CSE200: Complexity theory
Randomized algorithms

Shachar Lovett

September 7, 2021

1 Randomized Turing machines

So far we considered several computational resources: time, space, nondeterminism and nonuniformity. Now we turn to study randomness. Randomness is a useful mechanism that often simplifies algorithms for many problems. First, we formally define the model of a randomized Turing machine.

A randomized Turing machine is a Turing machine which allows to "toss coins" during the computation. There are two equivalent ways to formalize this notion. The first is an implicit definition, where we assume that the TM has two transition functions δ_1, δ_2 instead of just one, and at each step it uses one of them randomly and uniformly. The other definition, which is sometimes easier to argue about, is an explicit definition. We assume that the randomness is given on a separate "random tape", which is filled with random bits. The random tape is read-only, and the head on the random tape can only move to the right. This means that each random bit can only by read once.

Let M be a randomized TM, which for simplicity we assume outputs one bit, 1 (accept) or 0 (reject). Given an input x, we denote by $M(x) \in \{0,1\}$ the output of the TM, which is a random variable. We say that M decides a language $L \subset \{0,1\}^*$ if the following holds for all inputs $x \in \{0,1\}^*$:

- If $x \in L$ then $\Pr[M(x) = 1] \geq 2/3$.
- If $x \notin L$ then $\Pr[M(x) = 0] \geq 2/3$.

If we define $L(x) = 1_{x \in L}$ then we can write this succinctly as

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

We will see later that the specific choice of $2/3$ is immaterial, and it can be replaced with any constant in the open interval $(1/2, 1)$ without any major changes.

Let $T : \mathbb{N} \to \mathbb{N}$ be a time bound. Given a randomized TM M, we say that M runs in time T if for every input x, $M(x)$ terminates after at most $T(|x|)$ steps with probability one.

We define BPTIME(T) to be the set of languages L that are decided by a randomized TM running in time T. Similar to the definition of P, we define BPP to be the class of languages that are decidable by randomized TMs running in polynomial time:

$$\text{BPP} = \cup_{c \geq 1} \text{BPTIME}(n^c).$$

2 Error reduction

In this section, we show that the specific choice of 2/3 in the definition of BPTIME and BPP is arbitrary, and can be replaced by any other constant between 1/2 and 1. Let $1/2 < p < 1$ be a parameter, and define $\text{BPTIME}_p(T)$ to be the class of languages L, for which there exists a randomized TM M such that

$$\Pr[M(x) = L(x)] \geq p \qquad \forall x \in \{0,1\}^*.$$

Recall that we defined $\text{BPTIME}(T) = \text{BPTIME}_{2/3}(T)$.

Lemma 2.1. *Let $0 < \varepsilon, \delta < 1/2$ and let $T : \mathbb{N} \to \mathbb{N}$ be a time bound. Assume that $L \in BPTIME_{1/2+\varepsilon}(T)$. Then $L \in BPTIME_{1-\delta}(mT)$ where*

$$m = O\left(\log(1/\varepsilon)/\delta^2\right).$$

The proof of Lemma 2.1 is based on the Chernoff-Hoeffding bound. Let $X_1, \ldots, X_m \in \{0,1\}$ be independent random variables and let $S = \sum X_i$ denote their sum. Then S is close to its expectation with high probability. For any $\varepsilon > 0$ it holds that

$$\Pr[|S - \mathbb{E}[S]| \geq \varepsilon m] \leq 2e^{-2\varepsilon^2 m}.$$

Proof of Lemma 2.1. Let $M \in \text{BPTIME}(T)$ be a randomized TM such that for every input x

$$\Pr[M(x) = L(x)] \geq 1/2 + \varepsilon.$$

We reduce the error by running M several times and taking the majority answer. Let $M'(x)$ be a randomized TM which runs $M(x)$ iteratively m times and returns the majority answer. We claim that

$$\Pr[M'(x) = L(x)] \geq 1 - \delta.$$

To prove this, fix an input x and let $X_1, \ldots, X_m$ be indicator variables where $X_i = 1$ if in the i-th instance of running $M(x)$, the answer was equal to $L(x)$. We have that $X_1, \ldots, X_m$ are independent random variables in $\{0,1\}$, and $\Pr[X_i = 1] \geq 1/2 + \varepsilon$ for each i. Note that $M'(x) \neq L(x)$ if $S = \sum X_i$ satisfies $S \leq m/2$, and that $\mathbb{E}[S] \geq (1/2 + \varepsilon)m$. Thus

$$\Pr[M'(x) \neq L(x)] = \Pr[S \leq m/2] \leq \Pr[|S - \mathbb{E}[S]| \geq \varepsilon m] \leq 2e^{-2\varepsilon^2 m} \leq \delta$$

by our choice of $m = O(\log(1/\delta)/\varepsilon^2)$. □

We can extend the definition of BPTIME to allow $p = p(n)$ to be a function which depends on the input length $n = |x|$. The same proof of Lemma 2.1 shows that if $\varepsilon, \delta : \mathbb{N} \to (0, 1/2)$ are functions, then

$$\mathrm{BPTIME}_{1/2+\varepsilon(n)}(T(n)) \subset \mathrm{BPTIME}_{1-\delta(n)}(m(n)T(n))$$

for

$$m(n) = O(\log(1/\delta(n))/\varepsilon(n)^2).$$

In particular, let $\mathrm{BPP}_{p(n)} = \cup_{c \geq 1} \mathrm{BPTIME}_{p(n)}(n^c)$. Recall that $\mathrm{BPP} = \mathrm{BPP}_{2/3}$. We obtain that BPP can equivalently be defined by randomized TMs that compute the correct answer with probability as low as $1/2 + 1/\mathrm{poly}(n)$, or as large as $1 - \exp(-\mathrm{poly}(n))$.

Corollary 2.2. *For any $a > 0$ we have*

$$BPP = BPP_{1/2+n^{-a}} = BPP_{1-\exp(-n^a)}.$$

3 Examples of randomized algorithms

In this section we give some examples of randomized algorithms, and study them with a complexity viewpoint. Our main goal is to demonstrate the power of randomness in algorithm design.

3.1 Finding the median

Let $A = \{a_1, \ldots, a_n\}$ be a set of numbers, and consider the problem of finding the median of the numbers. More generally, for an index $k \in [n]$, consider the problem of finding the k-th smallest number in the set, where the median corresponds to $k = n/2$. The following is a randomized algorithm that solves this problem.

Algorithm 1: Element(A, k)

Input: set $A = \{a_1, \ldots, a_n\}$; index $k \in [n]$
Output: k-th smallest number

```
1 Choose random p ∈ [n] (a pivot)
2 Compute L = {i : a_i ≤ a_p} and H = {i : a_i > a_p}
3 if |L| ≥ k then
4 |   return Element(L, k)
5 else
6 |   return Element(H, k − |L|)
7 end
```

It is straightforward to see that Element(A, k) always returns the correct answer. What is less clear is how fast does it take. We will bound the expected runtime.

Theorem 3.1. *For any set A of size $|A| = n$ and any $k \in [n]$, the expected running time of Element(A, k) is $O(n)$.*

Proof. Let $T(n,k)$ denote the expected running time for sets of size $|A| = n$ and index k. Our goal will be to prove by induction on n that $T(n,k) \leq Cn$ for some constant $C > 0$.

The time it takes to compute L, H in each iteration is cn for some constant c. If it happens that $|L| \geq k$ then we have

$$T(n,k) = cn + T(|L|, k)$$

and if $|L| < k$ then we have

$$T(n,k) = cn + T(n - |L|, k - |L|).$$

Note that as we choose the pivot randomly, we have that $|L| \in \{1, \ldots, n\}$ is uniform. Thus we obtain the following recursion:

$$T(n,k) = cn + \frac{1}{n}\left(\sum_{\ell=k}^{n} T(\ell,k) + \sum_{\ell=1}^{k-1} T(n-\ell, k-\ell)\right).$$

Applying the inductive hypothesis gives

$$\begin{aligned} T(n,k) &\leq cn + \frac{C}{n}\left(\sum_{\ell=k}^{n} \ell + \sum_{\ell=1}^{k-1}(n-\ell)\right) \\ &\leq cn + \frac{C}{n}\left(\sum_{\ell=1}^{n} \max(\ell, n-\ell)\right) \end{aligned}$$

The inner term equals

$$\sum_{\ell=1}^{n} \max(\ell, n-\ell) = 2\sum_{\ell=\lceil n/2 \rceil}^{n} \ell \leq n(n+1) - n/2(n/2-1) \leq (3/4)n^2 + 2n.$$

Thus we have

$$T(n,k) \leq cn + (3/4)Cn + 2C.$$

For large enough C, n we get that $T(n,k) \leq Cn$, as claimed. □

3.2 Primality testing

Let $n \geq 2$ be a positive integer. Our goal will be to test if n is a prime number or a composite. A basic approach is to use Fermat's little theorem.

Theorem 3.2 (Fermat's little theorem)**.** *Let p be a prime number. Then for any integer $1 \leq a \leq p-1$ it holds that $a^{p-1} \equiv 1 \pmod{p}$.*

If this was an if-and-only-if criteria, then it would suggest a very natural randomized algorithm to test if a number n is prime or not. Choose a random number $a \in [n-1]$ and check if $a^{n-1} \equiv 1 \pmod{n}$. Unfortunately, there are numbers n which are composite, but for which this condition also holds. Miller [5] found a way to fix this problem by somewhat modifying the test, but his algorithm correctness relied on the Riemann hypothesis. Rabin [6] found a way to remove the reliance on the Riemann conjecture by adding randomness to the algorithm. This is now known as the Miller-Rabin primality test.

Algorithm 2: IsPrime(n)

Input: Integer $n \geq 2$
Output: True if n is prime, False otherwise

```
Choose random a ∈ [n − 1]
if a^{n−1} ≢ 1 (mod n) then
    return False
end
Let r = n − 1
while a^r ≡ 1 (mod n) and r is even do
    Update r ← r/2
end
if a^r ≡ 1 (mod n) or a^r ≡ −1 (mod n) then
    return True
else
    return False
end
```

Here is some insight to why the algorithm works (we omit the full proof as it requires a bit of number theory, which is out of scope in this class). Assume n is prime. Then by Fermat's little theorem, $a^{n-1} \equiv 1 \pmod{n}$. Let r be an even number such that $a^r \equiv 1 \pmod{n}$. Then as n is prime, the numbers modulo n form a finite field, and we have that $a^{r/2}$ is root of 1, namely 1 or -1 modulo n. One can show that if n is not prime then the chance of a test passing for a random a is at most $3/4$.

In 2004, in a breakthrough work, Agrawal, Kayal and Saxena [2] found a deterministic polynomial-time algorithm for primality testing. The reason this was unexpected is that many believed that such a result would need a breakthrough in number theory, such as proving the Riemann hypothesis or a related result. However, as it turns out this was not needed, and their algorithm does not use fancy mathematics, but instead uses basic algebra in an ingenious way.

3.3 Reachability in undirected graphs

Let $G = (V, E)$ be an undirected graph on $|V| = n$ vertices. Two nodes $s, t \in V$ are connected if there exists a path between s and t. The classical algorithms to check if two nodes are connected are running either BFS or DFS from s, and checking if we ever reach t. These algorithms run in time at most $O(|V| + |E|) = O(n^2)$ and require space $O(n)$, as

they need to mark for each node if it was visited or not.

It turns out that one can construct algorithms which run in much lower space - $O(\log n)$ - if we are willing to use randomness. The algorithm is based on random walks. A random walk starts at a node $v \in V$, and at each step moves from v to a random neighbour u of v. The crucial theorem is that random walks of length $O(n^3)$ are likely to visit any node in the connected component of the start node. This suggests the following randomized algorithm for reachability.

Algorithm 3: Reachability(G, s, t)

Input: Undirected graph $G = (V, E)$, nodes $s, t \in V$
Output: True if s, t are connected, False otherwise

```
Set v = s, T = O(n^3)
for T steps do
    if v = t then
        return True
    end
    Let u be a random neighbour of v in G
    Update v ← u
end
return False
```

Let us explain why this algorithm runs in space $O(\log n)$. The algorithm at every point remembers a constant number of nodes in the work memory, as the input is stored in the read-only input memory. One can check that finding a random neighbour of a node can be implemented in $O(\log n)$ space as well. One simple solution, which runs in expected linear time, is at each time to choose a random node, and then accept it if it is a neighbour of v.

A natural question is whether this random walk algorithm also works for directed graphs. Unfortunately, the answer is negative. A simple example showing that is the following graph. The nodes are $V = \{0, \ldots, n\}$ and the edges are $E = \{(i-1, i) : i \in [n]\} \cup \{(i, 0) : i \in [n]\}$. It can be verified that there is a directed path from $s = 0$ to $t = n$, but that a random walk would take on expectation about 2^n steps to reach from 0 to n.

Finally, let us remark that Reingold [7] found a way to de-randomize the random walk algorithm, and this is how he proved that reachability in undirected graphs is in LOGSPACE.

3.4 Schöning randomized 3-SAT algorithm

Let $\varphi(x_1, \ldots, x_n) = C_1(x) \wedge \ldots \wedge C_m(x)$ be a 3-CNF formula. The 3-SAT problem (which is NP-complete) is to determine if there is a satisfying assignment for φ. Namely, an $x \in \{0,1\}^n$ such that $\varphi(x) = 1$. A trivial algorithm is to test all possible assignments, which takes time about 2^n. Schöning [9] found a randomized algorithm which solves the problem faster (but still in exponential time). As we will see, his algorithm has expected runtime of about $(4/3)^n$. The algorithm is based on "guided" random walks of possible solutions.

Algorithm 4: 3SAT-RandomWalk(φ)

Input: 3-CNF φ
Output: True or False

Choose $a \in \{0,1\}^n$ randomly
Repeat at most $3n$ times:

1. If $\varphi(a) = 1$ then **return** True
2. Else, pick a clause C_i such that $C_i(a) = 0$
3. Let x_j be a random variable in C_i
4. Flip the value of a_j

The runtime of 3SAT-RandomWalk is $O(n)$, so we don't hope that it will find a solution for φ with high probability. The main result is that the probability that it finds a solution is significantly higher than 2^{-n}, which is the probability that a random guess would work.

Theorem 3.3. *Assume φ is a satisfiable 3-SAT. Then 3SAT-RandomWalk finds a solution with probability at least $(3/4)^n$.*

Hence, if φ is satisfiable and we run 3SAT-RandomWalk $(4/3)^n$ times, then with high probability we will find a solution. The total runtime of the algorithm is then $O(n(4/3)^n)$.

3.5 Finding simple paths in graphs

Let $G = (V, E)$ be an undirected graph. We consider the problem of deciding if G has a simple path of length k. That is, k distinct nodes $v_1, \ldots, v_k$ such that all edges (v_i, v_{i+1}) exist. One solution is to try all options, which would take time n^k. Instead, we will show how randomness can be used to improve the runtime to $\text{poly}(n, 2^k)$.

The basic idea is to use random colorings to prune out the search space. A random k-coloring of G is a random map $\chi : V \to [k]$, which assigns each vertex v a uniform color in $1, \ldots, k$. Given a simple path $P = (v_1, \ldots, v_k)$, we say that P is colorful if its nodes obtain all k distinct colors. The first step is to show that this happens with a noticeable probability for a random coloring.

Claim 3.4. *Let $\chi : V \to [k]$ be a random coloring. Let $P = (v_1, \ldots, v_k)$ be a simple path of length k. Then*

$$\Pr_{\chi}[P \text{ is colorful }] \geq e^{-k}.$$

Proof. The number of possible colorings of the nodes of P is k^k. The number of choices where all the k colors are distinct is $k!$. So the probability that all the k colors are distinct is

$$\Pr_{\chi}[P \text{ is colorful }] = \frac{k!}{k^k} \geq \frac{(k/e)^k}{k^k} = e^{-k}.$$

□

The main idea will to be choose a random coloring, and then to try and find a simple path where all the k colors are distinct. Claim 3.4 shows that if G indeed has a simple path of length k, then we will find it with noticeable probability in this way. Note that a colorful path of length k must be a simple path, as the nodes cannot repeat (they have different colors).

Claim 3.5. *Let $G = (V, E)$ be a graph and $\chi : V \to [k]$ be a coloring. We can detect if G has a colorful path of length k in time poly$(n, 2^k)$.*

Proof. Consider the following variables. Given a nonempty set $A \subset [k]$ and a node $v \in V$, define $X_{A,v} \in \{0, 1\}$ to be 1 if there is a simple path in G of length $|A|$, whose nodes obtain the colors A, and whose last node is v.

We claim that we can compute $X_{A,v}$ for all A, v using a dynamic program. For $|A| = 1$, namely $A = \{a\}$, we set $X_{\{a\},v} = 1$ if $\chi(v) = a$. Assume that we already computed $X_{A,v}$ for all sets A of size ℓ. Let A be a set of size $\ell + 1$ and let $v \in V$. Our goal is to compute $X_{A,v}$. Let $c = \chi(v)$. If $c \notin A$ then $X_{A,v} = 0$. Otherwise, let $A' = A \setminus \{c\}$. Then $X_{A,v} = 1$ if $X_{A',u} = 1$ for some $u \sim v$. That is:

$$X_{A,v} = \bigvee_{u \sim v} X_{A',u}.$$

We can compute each $X_{A,v}$ in time $O(n)$, and hence all of them in time $O(n^2 2^k)$. Finally, G has a colorful path of length k if $X_{[k],v} = 1$ for some $v \in V$. □

The final algorithm is to choose $2^{O(k)}$ random colorings, and for each one to check if there is a simple path in G which is colorful under the coloring. Claim 3.4 shows that if G has a simple path of length k, then we will find one with high probability. The total runtime is poly$(n, 2^k)$ as claimed.

3.6 Polynomial identity testing

Let $p(x) = p(x_1, \ldots, x_n)$ be a polynomial with real-valued coefficients, of some bounded degree d. Given a vector $e = (e_1, \ldots, e_n)$ of powers let use shorthand $|e| = \sum e_i$

$$x^e = x_1^{e_1} \cdots x_n^{e_n}.$$

A general degree d polynomial can be expressed as

$$p(x) = \sum_{e:|e| \leq d} p_e x^e.$$

In other words, the degree of p is the maximal $|e|$ such that the coefficient p_e is nonzero.

Our goal is to check if p is the identically zero polynomial or not. Of course, if we were given the all the coefficients $\{p_e : |e| \leq d\}$ then we can check if they are all equal to zero. The only problem is that the number of coefficients is exponentially big. This raises the question

- how are we given the polynomial? we will see an example soon, but for now assume that p is given in some succinct form, that allows us to evaluate fast $p(a)$ on any input $a \in \mathbb{R}^n$. We will see that by evaluating a polynomial on random inputs, we can test efficiently if $p \equiv 0$ or not. The algorithm was discovered independently by Demillo and Lipton [3], Schwartz [10] and Zippel [12].

Lemma 3.6 (Demillo-Lipton-Schwartz-Zippel)**.** *Let $p(x) = p(x_1, \ldots, x_n)$ be a nonzero polynomial of degree d. Let $S \subset \mathbb{R}$ be any set of $|S| > d$. Let $a_1, \ldots, a_n \in S$ be chosen uniformly and independently. Then:*

$$\Pr[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}.$$

This gives a randomized algorithm to check if a low-degree polynomial is zero or not. Assume we know that $\deg(p) \leq d$, and take $S = \{1, \ldots, 2d\}$. Pick a random $a \in S^n$ and check if $p(a) = 0$. If p is not identically zero, then by Lemma 3.6 we have that $\Pr_a[p(a) = 0] \leq 1/2$. Repeating it a few times can reduce the error probability.

We next describe a surprising application. Let $G = (U, V, E)$ be a bipartite graph with $|U| = |V| = n$. A matching in G is a pairing of the nodes in both sides that are connected by an edge. That is, a one-to-one mapping $\pi : U \to V$ such that $(u, \pi(u)) \in E$ for all $u \in U$. A basic question in graph theory is whether a graph contains a matching, and the algorithmic question is to check this efficiently.

We define a polynomial p such that G contains a matching iff $p \not\equiv 0$. As a first step, we define an $n \times n$ matrix M as follows. Assume $U = \{u_1, \ldots, u_n\}, V = \{v_1, \ldots, v_n\}$ and define:

$$M_{i,j} = \begin{cases} x_{i,j} & \text{if } (u_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}.$$

Here, $x_{i,j}$ are new variables. We define $p(x) = \det M(x)$, which is a polynomial in $x = (x_{i,j})$ of degree n.

Claim 3.7. *G has a matching iff $p \not\equiv 0$.*

Proof. The determinant of any $n \times n$ matrix M is given by

$$\det(M) = \sum_{\pi} (-1)^{\text{sign}(\pi)} \prod_{i=1}^{n} M_{i,\pi(i)}.$$

Here, π runs over all permutations on $[n]$, and the sign of a permutation is a number in $\{0, 1\}$ which is not really relevant for us. For our matrix M, if π corresponds to a matching then $\prod_{i=1}^{n} M_{i,\pi(i)} = \prod_{i=1}^{n} x_{i,\pi(i)}$, and otherwise it is zero. So, $\det(M)$ contains a monomial for each matching in G, and these are distinct monomials so they do not cancel each other. In particular, it is the zero polynomial iff G has not matchings. □

Computing the polynomial $p(x) = \det M(x)$ symbolically takes exponential time. However, if we substitute $x = a$ for real values a, then we can compute $p(a) = \det M(a)$ efficiently,

since computing the determinant of a matrix of real values can be done in polynomial time. We thus get the following algorithm.

Algorithm 5: Bipartite-Matching(G)

Input: Bipartite graph $G = (U, V, E)$
Output: True or False
1 Choose $a \in \{1, 2, \ldots, 2n\}^n$ randomly
2 Define a matrix as follows: $M_{i,j} = a_{i,j}$ if $(u_i, v_j) \in E$ and 0 otherwise
3 Compute $\det(M)$
4 **return** True if the determinant is nonzero, and False if it is zero

We obtain the following guarantee: if G has no matching, the algorithm returns False always; if G has a matching, the algorithm returns True with probability at least $\geq 1/2$. By repeating it a few times we can decrease the error probability.

4 One-sided and zero error algorithms

We have seen several examples of randomized algorithms. Some of them made only one-sided error, while others made no error, but the runtime was random. We now define the corresponding complexity classes that capture these guarantees.

Definition 4.1 (One-sided error: RP)**.** *A language $L \subset \{0,1\}^*$ is in RP if there exists a poly-time Turing machine $M(x)$ such that*

- *If $x \in L$ then $\Pr[M(x) = 1] \geq 2/3$*
- *If $x \notin L$ then $\Pr[M(x) = 1] = 0$*

We define coRP to be the complement condition, when the errors are when $x \notin L$. Another way to define it is that $L \in$ coRP if $L^c \in RP$.

Some examples that we saw have one-sided error: the primality testing of Miller-Rabin, finding simple paths in graphs, and checking for bipartite matching are some examples.

Definition 4.2 (Zero error but randomized runtime: ZPP)**.** *A language $L \subset \{0,1\}^*$ is in ZPP if there exists a randomized Turing machine $M(x)$ such that*

1. *For all $x \in \{0,1\}^*$ we have $M(x) = L(x)$ with probability one.*
2. *The expected running time of $M(x)$ is polynomial in $|x|$.*

We can connect ZPP to RP and coRP by the following theorem.

Theorem 4.3. *$ZPP = RP \cap coRP$.*

Proof. Assume first that $L \in$ ZPP. We will prove that $L \in$ RP, where the proof that $L \in$ coRP is analogous. Let M be a randomized TM such that $M(x) = L(x)$ for all inputs with probability one, and where the expected runtime of $M(x)$ is $T(x) \leq c|x|^c$ for some constant $c > 0$. Define a new TM M' as follows:

1. Run $M(x)$ for at most $3T(x)$ steps.

2. If $M(x)$ terminates then return its answer.

3. Otherwise return False.

Fix an input x. The expected runtime of $M(x)$ is at most $T(x)$. By Markov's inequality, the probability that it doesn't terminate after $3T(x)$ steps is at most $1/3$. If $x \notin L$ then we do not make a mistake, as we return False in this case. If $x \in L$ then we do make a mistake, but it is happens with probability at most $1/3$. Thus $L \in RP$.

For the other direction, assume that $L \in \text{RP} \cap \text{coRP}$, and we will prove that $L \in \text{ZPP}$. As $L \in \text{RP}$ there is a poly-time TM M_1 such that

- If $x \in L$ then $\Pr[M_1(x) = 1] \geq 2/3$
- If $x \notin L$ then $\Pr[M_1(x) = 1] = 0$.

In particular, if $M_1(x) = 1$ then it must be that $x \in L$, whereas if $M_1(x) = 0$ we are not sure if $x \in L$ or not. As $L \in \text{coRP}$ there is a poly-time TM M_2 such that

- If $x \notin L$ then $\Pr[M_2(x) = 1] \geq 2/3$
- If $x \in L$ then $\Pr[M_2(x) = 1] = 0$.

In particular, if $M_2(x) = 1$ then it must be that $x \notin L$, whereas if $M_2(x) = 0$ we are not sure if $x \in L$ or not.

We will run both $M_1(x)$ and $M_2(x)$. If $M_1(x) = 1$ then we know that $x \in L$. If $M_2(x) = 1$ then we know that $x \notin L$. Otherwise, we will repeat the process until one of these conditions occur. By our assumptions, for every input x, we have $M_1(x) = 1$ or $M_2(x) = 1$ in each iteration with probability at least $2/3$. We use this to upper bound the expected number of iterations.

Let $T = T(x)$ be a random variable for the number of iterations. If $T(x) > k$ then we must have "failed" in the first k iterations (namely $M_1(x) = M_2(x) = 0$ in these iterations), which happens with probability at most $1/3$ in each iteration. As the iterations are independent we get

$$\Pr[T(x) > k] \leq (1/3)^k$$

We have

$$\mathbb{E}[T] = \sum_{k \geq 0} \Pr[T(x) > k] \leq \sum_{k \geq 0} (1/3)^k = 3.$$

□

5 Space bounded randomized algorithms

We can define complexity classes which combine randomized TM with space-bounded computation. We define BPL to be the class of languages L which can be decided by a two-sided randomized TM, which runs in logarithmic space, and RL the class for one-sided randomized TMs. It is conjectured that both BPL and RL are the same as LOGSPACE. That is, randomness does not help to save space. The best result known is by Saks and Zhou [8].

Theorem 5.1 (Saks-Zhou)**.** *$BPL \subset SPACE(\log^{3/2} n)$.*

6 Randomized algorithms vs other models

We now move to compare randomness to other resources we saw - nondeterminism and nonuniformity.

Theorem 6.1 (Adleman [1])**.** *$BPP \subset P/poly$.*

Proof. Let $L \in BPP$. It will be convenient here to use the explicit definition of randomized TM. That is, there is a deterministic poly-time $M(x, r)$ such that

$$\Pr_r[M(x, r) \neq L(x)] \geq 1/3$$

for all inputs x, where r is the randomness.

The first step is to reduce the error. Fix the input length to $|x| = n$. We will reduce the error to 2^{-2n} using Corollary 2.2. Thus we get a new poly-time TM $M'(x, r')$ with a longer random string r' for which

$$\Pr_{r'}[M'(x, r') \neq L(x)] \leq 2^{-2n} \qquad \forall x \in \{0, 1\}^n.$$

Applying the union bound, as there are only 2^n inputs of length n, and the error probability is at most 2^{-2n}, we get that

$$\Pr_{r'}[\exists x \in \{0, 1\}^n, M'(x, r') \neq L(x)] \leq 2^n \cdot 2^{-2n} = 2^{-n} < 1.$$

Thus, there must exists a fixing of the random inputs r^* such that

$$M'(x, r^*) = L(x) \qquad \forall x \in \{0, 1\}^n.$$

Set $C_n(x) = M'(x, r^*)$ where we hard-code r^*. It is a circuit of size poly(n) that computes L correctly on all inputs of length n. The nonuniformity is the choice of r^*, which can be different for every input length n. □

Theorem 6.2 (Gács-Lautemann-Sipser [11, 4])**.** *$BPP \subset \Sigma_2 \cap \Pi_2$.*

Proof. We will prove $BPP \subset \Sigma_2$. The claim $BPP \subset \Pi_2$ follows since BPP is closed under negation.

Let $L \in BPP$. Apply error reduction to get a poly-time Turing machine $M(x, r)$ such that for any input x with $|x| = n$, we have

- If $x \in L$ then $\Pr_r[M(x, r) = 1] \geq 1 - 2^{-n}$
- If $x \notin L$ then $\Pr[M(x, r) = 1] \leq 2^{-n}$.

The random string is $r \in \{0, 1\}^m$ where $m = \text{poly}(n)$. Given x let $R(x) = \{r \in \{0, 1\}^m : M(x, r) = 1\}$. Then we have

- If $x \in L$ then $|R(x)| \geq 2^m(1 - 2^{-n})$.
- If $x \notin L$ then $|R(x)| \leq 2^{m-n}$.

The main idea is to use quantifiers in order to distinguish between the huge set of witnesses for $x \in L$ and the tiny set of witnesses for $x \notin L$.

We will use the following definition: given a set $R \subset \{0, 1\}^m$ and a string $u \in \{0, 1\}^m$ define $R + u = \{r \oplus u : r \in R\}$ to be the "shift" of R by u. Note that $|R + u| = |R|$. We set $k = 2m/n$ below. The two claims below capture the needed features.

Claim 6.3. *Let $R \subset \{0, 1\}^m$ with $|R| \leq 2^{m-n}$. For any $u_1, \ldots, u_k \in \{0, 1\}^m$ we have*

$$\bigcup_{i=1}^{k} (R + u_i) \neq \{0, 1\}^m.$$

Proof. $|\cup_{i=1}^k (R + u_i)| \leq \sum_{i=1}^k |R + u_i| = k2^{m-n} < 2^m$ for large enough n. □

Claim 6.4. *Let $R \subset \{0, 1\}^m$ with $|R| \geq 2^m(1 - 2^{-n})$. Then there exist $u_1, \ldots, u_k \in \{0, 1\}^m$ such that $\cup_{i=1}^k (R + u_i) = \{0, 1\}^m$.*

Proof. The proof is by the probabilistic method. We will argue that by choosing $u_1, \ldots, u_k \in \{0, 1\}^m$ randomly then with high probability $\cup_{i=1}^k (R + u_i) = \{0, 1\}^m$. Hence such a choice must exist. To calculate the probability, note that for any specific $r \in \{0, 1\}^m$,

$$\Pr_{u_1, \ldots, u_k \in \{0,1\}^m}[r \notin \cup_{i=1}^k (R + u_i)] = \prod_{i=1}^{k} \Pr_{u_i}[r \notin R + u_i] = \prod_{i=1}^{k} \Pr_{u_i}[u_i \notin R + r] = (2^{-n})^k = 2^{-2m}.$$

Hence,

$$\Pr_{u_1, \ldots, u_k}[\exists r \in \{0, 1\}^m, r \notin \cup_{i=1}^m (R + u_i)] \leq 2^m 2^{-2m} = 2^{-m} < 1.$$

□

We use both claims to complete the proof. Consider the statement:

$$\phi(x, u_1, \ldots, u_k, r) = M(x, r + u_1) = 1 \vee \ldots \vee M(x, r + u_k) = 1.$$

and

$$\psi(x) = \exists u_1, \ldots, u_k \in \{0,1\}^m \; \forall r \in \{0,1\}^m \; \phi(x, u_1, \ldots, u_k, r).$$

Then we have that:

- $\phi \in P$ since $m, k = \text{poly}(n)$.
- This implies that $\psi \in \Sigma_2$.
- If $x \in L$ then $|R(x)| \geq 2^m(1 - 2^{-n})$ and hence $\psi(x) = 1$.
- If $x \notin L$ then $|R(x)| \leq 2^{m-n}$ and hence $\psi(x) = 0$.

We conclude that ψ computes L, and hence $L \in \Sigma_2$. □

7 Randomness as a proof technique

So far we discussed randomness as a computational resource. Here we show that it is also very powerful as a proof technique, where it stands at the core of the "probabilistic method" pioneered by Erdős. We demonstrate it by proving the existence of good codes.

First, we need a definition. Given $x, y \in \{0,1\}^n$ their hamming distance is the number of coordinates where they differ,

$$\Delta(x, y) = |\{i : x_i \neq y_i\}|.$$

Definition 7.1 (Codes)**.** *An (n, k, d) binary code is a set $C \subset \{0,1\}^n$ of size $|C| = 2^k$, such that for any distinct $x, y \in C$ it holds that $\Delta(x, y) \geq d$. We call n the block length, k the dimension and d the minimal distance of the code.*

Codes allow to encode messages in a way that is resilient to errors. An (n, k, d) codes allows to encode k-bit messages to n-bit codewords, by defining some one-to-one encoding map $E : \{0,1\}^k \to C$.

Claim 7.2. *An (n, k, d) code allows to detect $d - 1$ errors, and to correct $(d - 1)/2$ errors.*

Proof. Let $x \in C$ be a codeword, and let y be any other word obtained by corrupting at most $d - 1$ coordinates in x. Then y cannot be in C, since the minimal distance of C is d. Similarly, if $\Delta(x, y) \leq (d - 1)/2$ then we can recover x uniquely from y, as it is the closest codeword to y. □

We say a code is good if k, d are linear in n. Formally, we need to consider a family of (n_i, k_i, d_i) codes, where n_i are increasing and $k_i \geq cn, d_i \geq cn$ for some absolute constant $c > 0$. The follow lemma uses the probabilistic method to prove that good codes exist.

Lemma 7.3. *There exists an absolute constant $c > 0$ such the following holds. For any large enough n, there exist (n, k, d) codes with $k \geq cn, d \geq cn$.*

Proof. Let $c > 0$ to be chosen later, and set $k = cn, d = cn$. Let $C = \{x_1, \ldots, x_{2^k}\}$ where $x_i \in \{0,1\}^n$ are uniformly chosen. We will show that with high probability, $\Delta(x_i, x_j) \geq d$ for all distinct $i, j \in [2^k]$.

First, let $x, y \in \{0,1\}^n$ be uniform strings. The number of such pairs with $\Delta(x, y) \leq d$ is $2^n \binom{n}{\leq d}$ where we shorthand $\binom{n}{\leq d} = \sum_{i=0}^{d} \binom{n}{i}$. The total number of pairs is 2^{2n}. Thus

$$\Pr_{x,y}[\Delta(x, y) \leq d] = \frac{\binom{n}{\leq d}}{2^n} \approx \frac{(1/c)^{cn}}{2^n} \leq 2^{-n/2}$$

for a small enough value of $c > 0$. Thus, the probability that C is not a good code is at most

$$\Pr[C \text{ is not } (n, k, d)] = \Pr[\exists i \neq j,\ \Delta(x_i, x_j) \leq d] \leq 2^{2k} 2^{-n/2} = 2^{(2c-1/2)n}.$$

Again, for a small enough $c > 0$, this probability is less than 1 (in fact, it is exponentially small). Hence there are choices for C which are $(n, k = cn, d = cn)$ codes, and hence good codes. □

References

[1] L. Adleman. Two theorems on random polynomial time. In 19th Annual Symposium on Foundations of Computer Science (sfcs 1978), pages 75–83. IEEE, 1978.

[2] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. Annals of mathematics, pages 781–793, 2004.

[3] R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1977.

[4] C. Lautemann. Bpp and the polynomial hierarchy. Information Processing Letters, 17(4):215–217, 1983.

[5] G. L. Miller. Riemann's hypothesis and tests for primality. Journal of computer and system sciences, 13(3):300–317, 1976.

[6] M. O. Rabin. Probabilistic algorithm for testing primality. Journal of number theory, 12(1):128–138, 1980.

[7] O. Reingold. Undirected connectivity in log-space. Journal of the ACM (JACM), 55(4):17, 2008.

[8] M. Saks and S. Zhou. $BP_H SPACE(S) \subset DSPACE(S^{3/2})$. Journal of Computer and System Sciences, 58(2):376–403, 1999.

[9] T. Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039), pages 410–414. IEEE, 1999.

[10] J. T. Schwartz. Probabilistic algorithms for verification of polynomial identities. In International Symposium on Symbolic and Algebraic Manipulation, pages 200–215. Springer, 1979.

[11] M. Sipser. A complexity theoretic approach to randomness. In Proceedings of the fifteenth annual ACM symposium on Theory of computing, pages 330–335. Citeseer, 1983.

[12] R. Zippel. Probabilistic algorithms for sparse polynomials. In International Symposium on Symbolic and Algebraic Manipulation, pages 216–226. Springer, 1979.