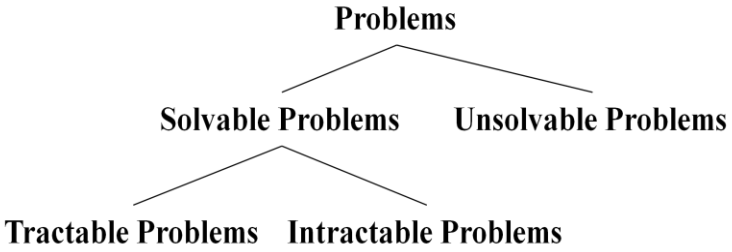
Module V

- **Introduction to Complexity Theory**
  - **Tractable and Intractable Problems**
  - **Complexity Classes – P, NP, NP- Hard and NP-Complete Classes**
  - **NP Completeness proof of Clique Problem and Vertex Cover Problem**
  - **Approximation algorithm**
    - **Bin Packing**
    - **Graph Coloring**
  - **Randomized Algorithms (Definitions of Monte Carlo and Las Vegas algorithms)**
    - **Randomized version of Quick Sort algorithm with analysis**

- **Introduction to Complexity Theory**
  - **Tractable and Intractable Problems**



    - When the complexity is expressed as some polynomial function over input size, then the concerned problem is tractable.
    - When the complexity is expressed as some exponential function over input size, then the concerned problem is intractable.
    - An intractable problem has a faster complexity growth as compared to tractable problems.
    - Tractable problem solutions are implemented in practice. They have polynomial time complexity.
    - According to Cook-Krap thesis, a problem that is in P is called tractable and that is not in P is called intractable.
    - Example of tractable problem
      - PATH problem: Given directed graph G, determine whether a directed path exists from vertex s to vertex t.
        - Time complexity = O(n)

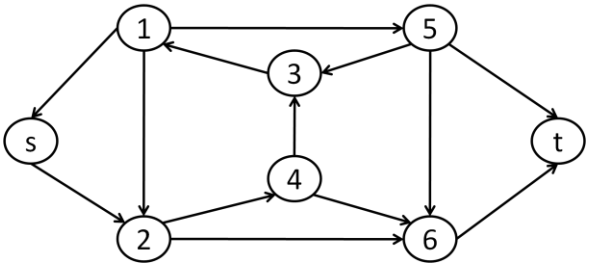
          Where n – total number of vertices
    - Example of intractable problem
      - Knapsack Problem
        - Time Complexity = $O(2^n)$
      - Traveling Salesman Problem
        - Time Complexity = $O(n^2 2^n)$

  - **Deterministic and Non-Deterministic Algorithms**
    - Each and every step of an algorithm is clear and unambiguous, then that algorithm is called deterministic.
    - Few steps in an algorithm is not defined well, then that algorithm is called non deterministic.

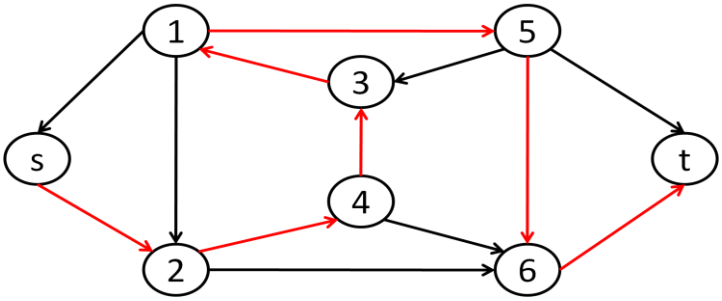
- **Complexity Classes**
  1. P
  2. NP
  3. NP-Hard
  4. NP-Complete

- **Class P**
  - Class P consists of those problems that are solvable in polynomial time.
  - P problems can be solved in time $O(n^k)$. Here n is the size of input and k is some constant.
  - Example:
    - **PATH Problem:** Given directed graph G, determine whether a directed path exists from s to t.
      - Algorithm
        - Inputs: <G,s,t> G – directed graph s,t – 2 nodes
        1. Place a mark on node s and enqueue it into an empty queue.
        2. Repeat step 3 until the queue is empty
        3. Dequeue the front element a. Mark all unvisited neighbors of a and enqueue those into the queue.
        4. If t is marked, then accept. Otherwise reject.
      - Complexity Calculation
        - Step 1 & 4 will execute exactly once.
        - Step 3 & 4 will execute atmost n times, where n is the number of nodes in G.
        - Time complexity = O(n).
        - This is a polynomial time algorithm.
  - Other Examples:
    - 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 be solved in exponential or factorial time. Suppose these problems have no polynomial time solution. We can verify these problems in polynomial time. These are called NP problems.
  - NP is a class of problem that having only non-polynomial time algorithm and a polynomial time verifier.
  - Example:
    - **Hamiltonian path(HAMPATH) Problem**
      - A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once.



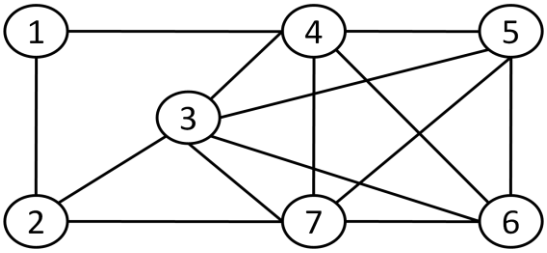
- The Hamiltonian path of the above graph is as follows



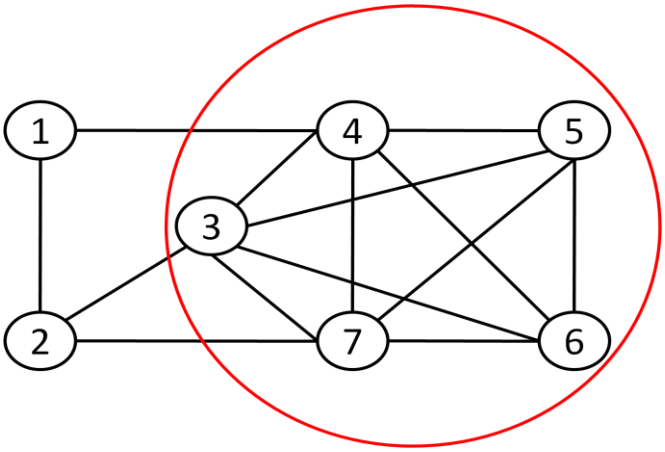
- There is no polynomial solution to find the Hamiltonian path from s to t in a given graph.
- The **HAMPATH** problem is to test whether a graph contains a hamiltonian path connecting 2 specified nodes.
- HAMPATH problem have a feature called polynomial verifiability. Here verifying the existence of a Hamiltonian path may be much easier than determining its existence.

- **HAMPATH Verifier Algorithm**
  - **Inputs**
    - G: the graph
    - s,t: two vertices
    - P: the path P_1, P_2, ... P_m where m is the number of nodes in the graph G
  - **Algorithm**
    1. Check whether s= P_1 and t= P_m. if either fails, reject
    2. Check for the repetition of the nodes in the list P. If any are found, reject.
    3. For each i, check whether (P_i, P_{i+1}) is an edge in G. Here i is varies from 1 to m-1. If any are not, reject.
    4. If all test have been passed, then accept it.
  - This algorithm runs in polynomial time. Therefore **HAMPATH problem is a NP problem.**

- **CLIQUE Problem**
  - A clique in an undirected graph is a sub-graph where every two nodes are connected by an edge.
  - A k-clique is a clique that contains k nodes.
  - Example:



  - This graph contains a 5-clique

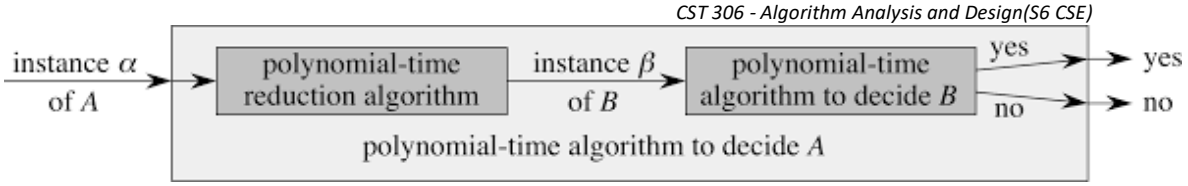


- **CLIQUE Problem**: To determine whether a graph contains a clique of specified size.
- There is no polynomial time algorithm exists for this problem. But we can verify this in polynomial time
- **CLIQUE Verifier Algorithm**
  - **Inputs**
    - G: the graph with V set of vertices and E set of edges
    - k: size of clique
    - V': sub-graph vertex set
  - **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.**

- Other examples of NP problem
  - CIRCUIT-SAT problem
  - 3CNF-SAT problem
  - Vertex cover problem
  - Independence set problem
  - Traveling Salesman problem
  - 3-coloring problem

- Whether P=NP? is one of the greatest unsolvable problem in theoretical computer science

- **Class NP- Hard**
  - **Polynomial Time Reductions**
    - Consider a decision problem A, which is like to solve in polynomial time.
    - Consider another decision problem B, which having a polynomial time algorithm.
    - Suppose that we have a procedure that transforms any instance α of A into some instance β of B 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.



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

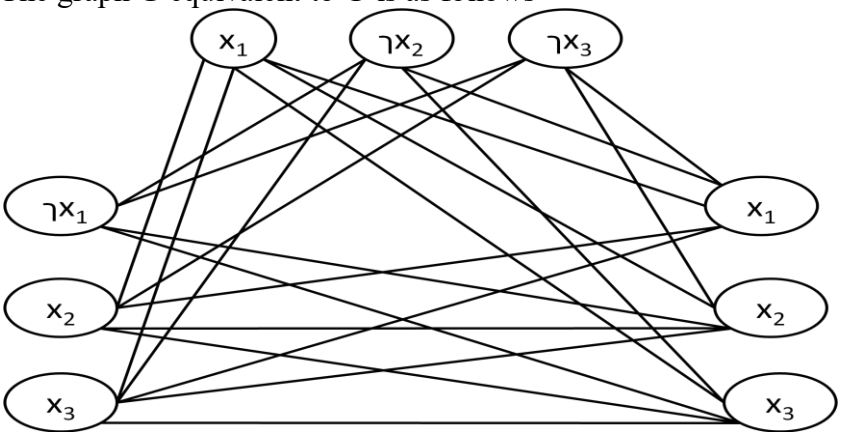
    $Y \leq_p X$ for every Y in NP
  - It means that X is as hard as all problems in NP.
  - If X can be solved in polynomial time, then all problems in NP can also solved in polynomial time.

- **Class NP-Complete**
  - If the problem is NP as well as NP-Hard, then that problem is NP Complete.
  - Example:
    - CIRCUIT-SAT problem: Given a Boolean circuit C, is there an assignment to the variables that causes the circuits to output 1?
    - SAT(Satisfiability) problem: Given a Boolean expression ϕ, is there an assignment to the variables that causes the expression to output 1?
    - 3-CNF-SAT
      - Literal: The variables and its negation in a Boolean formula
      - Clause: OR of one or more literals
        - Ex: $(x_1 \vee \neg x_2 \vee x_3)$
      - Conjunctive Normal Form(CNF): AND of clauses
        - Ex: $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3)$
      - 3-CNF: Each clause has exactly 3 distinct literals.
        - Ex: $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$
      - 3-CNF-SAT Problem: Given a 3-CNF expression ϕ, is there an assignment to the variables that causes the expression to output 1?
    - CLIQUE problem: Given a graph G(V, E) and an integer k, the problem is to determine if the graph contains a clique of size k
    - VERTEX COVER problem: Find the set of vertices that covers all the edges of the given graph.

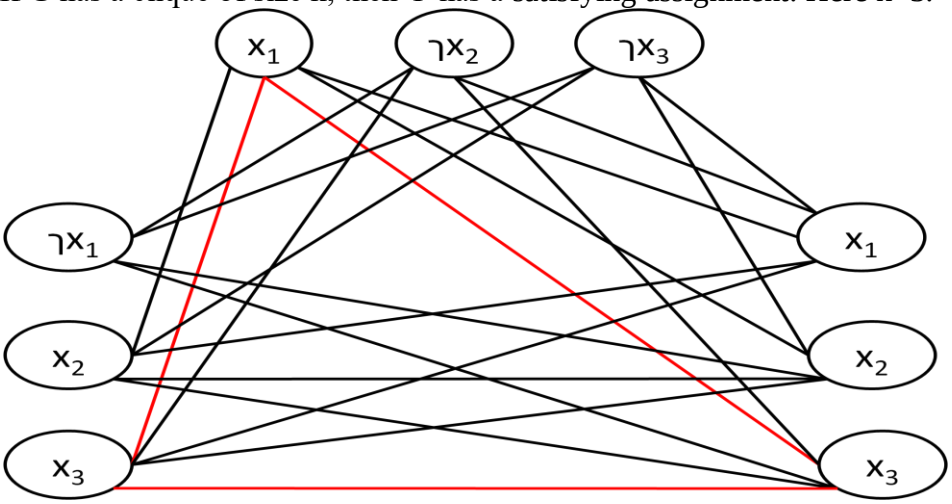
- **NP Completeness Proof**
  - Steps to prove that the given problem is NP Complete
    1. Prove that the given problem is NP
       - Write a polynomial time verification algorithm.
    2. Prove that the given problem is NP Hard
       - Write a polynomial time reduction algorithm from any NP problem to the given problem.

- **CLIQUE problem is NP Complete: Proof**
  - Step 1: Write a polynomial time verification algorithm to prove that the given problem is NP
    - Algorithm: Let G= (V,E), we use the set V' ⊆ V of k vertices in the clique as a certificate of G
      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.**
  - Step 2: Write a polynomial time reduction algorithm from 3-CNF-SAT problem to CLIQUE problem(3-CNF-SAT ≤p CLIQUE)
    - Algorithm
      - Let $\Phi = C_1 \wedge C_2 \ldots\ldots \wedge C_k$ be a Boolean formula in 3CNF with k clauses
      - Each clause C_r has exactly three distinct literals l^r_1, l^r_2, l^r_3.
      - Construct a graph G such that Φ is satisfiable iff G has a click of size k.
      - The graph G is constructed as follows
        - For each clause $C_r = (l^r_1 \vee l^r_2 \vee l^r_3)$ in Φ, we place a triple of vertices V^r_1, V^r_2 and V^r_3 in to V.
        - Put an edge between V^r_i to V^s_i if following two conditions hold
          - V^r_i and V^s_i in different triples(that is r!=s)
          - l^r_i is not a negation of l^s_i.

- Example: $\Phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
  - The graph G equivalent to Φ is as follows



- If G has a clique of size k, then Φ has a satisfying assignment. Here k=3.



- G can easily be constructed from Φ in polynomial time.
- So **CLIQUE problem is NP Hard**.

- Conclusion
  - CLIQUE problem is NP and NP Hard. So it is NP-Complete

- **Vertex Cover Problem is NP-Complete: Proof**
  - The vertex Cover of a graph is defined as a subset of its vertices, such for every edge in the graph, from vertex **u** to **v,** at least one of them must be a part of the vertex cover set.
  - Vertex cover problem is to find the minimum sized vertex cover of the given graph.

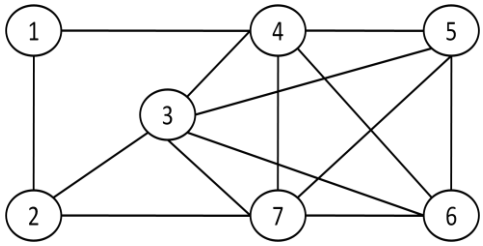
  - **Steps to prove that Vertex Cover is a NP-Complete problem**

    - 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
           a. 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.**

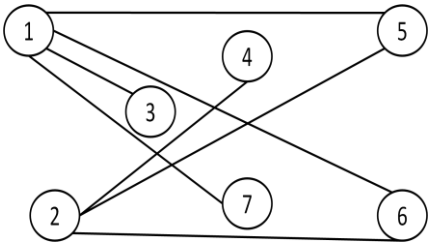
    - Step 2: Write a polynomial time reduction algorithm from CLIQUE problem to VERTEX COVER problem
      - **Algorithm**

        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.

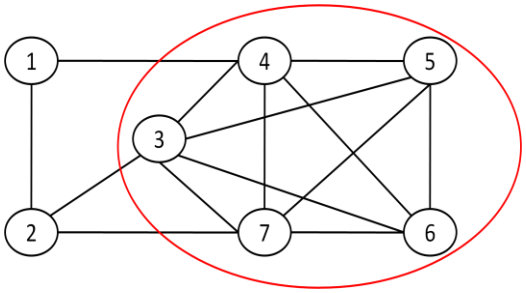
- Example:

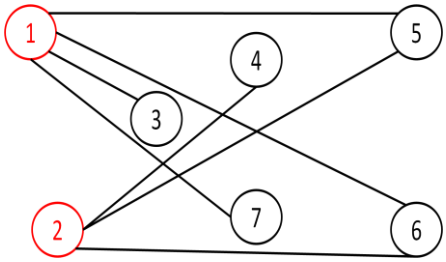


G=(V, E)



G'=(V, E')





Vertex cover of G' is {1,2}
Size of vertex cover of G' is 2.
If so G has a clique of size |V| - 2 = 5

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

- Conclusion
  - VERTEX COVER problem is NP and NP-Hard.
  - So it is NP-Complete

- **Traveling Salesman Problem is NP-Complete: Proof**
  - The salesman wishes to make a tour, visiting each city exactly once and finishing at city he starts from.
  - c(i,j): cost to travel from city i to city j.
  - The salesman wishes to make the tour whose total cost is minimum.
  - **Steps to prove that TSP is a NP-Complete problem**
    - Step 1: Write a polynomial time verification algorithm to prove that TSP problem is NP
      - Inputs: <G,P,k> P: TSP path $P_1,P_2, \ldots . P_n$ k: maximum tour cost
      - Algorithm:
        1. Test whether P contains each vertex exactly once.
        2. For each i between 1 and n-1, check whether $(P_i,P_{i+1},)$ is an edge of G.
        3. Check whether $(P_n,P_1,)$ is an edge of G
        4. Sum up the edge costs and check whether the sum is atmost k
        5. If all steps pass, then accept. Otherwise reject.
      - This algorithm will execute in polynomial time. Therefore **TSP problem is a NP problem.**

- Step 2: Write a polynomial time reduction algorithm from VERTEX-COVER problem to TSP problem(VERTEX-COVER $\leq_p$ TSP)
  1. Let G=(V,E) be an instance of HAM-CYCLE
  2. We can construct an instance of TSP as follows
     - Construct a complete graph G'=(V,E')
     - Define the cost function c for G'
       - c(i,j) = 0 if (i,j) is an edge in E
       - c(i,j) = 1 if (i,j) is not an edge in E
  - The instance of TSP is then (G',c,0).
  - The graph G has a hamiltonian cycle iff graph G' has a tour of cost atmost 0.

  - Instance of VERTEX-COVER is converted to instance of TSP in polynomial time
  - Therefore TSP is a NP-Hard problem
- Conclusion
  - TSP problem is NP and NP Hard. So it is NP-Complete

- **Examples**
  1. Consider the following algorithm to determine whether or not an undirected graph has a clique of size k. First, generate all subsets of the vertices containing exactly k vertices. Next, check whether any of the sub-graphs induced by these subsets is complete (i.e. forms a clique). Why is this not a polynomial-time algorithm for the clique problem, thereby implying that P = NP?

- **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 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*
    - 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.
  - **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.

- Different Types of Approximation Algorithms
  - **Absolute Approximation Algorithm** : An algorithm is Absolute Approximation Algorithm iff $|C^*-C| \leq k$, for some constant k
  - **f(n)-Approximation Algorithm:** An algorithm is f(n)-Approximation Algorithm iff $|C^*-C|/|C^*| \leq f(n)$, for $C^* > 0$
  - **ϵ- Approximation Algorithm:** An ϵ-Approximation Algorithm is an f(n)-Approximation Algorithm for which $f(n) \leq \epsilon$ for some constant ϵ

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

- **Bin Packing 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 :

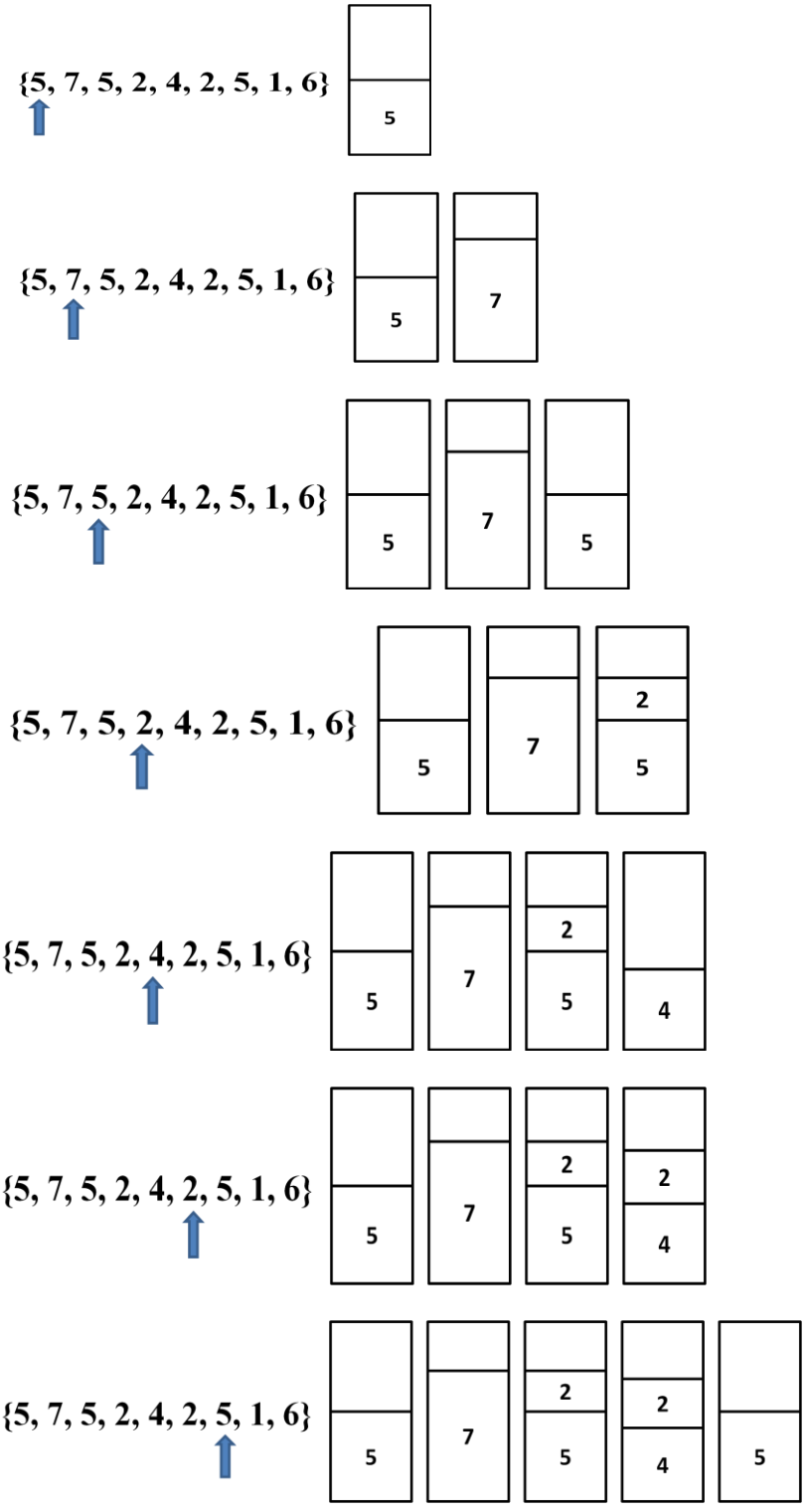
    Min no. of bins >= Ceil ((Total Weight) / (Bin Capacity))

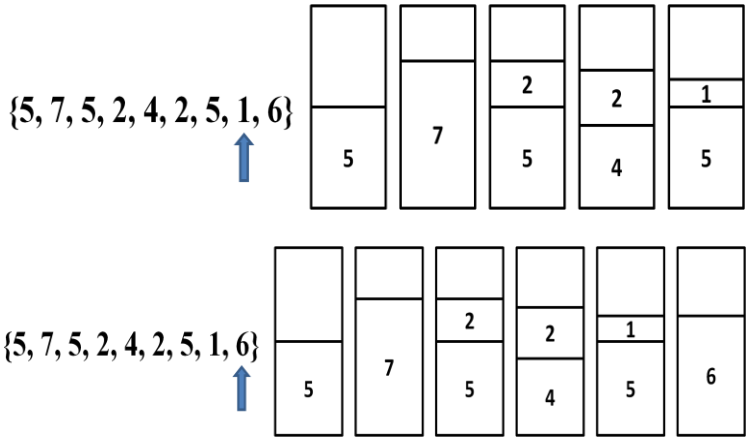
  - **Applications**
    - Loading of containers like trucks.
    - Placing data on multiple disks.
    - Job scheduling.
    - Packing advertisements in fixed length radio/TV station breaks.
    - Storing a large collection of music onto tapes/CD's, etc.

  - **Different Bin Packing Approximation Algorithms**
    - **Online Algorithm**
      - These algorithms are for Bin Packing problems where items arrive one at a time (in unknown order), each must be put in a bin, before considering the next item.
      - Some online bin packing algorithms are:
        - Next Fit Algorithm
        - First Fit Algorithm
        - Best Fit Algorithm
        - Worst Fit Algorithm
    - **Offline Algorithm**
      - In the offline version of bin packing, the algorithm can see all the items before starting to place them into bins.
      - Some online bin packing algorithms are:
        - First Fit Decreasing Algorithm
        - Best Fit Decreasing Algorithm

- **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 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 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 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 Algorithm**
  - Sort the items in the descending order of their size
  - Apply First fit algorithm
  - 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 Decreasing Algorithm**
  - Sort the items in the descending order of their size
  - Apply Best fit algorithm
  - Time Complexity
    - Best case Time Complexity = $\theta(n \log n)$
    - Average case Time Complexity = $\theta(n^2)$
    - Worst case Time Complexity = $\theta(n^2)$

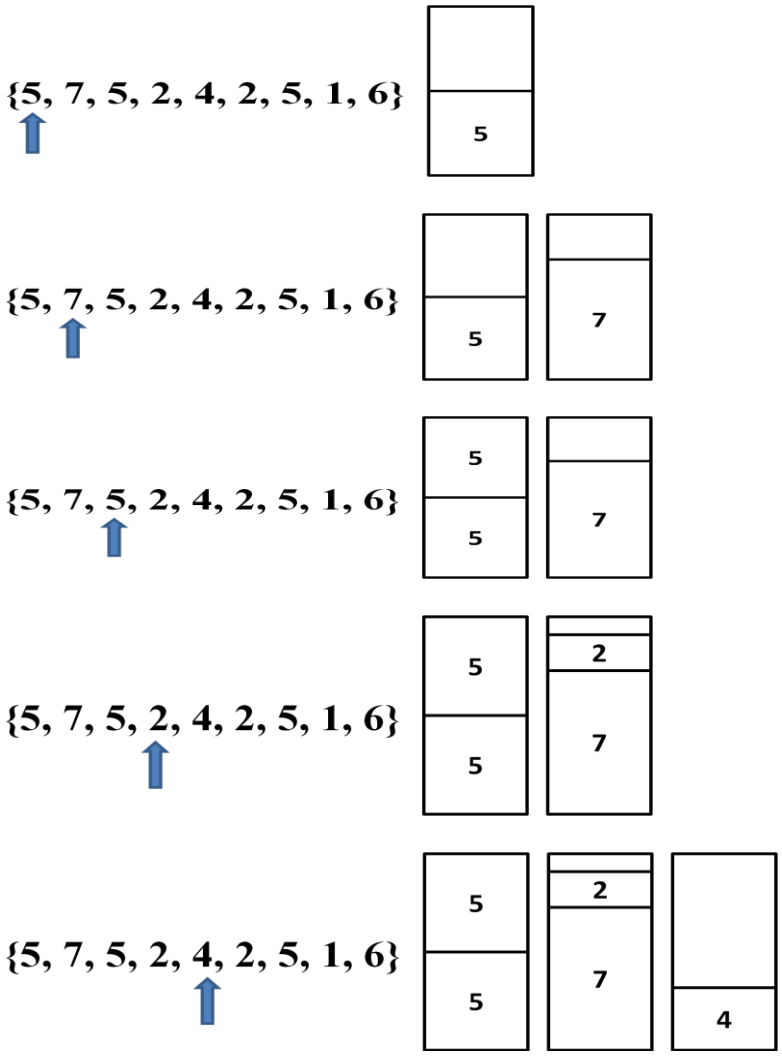
- **Example:** Apply different Bin packing approximation algorithms on the following items with bin capacity=10. Assuming the sizes of the items be {5, 7, 5, 2, 4, 2, 5, 1, 6}.
- **Solution**
  - Minimum number of bins >= Ceil ((Total Weight) / (Bin Capacity))
    = Ceil (37 / 10) = 4
  - **Next Fit**

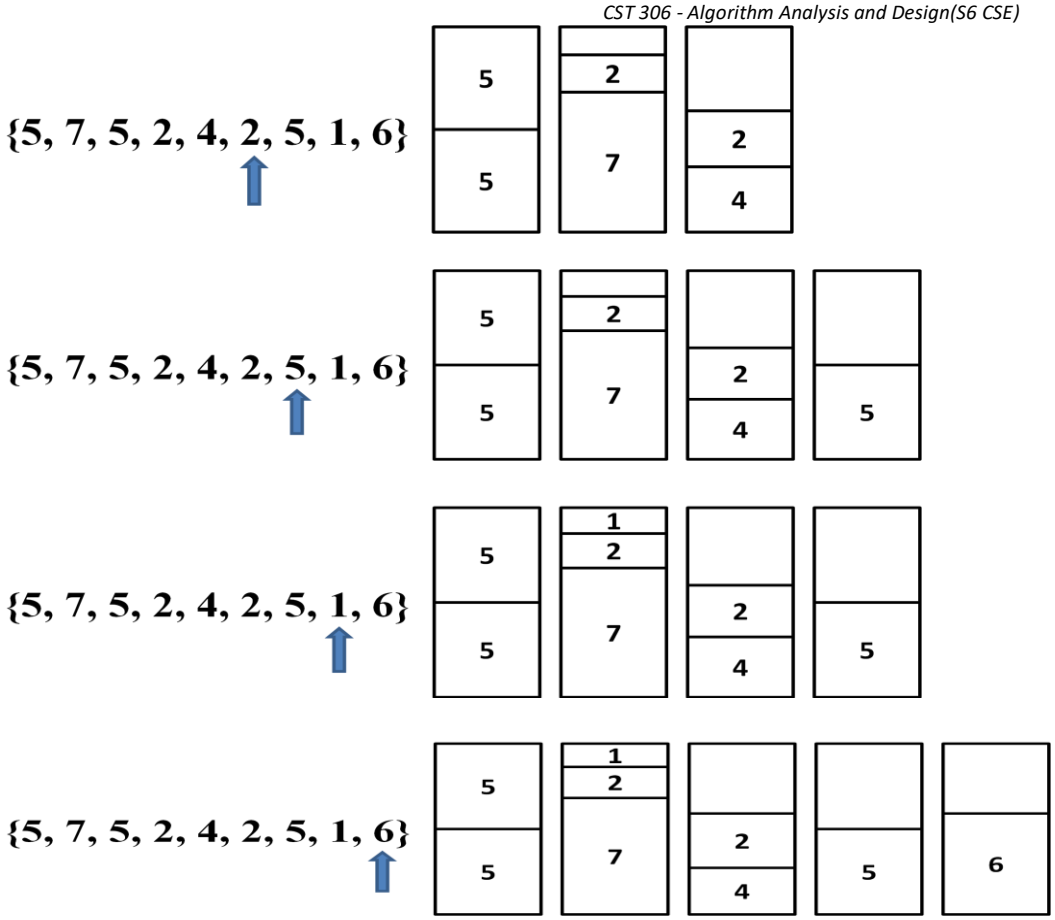




Number of bins required = 6

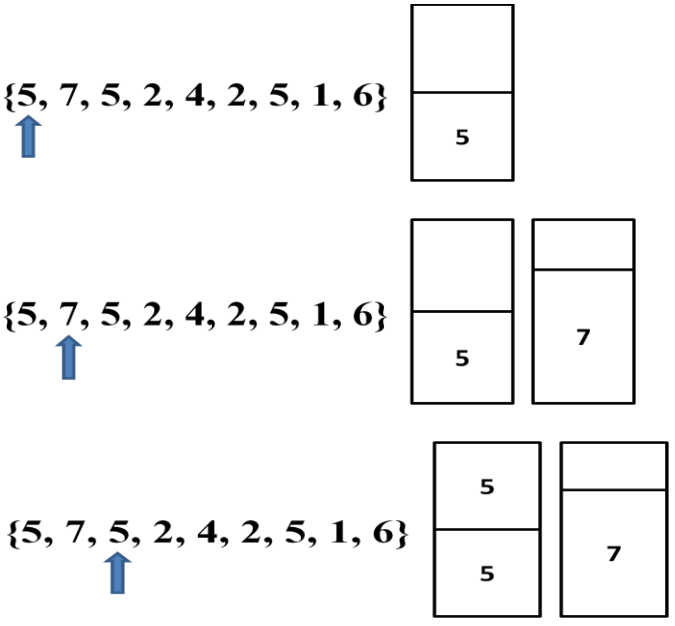
- **First Fit**

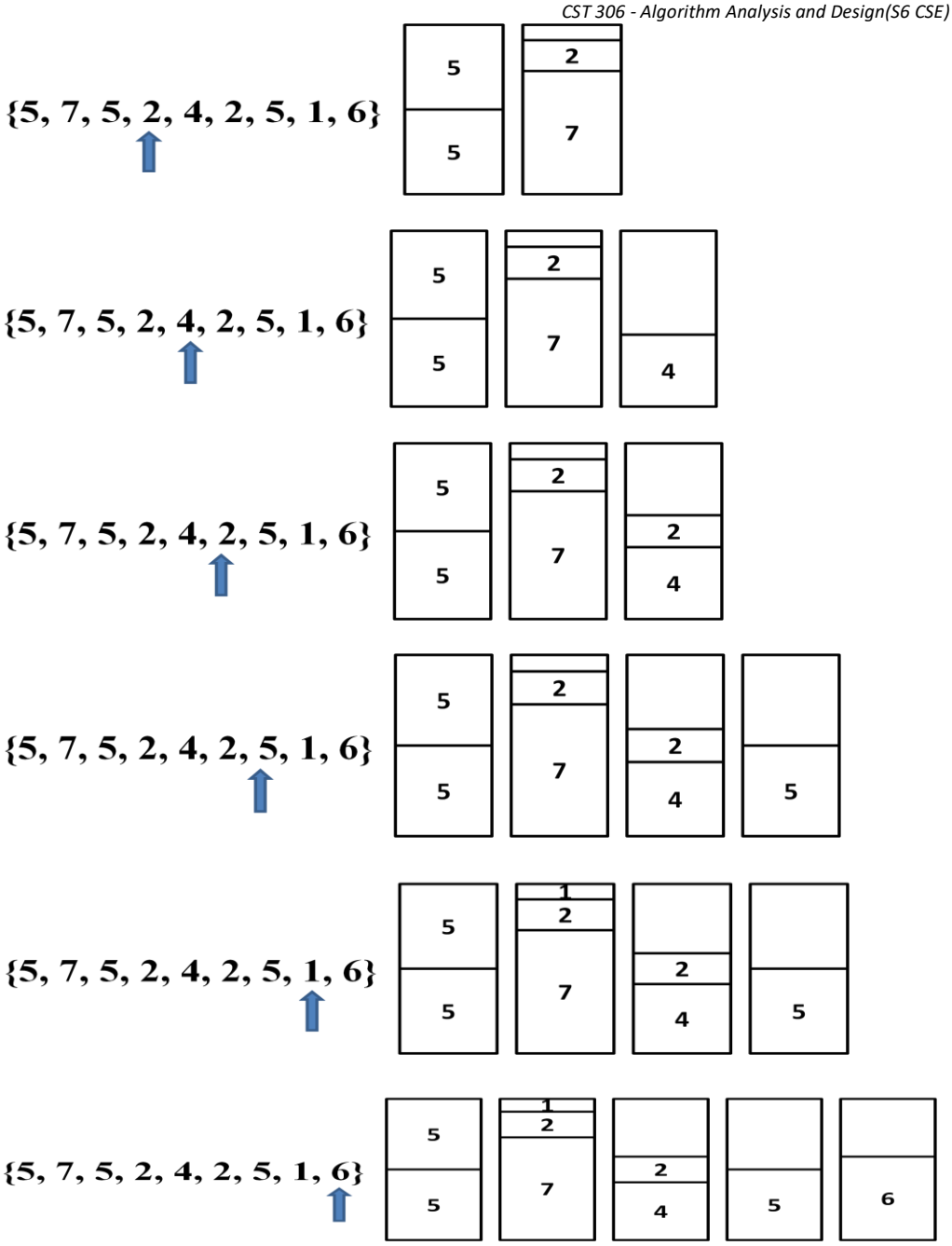




Number of bins required = 5

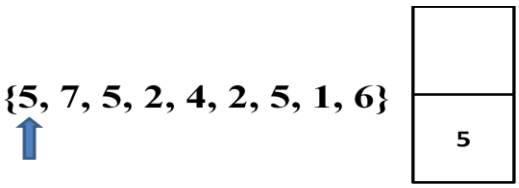
- **Best Fit**

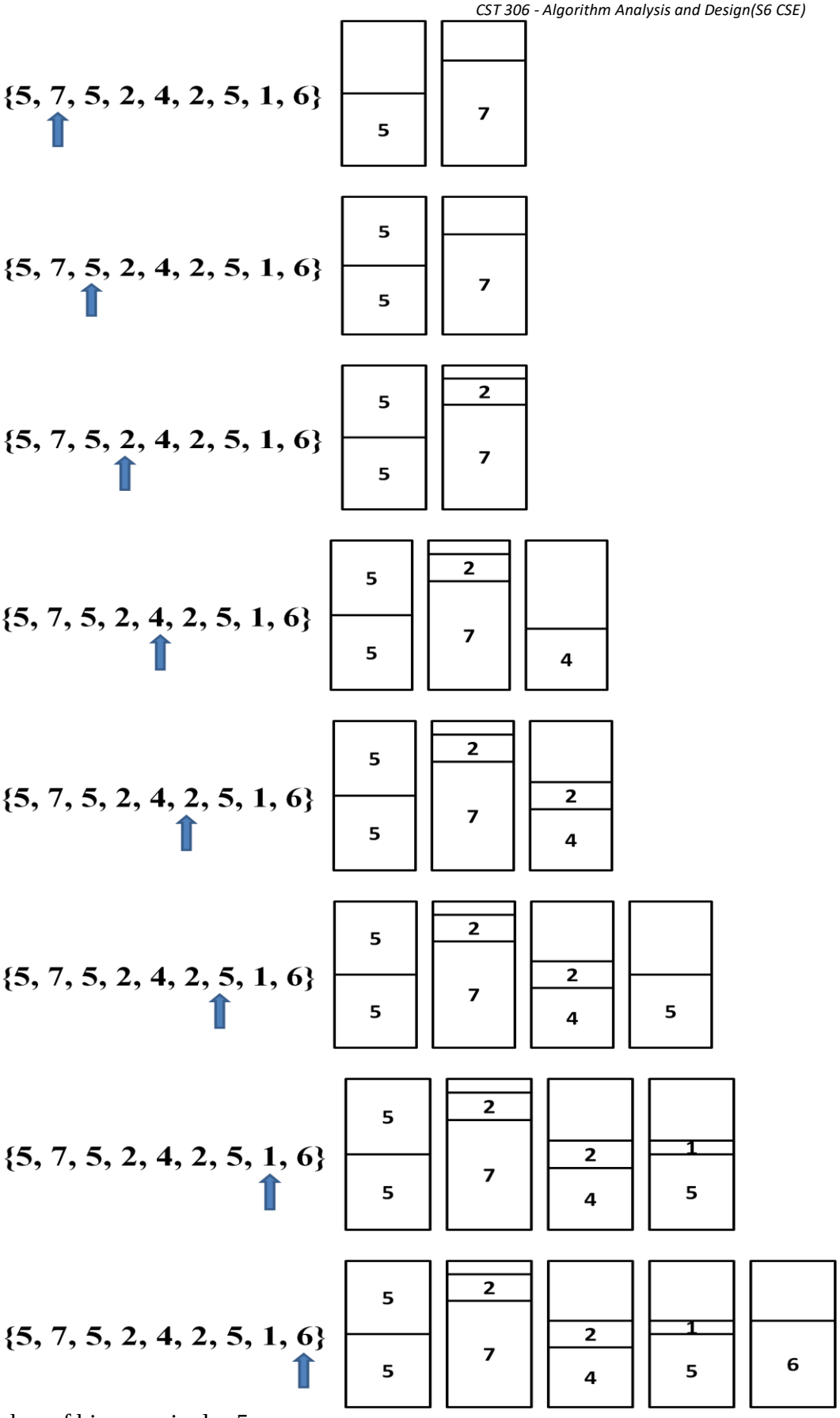




Number of bins required = 5

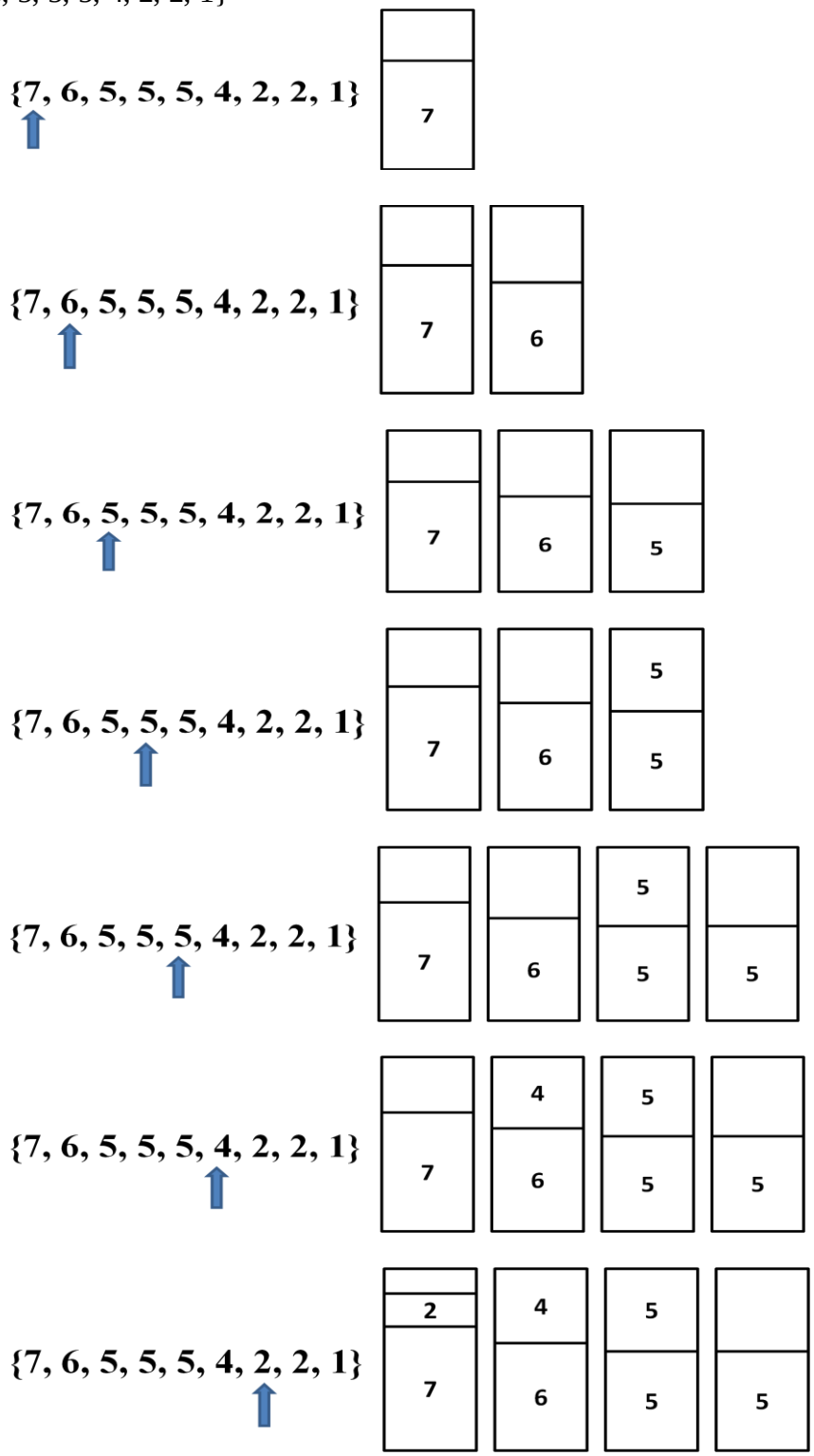
- **Worst Fit**

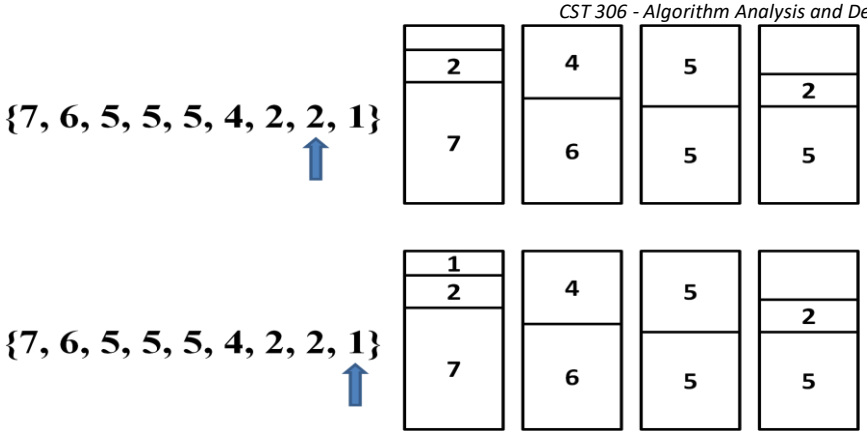




Number of bins required = 5

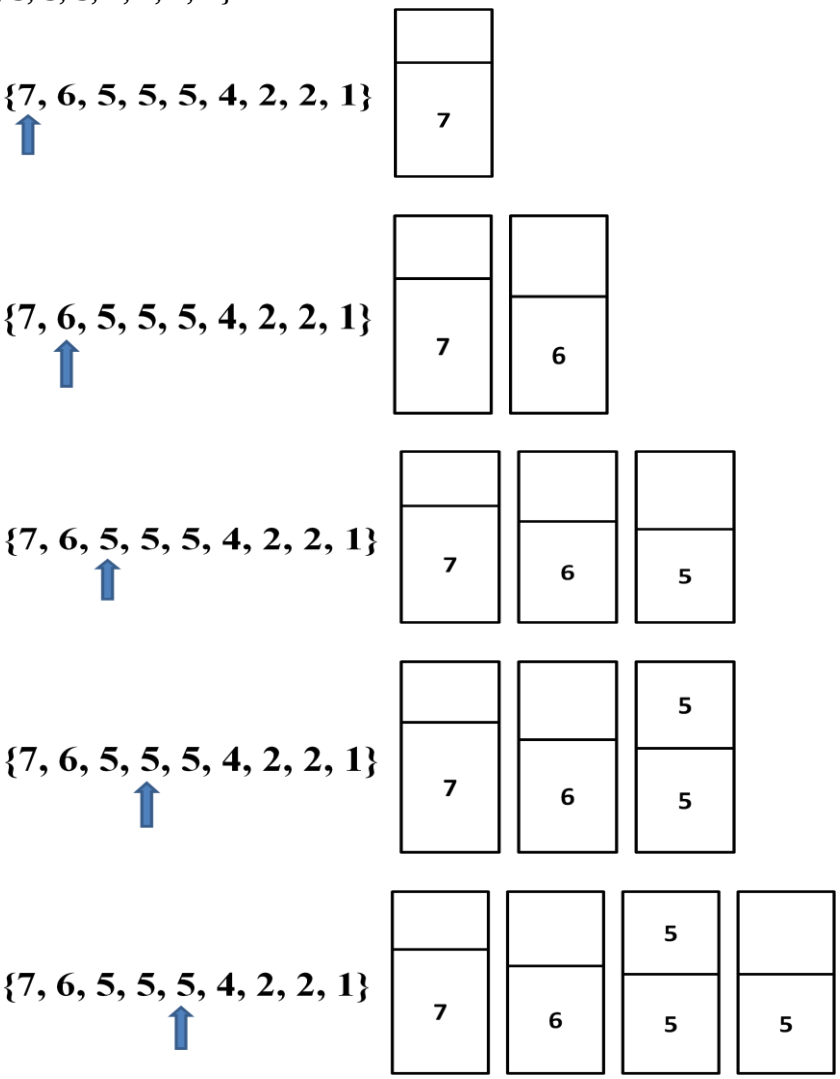
- **First Fit Decreasing**
  - Arrange the items in the decreasing order of the weight
    {7, 6, 5, 5, 5, 4, 2, 2, 1}

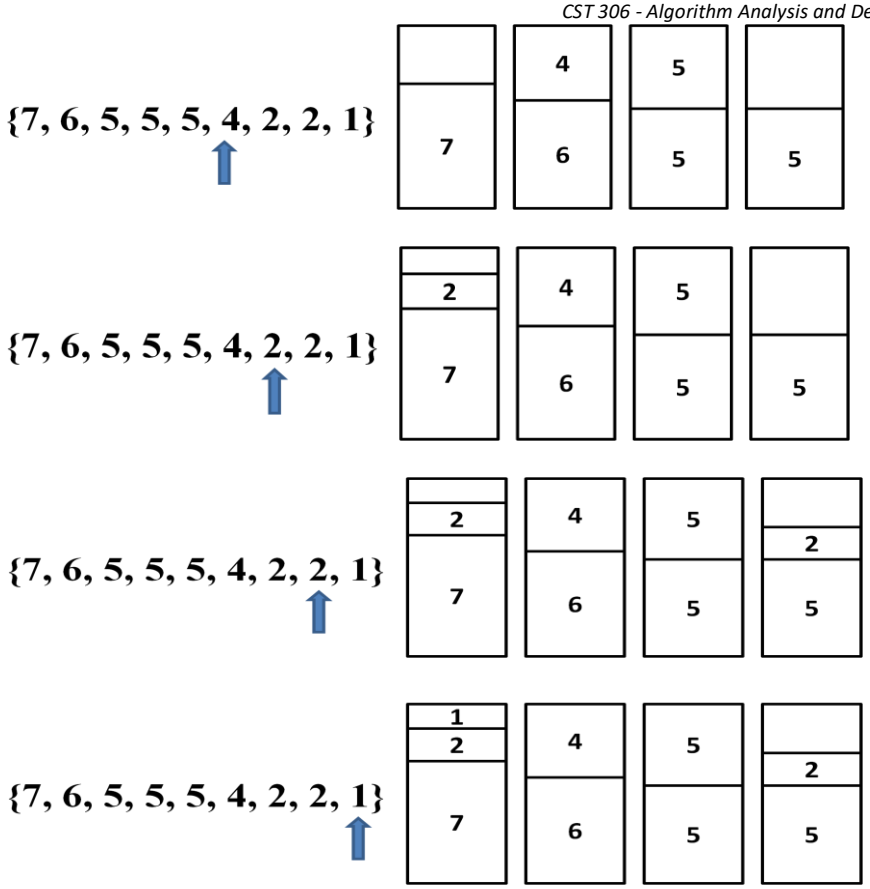




Number of bins required = 4

- **Best Fit Decreasing**
  - Arrange the items in the decreasing order of the weight
    {7, 6, 5, 5, 5, 4, 2, 2, 1}

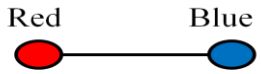




Number of bins required = 4

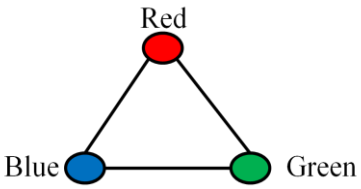
- **Graph Coloring**
  - **Different Graph coloring problems**
    - Vertex coloring
    - Edge coloring
    - Face coloring

  - **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 = ∅
    - A graph is 1-colorable iff E = ∅

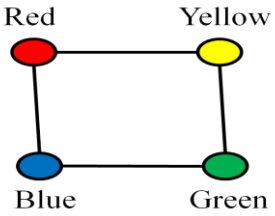


2 Colorable graph

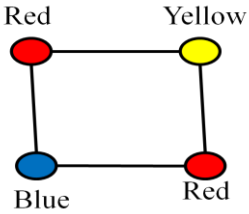
$\chi(G) = 2$



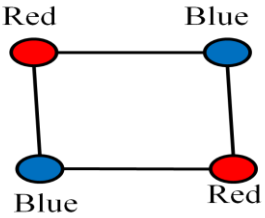
3 Colorable graph
$\chi(G) = 3$



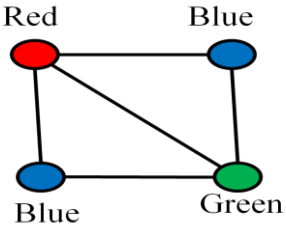
4 Colorable graph
$\chi(G) = 4$



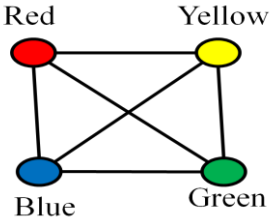
3 Colorable graph
$\chi(G) = 3$



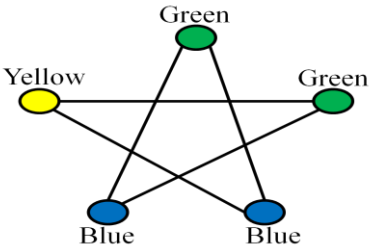
2 Colorable graph
$\chi(G) = 2$



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

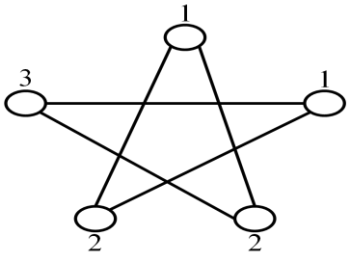


4 Colorable graph
$\chi(G) = 4$

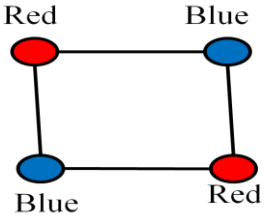


3 Colorable graph
$\chi(G) = 3$

Instead of using colors, we can use numbers/symbols.



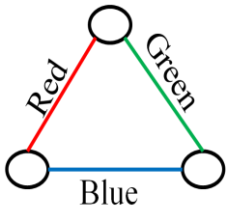
- **Chromatic Number**: It is the minimum number of colours with which a graph can be coloured.



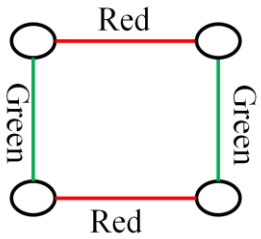
Here, Chromatic number = 2

  - $\chi(G) = 1$, 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.
- A graph whose chromatic number is k, then it is called **k-chromatic graph.**
- A subset of vertices assign to the same color is called a **color class**.

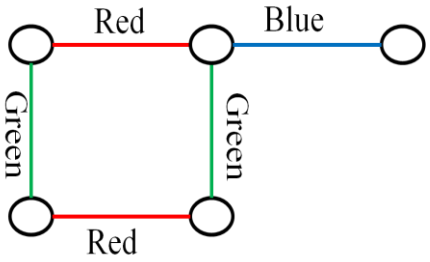
- **Edge coloring**
  - Given a graph G=(V,E), assign a color to each edges so that no two adjacent edges share the same color



$\chi(G) = 3$

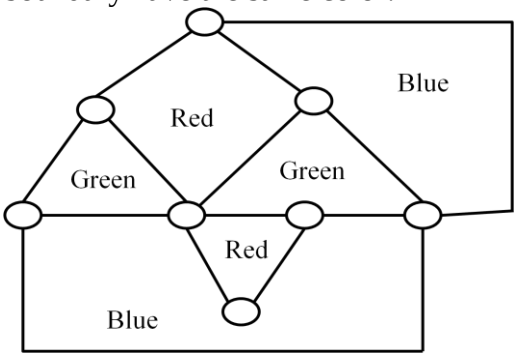


$\chi(G) = 2$



$\chi(G) = 3$

- **Face Coloring**
  - For a planar graph, assign a color to each face/region so that no two faces that shares boundary have the same color.



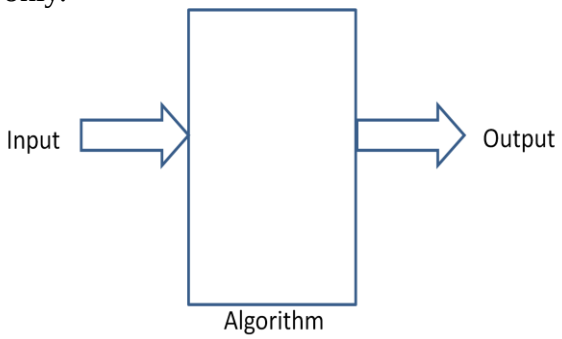
- **Graph Coloring Approximation Algorithm**
  - Graph coloring problem is a NP-Complete problem. But there are approximation algorithms
  - Important graph coloring problem is vertex coloring.
  - Following is the greedy approximation algorithm for vertex coloring.

```
Algorithm Approximate_Graph_Coloring(G, n)
{
  for i=1 to n do
  {
        for c=1 to n do
        {
            If no vertex adjacent to v_i has color c
            {
                Color v_i with c
                Break
            }
        }
  }
}
```

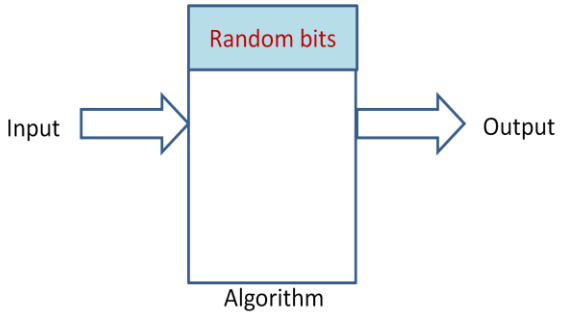
- **Time Complexity = $O(n^3)$**

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

- **<u>Randomized Algorithm</u>**
  - **Deterministic Algorithm**: The output as well as the running time are functions of the input only.



  - **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

▪ The computer is not capable of generating truly random numbers
- The computer can only generate pseudorandom numbers-numbers that are generated by a formula
- Pseudorandom numbers look random, but are perfectly predictable if you know the formula
- Pseudorandom numbers are not used for security applications
- Devices for generating truly random numbers do exist. They are based on radioactive decay, or on lava lamps

▪ It hopes to achieve good performance in the "average case" over all possible choices of random bits.

- **Type of Randomized Algorithms**

  - **Randomized Las Vegas Algorithms**
    - 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
    - Example: Suppose we want to find a number among n given numbers which is larger than or equal to the median
      - The best deterministic algorithm needs O(n) time to produce the result.
      - Algorithm
        - Suppose n is very large(n=100,000,000,000)
        - Choose 100 of the numbers with equal probability.
        - Find maximum among these numbers
        - Return the maximum.
      - Running time = O(1)
      - Probability of failure = $\frac{1}{2^{100}}$

- **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

  - **Las Vegas algorithm**

```
Algorithm findingA_LV(A, n)
{
        repeat
        {
                Randomly choose one element out of n elements
        }until('a' is found)
}
```

  - This algorithm succeeds with probability 1. The number of iterations varies and can be arbitrarily large, but the expected number of iterations is

    $$\lim_{n\to\alpha} \sum_{i=1}^{n} \frac{i}{2^i} = 2$$

  - The expected number of trials before success is 2.
  - Therefore the time complexity = O(1)

- **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);
}
```

- If an '*a*' is found, the algorithm succeeds, else the algorithm fails. 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)**

- **Example2: Randomized Quick Sort**
  - **Deterministic Quick Sort Algorithm**

Algorithm QuickSort(A[], low, high)

1. If low >= high, then EXIT
2. Let the 1st element of S as the pivot element, say x
3. Partition A[low..high] into two subarrays. The first subarray has all the elements of A that are less than x and the second subarray has all those that are greater than x. Now the index of x be pos.
4. QuickSort(A, low, pos-1)
5. QuickSort(A, pos+1, high)

- Time taken is depends only on the initial permutation of A
- In worst case (If the array elements are sorted), the running time = $O(n^2)$
- In average case, the expected running time = O(n log n)

  - **Randomized Quick Sort**

Algorithm randQuickSort(A[], low, high)

1. If low >= high, then EXIT
2. While pivot 'x' is not a Central Pivot.
   2.1. Choose uniformly at random a number from [low..high]. Let the randomly picked element be x.
   2.2. Count elements in A[low..high] that are smaller than x. Let this count be sc.
   2.3. Count elements in A[low..high] that are greater than x. Let this count be gc.

2.4. Let n = (high-low+1). If sc >= n/4 and gc >= n/4, then x is a central pivot.
3. Partition A[low..high] into two subarrays. The first subarray has all the elements of A that are less than x and the second subarray has all those that are greater than x. Now the index of x be pos.
4. randQuickSort(A, low, pos-1)
5. randQuickSort(A, pos+1, high)

- **Time Complexity:**
- **How many times while loop runs before finding a central pivot?**
  - The probability that the randomly chosen element is central pivot is 1/n.
  - Therefore, expected number of times the while loop runs is n.
  - Thus, the expected time complexity of step 2 is O(n).
- **What is overall Time Complexity in Worst Case?**
  - In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst case height of recursion tree is $\log_{3/4} n$ which is O(log n).
  - T(n) < T(n/4) + T(3n/4) + O(n)
  - T(n) < 2T(3n/4) + O(n)
  - Solution of above recurrence is O(n log n)

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