6.875: Cryptography and Cryptanalysis **September 17, 2021**

Recitation 2: Complexity Theory and Reductions

Instructor: Vinod Vaikuntanathan **TAs:** Lali Devadas and Sacha Servan-Schreiber

Contents

1 Background concepts and terminology

1.1 Decision problems and decidability

A **decision problem** is an assignment mapping some input space to $\{0, 1\}$. Inputs assigned to 0 are NO-instances, and inputs assigned to 1 are YES-instances (i.e., satisfying instances). We call the set of YES-instances the **language** of the decision problem, and say that an input is either "in the language" or "not in the language".

In words, suppose you're given a puzzle which you're not sure whether or not has a solution. Your job is to only *decide* if the puzzle is solvable or not, without necessarily finding a solution to the puzzle in the case that the answer is YES—a strictly weaker property.

Here are some examples of well-studied decision problems:

Problem	Decision
Tetris	Can you survive a given sequence of blocks?
Chess	Can a player force a win from a given board configuration?
Halting problem	Does a given computer program terminate for a given input?
s-t Shortest Path	Does a given G contain a path from s to t with weight at most d?
Negative Cycle	Does a given G contain a negative weight cycle?
Longest Path	Does a given G contain a *simple* path with weight at least d?
Subset Sum	Does a given set of integers A contain a subset with sum S?

To better reason about problems that are solvable and decidable, we will define a **program** to be a constant-length code segment to solve a problem. That is, a program is a piece of code that produces

Verifier	**Prover**
Hard problem instance I.	*Solution w such that $I(w) = YES$.*
Prove that I is a YES instance. $\longrightarrow$	
	Here is certificate string c. $\longleftarrow$
Check $(I, c) \stackrel{?}{=} YES$ in polynomial time.	

Figure 1: Illustration of NP-verification.

correct output (0 or 1) for every input such that the length of the code is independent of the input size. We refer to a decision problem as **decidable** if there exists a program to solve the problem in finite time. Note that not all problems are decidable! A famous example of a problem that is not decidable is the *halting problem* (deciding if a program will terminate).

1.2 Polynomial (P) and non-deterministic polynomial time (NP)

- **P** is the set of decision problems for which there is an algorithm A such that for every instance I of size n, A on I runs in $\mathsf{poly}(n)$ time and decides I correctly (i.e., outputs "YES"/"NO" correctly).
- **NP** is the set of decision problems for which there is an algorithm V (an efficient *verifier*) that takes as input an instance I of size n, and an efficiently checkable *certificate* bit string c of length $\mathsf{poly}(n)$. Specifically, we require that V always runs in $\mathsf{poly}(n)$ time and outputs "YES" $(I, c) =$ YES, and outputs "NO" if $(I, c) =$ NO. See Figure 1 for an illustration.

R	problems decidable in finite time	`// 'R' comes from recursive languages`
EXP	problems decidable in exponential time $2^{n^{O(1)}}$	`// most problems we think of are here`
P	problems decidable in polynomial time $n^{O(1)}$	`// "efficient" algorithms e.g., sorting numbers`

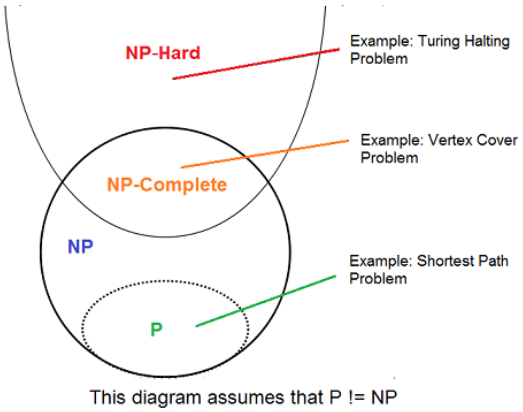


Figure 2: Source: https://www.geeksforgeeks.org/

You can think of the certificate as a proof that I is a YES-instance. If I is actually a NO-instance then no proof should work. Here are some examples of decision problems known to be in **NP**:

Problem	Certificate	Verifier
s-t Shortest Path	A path P from s to t	Adds the weights on P and checks if $\leq d$
Negative Cycle	A cycle C	Adds the weights on C and checks if < 0
Subset Sum	A set of items A'	Checks if $A' \in A$ has sum S
Tetris	Sequence of moves	Checks that the moves allow survival

How does NP fit into our landscape so far?

- **P** $\subseteq$ **NP**: if the problem is in **P**, it is decidable in $\mathsf{poly}(n)$ time so the verifier V can just solve the instance itself and ignore any certificate. In other words, you can always have an "empty" certificate $c = \texttt{null}$ where the verifier ignores c and just checks the instance itself.
- **NP** $\subseteq$ **EXP**: if the problem is in **NP**, it has a $\mathsf{poly}(n)$-time verifier V and at most $2^{n^{O(1)}}$ possible certificates, so an exponential time algorithm can run V on every possible certificate and return YES if V accepts any certificate and NO if V rejects every certificate.
- Does **P** = **NP**? **NP** = **EXP**? These are still open questions.

2 Deterministic reductions

2.1 Introduction

Reductons are a powerful tool in both complexity theory and in cryptography. The idea behind reductions is simple. Suppose you believe that deciding problem A is hard, i.e. takes longer than polynomial time.

Now, you're given a problem B which you're not sure about. B could be an easy or a hard problem. To prove that B is a hard problem, we must *reduce* the difficulty of problem A to the difficulty of problem B. How do we do this? Well, suppose you have a solver $\mathcal{F}$ for B (think of $\mathcal{F}$ as a function built into your computer; you can run it but you cannot modify or change its inputs). If given a valid instance of problem B, then $\mathcal{F}$ will give you a valid solution to B. Note that we make no assumptions on $\mathcal{F}$'s runtime since we're trying to prove a lower-bound on the time it takes to solve instances of problem A!

The key idea is that if you can "transform" an instance of A into an instance of B, then you can use $\mathcal{F}$ to solve A. If this transformation takes polynomial time, then it must be the case that B is at least as hard as A. Why? Because we've shown that if B is decidable in time T, you can decide an instance of A in time T + the runtime of your transformation. Since we believe A is not decidable in polynomial time, B also cannot be decidable in polynomial time.

We will now more formally define several types of reductions.

2.2 Karp reductions

Suppose you don't have a program which decides problem A, but you still want to solve an instance I_A of size n. One way to solve is to convert I_A into an instance I_B of a problem B which you do have a solver for: this is called a **Karp reduction** from problem A to problem B ($A \to B$). The conversion algorithm must

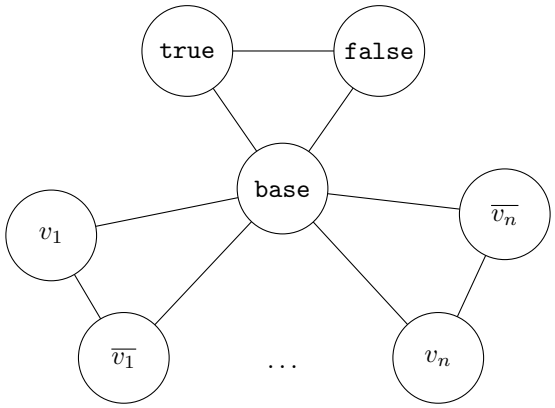
- run in time polynomial in the length of I_A (think about why), and
- convert YES-instances to YES-instances and NO-instances to NO-instances (i.e. I_B is a YES-instance of problem B iff I_A is a YES-instance of problem A).

If we have such a conversion, we can conclude that because B can be used to solve A, B is at least as hard as A ($A \leq B$).

2.2.1 Example: 3SAT → graph 3-colorability

Recall that the 3SAT problem takes a 3-CNF formula as input and asks whether or not the formula is satisfiable. 3SAT is an important NP-complete problem that we often use in reductions. Here we will use a reduction from 3SAT to show that graph 3-colorability is NP-hard [1]. Recall that the graph 3-coloring problem takes an undirected graph as input and asks whether or not it is possible to, using only 3 colors, color every vertex of the graph so that no edge connects two vertices of the same color.

Figure 3: Base graph for our 3SAT → graph 3-colorability reduction.



Example 1: 3SAT → graph 3-colorability.

Given a 3-CNF formula ϕ with m clauses and n variables $x_1, \ldots, x_n$, we want to construct a graph G_ϕ such that G_ϕ is 3-colorable if and only if ϕ is satisfiable (i.e. a YES-instance of 3SAT).
First, we'll construct a graph G which represents a variable assignment, i.e., a mapping $\{x_1, \ldots, x_n\} \to \{\texttt{true, false}\}$ (and is independent of ϕ). We have three nodes, `true, false, base`, connected in a triangle. We also have two nodes for each variable x_i: v_i and $\overline{v_i}$, which are connected to each other and the `base` node in a triangle. See fig. 3.
A 3-coloring of this graph G gives us a variable assignment for $x_1, \ldots, x_n$. How? `true, false, base` will each get different colors. For all i, either v_i matches `true` and $\overline{v_i}$ matches `false`, or vice versa (v_i matches `false` and $\overline{v_i}$ matches `true`). We naturally consider the former an assignment of x_i to `true` and the latter an assignment of x_i to `false`.
Now, we need to add to this graph the constraints of the actual formula ϕ. For every clause $l_i \vee l_j \vee l_k$ (i.e. the OR of three literals, which could be variables or their negation), we will add an OR "gadget" graph to G such that if the new composite graph is 3-colorable, then $l_i \vee l_j \vee l_k$ is satisfiable by some variable assignment (see the illustration above).

First let's consider the easier case of representing just $l_i \vee l_j$. Recall that we have nodes corresponding to l_i and l_j in G already. We want to create a new graph which is 3-colorable if and only if l_i or l_j is `true`. We do this as follows:

1. Add a new node to G corresponding to $(l_i \vee l_j)$. We can require this node to match `true` by connecting it in a triangle with `false, base`. To see why, observe that if $(l_i \vee l_j) =$ `false` then we have two adjacent `false` nodes in the graph, which is not 3-colorable (recall: `false` gets its own color).
2. Add two helper nodes l_i' and l_j' and connect them to $(l_i \vee l_j)$ in a triangle, so one is forced to match `false` (and the other one matches `base`).
3. Connect l_i' to l_i and l_j' to l_j. This forces either l_i or l_j to match `true`, since one is already connected to `base` and is now connected to a helper node which matches `false`.

Now that we have a node corresponding to $(l_i \vee l_j)$, representing $(l_i \vee l_j \vee l_k)$ is quite simple.

1. Disconnect $(l_i \vee l_j)$ from `false, base`, since we no longer require it to match `true`.
2. Follow the steps above to represent $(l_i \vee l_j) \vee l_k$.

(See fig. 4 for $x_1 \vee \neg x_2 \vee x_3$.) We repeat this process for each of the m clauses in ϕ to obtain G_ϕ. Since G originally had $2n + 3$ nodes and each clause adds 6 new nodes, G_ϕ will have $2n + 6m + 3$ nodes.

Exercise 1. Verify that this reduction takes polynomial time.

Exercise 2. Recall that the k-clique problem takes a graph as input and asks whether the graph has a clique, i.e. set of vertices all connected to one another, of size k. Taking inspiration from this logic about what vertices and the edges connecting them should represent, reduce 3SAT to k-clique. *Hint:* Set $k = m$ (the number of clauses in the 3SAT formula).

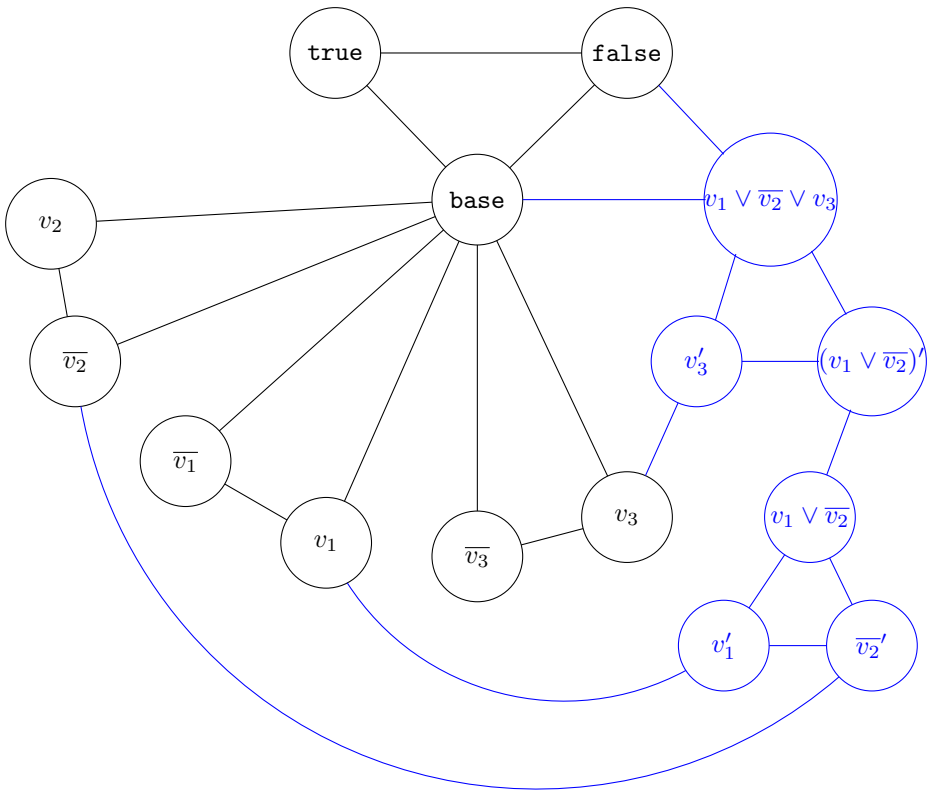
2.3 Cook reductions

We can generalize this notion of reductions by allowing the solver T for problem A to make subroutine calls to a solver S for problem B: this is called a **Cook reduction**. On any instance I_A, T takes a polynomial number of steps, where a "step" can be a standard unit of computation *or* a call to S on some instance I_B (think about how big I_B can be in the length of I_A).

2.3.1 Example: search-LWE → decision-LWE

The {search/decision}-LWE problem (originally introduced by Regev [2]) basically asks to solve a system of noisy linear equations. Formally, $\mathsf{LWE}_{n,m,q,\chi}$, for parameters $n, m, q \in \mathbb{Z}^+$ (we assume q

Figure 4: This shows G_ϕ for $\phi = x_1 \vee \neg x_2 \vee x_3$, with the original graph G (for just the three variables x_1, x_2, x_3) in black and the OR gadget graph highlighted in blue.



is prime) and an error distribution χ, gives as a challenge

$$\left(\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{m \times n}, \mathbf{b} \in \mathbb{Z}_q^m\right). \quad \texttt{// } m \times n \texttt{ matrix } \mathbf{A} \texttt{ and vector } \mathbf{b} \texttt{ of length } m.$$

In the decision version of the problem, either $\mathbf{b} \xleftarrow{R} \mathbb{Z}_q^m$ or $\mathbf{b} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e}$ for $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n, \mathbf{e} \xleftarrow{R} \chi_q^m$, and the adversary has to determine which way $\mathbf{b}$ was generated. In the search version of the problem, we always have $\mathbf{b} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e}$, and asks the adversary to return the *secret* $\mathbf{s}$.

We will show a Cook reduction from the search version of the problem to the decision version of the problem [3]. Essentially, we will show that if we can decide whether we are seeing noisy linear equations or random values, we can also find the secret (i.e. the solution to the noisy equations). This will remind you of the distinguisher-predictor reduction we saw in class for PRG next-bit unpredictability.

Example 2: Search-LWE $\rightarrow$ Decision-LWE.

Given a distinguisher $\mathcal{D}$ for decision-$\mathsf{LWE}_{n,m,q,\chi}$ which runs in time T with distinguishing advantage ϵ, we will build a algorithm $\mathcal{A}$ for search-$\mathsf{LWE}_{n,m',q,\chi}$ where $m' = m \cdot qn \cdot \frac{c \log n}{\epsilon^2}$, which runs in time $T' = T \cdot nq \cdot \frac{c \log n}{\epsilon^2}$ and is correct with probability $1 - \frac{1}{n}$ (c is a constant).
Our approach to solve search-$\mathsf{LWE}_{n,m',q,\chi}$ will be to "guess" the secret, one coordinate at a time. Let $\mathbf{s} = (s_1, \ldots, s_n)$. First we will describe $\mathcal{B}_i$, which on input $(\mathbf{A}, \mathbf{b} \leftarrow \mathbf{As} + \mathbf{e})$, guesses the i^{th} coordinate of $\mathbf{s}$. At a high level, $\mathcal{B}_i$ iterates through guesses in $\mathbb{Z}_q$ and determines if the current value is correct for s_i. It does so by ensuring that (1) when the guess is correct, the distinguisher $\mathcal{D}$ gets a new LWE sample as input and (2) when the guess is incorrect, $\mathcal{D}$ gets a uniformly random input. Because $\mathcal{D}$ succeeds with probability $\frac{1}{2} + \epsilon$, $\mathcal{B}_i$ repeats this process L times, where we set $L = \frac{c \log n}{\epsilon^2}$ for a sufficiently large constant c, and takes the majority output.

$\mathcal{B}_i(\mathbf{A}_\ell, \mathbf{b}_\ell)$

1 : **for** $0 \le j < q$ **do**
2 : $\quad \mathsf{guess}_i \leftarrow j$
3 : $\quad$ **for** $1 \le \ell \le L$ **do**
4 : $\quad\quad$ choose a fresh block $\mathbf{A}_\ell \in \mathbb{Z}_q^{m \times n}, \mathbf{b}_\ell \in \mathbb{Z}_q^m$
5 : $\quad\quad$ $\mathbf{c}_\ell \xleftarrow{R} \mathbb{Z}_q^m$, $\mathbf{C}_\ell \in \mathbb{Z}_q^{m \times n}$ is all zeroes except the i^{th} column is $\mathbf{c}_\ell$
6 : $\quad\quad$ $\mathbf{A}'_\ell \leftarrow \mathbf{A}_\ell + \mathbf{C}_\ell$, $\mathbf{b}'_\ell \leftarrow \mathbf{b}_\ell + \mathsf{guess}_i \cdot \mathbf{c}_\ell$
7 : $\quad\quad$ $d_\ell \leftarrow \mathcal{D}(\mathbf{A}'_\ell, \mathbf{b}'_\ell)$
8 : $\quad$ **endfor**
9 : $\quad$ **if** majority$(d_1, \ldots, d_L) = 1$ **then return** guess_i
10 : $\quad$ **endfor**

$\mathcal{A}$ runs $\mathcal{B}_i$ for $i \in [n]$ (feeding in new rows of $\mathbf{A}$ and $\mathbf{b}$ as needed) and returns $(\mathsf{guess}_1, \ldots, \mathsf{guess}_n)$.

Correctness. If the guess guess_i is correct, then $(\mathbf{A}'_\ell, \mathbf{b}'_\ell)$ is a fresh LWE sample, since

$$\begin{aligned}\mathbf{b}'_\ell &= \mathbf{b}_\ell + \mathsf{guess}_i \cdot \mathbf{c}_\ell = \mathbf{b}_\ell + s_i \cdot \mathbf{c}_\ell = (\mathbf{A}_\ell \mathbf{s} + \mathbf{e}_\ell) + s_i \cdot \mathbf{c}_\ell \\ &= (\mathbf{A}_\ell \mathbf{s} + s_i \cdot \mathbf{c}_\ell) + \mathbf{e}_\ell = (\mathbf{A}_\ell + \mathbf{C}_\ell)\,\mathbf{s} + \mathbf{e}_\ell = \mathbf{A}'_\ell \mathbf{s} + \mathbf{e}_\ell\end{aligned}$$

On the other hand, if the guess guess_i is incorrect, then $\mathbf{b}'_\ell$ is uniformly random, since

$$\begin{aligned}\mathbf{b}'_\ell &= \mathbf{b}_\ell + \mathsf{guess}_i \cdot \mathbf{c}_\ell = (\mathbf{A}_\ell \mathbf{s} + \mathbf{e}_\ell) + \mathsf{guess}_i \cdot \mathbf{c}_\ell \\ &= (\mathbf{A}_\ell \mathbf{s} + \mathsf{guess}_i \cdot \mathbf{c}_\ell) + \mathbf{e}_\ell = \mathbf{A}'_\ell \mathbf{s} + \mathbf{e}_\ell + (\mathsf{guess}_i - s_i) \cdot \mathbf{c}_\ell\end{aligned}$$

and the term $(\mathsf{guess}_i - s_i) \cdot \mathbf{c}_\ell$ is random and independent of $\mathbf{A}'_\ell \mathbf{s} + \mathbf{e}_\ell$ because (1) $(\mathsf{guess}_i - s_i)$ is nonzero, (2) q is prime, and (3) $\mathbf{c}_\ell$ is random and independent of $\mathbf{A}_\ell, \mathbf{s}, \mathbf{e}_\ell$. It follows that $\mathcal{D}$ will output 1 with probability at least $\frac{1}{2} + \epsilon$, in the case that $\mathsf{guess}_i = s_i$. Since we run $\mathcal{D}$ $L = \frac{c \log n}{\epsilon^2}$ times, if we set $c = 2(1 + 2\epsilon)$ it follows from a Chernoff bound that if $\mathsf{guess}_i = s_i$, then we have majority$(d_1, \ldots, d_L) \neq 1$ with probability at most $\frac{1}{n^2}$. Hence, by a union bound, with probability at least $1 - \frac{1}{n}$, for all $i \in [n]$ $\mathcal{B}_i$ guesses s_i correctly, i.e. $\mathcal{A}$ returns the correct answer.

Runtime. The runtime of single iteration of the inner **for** loop at line 3 is dominated by the runtime of $\mathcal{D}$, i.e. T. $\mathcal{B}_i$ runs L iterations of the inner loop for each of the (at most) q iterations of the outer **for** loop at line 1, and we run $\mathcal{B}_i$ for every $i \in [n]$. Thus, $\mathcal{A}$'s total runtime is $T' = T \cdot nq \cdot \frac{c \log n}{\epsilon^2}$.

Number of rows. Every call to $\mathcal{D}$ consumes m rows of $\mathbf{A}$ and $\mathbf{b}$, so by the above analysis, we need $m' = m \cdot nq \cdot \frac{c \log n}{\epsilon^2}$ rows of $\mathbf{A}$ and $\mathbf{b}$.

Exercise 3. Reduce search-3SAT to decision-3SAT: that is, given an algorithm $\mathcal{D}$ which can tell whether or not it's been given a satisfiable or unsatisfiable 3-CNF formula with non-negligible

advantage, build an algorithm $\mathcal{A}$ which can find a satisfying assignment for a 3-CNF formula.

3 Probabilistic reductions

3.1 Introduction

Both Cook and Karp reductions are *deterministic*; the reduction does not use any randomness, and will always act the same way if you feed it the same input. A more general class of reductions, which we will rely on heavily in this course, are **probabilistic reductions**. That is, the reduction has its own random coins, and might act differently if you feed it the same input twice. It might even succeed in solving the problem once and fail at solving the problem once. In Section 3.2 we will consider a special kind of probabilistic reduction.

3.2 Random self-reducibility

Random self-reducibility of a problem states that no distribution over instances of the problem is any harder than to solve with non-negligible advantage than the uniform distribution over instances of the problem (here "harder" means "taking more computational time"). This is a reduction between the *same* problem (hence the "self"), however, we reduce from a worst-case instance to an average-case instance (implying that a random instance is just as difficult to solve as the worst-case instance).

3.2.1 Example: Quadratic-Residue (QR) problem

In this example we consider the problem of finding **quadratic residues** mod N. A **quadratic residue** is a number $y \pmod N$ such that there exists a number $z \pmod N$ where

$$y = z^2 \pmod N.$$

It is widely believed that finding z given y (i.e., finding the square root of $y \pmod N$) is computationally intractable without knowing the *prime factorization* of N. The question we will answer in this example is: does this apply to *the average* y or just a few worst-case instances? In this example, our reduction will be *probabilistic* and occasionally fail. However, we will show that the probability of failure is *negligible*.

Example 3: Random self-reducibility for the Quadratic-Residue problem

Suppose we have a solver $\mathcal{A}$ that when given random $y \in \mathbb{Z}_N$ as input, outputs $z \in \mathbb{Z}_N$ such that $y = z^2 \pmod N$ with some probability p. We will build a solver $\mathcal{B}$ that given a *worst-case* input y^* (the "hardest" instance of the QR problem, which we don't know ahead of time) will somehow use $\mathcal{A}$ to recover a correct solution for y^* with probability close to p. That is, $\mathcal{B}$ will come up with an input that is distributed identically to a *random* instance of the QR problem so as to use $\mathcal{A}$ and thus recover a solution for y^*.

The reduction: We construct $\mathcal{B}$ as follows.

$\mathcal{B}$

1 : $r \xleftarrow{R} \mathbb{Z}_N$ `// a random number` $0 \ldots N-1$
2 : $y' \leftarrow y^* \cdot (r^2) \pmod N$
3 : $z' \leftarrow \mathcal{A}(y')$
4 : $z \leftarrow z' \cdot r^{-1} \pmod N$

Analysis. We have that y' is a random quadratic residue mod N. This holds because:

$$ab^2 = c^2,$$

for some c, i.e., multiplying by a random quadratic residue results in a new (random) quadratic residue. $\mathcal{A}$ works on uniformly random inputs, which in this case means it is expecting uniformly random quadratic residues. As such, the input given to $\mathcal{A}$ by $\mathcal{B}$ is distributed correctly. *Observe that the reduction doesn't work if we feed $y' := y^* \cdot r$ to $\mathcal{A}$; try to understand why that is.*

We now analyze the success probability of $\mathcal{B}$. The probability that b is a valid solution is contingent on r being *invertible* modulo N. When is r invertible modulo N? Some simple number theory tells us that r is invertible if and only if $\mathsf{GCD}(r, N) = 1$ (i.e., r is not a factor of N). Assuming that $N = (q_1 q_2)$ is a composite of two λ-bit primes q_1 and q_2, then the probability that $\mathcal{B}$ succeeds is

$$\begin{aligned}
\Pr[\mathcal{A} \text{ succeeds} \wedge r \text{ is invertible}] &= \underbrace{p}_{\mathcal{A} \text{ succeeds}} \cdot \underbrace{\left(\frac{\phi(N)}{N}\right)}_{r \text{ is invertible}} \\
&= p\left(\frac{(q_1-1)(q_2-1)}{N}\right) \\
&= p\left(\frac{N - q_1 - q_2 + 1}{N}\right) \\
&= p\left(1 - \frac{q_1 + q_2 - 1}{N}\right) \\
&\approx p - p\left(\frac{2^{\lambda+1}}{2^{2\lambda}}\right) \\
&= p - p\left(\frac{1}{2^{\lambda-1}}\right) = p - \mathsf{negl}(\lambda).
\end{aligned}$$

We've just shown that if $\mathcal{A}$ exists for any random instance of the QR problem, then $\mathcal{B}$ exists to solve **any** instance of the QR problem. We can therefore conclude that QR is random self-reducible. Observe that we make no restrictions on the *runtime* of $\mathcal{A}$, though we require the reduction itself to be polynomial time and succeed with some "good" probability.

Exercise 4. Show that the reduction works for N consisting of k prime factors (for some constant k).

Exercise 5. Show that it is sufficient for $\mathcal{B}$ to succeed with any (non-negligible) probability $q \leq p$ for the reduction to work. That is, we do not require $\mathcal{B}$ to succeed with probability that is negligibly close to p.

References

[1] Igor Potapov. 3-coloring is np-complete. https://cgi.csc.liv.ac.uk/~igor/COMP309/, 2021 (accessed 2021-09-15).

[2] Oded Regev. The learning with errors problem. *Invited survey in CCC*, 7(30):11, 2010.

[3] Vinod Vaikuntanathan. Lattices, learning with errors and post-quantum cryptography. https://people.csail.mit.edu/vinodv/CS294/lecturenotes.pdf, 2020.