive account of the topic. A very good source book on approaches to using randomization in algorithms.*
- **A new approach to the minimum cut problem.** D. Karger and C. Stein, Journal of the ACM, Vo. 43, No. 4, July 1006, pp 601-640.  
*Only for the very brave — a briefer and slightly easier account of this algorithm is contained in Chapter 10, section 10.2, pages 289–295, of Motwani and Raghavan's above book.*

*Thank you for your attention*



## Housing Horrors

Consider the following problem: you have  $k$  housing complexes and  $n$  total people to place into them. The problem is that some pairs of people really, *really* don't like one another and have made it clear that they don't want to be put into the same housing complex. Given a list of  $m$  constraints, design an algorithm to assign people to houses in a way that minimizes the number of constraints violated.

It turns out that this problem is **NP-hard**, so it's probably going to be difficult (if not impossible) to get an exact answer. Fortunately, we can design a randomized algorithm that, on expectation, will get a large fraction of the constraints satisfied.

When facing an **NP-hard** optimization problem, it's often useful, as an initial approach, to guess a totally random answer and see how well it does. Let's see what happens if we do that.

**Algorithm:** Assign people to houses uniformly at random.

# Housing Horrors

To analyze this algorithm, we can set up a random variable, which we'll call  $X$ , that represents the total number of constraints that we can satisfy. If we're interested in the average number of constraints that we can satisfy, we want to know  $E[X]$ . For notational simplicity, let's denote by  $S$  the set of all constraints, and say  $(i, j) \in S$  iff person  $i$  and person  $j$  should be placed separately.

As with many of the other randomized analyses we've seen so far, we'll try to write the random variable  $X$  as the sum of other random variables. That way, we can express  $E[X]$  as the sum of the expected values of other random variables using linearity of expectation. In our case, we can create an indicator random variable  $C_{ij}$  for each constraint saying that person  $i$  and person  $j$  should be kept separate, where  $C_{ij} = 1$  if they are placed into different locations and  $C_{ij} = 0$  otherwise. Then, we have that

$$X = \sum_{(i,j) \in S} C_{ij}$$

Therefore, by linearity of expectation:

$$\begin{aligned} E[X] &= E\left[\sum_{(i,j) \in S} C_{ij}\right] \\ &= \sum_{(i,j) \in S} E[C_{ij}] \\ &= \sum_{(i,j) \in S} P(i \text{ and } j \text{ are placed separately}) \end{aligned}$$

# Housing Horrors

So now if we can determine the probability that person  $i$  and person  $j$  are placed in separate housing, we can determine  $E[X]$ . Since we're assigning people completely randomly, the probability that  $i$  and  $j$  are placed into the same house is  $1/k$ . Therefore, the probability that  $i$  and  $j$  are not placed into the same house is  $(k-1)/k$ . Therefore:

$$\begin{aligned} E[X] &= \sum_{(i,j) \in S} P(i \text{ and } j \text{ are placed separately}) \\ &= \sum_{(i,j) \in S} \frac{k-1}{k} \\ &= \frac{m(k-1)}{k} \end{aligned}$$

In other words, on expectation, we can respect a  $(k-1)/k$  fraction of the total number of constraints. The maximum possible number of constraints we can satisfy is  $m$ , so our algorithm will always produce an answer that is within a  $(k-1)/k$  fraction of optimal. The above line of reasoning is pretty much what we could write as a correctness proof (asserting we were within a factor of  $(k-1)/k$  of optimal on expectation), assuming that we clean up some of the informal language.

# Majority Element Problem

Suppose you are interested in solving the majority element problem from the previous problem set using a randomized algorithm. Recall that in this problem, you want to try to determine whether there are strictly more than  $n / 2$  elements in an array with the same value, subject to the constraint that you can only learn whether two elements are equal or different.

Design an  $O(n)$ -time randomized algorithm where if there is a majority element, your algorithm returns one with probability at least  $1 - 10^{-9}$ , and if there is no majority element, your algorithm always returns “no majority.”

## Solution to The Majority Element Problem Revisited

The key insight behind the solution to this problem is that we can start with an algorithm that has a modest chance of success, then iterate it enough times to drive the error rate down to less than one in a billion. This particular algorithm works by starting with just over a 50% chance of success, then iterating it 30 times to amplify the probability to at least  $1 - 10^9$  (since  $(1 / 2)^{30} \approx 10^{-9}$ ).

**Algorithm:** Repeat this process 30 times: choose an element uniformly at random, then compare that element to every other element in the array. If more than  $n / 2 - 1$  elements compare equal to the element, return that element. If after 30 iterations this process has not returned a majority, return “no majority.”

## Correctness:

*Theorem:* If there is no majority element, our algorithm always returns “no majority.”

*Proof:* The algorithm only returns an element  $x$  if it finds that more than  $n / 2 - 1$  elements in the array are equal to  $x$ , meaning that more than  $n / 2$  total elements are equal to the element  $x$ . Consequently,  $x$  is a majority element. Therefore, if there is no majority element, no element chosen will be a majority, so after 30 iterations the algorithm will return “no majority.” ■

*Theorem:* If there is a majority element, it will be returned with probability at least  $1 - 10^{-9}$ .

*Proof:* Let  $\mathcal{E}$  be the event that our algorithm does not return a majority when one exists. Let  $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_{30}$  be the events that on iterations 1, 2, 3, ..., 30, the algorithm does not find a majority element. This means that

$$\mathcal{E} = \bigcap_{i=1}^{30} \mathcal{E}_i$$

Using the chain rule for conditional probability:

$$P(\mathcal{E}) = P(\mathcal{E}_1) \cdot P(\mathcal{E}_2 | \mathcal{E}_1) \cdot P(\mathcal{E}_3 | \mathcal{E}_2, \mathcal{E}_1) \cdots P(\mathcal{E}_{30} | \mathcal{E}_{29}, \mathcal{E}_{28}, \dots, \mathcal{E}_1)$$

Assuming iterations  $k - 1, k - 2, \dots, 1$  of the algorithm all failed to find a majority element, the probability that iteration  $k$  fails to return a majority element is the probability that on iteration  $k$  the algorithm chooses a non-majority element. Since a majority element exists and we choose elements uniformly at random, this occurs with probability less than  $1/2$ . Consequently, we have

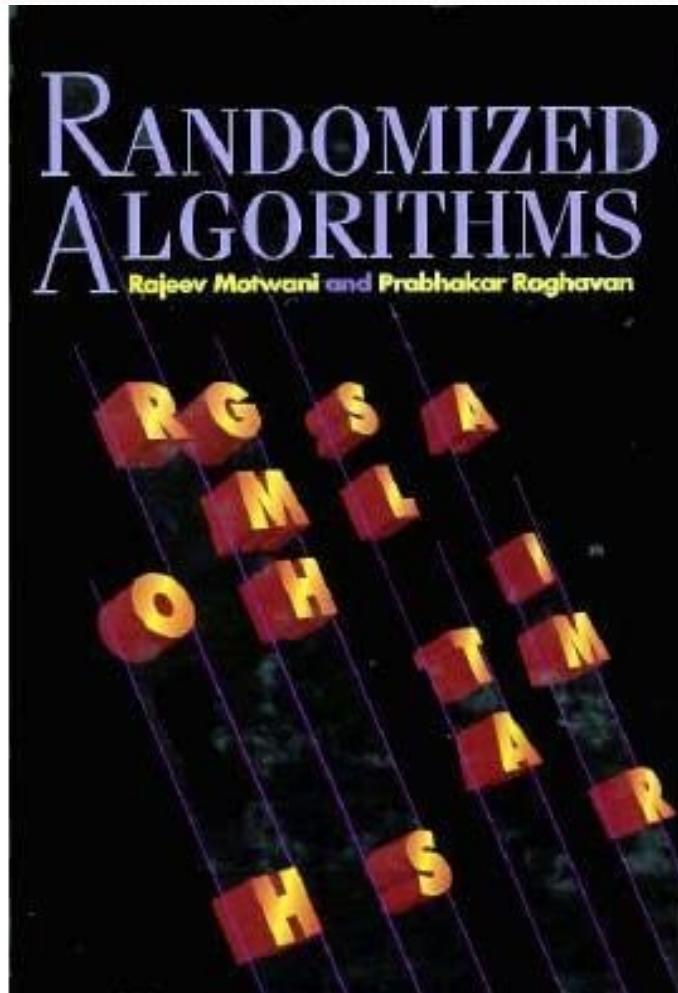
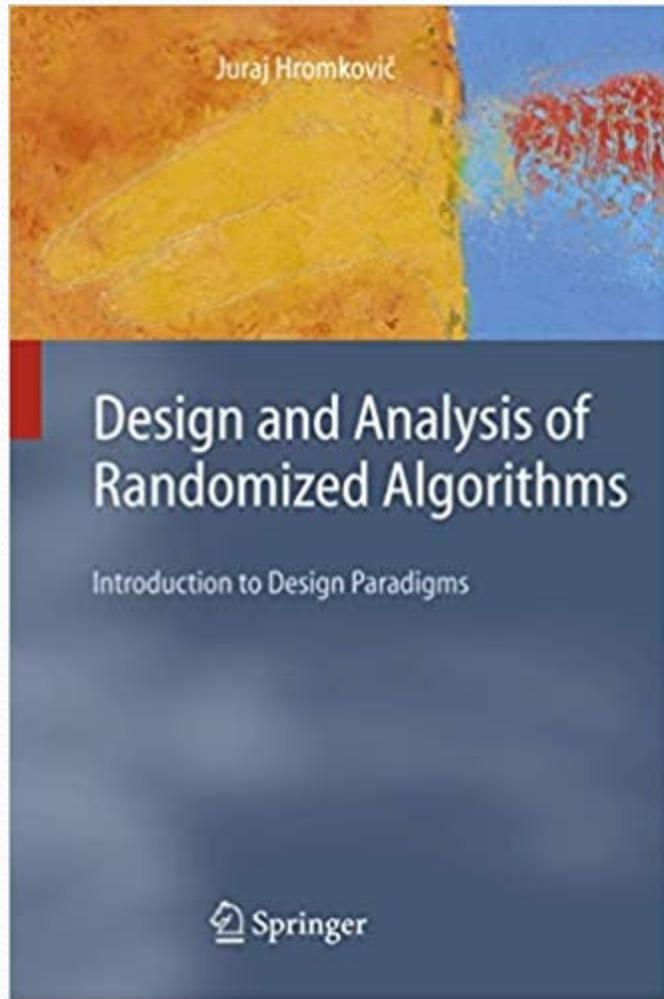
$$P(\mathcal{E}) < \prod_{i=1}^{30} \frac{1}{2} = \left(\frac{1}{2}\right)^{30}$$

Since  $(1/2)^{30} < 10^{-9}$ , this means that the probability that the algorithm does not find a majority if one exists is at least  $1 - (1/2)^{30} > 1 - 10^{-9}$ . ■

### Runtime:

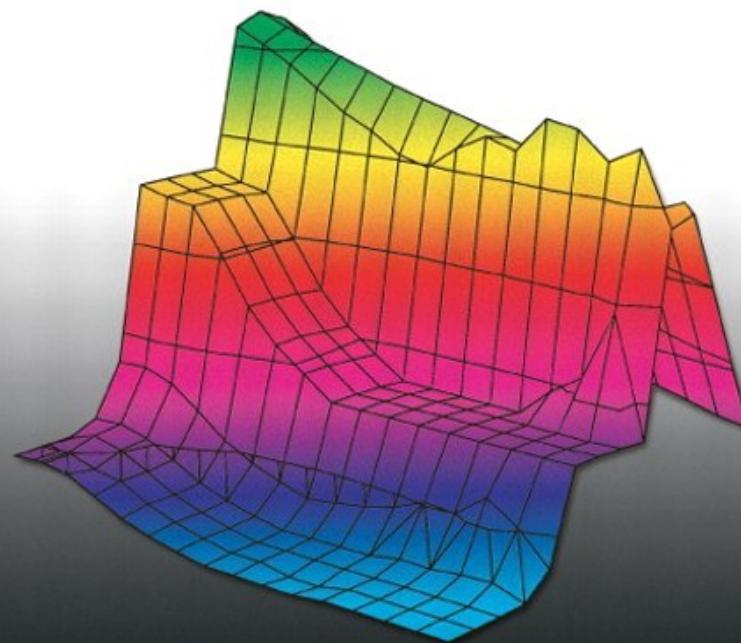
The algorithm runs for 30 iterations and on each iteration does  $\Theta(n)$  work picking a random element and comparing each element against it. Consequently, the total runtime is  $O(n)$ .

# Suggested Reading



# COMPUTER SYSTEMS

A PROGRAMMER'S  
PERSPECTIVE



Randal E. Bryant and David O'Hallaron