NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CX2001 Algorithms **2020/21 Semester 1**

7: Basic Computability Theory

School of Computer Science and Engineering *Nanyang Technological University*

7.1 Polynomial Time Algorithms

- An **algorithm** is said to be **polynomially bounded** if its worst-case complexity is bounded by a polynomial function of the input size.
- A **problem** is said to be **polynomially bounded** if there is a polynomially bounded algorithm for it.

  Sorting problems are bounded by $n \log n$. However, not all problems can be solved in polynomial time.

7.2 Tower of Hanoi

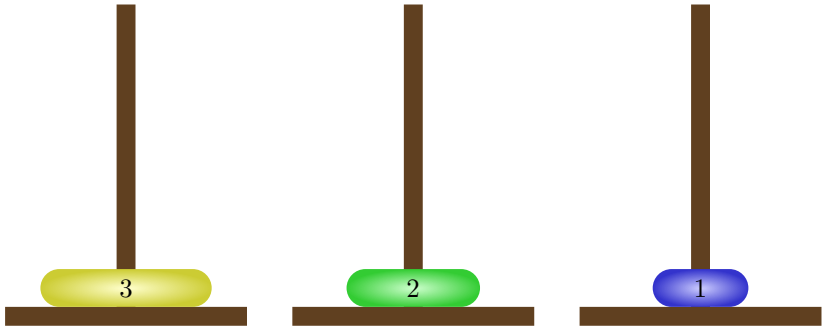
The Tower of Hanoi consists of three rods (towers) and a number of disks of different sizes which can slide onto any rod. Initially, the disks are stacked in ascending order of size on one rod, the smallest at the top and the largest at the bottom. The objective of this puzzle is to move all the disks to another rods. The rules of the puzzles:

1. Only one disk can be moved each time
2. The disk at the top of a rod will be removed and replaced on top of another rod or on an empty rod in each move.
3. No larger disk can be placed on top of any smaller disk.

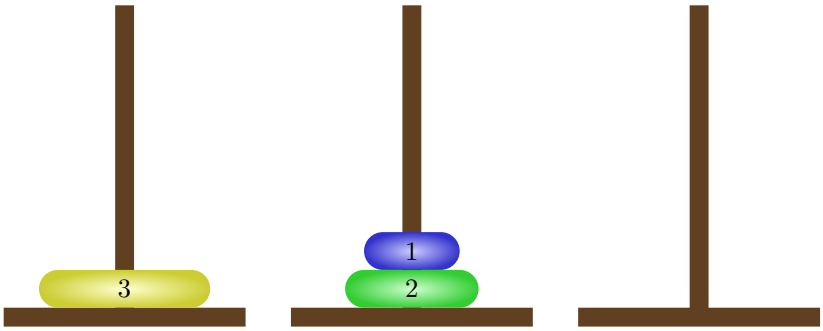
The puzzle with 3 disks can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$ where n is the number of disks.

7.2.1 Example of three-disk case

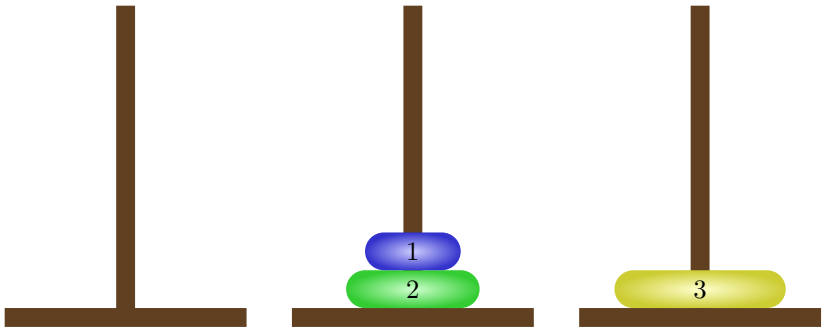
Moved disc from pole 1 to pole 3.



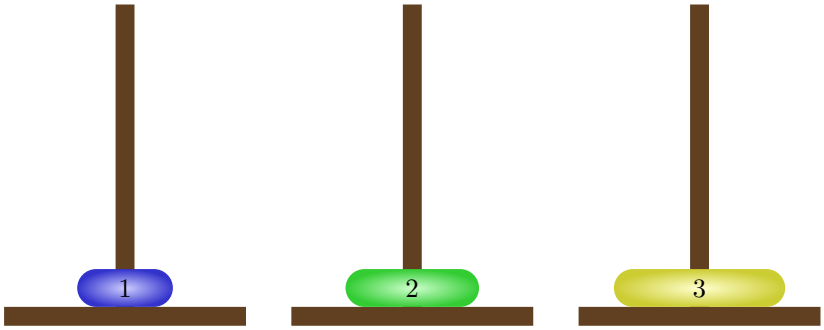
Moved disc from pole 1 to pole 2.



Moved disc from pole 3 to pole 2.



Moved disc from pole 1 to pole 3.

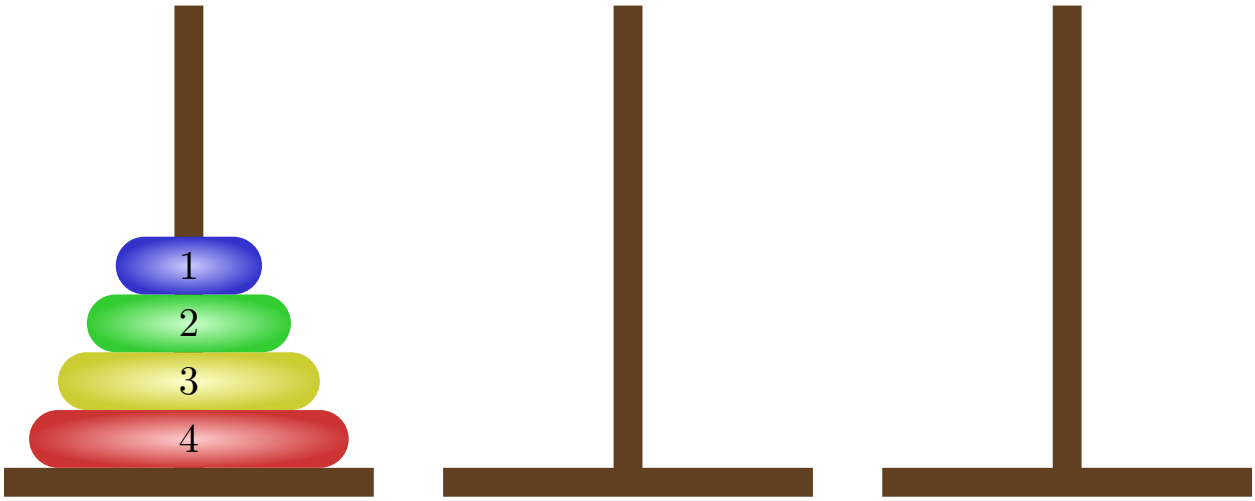


Moved disc from pole 2 to pole 1.

Moved disc from pole 2 to pole 3.

Moved disc from pole 1 to pole 3.

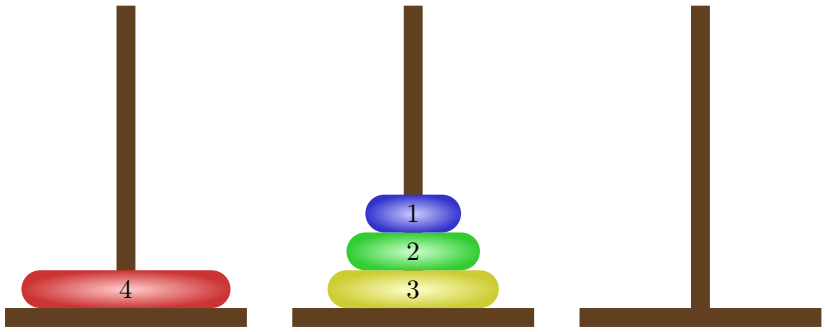
7.2.2 Animation of four-disk case



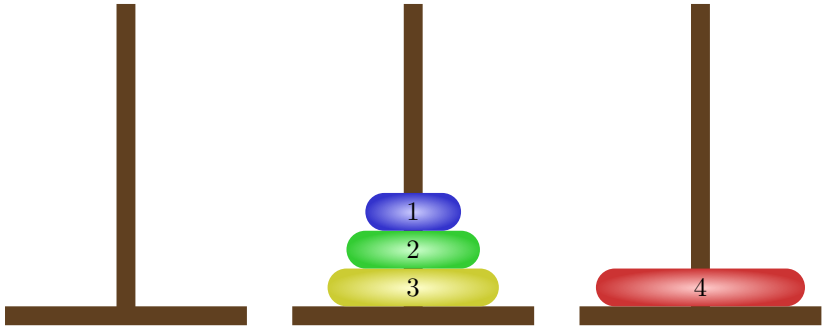
7.2.3 Algorithm

To move n disks from the source tower (left) to the target tower (right), we can divide the problem into three parts as follow:

1. Move $(n-1)$ disks from the source to the intermediate



2. Move the last disk from the source to the target



3. Move $(n-1)$ disks from the intermediate to the target

The step 1 and the step 3 are the sub-problems with $(n-1)$ disks. To resolve the sub-problems, we just need to reshuffle the roles of the towers. The sub-problems can be recursively resolved. The pseudo-code is given below:

```
void TowersOfHanoi(int n, int source, int temp, int target){
// Move n disks from tower source to tower target
// Use temp tower for intermediate storage
if (n > 0) {
   TowersOfHanoi(n-1, source, target, temp); //Move (n-1) disks from the source tower to the
     temp tower
   cout << "Move disk from " << source << " to  " << target << endl;
   TowersOfHanoi(n-1, temp, source, target); //Move (n-1) disks from the temp tower to the
    target tower
   }
}
```

7.2.4 Time Complexity

- For the computer program to solve this problem, the number of lines printed for N disks is 2^N–1.
- Given Tower of Hanoi problem with 64 disks, it takes $2^{64}-1$ moves

- Assuming each move takes 1 sec, the solution would require 1.84×10^{19} sec or about 5.85×10^{11} years!
- Assuming the printer can print 1000 lines a second, the program would require 5×10^{8} years to complete!
- Hard problems means that the best-known algorithm for the problem is expensive in running time.

7.3 Decision Problems

A **decision problem** is an algorithmic problem that has two possible answers: yes, or no. The purpose is merely to decide whether a certain property holds for the problem's input. In general, many problems initially are optimization problems which is finding a minimum or a maximum. Therefore, each optimization problem need to be recast as a decision problem before we can further discuss how 'hard' the problem is by the theory of NP-completeness. If we can provide evidence that a decision problem is hard, we also provide evidence that its related optimisation problem is hard.

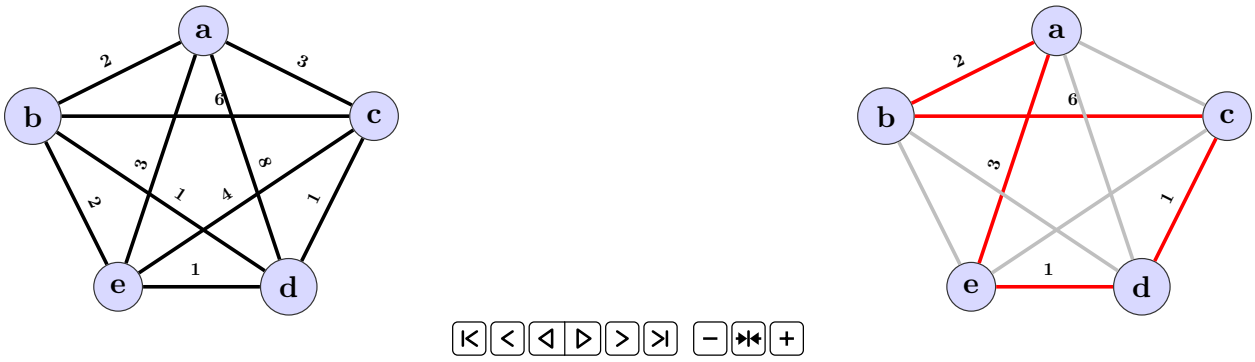
Definition 7.1 *The **class P** problem is a class of decision problems that are polynomially bounded. They can be solved in polynomial time by deterministic algorithms.*

The following are some example problems and their corresponding decision problems:

- The Shortest Path Problem: Is there a path from A to B with a distance less than k?
- The Minimum Spanning Tree Problem: Is there a spanning tree with a length of less than k?

7.3.1 Traveling-salesman problem (TSP)

Traveling-salesman Problem: Given a network of cities, **G**, the travelling salesman problem is to find a shortest possible route that visits each city and returns to the origin city.



- Decision problem is "Given a network of cities **G** and a number K, is there a route that visits each city and returns to the origin city with total cost of not more than K?"
- Consider all possible tours, terminating after we find a tour that costs no more than K.
- If we do not consider the direction and the cost between two cities is assumed the same for each opposite direction, there are total $\frac{(n-1)!}{2}$ permutations of routes

- The worst-case time complexity is $\mathcal{O}((n-1)!)$.
- If n=25, there are 3.10×10^{23} possible tours

7.4 NP, NP-Hard and NP-Completeness

Most of the algorithms we have learned so far have running time in $\mathcal{O}(n^k)$ for some constant k. These algorithms are known as **polynomial-time** algorithms. The **polynomially bounded** problems are polynomial-time solvable by deterministic algorithms. These problems are complexity class P.

7.4.1 Non-Determinism

Definition 7.2 *A non-deterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following:*

- *Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I but may be complete gibberish as well.*
- *Deterministic ("verification") stage: A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. Otherwise, it returns no or is allowed not to halt at all.*

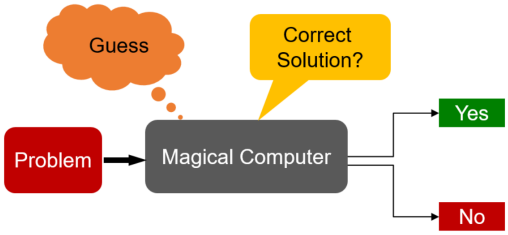
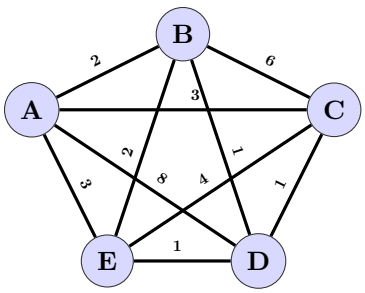
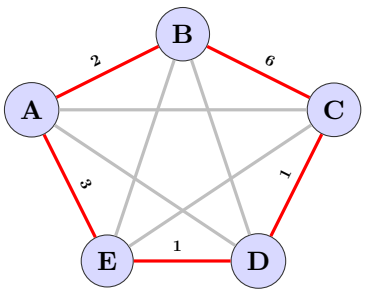


Figure 7.1: A magical Non-Deterministic Computer

A non-deterministic algorithm is said to be nondeterministic polynomial if the time efficiency of its verification stage is polynomial.

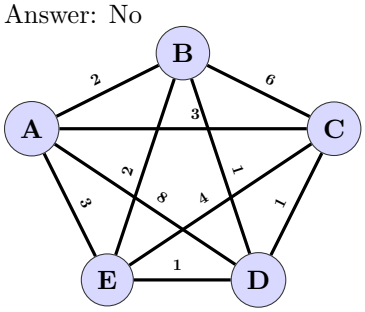
7.4.1.1 TSP: Is there a tour of all the cities with total cost of not more than 12?

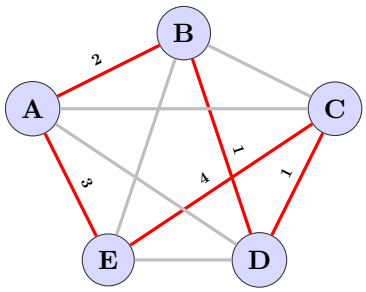




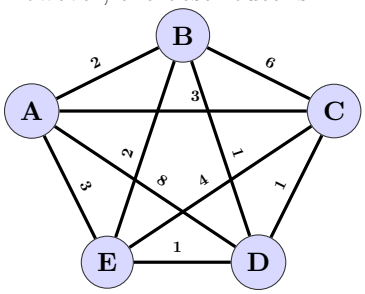
Guess: The distance of route ABCDEA is 13

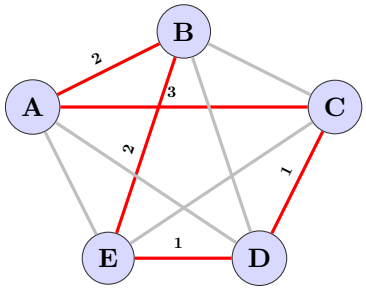
Answer: No





Guess: The distance of route ABDCEA is 11
Answer: Yes
However, the best route is ABEDCA which its distance is 9.





There is no known polynomial time algorithm with a deterministic computer to replace the polynomial time non-determinictic "guessing" algorithm.

7.4.2 P$\stackrel{?}{=}$NP

Is P = NP ? This is one of the most well-known open and unsolved problems in computer science. We are trying to find polynomial-time algorithms to solve any NP problem but no one can make it so far. This was introduced by Stepthen Cook in his seminal paper, "The complexity of theorem proving procedures" in 1971.

- Every problem which its solution can be quickly verified by a computer can also be quickly solved by a computer.
- One of the seven Millennium Prize Problems selected by the Clay Mathematics Institute to carry a USD1 million prize.

7.4.3 Reducibility

Definition 7.3 *A decision problem D_1 is said to be polynomially reducible to a decision problem, D_2, if there exists a function, $f(x)$, that transforms instances of D_1 to instances of D_2 such that:*

1. *$f(x)$ maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2*
2. *$f(x)$ is computable by a polynomial time algorithm. $f(x) \in P$*
3. *We can denote as $D_1 \leq_P D_2$*

Definition 7.4 *A class P of reductions is transitive if for all problems A, B, and C, $A \leq_P B$ and $B \leq_P C$ together imply $A \leq_P C$.*

Theorem 7.5 *If any NP-complete problem is polynomial-time solvable, then all NP-complete problems can be polynomial-time solvable and it implies P=NP. If any problem in NP is not polynomial-time solvable, then all NP-complete problems are not polynomial-time solvable.*

However, the conjecture above remains to be proved.

7.4.4 NP-Complete Problems

NP-complete (NPC) problems are a set of NP-problems

- which can be reduced in polynomial time to each other NP problem.
- which their solutions can still be verified in polynomial time

NPC problems are both in NP-Hard and NP.

7.4.5 NP-Hard Problems

NP-hard problems are those at least as hard as NP Problems. A problem H is defined as a NP-hard problem if every NP problem can be reduced in polynomial time to H. The NP-hard problems need not have solutions verifiable in polynomial time. It may not need to be a decision problem.

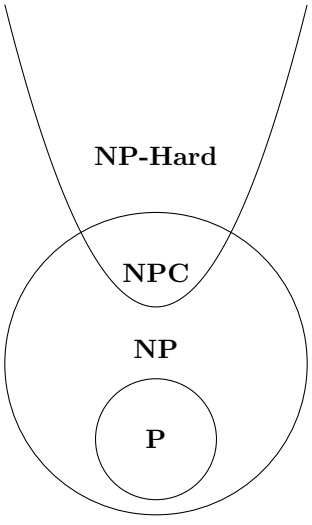


Figure 7.2: The definition of NP-hard and NP-complete problems are based on the assumption of $P \neq NP$. It is also most computer scientists view the relationship among P, NP and NPC.

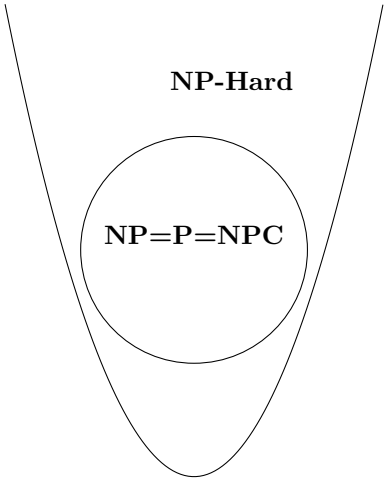
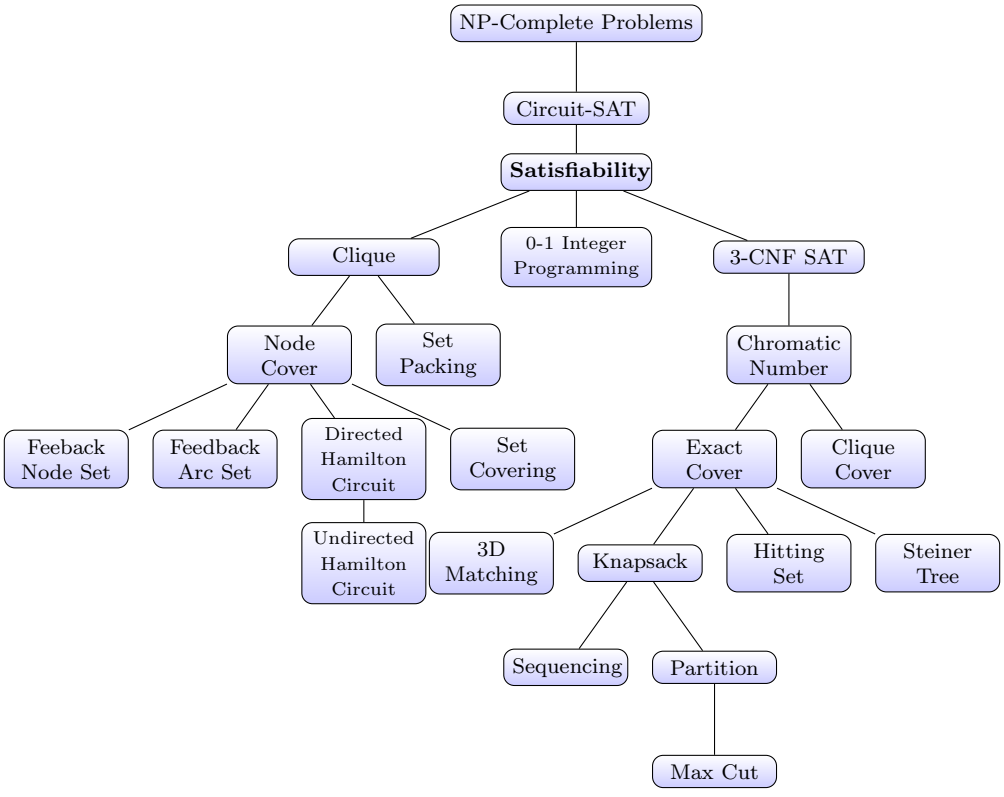


Figure 7.3: The relationship among P, NP and NPC if P=NP.

7.5 Classical NP-complete Problems



7.5.1 Satisfiability (SAT) problem

- The first NP-complete problem to be identified.

- A truth assignment for a boolean formula, ϕ is a set of values for the variables of ϕ
- A satisfying assignment is a truth that ϕ evaluates to 1 (true).
- SAT problem asks whether a given boolean formula is satisfiable (with a satisfying assignment)
- SAT = $\{\langle\phi\rangle : \phi$ is a satisfiable boolean formula$\}$

If ϕ has n variables, there are 2^n possible assignments. Intuitively, we can expect that a polynomial-time algorithm is unlikely to exist.

For example, the formula "a AND NOT b" is satisfiable because one can find the values a = TRUE and b = FALSE, which make (a AND NOT b) = TRUE.

In contrast, "a AND NOT a" is unsatisfiable.

7.5.2 Hamiltonian Circuit Problem

Determine whether a given graph has a Hamiltonian circuit – a path that starts and ends at the same vertex and passes through all the other vertices exactly once. TSP is finding the shortest Hamiltonian circuit in a complete graph with positive integer weights.

7.5.3 Knapsack problem

A knapsack of capacity C (a positive integer) and n objects with sizes $s_1, s_2, \ldots s_n$ and profits $p_1, p_2, \ldots p_n$ ($s_1, s_2, \ldots s_n$ and $p_1, p_2, \ldots p_n$ are positive integers).

- **Optimization problem**: Find the largest total profit of any subset of the objects that fits in the knapsack.

  The problem can be written as a integer linear programming problem:

$$\max_{\mathbf{x}} \sum_{i=1}^{n} p_i x_i$$

subject to

$$\sum_{i=1}^{n} s_i x_i \leq C$$

$$x_i \in \{1, 0\} \qquad i = 1, \ldots, n$$

- **Decision problem**: Given k, is there a subset of the objects that fits in the knapsack and has total profit at least k?

Example Application: C is amount of money to invest, sizes $s_1, s_2, \ldots s_n$ are investment amounts and profit is the expected return on investment.

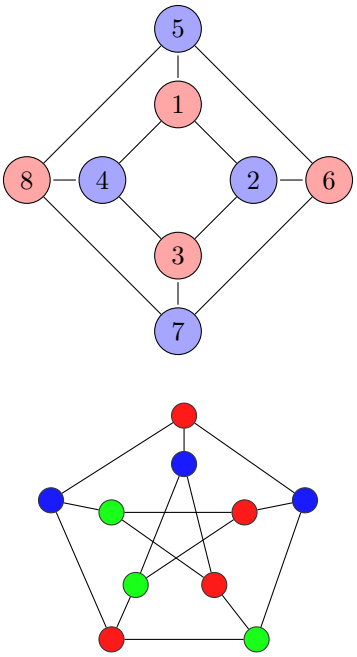
7.5.4 Chromatic Number/ Graph-colouring/ Vertex-colouring problem

For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.

- **Optimization problem**: What is the minimum number of colours required for colouring a graph G such that no two adjacent vertices share the same colour?
- **Decision problem**: Given G and a positive integer k, is there a colouring of G using at most k colours?

Some typical applications related to this problem are scheduling, resource allocation etc.

The following figures are examples of the vertext-colouring result.



7.5.5 The Clique Problem

The clique is a complete subgraph in an undirected graph.

The clique problem is the optimization problem of finding a clique of maximum size in a graph.

7.5.6 The Vertex-cover Problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ or both. That is , each vertex covers its incident edges and a vertex cover for G is a set of vertices that covers all the edges in E.

The vertex-over problem is to find a vertex cover of minimum size in a given graph.