14. Computational Complexity

The classes of problems which are respectively known and not known to have good algorithms are of great theoretical interest.

Jack Edmonds, 1966

Please read Chapter 34 NP-Completeness in in the textbook (Corman et al., 2009). I also recommend (Papadimitriou, 1994) and (Garey and Johnson, 1979) as references for computational complexity.

Decision Problems

A *decision problem* is a question (in some formal system) that has a `True` or `False` answer. A decision problem is *decidable* if there is an algorithm that correctly answers all of its instances. Here are some classic decision problems:

Imagine other decision problems. Convince yourself of the time complexity given for Sorted and Reachability

1. Sorted: Is the list $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ of integers sorted? Sorted can be solved in $O(n)$ time.
2. Reachability: Given a two vertices u and v in a graph G, is there a path from u to v? Reachability can be solved in $O(n^2)$ time, where n is the number of nodes in G.
3. 0—1 Knapsack: Given a knapsack that can hold weight C and a list of provisions $\langle p_k : k \in \mathbb{N} \rangle$ each of which has a weight w_k and value v_k. Is it possible to fill the knapsack with provisions weighing no more than C and having a total value of V or greater?

$$\sum w_k \leq C$$
$$\sum v_k \geq V$$

Presburger arithmetic is an example of class of decidable problems. Presburger arithmetic is the collection of statements P about the natural numbers $\mathbb{N}$ that only involve addition, equality, and Boolean operations among sub-expressions. The Presburger axioms are:

Propositional logic studies the truth of Boolean expressions (`True` or `False` values combined using AND, OR, and NOT, and operations that can be defined from these three basic operations.)

First-order logic introduces quantification of formula that involve variables which determine the truth of a expression.

1. $\neg(0 = x + 1)$
2. $x + 1 = y + 1 \Rightarrow x = y$

3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. Let $\mathtt{P}(n)$ be a first-order formula in the language of Presburger arithmetic about a natural number n. The induction axiom is:

$$(\mathtt{P}(0) \wedge (\forall n)(\mathtt{P}(n) \Rightarrow \mathtt{P}(n+1))) \Rightarrow (\forall n)(\mathtt{P}(n))$$

If P is a statement about Presburger arithmetic, then P is decidable, that is there is an algorithm that decides if P is `True` or `False`. Moreover,

- Presburger arithmetic is *consistent*: If P is derivable from (Presburger) axioms, then $\neg$P cannot be deduced from these axioms.
- Presburger arithmetic is *complete*: For each expression P, only one of P or $\neg$P is `True`, and the one that is `True` can be derived from the axioms.
- Presburger arithmetic is *decidable*: There is an algorithm that decides whether proposition P is `True` or `False`.

See (Stansifer, 1984) for additional details on on the history and significance of Presburger's discoveries.

Likewise, Gödel, and others, proved completeness for logical expressions in the first-order logic (Gödel, 1930).

Theorem 14: Gödel's Completeness Theorem

Every valid logical expression is provable. Equivalently, every logical expression is either satisfiable or refutable.

On the other hand, Gödel (Gödel, 1992) demonstrated how to construct propositions, from the Peano axioms for general arithmetic, that can not be proven `True` or `False`. Gödel realized that natural numbers could be used to name basic symbols, expressions over these symbols, and proofs. Let $G(s)$ be the Gödel number of symbol s. For instance, if

$$G(0) = 1 \qquad G(+) = 3 \qquad G(=) = 5 \qquad G(x) = 7$$

Then the axiom $x + 0 = x$ has Gödel number

$$G(x + 0 = x) = 2^7 3^3 5^1 7^5 11^7 = 1,131,912,171,637,632$$

If expression $\mathtt{Q}$ can be derived from $\mathtt{P}$ by some rule of inference, then there is a function f such that

$$f(G(\mathtt{P})) = G((q))$$

Let $\mathtt{P}(n)$ be a predicate and let $G = G(\mathtt{P}(n))$ be its Gödel number. Consider $\mathtt{P}(G)$. This expression has a Gödel number, call it G'. And, the development goes on from here, beyond the scope of these notes.

The upshot of Gödel's coding of mathematics is that he was able to code a self-referential statement such as: $\mathtt{P}$ = "there is no proof for this statement."

If $\mathtt{P}$ is `True`, then general arithmetic is incomplete: The statement $\mathtt{P}$ is `True`, but it has no proof.

If $\mathtt{P}$ is `False`, then general arithmetic is inconsistent: There is a proof of $\mathtt{P}$, yet $\mathtt{P}$ is `False`.

Theorem 15: Gödel's First Incompleteness Theorem

Every *consistent* formal proof system $\mathcal{F}$ about a sufficiently rich arithmetic is incomplete.

Theorem 15 says there are statements about the arithmetic we learned as children that are `True` but have no proof. Gödel's second theorem says you cannot prove a consistent arithmetic is consistent.

Theorem 16: Gödel's Second Incompleteness Theorem

If $\mathcal{F}$ is a consistent formal proof system about a sufficiently rich arithmetic, then there is no proof that $\mathcal{F}$ is consistent.

Turing Machines

An algorithm can be thought of as a Turing machine for some decision problem. Informally, a Turing machine uses a transition function δ to map the *current state* of the machine and the *character read* to a *next state*, a *character printed*, and a *direction* to move the read write head.

The next state k' either in $\mathbb{K}$, the set of states, or one of three special states: answers y "yes" and n, "no," or the "halt" state h. The read/write head can move $\leftarrow$ "left", $\rightarrow$ "right," or — "stay."

There are many ways to define a Turing machine. Here is Papadimitriou's (Papadimitriou, 1994) definition.

Definition 17: Turing Machine

A Turing machine is a 4-tuple $M = (\mathbb{K}, \Sigma, \delta, s)$ where:

1. *$\mathbb{K}$ is a finite set of states*
2. *$s \in \mathbb{K}$ is the initial (start) state*
3. *Σ is an alphabet (a finite set of symbols (characters). Σ contains two special symbols: $\sqcup$ and $\triangleright$, called* blank *and* first*, respectively.*
4. *δ is a transition function. It maps a (state, character) pair to a triple (next state, character, direction).*

$$\delta : (\mathbb{K}, \Sigma) \rightarrow (\mathbb{K} \cup \{h, y, n\}, \Sigma, \{\leftarrow, \rightarrow, —\})$$

Example: Turing machine to add 1

The transition function for a Turing machine can be defined by a state transition table. Consider adding one to a natural

number written in binary, for instance $n = (101010)_2 = 42$. *Assume after the first symbol* $\triangleright$, *each bit is written on a cell of a tape and the read/write head is positioned on the leading, leftmost, most significant bit,* 1 *in this case. A blank cell,* $\sqcup$, *lies after the rightmost, least significant bit.*
To add one to the n, *the Turing machine*

1. *Copies the bits from left-to-right until the blank cell is scanned.*
2. *When a blank is scanned, it backs up (to the left) and turns* 1*'s into* 0*'s until the first* 0 *is found.*
3. *When the first* 0 *is found, the machine changes the* 0 *into a* 1 *and halts.*

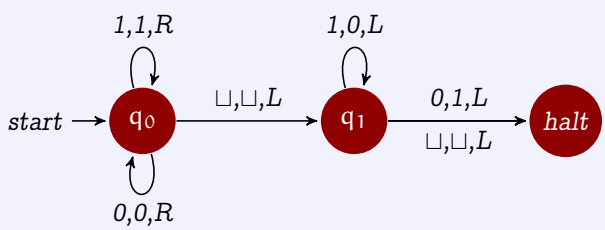
This can be described by the state transition table below. It reads: When in state q_0

- *If* 0 *or* 1 *is scanned, stay in state* q_0, *leave the bit unchanged, and move the read/write head right.*
- *If* $\sqcup$ *is scanned, move to state* q_1, *leave the blank unchanged, and move the head left.*

Similar transitions can be read for state q_1.

	0	1	$\sqcup$
q_0	$(q_0, 0, \rightarrow)$	$(q_0, 1, \rightarrow)$	$(q_1, \sqcup, \leftarrow)$
q_1	$(halt, 1, \leftarrow)$	$(q_1, 0, \leftarrow)$	$(halt, \sqcup, \leftarrow)$

The machine can also be described by a state transition diagram.



Consider how this machine operated on $(101010B)_2 = 42$. *It copies the bits from left-to-right until the blank* B *is scanned. It then moves back left and seeing the* 0, *changes it to* 1 *and halts.*
In a similar manner the string $(101011B)_2 = 43$, *is changed into* $(101100B)_2 = 44$.

The Universal Turing Machine

Turing showed (Turing, 1936) and important aspect of his machine: *It is possible to invent a single machine that can simulate any other Turing machine.* That is, there is a *universal* Turing machine, called U. The input to the universal machine U is a pair (M, x). The universal machine U computes $M(x)$, that is $U(M, x) = M(x)$.

197a ⟨*Universal Machine* 197a⟩≡

```
U(machine M, input x) {
  M(x);
}
```

The existence of a universal machine leads to *undecidable* problems, the most famous of which is the Haltling Problem.

Problem 15: The Halting Problem

Decision Problem: Given a Turing machine M and its input x, does M halt on x?

There is no algorithm that *decides* the halting problem. It may be possible to decide is a particular machine M halts on a particular input x, but there is no algorithm that answers the halting problem for every instance of M and x.

Define the halting language $\mathbb{H}$ is the set of all (machine, input) pairs such that M halts on x.

The symbol $\nearrow$ stands for "does not halt."

$$\mathbb{H} = \{(M, x) : M(x) \neq \nearrow\}$$

There is no Turing machine that decides whether or not $(M, x) \in \mathbb{H}$ for all pairs (M, x). The proof is by contradiction.

Consider the thought experiment of executing the pseudo-code below: The program accepts the encoding of a machine `M` as input. It runs `M` on `M`, looping forever if `M(M)` halts and halts if `M(M)` does not halt.

The *diagonalization* idea comes from Cantor's proof that the real numbers are uncountable.

Listing 69: The Diagonal Machine

197b ⟨*Diagonal Machine* 197b⟩≡

```
diagMac(machine M) {
  if (M(M) halts) then { Loop forever; }
    else halt;
}
```

The diagonalization name comes from running the program on itself.

197c ⟨*Diagonalization* 197c⟩≡

```
main diagMac(diagMac);
```

Now consider the logic:

- If `diagMac(diagMac)` halts, then `diagMac(diagMac)` loops forever, that is, `diagMac(diagMac)` does not halt.
- On the other hand, if `diagMac(diagMac)` does not halt, then `diagMac(diagMac)` halts.

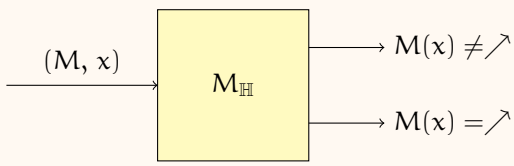
Therefore, there can be no test (algorithm) that correctly answers:

For all Turing machine `M` and for all inputs `x`, does `M` halt on `x`?

The traditional proof that the halting problem is undecidable goes something like this:

Proof: The Halting Problem is Undecidable

Pretend there is a Turing machine $M_{\mathbb{H}}$ *that decides the halting problem.*



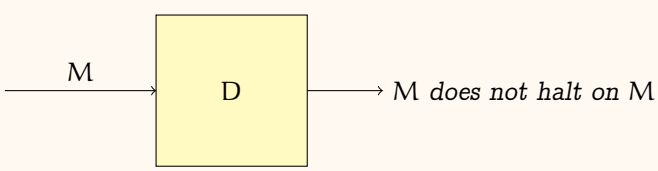
Use $M_{\mathbb{H}}$ *to construct a Turing machine* D *that accepts the encoding of a Turing machine* M *and runs* $M_{\mathbb{H}}$ *on* (M, M)*. The behavior of* D *is this:*

1. D *does not halt if* M *halts on* M*.*

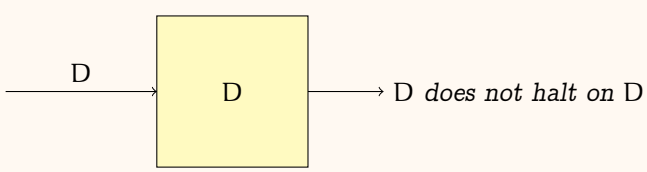
$$\text{If } (M(M) \neq \nearrow), \text{ then } D(M) = \nearrow.$$

2. D *halts if* M *does not halt on* M *(*$M(M) = \nearrow$*).*

$$\text{If } (M(M) = \nearrow), \text{ then } D(M) \neq \nearrow$$



Consider D(D)



1. D *does not halt (on input* D*) if* D *halts on* D

$$\text{If } (D(D) \neq \nearrow), \text{ then } D(D) = \nearrow.$$

2. D *halts (on input* D*), if* D *does not halt on* D

$$\text{If } (D(D) = \nearrow), \text{ then } D(D) \neq \nearrow$$

This contradiction implies that the halting machine $M_{\mathbb{H}}$ *cannot exist.*

Determinism versus Non-Determinism

By default, Turing machines are deterministic: Their transition functions δ are functions. When transitions are relaxed to be *relations*, the machine is said to be *non-deterministic*.

Definition 18: The P and NP Complexity Classes

The complexity class P *is the class of all decision problems where all problem instances can be solved in in polynomial time on a (deterministic) Turing machine.* $O(n^k)$*, where* n *is the size of the instance and* k *is a fixed natural number.*

The complexity class NP *is the class of all decision problems that solve all instances in polynomial time on a non-deterministic Turing machine.*

Intuitively, the class P is the set of all problems that can be *solved* in polynomial time. Such problems are said to be *tractable*, even though they may run for a very long time.

The class NP is the set of all problems that, when given an answer (a *certificate*), the answer can be *checked* to be correct in polynomial time. Cook in his seminal paper (Cook, 1971) clearly described these ideas and their implications.

A cubic algorithm on a problem of size $n = 10^6$ will take about 10^{18} steps, which at 10^{-9} seconds per step will take about 10^9 seconds. That's about 33 years given that there are π billion seconds per century.

Exaflop machines, (10^{18} floating point instructions per second) are being developed.

Problem 16: Satisfiability

Decision Problem: Given a Boolean expression B *of* n *literals in conjunctive normal form, does* B *have a* truth *assignment?*

Let P be a Boolean variable. Then P and ¬P are *literals*.

A *clause* is a disjunction of literals. For example $P \vee \neg Q$.

Conjunctive normal form is a conjunction of clauses, that is, an AND of ORs.

A *truth assignment* for a Boolean expression is an assignment of True or False to each variable such that the whole expression is True.

Example: SAT Problems

The expression

$$\phi = (P \vee Q) \wedge \neg P$$

is satisfied by $P = Q =$ *False.*
On the other hand, the expression

$$\phi = (P \vee Q \vee R) \wedge (P \vee \neg Q) \wedge (Q \vee \neg R) \wedge (R \vee \neg P) \wedge (\neg P \vee \neg Q \vee \neg R)$$

is unsatisfiable. Although you can reason about this expression to see it is unsatisfiable. Notice the expression is in conjunctive normal form so a satisfying truth assignment must satisfy all clauses.
The first clause requires at least one of the three variables be True. The next three clauses requires all three values be the same. (If P *is True, then* R *must be True, and then* Q *must be True. On the other hand, if* P *is False, then* Q *must be False, and then* R *must be False.)*
But, in general, you may need to check all 2^n *truth assignments to confirm an* n *variable Boolean expression is never satisfied.*

Cook describes the *satisfiability* (SAT) problem, which is clearly in NP but is not known to be in P. The non-deterministic algorithm guesses a satisfying truth assignment for ϕ and checks that it satisfies each clause in ϕ. On the other hand, no polynomial-time deterministic algorithm has ever been discovered for satisfiability. This leads to what is said to be the fundamental problem in theoretical computer science.

Problem 17: P versus NP

Decision Problem: Does $P = NP$*?*

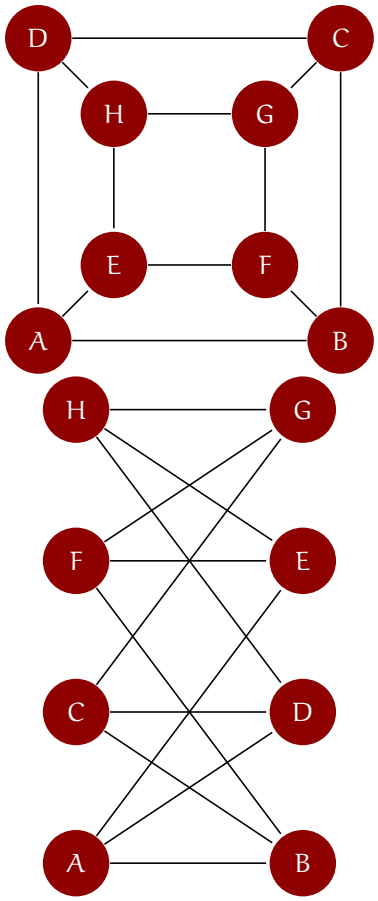
It is clear that P is a subset of NP. Whether the two classes of problems are the same remains unknown. I think the consensus is that $P \neq NP$. "Proofs" that $P \neq NP$ are proffered every so often, but at this time none has stood and no one knows for certain what the answer is.

Here are some sample NP problems. Reason that they belong to NP by convincing yourself that a answer could be checked in polynomial time.

Problem 18: Subgraph Isomorphism

Given two graphs $G_0 = (V_0, E_0)$ *and* $G_1 = (V, E_1)$*. Does* G_0 *contain a subgraph* (V, E) *such that* $|V| = |V_1|$, $|E| = |E_1|$*, and is there a one-to-one function* $f : V \mapsto V_1$ *such that* $\{u, v\} \in E$ *if and only if* $\{f(u), f(v)\} \in E_1$.

Clearly, if given nodes V and edges E, their cardinalities can be checked in polynomial time. Likewise, that f preserves edges and be checked in polynomial time.



Problem 19: Traveling Salesman

Given a finite set of cities $\mathbb{C} = \{c_0, \ldots, c_{n-1}\}$, *distances* $d(c_i, c_j) \in \mathbb{Z}^+$ *for each pair* $(c_i, c_j) \in \mathbb{C}$, *and a bound* $B \in \mathbb{Z}^+$. *Is there a tour of all cities with total length no more than* B. *That is, a permutation* $\langle c_{\pi(0)}, c_{\pi(1)}, \ldots, c_{\pi(n-1)} \rangle$ *of cities such that*

$$\left[\sum_{k=0}^{n-1} d(c_{\pi(k)}, c_{\pi(k+1)})\right] + d(c_{\pi(n)}, c_{\pi(0)}) \leq B$$

Clearly, given the tour, its cost can be computed in polynomial time.

Reductions

A classic problem solving technique is to reduce a new problem to an already solved problem. A *reduction* is an algorithm that solves problem A by transforming any instance of A to an equivalent instance of previously solved problem B. Such a reduction should be executable in

polynomial time. The notation

$$A \leq_P B$$

means if B can be solved in polynomial time, then A can be solved in polynomial time. This establishes potential ways to design algorithms.

On the other hand, if A cannot be solved in polynomial time, then neither can B, establishing *intractability*.

Consider reducing *matching* problem to *max-flow*.

Example: Matching reduced to Reachability

Given an bipartite graph (U, V, E)*, where* $|U| = |V| = n$*. Construct a network of nodes* $(U \cup V \cup \{s, t\}$ *where* s *is the source and* t *is the target (sink), and with edges*

$$\{(s, u) : u \in U\} \cup E \cup \{(v, t) : v \in V\}$$

where all capacities equal to 1*.*
Then the bipartite graph has a matching if and only if the network has a flow of value n*.*

Consider reducing *validity*: Is a Boolean expression E always `True`. It can be reduced to *satisfiability*.

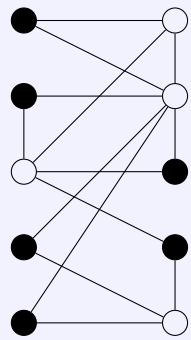
Definition 19: Validity of a Boolean Expression

A Boolean expression ϕ *is* valid *if it is* `True` *for every assignment of* `True` *or* `False` *to its variables.*

To show that Boolean expression ϕ valid, show that $\neg\phi$ is not satisfiable. If $\neg\phi$ has no satisfying truth assignment: $\neg\phi$ is always `False`. Therefore, ϕ is always `True` and valid.

Problem 20: Independent-Set

Given a graph $G = (V, E)$ *and an integer* k*, is there a subset of vertices* $S \subseteq V$ *such that* $|S| \geq k$*, and for each edge at most one of its endpoints is in* S*?*
The graph below shows an independent set of size 6*, the black nodes.*



Problem 21: Vertex-Cover

Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge at least one of its endpoints is in S?
The graph in problem show a vertex cover of size 4, the white nodes.

Theorem 17: Reducibility: Vertex-cover and Independent-Set

There is a polynomial time reduction of vertex-cover to independent-set. A subset of nodes S is an independent set if and only if $V - S$ is a vertex cover.

Proof: Reducibility: Vertex-cover and Independent-Set

Let S be an independent set of size k. Then $V - S$ is of size $n - k$. If (u, v) is an edge, then either $u \notin S$ or $v \notin S$ (or both) Therefore, either $u \in V - S$ or $v \notin V - S$ (or both). That is, for each edge at least one of its nodes is in $V - S$.
On the other hand, Let $V - S$ be a vertex cover of size $n - k$. Then S is of size k. Let $u \in S$ and $v \in S$. It must be $(u, v) \notin E$ because $V - S$ is a vertex cover. (If $(u, v) \in E$, then at least one of u or v is in a vertex cover.) Therefore, no two nodes in S are joined by an edge, that is, S is an independent set.

NP-*Complete Problems*

The book (Garey and Johnson, 1979) is the classic textbook on NP-completeness. A surprising number of problems have been shown to be NP-complete.

Definition 20: NP-Complete

A decision problem C is NP-complete if:

1. $C \in NP$, *and*
2. *Every problem in NP is reducible to C in polynomial time.*

Intuitively, NP-complete problems are the hardest in NP. It is not clear that there are any NP-complete problem C. And, showing that *every* problem in NP reduces to C seems to be an insurmountable task.

Cook's theorem addresses the first issue.

Theorem 18: Cook's Theorem

SAT is NP-complete.

The proof is well beyond the scope of this class.

The second issue is addressed by this result.

Lemma 1: Reduction from NP-complete problems

Let A and B be problems in NP.
If A is NP-complete and $A \leq_P B$, then B is NP-complete.

Example: 3SAT is NP-complete

Let ϕ be a Boolean expression in conjunctive normal form where each clause has at most 3 literals. Does ϕ have a truth assignment?

Co-NP *Problems*

PRIMES and COMPOSITE are examples of complementary problems.

What are precise definitions of "is prime" and "is composite"?

- PRIME $= \{n : n \in \mathbb{N}$ and n is prime$\}$.
- COMPOSITE $= \{n : n \in \mathbb{N}$ and n is composite$\}$.

Theorem 19: P is closed under complements

If problem X is in class P, then its complement $\bar{X}$ is in P too.
That is, if $X \in P$. then $\bar{X} \in P$, or more simply, $P = \text{co-}P$.

Proof: P is closed under complements

Let A be a polynomial time deterministic algorithm for decision problem X. An algorithm $\bar{A}$ for $\bar{X}$ runs A on an instance I of X. If A accepts I, then $\bar{A}$ rejects I. Conversely, if A rejects I, then $\bar{A}$ accepts I.

For the class NP, the relationship between NP and co-NP is not as clear.

If $X \in$ NP, then there is a *certificate* (a True *solution*) can be *checked* in polynomial time. The *complementary* problem $\bar{X}$ requires a polynomial time *disqualification*. That is, a short proof for *no* instances.

Definition 21: Co-NP

$$co\text{-}NP = \left\{X : \bar{X} \in NP\right\}$$

The COMPOSITE decision problem is: Given a natural number $n > 1$, does it have factors other than 1 and itelf? COMPOSITE is in NP. Given the prime factorization, you can quicky check that its product is n.

The PRIMES decision problem is: Given a natural number $n > 1$, does it have no factors other than 1 and itelf? By definition PRIMES is in co-NP.

Interesting: PRIMES $\in$ P was demonstrated by a 2002 paper (Agrawal et al., 2002).

The *subset sum* problem is in NP.

Problem 22: Subset Sum

Let $\mathbb{A}$ be a finite set of integers. Does $\mathbb{A}$ contain a non-empty subset the sums to 0?

You can check in linear time that values in a non-empty subset sum to 0.

The complementary subset sum problem requires that all non-empty subsets have non-zero sums.

Problem 23: Co-Subset Sum

Let $\mathbb{A}$ be a finite set of integers. Does every non-empty subset of $\mathbb{A}$ sum to a non-zero value?

Problem 24: Unsatisfiability

<u>Decision Problem:</u> Given a Boolean expression B of n literals in conjunctive normal form, does B have no satisfying truth *assignment?*

SAT is in NP: Given a truth assignment that satisfies a Boolean expression B, it can be checked in polynomial time.

UNSAT is in co-NP by definition, it may not be simple to prove there is no satisfying truth assignment.

Problem 25: No Hamiltonian Cycle

Decision Problem: Given a graph $G = (V, E)$, is there no simple cycle that contains every node of V?

Can give a permutation of nodes to prove there is a Hamiltonian cycle. How can you prove there is no Hamiltonian cycle?

NP-*Hard Problems*

Definition 22: NP-Hard

Decision problem H is NP-hard if every NP-complete problem C can be reduced to H in polynomial time.

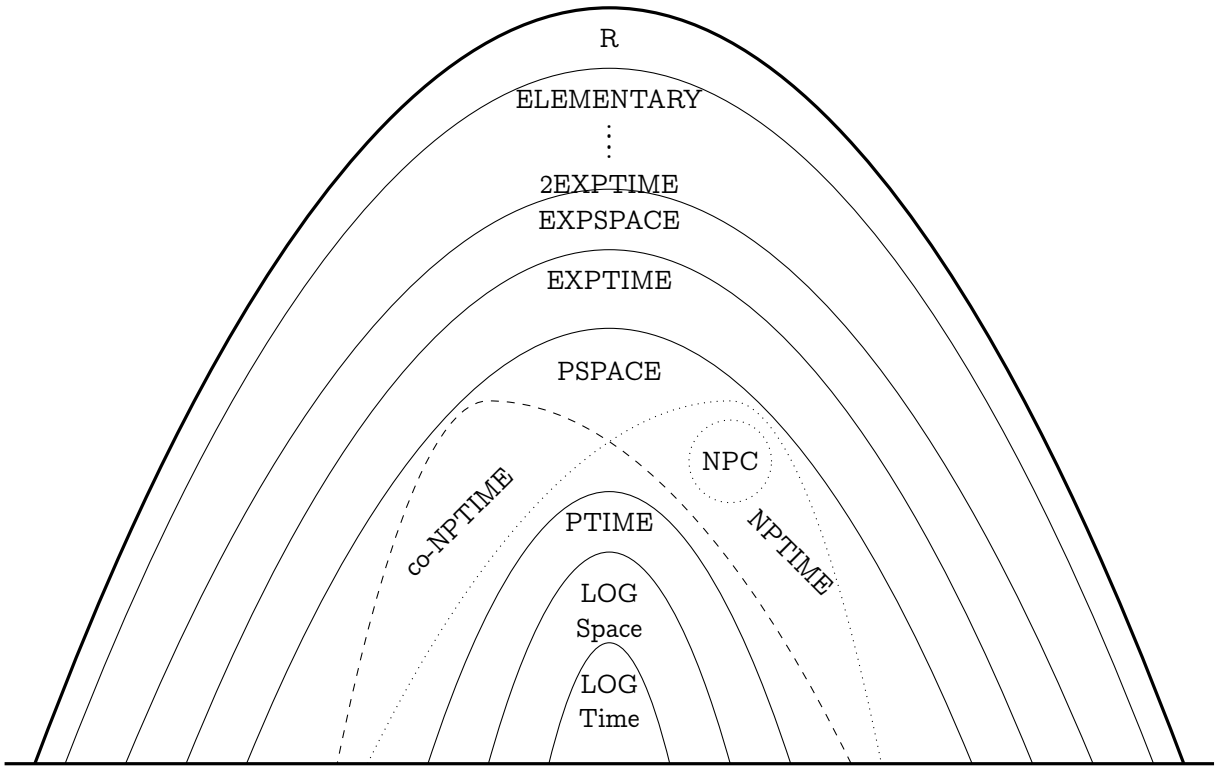
An NP-hard problem is at least as hard as the hardest problems in NP.

The *halting problem* is NP-hard. SAT can be reduced to the halting problem by transforming SAT into a Turing machine that tries all possible truth assignments for an instance I of SAT. When the machine finds a satsifying truth assignment it halts. Otherwise, if there is no satisfying truth assignment, the machine goes into an infinite loop.

PSPACE

Complexity Hierarchy

Computational complexity is complex. Here is an image from (Papadimitriou, 1994) (created by Sebastian Sardina) that shows the relationship among several complexity classes under common assumptions that have not been fully proven.


R
ELEMENTARY
2EXPTIME
EXPSPACE
EXPTIME
PSPACE
NPC
co-NPTIME
NPTIME
PTIME
LOG
Space
LOG
Time

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html.

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff. [page 120]

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360.

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15. [page 119]

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The* C *Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355. [page 119]

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857. [page 119]

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index

HASKELL programming language, 20, 31, 44, 116, 137
JAVA programming language, 31

Ackermann function, 174
Axioms
  Peano, 192

Bézout, Étienne, 47

C programming language, 30, 31, 42, 43, 55, 109, 152, 153
Calendar, 7
Cartesian product, 142
Catalan numbers, 156, 161
Computability, 191
Computable function, 28
Computer Language Benchmark Game, 30
Contradiction, 193

Dijkstra's algorithm, 183
DNA, 146

Empty
  String λ, 61

Fibonacci, 44, 46, 142
Floating point
  Machine epsilon, 50
Function
  Ackermamm, 174
Functional programming, 31

Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstader, Douglas, 25

Kleene, Stephen, 53
Knuth, Donald, 30

Lamé, Gabreil, 44
λ calculus, 26, 28
Linear congruence, 47
Listing
  C
    Matrix Chain Multiplication, 158
Listings
  C
    A Minimum Algorithm, 135
    Aho-Corasick, 89
    Boyer-Moore Pattern Matching, 72
    Brute-force Left-to-Right Pattern Matching, 56
    Bubble Sort, 109
    Build a Heap, 126
    Counting Sort, 128
    Direct Address Table Operations, 94
    Edit Distance, 152
    Find Set with Path Compression, 177
    Greatest Common Divisor, 42
    Hash Table with Chaining, 95
    Heap Sort, 127
    Heapify a List, 126
    Inner Product, 155
    Insertion Sort, 112
    Knuth–Morris–Pratt Pattern Matching, 67
    Maximum Subsequence Sum (Cubic), 34
    Maximum Subsequence Sum (Linear), 35
    Merge Sort, 118
    Morris–Pratt Pattern Matching, 58
    Partition about a pivot, 120
    Quick Sort, 120
    Radix Exchange Sort, 132
    Radix Sort, 129
    Randomized Partition, 136
    Randomized Selection, 138
    Rational Knapsack, 167
    Selection Sort, 114
    Shell Sort, 116
    Union by rank, 177
  Haskell
    Ackermann function, 174
    Apply a Function Repeatedly, 50
    Brute-force Left-to-Right Pattern Matching, 57
    Bubble Function, 108
    Bubble Sort, 109
    Extended Euclidean Algorithm, 48
    Fold from the left, 37
    Greatest Common Divisor, 44
    Inner Product, 155
    Insertion Sort, 112
    Matrix Multiplication, 155
    Maximum Subsequence Sum, 37
    Merge Sort, 117
    Merging Sorted Lists, 117
    Newton's Square Root Method, 51
    Newton's Square Root Recurrence, 50
    Quicksort, 120
    Randomized Partition, 137
    Relative Error Convergece Test, 50
    Sample Code Header, 31
    Second of a pair, 36
    Selection Sort, 114
    Sorted Decision Problem, 106

Splitting a List, 117
Swapping head with Another Element, 137
Pseudocode
Activity Selection, 170
Bucket Sort, 130
Dijkstra's Algorithm, 186
Kruskal Minimum Spanning Tree, 172
Prim Minimal Spanning Tree, 181
Lucas, Édouard, 44

Memorization, 141
Metric sace, 152

Newton's method, 13
Newton, Issac, 49
Noweb, 30, 55

Object-oriented programming, 31

Pattern Matching, 53, 54
Boyer-Moore, 71
Brute Force, 55
Knuth–Morris–Pratt, 67
Morris–Pratt, 58
Right-to-Left Brute Force, 70
φ, 142
Presburger arithmetic, 191

RAM, 26, 29

Sequence, 19
Lucas, 44
Sets
Cartesian product, 142
Simpson's rule, 51
Sorting Algorithms
Bubble Sort, 107
Bucket Sort, 129
Counting Sort, 128
Heap Sort, 124
Insertion Srt, 111
Merge Sort, 116
Quick Sort, 119
Radix Sort, 129
Selection Sort, 114
Shell Sort, 115
Time complexity, 127
String
Border, 61
Period, 61
Prefix, 61
Suffix, 61
Syllabus, 5

Theorems
Bezout's, 47
Euclidean division, 41
Lamé's, 45
Taylor's, 49, 51
Turing
Machine, 9, 26, 28, 192

Wikipedia, 201

xkcd
Algorithms by Complexity, 25
Calendar of meaningful dates, 7
Haskell, 30
Ineffective Sorts, 105
Traveling Salesman Problem, 141