# Chapter 1

# NP Completeness I

By Sariel Har-Peled, December 17, 2012[1] Version: 1.05

"Then you must begin a reading program immediately so that you man understand the crises of our age," Ignatius said solemnly. "Begin with the late Romans, including Boethius, of course. Then you should dip rather extensively into early Medieval. You may skip the Renaissance and the Enlightenment. That is mostly dangerous propaganda. Now, that I think about of it, you had better skip the Romantics and the Victorians, too. For the contemporary period, you should study some selected comic books."

"You're fantastic."

"I recommend Batman especially, for he tends to transcend the abysmal society in which he's found himself. His morality is rather rigid, also. I rather respect Batman."

– A confederacy of Dunces, John Kennedy Toole.

## 1.1 Introduction

The question governing this course, would be the development of efficient algorithms. Hopefully, what is an algorithm is a well understood concept. But what is an *efficient* algorithm? A natural answer (but not the only one!) is an algorithm that runs quickly.

What do we mean by quickly? Well, we would like our algorithm to:

(A) Scale with input size. That is, it should be able to handle large and hopefully huge inputs.

(B) Low level implementation details should not matter, since they correspond to small improvements in performance. Since faster CPUs keep appearing it follows that such improvements would (usually) be taken care of by hardware.

(C) What we will really care about are asymptotic running time. Explicitly, polynomial time.

[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

| Input size | $n^2$ ops | $n^3$ ops | $n^4$ ops | $2^n$ ops | $n!$ ops |
|---|---|---|---|---|---|
| 5 | 0 secs | 0 secs | 0 secs | 0 secs | 0 secs |
| 20 | 0 secs | 0 secs | 0 secs | 0 secs | 16 mins |
| 30 | 0 secs | 0 secs | 0 secs | 0 secs | $3 \cdot 10^9$ years |
| 50 | 0 secs | 0 secs | 0 secs | 0 secs | never |
| 60 | 0 secs | 0 secs | 0 secs | 7 mins | never |
| 70 | 0 secs | 0 secs | 0 secs | 5 days | never |
| 80 | 0 secs | 0 secs | 0 secs | 15.3 years | never |
| 90 | 0 secs | 0 secs | 0 secs | 15,701 years | never |
| 100 | 0 secs | 0 secs | 0 secs | $10^7$ years | never |
| 8000 | 0 secs | 0 secs | 1 secs | never | never |
| 16000 | 0 secs | 0 secs | 26 secs | never | never |
| 32000 | 0 secs | 0 secs | 6 mins | never | never |
| 64000 | 0 secs | 0 secs | 111 mins | never | never |
| 200,000 | 0 secs | 3 secs | 7 days | never | never |
| 2,000,000 | 0 secs | 53 mins | 202.943 years | never | never |
| $10^8$ | 4 secs | 12.6839 years | $10^9$ years | never | never |
| $10^9$ | 6 mins | 12683.9 years | $10^{13}$ years | never | never |

Figure 1.1: Running time as function of input size. Algorithms with exponential running times can handle only relatively small inputs. We assume here that the computer can do $2.5 \cdot 10^{15}$ operations per second, and the functions are the exact number of operations performed. Remember – never is a long time to wait for a computation to be completed.

In our discussion, we will consider the input size to be $n$, and we would like to bound the overall running time by a function of $n$ which is asymptotically as small as possible. An algorithm with better asymptotic running time would be considered to be *better*.

**Example 1.1.1.** It is illuminating to consider a concrete example. So assume we have an algorithm for a problem that needs to perform $c2^n$ operations to handle an input of size $n$, where $c$ is a small constant (say 10). Let assume that we have a CPU that can do $10^9$ operations a second. (A somewhat conservative assumption, as currently [Jan 2006][②], the blue-gene supercomputer can do about $3 \cdot 10^{14}$ floating-point operations a second. Since this super computer has about $131,072$ CPUs, it is not something you would have on your desktop any time soon.) Since $2^{10} \approx 10^3$, you have that our (cheap) computer can solve in (roughly) 10 seconds a problem of size $n = 27$.

But what if we increase the problem size to $n = 54$? This would take our computer about 3 million years to solve. (It is better to just wait for faster computers to show up, and then try to solve the problem. Although there are good reasons to believe that the exponential growth in computer performance we saw in the last 40 years is about to end. Thus, unless a

[②]But the recently announced Super Computer that would be completed in 2012 in Urbana, is naturally way faster. It supposedly would do $10^{15}$ operations a second (i.e., petaflop). Blue-gene probably can not sustain its theoretical speed stated above, which is only slightly slower.

substantial breakthrough in computing happens, it might be that solving problems of size, say, $n = 100$ for this problem would forever be outside our reach.)

The situation dramatically change if we consider an algorithm with running time $10n^2$. Then, in one second our computer can handle input of size $n = 10^4$. Problem of size $n = 10^8$ can be solved in $10n^2/10^9 = 10^{17-9} = 10^8$ which is about 3 years of computing (but blue-gene might be able to solve it in less than 20 minutes!).

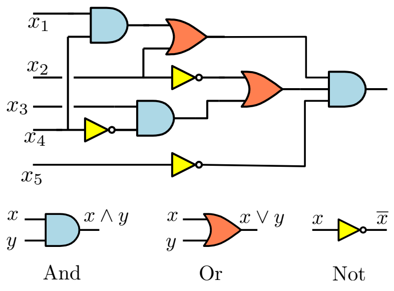
Thus, algorithms that have asymptotically a polynomial running time (i.e., the algorithms running time is bounded by $O(n^c)$ where $c$ is a constant) are able to solve large instances of the input and can solve the problem even if the problem size increases dramatically.

**Can we solve all problems in polynomial time?** The answer to this question is unfortunately no. There are several synthetic examples of this, but it is believed that a large class of important problems can not be solved in polynomial time.

## Circuit Satisfiability

**Instance**: A circuit $C$ with $m$ inputs
**Question:** Is there an input for $C$ such that $C$ returns true for it.

As a concrete example, consider the circuit depicted on the right.



Currently, all solutions known to Circuit Satisfiability require checking all possibilities, requiring (roughly) $2^m$ time. Which is exponential time and too slow to be useful in solving large instances of the problem.

This leads us to the most important open question in theoretical computer science:

**Question 1.1.2.** *Can one solve Circuit Satisfiability in polynomial time?*

The common belief is that Circuit Satisfiability can **NOT** be solved in polynomial time. Circuit Satisfiability has two interesting properties.

(A) Given a supposed positive solution, with a detailed assignment (i.e., proof): $x_1 \leftarrow 0$, $x_2 \leftarrow 1, ..., x_m \leftarrow 1$ one can verify in polynomial time if this assignment really satisfies $C$. This is done by computing what every gate in the circuit what its output is for this input. Thus, computing the output of $C$ for its input. This requires evaluating the gates of $C$ in the right order, and there are some technicalities involved, which we are ignoring. (But you should verify that you know how to write a program that does that efficiently.)
Intuitively, this is the difference in hardness between coming up with a proof (hard), and checking that a proof is correct (easy).

(B) It is a ***decision problem***. For a specific input an algorithm that solves this problem has to output either TRUE or FALSE.

## 1.2 Complexity classes

**Definition 1.2.1 (P: Polynomial time).** Let P denote is the class of all decision problems that can be solved in polynomial time in the size of the input.

**Definition 1.2.2 (NP: Nondeterministic Polynomial time).** Let NP be the class of all decision problems that can be verified in polynomial time. Namely, for an input of size $n$, if the solution to the given instance is true, one (i.e., an oracle) can provide you with a proof (of polynomial length!) that the answer is indeed TRUE for this instance. Furthermore, you can verify this proof in polynomial time in the length of the proof.

Clearly, if a decision problem can be solved in polynomial time, then it can be verified in polynomial time. Thus, $\mathrm{P} \subseteq \mathrm{NP}$.

**Remark.** The notation NP stands for Non-deterministic Polynomial. The name come from a formal definition of this class using Turing machines where the machines first guesses (i.e., the non-deterministic stage) the proof that the instance is TRUE, and then the algorithm verifies the proof.

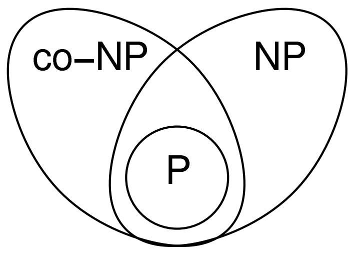


Figure 1.2: The relation between the different complexity classes P, NP, and co-NP.

**Definition 1.2.3 (co-NP).** The class co-NP is the opposite of NP – if the answer is FALSE, then there exists a short proof for this negative answer, and this proof can be verified in polynomial time.

See Figure 1.2 for the currently *believed* relationship between these classes (of course, as mentioned above, $\mathrm{P} \subseteq \mathrm{NP}$ and $\mathrm{P} \subseteq \text{co-NP}$ is easy to verify). Note, that it is quite possible that $\mathrm{P} = \mathrm{NP} = \text{co-NP}$, although this would be extremely surprising.

**Definition 1.2.4.** A problem $\Pi$ is NP-Hard, if being able to solve $\Pi$ in polynomial time implies that $\mathrm{P} = \mathrm{NP}$.

**Question 1.2.5.** *Are there any problems which are* NP-Hard*?*

Intuitively, being NP-Hard implies that a problem is ridiculously hard. Conceptually, it would imply that proving and verifying are equally hard - which nobody that did CS 573 believes is true.

In particular, a problem which is NP-Hard is at least as hard as ALL the problems in NP, as such it is safe to assume, based on overwhelming evidence that it can not be solved in polynomial time.

**Theorem 1.2.6 (Cook's Theorem).** *Circuit Satisfiability is* NP-Hard.

**Definition 1.2.7.** A problem $\Pi$ is NP-Complete (NPC in short) if it is both NP-Hard and in NP.

Clearly, Circuit Satisfiability is NP-Complete, since we can verify a positive solution in polynomial time in the size of the circuit,

By now, thousands of problems have been shown to be NP-Complete. It is extremely unlikely that any of them can be solved in polynomial time.

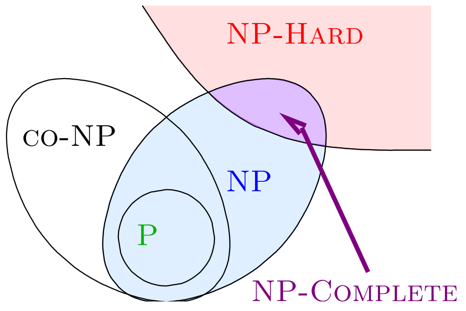


Figure 1.3: The relation between the complexity classes.

**Definition 1.2.8.** In the formula satisfiability problem, (a.k.a. SAT) we are given a formula, for example:

$$\left(a \vee b \vee c \vee \overline{d}\right) \iff \left((b \wedge \overline{c}) \vee \overline{(\overline{a} \Rightarrow d)} \vee (c \neq a \wedge b)\right)$$

and the question is whether we can find an assignment to the variables $a, b, c, \ldots$ such that the formula evaluates to TRUE.

It seems that SAT and Circuit Satisfiability are "similar" and as such both should be NP-Hard.

## 1.2.1 Reductions

Let $A$ and $B$ be two decision problems.

Given an input $I$ for problem $A$, a *reduction* is a transformation of the input $I$ into a new input $I'$, such that

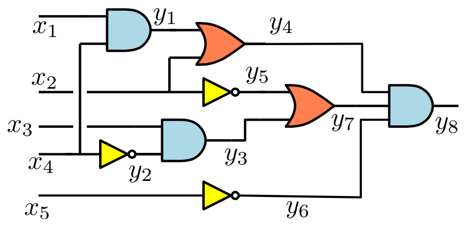
$$A(I) \;\; is \;\; \text{TRUE} \qquad \Leftrightarrow \qquad B(I') \;\; is \;\; \text{TRUE}.$$

Thus, one can solve $A$ by first transforming and input $I$ into an input $I'$ of $B$, and solving $B(I')$.

This idea of using reductions is omnipresent, and used almost in any program you write.

Let $T : I \to I'$ be the input transformation that maps $A$ into $B$. How fast is $T$? Well, for our nefarious purposes we need ***polynomial reductions***; that is, reductions that take polynomial time.

For example, given an instance of Circuit Satisfiability, we would like to generate an equivalent formula. We will explicitly write down what the circuit computes in a formula form. To see how to do this, consider the following example.



| | | |
|---|---|---|
| $y_1 = x_1 \wedge x_4$ | $y_2 = \overline{x_4}$ | $y_3 = y_2 \wedge x_3$ |
| $y_4 = x_2 \vee y_1$ | $y_5 = \overline{x_2}$ | $y_6 = \overline{x_5}$ |
| $y_7 = y_3 \vee y_5$ | $y_8 = y_4 \wedge y_7 \wedge y_6$ | $y_8$ |

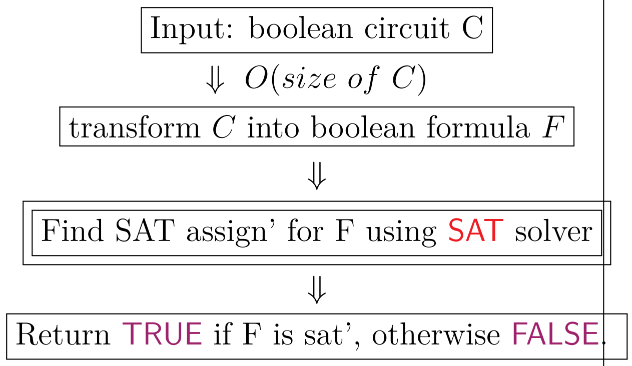
We introduced a variable for each wire in the circuit, and we wrote down explicitly what each gate computes. Namely, we wrote a formula for each gate, which holds only if the gate computes correctly the output for its given input.

The circuit is satisfiable **if and only if** there is an assignment such that all the above formulas hold. Alternatively, the circuit is satisfiable if and only if the following (single) formula is satisfiable

$$\begin{aligned}(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = y_2 \wedge x_3)\\ \wedge (y_4 = x_2 \vee y_1) \wedge (y_5 = \overline{x_2})\\ \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5)\\ \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8.\end{aligned}$$

It is easy to verify that this transformation can be done in polynomial time.

The resulting reduction is depicted in Figure 1.4.



Input: boolean circuit C
⇓ $O(size\ of\ C)$
transform $C$ into boolean formula $F$
⇓
Find SAT assign' for F using SAT solver
⇓
Return TRUE if F is sat', otherwise FALSE.

Figure 1.4: Algorithm for solving CSAT using an algorithm that solves the SAT problem

Namely, given a solver for SAT that runs in $T_{\mathsf{SAT}}(n)$, we can solve the CSAT problem in time

$$T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n)),$$

where $n$ is the size of the input circuit. Namely, if we have polynomial time algorithm that solves SAT then we can solve CSAT in polynomial time.

Another way of looking at it, is that we believe that solving CSAT requires exponential time; namely, $T_{\mathsf{CSAT}}(n) \geq 2^n$. Which implies by the above reduction that

$$2^n \leq T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n)).$$

Namely, $T_{\mathsf{SAT}}(n) \geq 2^{n/c} - O(n)$, where $c$ is some positive constant. Namely, if we believe that we need exponential time to solve CSAT then we need exponential time to solve SAT.

This implies that if SAT $\in$ P then CSAT $\in$ P.

We just proved that SAT is as hard as CSAT. Clearly, SAT $\in$ NP which implies the following theorem.

**Theorem 1.2.9.** *SAT (formula satisfiability) is* NP-Complete.

## 1.3 More NP-Complete problems

### 1.3.1 3SAT

A boolean formula is in conjunctive normal form (`CNF`) if it is a conjunction (AND) of several *clauses*, where a clause is the disjunction (or) of several *literals*, and a literal is either a variable or a negation of a variable. For example, the following is a `CNF` formula:

$$\overbrace{(a \vee b \vee \overline{c})}^{\textit{clause}} \wedge (a \vee \overline{e}) \wedge (c \vee e).$$

**Definition 1.3.1.** `3CNF` formula is a `CNF` formula with *exactly* three literals in each clause.

The problem 3SAT is formula satisfiability when the formula is restricted to be a `3CNF` formula.

**Theorem 1.3.2.** *3SAT is* NP-Complete.

*Proof*: First, it is easy to verify that 3SAT is in NP.

Next, we will show that 3SAT is NP-Complete by a reduction from CSAT (i.e., Circuit Satisfiability). As such, our input is a circuit $C$ of size $n$. We will transform it into a `3CNF` in several steps:

(A) Make sure every AND/OR gate has only two inputs. If (say) an AND gate have more inputs, we replace it by cascaded tree of AND gates, each one of degree two.

(B) Write down the circuit as a formula by traversing the circuit, as was done for SAT. Let $F$ be the resulting formula.
A clause corresponding to a gate in $F$ will be of the following forms: (i) $a = b \wedge c$ if it corresponds to an AND gate, (ii) $a = b \vee c$ if it corresponds to an OR gate, and (iii) $a = \overline{b}$ if it corresponds to a NOT gate. Notice, that except for the single clause corresponding to the output of the circuit, all clauses are of this form. The clause that corresponds to the output is a single variable.

(C) Change every gate clause into several `CNF` clauses.

   (i) For example, an AND gate clause of the form $a = b \wedge c$ will be translated into

$$\left(a \vee \overline{b} \vee \overline{c}\right) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)\,. \tag{1.1}$$

   Note that Eq. (1.1) is true if and only if $a = b \wedge c$ is true. Namely, we can replace the clause $a = b \wedge c$ in $F$ by Eq. (1.1).

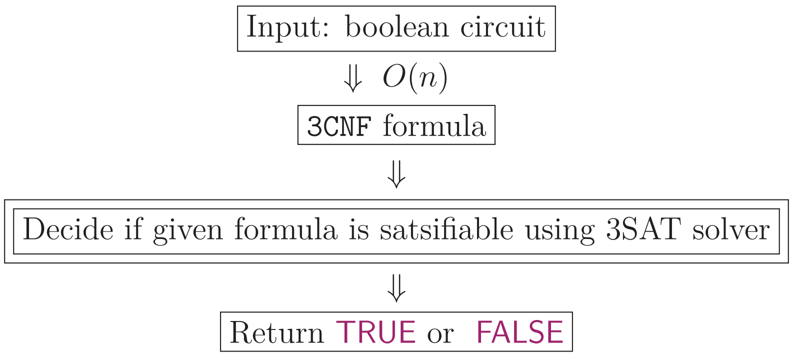


Figure 1.5: Reduction from CSAT to 3SAT

(ii) Similarly, an OR gate clause the form $a = b \vee c$ in $F$ will be transformed into

$$(\overline{a} \vee b \vee c) \wedge (a \vee \overline{b}) \wedge (a \vee \overline{c}).$$

(iii) Finally, a clause $a = \overline{b}$, corresponding to a NOT gate, will be transformed into

$$(a \vee b) \wedge (\overline{a} \vee \overline{b}).$$

(D) Make sure every clause is exactly three literals. Thus, a single variable clause $a$ would be replaced by

$$(a \vee x \vee y) \wedge (a \vee \overline{x} \vee y) \wedge (a \vee x \vee \overline{y}) \wedge (a \vee \overline{x} \vee \overline{y}),$$

by introducing two new dummy variables $x$ and $y$. And a two variable clause $a \vee b$ would be replaced by

$$(a \vee b \vee y) \wedge (a \vee b \vee \overline{y}),$$

by introducing the dummy variable $y$.

This completes the reduction, and results in a new 3CNF formula $G$ which is satisfiable if and only if the original circuit $C$ is satisfiable. The reduction is depicted in Figure 1.5. Namely, we generated an equivalent 3CNF to the original circuit. We conclude that if $T_{\mathsf{3SAT}}(n)$ is the time required to solve 3SAT then

$$T_{CSAT}(n) \leq O(n) + T_{3SAT}(O(n)),$$

which implies that if we have a polynomial time algorithm for 3SAT, we would solve CSAT is polynomial time. Namely, 3SAT is NP-Complete. ■

## 1.4 Bibliographical Notes

Cook's theorem was proved by Stephen Cook (http://en.wikipedia.org/wiki/Stephen_Cook). It was proved independently by Leonid Levin (http://en.wikipedia.org/wiki/Leonid_Levin) more or less in the same time. Thus, this theorem should be referred to as the Cook-Levin theorem.

The standard text on this topic is [GJ90]. Another useful book is [ACG$^+$99], which is a more recent and more updated, and contain more advanced stuff.

## Bibliography

[ACG$^+$99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation.* Springer-Verlag, Berlin, 1999.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.