CHAPTER 2

Theoretical Computer Science 101

Before we consider quantum computing, it is worthwhile to review classical computing. Modern computers are very complicated. People hence study many abstractions of the workings of a computer, called "models of computation". In this chapter, we will introduce three such models of computation.

1. Traditional Computer Science

Computer Science grew out of the work led by David Hilbert, who made significant contributions to the field of mathematics, including the development of formal axiomatic systems, which laid the foundation for the study of mathematical logic and the formalization of algorithms. His work in these areas has influenced the development of theoretical computer science, including the study of computability and complexity theory. Additionally, Hilbert's work on geometry and his development of the concept of Hilbert spaces have had an impact on the field of computer graphics as well as quantum mechanics. In the context of this chapter it is important to stress that it is Hilbert who tried to distinguish between problems that can be solved by simple methods and those which can not.

Much of computer science uses a language-inspired definition of a decision problem. One starts with a finite alphabet A. By stringing elements of the alphabet one after another, one obtains strings of finite or countably infinite length. A set of strings is called a language. A decision problem is defined by a fixed set S, which is a subset of the language U of all possible strings over the alphabet A. A particular instance of the decision problem is to decide, given an element $u \in U$, whether u is included in S.

Example 2.1 (Primality testing.). For example, the alphabet could be composed of binary digits $A = \{0, 1\}$, U could be the set of all natural numbers encoded in binary, and the set S could be the binary encodings of prime numbers. The decision problem is the inclusion of an arbitrary binary encoding of a natural number in the set of S. ♢

Several models of computation were devised. Alan Turing introduced a model, where characters are stored on an infinitely long tape, with a read/write head scanning one square at any given time and having very simple rules for changing its internal state based on the symbol read and current state. Another influential model, called Lambda Calculus, has been introduced by Alonzo Church. Many of these formalisms turn out to be equivalent in computational power, *i.e.*, any computation that can be carried out with one can be carried out with any of the others. As it turns out, quantum computing may be one of the first models where this is not the case.

1.1. Turing Machines.

Formally, one can define a Turing machine using:

- a finite, non-empty set Q of objects, representing states
- a subset F of Q, corresponding to "accepting" states, where computation halts
- $q_0 \in Q$, the initial state
- a finite, non-empty set Γ of objects, representing the symbols to be used on a tape
- a partial function $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, 1\}$ where for a combination of a state and symbol read from the tape, we get the next state, the symbol to write onto the tape, and an instruction to shift the tape left (-1), right (+1), or keep in its position (0).

Notice that here we assume the input is on the tape, at the beginning.

Example 2.2 (There and Back Again.). Let us, for example, construct a machine, which scans over an integer encoded in binary and delimited by "blank" on the tape from left to right, and back. This is not very useful, but will be easy to understand:

- $Q = \{\text{goingright}, \text{goingleft}, \text{halt}\}$
- $F = \{\text{halt}\}$
- $q_0 = \text{goingright}$
- $\Gamma = \{0, 1, \text{“}blank''\}$
- δ given by the table below:

Current state	Scanned symbol	Print symbol	Move tape	Next state
goingright	0	0	1	goingright
goingright	1	1	1	goingright
goingright	blank	blank	-1	goingleft
goingleft	0	0	-1	goingleft
goingleft	1	1	-1	goingleft
goingleft	blank	blank	0	halt

♢

Exercise 2.3. Consider the following simulator of a Turing machine (TM):

```
def turing(code, tape, initPos = 0, initState = "1"):
  position = initPos
  state = initState
  while state != "halt":
    print f"{state} : {position} in {tape}"
    symbol = tape[position]
    (symbol, direction, state) = code[state][symbol]
    if symbol != "noWrite": tape[position] = symbol
    position += direction
```

code/ch1/turing.py

Implement a TM, which checks whether an integer, which is encoded on the tape as in binary and delimited by "blank" on both ends of the tape, is odd. If so, it should replace all symbols representing the integer with "1". Otherwise, it should replace all symbols representing the integer with "0".

Exercise 2.4. Consider the same simulator of a Turing machine (TM) as in Exercise 2.3. Implement a TM, which adds two integers, encoded on the tape in binary and delimited by "blank" on both ends of the tape and between the numbers. Replace both numbers with the result.

Exercise 2.5. Consider the simulator of a Turing machine (TM) as in Exercise 2.3. Implement a TM, which multiplies two integers, which are encoded on the tape in unary and and delimited by "blank" on both ends and between the numbers. Do not replace the numbers, but append the result after yet another blank.

Hint: Unary encoding means that the number of occurrences of a particular symbol (e.g., "1") is equal to the number (e.g., "11111" stands for 5).

1.2. Computability. Computability studies these models of computation, and asks which problems can be proven to be unsolvable by a computer. For example:

Example 2.6 (The Halting Problem). Given a program and an input to the program, will the program eventually stop when given that input? ♢

A silly solution would be to just run the program with the given input, for a reasonable amount of time. If the program stops, we know the program stops. But if the program doesn't stop in a "reasonable" amount of time, we cannot conclude that it *won't* stop. Maybe we didn't wait long enough. Alan Turing

proved the Halting problem to be undecidable in 1936. This could be seen as a special case of Gödel's First Incompleteness Theorem (1929).

To give another example,

EXAMPLE 2.7 (Hilbert's Tenth Problem). Given a polynomial equation with integer coefficients and a finite number of unknowns, is there a solution with all unknowns taking integer values? ♢

In 1970, Yuri Matiyasevich showed the undecidability Hilbert's Tenth Problem, building upon the work of Martin Davis, Hilary Putnam and Julia Robinson.

1.3. Complexity theory. Some problems are solvable by a computer, but require such a long time to compute that the solution is impractical. Here, we express the run time as a function from the dimensions of the input to the numbers of steps of a Turing machine (or similar).

EXAMPLE 2.8 (Fischer-Rabin Theorem.). For example, let us have a logic featuring 0, 1, the usual addition, and where the axioms are a closure of the following:

- $\neg(0 = x + 1)$
- $x + 1 = y + 1 \Rightarrow x = y$
- $x + 0 = x$
- $x + (y + 1) = (x + y) + 1$
- For a first-order formula $P(x)$ (i.e., with the universal and existential quantifiers) with a free variable x, $(P(0) \wedge \forall x(P(x) \Rightarrow P(x + 1))) \Rightarrow \forall y P(y)$ ("induction").

This is known as the Presburger arithmetic. Fischer and Rabin proved in 1974 that any classical algorithm that decides the truth of a statement of length n in Presburger arithmetic has a runtime of at least $2^{2^{cn}}$ for some constant c, because it may need to produce an output of that size. Hence, this problem needs more than exponential run time. ♢

Complexity theory deals with questions concerning the time or space requirements of given problems: the *computational cost*. For algorithms working with finite strings from a finite alphabet, this is often surprisingly easy.

1.4. Computational Complexity of Discrete Algorithms. The term *analysis of algorithms* is used to describe general approaches to putting the study of the performance of computer programs on a scientific basis. One such approach[1] concentrates on determining the growth of the worst-case performance of the algorithm (an "upper bound"): An algorithm's "order" suggests asymptotics of the number of operations carried out by the algorithm on a particular input, as a function of the dimensions of the input.

EXAMPLE 2.9. For example, we might find that a certain algorithm takes time $T(n) = 3n^2 - 2n + 6$ to complete a problem of size n. If we ignore

- constants (which makes sense because those depend on the particular hardware/virtual machine the program is run on), and
- slower growing terms such as $2n$,

we could say "$T(n)$ grows at the order of n^2". ♢

1.5. The Bachmann–Landau Notation. Let us introduce a formalisation of the notion of asymptotics. The formalisation known as "Big O notation" or "Bachmann–Landau notation" goes back at least to 1892 and Paul Gustav Heinrich Bachmann, according to some sources, although it was reinvented many times over. Suppose our A requires $T(n)$ operations to complete the algorithm in the longest possible case. Then we may say A is $O(g(n))$ if $|T(n)/g(n)|$ is bounded from above as $n \to \infty$. The fastest growing term in $T(n)$ dominates all the others as n gets bigger and so is the most significant measure of complexity.

Similarly to "Big O", there are 4 more notions, as summarised in Table 2.1. Formally, suppose f and

[1]Introduced by Hartmanis and Stearns in: Juris Hartmanis and Richard Stearns (1965), On the computational complexity of algorithms, *Trans. Amer. Math. Soc.*, **117**:285–306; and popularised by Aho, Hopcroft and Ullman.

Notation	Definition	Analogy
$f(n) = O(g(n))$	see Def. 2.10	$\leq$
$f(n) = o(g(n))$	see Def.	$<$
$f(n) = \Omega(g(n))$	$g(n) = O(f(n))$	$\geq$
$f(n) = \omega(g(n))$	$g(n) = o(f(n))$	$>$
$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$ and $g(n) = O(f(n))$	$=$

TABLE 2.1. An overview of the Bachmann–Landau notation.

g are two real-valued functions defined on some subset of $\mathbb{R}$ and consider the following:

DEFINITION 2.10. We write:

$$f(x) = O(g(x)) \qquad \text{(or, to be more precise, } f(x) = O(g(x)) \text{ for } x \to \infty)$$

if and only if there exist constants N and $C > 0$ such that

$$|f(x)| \leq C|g(x)| \text{ for all } \ x > N \quad \text{or, equivalently, } \frac{|f(x)|}{|g(x)|} \leq C \text{ for all } \ x > N.$$

That is, $|f(x)/g(x)|$ is bounded from above as $x \to \infty$. Intuitively, this means that f does not grow faster than g. The letter "O" is read as "order" or just "Oh".

DEFINITION 2.11. We also write:

$$f(x) = \Omega(g(x)) \qquad \text{(for } x \to \infty)$$

if and only if there exist constants N and $C > 0$ such that

$$|f(x)| \geq C|g(x)| \text{ for all } x > N \quad \text{or, equivalently, } \frac{|f(x)|}{|g(x)|} \geq C \text{ for all } x > N.$$

That is, $|f(x)/g(x)|$ is bounded from below by a *positive* (*i.e.*, non-zero) number as $x \to \infty$. Intuitively, this means that f does not grow more slowly than g (*i.e.*, $g(x) = O(f(x))$). The letter "Ω" is read as "omega" or just "bounded from below by".

DEFINITION 2.12.

$$f(x) = \Theta(g(x)) \qquad \text{(for } x \to \infty)$$

if and only if there exist constants N, C and $D > 0$ such that

$$D|g(x)| \leq |f(x)| \leq C|g(x)| \text{ for all } x > N \quad \text{or, equivalently, } D \leq \frac{|f(x)|}{|g(x)|} \leq C \text{ for all } x > N.$$

That is, $|f(x)/g(x)|$ is bounded from both above and below by positive numbers as $x \to \infty$. Intuitively, this means that f grows roughly at the same rate as g.

EXAMPLE 2.13. Let us consider algorithm A with parameter n and polynomial run-time $O(n^k)$. By our definition of O, the algorithm is of order $O(n^k)$ if $|T(n)/n^k|$ is bounded from above as $n \to \infty$, or — equivalently — there are real constants $a_0, a_1, \ldots, a_k$ with $a_k > 0$ so that A requires

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

operations to complete in the worst case. Note that k is an integer constant independent of the algorithm input and independent of the parameter n. It may be that there is no such polynomial for the number of operations in terms of n. If there is such a polynomial, A is usually considered "good" as it does not require "very many" operations. ♢

This notation can also be used with multiple parameters and with other expressions on the right hand side of the equal sign. The notation:

$$f(n, m) = n^2 + m^3 + O(n + m)$$

represents the statement:

$$\text{there exist } C, N \text{ such that, for all } n, m > N : f(n, m) \leq n^2 + m^3 + C(n + m).$$

Notation	*Name*
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial

TABLE 2.2. Classes of functions commonly encountered in algorithm analysis

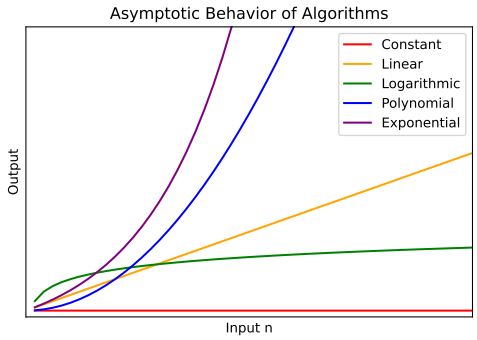

FIGURE 2.1. Schematic of the asymptotic runtime of algorithms as a function of their input size n.

Similarly, $O(mn^2)$ would mean the number of operations the algorithm carries out is a polynomial in two indeterminates n and m, with the highest degree term being mn^2, *e.g.*, $2mn^2 + 4mn - 6n^2 - 2n + 7$. This is most useful if we can relate m and n (*e.g.*, in dense graphs we have $m = O(n^2)$, so $O(mn^2)$ would mean $O(n^4)$ there).

Table 2.2 lists a number of classes of functions that are commonly encountered in the analysis of algorithms. Here, c is some arbitrary positive real constant. Once again, if a function $f(n)$ is a sum of functions, the fastest growing one determines the order of $f(n)$. E.g.: If $f(n) = 10\log(n) + 5(\log(n))^3 + 7n + 3n^2 + 6n^3$, then $f(n) = O(n^3)$.

One caveat here: the number of summands must be constant and may not depend on n.

Note that $O(n^c)$ and $O(c^n)$ are very different. The former is polynomial, the latter is exponential and grows much, much faster, no matter how big the constant c is. A function that grows faster than $O(n^c)$ is called *superpolynomial*. One that grows slower than $O(c^n)$ is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential.

Note, too, that $O(\log n)$ is exactly the same as $O(\log(n^c))$. The logarithms differ only by a constant factor, and the big O notation ignores such constant factors. Similarly, logarithms with different constant bases are equivalent.

EXERCISE 2.14. Prove that any later function in the above table grows faster than any earlier function. *Hint*: you need several small proofs. Also, each function is differentiable.

1.6. P and NP. Perhaps the best known question in Computer Science asks whether it can be harder to solve a problem than to check a given solution.

In complexity theory there are two commonly used classes of (decision) problems:

- The class **P** consists of all those decision problems that can be solved on a deterministic Turing machine in an amount of time that is polynomial in the size of the input, *i.e.*, $O(n^k)$ for some

constant k. Intuitively, we think of the problems in **P** as those that can be solved "reasonably fast".
- The class **NP** consists of all those decision problems whose *solutions* (called witnesses) can be verified in polynomial time on a Turing machine. That is, given a proposed solution to the problem, we can check that it really *is* a solution in polynomial time.

Formally: A language $L \subset \{0,1\}^*$ is in **NP**, if there exists a deterministic Turing machine M and a polynomial p such that upon receipt of:

- an input string x, e.g., $x \in \{0,1\}^*$,
- a witness of length $p(|x|)$

M runs in time polynomial in $|x|$ and

- for all $x \in L$, there exists y such that M accepts (x, y) ("completeness"),
- for all $x \notin L$, for all y, (x, y) is rejected ("soundness").

2. Randomized Algorithms

It seems quite unlikely that the Turing machine can produce a truly random number. But would the availability of a source of randomness make a Turing machine more powerful? We will formalise the question using the classes of Probabilistic Polynomial Time (PP) and Bounded-Error Probabilistic Polynomial Time (BPP), where BPP $\subset$ PP. It is not known whether BPP is equal to P or NP, i.e., whether the source of randomness helps at all or whether having access to a source of randomness makes a deterministic Turing machine as powerful as a non-deterministic Turing machine, despite much attention paid to the questions over the past couple of decades. On the other hand, it is known that NP $\subset$ PP and, in a somewhat different formalisation of Bennett and Gill [1981], we will see that the source of randomness does render many classes of computation (LOGSPACE^A, P^A, NP^A, PP^A, and PSPACE^A) properly contained in this order, with probability 1 with respect to random oracles A.

2.1. Definitions. In two important definitions of randomized computation, one considers a deterministic Turing machine M, which receives:

- an input string x, such as $x \in \{0,1\}^*$,
- a random string y, such as a realization $y \in \{0,1\}^*$ of a random variable Y

and

- accepts the input (x, y) for all x that we would like to be accepted with a certain probability,
- rejects (x, y) for all x we would like to be rejected with a certain probability,

where the probability is with respect to Y.

PP. A language $L \subset \{0,1\}^*$ is in PP, if there exists a deterministic Turing machine M and a polynomial p such that upon receipt of:

- an input string x, e.g., $x \in \{0,1\}^*$,
- a realisation y of length $p(|x|)$, e.g., $y \in \{0,1\}^{p(|x|)}$, of a random variable Y

M runs in time polynomial in $|x|$ and

- for all $x \in L$, (x, y) is accepted with a probability strictly greater than $1/2$,
- for all $x \notin L$, (x, y) is accepted with a probability less than or equal than $1/2$,

where the probability is with respect to Y.

In PP, we hence ask only for some "distinguishability". The "distinguishing" can, however, take arbitrarily long. Consider, for instance, a Turing machine M of the definition, that

- for all $x \in L$, (x, y) is accepted with probability $1/2 + 1/2|x|$
- for all $x \notin L$, (x, y) is accepted with probability $1/2 - 1/2|x|$.

For any number of trials, there is an $|x|$ that makes those necessary to achieve a fixed probability of the answer being correct. Notice that the number of trials grows exponentially with $|x|$.

Alternatively, PP is the set of languages, for which there is a variant of a non-deterministic Turing machine that stops in polynomial time with the acceptance condition being that more than one half of computational paths accept. For this reason, one sometimes refers to PP as Majority-P. It is thus clear that NP $\subseteq$ PP.

PP is often thought of as a counting class. Recall that the permanent of an $n \times n$ matrix $A = (a_{ij})$ is

$$\mathrm{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}. \tag{2.1}$$

Valiant [1979] showed that computing permanents is at least as hard as many so-called counting problems (#P-hard), and it is hard (#P-complete) even for matrices having only entries 0 or 1. The language $\{(A,k)|$ the permanent of A is at least $k\}$ is complete for PP, but it is believed to be outside of P. Alternatively, in terms of the number of accepting and rejecting paths, PP can be seen as computing the high-order bit of a #P function.

BPP. Let ϵ be a constant $0 < \epsilon < 1/2$. A language $L \subset \{0,1\}^*$ is in BPP, if there exists a deterministic Turing machine M and a polynomial p such that upon receipt of:

- an input string x, e.g., $x \in \{0,1\}^*$,
- a realisation y, e.g., $y \in \{0,1\}^{p(|x|)}$, of a random variable Y in dimension $p(|x|)$

M runs in time polynomial in $|x|$ and

- for all $x \in L$, (x,y) is accepted with a probability strictly greater than $1-\epsilon$,
- for all $x \notin L$, (x,y) is accepted with a probability less than or equal to ϵ,

where the probability is with respect to Y.

BPP can be seen as a subset of PP, for which there are efficient probabilistic algorithms. Indeed: the constant ϵ is independent of the dimension $|x|$, and thus any desired probability of correctness can be had with the number of trials independent of $|x|$ by the so-called *amplification of probability*. The majority vote of k trials will be wrong with probability:

$$\sum_{S \subseteq \{1,2,\ldots,k\}, |S| \leq k/2} (1-\epsilon)^{|S|} \epsilon^{k-|S|} \tag{2.2}$$

$$= ((1-\epsilon)\epsilon)^{k/2} \sum_{S \subseteq \{1,2,\ldots,k\}, |S| \leq k/2} \left(\frac{\epsilon}{1-\epsilon}\right)^{k/2-|S|} \tag{2.3}$$

$$< 2^k (\sqrt{(1-\epsilon)\epsilon})^k = \lambda^k \tag{2.4}$$

for some $\lambda = 2\sqrt{\epsilon(1-\epsilon)} < 1$. Cf. 4.1 in Kitaev et al. [2002].

How large is BPP within PP? It turns out that BPP is a substantial subset of PP. Bennett and Gill [1981] have shown that for a language $L \subset \{0,1\}^*$, the following are equivalent:

- $L \in$ BPP.
- For almost all oracles A, $L \in P^A$, wherein the almost all is with respect to a particular measure over the oracles.

Probabilistic Computation of Arora and Barak. It turns out that BPP has yet another definition, due to [Arora and Barak, 2009, Section 20.2], which is very instructive. It uses a seemingly different model of computation. There, one works with 2^N-dimensional vector $v \in [0,1]^{2^N}$, which we index with values from $\{0,1\}^N$, and which satisfies $\sum_{i \in \{0,1\}^N} v_i = 1$. This vector should be seen as a representation of a probability mass function of a random variable over $\{0,1\}^N$. One cannot access the values of v directly; rather, one obtains $i \in \{0,1\}^N$ with probability v_i, when one attempts to access v.

Let us introduce a special notation $|i\rangle$ for the representation of (so-called degenerate) distributions, where all the mass is concentrated in $v_i = 1$ for some $i \in \{0,1\}^N$. Because $|i\rangle_{i \in \{0,1\}^N}$ is a basis for $\mathbb{R}^{2^N}$, any

v can be represented as $\sum_{i\in\{0,1\}^N} v_i |i\rangle$. For the example of $N = 1$, we have $v = v_0 |0\rangle + v_1 |1\rangle$. The only operations permitted are linear stochastic functions $U : \mathbb{R}^{2^N} \to \mathbb{R}^{2^N}$ applied to the vector v, where linearity suggests $U(v) = \sum_{i\in\{0,1\}^N} v_i U(|i\rangle)$ and stochasticity suggests $\sum_{i\in\{0,1\}^N} U(v)_i = 1$ for all v satisfying $\sum_{i\in\{0,1\}^N} v_i = 1$. Notice that U can be represented by a matrix with non-negative entries, wherein each column sums up to 1. U can be a composition of multiple linear stochastic functions $U = U_L, U_{L-1}, \cdots U_2, U_1, U_i : \mathbb{R}^{2^N} \to \mathbb{R}^{2^N}$, where each U_i will represent the so-called gate and L will be the known as the depth of the circuit.

Let a probability threshold be a constant strictly larger than 1/2. A language $L \subset \{0,1\}^n$ is in BPP, if and only if its corresponding indicator function $F(x) : \{0,1\}^n \to \{0,1\}$ can be computed probabilistically in polynomial time such that:

(1) one starts with $v \in [0,1]^{2^N}$, for some $N \geq n$ dependent on F, with an initial state $|x, 0^{N-n}\rangle$ consisting of the input padded to length N by zeros;
(2) applies a linear stochastic function $U : \mathbb{R}^{2^N} \to \mathbb{R}^{2^N}$ to v, whose matrix representation can be computed in a sparse format by a Turing machine from all-ones input in time polynomial in n
(3) obtains a random variable Y, wherein $F(x)$ is followed by $N-1$ arbitrary subsequent symbols with probability at least as high as the probability threshold, while the random variable Y has value y with probability v_y for the value v of some final register.

Exercise 2.15. Prove the equivalence. Hint: find a way of generating $N-n$ Bernoulli random variables by a suitable U.

3. Quantum Algorithms

Now, one can obtain the class of BQP by replacing the real-valued vectors with complex-valued vectors:

Let a probability threshold be a constant strictly larger than 1/2. A language $L \subset \{0,1\}^n$ is in BQP, if and only if its corresponding indicator function $F(x) : \{0,1\}^n \to \{0,1\}$ can be computed probabilistically such that:

(1) one starts with an N-qubit register, for some $N \geq n$ dependent on F, with an initial state $|x, 0^{N-n}\rangle$ consisting of the input padded to length N by zeros;
(2) applies a linear function $U : \mathbb{C}^{2^N} \to \mathbb{C}^{2^N}$ to v, whose matrix representation (a unitary matrix in $\mathbb{C}^{2^N \times 2^N}$) can be computed in a sparse format by a Turing machine from all-ones input in time polynomial in n
(3) obtains a random variable Y, wherein $F(x)$ is followed by $N-1$ arbitrary subsequent symbols with probability at least as high as the probability threshold, wherein the random variable Y has value y with probability $|v_y|^2$ for the value v of some final register.

See Figure 2.2 for an overview if the complexity classes discussed, under mild assumptions. For example, the separations of BPP and BQP are known to be strict only in a relativized model of Yamakawa and Zhandry [2022], similar in spirit to the work of Bennett and Gill [1981], with probability one.

See also Abbas et al. [2023] for a high-level discussion.

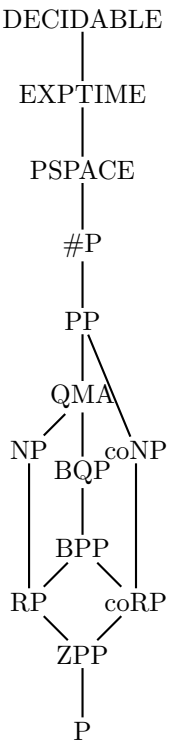

FIGURE 2.2. An overview of some of the non-strict inclusions among complexity classes.

Bibliography

Amira Abbas, Andris Ambainis, Brandon Augustino, Andreas Bärtschi, Harry Buhrman, Carleton Coffrin, Giorgio Cortiana, Vedran Dunjko, Daniel J Egger, Bruce G Elmegreen, et al. Quantum optimization: Potential, challenges, and the path forward. *arXiv preprint arXiv:2312.02279*, 2023.

Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

C. Bennett and J. Gill. Relative to a random oracle A, $\mathbf{p}^A \neq \mathbf{np}^A \neq \text{co-}\mathbf{np}^A$ with probability 1. *SIAM J. Comput.*, 10(1):96–113, 1981. doi: 10.1137/0210008.

A Yu Kitaev, AH Shen, and MN Vyalyi. *Classical and Quantum Computation*. American Mathematical Society, Providence, RI, 2002. Graduate Studies in Mathematics vol. 47).

L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(79)90044-6.

Takashi Yamakawa and Mark Zhandry. Verifiable quantum advantage without structure. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 69–74, 2022. doi: 10.1109/FOCS54457.2022.00014.