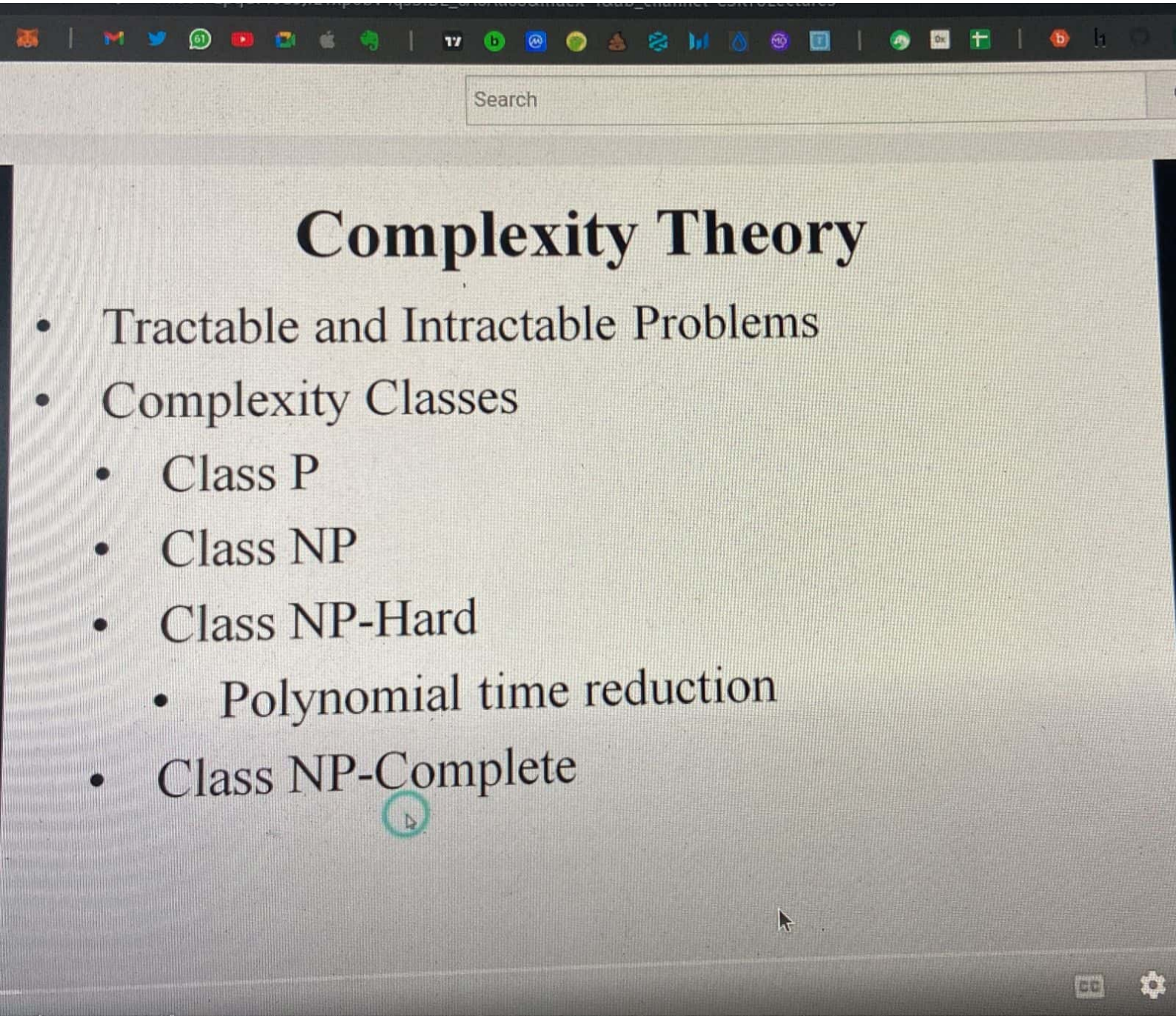
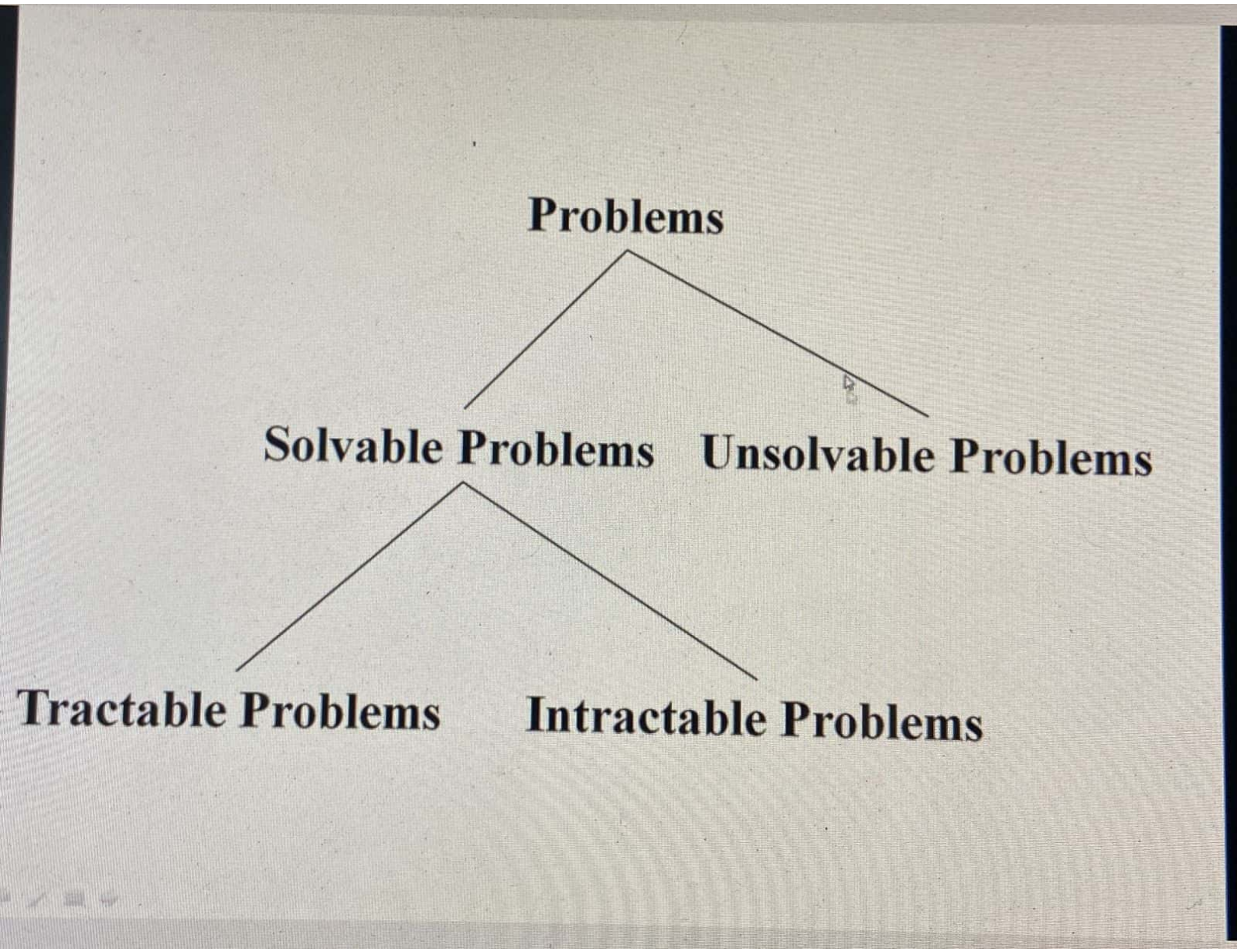




Tractable Problems

- Algorithm complexity is polynomial
- Tractable problem solutions are implemented in practice.
- Example:
  - **PATH problem**: Given directed graph G, determine whether a directed path exists from vertex s to vertex t.
    - Time complexity = O(n)

      where n – total number of vertices

Intractable Problems

- Algorithm complexity is exponential
- An intractable problem has a faster complexity growth as compared to tractable problems.
- Example:
  - Knapsack Problem
    - Time Complexity = $O(2^n)$
  - Traveling Salesman Problem
    - Time Complexity = $O(n^2 2^n)$

Complexity Classes

- 4 complexity classes
  - P
  - NP
  - NP-Hard
  - NP-Complete

Class P

- Class P problems are solvable in polynomial time.
- Time Complexity = $O(n^k)$.

  n: size of input k: constant

- Example:
  - **PATH Problem:** Given directed graph G, determine whether a directed path exists from s to t.

Class P

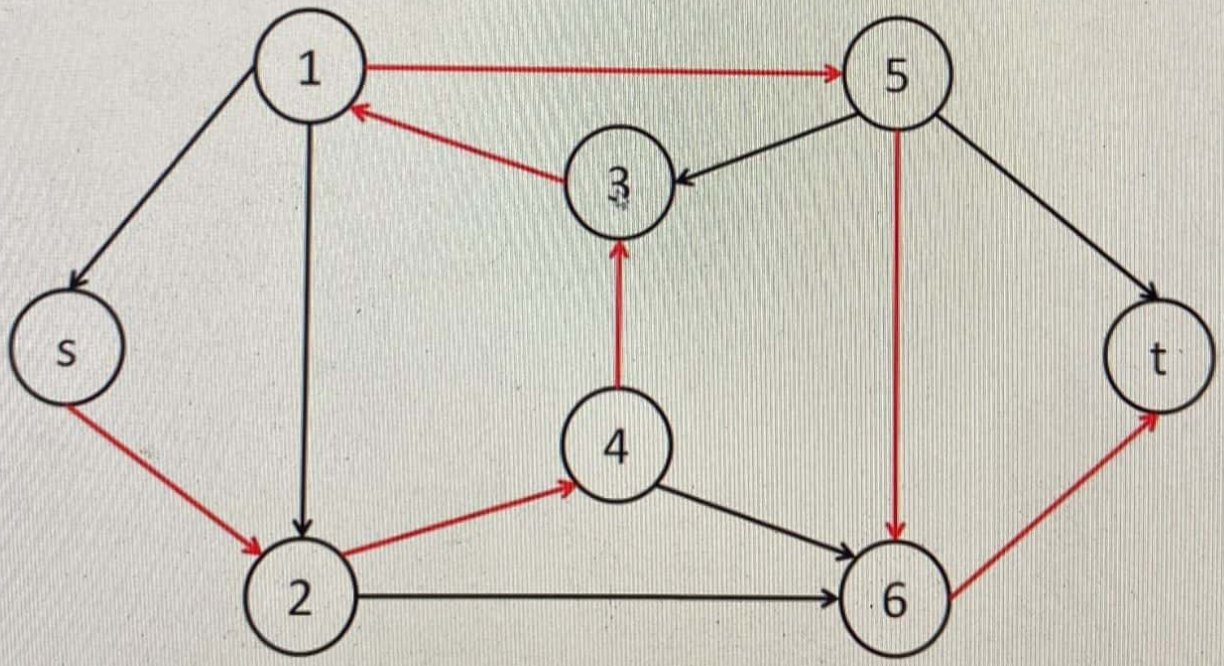
- **More Examples of P Problem:**
  - Single Source Shortest Path problem using Dijkstra's Greedy method.
  - Multistage Graph problem implemented using forward or backward dynamic programming.
  - Minimum cost spanning tree using Prim's or Kruskal's method.
  - Network flow problem using Ford-Fulkerson algorithm.

Class NP

- Some problems can only be solved in exponential or factorial time.
- Suppose we can able to verify these problems in polynomial time.
- Then these problems are called NP problems.

Class NP – Example(HAMPATH Problem)

- A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once



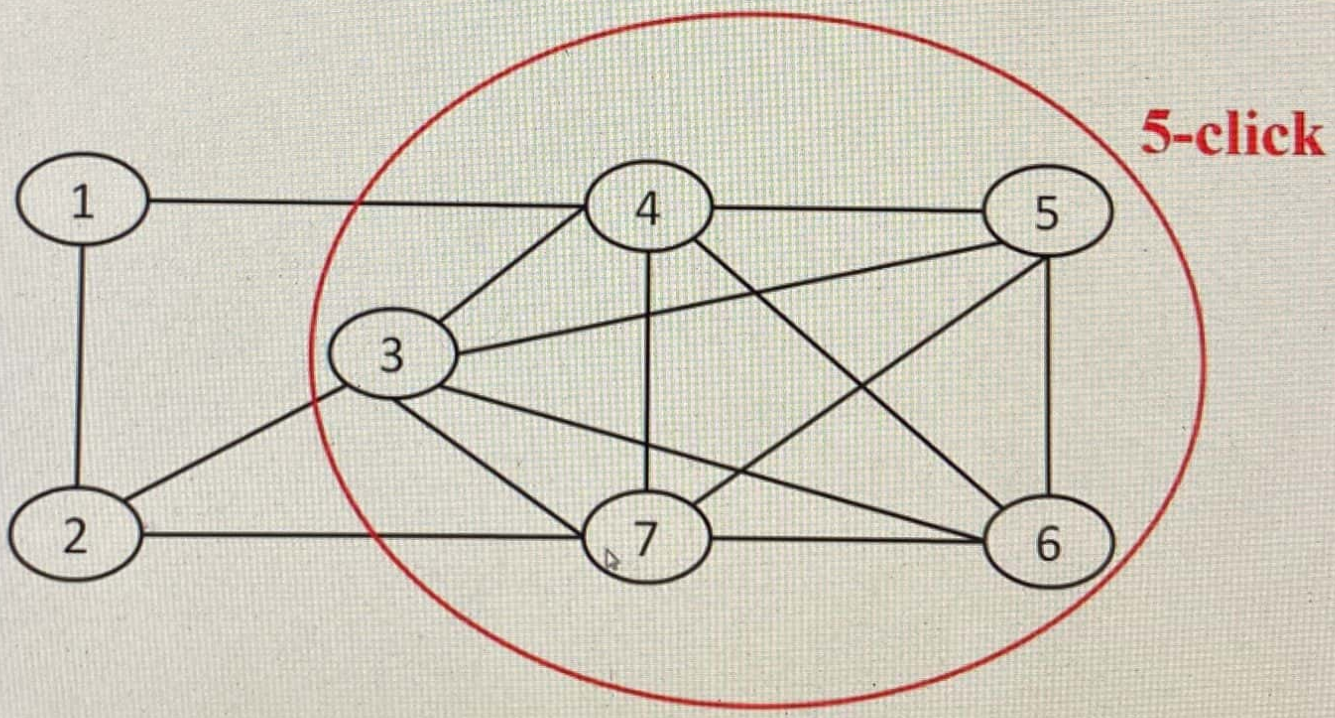
Class NP – Example(HAMPATH Problem)

- The **HAMPATH problem** is to test whether a graph contains a hamiltonian path connecting 2 specified nodes.

- There is no polynomial solution.
- HAMPATH problem have a feature called polynomial verifiability.

Class NP – Example(CLIQUE Problem)

- A clique in an undirected graph is a sub-graph where every two nodes are connected by an edge.



Class NP – Example(CLIQUE Problem)

- **CLIQUE Problem**: To determine whether a graph contains a clique of specified size.

- There is no polynomial time algorithm.
- We can verify this in polynomial time

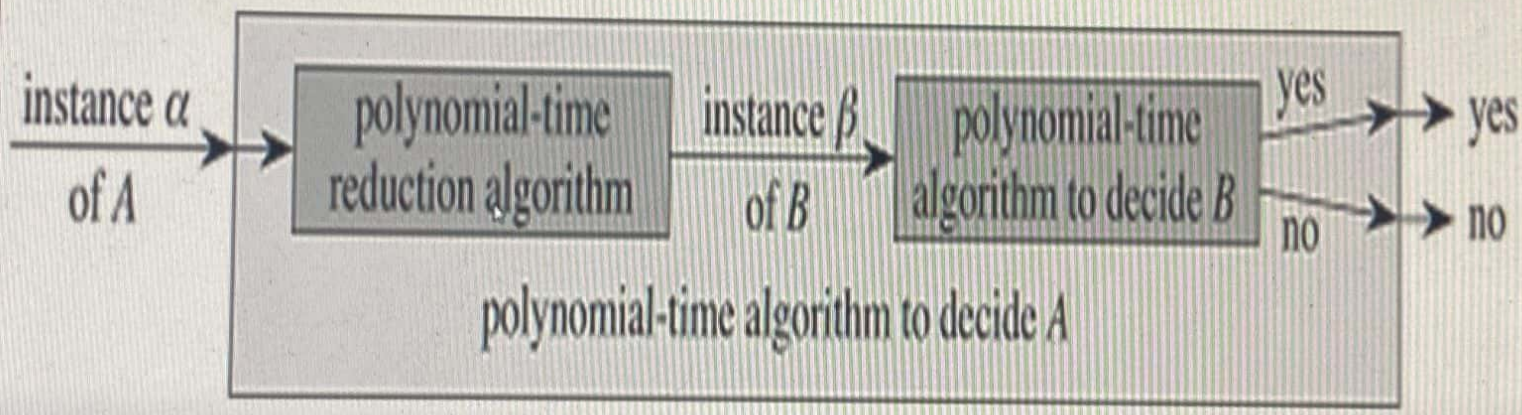
Class NP – Example(CLIQUE Problem)

- **CLIQUE Verifier Algorithm**
  - **Inputs:** <G,k,V'>
  - **Algorithm**
    1. Test whether V' is a set of k vertices in the graph G
    2. Check whether for each pair (u,v) ∈ V', the edge (u,v) belongs to E.
    3. If both steps pass, then accept. Otherwise reject.
  - This algorithm will execute in polynomial time.
  - Therefore **CLIQUE problem is a NP problem.**

Polynomial Time Reduction

- Suppose we have 2 decision problems
  1. A
     - We like to solve in polynomial time
     - Instance of A is α
  2. B
     - It having a polynomial time algorithm.
     - Instance of B is β

Polynomial Time Reduction



13:06 / 19:19

Polynomial Time Reduction

- Suppose that we have a procedure that transforms α to β with the following characteristics.
  1. The transformation takes polynomial time
  2. The answers are the same. That is, the answer for α is "yes" iff the answer for β is also "yes".

- Such a procedure is called **polynomial time reduction**.

CLASS NP-Hard

- If a decision problem X is NP-Hard if every problem in NP is polynomial time reducible to X.

  $Y \leq_p X$ Y is all NP problems

- X is as hard as all problems in NP

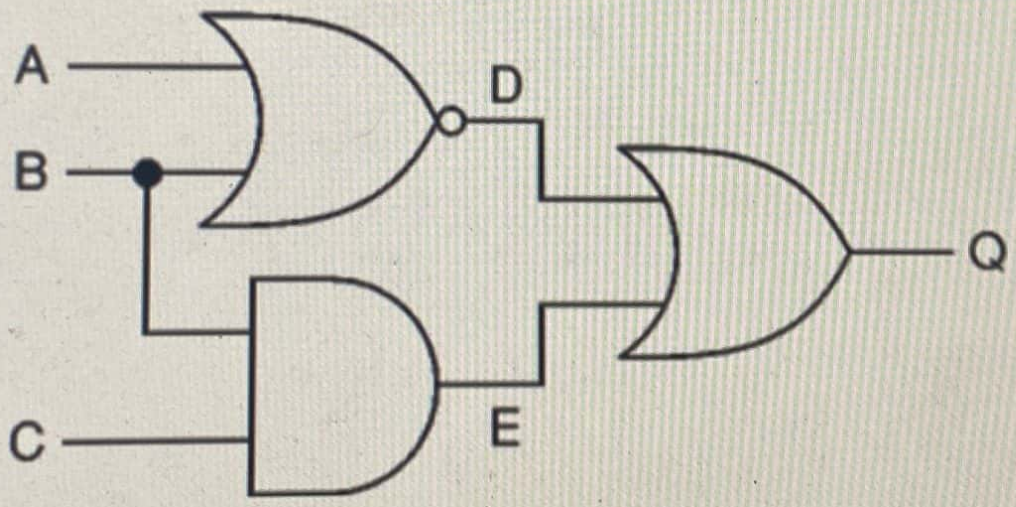
- If X can be solved in polynomial time, then all problems in NP can also solved in polynomial time.

CLASS NP-Complete

- If the problem is **NP as well as NP-Hard**, then that problem is NP Complete.

CLASS NP-Complete

- **Examples:**
  - **CIRCUIT-SAT problem**: Given a Boolean circuit C, is there an assignment to the variables that causes the circuits to output 1?



NP-Complete Proof

- **Steps:**

1. Prove that the given problem is NP
   - Write a polynomial time verification algorithm.

2. Prove that the given problem is NP Hard
   - Write a polynomial time reduction algorithm from any NP problem to the given problem.

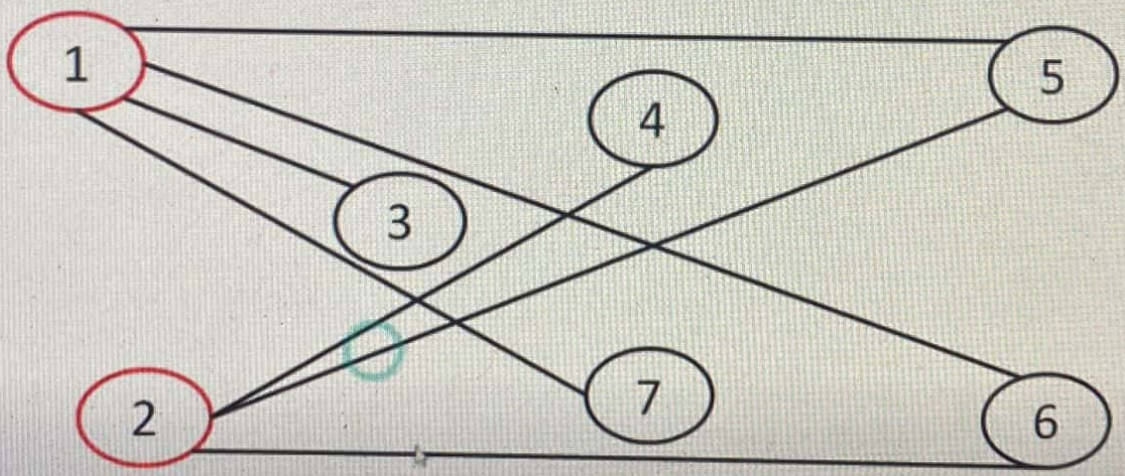
Prove that CLIQUE Problem is NP-Complete

- **Step 1:** Write a polynomial time verification algorithm to prove that the given problem is NP
- **Inputs:** <G,k,V'>
- **Verifier Algorithm**:
  1. Test whether V' is a set of k vertices in the graph G
  2. Check whether for each pair (u,v) $\in$ V', the edge (u,v) belongs to E.
  3. If both steps pass, then accept. Otherwise reject.

- This algorithm will execute in polynomial time. Therefore **CLIQUE problem is a NP problem.**

Prove that VERTEX COVER Problem is NP-Complete

- G=(V, E)



Prove that VERTEX COVER Problem is NP-Complete

- **Step 1:**Write a polynomial time verification algorithm to prove that the given problem is NP
- **Inputs:** <G, k,V'>
- **Verifier Algorithm**:
  1. count = 0
  2. for each vertex v in V' remove all edges adjacent to v from set E
     1. increment count by 1
  3. if count = k and E is empty then the given solution is correct
  4. else the given solution is wrong
- This algorithm will execute in polynomial time. Therefore **VERTEX COVER problem is a NP problem.**

Prove that VERTEX COVER Problem is NP-Complete

- **Step 2:**Write a polynomial time reduction algorithm from CLIQUE problem to VERTEX COVER problem
- **Algorithm**

  Inputs: <G=(V,E), k>

  1. Construct a graph G', which is the complement of Graph G
  2. If G' has a vertex cover of size |V| - k, then G has a clique of size k.

- This reduction algorithm(CLIQUE to VERTEX COVER) is a polynomial time algorithm
- So VERTEX COVER problem is NP Hard

Approximation Algorithm

- **Approximate Solution**: A feasible solution with value close to the value of optimal solution is called an approximate solution
- **Approximation Algorithms**: An algorithm that returns near optimal solution is called Approximation Algorithm.
- Approximation algorithms have two main properties:
  - They run in polynomial time
  - They produce solutions close to the optimal solutions
- Approximation algorithms are useful to give approximate solutions to NP complete optimization problems.
- It is also useful to give fast approximations to problems that run in polynomial time.

Approximation Algorithm

- **Approximation Ratio / Approximation Factor**
  - For given problem, C is the result obtained by the algorithm and C* is the optimal result.
  - The approximation ratio of an algorithm is the ratio between the result obtained by the algorithm and the optimal result.
  - For **maximization problem**, $0 < C \leq C^*$, Approximation Ratio = C*/C
  - For **minimization problem**, $0 < C^* \leq C$, Approximation Ratio = C/C*

Approximation Algorithm

- **Approximation Ratio / Approximation Factor**
  - The approximation ratio of an approximation algorithm is never less than 1.
  - Approximation ratio and computational time are inversely proportional.
  - Approximation ratio and quality of the result are also inversely proportional

Search

Approximation Algorithm

- **k-Approximation Algorithm**: An algorithm with approximation ratio k is called a k-approximation algorithm.
- 1-approximation algorithm produces an optimal solution
- An approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

2:00 / 21:33

Search

Approximation Algorithm

- **Examples**
  - Bin Packing Algorithm
  - Graph Coloring Algorithm

2:07 / 21:33

Bin Packing Approximation Algorithm

- Given n items of different weights and bins each of capacity c, assign each item to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity

- The lower bound on minimum number of bins required can be given as :

  Min no. of bins >= Ceil ((Total Weight) / (Bin Capacity))

Bin Packing Approximation Algorithm

- **Applications**
  - Loading of containers like trucks.
  - Placing data on multiple disks.
  - Job scheduling.
  - Packing advertisements in fixed length radio/TV station breaks.
  - Storing a large collection of music onto tapes/CD's, etc.

Bin Packing Approximation Algorithm

- **Different Bin Packing Approximation Algorithms**
  - **Online Algorithm**
    - Next Fit Algorithm
    - First Fit Algorithm
    - Best Fit Algorithm
    - Worst Fit Algorithm
  - **Offline Algorithm**
    - First Fit Decreasing Algorithm
    - Best Fit Decreasing Algorithm

Next Fit - Example

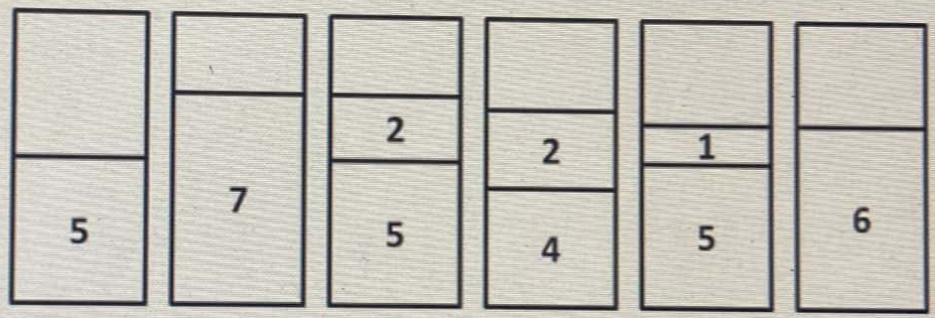
- Apply Next Fit Bin packing approximation algorithms on the following items with bin capacity=10. Assuming the sizes of the items be {5, 7, 5, 2, 4, 2, 5, 1, 6}

- Minimum number of bins >= Ceil ((Total Weight) / (Bin Capacity))

  = Ceil (37 / 10) = 4

Next Fit - Example

{5, 7, 5, 2, 4, 2, 5, 1, 6}



Number of bins required = 6

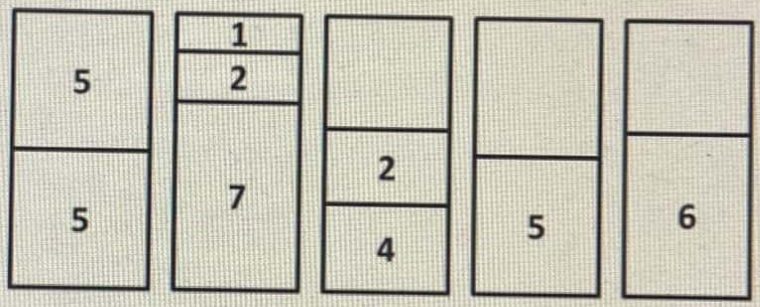
Next Fit Algorithm

- If the current item is fit in the same bin as the last item, then insert it in the same bin.
- Otherwise use the new bin

- **Time Complexity**
  - Best case Time Complexity = $\theta(n)$
  - Average case Time Complexity = $\theta(n)$
  - Worst case Time Complexity = $\theta(n)$

First Fit - Example

{5, 7, 5, 2, 4, 2, 5, 1, 6}



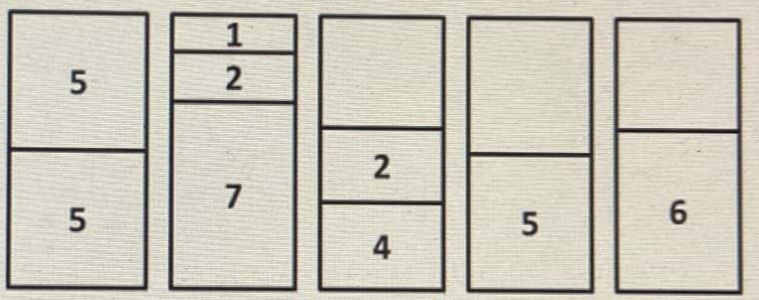
First Fit Algorithm

- Scan the previous bins in order and find the first bin that it fits.
- If such bin exists, place the item in that bin
- Otherwise use a new bin.

- **Time Complexity**
  - Best case Time Complexity = $\theta(n \log n)$
  - Average case Time Complexity = $\theta(n^2)$
  - Worst case Time Complexity = $\theta(n^2)$

Best Fit - Example

{5, 7, 5, 2, 4, 2, 5, 1, 6}



Number of bins required = 5

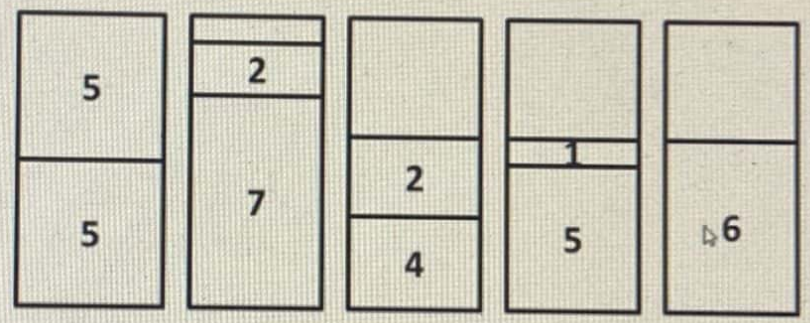
Best Fit Algorithm

- Scan the previous bins and find a bin that having minimum remaining capacity that can accommodates this item.
- If such bin exists, place the item in that bin
- Otherwise use a new bin

- **Time Complexity**
  - Best case Time Complexity = $\theta(n \log n)$
  - Average case Time Complexity = $\theta(n^2)$
  - Worst case Time Complexity = $\theta(n^2)$

Worst Fit - Example

{5, 7, 5, 2, 4, 2, 5, 1, 6}



Worst Fit Algorithm

- Scan the previous bins and find a bin that having maximum remaining capacity that can accommodates this item.
- If such bin exists, place the item in that bin
- Otherwise use a new bin

- **Time Complexity**
  - Best case Time Complexity = $\theta(n \log n)$
  - Average case Time Complexity = $\theta(n^2)$
  - Worst case Time Complexity = $\theta(n^2)$

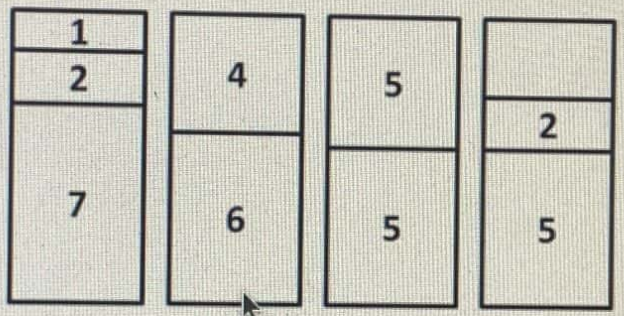
First Fit Decreasing - Example

- Apply First Fit Decreasing Bin packing approximation algorithms on the following items with bin capacity=10. Assuming the sizes of the items be {5, 7, 5, 2, 4, 2, 5, 1, 6}

- Arrange the items in the decreasing order of their size {7, 6, 5, 5, 5, 4, 2, 2, 1}

First Fit Decreasing - Example

{7, 6, 5, 5, 5, 4, 2, 2, 1}



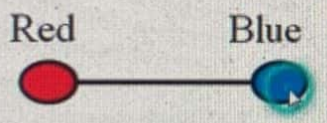
Number of bins required = 4

Graph Coloring

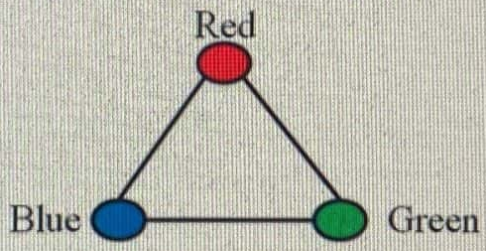
- **Different Graph coloring problems**
  - Vertex coloring
  - Edge coloring
  - Face coloring

Vertex Coloring

- Assignment of colors to vertices in a graph such that no two adjacent vertices share the same color
- A graph is 0-colorable iff $V = \emptyset$
- A graph is 1-colorable iff $E = \emptyset$

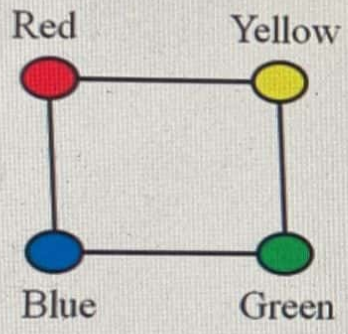


2 Colorable graph
$\chi(G) = 2$

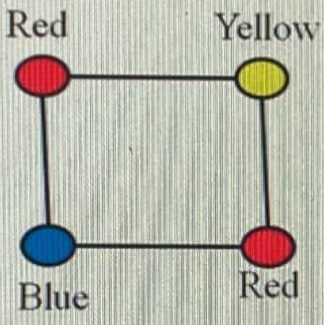


3 Colorable graph
$\chi(G) = 3$

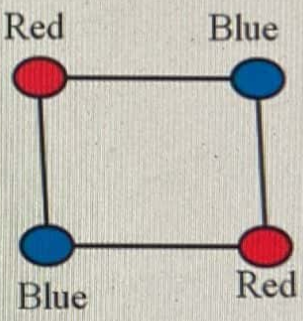
Vertex Coloring



4 Colorable graph
$\chi(G) = 4$



3 Colorable graph
$\chi(G) = 3$



2 Colorable graph
$\chi(G) = 2$

Vertex Coloring

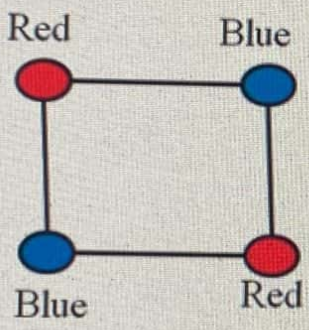
- $\chi(G) = 1$, if G is a null graph. A null graph is a graph that contains vertices but no edges.
- All other graphs $\chi(G) >= 2$.

- **Four Color Theorem**: For Every Planar graph, the chromatic number is less than or equal to 4.

- A graph is k-colorable if it has k colors.

Vertex Coloring

- **Chromatic Number**: It is the minimum number of colours with which a graph can be coloured.

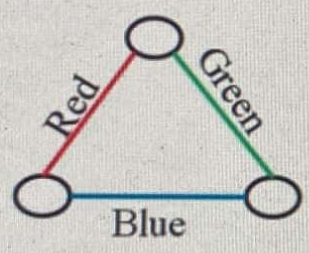


Chromatic number = 2

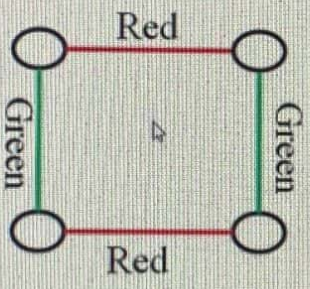
- A graph whose chromatic number is k, then it is called **k-chromatic graph**

Edge Coloring

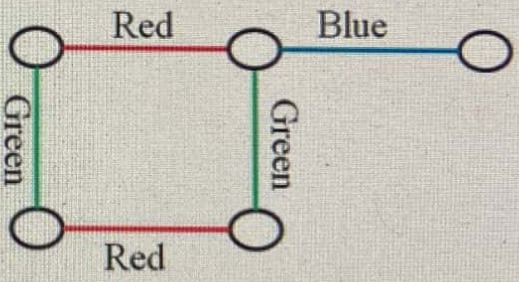
- Given a graph G=(V,E), assign a color to each edges so that no two adjacent edges share the same color.



$\chi(G) = 3$



$\chi(G) = 2$

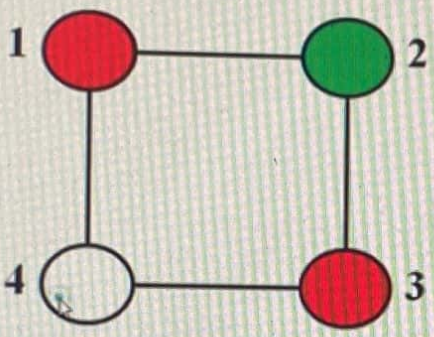


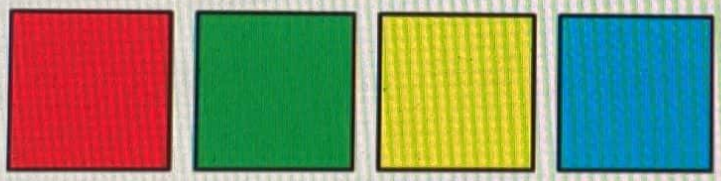
$\chi(G) = 3$

Graph Coloring Approximation Algorithm

- Graph coloring problem is a NP-Complete problem. But there are approximation algorithms
- Important graph coloring problem is vertex coloring.
- Following is the greedy approximation algorithm for vertex coloring

Graph Coloring Approximation Algorithm





Colors 1 2 3 4

CS KTU Lectures

Graph Coloring Approximation Algorithm

Algorithm Approximate_Graph_Coloring(G, n)

1. for i=1 to n do
    1. for c=1 to n do
        1. If no vertex adjacent to v_i has color c
            1. Color v_i with c
            2. Break

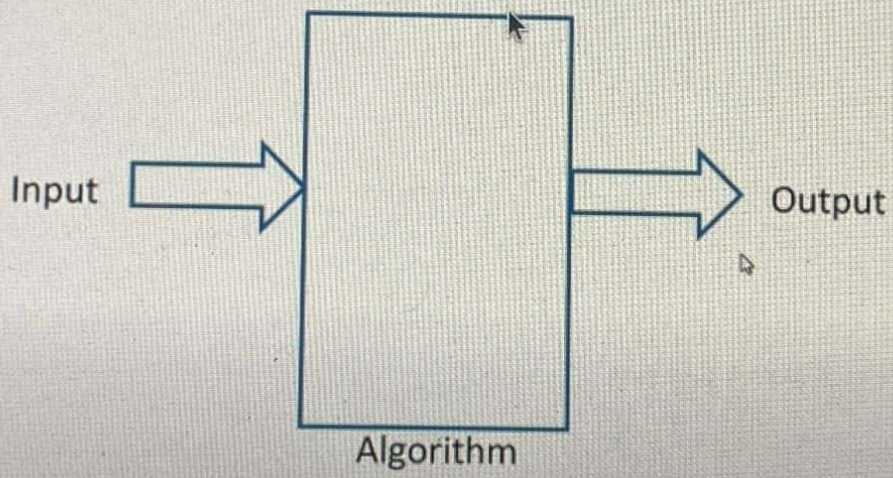
Graph Coloring Approximation Algorithm

- **Time Complexity = $O(n^3)$**

- **Applications of graph coloring**
  - Prepare time table
  - Scheduling
  - Register allocation
  - Mobile radio frequency assignment
  - Map coloring

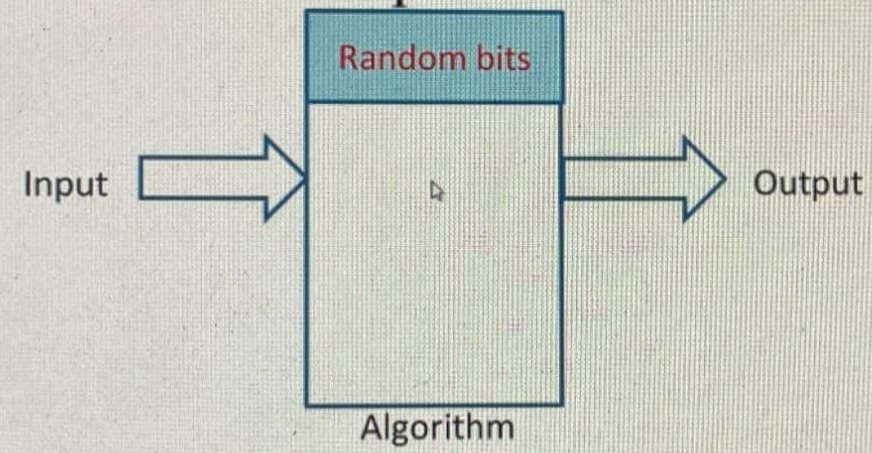
Randomized Algorithms

- **Deterministic Algorithm**: The output as well as the running time are functions of the input only



Randomized Algorithms

- **Randomized Algorithm**: The output or the running time are functions of the input and random bits chosen



- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm
- Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms

Monti Carlo, Randomized Quick Sort

Randomized Algorithms

- **Advantage:**
  - For many problems, a randomized algorithm is the simplest and the fastest
  - Many NP-hard/NP Complete problems can be easily solvable

Randomized Algorithms

- **Type of Randomized Algorithms**
  - Randomized Las Vegas Algorithms
  - Randomized Monte Carlo Algorithms:

Randomized Las Vegas Algorithm

- Output is always correct and optimal.
- Running time is a random number
- Running time is not bounded
- Example: Randomized Quick Sort

Randomized Monte Carlo Algorithms

- May produce correct output with some probability
- A Monte Carlo algorithm runs for a fixed number of steps. That is the running time is deterministic

Randomized Algorithms

- **Example1**: Finding an 'a' in an array of n elements
- **Input**: An array of $n \geq 2$ elements, in which half are 'a's and the other half are 'b's
- **Output**: Find an 'a' in the array

Monte Carlo algorithm

```
Algorithm findingA_MC(A, n, k )
{
        i=0;
        repeat
        {
                Randomly select one element out of n elements
                i=i+1;
        }until(i=k or 'a' is found);
}
```

- After k iterations, the probability of finding an 'a' is Pr[find a] = $1-(1/2)^k$
- This algorithm does not guarantee success, but the run time is bounded. The number of iterations is always less than or equal to k.
- Therefore the time complexity = **O(k)**