1 Pseudorandom Generators

The crocodile dentist `https://en.wikipedia.org/wiki/Crocodile_Dentist` is a children's game where the players take turns in pressing the teeth of a toy crocodile. One of the teeth is the "hurting" tooth of the crocodile, and the crocodile snaps when one presses it. In the crocodile game, the player takes turn pressing one tooth. The goal of the game is to avoid the hurting tooth. The game is intended for small children, and part of the excitement of the game stems from the fear of the crocodile snapping (although the crocodile is usually well-designed and will not hit the finger...).
For this game to be fun, the crocodile needs to choose a random tooth in each round. But the toy crocodile cannot throw a die and it cannot draw a uniformly random tooth each round. Thus, the crocodile can only sample a tooth that *looks random* although it is actually not. *Pseudorandomness* is a crucial building block of cryptography and refers to something that, indeed, looks random although it isn't. This lecture is going to answer the following questions:

(1) What does it mean for a distribution to *look like* another distribution? (Computational indistinguishability)

(2) What does it mean to *generate pseudorandomness*? (Pseudorandom generators)

(3) *How* can we generate pseudorandomness? (We can generate pseudorandomness based on one-way functions!)

Pseudorandom Generators: Practice and Intuition Random values are a useful tool for system design. For example, when writing items into a database at a random index, then with high probability, there is no other data item at this index stored already, at least when assuming that the index space is large. Solving the same problem deterministically requires state which needs to be synchronized across those writing into the database...

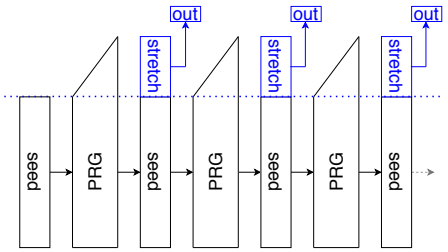

Figure 1: How to use a PRG.

...and this is just one simple example of a case where randomness is convenient. Because randomness is so useful, operating systems and languages tend to provide some form of pseudorandom generators. But what is a reasonable way to generate randomness? Of course, there is a possibility of extracting randomness from one's environment [1], but this means that we need to be sure to always have

[1] There is an entire research field dedicated to randomness extraction, see `https://cacm.acm.org/magazines/2017/1/211100-pure-randomness-extracted-from-two-poor-sources/fulltext` or `https://www.youtube.com/watch?v=B4vBvPr0wow` for a video survey, `https://cs.haifa.ac.il/~ronen/online_papers/ICALPinvited.pdf` for a written introduction on the research level and `https://people.seas.harvard.edu/~salil/pseudorandomness/extractors.pdf` for a textbook introduction to randomness extraction (which might be most suitable for most readers of this document, but I did not want to omit the other ressources).

enough randomness (or "entropy", as some prefer to say) in our environment for the randomized operations we'd like to perform. What if we perform many randomized operations at once? Should we then, say, engage in entropy pool building[2]?
The answer given by pseudorandom generators is that, if you ever had enough randomness, then you can keep generating as much randomness as you want. So, what is a pseudorandom generator? As pseudorandom generator is a function which outputs bitstrings that *look like* uniformly random values although they are actually not uniformly random. As illustrated in Figure 1, a pseudorandom generator is a *length-expanding* function which maps strings (or "seeds") of length λ to longer strings of length $\lambda + s(\lambda)$. That is,

> *A pseudorandom generator (PRG) is a deterministic function which maps a truly random string to a longer, pseudorandom string.*

Since the PRG cannot distinguish whether the input is truly random or just pseudorandom, the following statement is also true:

> *A pseudorandom generator (PRG) is a deterministic function which maps a pseudorandom string to a longer, pseudorandom string.*

However, the second statement follows from the first and from the definition of pseudorandomness, so we will usually only refer to the first formulation. As Figure 1 illustrates, from one application of the PRG, we obtain $s(\lambda)$ many pseudo-random bits for us to use, and the remaining λ bit, the seed, can be fed into the PRG to obtain $s(\lambda)$ further pseudorandom bits and so forth. This way, in $s(\lambda)$ chunks, we can generate as much pseudorandom bits as desired. This is not even very inefficient. We know that pseudorandom generators can be built where each output bit depends only on 5 input bits[3].
To summarize four properties of a PRG:

- A PRG is a *deterministic* function.
- A PRG efficiently computable.
- A PRG is *length-expanding*.
- The output of a PRG *looks like* a uniformly random string of the same length, i.e., drawing a uniformly random x from $\{0,1\}^\lambda$ and computing $y \leftarrow g(x)$ looks to any efficient algorithm like a uniformly random string y of the same length as $g(x)$.

A reflection on domains That this is possible is actually quite amazing. Consider stretch $s(\lambda) = \lambda$. In this case, $|g(x)| = |x| + |x| = 2|x|$. Now, the image of g contains at most 2^λ many values. However, there are $2^{2\lambda}$ many strings of length 2λ. Therefore, the probability that a uniformly random string from $\{0,1\}^{2\lambda}$ is contained in the image of g is exponentially small. Namely, it is at most $\frac{2^\lambda}{2^{2\lambda}} = \frac{1}{2^\lambda}$. And yet, the output of g looks like a uniformly random

[2] In fact, entropy pool building is a research field closely related to the field of randomness extraction, see, e.g., `https://eprint.iacr.org/2019/198` and `https://www.youtube.com/watch?v=iUCJh_liDgA`

[3] See `https://eprint.iacr.org/2018/1162` for a practical study of this question and see `http://www.eng.tau.ac.il/~bennyap/pubs/nc0.pdf` for a complexity-theoretic approach.

string of the same length. That something like this should be possible is very surprising. We will even see, in this lecture, how to build a PRG. It is known that from any one-way function (OWF), we can build a PRG. This is a celebrated theorem by Håstad, Impagliazzo, Levin and Luby [4] and known as the HILL theorem, by the initials of the authors. In this lecture, we are going to see a simpler variant of the HILL theorem. Yet, before turning to the construction, let us properly define what a PRG is.

Definition As for one-wayness, we capture the security of a PRG via an experiment. However, now, instead of using just a single experiment, we use *two* experiments and require that it should be hard for an (efficient) adversary to determine in which experiment it is playing. One of the experiments, the *real* experiment, draws a uniformly random x from $\{0,1\}^\lambda$, computes $y \leftarrow g(x)$ and gives y to the adversary. We denote this *real* experiment by $\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$, see Fig. 2. In turn, the *ideal* experiment $\mathsf{Exp}^{\mathsf{PRG},1}_{s,\mathcal{A}}(1^\lambda)$ draws a uniformly random y from $\{0,1\}^{\lambda+s(\lambda)}$ and gives it to the adversary $\mathcal{A}$.

$\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{PRG},1}_{s,\mathcal{A}}(1^\lambda)$
$x \leftarrow\$ \{0,1\}^\lambda$	
$y \leftarrow g(x)$	$y \leftarrow\$ \{0,1\}^{\lambda+s(\lambda)}$
$b^* \leftarrow\$ \mathcal{A}(1^\lambda, y)$	$b^* \leftarrow\$ \mathcal{A}(1^\lambda, y)$
return b^*	**return** b^*

Figure 2: Security experiments for PRGs.

Now, we need to say that the adversary cannot *distinguish*. How do we formalize this? We *compare* the probability that an adversary returns 1 in the real experiment $\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$ and the probability that an adversary returns 1 in the ideal experiment $\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$. We say that the adversary is a successful distinguisher, if the difference between these probabilities is kind of big, which, in the context of this course, means that it is non-negligible (non-negligible = not negligible). We define negligible functions formally in Appendix A and now state the definition of a pseudorandom generator.

Definition 1.1 (Pseudorandom generator (PRG))**.** A pseudorandom generator with stretch s is a function

$$g : \{0,1\}^* \to \{0,1\}^*$$

that can be computed in deterministic polynomial time, which satisfies the correctness criteria

$$\forall \lambda \in \mathbb{N} : \ s(\lambda) \geq 1$$
$$\forall \lambda \in \mathbb{N} \ \forall x \in \{0,1\}^\lambda : \ |g(x)| = \lambda + s(\lambda)$$

Let $\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$ and $\mathsf{Exp}^{\mathsf{PRG},1}_{g,\mathcal{A}}(1^\lambda)$ be the security experiments defined in Figure 2, then g is pseudorandom if for all PPT adversaries $\mathcal{A}$ the difference

$$\mathsf{Adv}^{\mathsf{PRG}}_{g,s,\mathcal{A}}(\lambda) := \left| \Pr\left[\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{PRG},1}_{s,\mathcal{A}}(1^\lambda) = 1\right] \right|$$

is negligible in λ.

[4] See `https://cseweb.ucsd.edu/~russell/sicomp.ps` for the original paper and `https://eccc.weizmann.ac.il/report/2010/089/` for a modern version by Haitner, Reingold and Vadhan. See `https://www.youtube.com/watch?v=5sTrXuykEKg` for a talk by Vadhan.

Properties of pseudorandom generators Unlike one-way functions, PRGs yield output which is pseudorandom. Therefore, there are one-way functions which are not pseudorandom generators, namely the one-way functions which append a lot of zeroes in the end. Although they might be length-expanding, their output looks very much not like a uniformly random string. Phrasing this intuition formally to practice the use of security experiments is one of the exercises on this week's exercise sheet. However, very much like one-way functions, a pseudorandom generator might leak a lot of information. The way to prove this is by generic counterexample: We take a PRG, apply it only to half of its input and append the other half of the input. This function is length-expanding and pseudorandom, if the original PRG was length-expanding and pseudorandom (exercise). By this example, we know that some PRGs might leak part of their input (while still satisfying the definition of a PRG. Therefore, if hiding all information about the input is desired, a PRG is not a suitable choice.

Further Reading If you are interested in reading more about pseudorandom generators, you can have a look at Section 3.3 in the Crypto Companion `https://vlipiainen.github.io/cryptoproofcompanion/CryptoProofCompanion.pdf`.

2 The HILL Theorem: PRGs from OWFs

We are not going to study the HILL result in its full generality. Instead, we are going to see how to extract a *single* pseudorandom bit from a OWF. While that does not give us a pseudorandom generator (PRG) immediately, it goes into the right direction. Such a pseudorandom bit, extracted from (the input of) a one-way function, is called a *hardcore bit*. The Definition 2.1 states that this bit is pseudorandom (i.e., indistinguishable from a uniformly random bit), even when given the output of the OWF.

$\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{HB},1}_{f,b,\mathcal{A}}(1^\lambda)$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$	$y \leftarrow f(x)$
$z \leftarrow b(x)$	$z \leftarrow\!\!\$\ \{0,1\}$
$d^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, y, z)$	$d^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, y, z)$
return d^*	**return** d^*

Figure 3: Harcore bit security experiments

Definition 2.1 (Hardcore Bits (Indistinguishability Formulation))**.** A polytime computable deterministic function $b : \{0,1\}^* \to \{0,1\}$ is a *hardcore bit* for a one-way function f if for all PPT adversaries $\mathcal{A}$, the difference

$$\mathsf{Adv}^{\mathsf{HB}}_{f,b,\mathcal{A}}(\lambda) = \left| \Pr\left[\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{HB},1}_{f,\mathcal{A}}(1^\lambda) = 1\right] \right|$$

is negligible in λ, where $\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}}(1^\lambda)$ and $\mathsf{Exp}^{\mathsf{HB},1}_{f,\mathcal{A}}(1^\lambda)$ are the security experiments defined in Figure 3.

Before seeing how we can build a hardcore bit, let us see how a hardcore bit helps us in our project to build a PRG from an OWF. Instead of looking at arbitrary OWFs as the HILL theorem does, we look at one-way functions which are a little bit more friendly (e.g., they don't have lots of zeroes). Namely, we look at one-way functions which, if we feed them a uniform distribution, their output distribution is also uniformly random. This is the case for one-way functions which are *length-preserving* and *bijective*, i.e., each output has a pre-image, and

two different domain values map to two different image values. Note that many of the examples of one-way functions we saw before in this lecture do not have this property, e.g., because the last bit of their output is always 0 whereas the last bit of the uniform distribution is 0 only with probability $\frac{1}{2}$.

Theorem (Simplified HILL)**.** Let f be a length-preserving, bijective one-way function and assume that there exists a hardcore bit b for f. Then the following function is a pseudorandom generator with stretch $s(n) = 1$:

$$\forall x \in \{0,1\}^* \; g(x) := f(x)||b(x)$$

Proof. To show this, we need to use the security of b and show that it implies the security of g. As before, we do this by a transformation. Namely, we show that any adversary $\mathcal{A}_g$ against g can be turned into an adversary $\mathcal{A}_b$ against b (and then, since efficient adversaries against b cannot exist (by assumption), efficient adversaries against g cannot exist either). In the following, consider an arbitrary PPT adversary $\mathcal{A}_g$ against g, then we build $\mathcal{A}_b$ as follows:

$\mathcal{A}_b\,(y, z, 1^n)$

$y' \leftarrow y||z$

$d^* \leftarrow\!\$\; \mathcal{A}_g(y', 1^n)$

return d^*

Observe that if $\mathcal{A}_g$ is a (probabilistic) polynomial-algorithm, then $\mathcal{A}_b$ is also a probabilistic polynomial-time algorithm. Moreover, we show the following claim:

Claim 1. It holds that

$$\Pr\left[\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}_b}(1^\lambda) = 1\right] = \Pr\left[\mathsf{Exp}^{\mathsf{PRG},0}_{g,1,\mathcal{A}_g}(1^\lambda) = 1\right] \tag{1}$$

and

$$\Pr\left[\mathsf{Exp}^{\mathsf{HB},1}_{f,\mathcal{A}_b}(1^\lambda) = 1\right] = \Pr\left[\mathsf{Exp}^{\mathsf{PRG},1}_{g,1,\mathcal{A}_g}(1^\lambda) = 1\right] \tag{2}$$

and thus, by the definition of the advantages,

$$\mathsf{Adv}^{\mathsf{HB}}_{f,b,\mathcal{A}_b}(\lambda) = \mathsf{Adv}^{\mathsf{PRG}}_{g,1,\mathcal{A}_g}(\lambda).$$

Let us quickly recall why it suffices to show Claim 1. Claim 1 shows that the advantage of $\mathcal{A}_b$ against the hardcore bit b of OWF f, denoted by $\mathsf{Adv}^{\mathsf{HB}}_{f,b,\mathcal{A}_b}(\lambda)$, is equal to the advantage of $\mathcal{A}_g$ against the PRG g, denoted by $\mathsf{Adv}^{\mathsf{PRG}}_{g,1,\mathcal{A}_g}(\lambda)$. Thus, if $\mathsf{Adv}^{\mathsf{PRG}}_{g,1,\mathcal{A}_g}(\lambda)$ were non-negligible, then $\mathsf{Adv}^{\mathsf{HB}}_{f,b,\mathcal{A}_b}(\lambda)$ would be non-negligible and we would have reached a contradiction with the assumption that b is a hardcore bit for f. Therefore, we reached a contradiction and $\mathsf{Adv}^{\mathsf{PRG}}_{g,1,\mathcal{A}_g}(\lambda)$ cannot be non-negligible. Thus, the PRG g is a secure PRG assuming that b is a hardcore bit for f.
We now turn to proving Claim 1. We start with Equation 1. Note that in the experiments the stretch $s = 1$.

$\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}_b}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}_b}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}_g}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}_g}(1^\lambda)$
$x \leftarrow\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$	$y \leftarrow f(x)$	$y \leftarrow f(x)\|\|b(x)$	$y \leftarrow g(x)$
$z \leftarrow b(x)$	$z \leftarrow b(x)$		
$d^* \leftarrow\!\$\ \mathcal{A}_b(1^\lambda, y, z)$	$y' \leftarrow y\|\|z$		
	$d^* \leftarrow\!\$\ \mathcal{A}_g(1^\lambda, y')$	$d^* \leftarrow\!\$\ \mathcal{A}_g(1^\lambda, y)$	$d^* \leftarrow\!\$\ \mathcal{A}_g(1^\lambda, y)$
return d^*	**return** d^*	**return** d^*	**return** d^*

The first (left-most) column contains $\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}_b}(1^\lambda)$. From the first to the second column, we inline the code of $\mathcal{A}_b$, marked in grey.
The fourth (right-most) column contains $\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}_g}(1^\lambda)$. From the fourth to the third column, we inline the code of g, marked in grey.
Now, we need to argue that the code of column 3 and column 2 behaves indeed in the same way. In column 3, let us first rename y into y'. Then, we observe that $y' \leftarrow f(x)||b(x)$ and $y \leftarrow f(x);\ z \leftarrow b(x),\ y' \leftarrow y||z$ has the same behaviour which concludes the proof of Equation 1.
Let us now turn to Equation 2.

$\mathsf{Exp}^{\mathsf{HB},1}_{f,\mathcal{A}_b}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{HB},1}_{f,b,\mathcal{A}_b}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{PRG},1}_{g,s,\mathcal{A}_g}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{PRG},1}_{g,s,\mathcal{A}_g}(1^\lambda)$
$x \leftarrow\!\$\ \{0,1\}^\lambda$			
$y \leftarrow f(x)$	$y \leftarrow\!\$\ \{0,1\}^\lambda$	$y' \leftarrow\!\$\ \{0,1\}^{\lambda+1}$	$y \leftarrow\!\$\ \{0,1\}^{\lambda+1}$
$z \leftarrow\!\$\ \{0,1\}$	$z \leftarrow\!\$\ \{0,1\}$		
$d^* \leftarrow\!\$\ \mathcal{A}_b(1^\lambda, y, z)$	$y' \leftarrow y\|\|z$		
	$d^* \leftarrow\!\$\ \mathcal{A}_g(1^\lambda, y')$	$d^* \leftarrow\!\$\ \mathcal{A}_g(1^\lambda, y')$	$d^* \leftarrow\!\$\ \mathcal{A}_g(1^\lambda, y)$
return d^*	**return** d^*	**return** d^*	**return** d^*

The first (left-most) column contains $\mathsf{Exp}^{\mathsf{HB},1}_{f,\mathcal{A}_b}(1^\lambda)$. From the first to the second column, we inline the code of $\mathcal{A}_b$, marked in grey. Moreover, we replace $x \leftarrow\!\$\ \{0,1\}^\lambda;\ y \leftarrow f(x)$ by $y \leftarrow\!\$\ \{0,1\}^\lambda$ since f is length-preserving and bijective and preserves the uniform distribution and since the variable x is not accessed anywhere else in the experiment.
The fourth (right-most) column contains $\mathsf{Exp}^{\mathsf{PRG},1}_{g,s,\mathcal{A}_g}(1^\lambda)$. From the fourth to the third column, we rename variable y to y', marked in grey.
Now, we need to argue that the code of column 3 and column 2 behaves indeed in the same way.
This follows by realizing that sampling $z \leftarrow\!\$\ \{0,1\}$ and $y \leftarrow\!\$\ \{0,1\}^\lambda$ and appending the two into $y' \leftarrow y||z$ is the same as sampling $y' \leftarrow\!\$\ \{0,1\}^{\lambda+1}$ which concludes the proof of Equation 2. □

3 Constructing a hardcore bit

Theorem 2 assumes the existence of a hardcore bit, but what actually guarantees the existence of a hardcore bit? Can we assume that every length-preserving, bijective one-way function has a hardcore bit? Maybe, can we even have a

universal hardcore bit which works for *any* one-way function? The answer to the latter question is no, and one of the exercises this week is to prove this. However, as we will see shortly, when we turn f into a different function

$$f_{\mathrm{GL}}(x, r) := f(x)||r$$

which leaks half of its input, then we can show that this modified function f_{GL} has a hardcore bit. The idea for this construction is brillant, actually. From our counterexamples last week, it follows that the first bit cannot be a hardcore bit, since, if the function leaks its first half, then it also leaks the first bit, so, given $f(x)$, the first bit of x would then be easy to distinguish. A same analysis applies to the last bit or the 3rd bit or really any input bit. A similar analysis even applies to the xor of all the bits, since leaking the xor of all the bits does not affect the one-wayness (see Exercise Sheet 1). However, the brillant idea of Goldreich and Levin is to take the xor of a *random subset* of x. Namely, in the function $f_{\mathrm{GL}}(x, r)$, the second input r indicates which bits we take. Before going to the general definition, let us see an example:
Consider

$$x = 101001$$

and

$$r = 101101$$

, then the Goldreich-Levin hardcore bit $b_{GL}(x, r)$ is

$$(1 \wedge 1) \oplus (0 \wedge 0) \oplus (1 \wedge 1) \oplus (0 \wedge 1) \oplus (0 \wedge 0) \oplus (1 \wedge 1),$$

which is equal to

$$1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1.$$

Now, the general construction of the Goldreich-Levin hardcore bit is

$$b_{\mathrm{GL}}(x, r) := \bigoplus_{i=1}^{|x|} x_i \wedge r_i$$

and the Goldreich-Levin theorem states that this is a hardcore bit for f_{GL}.

Theorem (Goldreich-Levin)**.** Let f be a one-way function, and consider the tranformed function $f_{\mathrm{GL}}(x, r) := f(x)||r$, then b_{GL} is a hardcore bit for f_{GL}.

The proof of the Goldreich-Levin hardcore bit is very nice from the perspective of amplification. After all, we need to take a distinguisher and turn it into an inverter for a one-way function. In *Foundations of Cryptography I*, you can find a very nice description of this proof, and in the exercises, we'll have a simplified version of it.

4 Important take-away

This lecture introduced *pseudorandom generators* which are *deterministic*, *length-expanding* functions such that their output is indistinguishable from uniform. We formalized this notion by asking the adversary to distinguish between a *real* and an *ideal* experiment. This type of real-ideal-way of formulating security

experiments will accompany us through most of the course, so it is a good thing to prioritize in understanding.
The second most important content of this lecture is to remember that the existence of one-way functions implies the existence of PRGs and vice versa. In short:

$$\exists \text{ OWF} \Leftrightarrow \exists \text{ PRG}$$

A Negligible Functions

Definition A.1 (Negligible Function)**.** A function $\nu : \mathbb{N} \to \mathbb{R}_0^+$ is negligible if it converges to 0 faster than any positive inverse polynomial, i.e., for all constants c, there is a natural number $N \in \mathbb{N}$ such that for all $\lambda > N$, it holds that $\nu(\lambda) < \frac{1}{\lambda^c}$.

Recall that the definition of negligible functions is a bit technical, but it is convenient to work with as long as we are aware of the following two properties:

Claim 2. For two negligible functions $\nu : \mathbb{N} \to \mathbb{R}_0^+$ and $\mu : \mathbb{N} \to \mathbb{R}_0^+$, the following hold:

- $\nu + \mu : \mathbb{N} \to \mathbb{R}_0^+$, $\lambda \mapsto \nu(\lambda) + \mu(\lambda)$ is negligible.
- For every positive polynomial p, $p \cdot \nu : \mathbb{N} \to \mathbb{R}_0^+$, $\lambda \mapsto p(\lambda) \cdot \nu(\lambda)$ is negligible.