# 3 Randomized Algorithms and Complexity Classes

So far we have looked at classical *deterministic algorithms*. Such algorithms perform the same steps every time they are presented the same input and thus produce the same output. This is certainly a desirable behavior. Nevertheless, we shall now look at algorithms which are allowed to perform different computations if the same input is given repeatedly. We shall later see how we can benefit from this concept in two ways. First, we will use it to classify certain problems as being hard. Secondly, we shall use these algorithms to solve problems fast which are hard (or, at least, more difficult or less elegantly) to solve for deterministic algorithms (the notion of "solving" has to be made precise in this context).

## 3.1 Definition of Randomized Algorithms

We are introducing the non-deterministic behavior in the following way. The algorithms now have access to a *fair coin*. This is a device that on demand outputs a 0 or 1; we shall call the execution of one such experiment a *coin flip* or *trial*. The numbers are assumed to be generated **independently according to uniform distribution** on $\{0, 1\}$. This implies that the probability of seeing a 0 in a trial is 0.5 as is that for observing a 1. The algorithms can use these coin flips in the following way through a *randomized IF-statement*:

**if** ($coin = 1$) **then**
  do something
**else**
  do something else
**end if**

If random numbers other than 0 and 1 are needed, these can be approximately produced by repeatedly using the fair coin.

If it is helpful, we assume that we have access to a *random number generator*. This is a subroutine $rand(a, b)$ which receives two integers $a, b$, $a < b$ and returns a random number drawn **independently according to uniform distribution** from the set $\{a, a+1, \ldots, b-1, b\}$. Setting $a = 0$ and $b = 1$ will make the random generator into the fair coin. One can use this to assign a random number to a variable.

INTEGER $v$
$v \leftarrow \text{rand}(a, b)$

Generating a random number counts as one computational step. We shall say that the algorithms *calls* the random number generator.

**Remark 3.1** We shall always assume that we are dealing with ideal random number generators, i.e, that they produce numbers that are stochastically **independent** and produced **according to uniform distribution**. Software random generators do not meet this requirement because they themselves are deterministic algorithms. Nevertheless, good random generators – like RAND55 described in the appendix – are close enough to ideal random generators for most practical purposes.

**Definition 3.2** An algorithm which has access to a random number generator is called a *randomized algorithm*.

Note that deterministic algorithms can be understood as a special case of randomized algorithms that have access to but never use the random number generator.

For randomized algorithms, the notions of running time as given in Definition 2.13 do not make sense any more. The number of steps until the algorithm stops is not determined even for a fixed input; it depends on the outcomes of the random experiments.

Also the output is not determined any longer, because it depends on the coin flips. Both running time and output are **random variables** or random vectors. If there is no control on these variables then a randomized algorithm is not of great use. Often, however, there is a **probabilistic (probability-theoretical) control** on these quantities. At least, a finite expected value of the running time is desired.

**Definition 3.3 [Running time of randomized algorithms]** Let $R$ be a randomized algorithm.

- The *expected running time* $\boldsymbol{E}\big[\, T^R(\boldsymbol{X}) \,\big]$ of an algorithm $R$ for a fixed input $\boldsymbol{X}$ is the expected value of the number of atomic computational steps that $R$ performs on input $\boldsymbol{X}$ until it stops ($\boldsymbol{E}\big[\, T^R(\boldsymbol{X}) \,\big] = \infty$ is possible).

- The *worst-case running time* $T_w^R(n)$ of algorithm $R$ for input size $n$ is the longest running time of the algorithm on an input of size $n$, where a worst possible outcome of the random numbers and the input is assumed:

$$T_{\mathrm{w}}^R(n) := \max\{t \mid \boldsymbol{P}\big[\, T^R(\boldsymbol{X}) = t \,\big] > 0, \|\boldsymbol{X}\| = n\}$$

- The *best-case running time* $T_b^R(n)$ of algorithm $R$ for input size $n$ is the shortest running time of the algorithm on an input of size $n$, where a best possible outcome of the random numbers and the input is assumed:

$$T_{\mathrm{b}}^R(n) := \min\{t \mid \boldsymbol{P}\big[\, T^R(\boldsymbol{X}) = t \,\big] > 0, \|\boldsymbol{X}\| = n\}$$

In the following, we shall present an example to which we shall return later in Chapter 5 where we perform a thorough analysis of the expected running time.

## 3.2 Examples of Randomized Algorithms

In this chapter, we shall use randomized algorithms to classify problems as more or less hard. Besides this, there are good reasons for using randomized algorithms in practical applications for the following reasons.

- A better running time than deterministic algorithms.
- Less memory requirements than deterministic algorithms.
- Easier implementation than deterministic algorithms.
- The opportunity to foil an adversary.

---

**Example 3.4 [Expected Running Time]** Consider an algorithm which performs a loop in which a coin is flipped. The loop is exited if the coin returns a 1.

**while** $(coin = 0)$ **do**
  do something
**end while**

This program does not terminate if the coin never shows 1. We shall now calculate the probability that the loop is performed exactly $n$ times. If $X$ is the random variable representing the number of times the loop is performed until termination then we are interested in the quantities $\boldsymbol{P}[\, X = n \,]$, $n \in \mathbb{N}$. For the loop not to be executed at all $(n = 0)$, the first coin flip has to be 1. This happens with probability 0.5, whence $\boldsymbol{P}[\, X = 0 \,] = 0.5$. For the loop to be executed exactly once $(n = 1)$, the first coin flip

has to be 0 **and** the second has to be 1. This happens with probabilities 0.5 and 0.5 respectively, whence $\boldsymbol{P}[\,X = 1\,] = 0.5 \cdot 0.5 = 0.25$. Here we use Formula (A.1) as the coin flips are independent. For the loop to be executed exactly $n$ times, the first $n$ coin flips have **all** to be 0 **and** the $(n+1)$-st coin flip has to be 1. This happens with probabilities $0.5^n$ and 0.5 respectively, whence the general formula is

$$\boldsymbol{P}[\,X = n\,] = 0.5^n \cdot 0.5 = 0.5^{n+1} \ .$$

Note that the sum $\sum_{n=0}^{\infty} \boldsymbol{P}[\,X = n\,]$ of these probabilities is 1 and that the probabilities are exponentially decreasing with $n$. Long running times are highly unlikely.

We now compute the expected running time of the algorithm, i.e., the expected number $\boldsymbol{E}[\,X\,]$ of times the loop is executed. By the Formula (A.6), this is the sum of the possible executions of the loop $(0, 1, 2, \ldots)$ times the probabilities that they occur:

$$\boldsymbol{E}[\,X\,] = \sum_{n=0}^{\infty} n \cdot \boldsymbol{P}[\,X = n\,] = \sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n+1} \ .$$

We claim that this sum is 1. To prove the claim, we use an *accounting method* (also called *bookkeeper's trick*) of reordering the terms and summing in a different way:

$$\begin{array}{rcccccccccccl}
\sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n+1} & = & 0 \cdot \frac{1}{2} & + & 1 \cdot \left(\frac{1}{2}\right)^2 & + & 2 \cdot \left(\frac{1}{2}\right)^3 & + & 3 \cdot \left(\frac{1}{2}\right)^4 & + & 4 \cdot \left(\frac{1}{2}\right)^5 & + \cdots & \\
& = & 0 & + & \left(\frac{1}{2}\right)^2 & + & \left(\frac{1}{2}\right)^3 & + & \left(\frac{1}{2}\right)^4 & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & + & \left(\frac{1}{2}\right)^3 & + & \left(\frac{1}{2}\right)^4 & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & & & + & \left(\frac{1}{2}\right)^4 & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & & & & & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & & & & & & \ddots & + \cdots & \\
& = & 0 & + & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + \cdots & = \frac{1}{2} \\
& & & & & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + \cdots & = \frac{1}{4} \\
& & & & & & & + & \frac{1}{16} & + & \frac{1}{32} & + \cdots & = \frac{1}{8} \\
& & & & & & & & & + & \frac{1}{32} & + \cdots & = \frac{1}{16} \\
& & & & & & & & & & \ddots & + \cdots & = \frac{1}{32} \\
\hline
& & & & & & & & & & & & 1
\end{array}$$

After expanding the formula, we compute the sums $S_n$ over the rows to get the well known value $S_n = (1/2)^n$ for the $n$-th row. Then we sum the row sums to get $\sum_{n=1}^{\infty} S_n = \sum_{n=1}^{\infty} \frac{1}{2}^n$ which is 1. The reordering of the infinite sum is allowed because all terms are positive.

---

Appendix B.3 contains more general formulas that allow us to treat infinite sums as in the previous example.

---

**Example 3.5 [Foiling an Adversary]** Assume that we are given $n = 3k$ balls. A third is colored red, a third green, and the last third blue. The balls are also numbered $1, \ldots, n$, but the numbers do not give information on the color. We have to find three equally colored balls. We do not see the balls but we can ask someone a questions like: "Do the balls 1, 12, and 142 have the same color?" In case of a "no" we are not told the colors. We could enumerate all $\binom{n}{3}$ triples $\{i, j, m\}$ of different numbers and ask the corresponding question.

Now suppose the balls are numbered by an *adversary* whose aim it is to maximize the running time. If the adversary knows what order we enumerate the triples in, he will number the balls in such a way that we find a triple of one color as late as possible.

Now consider a randomized strategy, which picks a triple at random until it is sccessful.

```
boolean success
success ← FALSE
while not success do
  i ← rand(1, n)
  j ← rand(1, n)
  m ← rand(1, n)
  if i, j, m are different then
    if i, j, m have the same color then success ← TRUE
    end if
  end if
end while
```

Here are two conditions to be met to be successful. First, we have to get three **different** numbers in the three calls to $rand(1, n)$. Secondly, if the numbers are different then the tree balls have to have the same color.

Let us denote the probability of the first condition by $q$. We claim that $q = 1-3/n+2/n^2$ and leave the proof to the reader. If the three balls are different then we compute the probability of them having the same color as follows. Assume the first ball is red. Then the chances for the second one to be red also are $(k-1)/(n-1)$. Then the chances for the third one to be red also are $(k-2)/(n-2)$. Hence the probability for three red balls is $((k-1)(k-2))/((n-1)(n-2))$ if the first is red. So the probability of getting three balls with the same color is

$$\frac{(k-1)(k-2)}{(n-1)(n-2)},$$

which approaches $(1/9)$ from below if $n$ grows. Altogether, the probability $p$ of success in a single execution of the loop is

$$p := (1 - 3/n + 2/n^2) \cdot \frac{(k-1)(k-2)}{(n-1)(n-2)} \ .$$

Let us determine the probability $\boldsymbol{P}[\,X = t\,]$ that it takes exactly $t$ rounds (executions of the while loop) to success. In order that the first success is in round $t$, there must be failures in rounds $1, \ldots, t-1$ and a success in round $t$, i.e.,

$$\boldsymbol{P}[\,X = t\,] = (1-p)^{t-1} \cdot p.$$

The expected running time is:

$$\boldsymbol{E}[\,X\,] = \sum_{t=1}^{\infty} t \cdot (1-p)^{t-1} \cdot p.$$

In the table below the values for $\boldsymbol{P}[\,X = t\,]$ and $k = 100$ ($n = 300$) are listed for $t = 1, \ldots, 10$.

| $t$ | $(1-p)^{t-1} \cdot p$ | $t \cdot (1-p)^{t-1} \cdot p$ |
|---|---|---|
| 1 | 0.1078000000 | 0.1078000000 |
| 2 | 0.09617916000 | 0.1923583200 |
| 3 | 0.08581104655 | 0.2574331397 |
| 4 | 0.07656061573 | 0.3062424629 |
| 5 | 0.06830738136 | 0.3415369068 |
| 6 | 0.06094384565 | 0.3656630739 |
| 7 | 0.05437409909 | 0.3806186936 |
| 8 | 0.04851257120 | 0.3881005696 |
| 9 | 0.04328291603 | 0.3895462443 |
| 10 | 0.03861701768 | 0.3861701768 |

Using the waiting time argument (Lemma A.2), the expected running time is $\boldsymbol{E}[\,X\,] = 1/p$, which for $n = 300$ is approximately 9.276.

---

---

**Example 3.6 [MaxCut]** The MaximumCut (or MaxCut) problem is defined on Page 61. The following randomized algorithm computes a cut in a graph $G = (V, E)$, i.e., it partitions $V = \{v_1, v_2, \ldots, v_n\}$ into $V_1$, $V_2$. We want to determine the expected number $X$ of edges between $V_1$, $V_2$.

**Algorithm 3.7**

**for** $i = 1, 2, \ldots, |V|$ **do**
  **if** coin $= 0$ **then**
    put $v_i$ into $V_1$
  **else**
    put $v_i$ into $V_2$
  **end if**
**end for**

**Claim 3.8** *Algorithm 3.7 runs in time $O(n)$. The expected number of edges between $V_1$, $V_2$ is $|E|/2$.*

**Proof.** The running time obviously is $O(n)$. Consider an edge $\{a, b\} \in E$. Let $X$ be the random variable denoting the number of edges in the cut. Edge $\{a, b\}$ is in the cut iff

$$[(a \in V_1) \wedge (b \in V_2)] \vee [(b \in V_1) \wedge (a \in V_2)] \tag{3.1}$$

As we use a fair coin, we have

$$\boldsymbol{P}[\, a \in V_1 \,] = 0.5, \quad \boldsymbol{P}[\, a \in V_2 \,] = 0.5, \quad \boldsymbol{P}[\, b \in V_1 \,] = 0.5, \quad \boldsymbol{P}[\, b \in V_2 \,] = 0.5 \; .$$

Then event 3.1 happens with probability

$$\begin{aligned} \boldsymbol{P}[\, \{a, b\} \text{ is in cut} \,] &= \boldsymbol{P}[\, a \in V_1 \,] \cdot \boldsymbol{P}[\, b \in V_2 \,] + \boldsymbol{P}[\, b \in V_1 \,] \cdot \boldsymbol{P}[\, a \in V_2 \,] \\ &= 0.5 \cdot 0.5 + 0.5 \cdot 0.5 = 0.5. \end{aligned}$$

Using Equation A.19, we get

$$\boldsymbol{E}[\, X \,] = \sum_{\{a,b\} \in E} \boldsymbol{P}[\, \{a, b\} \text{ is in cut} \,] = \sum_{\{a,b\} \in E} 0.5 = 0.5 \cdot |E| \,,$$

which finishes the proof. □

---

Let us sketch how one uses such algorithms in practice:

- Given a graph $G(V, E)$.
- Run the randomized algorithm, and let $C$ be the resulting cut, i.e., the set of edges between $V_1$ and $V_2$.
- If $|C| \geq (1/2)\,|E|$, be happy and stop.
- If $|C| < (1/2)\,|E|$, run the algorithm again.
- Reason: Because the **expected** size of a cut is $(1/2)\,|E|$ **some** executions of the algorithm have to yield cuts of that size or larger.
- We repeat until we are lucky.
- As the running time is fast, we can do this many times.

---

**Example 3.9 [Graph Coloring]** The problem is defined on Page 61.

Our algorithm RandGC receives the graph $G$ and the colors $1, \ldots, k$ as input. It randomly assigns a color to every vertex and then checks whether this is a legal coloring.

```
proc RandGC(G,k)
for all nodes v ∈ V do
  color(v) ← rand(1, k)
end for
for all edges {v, w} ∈ E do
  if color(v) = color(w) then
    return(NO)
  end if
end for
return(YES)
end proc
```

If the graph $G$ does not have a $k$-coloring then algorithm RandGC will always answer NO. If $G$ does have a $k$-coloring, the algorithm **can** guess it. That means, there is a positive chance that all assignments of the form $color(v) \leftarrow rand(1, k)$ will produce such a coloring. In the worst case, $G$ has only a single $k$-coloring (up to permuting the colors). We very crudely estimate the probability to guess this. Then for every vertex the chance to receive the right color is $(1/k)$. The probability that all nodes get the right color thus is

$$\boldsymbol{P}[\,\text{success}\,] = \left(\frac{1}{k}\right)^n \cdot k! \qquad \text{where} \qquad n = |V|\,.$$

The term $k!$ is the number of permutations of the $k$ colors. For $n$ only slightly larger than $k$, this probability is very small.

---

More examples will be presented in Chapter 5.

## 3.3 Complexity Classes

In this chapter, we shall use the concept of randomized algorithms to classify problems as being hard. To this end, we augment randomized algorithms in such a way that they always terminate after a predetermined number of steps. Let $f\colon \mathbb{N} \mapsto \mathbb{N}$ be an easily computable function. We want to ensure that the algorithms always stop after $f(n)$ steps, where $n$ is the input size. Let $A$ be a randomized algorithm. We construct a terminating algorithm $A'$ as follows. Algorithm $A'$ first computes the input length $n$ by

scanning the input. Then $A'$ computes the runtime bound $f(n)$ and initializes a counter $c = 0$. Then the code of $A$ follows, where at every instruction (atomic computational step) of $A$, code is added to perform the following:

- Increment the counter $c$ by one.
- Check if $c$ exceeds the running time bound ($c > f(n)$).
- If this is the case then $A'$ is terminated. If an output is expected, $A'$ will deliver a default value, for example DON'T KNOW.
- If $A'$ stops earlier, it returns the output which $A$ would have returned.

Sometimes, a less complicated way can be chosen to guarantee termination after a fixed number of steps. In the program below, it suffices to guarantee termination of the while-loop because there is no call to the random generation in its body.

$bound \leftarrow x$
$count \leftarrow 0$
**while** $((count < bound) \wedge (coin = 0))$ **do**
  do something
  $count \leftarrow count + 1$
**end while**

Assume that "do something" takes constant time.

**Definition 3.10** A randomized algorithm whose running time is bounded by a function $f\colon \mathbb{N} \mapsto \mathbb{N}$ is called *$f$-bounded.*

That is, the algorithm stops after at most $f(|\boldsymbol{x}|)$ steps on input $\boldsymbol{x}$. The output might be the default output ("DON'T KNOW").

Note that using the method to terminate the algorithm described before the definition increases the running time only by a constant factor. Also note that the function $f$ measures the atomic computational steps of the original algorithm $A$. We shall now present a new view on randomized algorithms using the following observation.

**Observation 3.11** *Let $f\colon \mathbb{N} \mapsto \mathbb{N}$ be a function. On input $\boldsymbol{X}$ an $f$-bounded randomized algorithm makes at most $f(\|\boldsymbol{X}\|)$ calls to the random number generator.*

We can thus produce enough random numbers in advance by calling the random number generator $f(\|\boldsymbol{X}\|)$ times and supply them to the algorithm as an additional input $R$. The algorithm then stores these numbers in a list. Every time it needs a new

random number, it takes a **new** one from the list instead of calling the random number generator. We shall write

$$A(\boldsymbol{X}, R)$$

to indicate that $f$-bounded algorithm $A$ is run on input $\boldsymbol{X}$, and $R$ is (a coding of) a sequence of $f(\|\boldsymbol{X}\|)$ random numbers. We shall always assume without further notice that the random numbers in $R$ are of the correct type. So, if $A$ uses a fair coin then $R$ will be a random bit string, if $A$ needs numbers from the set $\{2, 3, 4\}$ then $R$ will consist of such numbers.

When talking of the input size, we only consider the "true" input $\boldsymbol{X}$, not the random string $R$. The reason is that the string $R$ is used to model the – otherwise internal – randomization of the algorithm.

The randomness of an algorithm is now induced through the additional argument $R$. For fixed $\boldsymbol{X}$ and $R$, the algorithm behaves deterministically. Different random information $R$ might, however, give different outputs for the same $\boldsymbol{X}$. Hence, the output is a random variable with respect to the the distribution of $R$. Let $\boldsymbol{X}$ be an input for a $f$-bounded randomized algorithm $A$. Let $a, b \in \mathbb{N}$, $a \leq b$ be such that $A$ only used random numbers from $\{a, \ldots, b\}$. Then for $A(\boldsymbol{X}, R)$ the additional argument $R$ is an element of $\{a, \ldots, b\}^{f(\|X\|)}$. We make the following general assumption:

**Assumption 3.12** *Let $\boldsymbol{X}$ be an input for a $f$-bounded randomized algorithm $A$. The random argument $R$ of an algorithm is generated according to* ***uniform distribution*** *on $\{a, \ldots, b\}^{f(\|\boldsymbol{X}\|)}$.*

The random argument $R$ can be regarded as **help information** for the algorithm. One could imagine that this information helps the algorithm do the computations fast. As the help information is random, we can only hope that it really helps **most of the times**. That is, we want to focus on situations where there is some **probabilistic guarantee** that the random information really helps.

We shall classify problems according to the strength of the probabilistic guarantee that can be imposed on algorithms solving them. We restrict ourselves to yes-no-problems. A *complexity class* is a set of problems, i.e., a set of languages possibly over different alphabets. Problems in the same class have similar computational complexity. We shall define the classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{RP}$, $\mathcal{BPP}$, and $\mathcal{ZPP}$ below. Many more classes have been defined over the years but these remained the most important ones.

The first class of problems we describe are those which are efficiently solvable in the classical sense, i.e., by deterministic algorithms.

**Definition 3.13** A yes-no–problem is in $\mathcal{P}$ if there is a polynomial $p$ and a **deterministic** $p$-bounded algorithm $A$ such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & A(\boldsymbol{X}) = \text{YES}. \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & A(\boldsymbol{X}) = \text{NO}. \end{array}$$

We start with very weak probabilistic conditions by defining when a problem is in the *complexity class* $\mathcal{NP}$, where $\mathcal{NP}$ stands for *non-deterministic polynomial time*.

**Definition 3.14** A yes-no–problem is in $\mathcal{NP}$ if there is a polynomial $p$ and a randomized $p$-bounded algorithm $A$ such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \exists R, \|R\| \leq p(\|\boldsymbol{X}\|) \quad : & A(\boldsymbol{X}, R) = \text{YES}. \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \forall R \quad : & A(\boldsymbol{X}, R) = \text{NO}. \end{array}$$

Here, $R$ is a sequence of random numbers of the type required by the algorithm. An algorithm with these properties is called an $\mathcal{NP}$*-algorithm*.

Let us discuss the definition. The first condition only requires that for YES-inputs there is **some** random information $R$ such that the algorithm computes the correct answer. The algorithm is allowed to compute the wrong answer for all other choices of $R$. For NO-inputs, the algorithm has to **always** give the correct answer no matter what random information $R$ is supplied. We call such a behavior *one-sided error*, because errors are only on the YES-inputs.

We will use the discussion in the last paragraph to reformulate Definition 3.14 in terms of probabilities. The first condition expresses that the chances that for a fixed YES-input $\boldsymbol{X}$ a randomly chosen $R$ will lead to a correct result might be small but is not zero. The second condition states that the chances of a wrong answer on a NO-input are zero.

**Definition 3.15** A yes-no–problem is in $\mathcal{NP}$ if there is a polynomial $p$ and a randomized $p$-bounded algorithm $A$ such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & > \quad 0, \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = \quad 1, \end{array}$$

where $\boldsymbol{P}_R[\, Z\,]$ denotes the probability of event $Z$ over uniform distribution of $R$, $\|R\| \leq p(\|\boldsymbol{X}\|)$

If we strengthen the probabilistic conditions, we get other complexity classes.

**Definition 3.16** A yes-no–problem is in $\mathcal{RP}$ (*random polynomial time*) if there is a polynomial $p$ and a randomized $p$-bounded algorithm $A$ such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & \geq & \frac{1}{2}, \\ \text{True answer for } \boldsymbol{X} \text{ is NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = & 1. \end{array}$$

An algorithm with these properties is called an $\mathcal{RP}$*-algorithm.*

$\mathcal{RP}$-algorithms are also called *Monte Carlo*-algorithms. They have one-sided error. In contrast to $\mathcal{NP}$-algorithms, there is a good chance of getting the correct result for YES-inputs, namely at least 50%.

**Definition 3.17** A yes-no–problem is in $\mathcal{BPP}$ (*bounded error probabilistic polynomial*) if there is a polynomial $p$ and an $\varepsilon > 0$ (independent of $\|\boldsymbol{X}\|$) and a randomized $p$-bounded algorithm $A$ such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & \geq & \frac{1}{2} + \varepsilon. \\ \text{True answer for } \boldsymbol{X} \text{ is NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & \geq & \frac{1}{2} + \varepsilon. \end{array}$$

An algorithm with these properties is called an $\mathcal{BPP}$*-algorithm.*

$\mathcal{BPP}$-algorithms are allowed to have two-sided error, but the probability of answering correctly should be strictly greater than $1/2$ in both cases. Note that we only demand that *there is* a positive constant $\varepsilon$, and not all possible values of $\varepsilon$ have to be considered to prove that a problem is in $\mathcal{BPP}$. No matter if the success probability (the probability of answering correctly) of a randomized $p$-bounded algorithm is 99%, or 75%, or 50.1% etc., these success probabilities are sufficient to prove that the problem is in $\mathcal{BPP}$ (using $\varepsilon = 0.49$, $\varepsilon = 0.25$, and $\varepsilon = 0.01$, respectively). In the literature, people sometimes define $\mathcal{BPP}$ with success probability $2/3$ and sometimes with success probability $3/4$, all of which are equivalent. The proof is left to the reader.

**Definition 3.18** A yes-no–problem is in $\mathcal{ZPP}$ (*zero error probabilistic polynomial*) if there is a polynomial $p$ and a randomized $p$-bounded algorithm $A$ such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{llllll}
\text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & \geq & \frac{1}{2}. \\
\text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = & 0. \\
\text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & \geq & \frac{1}{2}. \\
\text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & = & 0.
\end{array}$$

An algorithm with these properties is called a *$\mathcal{ZPP}$-algorithm*.

Note that the definition of $\mathcal{ZPP}$ allows that we do not get an answer at all. Alternatively, one can assume the answer DON'T KNOW in that case. $\mathcal{ZPP}$-algorithms are also called *Las Vegas*-algorithms.

The above definitions are summarized in the following table:

| **Class** | **True answer YES** | **True answer NO** |
|---|---|---|
| $\mathcal{NP}$ | $\boldsymbol{P}[\,\text{YES}\,] > 0$ | $\boldsymbol{P}[\,\text{NO}\,] = 1$ |
| $\mathcal{RP}$ | $\boldsymbol{P}[\,\text{YES}\,] \geq \frac{1}{2}$ | $\boldsymbol{P}[\,\text{NO}\,] = 1$ |
| $\mathcal{BPP}$ | $\boldsymbol{P}[\,\text{YES}\,] \geq \frac{1}{2} + \varepsilon$ | $\boldsymbol{P}[\,\text{NO}\,] \geq \frac{1}{2} + \varepsilon$ |
| $\mathcal{ZPP}$ | $\boldsymbol{P}[\,\text{YES}\,] \geq \frac{1}{2}\,, \quad \boldsymbol{P}[\,\text{NO}\,] = 0$ | $\boldsymbol{P}[\,\text{NO}\,] \geq \frac{1}{2}\,, \quad \boldsymbol{P}[\,\text{YES}\,] = 0$ |

**Table 3.1**: Overview over the complexity classes.

**Theorem 3.19** *The following relations hold among the complexity classes, see also Figure 3.1 for an illustration.*

$$\begin{aligned}
\mathcal{P} &\subseteq \mathcal{ZPP} \\
\mathcal{ZPP} &\subseteq \mathcal{RP} \\
\mathcal{RP} &\subseteq \mathcal{NP} \\
\mathcal{RP} &\subseteq \mathcal{BPP} \\
\mathcal{P} &\subseteq \mathcal{BPP}
\end{aligned}$$

**Proof.** The proof is by looking at the definitions of the classes. If the definition of class $\mathcal{A}$ is a restriction of the definition of class $\mathcal{B}$ then we have $\mathcal{A} \subset \mathcal{B}$. Only the relation $\mathcal{RP} \subseteq \mathcal{BPP}$ is non-trivial to prove. The additional argument required here is

as follows: Boost the success probability of the $\mathcal{RP}$-algorithm to at least $1/2 + \epsilon$ by invoking the original $\mathcal{RP}$-algorithm several times. The details should be filled in by the reader (in an exercise). □

Since the success probability of an $\mathcal{NP}$-algorithm might be exponentially small (e. g., $2^{-n}$, where $n$ is the input length), polynomially many calls of an $\mathcal{NP}$-algorithm are not necessarily enough to obtain a reasonable success probability. In particular, exponentially many calls might be needed to boost the success probability to at least $1/2$. Hence, boosting the success probability does not work to show that $\mathcal{NP} \subseteq \mathcal{RP}$, and most likely, this statement does not hold anyway.

Finally, we remark that the relation between $\mathcal{NP}$ and $\mathcal{BPP}$ is not known.

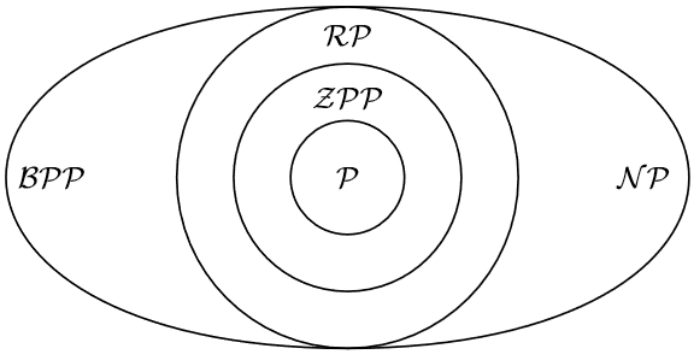


**Figure 3.1**: The relation of the complexity classes.