4 Basing Cryptography on Intractable Computations

John Nash was a mathematician who earned the 1994 Nobel Prize in Economics for his work in game theory. His life story was made into a successful movie, *A Beautiful Mind*.

In 1955, Nash was in correspondence with the United States National Security Agency (NSA),[1] discussing new methods of encryption that he had devised. In these letters, he also proposes some general principles of cryptography (bold highlighting not in the original):

> *. . . in principle the enemy needs very little information to begin to break down the process. Essentially, as soon as λ bits*[2] *of enciphered message have been transmitted the key is about determined. This is no security, for a practical key should not be too long.* ***But this does not consider how easy or difficult it is for the enemy to make the computation determining the key. If this computation, although possible in principle, were sufficiently long at best then the process could still be secure in a practical sense.***

Nash is saying something quite profound: **it doesn't really matter whether attacks are *impossible*, only whether attacks are *computationally infeasible*.** If his letters hadn't been kept classified until 2012, they might have accelerated the development of "modern" cryptography, in which security is based on intractable computations. As it stands, he was decades ahead of his time in identifying one of the most important concepts in modern cryptography.

4.1 What Qualifies as a "Computationally Infeasible" Attack?

Schemes like one-time pad cannot be broken, even by an attacker that performs a **brute-force** attack, trying all possible keys (see Exercise 1.5). However, all future schemes that we will see can indeed be broken by such an attack. Nash is quick to point out that, for a scheme with λ-bit keys:

> *The most direct computation procedure would be for the enemy to try all 2^λ possible keys, one by one. Obviously this is easily made impractical for the enemy by simply choosing λ large enough.*

---

[1]The original letters, handwritten by Nash, are available at: https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/nash-letters/nash_letters1.pdf.

[2]Nash originally used r to denote the length of the key, in bits. In all of the excerpts quoted in this chapter, I have translated his mathematical expressions into our notation (λ).

© Copyright Mike Rosulek. Creative Commons BY-NC-SA 4.0. Latest version at joyofcryptography.com.

We call λ the **security parameter** of the scheme. It is like a knob that allows the user to tune the security to any desired level. Increasing λ makes the difficulty of a brute-force attack grow exponentially fast. Ideally, when using λ-bit keys, *every* attack (not just a brute-force attack) will have difficulty roughy 2^λ. However, sometimes faster attacks are inevitable. Later in this chapter, we will see why many schemes with λ-bit keys have attacks that cost only $2^{\lambda/2}$. It is common to see a scheme described as having ***n*-bit security** if the best known attack requires 2^n steps.

Just how impractical is a brute-force computation on a 64-bit key? A 128-bit key? Huge numbers like 2^{64} and 2^{128} are hard to grasp at an intuitive level.

Example *It can be helpful to think of the cost of a computation in terms of monetary value, and a convenient way to assign such monetary costs is to use the pricing model of a cloud computing provider. Below, I have calculated roughly how much a computation involving 2^λ CPU cycles would cost on Amazon EC2, for various choices of λ.*[3]

clock cycles	*approx cost*	*reference*
2^{50}	*\$3.50*	*cup of coffee*
2^{55}	*\$100*	*decent tickets to a Portland Trailblazers game*
2^{65}	*\$130,000*	*median home price in Oshkosh, WI*
2^{75}	*\$130 million*	*budget of one of the Harry Potter movies*
2^{85}	*\$140 billion*	*GDP of Hungary*
2^{92}	*\$20 trillion*	*GDP of the United States*
2^{99}	*\$2 quadrillion*	*all of human economic activity since 300,000 BC*[4]
2^{128}	*really a lot*	*a billion human civilizations' worth of effort*

Remember, this table only shows the cost to perform 2^λ clock cycles. *A brute-force attack checking 2^λ keys would take many more cycles than that! But, as a disclaimer, these numbers reflect only the* retail *cost of performing a computation, on fairly standard* general-purpose hardware. *A government organization would be capable of manufacturing special-purpose hardware that would significantly reduce the computation's cost. The exercises explore some of these issues, as well as non-financial ways of conceptualizing the cost of huge computations.*

Example *In 2017, the first collision in the SHA-1 hash function was found (we will discuss hash functions later in the course). The attack involved evaluating the SHA-1 function 2^{63} times on a cluster of GPUs. An article in* Ars Technica[5] *estimates the monetary cost of the attack as follows:*

> *Had the researchers performed their attack on Amazon's Web Services platform, it would have cost \$560,000 at normal pricing. Had the researchers been patient and waited to run their attack during off-peak hours, the same collision would have cost \$110,000.*

[3]As of October 2018, the cheapest class of CPU that is suitable for an intensive computation is the m5.large, which is a 2.5 GHz CPU. Such a CPU performs 2^{43} clock cycles per hour. The cheapest rate on EC2 for this CPU is 0.044 USD per hour (3-year reserved instances, all costs paid upfront). All in all, the cost for a single clock cycle (rounding down) is 2^{-48} USD.

[4]I found some estimates (https://en.wikipedia.org/wiki/Gross_world_product) of the gross world product (like the GDP but for the entire world) throughout human history, and summed them up for every year.

[5]https://arstechnica.com/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-de

Asymptotic Running Time

It is instructive to think about the monetary cost of an enormous computation, but it doesn't necessarily help us draw the line between "feasible" attacks (which we want to protect against) and "infeasible" ones (which we agreed we don't need to care about). We need to be able to draw such a line in order to make security definitions that say "only feasible attacks are ruled out."

Once again, John Nash thought about this question. He suggested to consider the **asymptotic** cost of an attack — how does the cost of a computation scale as the security parameter λ goes to infinity?

> *So a logical way to classify enciphering processes is by* ***the way in which the computation length for the computation of the key increases with increasing length of the key. This is at best exponential*** *and at worst probably a relatively small power of λ, $a \cdot \lambda^2$ or $a \cdot \lambda^3$, as in substitution ciphers.*

Nash highlights the importance of attacks that run in polynomial time:

Definition 4.1 *A program runs in* ***polynomial time*** *if there exists a constant $c > 0$ such that for all sufficiently long input strings x, the program stops after no more than $O(|x|^c)$ steps.*

Polynomial-time algorithms scale reasonably well (especially when the exponent is small), but exponential-time algorithms don't. It is probably no surprise to modern readers to see "polynomial-time" as a synonym for "efficient." However, it's worth pointing out that, again, Nash is years ahead of his time relative to the field of computer science.

In the context of cryptography, our goal will be to ensure that no polynomial-time attack can successfully break security. We will not worry about attacks like brute-force that require exponential time.

Polynomial time is not a perfect match to what we mean when we informally talk about "efficient" algorithms. Algorithms with running time $\Theta(n^{1000})$ are technically polynomial-time, while those with running time $\Theta(n^{\log \log \log n})$ aren't. Despite that, polynomial-time is extremely useful because of the following **closure property**: repeating a polynomial-time process a polynomial number of times results in a polynomial-time process overall.

Potential Pitfall: Numerical Algorithms

When we study public-key cryptography, we will discuss algorithms that operate on very large numbers (e.g., thousands of bits long). You must remember that representing the number N on a computer requires only $\sim \log_2 N$ bits. This means that $\log_2 N$, rather than N, is our security parameter! We will therefore be interested in whether certain operations on the number N run in polynomial-time as a function of $\log_2 N$, rather than in N. Keep in mind that the difference between running time $O(\log N)$ and $O(N)$ is the difference between writing down a number and counting to the number.

For reference, here are some numerical operations that we will be using later in the class, and their known efficiencies:

Efficient algorithm known:	**No known efficient algorithm:**
Computing GCDs	Factoring integers
Arithmetic mod N	Computing $\phi(N)$ given N
Inverses mod N	Discrete logarithm
Exponentiation mod N	Square roots mod composite N

Again, "efficient" means polynomial-time. Furthermore, we only consider polynomial-time algorithms that run on standard, *classical* computers. In fact, all of the problems in the right-hand column *do* have known polynomial-time algorithms on *quantum* computers.

4.2 What Qualifies as a "Negligible" Success Probability?

It is not enough to consider only the running time of an attack. For example, consider an attacker who just tries to guess a victim's secret key, making a single guess. This attack is extremely cheap, but it still has a nonzero chance of breaking security!

In addition to an attack's running time, we also need to consider its success probability. We don't want to worry about attacks that are as expensive as a brute-force attack, and we don't want to worry about attacks whose success probability is as low as a blind-guess attack.

An attack with success probability 2^{-128} should not really count as an attack, but an attack with success probability 1/2 should. Somewhere in between 2^{-128} and 2^{-1} we need to find a reasonable place to draw a line.

Example *Now we are dealing with extremely tiny probabilities that can be hard to visualize. Again, it can be helpful to conceptualize these probabilities with a more familiar reference:*

probability	*equivalent*
2^{-10}	*full house in 5-card poker*
2^{-20}	*royal flush in 5-card poker*
2^{-28}	*you win this week's Powerball jackpot*
2^{-40}	*royal flush in 2 consecutive poker games*
2^{-60}	*the next meteorite that hits Earth lands in this square* $\longrightarrow$

As before, it is not clear exactly where to draw the line between "reasonable" and "unreasonable" success probability for an attack. Just like we did with polynomial running time, we can also use an **asymptotic** approach to define when a probability is negligibly small. Just as "polynomial time" considers how fast an algorithm's running time approaches infinity as its input grows, we can also consider how fast a success probability approaches zero as the security parameter grows.

In a scheme with λ-bit keys, a blind-guessing attack succeeds with probability $1/2^\lambda$. Now what about an attacker who makes 2 blind guesses, or λ guesses, or λ^{42} guesses? Such an attacker would still run in polynomial time, and has success probability $2/2^\lambda$, $\lambda/2^\lambda$, or $\lambda^{42}/2^\lambda$. However, no matter what polynomial you put in the numerator, the probability still goes to zero. Indeed, $1/2^\lambda$ **approaches zero so fast that no polynomial can "rescue" it**; or, in other words, it approaches zero faster than 1 over any polynomial. This idea leads to our formal definition:

Definition 4.2 (Negligible) *A function f is **negligible** if, for every polynomial p, we have $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.*

In other words, a negligible function approaches zero so fast that you can never catch up when multiplying by a polynomial. This is exactly the property we want from a security guarantee that is supposed to hold against all polynomial-time adversaries. If a polynomial-time attacker succeeds with probability f, then repeating the same attack p independent times would still be an overall polynomial-time attack (if p is a polynomial), and its success probability would be $p \cdot f$.

When you want to check whether a function is negligible, you only have to consider polynomials p of the form $p(\lambda) = \lambda^c$ for some constant c:

Claim 4.3 *If for every integer c, $\lim_{\lambda\to\infty} \lambda^c f(\lambda) = 0$, then f is negligible.*

Proof Suppose f has this property, and take an arbitrary polynomial p. We want to show that $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.

If d is the degree of p, then $\lim_{\lambda\to\infty} \frac{p(\lambda)}{\lambda^{d+1}} = 0$. Therefore,

$$\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = \lim_{\lambda\to\infty}\left[\frac{p(\lambda)}{\lambda^{d+1}}\left(\lambda^{d+1}\cdot f(\lambda)\right)\right] = \left(\lim_{\lambda\to\infty}\frac{p(\lambda)}{\lambda^{d+1}}\right)\left(\lim_{\lambda\to\infty}\lambda^{d+1}\cdot f(\lambda)\right) = 0\cdot 0.$$

The second equality is a valid law for limits since the two limits on the right exist and are not an indeterminate expression like $0\cdot\infty$. The final equality follows from the hypothesis on f. ∎

Example *The function $f(\lambda) = 1/2^\lambda$ is negligible, since for any integer c, we have:*

$$\lim_{\lambda\to\infty} \lambda^c/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)}/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)-\lambda} = 0,$$

since $c\log(\lambda)-\lambda$ approaches $-\infty$ in the limit, for any constant c. Using similar reasoning, one can show that the following functions are also negligible:

$$\frac{1}{2^{\lambda/2}}, \qquad \frac{1}{2^{\sqrt{\lambda}}}, \qquad \frac{1}{2^{\log^2\lambda}}, \qquad \frac{1}{\lambda^{\log\lambda}}.$$

Functions like $1/\lambda^5$ approach zero but not fast enough to be negligible. To see why, we can take polynomial $p(\lambda) = \lambda^6$ and see that the resulting limit does not satisfy the requirement from Definition 4.2:

$$\lim_{\lambda\to\infty} p(\lambda)\frac{1}{\lambda^5} = \lim_{\lambda\to\infty}\lambda = \infty \neq 0$$

In this class, when we see a negligible function, it will typically always be one that is easy to recognize as negligible (just as in an undergraduate algorithms course, you won't really encounter algorithms where it's hard to tell whether the running time is polynomial).

Definition 4.4 ($f \approx g$) *If $f, g : \mathbb{N} \to \mathbb{R}$ are two functions, we write $f \approx g$ to mean that $\left|f(\lambda) - g(\lambda)\right|$ is a negligible function.*

We use the terminology of negligible functions exclusively when discussing probabilities, so the following are common:

$$\begin{aligned}
\Pr[X] \approx 0 \quad &\Leftrightarrow \quad \text{“event } X \text{ almost never happens”} \\
\Pr[Y] \approx 1 \quad &\Leftrightarrow \quad \text{“event } Y \text{ almost always happens”} \\
\Pr[A] \approx \Pr[B] \quad &\Leftrightarrow \quad \text{“events } A \text{ and } B \text{ happen with} \\
&\qquad\ \text{essentially the same probability”}
\end{aligned}$$[6]

Additionally, the $\approx$ symbol is *transitive*:[7] if $\Pr[X] \approx \Pr[Y]$ and $\Pr[Y] \approx \Pr[Z]$, then $\Pr[X] \approx \Pr[Z]$ (perhaps with a slightly larger, but still negligible, difference).

4.3 Indistinguishability

So far we have been writing formal security definitions in terms of interchangeable libraries, which requires that two libraries have *exactly the same* effect on *every* calling program. Going forward, our security definitions will not be quite as demanding. First, we only consider polynomial-time calling programs; second, we don't require the libraries to have exactly the same effect on the calling program, only that the difference in effects is negligible.

Definition 4.5 (Indistinguishable) *Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be two libraries with a common interface. We say that $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are **indistinguishable**, and write $\mathcal{L}_{\text{left}} \approx \mathcal{L}_{\text{right}}$, if for all polynomial-time programs $\mathcal{A}$ that output a single bit,* $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]$.

*We call the quantity $\big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]\big|$ the **advantage** or **bias** of $\mathcal{A}$ in distinguishing $\mathcal{L}_{\text{left}}$ from $\mathcal{L}_{\text{right}}$. Two libraries are therefore indistinguishable if all polynomial-time calling programs have negligible advantage in distinguishing them.*

From the properties of the "$\approx$" symbol, we can see that indistinguishability of libraries is also transitive, which allows us to carry out hybrid proofs of security in the same way as before.

Example *Here is a very simple example of two indistinguishable libraries:*

$\mathcal{L}_{\text{left}}$
PREDICT(x):
$s \leftarrow \{0,1\}^{\lambda}$
return $x \overset{?}{=} s$

$\mathcal{L}_{\text{right}}$
PREDICT(x):
return `false`

[6] $\Pr[A] \approx \Pr[B]$ doesn't mean that events A and B almost always happen **together** (when A and B are defined over a common probability space) — imagine A being the event "the coin came up heads" and B being the event "the coin came up tails." These events have the same probability but never happen together. To say that "A and B almost always happen together," you'd have to say something like $\Pr[A \oplus B] \approx 0$, where $A \oplus B$ denotes the event that *exactly one* of A and B happens.

[7] It's only transitive when applied a polynomial number of times. So you can't define a whole series of events X_i, show that $\Pr[X_i] \approx \Pr[X_{i+1}]$, and conclude that $\Pr[X_1] \approx \Pr[X_{2^n}]$. It's rare that we'll encounter this subtlety in this course.

Imagine the calling program trying to predict which string will be chosen when uniformly sampling from $\{0,1\}^\lambda$*. The left library tells the calling program whether its prediction was correct. The right library doesn't even bother sampling a string, it just always says "sorry, your prediction was wrong."*

Here is one obvious strategy (maybe not the best one, we will see) to distinguish these libraries. The calling program $\mathcal{A}_{obvious}$ *calls* PREDICT *many times and outputs 1 if it ever received* `true` *as a response. Since it seems like the argument to* PREDICT *might not have any effect, let's just use the string of all-*`0`*s as argument every time.*

$\mathcal{A}_{obvious}$

do q times:
  if PREDICT(0^λ) = `true`
    return 1
return 0

- $\mathcal{L}_{\text{right}}$ *can never return* `true`*, so* $\Pr[\mathcal{A}_{obvious} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1] = 0$.
- *In* $\mathcal{L}_{\text{left}}$ *each call to* PREDICT *has an independent probability* $1/2^\lambda$ *of returning* `true`*. So* $\Pr[\mathcal{A}_{obvious} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1]$ *is surely non-zero. Actually, the exact probability is a bit cumbersome to write:*

$$\begin{aligned}\Pr[\mathcal{A}_{obvious} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] &= 1 - \Pr[\mathcal{A}_{obvious} \diamond \mathcal{L}_{\text{left}} \Rightarrow 0] \\ &= 1 - \Pr[\textit{all } q \textit{ independent calls to } \text{PREDICT} \textit{ return } \texttt{false}] \\ &= 1 - \left(1 - \frac{1}{2^\lambda}\right)^q\end{aligned}$$

Rather than understand this probability, we can just compute an upper bound for it. Using the union bound, we get:

$$\begin{aligned}\Pr[\mathcal{A}_{obvious} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] &\leqslant \Pr[\textit{first call to } \text{PREDICT} \textit{ returns } \texttt{true}] \\ &\quad + \Pr[\textit{second call to } \text{PREDICT} \textit{ returns } \texttt{true}] + \cdots \\ &= q\frac{1}{2^\lambda}\end{aligned}$$

*This is an overestimate of some probabilities (*e.g.*, if the first call to* PREDICT *returns* `true`*, then the second call isn't made). More fundamentally,* $q/2^\lambda$ *exceeds 1 when* q *is large. But nevertheless,* $\Pr[\mathcal{A}_{obvious} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] \leqslant q/2^\lambda$.

We showed that $\mathcal{A}_{obvious}$ *has non-zero advantage. This is enough to show that* $\mathcal{L}_{\text{left}} \not\approx \mathcal{L}_{\text{right}}$.

We also showed that $\mathcal{A}_{obvious}$ *has advantage at most* $q/2^\lambda$*. Since* $\mathcal{A}_{obvious}$ *runs in polynomial time, it can only make a polynomial number* q *of queries to the library, so* $q/2^\lambda$ *is negligible. However, this is not enough to show that* $\mathcal{L}_{\text{left}} \approx \mathcal{L}_{\text{right}}$ *since it considers only a single calling program. To show that the libraries are indistinguishable, we must show that* ***every*** *calling program's advantage is negligible.*

In a few pages, we will prove that for ***any*** $\mathcal{A}$ *that makes* q *calls to* PREDICT*,*

$$\Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]\Big| \leqslant \frac{q}{2^\lambda}.$$

For any polynomial-time $\mathcal{A}$*, the number* q *of calls to* PREDICT *will be a polynomial in* λ*, making* $q/2^\lambda$ *a negligible function. Hence,* $\mathcal{L}_{\text{left}} \approx \mathcal{L}_{\text{right}}$.

Other Properties

Lemma 4.6 ($\approx$ facts) *If $\mathcal{L}_1 \equiv \mathcal{L}_2$ then $\mathcal{L}_1 \approx \mathcal{L}_2$. Also, if $\mathcal{L}_1 \approx \mathcal{L}_2 \approx \mathcal{L}_3$ then $\mathcal{L}_1 \approx \mathcal{L}_3$.*

Analogous to Lemma 2.11, we also have the following library chaining lemma, which you are asked to prove as an exercise:

Lemma 4.7 (Chaining) *If $\mathcal{L}_{\text{left}} \approx \mathcal{L}_{\text{right}}$ then $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}} \approx \mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$ for any polynomial-time library $\mathcal{L}^*$.*

Bad-Event Lemma

A common situation is when two libraries are expected to execute exactly the same statements, until some rare & exceptional condition happens. In that case, we can bound an attacker's distinguishing advantage by the probability of the exceptional condition.

More formally,

Lemma 4.8 (Bad events) *Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be libraries that each define a variable named 'bad' that is initialized to 0. If $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ have identical code, except for code blocks reachable only when bad $= 1$ (e.g., guarded by an "if bad $= 1$" statement), then*

$$\Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]\Big| \leqslant \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \textit{ sets bad} = 1].$$

★ Proof Fix an arbitrary calling program $\mathcal{A}$. In this proof, we use conditional probabilites[8] to isolate the cases where bad is changed to 1. We define the following events:

- $\mathcal{B}_{\text{left}}$: the event that $\mathcal{A} \diamond \mathcal{L}_{\text{left}}$ sets bad to 1 at some point.
- $\mathcal{B}_{\text{right}}$: the event that $\mathcal{A} \diamond \mathcal{L}_{\text{right}}$ sets bad to 1 at some point.

We also write $\overline{\mathcal{B}_{\text{left}}}$ and $\overline{\mathcal{B}_{\text{right}}}$ to denote the corresponding complement events. From conditional probability, we can write:

$$\begin{aligned}
\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] &= \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \mathcal{B}_{\text{left}}] \Pr[\mathcal{B}_{\text{left}}] \\
&\quad + \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{left}}}] \Pr[\overline{\mathcal{B}_{\text{left}}}] \\
\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1] &= \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \mathcal{B}_{\text{right}}] \Pr[\mathcal{B}_{\text{right}}] \\
&\quad + \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{right}}}] \Pr[\overline{\mathcal{B}_{\text{right}}}]
\end{aligned}$$

Our first observation is that $\Pr[\mathcal{B}_{\text{left}}] = \Pr[\mathcal{B}_{\text{right}}]$. This is because at the time bad is changed to 1 for the *first* time, the library has only been executing instructions that are the same in $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$. In other words, the choice to set bad to 1 is determined by the same sequence of instructions in both libraries, so it occurs with the same probability in both libraries.

As a shorthand notation, we define $p^* \stackrel{\text{def}}{=} \Pr[\mathcal{B}_{\text{left}}] = \Pr[\mathcal{B}_{\text{right}}]$. Then we can write the advantage of $\mathcal{A}$ as:

$$\text{advantage}_{\mathcal{A}} = \Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]\Big|$$

[8] The use of conditional probabilites here is delicate and prone to subtle mistakes. For a discussion of the pitfalls, consult the paper where this lemma first appeared: Mihir Bellare & Phillip Rogaway: "Code-Based Game-Playing Proofs and the Security of Triple Encryption," in Eurocrypt 2006. ia.cr/2004/331

$$
\begin{aligned}
&= \left| \begin{array}{r} \Big(\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \mathcal{B}_{\text{left}}] \cdot p^* + \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{left}}}](1-p^*)\Big) \\ -\Big(\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \mathcal{B}_{\text{right}}] \cdot p^* + \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{right}}}](1-p^*)\Big) \end{array} \right| \\
&= \left| \begin{array}{r} p^*\Big(\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \mathcal{B}_{\text{left}}] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \mathcal{B}_{\text{right}}]\Big) \\ (1-p^*)\Big(\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{left}}}] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{right}}}]\Big) \end{array} \right|
\end{aligned}
$$

In both of the expressions $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{left}}}]$ and $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \overline{\mathcal{B}_{\text{right}}}]$, we are conditioning on bad never being set to 0. In this case, both libraries are executing the same sequence of instructions, so the probabilities are equal (and the difference of the probabilities is zero). Substituting in, we get:

$$\text{advantage}_{\mathcal{A}} = p^* \Big| \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1 \mid \mathcal{B}_{\text{left}}] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1 \mid \mathcal{B}_{\text{right}}] \Big|$$

Intuitively, the proof is confirming the idea that differences can only be noticed between $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ when bad is set to 1 (corresponding to our conditioning on $\mathcal{B}_{\text{left}}$ and $\mathcal{B}_{\text{right}}$).

The quantity within the absolute value is the difference of two probabilities, so the largest it can be is 1. Therefore,

$$\text{advantage}_{\mathcal{A}} \leqslant p^* \stackrel{\text{def}}{=} \Pr[\mathcal{B}_{\text{left}}] = \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \text{ sets bad} = 1].$$

This completes the proof. ■

Example *Consider $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ from the previous example (where the calling program tries to "predict" the result of uniformly sampling a λ-bit string). We can prove that they are indistinguishable with the following sequence of hybrids:*

$\mathcal{L}_{\text{left}}$		$\mathcal{L}_{\text{hyb-L}}$		$\mathcal{L}_{\text{hyb-R}}$		$\mathcal{L}_{\text{right}}$
PREDICT(x): $s \leftarrow \{0,1\}^\lambda$ return $x \stackrel{?}{=} s$	$\equiv$	bad := 0 PREDICT(x): $s \leftarrow \{0,1\}^\lambda$ if $x = s$: bad := 1 return true return false	$\approx$	bad := 0 PREDICT(x): $s \leftarrow \{0,1\}^\lambda$ if $x = s$: bad := 1 return false	$\equiv$	PREDICT(x): return false

Let us justify each of the steps:

- $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{hyb-L}}$: *The only difference is that $\mathcal{L}_{\text{hyb-L}}$ maintains a variable "bad." Since it never actually reads from this variable, the change can have no effect.*

- $\mathcal{L}_{\text{hyb-L}}$ *and* $\mathcal{L}_{\text{hyb-R}}$ *differ only in the highlighted line, which can only be reached when bad = 1. Therefore, from the bad-event lemma:*

$$\Big| \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-R}} \Rightarrow 1] \Big| \leqslant \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \textit{ sets bad} = 1].$$

But $\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}$ only sets bad = 1 if the calling program successfully predicts s in one of the calls to PREDICT. *With q calls to* PREDICT, *the total probability of this happening is at most $q/2^\lambda$, which is negligible when the calling program runs in polynomial time. Hence $\mathcal{L}_{\text{hyb-L}} \approx \mathcal{L}_{\text{hyb-R}}$.*

- $\mathcal{L}_{\text{hyb-R}} \equiv \mathcal{L}_{\text{right}}$: *Similar to above, note how the first 3 lines of* PREDICT *in $\mathcal{L}_{\text{hyb-R}}$ don't actually do anything. The subroutine is going to return* `false` *no matter what. Both libraries have identical behavior.*

Since $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{hyb-L}} \approx \mathcal{L}_{\text{hyb-R}} \equiv \mathcal{L}_{\text{right}}$, this proves that $\mathcal{L}_{\text{left}} \approx \mathcal{L}_{\text{right}}$.

4.4 Birthday Probabilities & Sampling With/out Replacement

In many cryptographic schemes, the users repeatedly choose random strings (*e.g.*, each time they encrypt a message), and security breaks down if the same string is ever chosen twice. Hence, it is important that the probability of a repeated sample is *negligible.* In this section we compute the probability of such events and express our findings in a modular way, as a statement about the indistinguishability of two libraries.

Birthday Probabilities

If q people are in a room, what is the probability that two of them have the same birthday (if we assume that each person's birthday is uniformly chosen from among the possible days in a year)? This question is known as the **birthday problem**, and it is famous because the answer is highly unintuitive to most people.[9]

Let's make the question more general. Imagine taking q independent, uniform samples from a set of N items. What is the probability that the same value gets chosen more than once? In other words, what is the probability that the following program outputs 1?

$\mathcal{B}(q, N)$

```
for i := 1 to q:
  s_i ← {1, ..., N}
  for j := 1 to i − 1:
    if s_i = s_j then return 1
return 0
```

Let's give a name to this probability:

$$\text{BirthdayProb}(q, N) \stackrel{\text{def}}{=} \Pr[\mathcal{B}(q, N) \text{ outputs } 1].$$

It is possible to write an exact formula for this probability:

Lemma 4.9 $\quad$ *BirthdayProb*$(q, N) = 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right)$.

[9] It is sometimes called the "birthday paradox," even though it is not really a paradox. The *actual* birthday paradox is that the "birthday paradox" is not a paradox.

Proof Let us instead compute the probability that $\mathcal{B}$ outputs 0, which will allow us to then solve for the probability that it outputs 1. In order for $\mathcal{B}$ to output 0, it must avoid the early termination conditions in each iteration of the main loop. Therefore:

$$\begin{aligned}
\Pr[\mathcal{B}(q,N) \text{ outputs } 0] &= \Pr[\mathcal{B}(q,N) \text{ doesn't terminate early in iteration } i=1] \\
&\quad \cdot \Pr[\mathcal{B}(q,N) \text{ doesn't terminate early in iteration } i=2] \\
&\quad \vdots \\
&\quad \cdot \Pr[\mathcal{B}(q,N) \text{ doesn't terminate early in iteration } i=q]
\end{aligned}$$

In iteration i of the main loop, there are $i-1$ previously chosen values $s_1,\ldots,s_{i-1}$. The program terminates early if any of these are chosen again as s_i, otherwise it continues to the next iteration. Put differently, there are $i-1$ (out of N) ways to choose s_i that lead to early termination — all other choices of s_i avoid early termination. Since the N possibilities for s_i happen with equal probability:

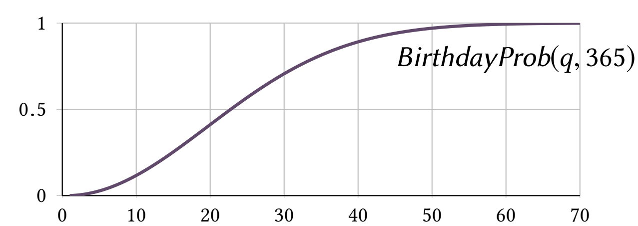
$$\Pr[\mathcal{B}(q,N) \text{ doesn't terminate early in iteration } i] = 1 - \frac{i-1}{N}.$$

Putting everything together:

$$\begin{aligned}
\mathsf{BirthdayProb}(q,N) &= \Pr[\mathcal{B}(q,N) \text{ outputs } 1] \\
&= 1 - \Pr[\mathcal{B}(q,N) \text{ outputs } 0] \\
&= 1 - \left(1-\frac{1}{N}\right)\left(1-\frac{2}{N}\right)\cdots\left(1-\frac{q-1}{N}\right) \\
&= 1 - \prod_{i=1}^{q-1}\left(1-\frac{i}{N}\right)
\end{aligned}$$

This completes the proof. ■

Example *This formula for BirthdayProb(q,N) is not easy to understand at a glance. We can get a better sense of its behavior as a function of q by plotting it. Below is a plot with $N=365$, corresponding to the classic birthday problem:*



With only $q=23$ people the probability of a shared birthday already exceeds 50%. The graph could be extended to the right (all the way to $q=365$), but even at $q=70$ the probability exceeds 99.9%.

Asymptotic Bounds on the Birthday Probability

It will be helpful to have an *asymptotic* formula for how BirthdayProb(q,N) grows as a function of q and N. We are most interested in the case where q is relatively small compared to N (*e.g.*, when q is a polynomial function of λ but N is exponential).

Lemma 4.10 (Birthday Bound) *If $q \leqslant \sqrt{2N}$, then*

$$0.632\frac{q(q-1)}{2N} \leqslant \mathit{BirthdayProb}(q,N) \leqslant \frac{q(q-1)}{2N}.$$

Since the upper and lower bounds differ by only a constant factor, it makes sense to write $\mathit{BirthdayProb}(q,N) = \Theta(q^2/N)$.

Proof We split the proof into two parts.

- To prove the upper bound, we use the fact that when x and y are positive,

$$\begin{aligned}(1-x)(1-y) &= 1-(x+y)+xy \\ &\geqslant 1-(x+y).\end{aligned}$$

More generally, when all terms x_i are positive, $\prod_i(1-x_i) \geqslant 1-\sum_i x_i$. Hence,

$$1-\prod_i(1-x_i) \leqslant 1-(1-\sum_i x_i) = \sum_i x_i.$$

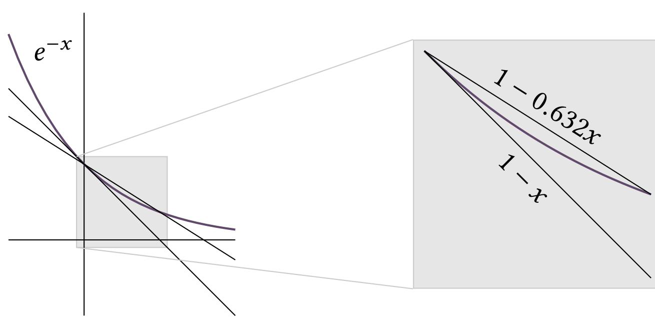
Applying that fact,

$$\text{BirthdayProb}(q,N) \stackrel{\text{def}}{=} 1-\prod_{i=1}^{q-1}\left(1-\frac{i}{N}\right) \leqslant \sum_{i=1}^{q-1}\frac{i}{N} = \frac{\sum_{i=1}^{q-1} i}{N} = \frac{q(q-1)}{2N}.$$

- To prove the lower bound, we use the fact that when $0 \leqslant x \leqslant 1$,

$$1-x \leqslant e^{-x} \leqslant 1-0.632x.$$

This fact is illustrated below. The significance of 0.632 is that $1-\frac{1}{e} = 0.63212\ldots$



We can use both of these upper and lower bounds on e^{-x} to show the following:

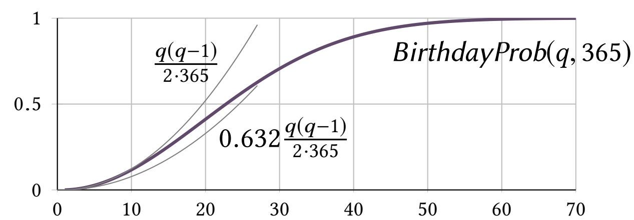
$$\prod_{i=1}^{q-1}\left(1-\frac{i}{N}\right) \leqslant \prod_{i=1}^{q-1} e^{-\frac{i}{N}} = e^{-\sum_{i=1}^{q-1}\frac{i}{N}} = e^{-\frac{q(q-1)}{2N}} \leqslant 1-0.632\frac{q(q-1)}{2N}.$$

With the last inequality we used the fact that $q \leqslant \sqrt{2N}$, and therefore $\frac{q(q-1)}{2N} \leqslant 1$ (this is necessary to apply the inequality $e^{-x} \leqslant 1 - 0.632x$). Hence:

$$\begin{aligned}\mathsf{BirthdayProb}(q,N) &\stackrel{\text{def}}{=} 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right)\\ &\geqslant 1 - \left(1 - 0.632\frac{q(q-1)}{2N}\right) = 0.632\frac{q(q-1)}{2N}.\end{aligned}$$

This completes the proof. ■

Example *Below is a plot of these bounds compared to the actual value of BirthdayProb(q,N) (for $N = 365$):*



As mentioned previously, BirthdayProb(q,N) grows roughly like q^2/N within the range of values we care about (q small relative to N).

The Birthday Problem in Terms of Indistinguishable Libraries

Below are two libraries which will also be useful for future topics.

$\mathcal{L}_{\text{samp-L}}$
SAMP():
$r \leftarrow \{0,1\}^\lambda$
return r

$\mathcal{L}_{\text{samp-R}}$
$R := \emptyset$
SAMP():
$r \leftarrow \{0,1\}^\lambda \setminus R$
$R := R \cup \{r\}$
return r

Both libraries provide a SAMP subroutine that samples a random element of $\{0,1\}^\lambda$. The implementation in $\mathcal{L}_{\text{samp-L}}$ samples uniformly and independently from $\{0,1\}^\lambda$ each time. It samples **with replacement,** so it is possible (although maybe unlikely) for multiple calls to SAMP to return the same value in $\mathcal{L}_{\text{samp-L}}$.

On the other hand, $\mathcal{L}_{\text{samp-R}}$ samples λ-bit strings **without replacement**. It keeps track of a set R, containing all the values it has previously sampled, and avoids choosing them again ("$\{0,1\}^\lambda \setminus R$" is the set of λ-bit strings excluding the ones in R). In this library, SAMP will never output the same value twice.

The "obvious" distinguishing strategy. A natural way (but maybe not the *only* way) to distinguish these two libraries, therefore, would be to call SAMP many times. If you ever see a repeated output, then you must certainly be linked to $\mathcal{L}_{\text{samp-L}}$. After some number of calls to SAMP, if you still don't see any repeated outputs, you might eventually stop and guess that you are linked to $\mathcal{L}_{\text{samp-R}}$.

Let $\mathcal{A}_q$ denote this "obvious" calling program that makes q calls to SAMP and returns 1 if it sees a repeated value. Clearly, the program can never return 1 when it is linked to $\mathcal{L}_{\text{samp-R}}$. On the other hand, when it is linked to $\mathcal{L}_{\text{samp-L}}$, it returns 1 with probability exactly $\text{BirthdayProb}(q, 2^\lambda)$. Therefore, the *advantage* of $\mathcal{A}_q$ is exactly $\text{BirthdayProb}(q, 2^\lambda)$.

This program behaves differently in the presence of these two libraries, therefore they are not *interchangeable*. But are the libraries *indistinguishable?* We have demonstrated a calling program with advantage $\text{BirthdayProb}(q, 2^\lambda)$. We have not specified q exactly, but if $\mathcal{A}_q$ is meant to run in polynomial time (as a function of λ), then q must be a polynomial function of λ. Then the advantage of $\mathcal{A}_q$ is $\text{BirthdayProb}(q, 2^\lambda) = \Theta(q^2/2^\lambda)$, which is *negligible!*

To show that the librares are indistinguishable, we have to show that *all* calling programs have negligible advantage. It is not enough just to show that this *particular* calling program has negligible advantage. Perhaps surprisingly, the "obvious" calling program that we considered is the *best possible* distinguisher!

Lemma 4.11 (Repl. Sampling) *Let $\mathcal{L}_{\text{samp-L}}$ and $\mathcal{L}_{\text{samp-R}}$ be defined as above. Then for all calling programs $\mathcal{A}$ that make q queries to the SAMP subroutine, the advantage of $\mathcal{A}$ in distinguishing the libraries is **at most** BirthdayProb$(q, 2^\lambda)$.*

In particular, when $\mathcal{A}$ is polynomial-time (in λ), q grows as a polynomial in the security parameter. Hence, $\mathcal{A}$ has negligible advantage. Since this is true for all polynomial-time $\mathcal{A}$, we have $\mathcal{L}_{\text{samp-L}} \approx \mathcal{L}_{\text{samp-R}}$.

Proof Consider the following hybrid libraries:

$\mathcal{L}_{\text{hyb-L}}$	$\mathcal{L}_{\text{hyb-R}}$

$R := \emptyset$
bad $:= 0$
 	
SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  if $r \in R$ then:
    bad $:= 1$
 	
  $R := R \cup \{r\}$
  return r
	$R := \emptyset$
	bad $:= 0$
	
	SAMP():
	  $r \leftarrow \{0,1\}^\lambda$
	  if $r \in R$ then:
	    bad $:= 1$
	    $r \leftarrow \{0,1\}^\lambda \setminus R$
	  $R := R \cup \{r\}$
	  return r

First, let us prove some simple observations about these libraries:

$\mathcal{L}_{\text{hyb-L}} \equiv \mathcal{L}_{\text{samp-L}}$: Note that $\mathcal{L}_{\text{hyb-L}}$ simply samples uniformly from $\{0,1\}^\lambda$. The extra R and bad variables in $\mathcal{L}_{\text{hyb-L}}$ don't actually have an effect on its external behavior (they are used only for convenience later in the proof).

$\mathcal{L}_{\text{hyb-R}} \equiv \mathcal{L}_{\text{samp-R}}$: Whereas $\mathcal{L}_{\text{samp-R}}$ avoids repeats by simply sampling from $\{0,1\}^\lambda \setminus R$, this library $\mathcal{L}_{\text{hyb-R}}$ samples r uniformly from $\{0,1\}^\lambda$ and retries if the result happens to be in R. This method is called *rejection sampling*, and it has the same effect[10] as sampling r directly from $\{0,1\}^\lambda \setminus R$.

Conveniently, $\mathcal{L}_{\text{hyb-L}}$ and $\mathcal{L}_{\text{hyb-R}}$ differ only in code that is reachable when bad = 1 (highlighted). So, using Lemma 4.8, we can bound the advantage of the calling program:

$$\begin{aligned}&\Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{samp-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{samp-R}} \Rightarrow 1]\Big|\\ &\qquad = \Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-R}} \Rightarrow 1]\Big|\\ &\qquad \leqslant \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \text{ sets bad} := 1].\end{aligned}$$

Finally, we can observe that $\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}$ sets bad := 1 only in the event that it sees a repeated sample from $\{0,1\}^\lambda$. This happens with probability $\text{BirthdayProb}(q, 2^\lambda)$. ■

Discussion

- Stating the birthday problem in terms of indistinguishable libraries makes it a useful tool in future security proofs. For example, when proving the security of a construction we can replace a uniform sampling step with a sampling-without-replacement step. This change has only a negligible effect, but now the rest of the proof can take advantage of the fact that samples are never repeated.

  Another way to say this is that, when you are thinking about a cryptographic construction, it is "safe to assume" that randomly sampled long strings do not repeat, and behave accordingly.

- However, if a security proof does use the indistinguishability of the birthday libraries, it means that the scheme can likely be broken when a user happens to repeat a uniformly sampled value. Since this becomes inevitable as the number of samples approaches $\sqrt{2^{\lambda+1}} \sim 2^{\lambda/2}$, it means the scheme only offers $\lambda/2$ bits of security. When a scheme has this property, we say that it has **birthday bound security.** It is important to understand when a scheme has this property, since it informs the size of keys that should be chosen in practice.

A Generalization

A calling program can distinguish between the previous libraries if SAMP ever returns the same value twice. In any given call to SAMP, the variable $\mathcal{R}$ denotes the set of "problematic" values that cause the libraries to be distinguished. At any point, $\mathcal{R}$ has only polynomially many values, so the probability of chosing such a problematic one is negligible.

Suppose we considered a different set of values to be problematic. As long as there are only polynomially many problematic values in each call to SAMP, the reasoning behind the proof wouldn't change much. This idea leads to the following generalization, in which the calling program explicitly writes down all of the problematic values:

[10] The two approaches for sampling from $\{0,1\}^\lambda \setminus R$ may have different running times, but our model considers only the input-output behavior of the library.

Lemma 4.12 *The following two libraries are indistinguishable, provided that the argument $\mathcal{R}$ to* SAMP *is passed as an explicit list of items.*

$\mathcal{L}_{\text{samp-L}}$	$\mathcal{L}_{\text{samp-R}}$
$\text{SAMP}(\mathcal{R} \subseteq \{0,1\}^\lambda)$: $r \leftarrow \{0,1\}^\lambda$ return r	$\text{SAMP}(\mathcal{R} \subseteq \{0,1\}^\lambda)$: $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$ return r

Suppose the calling program makes q calls to SAMP, and in the ith call it uses an argument $\mathcal{R}$ with n_i items. Then the advantage of the calling program is at most:

$$1 - \prod_{i=1}^{q} \left(1 - \frac{n_i}{2^\lambda}\right).$$

We can bound this advantage as before. If $\sum_{i=1}^q n_i \leqslant 2^\lambda$, then the advantage is between $0.632 \left(\sum_{i=1}^q n_i\right)/2^\lambda$ and $\left(\sum_{i=1}^q n_i\right)/2^\lambda$. When the calling program runs in polynomial time and must pass $\mathcal{R}$ as an explicit list (*i.e.*, take the time to "write down" the elements of $\mathcal{R}$), $\sum_{i=1}^q n_i$ is a polynomial in the security parameter and the calling program's advantage is negligible.

The birthday scenario corresponds to the special case where $n_i = i - 1$ (in the ith call, $\mathcal{R}$ consists of the $i-1$ results from previous calls to SAMP). In that case, $\sum_{i=1}^q n_i = q(q-1)/2$ and the probabilities collapse to the familiar birthday probabilities.

Exercises

4.1. In Section 4.1 we estimated the monetary cost of large computations, using pricing information from Amazon EC2 cloud computing service. This reflects the cost of doing a huge computation using a *general-purpose CPU*. For long-lived computations, the dominating cost is not the one-time cost of the hardware, but rather the cost of electricity powering the hardware. Because of that, it can be much cheaper to manufacture *special-purpose* hardware. Depending on the nature of the computation, special-purpose hardware can be significantly more energy-efficient.

This is the situation with the Bitcoin cryptocurrency. Mining Bitcoin requires evaluating the SHA-256 cryptographic hash function as many times as possible, as fast as possible. When mining Bitcoin today, the only economically rational choice is to use special-purpose hardware that does nothing except evaluate SHA-256, but is millions (maybe billions) of times more energy efficient than a general-purpose CPU evaluating SHA-256.

(a) The relevant specs for Bitcoin mining hardware are wattage and giga-hashes (or tera-hashes) per second, which can be converted into raw energy required per hash. Search online and find the most energy efficient mining hardware you can (*e.g.*, least joules per hash).

(b) Find the cheapest real-world electricity rates you can, anywhere in the world. Use these to estimate the monetary cost of computing $2^{40}, 2^{50}, \ldots, 2^{120}$ SHA-256 hashes.

(c) Money is not the only way to measure the energy cost of a huge computation. Search online to find out how much carbon dioxide (CO_2) is placed into the atmosphere per unit of electrical energy produced, under a typical distribution of power production methods. Estimate how many tons of CO_2 are produced as a side-effect of computing $2^{40}, 2^{50}, \ldots, 2^{120}$ SHA-256 hashes.

★ (d) Estimate the corresponding CO_2 concentration (parts per million) in the atmosphere as a result of computing $2^{40}, 2^{50}, \ldots, 2^{120}$ SHA-256 hashes. If it is possible without a PhD in climate science, try to estimate the increase in average global temperature caused by these computations.

4.2. Which of the following are negligible functions in λ? Justify your answers.

$$\frac{1}{2^{\lambda/2}} \quad \frac{1}{2^{\log(\lambda^2)}} \quad \frac{1}{\lambda^{\log(\lambda)}} \quad \frac{1}{\lambda^2} \quad \frac{1}{2^{(\log\lambda)^2}} \quad \frac{1}{(\log\lambda)^2} \quad \frac{1}{\lambda^{1/\lambda}} \quad \frac{1}{\sqrt{\lambda}} \quad \frac{1}{2^{\sqrt{\lambda}}}$$

4.3. Suppose f and g are negligible.

(a) Show that $f + g$ is negligible.

(b) Show that $f \cdot g$ is negligible.

(c) Give an example f and g which are both negligible, but where $f(\lambda)/g(\lambda)$ is not negligible.

4.4. Show that when f is negligible, then for every polynomial p, the function $p(\lambda)f(\lambda)$ not only approaches 0, but it is also negligible itself.

Hint: Use the contrapositive. Suppose that $p(\lambda)f(\lambda)$ is non-negligible, where p is a polynomial. Conclude that f must also be non-negligible.

4.5. Prove that the $\approx$ relation is transitive. Let $f, g, h : \mathbb{N} \to \mathbb{R}$ be functions. Using the definition of the $\approx$ relation, prove that if $f \approx g$ and $g \approx h$ then $f \approx h$. You may find it useful to invoke the *triangle inequality*: $|a - c| \leqslant |a - b| + |b - c|$.

4.6. Prove Lemma 4.6.

4.7. Prove Lemma 4.7.

★ 4.8. A *deterministic* program is one that uses no random choices. Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two *deterministic* libraries with a common interface. Show that either $\mathcal{L}_1 \equiv \mathcal{L}_2$, or else $\mathcal{L}_1$ & $\mathcal{L}_2$ can be distinguished with advantage 1.

4.9. Algorithm $\mathcal{B}$ in Section 4.4 has worst-case running time $O(q^2)$. Can you suggest a way to make it run in $O(q \log q)$ time? What about $O(q)$ time?

4.10. Assume that the last 4 digits of student ID numbers are assigned uniformly at this university. In a class of 46 students, what is the **exact** probability that two students have ID numbers with the same last 4 digits?

Compare this exact answer to the upper and lower bounds given by Lemma 4.10.

4.11. Write a program that experimentally estimates the $\textsf{BirthdayProb}(q, N)$ probabilities.

Given q and N, generate q uniformly chosen samples from $\mathbb{Z}_N$, with replacement, and check whether any element was chosen more than once. Repeat this entire process t times to estimate the true probability of $\textsf{BirthdayProb}(q, N)$.

Generate a plot that compares your experimental findings to the theoretical upper/lower bounds of $0.632\frac{q(q-1)}{2^{\lambda+1}}$ and $\frac{q(q-1)}{2^{\lambda+1}}$.

4.12. Suppose you want to enforce password rules so that at least 2^{128} passwords satisfy the rules. How many characters long must the passwords be, in each of these cases?

(a) Passwords consist of lowercase `a` through `z` only.

(b) Passwords consist of lowercase and uppercase letters `a`–`z` and `A`–`Z`.

(c) Passwords consist of lower/uppercase letters and digits `0`–`9`.

(d) Passwords consist of lower/uppercase letters, digits, and any symbol characters that appear on a standard US keyboard (including the space character).