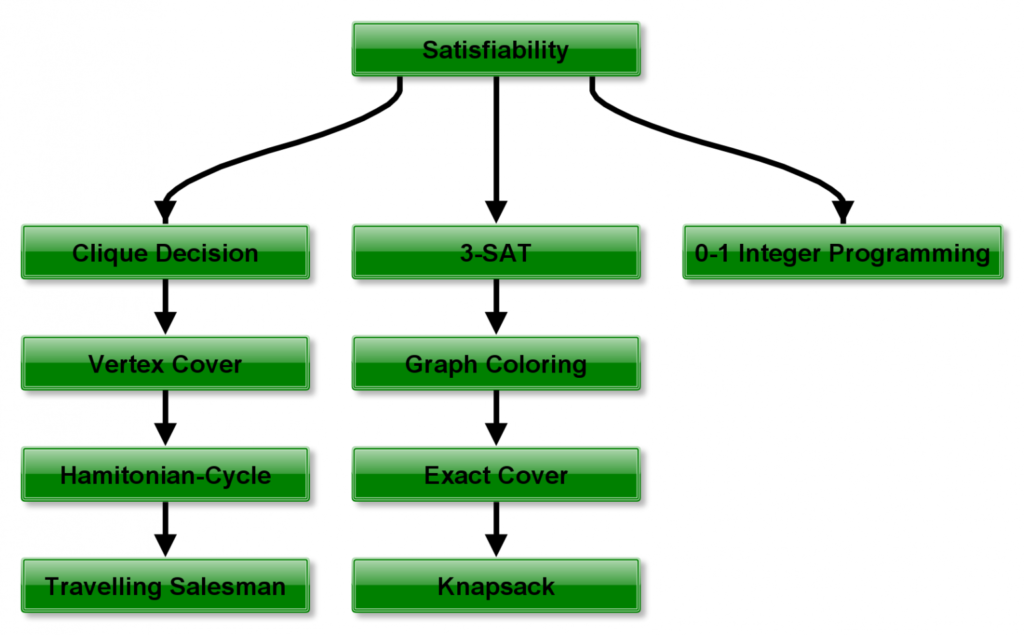
****

# Proof that SAT is NP Complete

SAT Problem: SAT(Boolean Satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given boolean formula. It asks whether the variables of a given boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called *satisfiable*. On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is *unsatisfiable*.

Problem: Given a boolean formula *f*, the problem is to identify if the formula *f* has a satisfying assignment or not.

Explanation: An instance of the problem is an input specified to the problem. An instance of the problem is a boolean formula *f*. Since an [**NP-complete**](https://www.geeksforgeeks.org/np-completeness-set-1/) problem is a problem which is both NP and NP-Hard, the proof or statement that a problem is NP-Complete consists of two parts:

1. *The problem itself is in NP class.*
2. *All other problems in NP class can be polynomial-time reducible to that.  
   (B is polynomial-time reducible to C is denoted as ≤ PC)*

If the 2nd condition is only satisfied then the problem is called NP-Hard.

But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time i.e., to show a problem is NP-complete then prove that the problem is in NP and any NP-Complete problem is reducible to that i.e. if B is NP-Complete and B ≤ PCFor C in NP, then C is NP-Complete. Thus, it can be verified that the SAT Problem is NP-Complete using the following propositions:

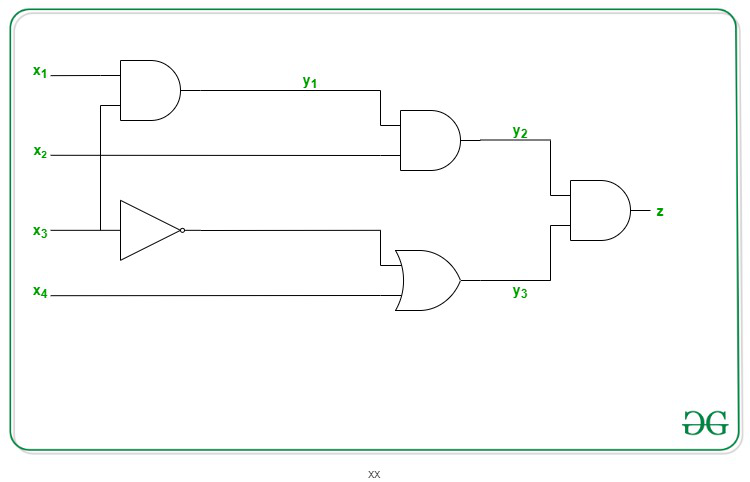
SAT is in NP:  
It any problem is in NP, then given a ‘certificate’, which is a solution to the problem and an instance of the problem(a boolean formula *f*) we will be able to check (identify if the solution is correct or not) certificate in polynomial time. This can be done by checking if the given assignment of variables satisfies the boolean formula.

SAT is NP-Hard:  
In order to prove that this problem is NP-Hard then reduce a known problem, Circuit-SAT in this case to our problem. The boolean circuit C can be corrected into a boolean formula as:

* For every input wire, add a new variable yi.
* For every output wire, add a new variable Z.
* An equation is prepared for each gate.
* These sets of equations are separated by ∩ values and adding ∩Z at the end.

This transformation can be done in linear time. The following propositions now hold:

* If there is a set of input, variable values satisfying the circuit then it can derive an assignment for the formula **f** that satisfies the formula. This can be simulated by computing the output of every gate in the circuit.
* If there is a satisfying assignment for the formula **f**, this can satisfy the boolean circuit after the removal of the newly added variables.

For Example: If below is the circuit then:  
[](https://media.geeksforgeeks.org/wp-content/uploads/20201008155204/idgf.jpg)

Therefore, the SAT Problem is NP-Complete.

**http://www.cs.ecu.edu/karl/6420/spr16/Notes/NPcomplete/3sat.html**

### 3-SAT

3-SAT is a restriction of SAT where each clause is required to have exactly 3 literals.

For example, formula

|  |
| --- |
| (¬*x* ∨ *y* ∨ ¬*w*) ∧ |
| (¬*y* ∨ *z* ∨ *w*) ∧ |
| (*x* ∨ ¬*z* ∨ *y*) |

has exactly 3 literals per clause.

### 3-SAT is NP-complete

Because 3-SAT is a restriction of SAT, it is not obvious that 3-SAT is difficult to solve. Maybe the restriction makes it easier.

But, in reality, 3-SAT is just as difficult as SAT; the restriction to 3 literals per clause makes no difference.

**Theorem.** 3-SAT is NP-complete.

**Proof.** There are two parts to the proof.

**Part (a).** We must show that 3-SAT is in NP. But we already showed that SAT is in NP. Surely, and nondeterministic algorith for SAT also works for 3-SAT; it does not care about the restriction to 3 literals per clause.

**Part (b).** We need to show, for every problem *X* in NP, *X* ≤ 3-SAT. But we can accomplish that by showing that SAT ≤p 3-SAT.

So our goal is to find a polynomial-time reduction from SAT to 3-SAT. The reduction is a polynomial-time computable function f that takes a clausal formula φ and yields a clausal formula φ′ with 3 literals per clause.

The reduction function works on one clause of φ at a time. Here is what it does on a clause *C*.

1. If *C* already has 3 literals, leave it alone.
2. If *C* has fewer than 3 literals, just duplicate one or two of the literals. For example, if *C* is (*x* ∨ ¬*y*), then the reduction replaces that clause by (*x* ∨ *x* ∨ ¬*y*).
3. Suppose that clause *C* is (ℓ1 ∨ ℓ2 ∨ … ∨ ℓ*n*), where *n* > 3. Create new variables λ1, λ2, etc., and replace *C* by clauses

|  |
| --- |
| (ℓ1 ∨ ℓ2 ∨ λ1) ∧ |
| (¬λ1 ∨ ℓ3 ∨ λ2) ∧ |
| (¬λ2 ∨ ℓ4 ∨ λ3) ∧ |
| (¬λ3 ∨ ℓ5 ∨ λ4) ∧ |
| … |
| (¬λ*n*−4 ∨ ℓ*n*−2 ∨ λ*n*−3) ∧ |
| (¬λ*n*−3 ∨ ℓ*n*−1 ∨ ℓ*n*) |

1. For example, clause (*x* ∨ ¬*y* ∨ *z* ∨ *u* ∨ ¬*v*) is replaced by

|  |
| --- |
| (*x* ∨ ¬*y* ∨ λ1) ∧ |
| (¬λ1 ∨ *z* ∨ λ2) ∧ |
| (¬λ2 ∨ *u* ∨ ¬*v*) |

1. Each clause gets its own new λ variables.
2. We need to show that doing this replacement does not affect whether the formula is satisfiable.
   1. Suppose that φ is satisfiable. Select an assignment *A* of truth values to φ's variables that makes φ true.

Since assignment *A* makes φ true, it must make at least one of the literals in clause *C* true.

Select a true literal ℓ*i* in *C*. Set λ*j* = true for *j* = 1, …, *i* − 2 and λ*j* = false for *j* = *i* − 1, ..., *n*−3. For example, if i = 4 and n = 7 then, with that assignment, the clauses in φ′ that represent clause *C* are as follows.

|  |
| --- |
| (ℓ1 ∨ ℓ2 ∨ true) ∧ |
| (false ∨ ℓ3 ∨ true) ∧ |
| (false ∨ ℓ4 ∨ false) ∧ |
| (true ∨ ℓ5 ∨ false) ∧ |
| (true ∨ ℓ6 ∨ ℓ7) |

Since ℓ4 is true, every clause is true.

So every clause can be satisfied, and φ′ is satisfiable.

* 1. Suppose that φ′ is satisfiable. Choose values for the variables to make φ′ true. We need to show that, no matter how the values of the λ variables are chosen, each original clause *C* must have one true literal. That is, at least one of ℓ1, …, ℓ*n* is true.

So suppose that all of ℓ1, …, ℓ*n* are false. From the first clause,

|  |
| --- |
| (ℓ1 ∨ ℓ2 ∨ λ1) |

λ1 must be true. But then, from the second clause,

|  |
| --- |
| (¬λ1 ∨ ℓ3 ∨ λ2) |

λ2 must also be true. Working through the clauses that correspond to original clause *C*, we see that λ1, …, λ*n*−3 must be true. But then the last clause,

|  |
| --- |
| (¬λ*n*−3 ∨ ℓ*n*−1 ∨ ℓ*n*) |

is false, contradicting the choice of variable values to make φ′ true. ◊

### 2-SAT

What if we restrict SAT even further, insisting that every clause have exactly 2 literals? Call that problem 2-SAT.

Then the problem becomes easy. There is a polynomial time algorithm to solve 2-SAT. The key to the algorithm is that, if you are looking at clause

(ℓ1 ∨ ℓ2)

and ℓ1 is false, then ℓ2 must be true. Choosing a value for a variable leads to other variable values being forced without much effort.

There is more to the algorithm than that, but that is the heart of it.

# Proof that 4 SAT is NP complete

4-SAT Problem: 4-SAT is a generalization of 3-SAT(k-SAT is SAT where each clause has k or fewer literals).

Problem Statement: Given a formula *f* in [**Conjunctive Normal Form**](https://www.geeksforgeeks.org/normal-and-principle-forms/)(CNF) composed of clauses, each of four literals, the problem is to identify whether there is a satisfying assignment for the formula *f*.

Explanation: An instance of the problem is an input specified to the problem. An instance of the 4-SAT problem is a CNF formula, and the task is to check whether there is a satisfying assignment for the formula. Since an [**NP-Complete**](https://www.geeksforgeeks.org/np-completeness-set-1/) is a problem which is both in NP and NP-hard, the proof for the statement that a problem is NP-Complete consists of two parts:

1. *The problem itself is in NP class.*
2. *All other problems in NP class can be polynomial-time reducible to that.  
   (B is polynomial-time reducible to C is denoted as B ≤ PC)*

If the 2nd condition is only satisfied then the problem is called [**NP-Hard**](https://www.geeksforgeeks.org/tag/nphard/).

But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time. That is why if we want to show a problem is NP-Complete we just show that the problem is in NP and any NP-Complete problem is reducible to that then we are done, i.e., if B is NP-Complete and B ≤ C. For C in NP, then C is NP-Complete. Thus, it can be verified that the 4-SAT Problem is NP-Complete using the following two propositions:

1. 4-SAT problem is in NP:  
   If any problem is in NP, then, given a ‘certificate’, which is a solution to the problem and an instance of the problem(a formula *f*, in this case), it can be verified(check whether the solution given is correct or not) that the certificate in polynomial time. This can be done in the following way:  
   Given an assignment for the variables belonging to the formula *f*, the assignment can be verified in linear time, if it satisfies the formula or not.
2. 4-SAT problem is NP-Hard:  
   In order to prove that the 4-SAT problem is NP-Hard, deduce a reduction from a known NP-Hard problem to this problem. Deduce a reduction from which the 3-SAT problem can be reduced to the 4-SAT problem. For each clause of the 3-SAT formula *f*, for example, a literal a and its corresponding complement a’ should be added to the formula. Let there be a clause c, such that c = u V v’ V w    
   To convert it in 4-SAT, we convert c to c’, such that,    
   c’ = (u V v’ V w V a) AND (u V v’ V w V a’).   
   After simulating this conversion, two properties hold :
   1. If 3-SAT has a satisfiable assignment, which means, every clause evaluates to true for a specific set of literal values, then 4-SAT will also hold, because each clause-set is formed by a combination of a literal and its complement, whose value won’t make any difference.
   2. If 4-SAT is satisfiable for any (u V v V w V a) and (u V v V w V a’), then 3-SAT is also satisfiable because a and a’ are complement, which indicates that the formula is satisfiable due to some other literal except a too.

Therefore, following the above propositions, the 4-SAT problem is NP-Complete.

**OTHER WAYS TO SAT** [**https://www.baeldung.com/cs/cook-levin-theorem-3sat**](https://www.baeldung.com/cs/cook-levin-theorem-3sat)

# Proof that Clique Decision problem is NP-Complete

**Prerequisite:** [NP-Completeness](https://www.geeksforgeeks.org/np-completeness-set-1/)

A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph. The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices. This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph or not.­­

To see diagram use link

https://www.geeksforgeeks.org/proof-that-clique-decision-problem-is-np-complete/

To prove that a problem is NP-Complete, we have to show that it belongs to both NP and NP-Hard Classes. (Since NP-Complete problems are NP-Hard problems which also belong to NP)

.

**The Clique Decision Problem belongs to NP** – If a problem belongs to the NP class, then it should have polynomial-time verifiability, that is given a certificate, we should be able to verify in polynomial time if it is a solution to the problem.

**Proof:**

1. Certificate – Let the certificate be a set S consisting of nodes in the clique and S is a subgraph of G.
2. Verification – We have to check if there exists a clique of size k in the graph. Hence, verifying if number of nodes in S equals k, takes O(1) time. Verifying whether each vertex has an out-degree of (k-1) takes O(k2) time. (Since in a complete graph, each vertex is connected to every other vertex through an edge. Hence the total number of edges in a complete graph = kC2 = k\*(k-1)/2 ). Therefore, to check if the graph formed by the k nodes in S is complete or not, it takes O(k2) = O(n2) time (since k<=n, where n is number of vertices in G).

Therefore, the Clique Decision Problem has polynomial time verifiability and hence belongs to the NP Class.

**The Clique Decision Problem belongs to NP-Hard**– A problem L belongs to NP-Hard if every NP problem is reducible to L in polynomial time. Now, let the Clique Decision Problem by C. To prove that C is NP-Hard, we take an already known NP-Hard problem, say S, and reduce it to C for a particular instance. If this reduction can be done in polynomial time, then C is also an NP-Hard problem. The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by the [Cook’s theorem](https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem). Therefore, every problem in NP can be reduced to S in polynomial time. Thus, if S is reducible to C in polynomial time, every NP problem can be reduced to C in polynomial time, thereby proving C to be NP-Hard.

Proof that the Boolean Satisfiability problem reduces to the Clique Decision Problem  
Let the boolean expression be – F = (x1 v x2) ^ (x1‘ v x2‘) ^ (x1 v x3)  where x1, x2, x3 are the variables, ‘^’ denotes logical ‘and’, ‘v’ denotes logical ‘or’ and x’ denotes the complement of x. Let the expression within each parentheses be a clause. Hence we have three clauses – C1, C2 and C3. Consider the vertices as – <x1, 1>; <x2, 1>; <x1’, 2>; <x2’, 2>; <x1, 3>; <x3, 3> where the second term in each vertex denotes the clause number they belong to. We connect these vertices such that –

1. No two vertices belonging to the same clause are connected.
2. No variable is connected to its complement.

Thus, the graph G (V, E) is constructed such that – V = { <a, i> | a belongs to Ci } and E = { ( <a, i>, <b, j> ) | i is not equal to j ; b is not equal to a’ } Consider the subgraph of G with the vertices <x2, 1>; <x1’, 2>; <x3, 3>. It forms a clique of size 3 (Depicted by dotted line in above figure) . Corresponding to this, for the assignment – <x1, x2, x3> = <0, 1, 1>  F evaluates to true. Therefore, if we have k clauses in our satisfiability expression, we get a max clique of size k and for the corresponding assignment of values, the satisfiability expression evaluates to true. Hence, for a particular instance, the satisfiability problem is reduced to the clique decision problem.

Therefore, the Clique Decision Problem is NP-Hard.

Conclusion  
The Clique Decision Problem is NP and NP-Hard. Therefore, the Clique decision problem is NP-Complete

**or**

# Proof that Clique Decision problem is NP-Complete | Set 2

**Prerequisite:** [NP-Completeness](https://www.geeksforgeeks.org/np-completeness-set-1/), [Clique problem](https://en.wikipedia.org/wiki/Clique_problem).

A clique in a graph is a set of vertices where each vertex shares an edge with every other vertex. Thus, a clique in a graph is a subgraph which is a complete graph.  
**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**.

**Explanation:**  
An instance of the problem is an input specified to the problem. An instance of the Clique problem is a graph G (V, E) and a positive integer K, and the problem is to check whether a clique of size K exists in G. Since an NP-Complete problem, by definition, is a problem which is both in NP and NP-hard, the proof for the statement that a problem is NP-Complete consists of two parts:

1. *The problem itself is in NP class*
2. *All other problems in NP class can be polynomial-time reducible to that.  
   (B is polynomial-time reducible to C is denoted as )*

If the 2nd condition is only satisfied then the problem is called **NP-Hard**.

But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time. That is why if we want to show a problem is NP-Complete we just show that the problem is in **NP** and any **NP-Complete** problem is reducible to that, then we are done, i.e. if **B** is **NP-Complete** and  for **C** in **NP**, then **C**is **NP-Complete**.

1. **Clique Problem is in NP**  
   If any problem is in NP, then, given a ‘certificate’, which is a solution to the problem and an instance of the problem (a graph G and a positive integer K, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

The certificate is a **subset V’** of the vertices, which comprises the vertices belonging to the clique. We can validate this solution by checking that each pair of vertices belonging to the solution are adjacent, by simply verifying that they share an edge with each other. This can be done in polynomial time, that is**O(V +E)** using the following strategy for graph**G(V, E)**:

flag=true

For every pair {u, v} in the subset V’:

Check that these two

vertices {u, v} share an edge

If there is no edge,

set flag to false and break

If flag is true:

Solution is correct

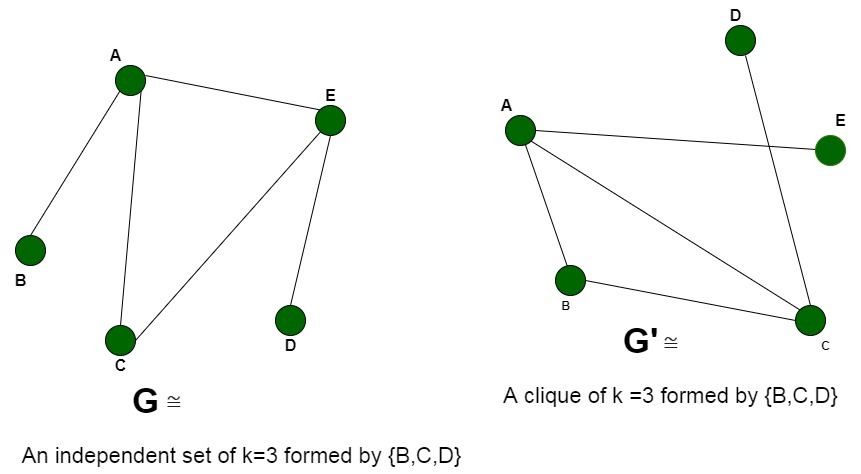
Else:

Solution is incorrect

1. **Clique Problem is NP-Hard**  
   To prove that the clique problem is NP-Hard, we take the help of a problem that is already NP-Hard and show that this problem can be reduced to the Clique problem.  
   For this, we consider the **Independent Set problem**, which is **NP-Complete** (and hence **NP-Hard**). Every instance of the independent set problem consisting of the graph **G (V, E)** and an integer **K** can be converted to the required graph **G’ (V’, E’)** and **K’** of the Clique problem. We will construct the graph G’ by the following modifications:  
   **V’ =V,** that is all the vertices of graph G are a part of the graph G’  
   **E’**= complement of the edges E, that is, the edges not present in the original graph G.  
   The graph **G’** is the complementary graph of G. The time required to compute the complementary graph **G’**requires a traversal over all the vertices and edges.  
   ***Time complexity:*** O (V+E)

We will now prove that the problem of computing the clique indeed boils down to the computation of the independent set. The reduction can be proved by the following two propositions:

* + Let us assume that the graph G contains a clique of size **K**. The presence of clique implies that there are **K**vertices in **G**, where each of the vertices is connected by an edge with the remaining vertices. This further shows that since these edges are contained in**G**, therefore they can’t be present in **G’**. As a result, these K vertices are not adjacent to each other in **G’** and hence form an Independent Set of size ***K***.
  + We assume that the complementary graph **G’** has an independent set of vertices of size**K’**. None of these vertices shares an edge with any other vertices. When we complement the graph to obtain**G,** these**K** vertices will share an edge and hence, become adjacent to each other. Therefore, the graph **G** will have a clique of size **K**.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200514000635/Untitled-Diagram-103.jpg)

Thus, we can say that there is a clique of size **K** in graph**G** if there is an independent set of size **K** in **G’** (complement graph). Therefore, any instance of the clique problem can be reduced to an instance of the Independent Set problem. Thus, the clique problem is**NP-Hard.**

**Conclusion:**

*Hence, the Clique Decision problem is NP-Complete*

# Proof that vertex cover is NP complete

Prerequisite – [Vertex Cover Problem](https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/), [NP-Completeness](https://www.geeksforgeeks.org/np-completeness-set-1/)  
**Problem –** Given a graph G(V, E) and a positive integer k, the problem is to find whether there is a subset V’ of vertices of size at most k, such that every edge in the graph is connected to some vertex in V’.

**Explanation –**  
First let us understand the notion of an instance of a problem. An instance of a problem is nothing but an input to the given problem. An instance of the Vertex Cover problem is a graph G (V, E) and a positive integer k, and the problem is to check whether a vertex cover of size at most k exists in G. Since an NP Complete problem, by definition, is a problem which is both in NP and NP hard, the proof for the statement that a problem is NP Complete consists of two parts:

1. **Proof that vertex cover is in NP –**  
   If any problem is in NP, then, given a ‘certificate’ (a solution) to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

The certificate for the vertex cover problem is a subset V’ of V, which contains the vertices in the vertex cover. We can check whether the set V’ is a vertex cover of size k using the following strategy (for a graph G(V, E)):

let count be an integer

set count to 0

for each vertex v in V’

remove all edges adjacent to v from set E

increment count by 1

if count = k and E is empty

then

the given solution is correct

else

the given solution is wrong

It is plain to see that this can be done in polynomial time. Thus the vertex cover problem is in the class NP.

1. **Proof that vertex cover is NP Hard –**  
   To prove that Vertex Cover is NP Hard, we take some problem which has already been proven to be NP Hard, and show that this problem can be reduced to the Vertex Cover problem. For this, we consider the Clique problem, which is NP Complete (and hence NP Hard).

*“In computer science, the clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph.”*

Here, we consider the problem of finding out whether there is a clique of size k in the given graph. Therefore, an instance of the clique problem is a graph G (V, E) and a non-negative integer k, and we need to check for the existence of a clique of  
size k in G.

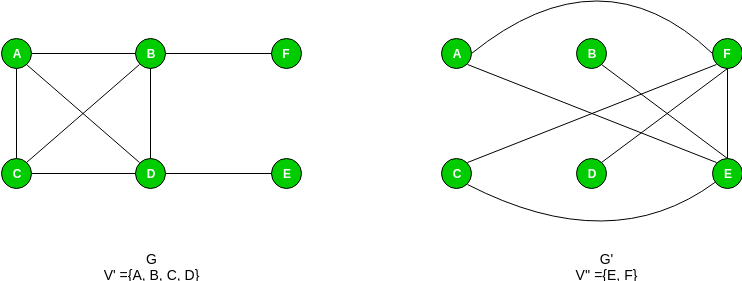
Now, we need to show that any instance (G, k) of the Clique problem can be reduced to an instance of the vertex cover problem. Consider the graph G’ which consists of all edges not in G, but in the complete graph using all vertices in G. Let us call this the complement of G. Now, the problem of finding whether a clique of size k exists in the graph G is the same as the problem of finding whether there is a vertex cover of size |V| – k in G’. We need to show that this is indeed the case.

Assume that there is a clique of size k in G. Let the set of vertices in the clique be V’. This means |V’| = k. In the complement graph G’, let us pick any edge (u, v). Then at least one of u or v must be in the set V – V’. This is because, if both u and v were from the set V’, then the edge (u, v) would belong to V’, which, in turn would mean that the edge (u, v) is in G. This is not possible since (u, v) is not in G. Thus, all edges in G’ are covered by vertices in the set V – V’.

Now assume that there is a vertex cover V’’ of size |V| – k in G’. This means that all edges in G’ are connected to some vertex in V’’. As a result, if we pick any edge (u, v) from G’, both of them cannot be outside the set V’’. This means, all  
edges (u, v) such that both u and v are outside the set V’’ are in G, i.e., these edges constitute a clique of size k.

Thus, we can say that there is a clique of size k in graph G if and only if there is a vertex cover of size |V| – k in G’, and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem. Thus, vertex cover is NP Hard. Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

To understand the proof, consider the following example graph and its complement:



See for – [Proof that Hamiltonian Path is NP-Complete](https://www.geeksforgeeks.org/proof-hamiltonian-path-np-complete/)

**Proof that Hamiltonian Cycle is NP-Complete**

**Prerequisite:** [NP-Completeness](https://www.geeksforgeeks.org/np-completeness-set-1/), [Hamiltonian cycle](https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/).

**Hamiltonian Cycle:** A cycle in an undirected graph G =(V, E) which traverses every vertex exactly once.

**Problem Statement:**Given a graph G(V, E), the problem is to determine if the graph contains a Hamiltonian cycle consisting of all the vertices belonging to V.  
**Explanation –**  
An instance of the problem is an input specified to the problem. An instance of the Independent Set problem is a graph G (V, E), and the problem is to check whether the graph can have a Hamiltonian Cycle in G.  
Since an NP-Complete problem, by definition, is a problem which is both in NP and NP-hard, the proof for the statement that a problem is NP-Complete consists of two parts:

1. *The problem itself is in NP class.*
2. *All other problems in NP class can be polynomial-time reducible to that.  
   (B is polynomial-time reducible to C is denoted as )*

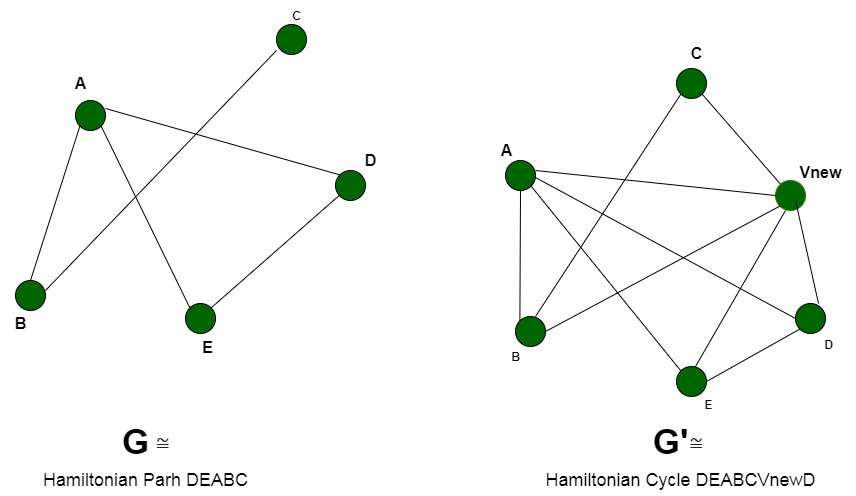
If the 2nd condition is only satisfied then the problem is called **NP-Hard**.

But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time. That is why if we want to show a problem is NP-Complete, we just show that the problem is in **NP** and if any **NP-Complete**problem is reducible to that, then we are done, i.e. if B is NP-Complete and  for C in NP, then C is NP-Complete.

1. **Hamiltonian Cycle is in NP**  
   If any problem is in NP, then, given a *‘certificate’*, which is a solution to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.  
   The certificate is a sequence of vertices forming Hamiltonian Cycle in the graph. We can validate this solution by verifying that all the vertices belong to the graph and each pair of vertices belonging to the solution are adjacent. This can be done in polynomial time, that is **O(V +E)** using the following strategy for graph G(V, E):
2. flag=true
3. For every pair {u, v} in the subset V’:
4. Check that these two have an edge between them
5. If there is no edge, set flag to false and break
6. If flag is true:
7. Solution is correct
8. Else:
9. Solution is incorrect
10. **Hamiltonian Cycle is NP Hard**  
    In order to prove the Hamiltonian Cycle is NP-Hard, we will have to reduce a known NP-Hard problem to this problem. We will carry out a reduction from the [Hamiltonian Path problem](https://www.geeksforgeeks.org/proof-hamiltonian-path-np-complete/) to the Hamiltonian Cycle problem.  
    Every instance of the Hamiltonian Path problem consisting of a graph**G =(V, E)** as the input can be converted to Hamiltonian Cycle problem consisting of graph **G’ = (V’, E’)**. We will construct the graph G’ in the following way:
    * **V’** = Add vertices V of the original graph G and add an additional vertex **Vnew** such that all the vertices connected of the graph are connected to this vertex. The number of vertices increases by 1, **V’ =V+1**.
    * **E’** = Add edges E of the original graph G and add new edges between the newly added vertex and the original vertices of the graph. The number of edges increases by the number of vertices V, that is, **E’=E+V**.

The new graph G’ can be obtained in polynomial time, by adding new edges to the new vertex, that requires O(V) time. This reduction can be proved by the following two claims:

* + Let us assume that the graph G contains a hamiltonian path covering the**V** vertices of the graph starting at a random vertex say **Vstart** and ending at Vend, now since we connected all the vertices to an arbitrary new vertex **Vnew** in G’.  
    We extend the original Hamiltonian Path to a Hamiltonian Cycle by using the edges **Vend**to **Vnew** and Vnew to Vstart respectively. The graph **G’** now contains the closed cycle traversing all vertices once.
  + We assume that the graph **G’** has a *Hamiltonian Cycle* passing through all the vertices, inclusive of **Vnew**. Now to convert it to a *Hamiltonian Path*, we remove the edges corresponding to the vertex **Vnew** in the cycle. The resultant path will cover the vertices V of the graph and will cover them exactly once.



Thus we can say that the graph **G’** contains a *Hamiltonian Cycle* iff graph **G** contains a *Hamiltonian Path*. Therefore, any instance of the *Hamiltonian Cycle*problem can be reduced to an instance of the *Hamiltonian Path* problem. Thus, the *Hamiltonian Cycle* is **NP-Hard**.

**Conclusion:** Since, the *Hamiltonian Cycle* is both, a **NP-Problem** and **NP-Hard**. Therefore, it is a **NP-Complete** problem.

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle ≤p TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***G' = (V, E')***, where

E′={(i,j):i,j∈Vandi≠jE′={(i,j):i,j∈Vandi≠j

Thus, the cost function is defined as follows −

t(i,j)={01if(i,j)∈E otherwiset(i,j)={0if(i,j)∈E1 otherwise

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G'*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G'***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G'*** has a tour of **0** cost.

Conversely, we assume that ***G'*** has a tour ***h'*** of cost at most **0**. The cost of edges in ***E'*** are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of ***h'*** is **0**. We therefore conclude that ***h'*** contains only edges in ***E***.

We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G'*** has a tour of cost at most **0**. TSP is NP-complete.

 is to find if a clique of size k exists in the given graph or not.