1 Introduction

Recall last lecture:

- Circuits: non-uniform computational model; the size of a circuit is the number of gates.
- A language $L$ is in $\text{SIZE}(T(n))$ if there exists a $T(n)$-size circuit family $\{C_n\}_{n \in \mathbb{Z}}$ such that for every $x \in \{0,1\}^n \Leftrightarrow C_n(x) = 1$.
- For every language $L$, $L \in \text{SIZE}(O(2^n))$.
- At the same time almost all languages require circuits of size $\approx 2^n/n$. This followed from a counting argument.
- We let $P/poly := \bigcup c \text{SIZE}(n^c)$.
- Kind of surprising at first, we showed (in exercise session) that $P/poly$ contains undecidable languages. The key reason is that it is a non-uniform computational model (we have a different circuit for each input length $n \in \mathbb{N}$).
- We also showed that $P \subseteq P/poly$ by showing that for any language $L \in P$ (and any $n \in \mathbb{N}$) we can in polynomial time construct a circuit $C_n$ of polynomial size such that $C_n(x) = 1 \Leftrightarrow x \in L$ for all $x \in \{0,1\}^n$.
- Finally, we gave an alternative proof of the Cook-Levin Theorem using circuits. Specifically, we showed that that an (Oblivious) TM machine that takes $T(n)$ steps can be written as a $O(T(n))$ sized circuit in time $O(T(n))$.

Today:

- We discuss randomized computing. What problems have better randomized algorithms than deterministic ones?
- Different randomized complexity classes.
- Connecting randomization to circuits (Adleman’s Theorem).
2 Randomized computation

As this is a complexity course, we wish to understand the power of computing when we are allowed to flip a coin.

Let us first define randomized computation formally using probabilistic TMs.

**Definition 1** A **probabilistic Turing machine (PTM)** is a TM with two transition functions \( \delta_0 \) and \( \delta_1 \). To execute a PTM \( M \) on an input \( x \), we choose in each step with probability \( \frac{1}{2} \) to apply the transition function \( \delta_0 \) and with probability \( \frac{1}{2} \) to apply the transition function \( \delta_1 \). The machine only outputs 1 (“Accept”) or 0 (“Reject”).

This definition is a little abstract at first. It is indeed hard to design algorithms when thinking in this low level abstraction. The following less formal but intuitive definition can be helpful: A randomized algorithm is an algorithm that has the ability to toss coins.

2.1 Some examples of randomized algorithms

2.1.1 Finding a median

Given \( n \) integers \( a_1, \ldots, a_n \) how do you find the median by a fast algorithm?

The standard way is to solve the following slightly more general problem: Given integers \( a_1, \ldots, a_n \) and \( 1 \leq k \leq n \), find the \( k \) largest integer.

The randomized algorithm is recursive and works as follows

1. Pick a random \( i \in [n] \) and let \( x = a_i \).
2. Scan the list \( \{a_1, \ldots, a_n\} \) and count the number \( m \) of \( a_i \)'s such that \( a_i \leq x \).
3. If \( m = k \), then output \( x \).
4. Otherwise, if \( m > k \), then copy to a new list \( L \) all elements such that \( a_i < x \) and find the \( k \):th largest integer in \( L \) (which is a smaller instance).
5. Otherwise (if \( m < k \)) copy to a new list \( H \) all elements such that \( a_i > x \) and find \( k - m \):th largest integer in \( H \) (which again is a smaller instance).

An analysis similar to the analysis of QuickSort shows that this algorithm runs in expected linear time. There is also a deterministic algorithm that runs in linear time but it is much more involved and harder/slower to implement.

2.1.2 Polynomial identity testing

How do you efficiently check whether two polynomials \( P \) and \( Q \) are identical?

This is equivalent to checking whether a single polynomial is equal to zero, i.e., check whether \( P - Q \equiv 0 \).

We assume the polynomials are given implicitly (think determinant, permanent). Note that e.g. the polynomial \( \prod_{i=1}^{n}(1 + x_i) \) can be evaluated efficiently but has \( 2^n \) many terms. This means that to check whether a polynomial is equivalent to 0 we cannot afford to write out all the terms.

The simple randomized algorithm is based on the Schwartz-Zippel Lemma:

**Lemma 2** Let \( p(x_1, \ldots, x_m) \) be a nonzero polynomial of total degree at most \( d \). Let \( S \) be a finite set of integers. Then, if \( a_1, \ldots, a_m \) are randomly chosen from \( S \), then

\[
\Pr[p(a_1, a_2, \ldots, a_m) \neq 0] \geq 1 - \frac{d}{|S|}.
\]
This suggests the following simple algorithm to check whether a polynomial $p$ of degree $d$ is equivalent to 0:

1. Choose $a_1, \ldots, a_m$ at random from $\{1, \ldots, 3d\}$.
2. Output that the polynomial is equivalent to 0 if $p(a_1, \ldots, a_m) = 0$.

Note that the algorithm is always correct if $p \equiv 0$. If $p \not\equiv 0$ then by Lemma 2 it succeeds with probability at least $2/3$. This probability can be boosted by repeating the algorithm more times.

It remains a major open problem to find an efficient deterministic algorithm for polynomial identity testing.

3 Two-sided, One-sided and Zero-Sided Error

In our examples, we saw different types of randomized algorithms. One that always reported a true answer with expected polynomial time running time. Another that always runs in polynomial time but could with a small probability output the wrong answer. Let’s make these differences formal.

**Definition 3 (Two-sided error)** Let $\text{BPTIME}(T(n))$ be the class of languages that contain language $L$ if there is a probabilistic TM $M$ running in time $T(n)$ satisfying

$$\Pr[M(x) = L(x)] \geq 2/3 \quad \text{for every } x \in \{0,1\}^*.$$ 

Let $\text{BPP} = \cup_c \text{BPTIME}(n^c)$.

**Definition 4 (One-sided error)** $\text{RTIME}(T(n))$ contains every language $L$ for which there is a probabilistic TM $M$ running in $T(n)$ time such that

$$x \in L \Rightarrow \Pr[M(x) = 1] \geq 2/3$$

$$x \not\in L \Rightarrow \Pr[M(x) = 0] = 1.$$ 

Let $\text{RP} = \cup_c \text{RTIME}(n^c)$.

We also define the class capturing the other one-sided error (on inputs not in the language) as $\text{coRP} = \{L : \overline{L} \in \text{RP}\}$.

- Note that polynomial identity testing is in $\text{coRP}$.

**Definition 5 (Zero-sided error)** The class $\text{ZTIME}(T(n))$ contains all the languages for which there is a machine $M$ that runs in expected time $O(T(n))$ such that for every input $x$, whenever $M$ halts on $x$, we have $M(x) = L(x)$.

Define $\text{ZPP} = \cup_c \text{ZTIME}(n^c)$.

- Finding the median is morally in $\text{ZPP}$. (Only morally, since we didn’t define it as a decision problem.)

4 Error reduction

- You may have wondered why the constant $2/3$ came up in the definition of $\text{BPTIME}$ and $\text{RTIME}$.

- Well it is an arbitrary choice and it doesn’t really matter because we can always improve our error probability by repetition.

To see that let us prove the following:
Lemma 6 Let $L \in \text{BPP}$. Then there is a polynomial time probabilistic TM $M$ such that

$$\Pr[M(x) = L(x)] \geq 1 - \frac{1}{2|x|+1}.$$  

Proof

• As $L$ is in $\text{BPP}$, there is a polynomial time probabilistic TM $M'$ such that $\Pr[M'(x) = L(x)] = 2/3$.

• The machine $M$ simply does the following:

  For every input $x \in \{0,1\}^*$, run $M'(x)$ for $k = 100|x|$ times obtaining outputs $y_1,\ldots,y_k \in \{0,1\}$. If the majority of these outputs is 1, then output 1; otherwise, output 0.

• To analyze $M$, define for every $i \in [k]$ the random indicator variable $X_i$ to equal 1 if $y_i = L(x)$ and to equal 0 otherwise.

• Note that $X_1,\ldots,X_k$ are independent Boolean random variables with $E[X_i] = \Pr[X_i = 1] = 2/3$. Note also that our algorithm returns the right answer if $X_1 + X_2 + \cdots + X_k > k/2$.

• Let $\mu = E[X_1 + \ldots X_k] = 2k/3$. Now applying a Chernoff bound yields that

  $$\Pr[M \text{ outputs incorrect answer}] \leq \Pr\left[\left|\sum_{i=1}^{k} X_i - \mu\right| \geq \frac{1}{4}\mu\right]$$

  $$< e^{-\Omega(\mu)} < 2^{-(n+1)},$$

  by the choice of $k$.

• Hence, we have defined a polynomial time probabilistic TM $M$ such that

  $$\Pr[M(x) = L(x)] \geq 1 - \frac{1}{2^{n+1}} \quad \text{for all } x \in \{0,1\}^*.$$  

\[\square\]

Notice that we can further improve the error probability by increasing the number of repetitions.

5 Adleman’s Theorem: $\text{BPP} \subseteq \text{P/poly}$

• It is believed that $\text{P} = \text{BPP}$. (There are several good reasons. Perhaps the best one is by Impagliazzo and Wigderson: if SAT does not have circuits of size $2^{o(n)}$ then $\text{P} = \text{BPP}$.)

• Therefore, we should expect that $\text{BPP} \subseteq \text{P/poly}$ since $\text{P} \subseteq \text{P/poly}$.

Theorem 7 (Adleman’78) $\text{BPP} \subseteq \text{P/poly}$.

Proof

• Let $L \in \text{BPP}$. By Lemma 6, there exists a probabilistic TM $M$ such that on inputs of length $n$ satisfies

  $$\Pr[M(x) = L(x)] \geq 1 - \frac{1}{2^n+1}.$$  

4
Here, the probability is over the random/probabilistic choices of $M$. If we let $M(x, r)$ denote the execution of $M$ with random choices $r \in \{0, 1\}^{\text{poly}(n)}$. Then we can write this probability as
\[
\Pr_r[M(x, r) = L(x)] \geq 1 - \frac{1}{2^{n+1}}, \quad \text{or equivalently as} \quad \Pr_r[M(x, r) \neq L(x)] < \frac{1}{2^{n+1}}.
\]

Now consider all inputs $x \in \{0, 1\}^n$ of length $n$. Then a simple union bound yields
\[
\Pr_r[M(x, r) \neq L(x) \text{ for one } x \in \{0, 1\}^n] \leq \sum_{x \in \{0, 1\}^n} \Pr_r[M(x, r) \neq L(x)] < 2^n \cdot \frac{1}{2^{n+1}} = 1/2.
\]

This means that for each $n \in \mathbb{N}$, there exist random choices $r_n \in \{0, 1\}^{\text{poly}(n)}$ such that
\[
M(r_n, x) = L(x) \quad \text{for all } x \in \{0, 1\}^n.
\]

As $M(r_n, \cdot)$ is a deterministic polynomial time execution we can write down a polynomial size circuit $C_n$ (in the same way we did last lecture) so that
\[
C_n(x) = M(r_n, x) = L(x) \quad \text{for all } x \in \{0, 1\}^n.
\]

It follows that $L$ has a polynomial sized circuit family $\{C_n\}_{n \in \mathbb{N}}$, which completes the proof.

6 Some comments

- Randomization seems to help when designing algorithms from an intuitive point of view.

- However, it is believed that it does not change the power of polynomial time computation, i.e., that $\mathbf{P} = \mathbf{BPP}$.

- It is actually open to prove that $\mathbf{BPTIME}(n) \subsetneq \mathbf{BPTIME}(n^{100})$ and $\mathbf{BPP} \subsetneq \mathbf{NEXP}$.

- Moreover, it is known that in order to prove $\mathbf{P} = \mathbf{BPP}$ one also has to prove new interesting circuit lower bounds.

- We are thus quite far from proving $\mathbf{P} = \mathbf{BPP}$ but at least we could show that $\mathbf{BPP}$ has polynomial size circuits.

- $\mathbf{P/poly}$ is a mysterious class. We know that $\mathbf{NP} \not\subseteq \mathbf{P/poly}$ would imply $\mathbf{P} \neq \mathbf{NP}$. However, is that reasonable to expect as $\mathbf{P/poly}$ also contains undecidable languages? In the next lecture, we will show that this is indeed reasonable to expect because otherwise something called the polynomial hierarchy collapses (Karp-Lipton Theorem).

- This gives the hope that we can resolve the $\mathbf{P}$ vs $\mathbf{NP}$ question by studying circuits. However, Razborov showed that such a proof would need to not be “natural”.

- We then start with interactive proofs.