Reducing Reductions to Intuition

Douglas Wikström
dog@kth.se

November 9, 2021

Abstract

The goal of this note is: to give an introduction to various types of reductions that are common in complexity theory through intuition and physical or everyday analogs, but still present rigorous definitions.

1 Introduction

We introduce Turing machines, relations, languages, problems, and reductions as in any textbook, but the goal of this presentation is to do this in a way that emphasizes that little of these ideas are really surprising to the reader.

Computation is another word for deciding something based on some input, relations merely relate objects, language is a fancy way of saying subset, we solve problems every day, and we use routinely use reductions to solve problems. Thus, throughout we try to give intuition based on well-known physical or virtual concepts for every notion we introduce. Let us begin by considering a puzzle familiar with most readers.

1.1 Positive View: Solving Rubic's Cube

There are several ways to solve a Rubic's cube, but the beginner's method[1] is to: (1) solve the bottom layer including center pieces, (2) solve the second layer, (3) solve a cross, and (4) permute the corners into place. For each step there is what cubers call an *algorithm*, i.e., a predefined sequence of moves.

Thus, solving any cube is *reduced* to applying the algorithms in the right order. In fact, we could teach the algorithms to four people A, B, C, and D, respectively, and a fifth person R could orchestrate the process by simply asking them in order to execute their algorithms. Interestingly, R does not need to know more than the order of the four and could consider them jointly as an *oracle* which on input a name of a person and a cube returns the cube after applying the appropriate algorithm.

Although there are physical constraints, a Rubic's cube can in principle be generalized to dimensions $n \times n \times n$ for any $n \in \mathbb{N}$ and any cube can be represented as a bit string.

If we are interested in the complete sequence of moves, we call the problem a *construction problem* and in this case R would need the other people to provide their operations as part of its output to form the complete constructive solution. There are of course a very large number of possible sequences of moves to solve the cube. It is quite natural to expect a third party to want to verify a constructive solution *efficiently*, which often is interpreted as polynomial time in n.

[1] GP Woodworking & Designs, Greg's Channel `https://www.youtube.com/watch?v=6Umv5q9yCcg`.

We could also ask for the minimal number of moves needed to solve a given cube, without paying any attention to the sequence of moves used, in which case it would be an *optimization problem*.

Finally, we could ask if the solved cube can be reached with a sequence of moves of length at most $t \in \mathbb{N}$ provided along with the cube, in which case we have derived a *decision problem* from the optimization problem.

1.2 Negative View: Failing to Solve Rubic's Cube

Above we used a reduction to four standard "algorithms" from the cubing community to illustrate how a reduction without deeper knowledge of a problem can outsource solving parts of the problem. We can also illustrate the other, equally important, use of reductions in complexity theory with Rubic's cube.

Consider $5 \times 5 \times 5$ cubes.[2] We observe that some such cubes are really $3 \times 3 \times 3$ cubes in disguise, i.e., each $2 \times 2 \times 2$ corner of the cube is locally consistent, and that such a cube can be solved keeping these corners intact as if it was a $3 \times 3 \times 3$ cube. Thus, if solving $3 \times 3 \times 3$ is hard, then solving $5 \times 5 \times 5$ cubes is hard at least for some cubes, or more generally: solving $n \times n \times n$ cubes for odd n is hard if solving $3 \times 3 \times 3$ cubes is hard.

NP hard problems, e.g., 3-SAT, is a class of problems believed to be hard to solve. Thus, one way of arguing that another problem $\mathcal{P}$ is hard is to show that we can solve 3-SAT if we had an oracle for solving $\mathcal{P}$.

This kind of argument is commonly used in everyday life. A statement such as "I cannot become a ballet dancer because I am not flexible enough." is a reduction of being able to become a ballet dancer to at least be flexible enough. Bystanders accept the claim if they believe that there is no way that the person who makes the claim can ever be flexible enough, regardless of everything else it takes to become a ballet dancer.[3]

1.3 Computational Model

To discuss reductions we need a computational model. In complexity theory the two most common models are Turing machines and circuit families. For completeness we provide definitions for the former in Appendix A, but the reader should be able to follow the note even without reading this.

We merely remind the reader that if A is an algorithm that uses a function f as an oracle, then this is denoted $\mathsf{A}^{f(\cdot)}$ and simply means that the algorithm can evaluate the function f instantly and only pay the computational cost for writing the input and reading the output.

2 Languages and Relations

A language is a subset of a countable set and in computer science our canonical countable set is the set of all finite strings $\{0,1\}^*$. A relation is similarly a subset of a Cartesian product of two sets, but in computer science we restrict our attention to $\{0,1\}^* \times \{0,1\}^*$.

If $\mathcal{R} \subset \{0,1\}^* \times \{0,1\}^*$ is a relation, then we denote by $\mathcal{R}(x) = \{w \mid (x,w) \in \mathcal{R}\}$ the set of witnesses of x, and we denote by $\rho_{\mathcal{R}}$ the characteristic function of $\mathcal{R}$. The characteristic function simply outputs 1 on input $(x,w) \in \mathcal{R}$ and 0 otherwise.

[2] J Perm `https://www.youtube.com/watch?v=d1I-jJlVwB4`.

[3] They could of course all be wrong! Perhaps the person is one in a million that is a late bloomer. Similarly, it *could* be the case, against all belief, that $\mathsf{NP} = \mathsf{P}$ which would mean that there are polynomial time algorithms for all NP complete problems.

Intuition 1 (Mathematical Proof)**.** Mathematics is based on a meta language and axioms which are taken for granted. For each statement there may be several different proofs[4]. If we pair up each provable statement[5] with its set of proofs we get a *relation*. Needless to say, short proofs are preferable, since otherwise nobody bothers reading the proof, except a computer! A classical example is the computer assisted proof of the four color theorem[6]. The set of statements for which there is a proof is a *language*.

The intuition motivates the following definitions.

Definition 1 (NP Relation)**.** A relation $\mathcal{R} \subset \{0,1\}^* \times \{0,1\}^*$ is an *NP relation* if $\rho_{\mathcal{R}}$ is polynomial time computable and there exists a constant c such that for every $x \in \{0,1\}^*$: $\mathcal{R}(x) \subset \{0,1\}^{|x|^c}$.

The second requirement ensures that we can think of the running time in terms of $|x|$. Given an NP relation we can define a language

$$L_{\mathcal{R}} = \left\{x \in \{0,1\}^* \mid \exists w \in \{0,1\}^* : (x,w) \in \mathcal{R}\right\}$$

and define the notion of an NP language.

Definition 2 (NP Language)**.** A language $L \subset \{0,1\}^*$ is an *NP language* if there is an NP relation $\mathcal{R}$ such that $L = L_{\mathcal{R}}$.

There may be more than one NP relation that defines the same language. Thus, the definition simply says that there is at least one way of describing the language using a relation which is polynomially balanced and has a characteristic function which is polynomial time computable.

3 Combinatorial Problems

Combinatorial problems come in various forms and are expressed in terms of languages and relations, but we may order the types of problems by how much information is expected by a solution.

Intuition 2 (Cars)**.** Given the regulations for street cars, it is a *construction problem* to output a design of a car that satisfies the regulations. It is an *optimization problem* to determine the top speed, fastest acceleration, or any other measurable property, of a car that satisfies the regulations for street cars. It is a *decision problem* to, given a speed s, output 1 if it is possible to construct a car that has a top speed of at least s and output 0 otherwise.

We may generalize from our intuition and state this informally as follows before we give rigorous definitions.

1. To solve a construction problem, an algorithm must for any instance construct and output a feasible solution.

2. To solve an optimization problem, an algorithm must for any input find the maximum (or minimum) value of any feasible solution, using a given measure.

[4]Hundreds in the case of the quadratic residuosity law!
[5]We cannot consider true statements, since Gödel proved that some true statements lack proofs.
[6]`https://www.liquisearch.com/four_color_theorem/history/proof_by_computer`

3. To solve a decision problem derived from an optimization problem, an algorithm need only output a bit indicating if there is a solution with a measure greater or equal to (or smaller or equal to) a given threshold.

Definition 3 (Construction Problem)**.** A *construction problem* is a tuple $\mathcal{P} = (I, f)$, where $I \subset \{0,1\}^*$ is a set of instances, and for every instance $x \in I$: $f(x) \subset \{0,1\}^*$ is a finite set of solutions. An algorithm A solves $\mathcal{P}$ in time $T(\cdot)$ if it for every $x \in I$ outputs $y \in f(x)$ in time $T(|x|)$.

If the relation $\mathcal{R} = \{(x, w) \mid x \in I \wedge w \in f(x)\}$ defined by $\mathcal{P}$ is an NP relation, then it is an *NP construction problem.*

Definition 4 (Optimization Problem)**.** An *optimization problem* is a tuple $\mathcal{P} = (I, f, v, g)$, where (I, f) is a construction problem with relation $\mathcal{R}$, $v : \mathcal{R} \to \mathbb{R}_+$ is a polynomial time computable measure, and a goal function $g \in \{\min, \max\}$. An algorithm A solves $\mathcal{P}$ in time $T(\cdot)$ if it for every $x \in I$ outputs $m(x) = g_{w \in f(x)}\{v(x, w)\}$ in time $T(|x|)$.

If $\mathcal{R}$ is an NP relation, then it is an *NP optimization problem.*

Definition 5 (Decision Problem)**.** A *decision problem* is a pair $\mathcal{P} = (I, L)$, where $L \subset I \subset \{0,1\}^*$. An algorithm A solves $\mathcal{P}$ if for every $x \in I$ it outputs 1 if $x \in L$ and 0 otherwise.

If $\mathcal{P}$ is an *NP decision problem* if L is an NP language.

Definition 6 (Derived Decision Problem)**.** A *decision problem* of a max-optimization problem $\mathcal{P} = (I, f, v, g)$ with relation $\mathcal{R}$ and is the decision problem for the language $L = \{(x, t) \mid x \in I \wedge m(x) \geq t\}$. The decision problem for a min-optimization problem is defined correspondingly.

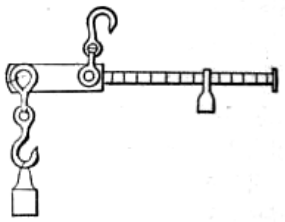
3.1 Reduction of an Optimization Problem to a Decision Problem

It is natural to ask to what extent we can reduce one of the above problems to another. It is trivial to reduce a decision problem to its optimization problem, since it is merely a matter of comparing two numbers after computing the optimal value.

The following physical analogue gives the right idea for reducing an optimization problem to its decision problems.

Intuition 3 (Steelyard Balance Scale[7])**.**
A steelyard balance scale is used as follows. The item to be weighed is attached to the hook and then the counter weight is placed in some position.

If the bar slants to the left, then the counter weight is moved to the right, and if the bar slants to the right, then the counter weight is moved to the left. This is repeated until the bar is balanced. In each attempt we get a binary decision. Thus, we may use binary search to position the counter weight correctly.



Lemma 1. If $\mathcal{P}$ is an NP optimization problem and $\mathcal{P}'$ its NP decision problem, then $\mathcal{P}$ is reducible in polynomial time to $\mathcal{P}'$.

Proof. Consider a minimization problem. For a given instance x we may consider the outputs $b_1, b_2, b_3, \ldots$ of the oracle for the decision problem on inputs $(x, 0), (x, 1), (x, 2) \ldots$, but testing all values is too slow in general. However, we know that the sequence of decision bits is monotonically non-decreasing and shifts from 0 to 1 within $2^{p(|x|)}$ for some

[7] `https://en.wikipedia.org/wiki/Steelyard_balance`

polynomial $p(\cdot)$, since $\mathcal{P}$ is an NP problem, i.e., the optimal value must be possible to write in polynomial time.

Thus, we proceed as follows: we set $s = 0$ and query the decision oracle to get b_{2^i} for $i = 0, 1, 2, \ldots$ until for some j we have $b_{2^j} = 1$. Then we set $e = 2^j$ and perform a binary search between the indices s and e for the index k such that $b_{k-1} = 0$ and $b_k = 1$. This requires at most a polynomial number of oracle queries in $|x|$. The maximization case is similar. □

4 Reductions of Decision Problems

We now have all the necessary intuition and notation to consider reductions of decision problems.

4.1 Turing Reductions

A Turing reductions is a reduction of one decision problem to another without any restrictions. We give intuition and examples for general reductions below, since it is harder to give good examples for decision problems, but they illustrate the generality of how a Turing reduction may interact with its oracle.

Intuition 4 (Pocket Calculator)**.** A human can describe, understand, and prove the correctness of an abstract calculation without being able to perform the calculation for more than a handful of small inputs.

With the access of a simple pocket calculator the computations can be performed by even a school kid. The pocket calculator is an oracle to humans, but the way it is used can be quite creative.

Example 1 (Addition)**.** Denote by $\mathcal{P}_{\mathsf{ADD}_n}$ the problem of adding two natural numbers represented in basis two.

Suppose that $\mathsf{ADD}_2 : \{0,1\}^3 \to \{0,1\}^2$ is an algorithm that takes as input three bits a_0, b_0, and c_0, where c_0 is called the carry-in, and gives as output $a_0 + b_0 + c_0$ in basis two (c_1, s_0), where c_1 is called the carry-out, i.e., $a_0 + b_0 + c_0 = 2 \cdot c_1 + s_0$. Two bits suffice to represent the former sum, since it is bounded by 3.

Then we can construct an addition algorithm $\mathsf{ADD}_n : \{0,1\}^{2n+1} \to \{0,1\}^{n+1}$ for n-bit integers a and b in basis two as follows. Set $c_0 = 0$, compute $(c_{i+1}, s_i) = \mathsf{ADD}_2(c_i, a_i, b_i)$ for $i = 0, \ldots, n-1$, and output $(c_n, s_{n-1}, \ldots, s_0)$, which is $a + b$ in basis two.

Thus, if we have an oracle $\mathsf{ADD}_n(\cdot, \cdot, \cdot)$, then we can implement addition for integers of any bit length using a Turing machine that merely manipulates a couple of bits a at a time.

Example 2 (Ford-Fulkerson)**.** Let $\mathcal{P}_{\mathsf{BFS}}$ be the problem of enumerating the nodes in a graph starting from a node s such that all nodes at distance d are visited before any node at a distance $d + 1$ are visited for $d = 1, 2, 3, \ldots$ until a node is encountered that satisfy predicate $\rho(\cdot)$, or all nodes are enumerated. The breadth-first search algorithm BFS_ρ that simply halts whenever it encounters a node t such that $\rho(t) = 1$ solves this problem.

Let $\mathcal{P}_{\mathsf{short}}$ the problem of finding the shortest path from a node s to a node t in a graph. Then $\mathsf{S}(G, s, t) = \mathsf{BFS}_\rho(G, s)$, where $\rho(x) = (x = t)$ solves this problem if we also keep track of predecessors in the order in which nodes are visited.

Denote by $\mathcal{P}_{\mathsf{flow}}$ the problem of finding the maximal flow from a node s to a node t in a graph G. The Ford-Fulkersson maximum flow algorithm $\mathsf{F}(G, s, t)$ repeatedly invokes $\mathsf{S}(G, s, t)$ on an evolving residual graph to find additional flow between s and t.

In other words we can: (1) reduce the maximum flow problem to the problem of finding a shortest path, and (2) reduce the problem of finding a shortest path to breadth-first search, in polynomial time.

4.2 Cook Reductions

Although Turing reductions are interesting in their own right the definition of a Turing reduction is rather generous in that its running time is unbounded. Consider the following two real-world examples to understand why this matters.

Intuition 5 (Cutting a Circle)**.** Recall Archimedes idea for approximating the area of the circle: compute the area of a regular polygon with k vertices and increase k until the approximation is sufficiently precise.[8]

Imagine that we had an oracle for cutting off a single protruding vertex with a straight cut. Then we can by repeatedly calling the oracle start with a square sheet of paper and turn it into a polygon which approximates a circle arbitrarily well.

However, the number of vertices grows exponentially fast. Thus, although this is a reduction of approximating the circle to the problem of cutting off a protruding vertex it is not an efficient one.

Intuition 6 (Tiling)**.** Consider the problem of tiling a bathroom with perfect boundary distances between the tiles. We could use a yardstick to position each tile separately, but this would be time consuming even if the number of tiles is bounded.

Suppose that we had an oracle for positioning each tile. Then tiling would amount to applying tile fix. The reason why tiling perfectly can be done efficiently is that a tiling wedge is an effectively a positioning oracle.

A Cook reduction is also called a *polynomial time reduction*, or even *poly-time reduction*, and simply means that we require that a Turing reduction runs in polynomial time. This is illustrated in Figure 4.2.

Definition 7 (Cook Reduction)**.** A *Cook reduction* from a decision problem $\mathcal{P}$ to a decision problem $\mathcal{P}'$ is a polynomial time Turing reduction from $\mathcal{P}$ to $\mathcal{P}'$.

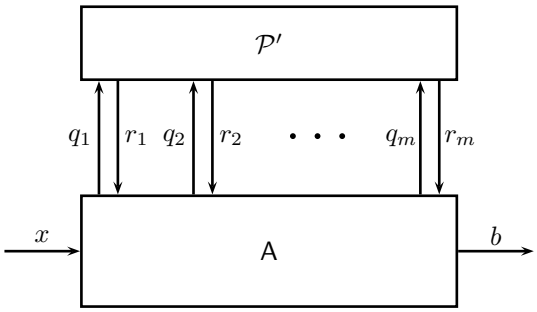


Figure 1: The running time of a Cook reduction A is bounded by some polynomial $p(n)$ in the input size n. This implies that it also queries its oracle at most $m(n) \leq p(n)$ times.

Cook reductions behave well under composition in the sense that we can stack a constant number of reductions of problems on top of each other, each solving a problem using the previous as an oracle and end up with another Cook reduction. Moreover, if the first problem is solvable in polynomial time, then the last is as well. This is captured by the following lemma.

[8] `https://www.youtube.com/watch?v=zUVx0TQaxME`

Lemma 2. If $\mathsf{A}^{\mathcal{P}_0(\cdot)}$ is a Cook reduction of $\mathcal{P}_1$ to $\mathcal{P}_0$, and $\mathsf{B}^{\mathcal{P}_1(\cdot)}$ is a Cook reduction of $\mathcal{P}_2$ to $\mathcal{P}_1$, then there is a Cook reduction $\mathsf{C}^{\mathcal{P}_0(\cdot)}$ from $\mathcal{P}_2$ to $\mathcal{P}_0$.

Furthermore, if $\mathcal{P}_0$ is solvable in polynomial time, then there is a polynomial time algorithm for solving $\mathcal{P}_2$.

4.3 Karp Reductions

A Karp reduction is a special type of Cook reduction which translates an instance of one problem to an instance of another in such a way that solving the latter gives the answer to the former problem. The following intuition illustrates the principle.

Intuition 7 (Volume)**.** Let $\mathcal{P}$ be the problem of measuring the volume of a stone of arbitrary shape, but bounded to fit into a given container, and let $\mathcal{P}'$ be the problem of measuring the volume of water.

We may translate an instance of the former to an instance of the latter by placing the container filled with water within a larger container and simply placing the stone into the filled container. A stone is an instance of $\mathcal{P}$ and the spill water is an instance of $\mathcal{P}'$. A single measurement is needed to solve the latter problem.

Suppose that there is a polynomial time algorithm T that translates an instance x of a decision problem $\mathcal{P}$ into an instance x' of a decision problem $\mathcal{P}'$ such that $x' \in L'$ if and only if $x \in L$. Then we can trivially construct reduction as follows. Given an instance x, compute an instance $x' = \mathsf{T}(x)$ of $\mathcal{P}'$, and output the reply to the query x' to the oracle $\mathcal{P}'(x')$. We stress that the map is an *injection* and not necessarily a bijection. Even if there is a corresponding polynomial time algorithm T' in the reverse direction we do not necessarily have $\mathsf{T}'(\mathsf{T}(x)) = x$.

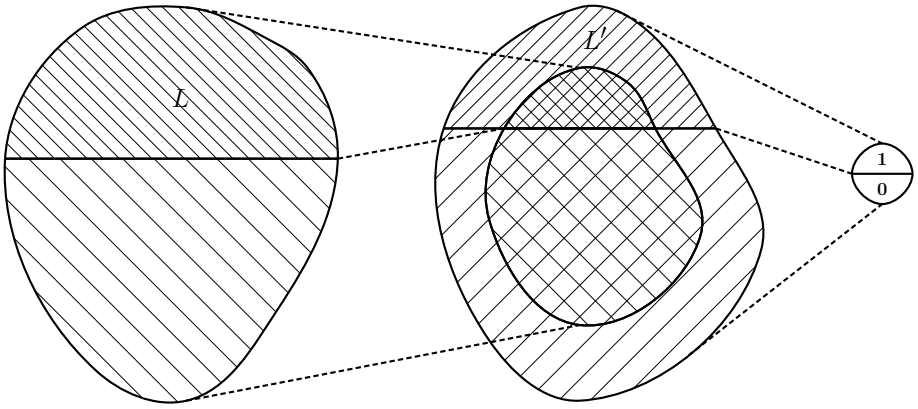


Figure 2: A Karp reduction is essentially an efficiently computable injection that maps L into a subset of L', and the complement of L into a subset of the complement of L'.

In other words, a Karp reduction uses its oracle only once and only in the natural way. We may define it as follows, which is illustrated in Figure 4.3.

Definition 8 (Karp Reduction)**.** A *Karp reduction* $\mathsf{A}^{\mathcal{P}'(\cdot)}$ from $\mathcal{P}$ to $\mathcal{P}'$ is a Cook reduction defined by a $\mathsf{A}^{\mathcal{P}'(\cdot)}(x) = \mathcal{P}'\big(\mathsf{T}(x)\big)$, where T is a polynomial time algorithm such that $\mathsf{T}(L) \subset L'$ and $\mathsf{T}\big(\overline{L}\big) \subset \overline{L'}$.

It is easy to see that this is a stronger requirement than simply restricting a Cook reduction to make exactly one oracle query. Indeed, it may well be possible for some problems to, e.g., construct an algorithm $\mathsf{T}_\perp$ such that $\mathsf{T}_\perp(L) \subset \overline{L'}$ and $\mathsf{T}_\perp(\overline{L}) \subset L'$

and let $\mathsf{A}_{\perp}^{\mathcal{P}'(\cdot)}(x) = 1 - \mathcal{P}'\big(\mathsf{T}_{\perp}(x)\big)$. This queries its oracle exactly once, but it is strictly speaking not a Karp reduction of $\mathcal{P}$ to $\mathcal{P}'$.

What makes Karp reductions important is that they are strikingly intuitive, relatively easy to analyze, and sufficient to prove NP completeness for a wide variety of problem. If we think of the oracle as a subroutine with unbounded running time, then we can depict a Karp reduction as follows.

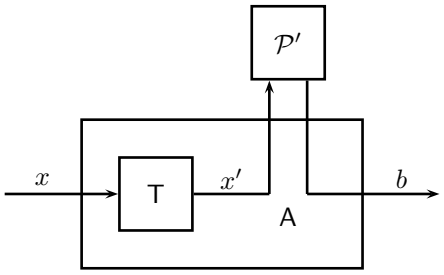


Figure 3: A Karp reduction takes an instance x of $\mathcal{P}$ as input, translates it into an instance x' of $\mathcal{P}'$, and queries its oracle exactly once on x' and outputs its reply.

We are now ready to consider the archetypal example of a Karp reduction, which has the special property that the two problems are reducible to each other in such a way that the composition of the translations is the identity function.

Example 3 (Clique to Independent Set)**.** An instance of the clique decision problem $\mathcal{P}_{\text{clique}}$ consists of an undirected simple graph $G = (V, E)$ and a natural number k and solving it means deciding if G contains a clique[9] of size k or not. A solution to the independence set problem $\mathcal{P}_{\text{ind}}$ is instead to decide if there is an independence set[10] of size k. The complement of a graph $G = (V, E)$ is the graph $G' = (V, E')$, where the edges are defined by $E' = \big\{e \in V \times V \mid e \notin E\big\}$.

It is easy to see that we can compute the complement of a graph in time $O(|V|^2)$ regardless of representation. Thus, if we have an algorithm C for the former problem, then we can construct an algorithm I for the latter by simply defining $\mathsf{I}(G) = \mathsf{C}(G')$.

This is a Karp reduction, since we transform an instance of G into an instance G'.

Interestingly, something is lost in translation. The topological properties of a graph and its complement are different, which means that approximation algorithms do not perform equally well on the problems. This is a nice example of why the study of approximation algorithms uncovers more than simply how well we can find reasonably good solutions to problems believed to be hard.

5 Comparison of Reductions

We can summarize some properties of the reductions we have covered so far in a table. Needless to say, polynomial time is a crude measure and sometimes we wish to keep track of running time and the number of oracle calls separately.

[9] A clique is a set of nodes such that there is an edge between every pair of nodes.

[10] An independence set is a set of nodes which have no edges between them.

Reduction	Running time	Oracle queries
Turing reduction	Unbounded	Unbounded
Cook reduction	Polynomial time	Polynomial
Karp reduction	Polynomial time	1

Table 1:

A Turing Machines

Recall the definition of a Turing machine.[11][12] It has a finite set of states Q and uses a finite alphabet of symbols Γ. A subset $\Sigma \subseteq \Gamma \setminus \{b\}$ of the alphabet, except the blank, is used for inputs. One state s is defined to be the state in which the machine is when started, and another set $A \subseteq Q$ is the set of states where we consider the Turing machine to have halted.

A Turing machine also has access to a tape, i.e., an array where each cell can store a symbol from Γ. When the execution starts the Turing machine reads from a given position of the tape, which may be thought of as position 1 of a tape with a cell for each integer, i.e., it is infinite in both directions. Furthermore, the tape is initialized with the input $x = (x_1, \ldots, x_n)$ in the cells at positions $1, 2, 3, \ldots, n$, and blanks in every other cell.

Finally, the execution proceeds by repeated application of a transition function δ that takes the current state $q \in Q \setminus A$ and the contents $y \in \Gamma$ of the cell c at which the Turing machine is positioned, and outputs: the new state $q' \in Q$ of the Turing machine, the new contents $y' \in \Gamma$ of c, either the symbol L or R indicating if the Turing machine is positioned to the left or right of c after the state transition.

There are several great physical Turing machines built from LEGO and wood in clips on YouTube[13][14] [15] Note that if a Turing machine only uses a constant number of cells during its execution, then it is a finite automaton. Thus, the tape is essential to make Turing machines as computationally powerful as they are.

Definition 9 (Turing Machine)**.** A *Turing machine* is a tuple $M = (Q, \Gamma, b, \Sigma, \delta, s, A)$, where:

- Q is a finite non-empty set of states,
- Γ is a finite non-empty alphabet, where $b \in \Gamma$ is the blank symbol, and $\Sigma \subseteq \Gamma \setminus \{b\}$ are the input symbols,
- $\delta : (Q \setminus A) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $s \in Q$ is the starting state, and $A \subseteq Q$ are the accepting states.

Although the cells may be thought of as indexed by integers from the starting cell at index 1, there is no mechanism for the Turing machine to know at which index it is currently located relative the starting cell unless it implements this on its own.

To simplify the descriptions of Turing machines it is the convention that a partial definition of δ on a subset of $D \subset Q \setminus A \times \Gamma$ is completed to the full domain $Q \setminus A \times \Gamma$ by setting δ to some state in A for any element outside D.

[11] Computerphile `https://www.youtube.com/watch?v=dNRDvLACg5Q`
[12] EngMicroLectures `https://www.youtube.com/watch?v=GIQDA5Gnxkc`
[13] Made from LEGO `https://www.youtube.com/watch?v=cYw2ewoO6c4`.
[14] Made from LEGO `https://www.youtube.com/watch?v=kGvf2qESZsQ`.
[15] Made from wood `https://www.youtube.com/watch?v=vo8izCKHiF0`.

We may always think of the alphabet of a Turing machine as $\Gamma = \{0, 1, \bot\}$, where $\bot$ is the blank symbol, (or even simply $\{0, 1\}$), but it is convenient to have the liberty of using a larger alphabet with an intuitive meaning as illustrated below.

Example 4 (Alphabet of Turing Machine)**.** We can let $\Gamma = \{0, 1\}^8$, i.e., the set of all bytes, and let NULL $b = 00000000$ be the blank. Perhaps we know that the input is a sequence of capital letters encoded in ASCII, i.e., a binary strings in the set $\Sigma = \{01000001, 01000010, \ldots, 01011010\}$. Note that this is a strict subset of $\Gamma \setminus \{b\}$.

The reason that we tend to think loosely of Turing machines as "reading the input" for the most convenient alphabet, e.g., a representation of the vertices of a graph, without necessarily specifying the representation explicitly, is that provided that if the alphabet is finite we can encode it with bits and effectively embed a subroutine of finite size in the Turing machine that deals with encoding, decoding, and writing symbols using our encoding.

In short, as long as we consider asymptotic running times we may use the most natural symbols as long as they are finitely many and use operations on them as the unit cost. We must still take into account high level representation, e.g., how a graph is represented.

A.1 Multi-tape Turing Machines

To introduce oracles it is natural to first introduce multi-tape Turing machines. They simply have a constant k number of infinite tapes instead of only one and may be thought of as multiple Turing machines working in unison on a joint state.

Definition 10 (Multi-tape Turing Machine)**.** A *k-tape Turing machine* is a Turing machine except that the transition function has the form

$$\delta : Q \setminus A \times \Gamma^k \to Q \times (\Gamma \times \{S, L, R\})^k$$

where S indicates that the tape head at a given position does not move.

For a single tape Turing machine there is no need for the symbol S, since it is pointless to repeatedly rewrite a given cell (any computation this enables can be embedded in the Turing machine itself), but for multi-tape Turing machines it is quite natural.

The multi-tape Turing machine simply executes as before with state transitions, but in each transition it now reads and writes k cells of its respective tapes and either remains in place, or moves left or right as before separately for each tape.

Theorem 1. A k-tape Turing machine with running time $T(n)$ in its input size n, where k is constant, can be simulated by a single tape Turing machine in time $O\big(T(n)^2\big)$.

This theorem is interesting in both directions. On the one hand it illustrates that the choice of computational does model matter when we care about details, i.e., the Turing machine is not *the* computational model. On the other hand it shows that for polynomial time reductions it does not matter if we allow ourselves a constant number of tapes. There are similar results that relate random access memory (RAM) machines and Turing machines, but in the end we need some sensible mathematical model of computation much like mathematics needs a meta language and axioms.

A.2 Oracle Turing Machines

There are several ways to define the interaction between a Turing machine and an oracle for evaluating a function, but with an additional tape it is easy. We simply dedicate the first tape to be used for writing a query and reading the reply.

Definition 11 (Oracle Turing Machine)**.** An *f-oracle k-tape Turing machine* is a $(k+1)$-tape Turing machine with the following restrictions.

- **Query and response states.** Q contains the states q_{query} and q_{reply}.
- **Write query and read reply.** For every output $(q, (x_i, d_i)_{i\in[k+1]})$ of δ: $d_1 = \{R, S\}$.
- **Query.** For every $\gamma \in \Gamma^k$: $\delta(q_{\text{query}}, \gamma) = (q_{\text{reply}}, \gamma, S^k)$ and (i) the contents x of the first tape is replaced by $f(x)$ and all other cells blanked, and (ii) the position of the first tape is set to the cell containing the first symbol of $f(x)$.

The definition simply says that:

1. We have query and reply states to let the Turing machine query the oracle and know when to read the reply.
2. The first tape is dedicated to input/outputs from the oracle. It can only move to the right on this tape while writing a query or reading a reply.
3. To state a query it must write the complete query on the first tape and perform a query transition to receive the reply which it can then read.

A single-tape Turing machine with an oracle is restricted such that it only gains the ability to write a query, request the reply, and read the reply.

We could define a special part of a single tape Turing machine to be used for queries and responses, e.g., since the tape infinite in only one direction is enough to simulate a tape which is infinite in both directions we could have defined one part to be used for our oracle interaction.

Intuition 8. The point of this is that such details do not matter for our purposes more than one cares if a device of a PC is connected using DMA or over the bus of the motherboard, or how parameters of a subroutine call are ordered on the stack by a compiler.

What *does* matter is that the oracle machine must formulate and explicitly write down the query and read the response. The analysis of the running time of the algorithm must take this into account.

A useful way to think about an oracle Turing machine is as a program with access to a subroutine which responds immediately, i.e., we do not include the running time of the subroutine in the running time of the algorithm. This is depicted in Figure A.2.

This corresponds to our intuition about solving an instance of a problem somehow by repeatedly solving instances of another problem, where we do not count the time spent to solve the other problem.

A.3 Probabilistic Turing Machines

A probabilistic Turing machine is easy to define starting from a multi-tape Turing machine.

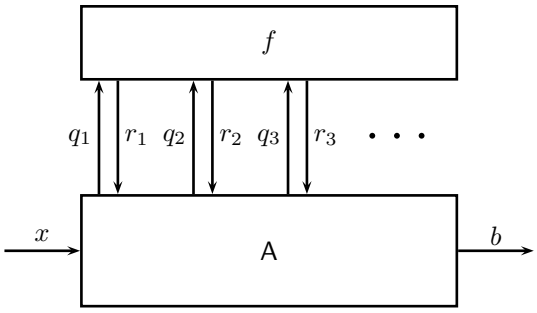


Figure 4: The algorithm A repeatedly: hands a query q_i to the oracle $f(\cdot)$, which immediately responds with a reply $r_i = f(q_i)$, for $i = 1, 2, 3, \ldots,$. In general we do not know how many queries A asks or how it uses the replies in its computation, except that the number of queries is bounded by its running time.

Definition 12 (Probabilistic Turing Machine)**.** An *probabilistic k-tape Turing machine* with randomness $C \subset \Gamma$ is a $k+1$-tape Turing machine where

- **Read random symbol.** For every output $(q, (x_i, d_i)_{i\in[k+1]})$ of δ: $d_1 \in \{R, S\}$.
- **Initialized with randomness.** At the start of the execution each cell of the first tape is initialized with uniformly and independently chosen symbols from C.

Note that the random tape as a whole is not a *discrete* random variable, but for any algorithm with bounded running time the number of cells accessed is finite which means that it is a discrete random variable.

On the choice of sampled symbols. Usually, we consider the random tape to contain uniformly and independently sampled bits, but what happens if each cell contains one of B symbols, which may be thought of as integers in $[0, B-1]$ instead?

We can read s elements $x_1, \ldots, x_s$ and form an integer $x = \sum_{i=0}^{s-1} B^i x_i$ in $[0, B^s - 1]$. Then we can simply define $y = x \bmod m$ for some m needed by the algorithm. If q is the largest integer such that $qm < B^s$, then the statistical error for a single symbol relative the uniform distribution is bounded by $(B^s - qm)/B^s \leq m/B^s = 2^{\log m - s \log B}$.

In short, if we need t sampled symbols in an algorithm we can by making s large enough make sure that they are statistically close to the uniform distribution at multiplicative cost of n. Thus, in the context of polynomial time reductions for probabilistic reductions, we may assume that we can sample uniformly and independently from any finite interval of the integers. This is enough, e.g., to sample a random vertex or edge in a graph.