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 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.

> ***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^{100} CPU cycles towards a cryptographic attack. 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.

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.* But cryptographic algorithms often take multiple different inputs to serve various purposes. 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 λ 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 λ bits long. Then a blind guess is correct with probability $1/2^\lambda$. Now what about an adversary who makes 2 guesses, or λ guesses, or λ^{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 $\lambda^{42}/2^\lambda$.

Our starting point here is $1/2^\lambda$, and the nice thing about that probability is that, no matter what polynomial we place on top, we still get a very small probability indeed. This idea gives rise 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 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 $p \cdot f$.

Example

The function $f(\lambda) = 1/2^\lambda$ is negligible, since for any polynomial $p(\lambda) = \lambda^c$, 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.

Similarly, functions like $1/2^{\lambda/2}$, $1/2^{\log^2\lambda}$, and $1/n^{\log n}$ are all negligible. ◆

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 4.3 ($f \approx g$)

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:

$$\begin{aligned} \Pr[X] \approx 0 &\iff \text{"event } X \text{ almost never happens"} \\ \Pr[Y] \approx 1 &\iff \text{"event } Y \text{ almost always happens"} \\ \Pr[A] \approx \Pr[B] &\iff \text{"events } A \text{ and } B \text{ happen with essentially the same probability"}^{4} \end{aligned}$$

Additionally, the $\approx$ symbol is *transitive*:[5] 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 effect is negligible.

Definition 4.4 (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.

And similar to before, we also have a library composition lemma (you are asked to prove this as an exercise):

Lemma 4.5 (Composition)

If $\mathcal{L}_{\text{left}} \approx\!\!\!\approx \mathcal{L}_{\text{right}}$ then $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}} \approx\!\!\!\approx \mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$ 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.[6] 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 4.6 (Bad events)

Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be libraries that each define a variable 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.,

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

[5] 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.

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

guarded by an "if bad = 1" statement), then

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

4.4 Sampling with Replacement & the Birthday Bound

Below is an example of two libraries which are not *interchangeable* (they do have mathematically different behavior), but are *indistinguishable.* These two libraries happen to be convenient for later topics, as well.

Lemma 4.7 (Repl. Sampling) *Define two libraries as below.*

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

```
SAMP():
  r ← {0,1}^λ
  return r
```

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

```
R := ∅

SAMP():
  r ← {0,1}^λ \ R
  R := R ∪ {r}
  return r
```

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 $q(q-1)/2^{\lambda+1}$.*

In particular, when $\mathcal{A}$ is polynomial-time (in λ), its advantage is negligible. Hence, $\mathcal{L}_{\text{samp-L}} \approx \mathcal{L}_{\text{samp-R}}$.

$\mathcal{L}_{\text{samp-L}}$ uniformly samples λ-bit strings with replacement (*i.e.*, allowing for duplicates). In $\mathcal{L}_{\text{samp-R}}$, we've initialized the set R outside of any subroutine, by which we mean that R is a *static* (its value is maintained across different calls to SAMP) and *private* (its value is not directly accessible to the calling program). $\mathcal{L}_{\text{samp-R}}$ uses the set R to keep track of which values have been previously sampled, to sample *without* replacement and ensure that there are no duplicate outputs.

The overall idea here is that the only way to distinguish the two libraries *seems to be* to wait for a repeated output of the SAMP algorithm. But when sampling uniformly from a huge set with 2^λ items, it is very unlikely (but not impossible!) to ever see a repeated value unless one asks for a huge number of samples. For that reason, the two libraries should be indistinguishable. This is an accurate intuition, but it's not a formal proof. Among other things, we need to determine just how unlikely repeated outputs are, and argue that the distinguishing advantage is related to the probability of repeated outputs.

Proof Consider the following hybrid libraries:

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

$R := \emptyset$
bad := 0

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

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

$R := \emptyset$
bad := 0

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  if $r \in R$ then bad := 1
  if bad = 1:
    while $r \in R$:
      $r \leftarrow \{0,1\}^\lambda$
  $R := R \cup \{r\}$
  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 of (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[7] as sampling r directly from $\{0,1\}^\lambda \setminus R$.

We can also see that $\mathcal{L}_{\text{hyb-L}}$ and $\mathcal{L}_{\text{hyb-R}}$ differ only in code that is reachable only when bad = 1 (highlighted). So, using Lemma 4.6, we can bound the advantage of an adversary as:

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

It suffices to show that in $\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}$ the variable bad is set 1 only with negligible probability. It turns out to be easier to reason about the complement event, and argue that bad remains 0 with probability negligibly close to 1.

Suppose $\mathcal{A}$ makes q calls to the SAMP subroutine. Let $r_1, \ldots, r_q$ be the responses of these calls. For variable bad to remain 0, it must be the case that for each i, we have $r_i \notin \{r_1, \ldots, r_{i-1}\}$. Then,

$$\begin{aligned}
&\Pr[\text{bad remains 0 in } \mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}] \\
&\quad = \prod_{i=1}^{q} \Pr\left[r_i \notin \{r_1, \ldots, r_{i-1}\} \,\middle|\, r_1, \ldots, r_{i-1} \text{ all distinct}\right]
\end{aligned}$$

[7] 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.

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

For the inequality we used the convenient fact that when x and y are positive, we have $(1-x)(1-y) = 1-(x+y)+xy \geq 1-(x+y)$. More generally, when all terms x_i are positive, $\prod_i (1-x_i) \geq 1 - \sum_i x_i$.

Summing up, the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}_{\text{samp-L}}$ and $\mathcal{L}_{\text{samp-R}}$ is:

$$\begin{aligned}
&\left|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{samp-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{samp-R}} \Rightarrow 1]\right| \\
&\quad \leq \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \text{ sets bad} = 1] \\
&\quad = 1 - \Pr[\text{bad remains 0 in } \mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}] \\
&\quad \leq 1 - \left(1 - \frac{q(q-1)}{2^{\lambda+1}}\right) \\
&\quad = \frac{q(q-1)}{2^{\lambda+1}}.
\end{aligned}$$

When $\mathcal{A}$ is a polynomial-time distinguisher, q is a polynomial function of λ, so $\mathcal{A}$'s advantage is negligible. This shows that $\mathcal{L}_{\text{samp-L}} \approx \mathcal{L}_{\text{samp-R}}$. ■

Birthday Bound

As part of the previous proof, we showed that the probability of repeating an item while taking q uniform samples from $\{0,1\}^\lambda$ is $O(q^2/2^\lambda)$. This is an upper bound, but we can come up with a lower bound as well:

Lemma 4.8 (Birthday Bound) *When taking q uniform samples from $\{0,1\}^\lambda$, where $q \leq \sqrt{2^{\lambda+1}}$, the probability of encountering a repeated value is at least $0.632\frac{q(q-1)}{2^{\lambda+1}}$. In particular, when $q = \sqrt{2^{\lambda+1}}$, a repeated value is encountered with probability at least 1/2.*

Proof We use the fact that for $x \in [0,1]$, we have $1 - x \leq e^{-x} \leq 1 - (0.632\ldots)x$, where the constant $0.632\ldots$ is $1 - \frac{1}{e}$.

Let $r_1, \ldots, r_q$ be the values uniformly sampled from $\{0,1\}^\lambda$. As above, we *fail* to encounter a repeated value if the r_i's are all distinct. Applying the bounds from above, we have:

$$\begin{aligned}
\Pr[r_1, \ldots, r_q \text{ all distinct}] &= \prod_{i=1}^{q} \left(1 - \frac{i-1}{2^\lambda}\right) \\
&\leq \prod_{i=1}^{q} e^{-\frac{i-1}{2^\lambda}} = e^{-\sum_{i=1}^{q} \frac{i-1}{2^\lambda}} = e^{-\frac{q(q-1)}{2 \cdot 2^\lambda}} \\
&\leq 1 - (0.632\ldots)\frac{q(q-1)}{2 \cdot 2^\lambda}.
\end{aligned}$$

The second inequality follows from the fact that $q(q-1)/2^{\lambda+1} < 1$ by our bound on q. So the probability of a repeated value among the r_i's is

$$1 - \Pr[r_1, \ldots, r_q \text{ all distinct}] \geq 0.632\frac{q(q-1)}{2^{\lambda+1}}.$$

■

More generally, when sampling uniformly from a set of N items, taking $\sqrt{2N}$ samples is enough to ensure a repeated value with probability 0.5. The bound gets its name from considering $q = \sqrt{2 \cdot 365} \approx 27$ people in a room, and assuming that the distribution of birthdays is uniform across the calendar. With probability at least 0.5, two people will share a birthday. This counterintuitive fact is often referred to as the *birthday paradox*, although it's not an actual paradox.[8]

In the context of cryptography, we often design schemes in which the users take many uniform samples from $\{0,1\}^\lambda$. Security often breaks down when the same value is sampled twice, and this happens after about $\sqrt{2^{\lambda+1}} \sim 2^{\lambda/2}$ steps.

Exercises

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

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

4.2. 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.

4.3. 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*: $|a-c| \leq |a-b| + |b-c|$.

4.4. Prove Lemma 4.5.

[8] I think the real paradox is that the "birthday paradox" is not a paradox.