

Notes on Complexity Theory

Luke Chu

Last updated: 2026-10-22

These notes cover the basics of complexity theory, building up to the theory of average-case complexity, and in particular, to the theory of quantum average-case complexity. The sections on basic complexity classes, the polynomial hierarchy, and boolean circuits are based on the book *Computational Complexity: A Modern Approach* by Sanjeev Arora and Boaz Barak [AB09]. The section on average case complexity is primarily based on *Average-Case Complexity* by Andrej Bogdanov and Luca Trevisan [BT21].

Contents

1 Basic Complexity Classes	2
1.1 Turing Machines	2
1.1.1 Modelling Computation	2
1.1.2 Universality	3
1.2 Uncomputability	4
1.3 The Complexity Class P	4
1.4 The Complexity Class NP	5
1.4.1 NP-completeness	6
1.5 The Complexity Class coNP	7
2 The Polynomial Hierarchy	8
2.1 The Complexity Class Σ_2^P	8
2.2 The Polynomial Hierarchy	9
2.3 Properties of the Polynomial Hierarchy	9
3 Boolean Circuits	11
4 Randomized Algorithms	11
5 Quantum Computation	11
5.1 The Quantum Turing Machine Model	11
5.2 The Quantum Circuit Model	11
5.3 Grover's Search Algorithm	11
5.4 Simon's Algorithm	11
5.5 Shor's Algorithm	11
6 Average Case Complexity	11
6.1 Distributional Problems	11
6.2 Average Polynomial Time	11
6.2.1 The Complexity Class AvgP	11
6.2.2 The Complexity Class AvgBPP	13
6.2.3 The Complexity Class distNP	14
6.3 A Complete Problem for distNP	14
6.4 A Complete Problem for (NP, PSamp)	14
6.5 A Complete Problem for AvgBPP	14
7 Quantum Average Case Complexity	15
8 Examples of Quantum Average Case Complexity	15

8.1 Random Hamiltonian Simulation	15
Bibliography	15

1 Basic Complexity Classes

1.1 Turing Machines

1.1.1 Modelling Computation

The Turing Machine is a useful model of computation that allows us to formally define complexity classes. In particular, any other physically realisable model of computation can be simulated by a Turing Machine with little loss of efficiency (ignoring the possibility of quantum computation, which we will address later in these notes).

We now formally define a Turing Machine.

Definition 1.1 (Turing Machine) A Turing Machine is described by a tuple (Γ, Q, δ) containing:

- A finite set Γ of symbols, called the *tape alphabet*. We assume for simplicity that Γ contains a special blank symbol $\square$, a special start symbol $\triangleright$, and the numbers 0 and 1.
- A finite set Q of states in which the TM's finite control can be in. We assume that Q contains a special start state q_{start} and a special halting state q_{halt} .
- A transition function $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times \{L, S, R\}^k$, where $k \geq 2$ (see Remark 1.2).

The Turing Machine operates as follows. It has k tapes, each of which is infinite towards the right. The first tape is the input tape, which contains the input string x on its leftmost cells and is blank elsewhere. The other tapes are work tapes, which are all initially blank. The TM's finite control is initially in the start state q_{start} and the tape heads are all positioned at the leftmost cell of their respective tapes. At each step, the TM reads the symbols under the tape heads, applies the transition function δ to determine the next state, what symbols to write on each tape, and how to move each tape head (left, stay, right). The TM continues to operate until it reaches the halting state q_{halt} , at which point it stops. If the TM tries to go left from the leftmost cell of a tape, it remains in the same position.

Remark 1.2 This is the definition of a multi-tape Turing Machine, where the first tape is the input tape and the other tapes are work tapes. It is also possible to define a single-tape Turing Machine but it is known that multi-tape Turing Machines are equivalent to single-tape Turing Machines up to quadratic slowdown.

Definition 1.3 (Computing a function and running time) Let $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ and let $T : \mathbb{N} \rightarrow \mathbb{N}$ be some functions, and let M be a TM. We say that M *computes* f if for every input $x \in \{0, 1\}^*$, M halts with $f(x)$ written on its output tape¹. If the number

of steps that M takes to halt on input x is at most $T(|x|)$ for every input x , we say that M runs in time $T(n)$.

1.1.2 Universality

Clearly we can represent TMs by binary strings. We can do this by encoding each element of the TM (Γ, Q, δ) as a binary string, and then using a binary encoding of a tuple to encode the entire TM. Furthermore, for simplicity, we would like the property that every binary string $x \in \{0, 1\}^*$ represents a valid TM. We can easily achieve this by assigning all invalid binary strings to a trivial TM that immediately halts without doing anything.

With this, we can define a *Universal Turing Machine* U that takes as input (M, x) , where M is understood to be the binary encoding of a TM² and x is an input string. The universal TM then simulates M on input x , and halts with the same output as M would have on input x .

The existence of an *efficient* universal Turing Machine is a fundamental result. In particular, we have the following theorem.

Theorem 1.4 (Efficient Universal Turing Machine) There exists a universal Turing Machine U such that for every TM M and input x , if M runs in time $T(|x|)$, then U runs in time $cT(|x|)^2$ for some constant c that depends on M but not on x .

Proof sketch. The idea is to have U simulate M step by step. We first start by converting M into a single-tape TM $\tilde{M}$ (see Remark 1.2). This can incur a quadratic slowdown. We can also convert M into a machine with the alphabet $\{\square, \triangleright, 0, 1\}$. From now on, we can assume that M is a single-tape TM with the alphabet $\{\square, \triangleright, 0, 1\}$.

We then construct the universal TM U as follows. U first decodes its input (M, x) onto the following tapes:

- The first tape contains the input x . This will represent the work tape of M (remember, we are assuming that M is a single-tape TM).
- The second tape contains the transition function δ of M .
- The third tape contains the current state of M . This is initially set to the start state q_{start} .

To simulate one computational step of M , U first reads the current state of M from the third tape, and then reads the symbol under the first tape head (which is the work tape of M). It then scans through the second tape to find the corresponding transition in δ . This takes time proportional to the size of δ , which is a constant that depends on M but not on x . Once it finds the transition, it updates the first and third tapes accordingly, and moves the first tape head according to the transition. This simulates one step of M . $\square$

¹We can adopt the convention that the output tape is the first work tape, and that the input tape is read-only.

²We are using an abuse of notation here by using M to mean both the TM and the binary string representing the TM

Remark 1.5 Theorem 1.4 can be strengthened to show that there exists a universal TM that runs in time $cT(|x|) \log T(|x|)$.

1.2 Uncomputability

Theorem 1.6 There exists a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ that is not computable by any TM.

Proof. We can easily construct an explicit example of such a function.

Let f be defined as follows. For every binary string $x \in \{0, 1\}^*$, let $f(x) = 1$ if the TM represented by x halts on input x and returns 0. Otherwise, let $f(x) = 0$ (i.e. if the TM represented by x does not halt on input x , or if it halts and returns something else other than 0).

Now suppose for contradiction that there exists a TM M that computes f . Let x_M be the binary string representing M . We have two cases:

- If M halts on input x_M and returns 0, then by the definition of f , we have $f(x_M) = 1$. But since M computes f , we also have $f(x_M) = M(x_M) = 0$, which is a contradiction.
- If M does not halt on input x_M , or if it halts and returns something other than 0, then by definition of f , we have $f(x_M) = 0$. But since M computes f , we also have $f(x_M) = M(x_M) = 1$, which is a contradiction.

So in either case, we reach a contradiction. Therefore, no TM can compute f . $\square$

Example 1.7 (The halting problem) A more natural example of an uncomputable function is the *halting problem*, which is the function $\text{HALT} : \{0, 1\}^* \rightarrow \{0, 1\}$ which returns 1 if the TM represented by the input string halts on that input, and returns 0 otherwise.

If we had a TM M_{Halt} that computes HALT , we could use it to compute the function f defined in the proof of Theorem 1.6, which is a contradiction. The idea is that we can use M_{Halt} to first determine whether some TM M_x halts on input x . If it does not halt, we can immediately return 0. If it does halt, we can simulate M_x on input x using the universal TM to determine what it returns.

Therefore, the halting problem is uncomputable.

1.3 The Complexity Class P

A common framework for studying the (time) complexity of problems is in terms of *languages* and *decision problems*. A language L is a subset of $\{0, 1\}^*$ and a decision problem is a problem that can be phrased as a question of whether a given string $x \in \{0, 1\}^*$ belongs to some language L . Informally, we can think of decision problems as yes/no questions about strings.

We say that a TM M *decides* a language L if M computes $f_L : \{0, 1\}^* \rightarrow \{0, 1\}$ where $f(x) = 1 \Leftrightarrow x \in L$. f_L is also called the *characteristic function* of L .

Definition 1.8 (The class DTIME) Let $T : \mathbb{N} \rightarrow \mathbb{N}$ be a function. We define the complexity class $\text{DTIME}(T(n))$ as the set of languages L such that there exists a TM M that decides L and runs in time $cT(n)$ for some positive constant c .

The D in the name **DTIME** stands for “deterministic” since we are considering deterministic Turing Machines. We can also define the class $\text{NTIME}(T(n))$ in a similar way, but using non-deterministic Turing Machines instead.

Now we can try to define what we mean by “efficiently solvable” problems. A common definition is that a problem is efficiently solvable if it can be solved in polynomial time. This leads us to the following definition.

Definition 1.9 (The class P) $P = \cup_{c \in \mathbb{N}} \text{DTIME}(n^c)$.

We say that some decision problem has an efficient algorithm if it is in **P**.

Remark 1.10 The definition of **P** seems to depend on the choice of the model of computation (e.g. Turing Machines, RAM machines, etc.). However as we claimed earlier, any physically realisable model of computation can be simulated by a Turing Machine with at most a polynomial slowdown. Therefore, the definition of **P** is robust to the choice of model of computation.

1.4 The Complexity Class NP

Informally, the class **NP** represents problems whose solutions can be *verified* in polynomial time.

Definition 1.11 (The class NP) A language L is in **NP** if there exists a polynomial $p(n)$ and a polynomial-time TM M such that $\forall x \in \{0, 1\}^*$, $x \in L$ iff $\exists u \in \{0, 1\}^{p(|x|)}$ such that $M(x, u) = 1$. If there exists a string u such that $M(x, u) = 1$, then u is called a *certificate* of x . M is called the *verifier* of L .

Remark 1.12 The definition of **NP** can be equivalently phrased in terms of non-deterministic Turing Machines. In particular, we can start by defining the class $\text{NTIME}(T(n))$ like we did for $\text{DTIME}(T(n))$, but using non-deterministic Turing Machines instead. Then we can define $\text{NP} = \cup_{c \in \mathbb{N}} \text{NTIME}(n^c)$.

Remark 1.13 Clearly we have $P \subseteq \text{NP}$ since the certificate u can be the empty string. However, it is an open question whether $P = \text{NP}$ or if **P** is a strict subset of **NP**.

Example 1.14 TODO: Give some examples of problems in **NP**.

1.4.1 NP-completeness

Some problems in **NP** have the fascinating property that if there were an efficient algorithm for them, then there would be an efficient algorithm for *every* problem in **NP**. These problems can be thought of as the “hardest” problems in **NP** and are called **NP-hard**.

If furthermore this problem is also in **NP**, then it is called **NP-complete**. In other words, $\text{NP-complete} = \text{NP} \cap \text{NP-hard}$.

To make this more precise, we need to define what it means for one problem to be “at least as hard” as another problem. We do this using the notion of polynomial-time reductions.

Definition 1.15 (Polynomial-time reductions, NP-hardness) Let $L_1, L_2 \subseteq \{0, 1\}^*$ be two languages. We say that L_1 is *polynomial-time reducible* to L_2 , denoted $L_1 \leq_p L_2$, if there exists a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for every $x \in \{0, 1\}^*$, $x \in L_1$ iff $f(x) \in L_2$. The function f is called a *reduction* from L_1 to L_2 . We say that $L \in \text{NP-hard}$ if $\forall L' \in \text{NP}, L' \leq_p L$.

Lemma 1.16 If $L_1 \leq_p L_2$ and $L_2 \in \text{P}$, then $L_1 \in \text{P}$. In other words, **P** is closed under polynomial-time reductions.

Proof. We can construct a polynomial-time algorithm for L_1 as follows. Given an input x , we first compute $f(x)$ in polynomial time, and then run the polynomial-time algorithm for L_2 on input $f(x)$. The input to the algorithm for L_2 has size at most polynomial in the size of x , so the overall running time is polynomial in the size of x . Therefore, $L_1 \in \text{P}$. $\square$

Similarly, we can easily show that if $L_1 \leq_p L_2$ and $L_2 \in \text{NP}$, then $L_1 \in \text{NP}$. In other words, **NP** is also closed under polynomial-time reductions.

Remark 1.17 We cannot use polynomial-time reductions to classify problems *within* **P**, such as for defining a notion of **P**-completeness. This is because polynomial-time reductions have the same power as the class **P** itself. In particular, if we define **P**-hardness using polynomial-time reductions, then almost every problem in **P** would be **P**-hard, since we can solve the problem outright in the reduction itself.

To create a theory about the internal structure of **P**, we need to use a more restrictive notion of reductions, such as log-space reductions.

Example 1.18 Some examples of **NP-complete** problems include:

- The Boolean satisfiability problem (SAT): given a boolean formula, determine if there exists an assignment of truth values to the variables that make the formula evaluate to true (Theorem 1.19).

- The Hamiltonian cycle problem: given a graph, determine if there exists a cycle that visits every vertex exactly once.
- The clique problem: given a graph and an integer k , determine if there exists a complete subgraph (clique) of size k .

Theorem 1.19 (Cook-Levin theorem) The Boolean satisfiability problem (SAT) is NP-complete.

Proof. TODO

□

1.5 The Complexity Class coNP

Given some language $L \subseteq \{0, 1\}^*$, we can define its complement language $\bar{L} = \{0, 1\}^* \setminus L$.

This allows us to define the complexity class **coNP** as follows.

$$\text{coNP} = \{L \subseteq \{0, 1\}^* \mid \bar{L} \in \text{NP}\}.$$

Similarly to how we defined **NP** in terms of certificates and verifiers, we can do the same for **coNP**.

Definition 1.20 (The class coNP) A language L is in **coNP** if there exists a polynomial $p(n)$ and a polynomial-time TM M such that $\forall x \in \{0, 1\}^*, x \in L$ iff $\forall u \in \{0, 1\}^{p(|x|)}$ such that $M(x, u) = 1$.

TODO: finish section on **coNP**.

Summary:

- The notion of *computability* is formalised using Turing Machines. Not all functions are computable, and the halting problem is a canonical example of an uncomputable function.
- We often work in the framework of *decision problems* and *languages*. These decision problems can be classified into *complexity classes*, based on the resources required to solve them. The most fundamental complexity classes are **P** and **NP** which represent problems that can be solved in polynomial time and problems whose solutions can be verified in polynomial time, respectively. The question of whether $P = NP$ is one of the most important open questions in computer science.
- Some problems in **NP** are **NP-hard**, meaning that they are the “hardest” problems in **NP**. If any **NP-hard** problem can be solved in polynomial time, then every problem in **NP** can be solved in polynomial time. The class **NP-complete** is the intersection of **NP** and **NP-hard**.
- $NP = P$ iff there exists a **NP-complete** problem that is in **P**.
- The Cook-Levin theorem shows that the Boolean satisfiability problem (SAT) is **NP-complete**.

2 The Polynomial Hierarchy

The *polynomial hierarchy*, or **PH**, is a hierarchy of complexity classes that generalizes the classes **P**, **NP**, and **coNP**. One way to define the polynomial hierarchy is using alternating quantifiers.

2.1 The Complexity Class Σ_2^P

To motivate the study of **PH**, consider the following problem. We know that the problem of determining whether a graph G has an independent set of size k is in **NP**.

$$\text{INDSET} = \{(G, k) \mid G \text{ has an independent set of size } \geq k\}.$$

This problem is in **NP** because we can clearly verify a certificate (the independent set) in polynomial time. However, consider now the “exact independent set” problem:

$$\text{EXACT-INDSET} = \{(G, k) \mid \text{The maximum independent set of } G \text{ is of size } k\}.$$

Here it doesn’t seem like we can verify a certificate in polynomial time. In particular, we need to verify that there *exists* an independent set of size k , and that *for all* independent sets, they are at most of size k .

To capture this kind of problem, we introduce the class Σ_2^P .

Definition 2.1 (The class Σ_2^P) A language L is in Σ_2^P if there exists a polynomial $p(n)$ and a polynomial-time TM M such that

$$\forall x \in \{0, 1\}^*, x \in L \text{ iff } \exists u \in \{0, 1\}^{p(|x|)} \forall v \in \{0, 1\}^{p(|x|)} \text{ such that } M(x, u, v) = 1.$$

Remark 2.2 Σ_2^P clearly contains both **NP** and **coNP** since we can construct M to ignore either u or v .

Example 2.3 The “exact independent set” problem described above is in Σ_2^P since we can take u to be a independent set of size k , and v to be any independent set. The verifier M can then check that u is indeed an independent set of size k and that v is not an independent set of size greater than k .

2.2 The Polynomial Hierarchy

To define the polynomial hierarchy, we can define a sequence of classes Σ_k^P and Π_k^P for $k \geq 0$ as follows:

Definition 2.4 (The classes Σ_k^P and Π_k^P)

- $\Sigma_0^P = \Pi_0^P = \mathbf{P}$.
- For $k \geq 1$, a language L is in Σ_k^P if there exists a polynomial $p(n)$ and a polynomial-time TM M such that $\forall x \in \{0, 1\}^*$,

$$x \in L \text{ iff } \exists u_1 \in \{0, 1\}^{p(|x|)} \forall u_2 \in \{0, 1\}^{p(|x|)} \dots Q_k u_k \in \{0, 1\}^{p(|x|)} M(x, u_1, \dots, u_k) = 1,$$

where Q_k is an existential quantifier if k is odd and a universal quantifier if k is even.

- For $k \geq 1$, a language L is in Π_k^P if there exists a polynomial $p(n)$ and a polynomial-time TM M such that $\forall x \in \{0, 1\}^*$,

$$x \in L \text{ iff } \forall u_1 \in \{0, 1\}^{p(|x|)} \exists u_2 \in \{0, 1\}^{p(|x|)} \dots Q_k u_k \in \{0, 1\}^{p(|x|)} M(x, u_1, \dots, u_k) = 1,$$

where Q_k is an existential quantifier if k is even and a universal quantifier if k is odd.

Note that $\Sigma_1^P = \mathbf{NP}$ and $\Pi_1^P = \mathbf{coNP}$.

We are now ready to define the polynomial hierarchy.

Definition 2.5 (The polynomial hierarchy **PH)** $\mathbf{PH} = \bigcup_{k \geq 0} \Sigma_k^P$.

Lemma 2.6 $\mathbf{PH} = \bigcup_{k \geq 0} \Pi_k^P$.

Proof. This follows from the fact that $\Sigma_k^P \subseteq \Pi_{k+1}^P \subseteq \Sigma_{k+2}^P$. □

2.3 Properties of the Polynomial Hierarchy

We believe that $\mathbf{NP} \neq \mathbf{coNP}$. We can generalize this to the conjecture that $\Sigma_k^P \neq \Sigma_{k+1}^P$ for all $k \geq 1$. This is often stated as *the polynomial hierarchy does not collapse*.

If for some k , $\Sigma_k^P = \Sigma_{k+1}^P$, then we say that *the polynomial hierarchy collapses to the k -th level*. This also implies that $\text{PH} = \Sigma_k^P = \bigcup_{k \geq 0} \Sigma_k^P$.

Theorem 2.7

- For every $k > 0$, if $\Sigma_k^P = \Pi_k^P$, then $\text{PH} = \Sigma_k^P$, i.e. the polynomial hierarchy collapses to the k -th level.
- If $\text{P} = \text{NP}$, then $\text{P} = \text{PH}$, i.e. the polynomial hierarchy collapses to P .

Proof. TODO □

Every level Σ_k^P and Π_k^P have complete problems. However, PH itself does not have any complete problems unless the polynomial hierarchy collapses.

Property 2.8 If there exists a language L that is PH -complete, then the polynomial hierarchy collapses to some level k .

Proof. If $L \in \text{PH}$ is PH -complete, then $L \in \Sigma_k^P$ for some finite k . But by definition of PH -completeness, all languages in PH are polynomial-time reducible to L . Therefore every language $L' \in \text{PH}$ is in Σ_k^P , which implies that PH collapses to the k -th level. □

Property 2.9 $\text{PH} \subseteq \text{PSPACE}$.

Corollary 2.9.1 $\text{PH} \neq \text{PSPACE}$ unless the polynomial hierarchy collapses to some level k .

Proof. PSPACE has complete problems, but PH does not have any complete problems unless it collapses to some level k . □

Remark 2.10 Similar to non-deterministic Turing Machines, we can also define *alternating Turing Machines*, or ATMs which can be used to model the polynomial hierarchy.

Recall that a non-deterministic Turing Machine accepts if there exists some sequence of non-deterministic choices that leads to acceptance. An alternating Turing Machine generalizes this by allowing both existential and universal states. Each state of the ATM is labeled as either existential or universal. The ATM accepts if there exists a sequence of choices that leads to acceptance, where the choices at existential states are existentially quantified and the choices at universal states are universally quantified.

We can then define PH in terms of ATMs who only have a fixed number of alternations between existential and universal states.

We can also define classes such as $\text{ATIME}(T(n))$ and $\text{AP} = \bigcup_{c \in \mathbb{N}} \text{ATIME}(n^c)$. In particular, these classes allow unbounded number of alternations between existential and universal quantifiers. We can show that $\text{AP} = \text{PSPACE}$.

Remark 2.11 Finally, it should be noted that the polynomial hierarchy can also be defined in terms of oracle Turing Machines. In particular, we have $\Sigma_k^P = \text{NP}^{\Sigma_{k-1}^P}$, and $\Pi_k^P = \text{coNP}^{\Sigma_{k-1}^P}$.

Summary:

- The polynomial hierarchy PH is a hierarchy of complexity classes that generalizes P, NP, and coNP. It is defined using alternating quantifiers, and can also be defined in terms of alternating Turing Machines.
- The polynomial hierarchy is believed to not collapse, i.e. $\Sigma_k^P \neq \Sigma_{k+1}^P$ for all $k \geq 1$. If it does collapse, then it collapses to some level k .
- Every level of the polynomial hierarchy has complete problems, but the polynomial hierarchy itself does not have any complete problems unless it collapses to some level k .

3 Boolean Circuits

4 Randomized Algorithms

We would like to investigate the power of Turing Machines that are allowed to make random coin tosses.

5 Quantum Computation

5.1 The Quantum Turing Machine Model

5.2 The Quantum Circuit Model

5.3 Grover's Search Algorithm

5.4 Simon's Algorithm

5.5 Shor's Algorithm

6 Average Case Complexity

6.1 Distributional Problems

6.2 Average Polynomial Time

6.2.1 The Complexity Class AvgP

A naive way to define average polynomial time is to say that a distributional problem (L, D) is average polynomial-time solvable if the *expected running time* of the algorithm A is polynomial in the size of the input. That is, if A satisfies the following condition:

$$\mathbb{E}_{x \sim D_n}[t_A(x)] = \text{poly}(n).$$

However, this definition does not seem to properly capture the notion of “efficient on average”. Consider the following example. Let (L, D) be a distributional problem that can be solved by some algorithm A in polynomial time for $1 - 2^{-n}$ of the inputs of length n , but takes exponential time 2^n for the remaining 2^{-n} inputs. This has expected running time:

$$\begin{aligned}\mathbb{E}_{x \sim D_n}[t_A(x)] &= (1 - 2^{-n}) \cdot \text{poly}(n) + 2^{-n} \cdot 2^n \\ &= \text{poly}(n) + O(1).\end{aligned}$$

Now consider some other algorithm B that solves the same problem but is quadratically slower than A on all inputs. That is, $t_B(x) = t_A(x)^2$. We would still expect B to be considered average polynomial-time solvable, since it is only quadratically slower than A . However, the expected running time of B is:

$$\begin{aligned}\mathbb{E}_{x \sim D_n}[t_B(x)] &= \mathbb{E}_{x \sim D_n}[t_A(x)^2] = (1 - 2^{-n}) \cdot \text{poly}(n)^2 + 2^{-n} \cdot 2^{2n} \\ &= O(2^n).\end{aligned}$$

So B would not be considered average polynomial-time solvable under this definition.

Instead, we can define average polynomial time in terms of a polynomial trade-off between the running time and fraction of inputs that the algorithm runs in polynomial time. This motivates the following definition of average polynomial time.

Definition 6.1 (Average polynomial time, trade-off definition) An algorithm A is said to run in *average polynomial time* for a distribution D if there exists some $\varepsilon > 0$ and some polynomial $p(n)$ such that for all t ,

$$\Pr_{x \sim D_n}[t_A(x) \geq t] \leq \frac{p(n)}{t^\varepsilon}.$$

Remark 6.2 This definition bears strong resemblance to Markov’s inequality, which states that for a non-negative random variable X and any $a > 0$, we have

$$\Pr(X \geq a) \leq \frac{\mathbb{E}[X]}{a}.$$

TODO: reason out what the intuitive significance of this connection is.

Lemma 6.3 An algorithm A runs in average polynomial time (according to Definition 6.1) iff it satisfies the following condition for some $\varepsilon > 0$:

$$\mathbb{E}_{x \sim D_n}[t_A(x)^\varepsilon] = O(n).$$

Furthermore, $O(n)$ can be replaced by any polynomial in n .

Proof. □

This provides an alternative definition of average polynomial time that is equivalent to Definition 6.1.

TODO: complete section

We can also define the notion of average polynomial time using *errorless heuristic schemes*.

Definition 6.4 (Errorless heuristic scheme, the class AvgP) An *errorless heuristic scheme* for a distributional problem (L, D) is an algorithm $A(x, \delta)$ that takes as input an instance x and a parameter $\delta \in (0, 1)$, and outputs either $L(x)$ or a special symbol $\perp$ indicating that it failed to solve the problem. The algorithm is allowed to fail with probability at most δ over the choice of x from the distribution D_n , where $n = |x|$.

In particular, we say that a distributional problem (L, D) is polynomial-time solvable on average if there exists an algorithm $A(x, \delta)$ such that:

- (Effectiveness) $A(x, \delta)$ runs in time $\text{poly}(\frac{x}{\delta})$.
- (Correctness) $\forall x, A(x, \delta) \in \{L(x), \perp\}$.
- (Usefulness) $\Pr_{x \sim D_n}(A(x, \delta) = \perp) < \delta$

Such a problem is said to be in the class AvgP.

6.2.2 The Complexity Class AvgBPP

We can relax the condition that the algorithm A is errorless, and allow it to make bounded errors. This gives us the class AvgBPP.

Definition 6.5 (The class AvgBPP) A distributional problem (L, D) is in AvgBPP if there exists a randomized algorithm $A(x, \delta)$ such that:

- (Effectiveness) $A(x, \delta)$ runs in time $\text{poly}(\frac{x}{\delta})$.
- (Correctness) $\forall x, \Pr(A(x, \delta) \notin \{L(x), \perp\}) \leq \frac{1}{4}$.
- (Usefulness) $\Pr_{x \sim D_n}(\Pr_A(A(x, \delta) = \perp) \geq \frac{1}{4}) < \delta$.

For the final condition, note that the outer probability is over the choice of x from the distribution D_n , while the inner probability is over the internal coin tosses of A .

The algorithm A is also called a *randomized errorless heuristic scheme* for (L, D) . In this case, “errorless” refers to the proportion of inputs for which the algorithm fails (which can be made arbitrarily small), rather than the internal randomness of the algorithm.

Remark 6.6 ([Its10]) The constant $\frac{1}{4}$ in the correctness and usefulness conditions are arbitrary and can be replaced by anything in the interval $(2^{-\Omega(n)}, \frac{1}{2})$.

Proof. Suppose (L, D) is in **AvgBPP** with an algorithm $A(x, \delta)$ satisfying definition Definition 6.5. We can construct a new algorithm $B(x, \delta)$ as follows: on input x , we run $A(x, \delta)$ for $n = |x|$ number of times and return the most frequent answer among $\{\perp, 0, 1\}$.

We now apply the Chernoff-Hoeffding bound which says that for $X_1, \dots, X_N$ independent and identically distributed random variables with $X_i \in [0, 1]$ and $\mathbb{E}[X_i] = \mu$, we have:

$$\Pr\left(\left|\frac{\sum_i X_i}{N} - \mu\right| \geq \varepsilon\right) \leq 2e^{-2n\varepsilon^2}.$$

Applying this with $X_i \sim A(x, \delta) = \perp$, we get that if $\Pr(A(x, \delta) = \perp) \leq \frac{1}{4}$ then $\Pr(B(x, \delta) = \perp) = g(n)$, where $g(n) = 2^{-\Omega(n)}$ is some function.

Therefore, we have

$$\Pr_{x \sim D_n} \left(\Pr_B(B(x, \delta) = \perp) \geq g(n) \right) \leq \Pr_{x \sim D_n} \left(\Pr_A(A(x, \delta) = \perp) \geq \frac{1}{4} \right) < \delta. \quad \square$$

6.2.3 The Complexity Class **distNP**

TODO: $\text{distNP} := (\text{NP}, \text{PComp})$.

6.3 A Complete Problem for **distNP**

We introduce the *bounded halting problem* which is complete for problems in **distNP**.

$$\text{BH} = \{(M, x, 1^t) \mid M \text{ is a Turing Machine that accepts } x \text{ in time } \leq t\}.$$

We pair this with the *approximately* uniform distribution U^{BH} and claim that the pair $(\text{BH}, U^{\text{BH}})$ is **distNP**-complete.

TODO: precise statement and theorem.

Proof. TODO □

6.4 A Complete Problem for $(\text{NP}, \text{PSamp})$

6.5 A Complete Problem for **AvgBPP**

There is currently no known complete problem for **BPP** (although of course if $\text{P} = \text{BPP}$, then every **P**-complete problem is also **BPP**-complete). So somewhat surprisingly, we do know of a complete problem for the distributional class $(\text{AvgBPP}, \text{PSamp})^3$, due to [Its10].

Recall that the main obstacle to finding a complete problem for **BPP** is the absence of an efficient enumeration of all bounded error probabilistic Turing Machines. One way around this is to consider **promise-BPP** instead, for which a complete problem is known.

³The notation $(\text{AvgBPP}, \text{PSamp})$ denotes the distributional complexity class containing all (L, D) in **AvgBPP** where $D \in \text{PSamp}$.

We will now describe a complete distributional problem (C, R) for $(\text{AvgBPP}, \text{PSamp})$.

Let $M^{\leq m}$ denote the probabilistic Turing Machine M that is restricted to run for at most m steps. If M has not halted after m steps, it outputs the failure symbol $\perp$.

Let $\text{freq}(M(x))$ denote the most frequent output of M on input x over all possible internal coin tosses of M , and let $\text{prob}(M(x))$ be the probability of $\text{freq}(M(x))$.

All strings in C have the form $(M, y, 1^m, b, S, 1^s)$, where:

- M is a probabilistic Turing Machine,
- y is an input string to M ,
- m is the maximum steps that M can take, with $m > |y_0| + N_0$ for some sufficiently large constant N_0 ,
- $b \in \{0, 1\}$ is the answer that would be used instead of $\perp$ for the machine $M^{\leq m}$,
- S is the encoding of a sampler
- 1^s is the maximum steps that S can take.

The language C is defined by the following characteristic function:

$$C(M, y, 1^m, b, S, 1^s) = \begin{cases} b & \text{if } \Pr_M(M^{\leq m}(y) = \perp) \geq \frac{1}{4}, \\ \text{freq}(M^{\leq m}(y)) & \text{otherwise.} \end{cases}$$

In particular, this defines a (deterministic) procedure that needs to check every possible sequence of coin tosses of $M^{\leq m}$ to determine the most frequent output.

The central idea in constructing the sampler R is to assign to machines $M^{\leq m}$ that do not have a clear majority answer a very small probability. This ensures that C , which is defined using expensive exhaustive computation, is efficient randomized on average.

Summary:

- Average case complexity is a framework for studying the complexity of problems under some distribution over the inputs. The central objects of study are *distributional problems* (L, D) , where L is a language and D is an ensemble of distributions over the inputs.
- TODO: finish summary

7 Quantum Average Case Complexity

8 Examples of Quantum Average Case Complexity

8.1 Random Hamiltonian Simulation

Bibliography

- [AB09] S. Arora and B. Barak, *Computational Complexity*. 2009.
- [BT21] A. Bogdanov and L. Trevisan, “Average-Case Complexity,” no. arXiv:cs/0606037, Aug. 2021, doi: 10.48550/arXiv.cs/0606037.

- [Its10] D. Itsykson, “Structural complexity of AvgBPP,” *Annals of Pure and Applied Logic*, vol. 162, no. 3, pp. 213–223, Dec. 2010, doi: 10.1016/j.apal.2010.09.006.