- Start as early as possible, and contact the instructor if you get stuck.
- See the course outline for details about the course's grading policy and rules on collaboration.
- Submit your completed solutions to **Crowdmark**.

1. **Three-sided coin [5 marks]**
   Suppose we want to sample uniformly from three possibilities, $\{a, b, c\}$, using a stream of uniformly random bits. For all $B \in \mathbb{N}$, show that if a probabilistic Turing machine always uses fewer than B bits then it cannot sample from the *exact* three outcome distribution.
   (a) Show that if a probabilistic Turing machine *always* uses a fixed number of bits to output a, b, or c, then it cannot sample them uniformly.

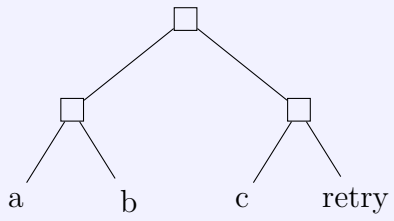
   Solution

   Suppose the Turing machine uses less than B bits. These are independent, uniformly random bits, so all 2^B randomness strings are possible. For each random string, the Turing machine makes a deterministic choice. If there are n_a strings which produce a, n_b strings which produce b, and n_c strings which produce c, then the probabilities $n_a/2^B$, $n_b/2^B$, and $n_c/2^B$ cannot all be equal because $3n_a = 2^B$ is impossible by the deep number-theoretic fact that $3 \nmid 2^B$.

   (b) Give an algorithm that samples exactly from the uniform distribution on $\{a, b, c\}$. Analyze the expected runtime of your algorithm.

   Solution

   Flip 2 coins for 4 possible outcomes. Divide 4 three ways as evenly as possible (one outcome each to a, b, c), and if you see the remaining outcome, reroll from scratch.

   Let T be a random variable for the number of rolls. The way this algorithm works, we get a recursion:

   $$T = \begin{cases} 2, & \text{w.p. } 0.75, \\ 2 + T' & \text{w.p. } 0.25. \end{cases}$$

   where T' is the number of rolls in the case we retry. Since $\mathbb{E}[T] = \mathbb{E}[T']$, we have

   $$\mathbb{E}[T] = 2 + \tfrac{1}{4}\mathbb{E}[T]$$

straightforwardly from the recursion, and it solves to $\mathbb{E}[T] = \frac{8}{3} \approx 2.666$. So despite the fact that no bounded number of coins suffices (by part (a)), and expected constant number of coins is enough.

2. BPPSPACE [**5 marks**]
Recall from class bounded-error polynomial space.

$$\mathsf{BPPSPACE} = \{\text{languages } \tfrac{1}{3}\text{-decided by a probabilistic TM in polynomial space}\}$$

In the time-space hierarchy theorem, we saw that $\mathsf{SPACE}(s) \subseteq \mathsf{TIME}(2^{O(s)})$ because if a machine falls into the same configuration twice, it will loop forever. For a probabilistic Turing machine, it is not so clear, since a random choice may break the machine out of the loop.
Design a probabilistic Turing machine (high-level steps, not state diagram) that runs for expected *doubly*-exponential time in the length of the input. Justify the correctness and runtime of your TM.[1]

Solution

The algorithm for the TM is simple: for input length n, we have $\mathsf{poly}(n)$ space. This can store a number with $\mathsf{poly}(n)$ digits and thus $2^{\mathsf{poly}(n)}$ size. We initialize the counter to zero and flip coins. If we see a 1, we increment the counter. If we see a 0, then we reset the counter to 0. We halt if the counter reaches its maximum value which, as previously mentioned, is $2^{\mathsf{poly}(n)}$.

While it is intuitively clear that a run of $\geq k$ tails (to halt the machine) will take $2^{\Theta k}$ time in expectation, we should do some math to verify it. Let X_k be a random variable for the number of coin flips to see a length-k run of tails. Observe that X_0 is the constant 0, and

$$\mathbb{E}[X_{k+1}] = 1 + \mathbb{E}[X_k] + \tfrac{1}{2}\mathbb{E}[X_{k+1}]$$

because to see a run of $k+1$ tails, we first have to see a run of k tails, plus one more. And 50% of the time, that coin flip is wrong, so we start over. This solves to

$$\mathbb{E}[X_k] = 2^{k+1} - 2.$$

The expected runtime is exponential in k, and we polynomial space allows k to be exponential in the input length. Hence, *doubly* exponential expected runtime.

3. **Space constructible and space hierarchy** [**12 marks**]
We saw the time-hierarchy theorem in class, and there is an analogous space hierarchy theorem:[2]

Theorem 1. *For all functions $f, g\colon \mathbb{N} \to \mathbb{N}$ such that*

[1] Nonetheless, it turns out that $\mathsf{PSPACE} = \mathsf{BPPSPACE}$.
[2] Not to be confused with the Time-Space hierarchy theorem we saw in class.

(a) $\log n \leq f = o(g)$*, and*
(b) g *is space constructible,*
we have $\mathsf{SPACE}(f) \subsetneq \mathsf{SPACE}(g)$.

We say a function $h\colon \mathbb{N} \to \mathbb{N}$ is *space constructible* if given $\mathtt{1}^n$ as input, we can construct $\mathtt{1}^{h(n)}$ in space $O(h(n))$, for all $n \in \mathbb{N}$.

(a) First, argue that for any Turing machine M deciding a language $\mathcal{L}(M)$ in $s(n)$ space, there exists a Turing machine M' deciding the same language, but using $\Gamma = \{\square, \mathtt{0}, \mathtt{1}\}$ and $\mathcal{O}(s(n))$ space.

Solution

The first lecture, we showed that you could encode elements of a finite set S as bit strings of length $\lceil S \rceil$. In case I never spelled it out explicitly, you can apply this to turn every tape alphabet symbol into a longer bit string.

Let's say k bits for this example. We rework the Turing machine to read, write, and move in blocks of k. For example, to read one symbol, we scan right k steps and "remember" the symbol with a depth-k binary tree of states. At any leaf of this tree, we know the tape symbol represented by those k bits, and can begin writing a new symbol over the old one. Writing obviously requires walking k steps to the left to overwrite the previous symbol. Finally, we walk k steps in one direction or the other get to the beginning of the next symbol.

The other issue to contend with is the input format. The input will be the same bit string on both M and M', but we want it to be the *encoded* bit string on for M', since we have built the TM to process that. It is a straightforward (if slow) process to convert the input by expanding one bit at a time, shifting the rest to the right, and repeating from where we left off (marked with a $\square$, for example).

The space cost is dominated by the fact that everything on the tape is $k = \lceil \log_2 |S| \rceil$ times longer, where S is the tape alphabet of M. Hence, it remains $\mathcal{O}(s(n))$ since S is constant.

Author's note: We go to the trouble of converting to binary because the next problem is much more confusing if the machine doing the simulation and the machine being simulated don't have the same alphabet.

(b) What is the space overhead of simulation? Assume the simulator and the machine being simulated use the same tape alphabet, Γ. Show that there exists an input/working tape Turing machine U which, given $\langle M \rangle x$ as input, simulates M on x in the same space plus $\mathcal{O}(\log n)$, where n is the length of x.

Solution

Since the simulator U and the machine being simulated are promised to have the same tape alphabet, it is sensible to have the work tape configuration of U look a lot like the work tape configuration of M. However, U also needs to retain $\langle M \rangle$ (conveniently on the input tape), the position of the working tape head (which it duplicates with its own working tape head), and the current state.

The current state is just a number $q \in Q \subseteq \mathbb{N}$. Clearly $|Q| \leq |\langle M \rangle| \leq n$, so we can encode q with $\mathcal{O}(\log n)$ bits. We cannot put the counter at the start or end of the working tape, since we want to use the working tape head of U to simulate the working tape head of M, and an excursion to the end of the tape would "lose our place". Therefore, the description of q has to follow the head around, e.g., a configuration yqz of M's tape becomes

$$y\square\langle q \rangle\square z$$

on U's tape. When the head moves, we shift the whole $\square\langle q \rangle\square$ segment (representing the tape head) left or right a step, without straying more than a step or two into y or z.

Thus, U simulates M with only the extra state information. This is an *additive* $\mathcal{O}(\log n)$ space overhead, as required.

(c) Prove the space hierarchy theorem *carefully*. Point out where you use each hypothesis. We encourage you to use the language

$$A = \{\langle M \rangle x : M \text{ rejects } \langle M \rangle x \text{ in } g(n) \text{ space}\}.$$

Solution

It is quite clear that $\mathsf{SPACE}(f) \subseteq \mathsf{SPACE}(g)$ since $f = o(g)$ (**note: hypothesis (a)**), and giving ourselves more space (even just asymptotically) can only increase the set of languages we can decide. It remains to prove that they are not equal. For this, we introduce the language

$$A = \{\langle M \rangle x \colon M \text{ rejects } \langle M \rangle x \text{ in } g(n) \text{ space}\}.$$

First, we argue $A \in \mathsf{SPACE}(g)$. By problem 3b, the simulation of M on $\langle M \rangle x$ uses $\mathcal{O}(s(n) + \log n)$ space, where $s(n)$ is the space cost of M. An easy modification allows us to track the space cost during the simulation. If we initialize $g(n)$ somewhere on the tape, the simulation can prematurely halt and reject if the simulation uses more than $g(n)$ space. Note that constructing $\langle g(n) \rangle$ requires g to be **space constructible**. In this case, the space usage remains $\mathcal{O}(g(n) + \log n)$, which is $\mathcal{O}(g(n))$ since $\log n = o(g(n))$ (**hypothesis (a) again**).

Next, we have to show $A \notin \mathsf{SPACE}(f)$. Suppose to the contrary that $A \in \mathsf{SPACE}(f)$. Let us construct a new Turing machine D which on input x does the following:

i. Computes its own description, $\langle D \rangle$, by the recursion theorem.
ii. Runs A on $\langle D \rangle x$.
iii. Return whatever A does.

The recursion theorem and $|\langle D \rangle|$ are independent of the input, so they count as $\mathcal{O}(1)$ space. Accepting and rejecting does not cost space. Therefore, the space usage of D is dominated by the simulation of A on $\langle D \rangle x$ in step (ii). Since A runs in $f(n)$ space, and the simulation only requires $\mathcal{O}(\log n)$ bits of overhead, it costs $\mathcal{O}(f(n))$ space overall (**using hypothesis** $\log n \leq f$).

Using the fact that $f(n) = o(g(n))$, for any constant $K \in \mathbb{N}$, there exists $n_0 \in \mathbb{N}$ such that $f(n) \leq g(n)/K$ for all $n \geq n_0$. In particular, for $K = 1$ there is a threshold n_0 such that $f(n) \leq g(n)$. Therefore, for inputs x of length $\geq n_0$, we have

$$\begin{aligned} A \text{ accepts } \langle D \rangle x &\iff D \text{ rejects } \langle D \rangle x \text{ in } g(n) \text{ space} \\ &\iff D \text{ rejects } \langle D \rangle x \\ &\iff A \text{ rejects } \langle D \rangle x. \end{aligned}$$

This is a contradiction, so $A \notin \mathsf{SPACE}(f)$.

(d) Construct a function f such that $f(n) \geq \log n$ and f is not space constructible. Hint: We don't have many tools to show that you *can't* compute a function.

Solution

Take an undecidable Boolean function $u \colon \mathbb{N} \to \{0, 1\}$ and let the non-space-constructible function be $f(n) = u(n) + 2n$. If f was space constructible, we don't even need to subtract off $2n$ because the *parity* of $f(n)$ will tell us the value of $u(n)$, and then an undecidable functio is not only decided, but decided in linear space. Contradiction $\implies$ f is not space constructible.

4. **Solar Powered PC [10 marks]**
Suppose L is a language decidable in polynomial space. Let's say we have enough time and space to decide it, but our computer has one major defect: it is solar powered, without a battery or permanent storage. Every night it forgets almost everything; *all* data from memory is lost except for
- the problem input, $x \in \{0, 1\}^n$,
- the current day (as a number $d \in \mathbb{N}$),
- 3 bits of output $b \in \{0, 1\}^3$, from the previous day.

Surprisingly, we can still decide L with this device. In particular, we will use the

three-bit programs from A4Q3, and assume the following theorem.[3]

Theorem 2. *For any polynomial-size family of formulas, we can construct an equivalent polynomial-length family of three-bit programs.*

(a) Recall the TQBF problem, and use it to show that any problem in PSPACE is solved by an exponential-size family of Boolean formulas.

Solution

Take any language $L \in \mathsf{PSPACE}$. There is a reduction $L \leq_P \mathsf{TQBF}$, and therefore a poly-time computable function h which maps instances of L to instances of TQBF. Without loss of generality, we can assume that all inputs of length n produce an instance of TQBF with the same number of quantifiers.

We know the polynomial-time computation h can be simulated by a polynomial-size and polynomial-depth family of circuits. By duplicating branches, we can turn this into an exponential-size formula (or collection of formulas, since there are technically multiple output bits).

There is Turing machine that takes a formula $\langle\phi\rangle$ and an assignment of all the variables $a_1 \dots a_n$, and evaluates the formula. By circuit simulation again, there is a polynomial-size/polynomial-depth circuit that does the same evaluate, and this also blows up to at most an exponential-size formula.

We can afford to evaluate ϕ at all possible assignments $a_1 \cdots a_n \in \{0,1\}^n$ in parallel by this method. Each evaluation is an exponential-length formula, and the bits of $\langle\phi\rangle$ are themselves exponential-length formulas of the input. All that remains is to evaluate TQBF from $\phi(0^n), \dots, \phi(1^n)$. For that, observe that $\exists$ and $\forall$ translate readily into ANDs and ORs.

$$\exists y\, \phi(x,y) \to \phi(x,0) \vee \phi(x,1)$$
$$\forall y\, \phi(x,y) \to \phi(x,0) \wedge \phi(x,1).$$

The n-fold-quantified TQBF formula will therefore expand into a depth-n formula of ϕ evaluations, which is therefore at most exponential length.

Note: An astute (and private) Piazza post pointed out the fact from class that every *Boolean function has an exponential-size formula. However, the argument above lets us* construct *the formula in a way that we can access branches of it. This lets us also* construct *the three-bit program in Theorem 2, and thus the solar powered computer can figure out what to do each day in polynomial time.*

If you tried to construct the exponential-size CNF we saw in class, you would need to evaluate L on inputs of length n. We can use circuit simulation

[3] You do *not* need to have solved A4Q3 for this problem.

again, but since PSPACE *computations can be exponentially long, it might be an exponentially-deep circuit, leading to a doubly-exponential-size formula.*

(b) Treat Theorem 2 as a black box (ignore how it was proved), and argue that we can scale it up to exponential-size formulas and exponentially long three-bit programs.

Solution

The idea is called *padding*, and I'm surprised we didn't run into it earlier.

Suppose we have an exponential size (say, 2^n) formula in n variables. Add a ton of dummy variables (e.g., $2^n - n$) which never appear in the formula, until the formula size is merely polynomial in the total number of variables ($m = 2^n$), dumnmy variables included. The now polynomial-size ($\mathcal{O}(m)$) family of formulas becomes a polynomial-size family of three-bit programs (e.g., $\mathcal{O}(m^2)$). The length of the program is a polynomial (m^2) of an exponential ($m = 2^n$) of the original variable count, so it is still merely exponential (4^n, in this case) in n.

We don't expect the dummy variables will appear in the three-bit program, but we can hard-code them to `0` to be sure, and it can't change the value of the program because they don't appear in the corresponding formula.

(c) Omitting some details[4], this finishes the proof.
Why doesn't the result contradict our normal intuition (not to mention the space hierarchy theorem) that some problems in PSPACE require polynomial space? Identify the loophole in a couple of sentences.

Solution

The solar powered computer takes the input x, day d, and three bits b. Since x is immutable (does not change during the whole computation) and b isn't enough space, by process of elimination, our polynomial space is being smuggled in through d.

More precisely, I believe you can simulate the solar-powered PC process in $\mathcal{O}(\log D)$ space, where D is the number of days required, since the input x doesn't count and b is $\mathcal{O}(1)$ space. Thus, for the languages L that provably require $\mathsf{poly}(n)$ space, it follows that this simulation method will require $2^{\Omega(\mathsf{poly}(n))}$ days.

[4]We omit the details of (i) computing the length of the program so we know when to stop, and (ii) computing the m'th program instruction on the m'th day.