Introduction:

- There are two types of problems:
  
  ✓ Problems whose time complexity is polynomial: $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$
    
    Examples: searching, sorting, merging, MST, etc.
  
  ✓ Problems with exponential time complexity: $O(2^n)$, $O(n!)$, $O(n^n)$, etc.
    
    Examples: TSP, n-queen, 0/1 knapsack, etc.

- Two classes of algorithms:
  
  ✓ P: The set of all problems, which can be solved by deterministic algorithms in polynomial time.
  
  ✓ NP: The set of all problems which can be solved by nondeterministic algorithms in polynomial time (NP: Nondeterministic Polynomial)
Non-deterministic algorithms:

- Unlike deterministic algorithms, each operation has several outcomes

- Example:
  - $x = \text{choice}(1..n)$;
  - $x$ may have any value between 1 and $n$
  - The time of this type of instruction is $O(1)$.
  - If there is a solution, the algorithm will terminate successfully; otherwise, it will terminate unsuccessfully.

Example1: Searching problem:

- input: $A(1..n)$ and $x$
- Output: index $j$ such that $A(j)=x$ if $x$ is in $A$ or $j=0$ if $x$ does not belong to $A$.

```
Ndsearch(A(1..n), x)
Integer j;
Begin
    J=choice(1..n);
    If A(j) =x
        Then
            Print(j);
        Else
            Print(0);
    Endif;
End;
```

- The complexity is $O(1)$;
Example 2: clique problem

- Definition: A maximal complete subgraph of a graph \( G=(V,E) \) is a clique.

- Input: - a graph \( G=(V,E) \) and an integer \( k \);
- Output: Determine if \( G \) has a clique of size at least \( k \).

- Brute force approach:

  ✓ The obvious way to solve this problem would be to subject all \( \binom{|V|}{k} \) subsets of \( V \) with cardinality \( k \) to test whether there is a clique.

- \( \text{Ndclique}(G,k) \);

  Integer \( I; X[1..n] \);
  Begin
  For \( I=1 \) to \( k \) do
    \( X[I] = \text{choice}(1..n) \);
  Endfor;
  If(\( X[1], X[2], ..., X[k] \)) is a clique
  Then
    print ("SUCCESS");
  else
    print("FAILURE");
  endif;
  end;
Example 3: Satisfiability: has a special role in the theory of computation.

- Definitions:
  - A literal is a boolean variable (its value is either true or false).
  - A logical formula is an expression that can be constructed using literals and the operations AND and OR.
  - The satisfiability problem is to determine if a logical formula is true for some assignment of truth values to the variables.

- Example:
  \[
  F = (x_1 \lor x_2) \land (\neg x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2)
  \]

  where \( x_i \in \{0,1\} \) \( 1 \leq i \leq 3 \) and \( C_i \) are called clauses

- Is there an assignment of truth values to the variables \( x_i \)'s that makes the formula \( F \) true ("Satisfies " it)?

- For \( n \) variables, one should consider \( 2^n \) possible assignments.
• Ndsatisfiability(E,n)
  Integer i;
  Begin
    For i=1 to n do
      x_i = choice(true, false);
    Endfor;
    If E(x_1, x_1, ..., x_n) is true
      Then
        Print "success";
      Else
        Print "Failure";
      Endif;
  End;

• Let n be the number of variables and p be the number of
  operations ANDs and Ors, the Ndsatisfiability takes
  O(max(n,p))

  Since in general p>>n, we have O(p).

NP-Complete problems:

• The theory of NP-completeness consists of two classes of
  problems:

  ✓ NP-complete problems
  ✓ NP-hard problems

• NP-hard problems:
  ✓ If an NP-hard problem can be solved in
    polynomial time then all NP-complete
    problems can be solved in polynomial
    time.
✓ In other words: A problem is NP-hard if every problem in NP is transformable to it

• NP-complete problems:

✓ A problem which is NP-complete will have the property that it can be solved in polynomial time iff all other NP-complete problems can be solved in polynomial time.

✓ In other words: A problem is NP-complete if it is both NP-hard and NP.

• Note:

✓ NP-complete problems are NP-hard
✓ All NP-hard problems are not NP-complete.

※ Open problem: $P = \text{NP}$

• Definition of reduction: $\propto$
✓ A1 is defined by T and A2, where T is a polynomial transformation

✓ A1 ≡ (T,A2) → P1 ∝ P2
       We say that P1 is reduced to P2.

✓ If P2 is polynomial, then P1 is also polynomial.

⇓ NP-complete:
• A problem is NP-complete:
  1) if A is NP
  2) every NP problem Q: Q ∝ P

⇓ Cook's theorem: Satisfiability is NP-Complete

• Theorem: If P1 ∝ P2 and P2 ∝ P3 → P1 ∝ P3

Proof:
$T$ is polynomial since $T_1$ and $T_2$ are polynomial $P_1 \preceq P_3$. 

$T$ is polynomial since $T_1$ and $T_2$ are polynomial $P_1 \preceq P_3$. 
Theorem: Given a problem P, If
1) P is NP and
3) \( \exists \) NP-complete problem Q: Q \( \propto \) P
Then P is NP-complete.

Proof:
We have to prove that P is NP and every NP problem R, R \( \propto \) P

- P is NP be definition
- Let R be in NP

R \( \propto \) Q since Q is NP-complete by definition

And Q \( \propto \) P by definition

R \( \propto \) Q and Q \( \propto \) P \( \Rightarrow \) R \( \propto \) P for every NP problem R.

Corollaire:
To prove a new problem P is NP-complete, we first prove that it is NP, and then find an NP-complete problem Q that reduces to P.
Example: Node cover problem:

- Definition: Given a graph $G=(V,E)$, a subset $S$ of $V$ is a node cover of $G$ iff all edges are incident to at least one vertex in $S$.

$S = \{1,2\}$