**NP-Completeness | Set 1 (Introduction)**

O(n^k) where k constant IT IS A polynomial time algo

variable^constant (n^k)-> linearly or polynomial

variable or constant^variable (k ^n) -> exponential

We have been writing about efficient algorithms to solve complex problems, like [shortest path](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/), [Euler graph](https://www.geeksforgeeks.org/eulerian-path-and-circuit/), [minimum spanning tree](https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/), etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

**Can all computational problems be solved by a computer?** There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source [Halting Problem](http://en.wikipedia.org/wiki/Halting_problem)).   
Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exists for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved. 

.

**What are NP, P, NP-complete and NP-Hard problems?**

P is a set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

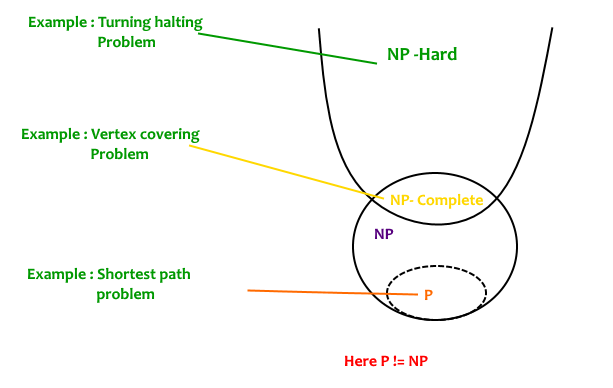
O(n^k) where k constant

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time OR NP is a class of problem which can be solved in DTM by exponential time OR NP is a class of problem which can be solved in NDTM by Polynomial TIME . P is subset of NP (any problem that can be solved by a deterministic machine in polynomial time can also be solved by a non-deterministic machine in polynomial time).   
Informally, NP is a set of decision problems that can be solved by a polynomial-time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source [Ref 1](http://www.youtube.com/watch?v=moPtwq_cVH8)).

NP-complete problems are the hardest problems in the NP set.  A decision problem L is NP-complete if:   
**1) L belongs NP or to** L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).   
**2)**

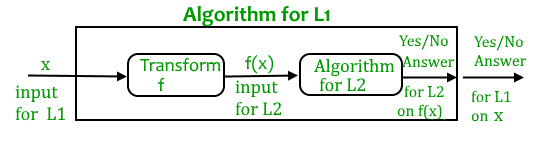
**B<=p A or** Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, the NP-Complete set is also a subset of the NP-Hard set.



**Decision vs Optimization Problems**   
NP-completeness applies to the realm of decision problems.  It was set up this way because it’s easier to compare the difficulty of decision problems than that of optimization problems.   In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source [Ref 2](http://uic.edu.hk/~taochen/teaching/comp3040/week13/l17.pdf)).   
For example, consider the [vertex cover problem](http://en.wikipedia.org/wiki/Vertex_cover) (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

**What is Reduction?**   
Let L1 and L2 be two decision problems. Suppose algorithm A2 solves L2. That is, if y is an input for L2 then algorithm A2 will answer Yes or No depending upon whether y belongs to L2 or not.   
The idea is to find a transformation from L1 to L2 so that algorithm A2 can be part of an algorithm A1 to solve L1.



Learning reduction, in general, is very important. For example, if we have library functions to solve certain problems and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find the minimum product path in a given directed graph where the product of path is the multiplication of weights of edges along the path. If we have code for Dijkstra’s algorithm to find the shortest path, we can take the log of all weights and use Dijkstra’s algorithm to find the minimum product path rather than writing a fresh code for this new problem.

**How to prove that a given problem is NP complete?**   
From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete.  By definition, it requires us to that show every problem in NP in polynomial time reducible to L.   Fortunately, there is an alternate way to prove it.   The idea is to take a known NP-Complete problem and reduce it to L.  If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

**What was the first problem proved as NP-Complete?**   
There must be some first NP-Complete problem proved by definition of NP-Complete problems.  [SAT (Boolean satisfiability problem)](http://en.wikipedia.org/wiki/Boolean_satisfiability_problem)is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don’t know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem is NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

**Difference between NP hard and NP complete problem**

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

**NP Problem:**   
The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by [Non-Deterministic Machine](https://www.geeksforgeeks.org/difference-between-deterministic-and-non-deterministic-algorithms/) in polynomial time.

[**NP-Hard Problem**](https://www.geeksforgeeks.org/tag/nphard/)**:**   
A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

[**NP-Complete Problem**](https://www.geeksforgeeks.org/algorithms-gq/np-complete-gq/)**:**

A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic  Turing machine can solve NP-Complete problem in polynomial time.

**Difference between NP-Hard and NP-Complete**:

| NP-hard | NP-Complete |
| --- | --- |
| NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time. | NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time. |
| To solve this problem, it do not have to be in NP . | To solve this problem, it must be both NP and NP-hard problems. |
| Do not have to be a Decision problem. | It is exclusively a Decision problem. |
| Example: Halting problem, Vertex cover problem, etc. | Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, Circuit-satisfiability problem, etc. |

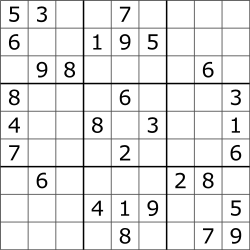
**P, NP, NP-Hard and NP-Complete Problems**

**P (Polynomial) problems**

P problems refer to problems where an algorithm would take a polynomial amount of time to solve, or where Big-O is a polynomial (i.e. O(1), O(n), O(n²), etc). These are problems that would be considered ‘easy’ to solve, and thus do not generally have immense run times.

NP (Non-deterministic Polynomial) Problems

NP problems were a little harder for me to understand, but I think this is what they are. In terms of solving a NP problem, the run-time would not be polynomial. It would be something like O(n!) or something much larger. However, this class of problems can be given a specific solution, and checking the solution would have a polynomial run-time. An example that helped me understand this a little better was a Sudoku game.



In order to solve this entire puzzle, the algorithm would have to check each 3x3 matrix to see which numbers are missing, then each row, then each column, and then make sure there are no repetitions of any digit from 0–9. This becomes more complex because the number of digits that are missing is inconsistent in each row, column, and matrix (i.e. top-left matrix is missing 4 digits while top-right matrix is missing 8 digits). Solving this problem would not have a polynomial run-time. However, if you were to feed this puzzle with a possible solution, it would be much less complex to check if there are any repetitions in the rows, columns and matrices. This is a simple check which would have a polynomial run-time.

In essence, NP class problems don’t have a polynomial run-time to *solve*, but have a polynomial run-time to *verify*solutions (difficult to solve, easy to check a given answer).

Reduction

I can’t really explain this one outside of using examples, so: we have two problems, A and B, and we know problem B is a P class problem. If problem A can be reduced, or converted to problem B, and this reduction takes a polynomial amount of time, then we can say that A is also a P class problem (A is reducible to B).

This concept is important to understand for the other two classes of problems.

NP-Hard Problems

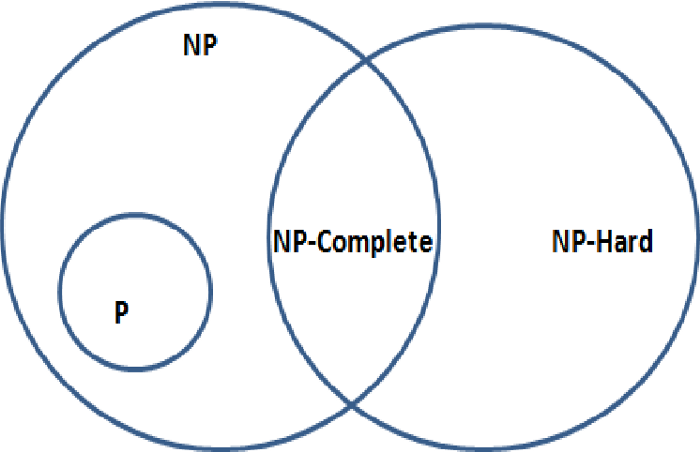
A problem is classified as NP-Hard when an algorithm for solving it can be translated to solve *any*NP problem. Then we can say, this problem is *at least* as hard as any NP problem, but it could be much harder or more complex.

NP-Complete Problems

NP-Complete problems are problems that live in both the NP and NP-Hard classes. This means that NP-Complete problems can be verified in polynomial time and that any NP problem can be reduced to this problem in polynomial time.

Below is a venn diagram of the different class spaces.

https://miro.medium.com/max/60/0*nedxMbwYe3CzPGga.png?q=20



NEW WAY

Deterministic vs. Nondeterministic Computations

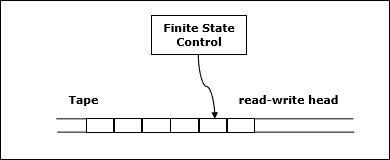
To understand class **P** and **NP**, first we should know the computational model. Hence, in this chapter we will discuss two important computational models.

Deterministic Computation and the Class P

Deterministic Turing Machine

One of these models is deterministic one-tape Turing machine. This machine consists of a finite state control, a read-write head and a two-way tape with infinite sequence.

Following is the schematic diagram of a deterministic one-tape Turing machine.



A program for a deterministic Turing machine specifies the following information −

* A finite set of tape symbols (input symbols and a blank symbol)
* A finite set of states
* A transition function

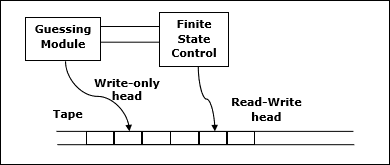
In algorithmic analysis, if a problem is solvable in polynomial time by a deterministic one tape Turing machine, the problem belongs to P class.

Nondeterministic Computation and the Class NP

Nondeterministic Turing Machine

To solve the computational problem, another model is the Non-deterministic Turing Machine (NDTM). The structure of NDTM is similar to DTM, however here we have one additional module known as the guessing module, which is associated with one write-only head.

Following is the schematic diagram.



If the problem is solvable in polynomial time by a non-deterministic Turing machine, the problem belongs to NP class.

Design and Analysis P and NP Class

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows −

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

* Finding the minimum number of colors needed to color a given graph.
* Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

* Whether a given graph can be colored by only 4-colors.
* Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer ‘yes’ for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size ***n***, if worst-case time complexity of an algorithm is ***O(nk)***, where ***k*** is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows −

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time ***O(nk)*** in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial ***p(n)*** such that the algorithm can solve any instance of size **n** in a time ***O(p(n))***.

Problem requiring ***Ω(n50)*** time to solve are essentially intractable for large ***n***. Most known polynomial time algorithm run in time ***O(nk)*** for fairly low value of ***k***.

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren’t asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.

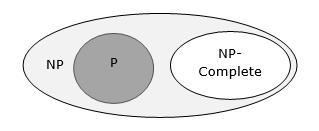
It is not known whether ***P = NP***. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If ***P ≠ NP***, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it’s easy to find a solution for the problem. The problem belongs to **NP**, if it’s easy to check a solution that may have been very tedious to find.

NP Hard and NP-Complete Classes

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

* **B** is in NP
* Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

* Determining whether a graph has a Hamiltonian cycle
* Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

* The circuit-satisfiability problem
* Set Cover
* Vertex Cover
* Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

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)∈Eotherwiset(i,j)={0if(i,j)∈E1otherwise

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.

P, NP, NP-Complete and NP-Hard Problems in Computer Science

* [**Algorithms**](https://www.baeldung.com/cs/category/algorithms)
* [**Core Concepts**](https://www.baeldung.com/cs/category/core-concepts)
* [**NP-Complete**](https://www.baeldung.com/cs/tag/np-complete)

**1. Introduction**

In computer science, there exist several famous unresolved problems, and  is one of the most studied ones. Until now, the answer to that problem is mainly “no”. And, this is accepted by the majority of the [academic world](https://en.wikipedia.org/wiki/P_versus_NP_problem). We probably wonder why this problem is still not resolved.

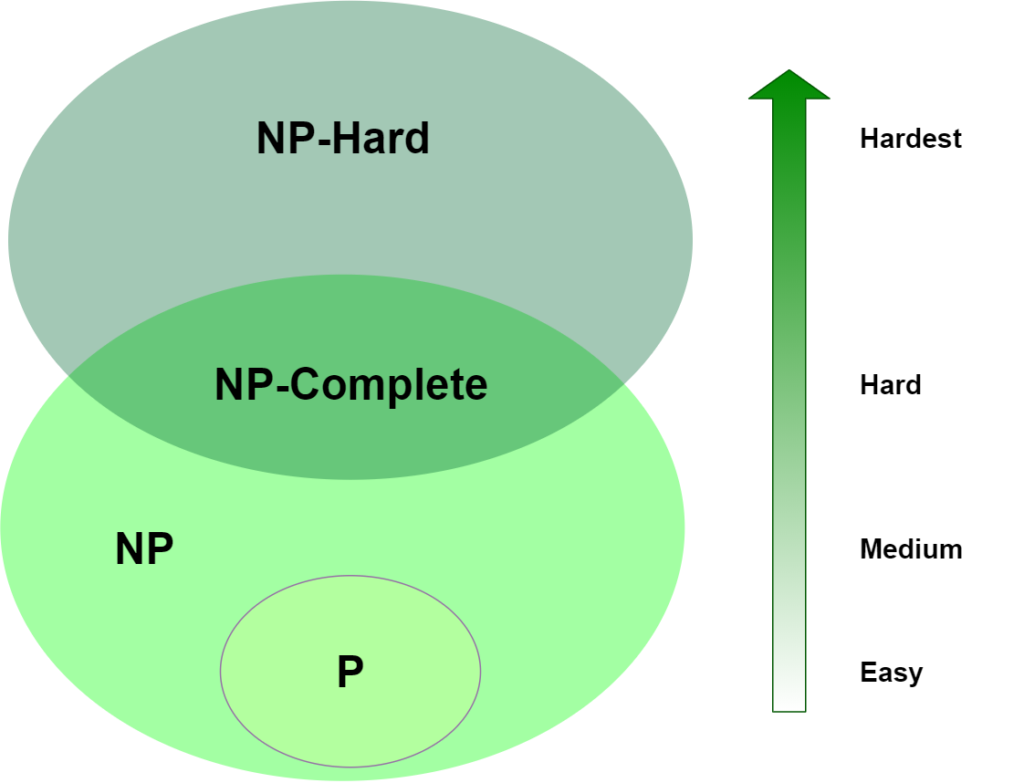
In this tutorial, we explain the details of this academic problem.  Moreover, we also show both  and  problems. Then, we also add definitions of  and . And in the end, hopefully, we would have a better understanding of why  is still an open problem.

**2. Classification**

To explain , *,*and others, let’s use the same mindset that we use to classify problems in real life. While we could use a wide range of terms to classify problems, in most cases we use an “Easy-to-Hard” scale.

Now,**in theoretical computer science, the classification and complexity of common problem definitions have two major sets;**  which is “Polynomial” time and which “Non-deterministic Polynomial” time. There are also  and  sets, which we use to express more sophisticated problems. In the case of rating from easy to hard, we might label these as “easy”, “medium”, “hard”, and finally “hardest”:

* Easy
* Medium
* Hard
* Hardest

And we can visualize their relationship, too:

Using the diagram, we assume that  and  are not the same set, or, in other words, we assume that . This is our apparently-true, but yet-unproven assertion. Of course, another interesting aspect of this diagram is that we’ve got some overlap between  and . We call  when the problem belongs to both of these sets.

Alright, so, we’ve mapped *,* ,  and  to “easy”, “medium”, “hard” and “hardest”, but how does we place a given algorithm in each category? For that, we’ll need to get a bit more formal through the next section.

Through the rest of the article, we generally prefer not to use units like “seconds” or “milliseconds”. Instead, we prefer proportional expressions like , , , and , using [Big-O notation](https://www.baeldung.com/cs/big-o-notation). Those mathematical expressions give us a clue about the [algorithmic complexity](https://www.baeldung.com/java-algorithm-complexity) of a problem.

**3. Problem Definitions**

Let’s quickly review some common Big-O values:

* – constant-time
* – logarithmic-time
* – linear-time
* – quadratic-time
* – polynomial-time
* – exponential-time
* – factorial-time

where  is a constant and  is the input size. The size of  also depends on the problem definition. For example, using a number set with a size of , the search problem has an average complexity between linear-time and logarithmic-time depending on the [data structure](https://www.bigocheatsheet.com/) in use.

**3.1. Polynomial Algorithms**

The first set of problems are polynomial algorithms that we can solve in polynomial time, like logarithmic, linear or quadratic time. If an algorithm is polynomial, we can formally define its time complexity as:

where  and  where  and  are constants and  is input size. **In general, for polynomial-time algorithms  is expected to be less than**. Many algorithms complete in polynomial time:

* All basic mathematical operations; addition, subtraction, division, multiplication
* Testing for primacy
* [**Hashtable lookup**](https://www.baeldung.com/java-hashmap-advanced), [**string operations**](https://www.baeldung.com/java-string-performance), [**sorting problems**](https://www.baeldung.com/java-sorting)
* Shortest Path Algorithms; **[Djikstra](https://www.baeldung.com/java-dijkstra)**, [**Bellman-Ford**](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm), Floyd-Warshall
* Linear and [**Binary Search Algorithms**](https://www.baeldung.com/java-binary-search) for a given set of numbers

As we talked about earlier, all of these have a complexity of  for some , and that fact places them all in . Of course, we don’t always have just one input, . But, so long as each input is a polynomial, multiplying them will still be a polynomial. For example, in [graphs](https://web.stanford.edu/class/cs97si/06-basic-graph-algorithms.pdf), we use  for edges and  for vertices, which gives us  for Bellman-Ford’s shortest path algorithm. Even if the size of the edge set is , the time complexity is still a polynomial, , so we’re still in .

We can’t always pinpoint the Big-O for an algorithm. Outside of Big-O, we can think about the problem description. Consider, for example, the game of checkers. What is the complexity of determining the optimal move on a given turn? If we constrain the size of the board to , then this is [believed](https://laatste.info/bb3/viewtopic.php?f=53&t=7817) to be a [polynomial-time problem](https://books.google.com.tr/books?id=TI-6BQAAQBAJ&pg=PA13&dq=8x8+checkers+complexity&hl=en&sa=X&ved=0ahUKEwjupqSBnpzoAhUmlIsKHVlYAZIQ6AEIMzAB#v=onepage&q=8x8%20checkers%20complexity&f=false), placing it in . But if we say it’s an  board, [it’s no longer in](https://doi.org/10.1137/0213018) . In this case, how we constrain the search space affects where we place it. Similarly, the Hamiltonian-Path problem has polynomial-time solutions for only some [types of input graphs](https://en.wikipedia.org/wiki/Hamiltonian_path_problem#Complexity).

Or another example is the stable roommate problem; it’s [polynomial-time](https://www.sciencedirect.com/science/article/abs/pii/0196677490900072) to match without a tie, but not when ties are allowed or when we include roommate preferences like married couples. (These variants are actually , which we’ll cover in a moment.) Still another factor to consider is the size of  relative to . If the input size is going to be near , then the algorithm is going to behave more like an exponential.

**3.2. NP Algorithms**

The second set of problems cannot be solved in polynomial time. However, they can be verified (or [certified](https://en.wikipedia.org/wiki/Certificate_(complexity))) in polynomial time. We expect these algorithms to have an exponential complexity, which we’ll define as:  where ,  and  where ,  and  are constants and  is the input size.  is a function of exponential-time when at least  and . As a result, we get . For example, we’ll see complexities like , ,  in this set of problems. There are several algorithms that fit this description. Among them are:

* [**Integer Factorization**](https://en.wikipedia.org/wiki/Integer_factorization) and
* [**Graph Isomorphism**](https://en.wikipedia.org/wiki/Graph_isomorphism_problem)

Both of these have two important characteristics: Their complexity is  for some  and their results can be verified in polynomial time. Those two facts place them all in , that is, the set of “Non-deterministic Polynomial” algorithms. Now, formally, we also state that these problems must be [decision problems](https://en.wikipedia.org/wiki/Decision_problem) – have a yes or no answer – though note that practically speaking, all [function problems](https://en.wikipedia.org/wiki/Decision_problem#Function_problems) can be transformed into decision problems. This distinction helps us to nail down what we mean by “verified”.

To speak precisely, then, an algorithm is in  if it can’t be solved in polynomial time and the set of solutions to any decision problem can be verified in polynomial time by a “[Deterministic Turing Machine](https://www.cs.umd.edu/~gasarch/COURSES/452/F14/p.pdf)“. What makes Integer Factorization and Graph Isomorphism interesting is that while we believe they are in , there’s no proof of whether they are in  and . Normally, all  algorithms are in , but they have another property that makes them more complex compared to  problems.

Let’s continue with that difference in the next section.

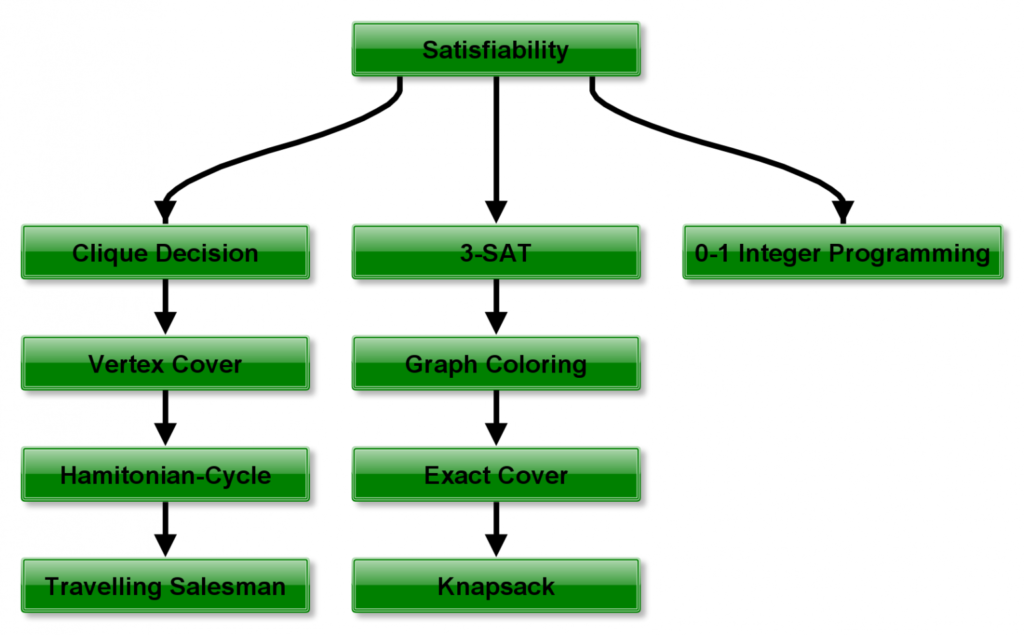
**3.3. NP-Complete Algorithms**

The next set is very similar to the previous set. Taking a look at the diagram, all of these all belong to , but are among the hardest in the set. Right now, there are more than 3000 of these problems, and the theoretical computer science community populates the list quickly. What makes them different from other  problems is a useful distinction called *completeness*. For any  problem that’s complete, there exists a polynomial-time algorithm that can transform the problem into any other -complete problem. This transformation requirement is also called *reduction*.

As stated already, there are numerous  problems proven to be complete. Among them are:

* [**Traveling Salesman**](https://en.wikipedia.org/wiki/Travelling_salesman_problem)
* [**Knapsack**](https://en.wikipedia.org/wiki/Knapsack_problem), and
* [**Graph Coloring**](https://en.wikipedia.org/wiki/Graph_coloring)

Curiously, what they have in common, aside from being in , is that each can be reduced into the other in polynomial time. These facts together place them in . The major and primary work of  belongs to [Karp](https://www.researchgate.net/publication/221580898_Reducibility_Among_Combinatorial_Problems). And his   problems are fundamental to this theoretical computer science topics. These works are founded on the [Cook-Levin](https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem) theorem and prove that the Satisfiability (SAT) problem is :



**3.4. NP-Hard Algorithms**

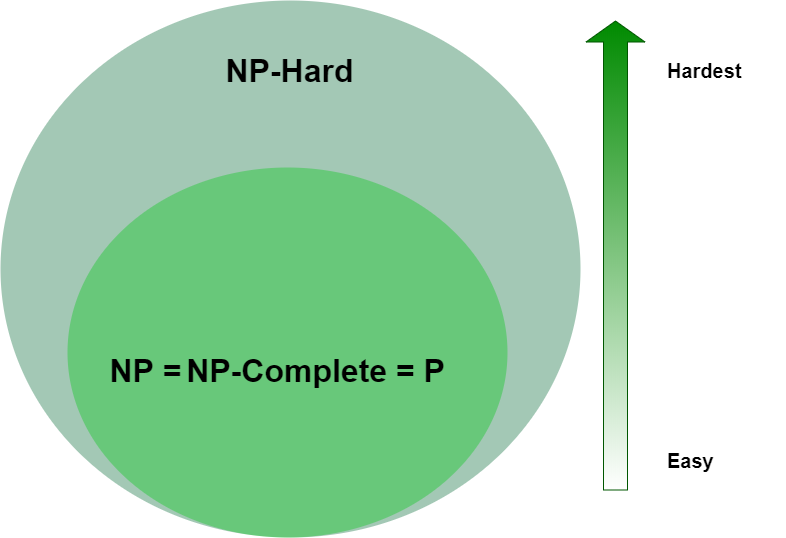
Our last set of problems contains the hardest, most complex problems in computer science. They are not only hard to solve but are hard to verify as well. In fact, some of these problems aren’t even decidable. Among the hardest computer science problems are:

* [**K-means**](https://www.baeldung.com/java-k-means-clustering-algorithm) Clustering
* [**Traveling Salesman Problem**](https://www.baeldung.com/java-simulated-annealing-for-traveling-salesman), and
* [**Graph Coloring**](http://web.math.princeton.edu/math_alive/5/Notes2.pdf)

These algorithms have a property similar to ones in  – they can all be reduced to any problem in . Because of that, these are in  and are at least as hard as any other problem in . A problem can be both in  and , which is another aspect of being .

This characteristic has led to a debate about whether or not Traveling Salesman is indeed . Since  and  problems can be verified in polynomial time, proving that an algorithm cannot be verified in polynomial time is also sufficient for placing the algorithm in .

**4. So, Does *P=NP*?**

A question that’s fascinated many computer scientists is whether or not all algorithms in  belong to :It’s an interesting problem because it would mean, for one, that any  or  problem can be solved in polynomial time. So far, proving that  as proven elusive. Because of the intrigue of this problem, it’s one of the [Millennium Prize Problems](https://en.wikipedia.org/wiki/Millennium_Prize_Problems) for which there is a $1,000,000 prize.

For our definitions, we assumed that , however,  may be possible. If it were so, aside from  or  problems being solvable in polynomial time, certain algorithms in  would also dramatically simplify. For example, if their verifier is  or , then it follows that they must also be solvable in polynomial time, moving them into  as well.

We can conclude that  means a radical change in computer science and even in the real-world scenarios. Currently, some security algorithms have the basis of being a requirement of too long calculation time. Many encryption schemes and algorithms in cryptography are based on the [number factorization](https://en.wikipedia.org/wiki/Integer_factorization) which the best-known algorithm with exponential complexity. If we find a polynomial-time algorithm, these algorithms become vulnerable to attacks.

**5. Conclusion**

Within this article, we have an introduction to a famous problem in computer science. Through the article, we focused on the different problem sets;, , , and . We also provided a good starting point for future studies and what-if scenarios when . Briefly after reading, we can conclude a generalized classification as follows:

* problems are quick to solve
* problems are quick to verify but slow to solve
* problems are also quick to verify, slow to solve and can be reduced to any other  problem
* problems are slow to verify, slow to solve and can be reduced to any other  problem

As a final note, if  has proof in the future, humankind has to construct a new way of security aspects of the computer era. When this happens, there has to be another complexity level to identify new hardness levels than we have currently.