# 4 Basing Cryptography on Limits of Computation

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):

> *We see immediately that in principle the enemy needs very little information to begin to break down the process. Essentially, as soon as $r$ 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.***
>
> *The most direct computation procedure would be for the enemy to try all $2^r$ possible keys, one by one. Obviously this is easily made impractical for the enemy by simply choosing $r$ large enough.*
>
> *In many cruder types of enciphering, particulary those which are not autocoding, such as substitution ciphers [letter for letter, letter pair for letter pair, triple for triple..] shorter means for computing the key are feasible, essentially because the key can be determined piece meal, one substitution at a time.*
>
> *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 $r$, $ar^2$ or $ar^3$, as in substitution ciphers.*
>
> *Now my general conjecture is as follows: For almost all sufficiently complex types of enciphering, especially where the instructions given by different portions of the key interact complexly with each other in the determination of their ultimate effects on the enciphering, the mean key computation length increases exponentially with the length of the key, or in other words, with the information content of the key.*
>
> *The significance of this general conjecture, assuming its truth, is easy to see. It means that* ***it is quite feasible to design ciphers that are effectively un-***

[1] The original letters, handwritten by Nash, are available at: http://www.nsa.gov/public_info/press_room/2012/nash_exhibit.shtml.

[2] Nash is using $r$ to denote the length of the key, in bits. We would use $\lambda$.

> ***breakable.*** *As ciphers become more sophisticated the game of cipher breaking by skilled teams, etc. should become a thing of the past.*

Nash's letters were declassified only in 2012, so they did not directly influence the development of modern cryptography. Still, his letters illustrate the most important idea of "modern" cryptography: that **security can be based on the *computational difficulty* of an attack rather than on the *impossibility* of an attack**. In other words, we are willing to accept that breaking a cryptographic scheme may be possible *in principle*, as long as breaking it would require too much computational effort to be feasible.

We have already discussed one-time-secret encryption and secret sharing. For both of these tasks we were able to achieve a level of security guaranteeing that attacks are *impossible in principle*. But that's essentially the limit of what can be accomplished with such ideal security. Everything else we will see in this class, and every well-known product of cryptography (public-key encryption, hash functions, etc.) has a "modern"-style level of security, which guarantees that attacks are merely *computationally infeasible*, not *impossible*.

## 4.1 Polynomial-Time Computation

Nash's letters also spell out the importance of distinguishing between computations that take a polynomial amount of time and those that take an exponential amount of time.[3] Throughout computer science, polynomial-time is used as a formal definition of "efficient," and exponential-time (and above) is synonymous with "intractible."

In cryptography, it makes a lot of sense to not worry about guaranteeing security in the presence of attackers with unlimited computational resources. Not even the most powerful nation-states can invest $2^{256}$ CPU cycles towards a cryptographic attack.[4] So the modern approach to cryptography (more or less) follows Nash's suggestion, demanding that breaking a scheme requires exponential 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 is not a perfect match to what we mean by "efficient." Polynomial time includes algorithms with running time $\Theta(n^{1000})$, while excluding those with running time $\Theta(n^{\log\log\log n})$. Despite that, it's 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.

[3] Nash was surprisingly ahead of his time here. Polynomial-time wasn't formally proposed as a natural definition for "efficiency" in computation until Alan Cobham, 10 years after Nash's letter was written (Alan Cobham, *The intrinsic computational difficulty of functions*, in *Proc. Logic, Methodology, and Philosophy of Science II*, 1965). Until Nash's letters were declassified, the earliest well-known argument hinting at the importance of polynomial-time was in a letter from Kurt Gödel to John von Neumann. But that letter is not nearly as explicit as Nash's, and was anyway written a year later.

[4] Consider a *Hitchhiker's Guide to the Universe* scenario, in which the entire planet Earth is a supercomputer. Suppose you have a 10GHz computer ($2^{33}$ cycles/sec) for every *atom* on earth ($2^{166}$). Running them all for the age of the earth ($2^{57}$ seconds) gives you a single $2^{256}$ computation.

### Security Parameter

The definition of polynomial-time is *asymptotic*, since it considers the behavior of a computer program *as the size of the inputs grows to infinity.* Cryptographic algorithms often take multiple different inputs to serve various purposes, so to be absolutely clear about our "measuring stick" for polynomial time, we measure the efficiency of cryptographic algorithms (and adversaries!) against something called the **security parameter**, which is the number of bits needed to represent secret keys and/or randomly chosen values used in the scheme. We will typically use $\lambda$ to refer to the security parameter of a scheme.

It's helpful to think of the security parameter as a tuning knob for the cryptographic system. When we dial up this knob, we increase the size of the keys in the system, the size of all associated things like ciphertexts, and the required computation time of the associated algorithms. Most importantly, the amount of effort required by the honest users grows reasonably (as a polynomial function of the security parameter) while the effort required to violate security increases faster than any (polynomial-time) adversary can keep up.

### Potential Pitfall: Numerical Algorithms

The public-key cryptographic algorithms that we will see are based on problems from abstract algebra and number theory. These schemes require users to perform operations on very large numbers. We must remember that representing the number $N$ on a computer requires only $\lceil \log_2(N+1) \rceil$ bits. This means that $\lceil \log_2(N+1) \rceil$, 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 $\lceil \log_2(N+1) \rceil$, 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 $N$ |

By "efficient," we mean polynomial-time. However, all of the problems in the right-hand column *do* have known polynomial-time algorithms on quantum computers.

## 4.2 Negligible Probabilities

In all of the cryptography that we'll see, an adversary can *always* violate security simply by guessing some value that was chosen at random, like a secret key. However, imagine a system that has 1024-bit secret keys chosen uniformly at random. The probability of correctly guessing the key is $2^{-1024}$, which is so low that we can safely ignore it.

We don't worry about "attacks" that have such a ridiculously small success probability. But we would worry about an attack that succeeds with, say, probability 1/2. Somewhere

between $2^{-1024}$ and $2^{-1}$ we need to find a sensible place to draw the line. In the same way that polynomial time formalizes "efficient" running times, we will use an *asymptotic* definition of what is a negligibly small probability.

Consider a scheme with keys that are $\lambda$ bits long. Then a blind-guessing attack may succeed with probability $1/2^\lambda$. Now what about an adversary who makes 2 blind guesses, or $\lambda$ guesses, or $\lambda^{42}$ guesses? Such an adversary would still run in polynomial time, and might succeed in its attack with probability $2/2^\lambda$, $\lambda/2^\lambda$, or $\lambda^{42}/2^\lambda$. The important thing is that, no matter what polynomial you put on top, the probability still goes to zero. Indeed, $1/2^\lambda$ **goes to zero so fast that no polynomial can "rescue" it.** This suggests 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 adversary succeeds with probability $f$, then running the same attack $p$ times would still be an overall polynomial-time attack (if $p$ is a polynomial), and potentially have success probability $p \cdot f$.

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:

$$\begin{aligned} \Pr[X] \approx 0 &\iff \text{“event } X \text{ almost never happens”} \\ \Pr[Y] \approx 1 &\iff \text{“event } Y \text{ almost always happens”} \\ \Pr[A] \approx \Pr[B] &\iff \text{“events } A \text{ and } B \text{ happen with essentially the same probability”}[5] \end{aligned}$$

Additionally, the $\approx$ symbol is *transitive*:[6] 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 $\mathcal{L}_{\text{left}} \approx\!\!\approx \mathcal{L}_{\text{right}}$, 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 $\left|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]\right|$ 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.

Analogous to Lemma 2.7, we also have the following library chaining lemma, which you are asked to prove as an exercise:

[5] $\Pr[A] \approx \Pr[B]$ 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.

[6] 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.

Lemma 4.6 (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 carry out exactly the same steps until some exceptional condition happens. In that case, we can bound an adversary's distinguishing advantage by the probability of the exceptional condition.

More formally, we can state a lemma of Bellare & Rogaway.[7] We present it without proof, since it involves a *syntactic* requirement. Proving it formally would require a formal definition of the syntax/semantics of the pseudocode that we use for these libraries.

Lemma 4.7 (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*

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

## 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 (and we assume that each person's birthday is uniformly chosen from among the possible days in a year), what is the probability that there are two people in the room with the same birthday? This question is known as the **birthday problem**, and it is famous because the answer is highly unintuitive to most people.[8]

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?

$\mathcal{B}$

```
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
```

[7] Mihir Bellare & Phillip Rogaway: "Code-Based Game-Playing Proofs and the Security of Triple Encryption," in Eurocrypt 2006. ia.cr/2004/331

[8] 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.

Let's give a name to this probability:

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

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

Lemma 4.8 $\mathit{BirthdayProb}(q,N) = 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right)$.

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:

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

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. Since there are $N$ choices for $s_i$, each with equal probability:

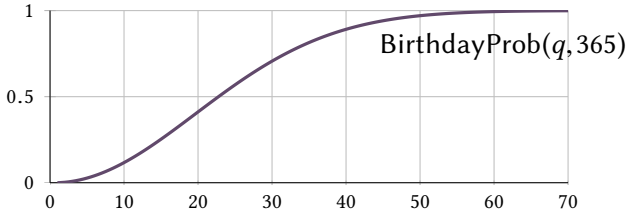
$$\Pr[\mathcal{B} \text{ doesn't terminate early in iteration } i] = 1 - \frac{i-1}{N}.$$

Putting everything together:

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

This completes the proof. ■

This formula for $\mathsf{BirthdayProb}(q,N)$ 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 $\lambda$ but $N$ is exponential).

Lemma 4.9 (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,

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

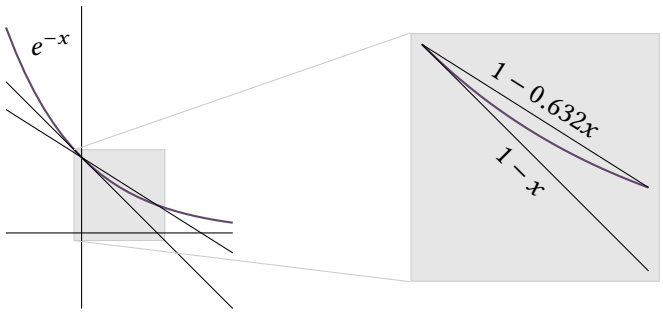
  Applying that fact,

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

- 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:

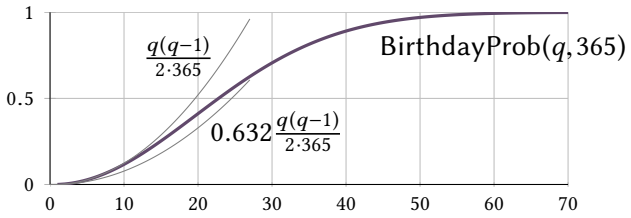
$$\prod_{i=1}^{q-1}\left(1-\frac{i}{N}\right) \leqslant \prod_{i=1}^{q-1} e^{-\frac{i}{N}} = e^{-\sum_{i=1}^{q-1}\frac{i}{N}} = e^{-\frac{q(q-1)}{2N}} \leqslant 1-0.632\frac{q(q-1)}{2N}.$$

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. ■

Below is a plot of these bounds compared to the actual value of $\mathsf{BirthdayProb}(q,N)$ (for $N = 365$):




As mentioned previously, $\mathsf{BirthdayProb}(q,N)$ 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 $\lambda$-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 $\lambda$-bit strings excluding the ones in $R$).[9] In this library, SAMP will never output the same value twice.

[9] In our convention, when a variable like $R$ is initialized outside of any subroutine, it means that the variable is *static* (its value is maintained across different subroutine calls) and *private* (its value is not directly accessible to the calling program).

**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 mean to run in polynomial time (as a function of $\lambda$), then $q$ must be a polynomial function of $\lambda$. 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 4.10 (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** $\text{BirthdayProb}(q, 2^\lambda)$.*

*In particular, when $\mathcal{A}$ is polynomial-time (in $\lambda$), then $q$ 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:

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

```
R := ∅
bad := 0

SAMP():
  r ← {0,1}^λ
  if r ∈ R then:
    bad := 1

  R := R ∪ {r}
  return r
```

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

```
R := ∅
bad := 0

SAMP():
  r ← {0,1}^λ
  if r ∈ R then:
    bad := 1
    r ← {0,1}^λ \ R
  R := R ∪ {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 4.7, we can bound the advantage of the calling program:

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

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 $\mathsf{BirthdayProb}(q, 2^\lambda)$. ■

### Discussion

- Cryptographic schemes are often designed so that users sample repeatedly from $\{0,1\}^\lambda$. Security often breaks down when the same value is sampled twice, and this happens in the range of $\sqrt{2^{\lambda+1}} \sim 2^{\lambda/2}$ steps. The birthday bounds can inform the choice of parameters used in real systems.

- 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.

## Exercises

4.1. Which of the following are negligible functions in $\lambda$? Justify your answers.

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

4.2. 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.

[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.

4.3. 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.

4.4. 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*: $|a - c| \leqslant |a - b| + |b - c|$.

4.5. Prove Lemma 4.6.

★ 4.6. 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.

4.7. For this problem, consider the following two libraries:

$\mathcal{L}_{\text{left}}$

AVOID($v \in \{0,1\}^\lambda$):
  return null

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  return $r$

$\mathcal{L}_{\text{right}}$

$\mathcal{V} := \emptyset$

AVOID($v \in \{0,1\}^\lambda$):
  $\mathcal{V} := \mathcal{V} \cup \{v\}$
  return null

SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{V}$
  return $r$

(a) Prove that the two libraries are indistinguishable. More precisely, show that if an adversary makes $q_1$ number of calls to AVOID and $q_2$ calls to SAMP, then its distinguishing advantage is at most $q_1 q_2 / 2^\lambda$. For a polynomial-time adversary, both $q_1$ and $q_2$ (and hence their product) are polynomial functions of the security parameter, so the advantage is negligible.

(b) Suppose an adversary makes a total of $n_i$ calls to AVOID (with distinct arguments) before making its $i$th call to SAMP, and furthermore the adversary makes $q$ calls to SAMP overall (so that $n_1 \leqslant n_2 \leqslant \cdots \leqslant n_q$). Show that the two libraries *can* be distinguished with advantage at least:

$$0.632 \cdot \frac{\sum_{i=1}^{q} n_i}{2^\lambda}$$

4.8. Prove the following generalization of the results in this chapter:

> Fix a value $x \in \{0,1\}^\lambda$. Then when taking $q$ uniform samples from $\{0,1\}^\lambda$, the probability that there exist two distinct samples **whose** XOR **is** $x$ is $\mathsf{BirthdayProb}(q, 2^\lambda)$.

*Hint:* One way to prove this involves applying Exercise 4.7. Another way involves applying Claim 2.3 to the program $\mathcal{B}$ in Section 4.4.

4.9. 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).