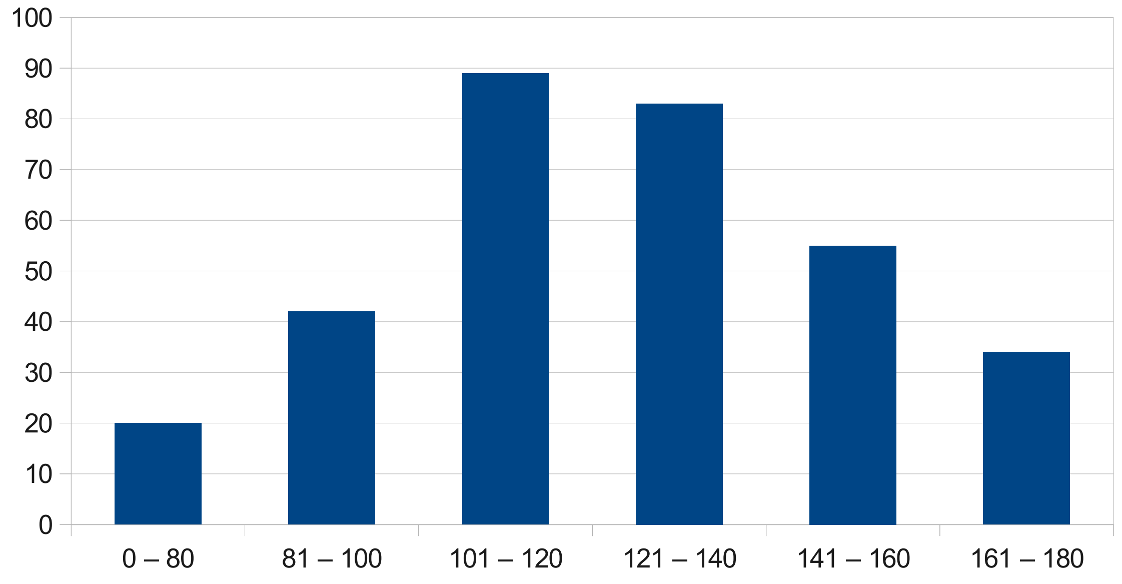
CS103 Handout 34S
Fall 2013 December 13, 2013

CS103 Final Exam Solutions

The distribution of the exam grades was as follows:



Overall, the final statistics were as follows:

75th Percentile: 143 / 180 (80%)

50th Percentile: 123 / 180 (68%)

25th Percentile: 106 / 180 (59%)

Please feel free to email the staff list if you have any questions on the exam or its grading. If you believe that your exam was graded erroneously, please contact Keith no later than **Friday, January 10, 2013** so that we can regrade it.

All exams are available for pickup in a specially-marked filing cabinet where we stored the midterm exams.

Thanks for a great quarter, and enjoy the break!

Problem One: Discrete Mathematics (20 Points Total)

Prove, by induction, that $c_n = (2 - 1/2^n)$mg for all $n \in \mathbb{N}$. This proves that the patient will never have more than 2mg of medicine in her bloodstream, even if she continues to take 1mg doses every hour.

Proof: By induction. Let $P(n)$ be "$c_n = (2 - 1/2^n)$mg." We will prove that $P(n)$ holds for every $n \in \mathbb{N}$.

As our base case, we prove $P(0)$, that $c_0 = (2 - 1/2^0)$mg. Since $2 - 1/2^0 = 1$, we will equivalently prove that $c_0 = 1$mg. This is the case because at time 0 hours, one dose of 1mg is administered, so the total amount of medicine in the patient's bloodstream is 1mg.

For the inductive step, assume that for some $n \in \mathbb{N}$ that $P(n)$ holds, meaning $c_n = (2 - 1/2^n)$mg. We will prove that $P(n + 1)$ holds, meaning $c_{n+1} = (2 - 1/2^{n+1})$mg. To see this, note that at time $n + 1$ hours, half of the medicine from n hours will have decayed, leaving $c_n / 2$ medicine in the patient's bloodstream, and then one more mg is administered. This means that $c_{n+1} = ½c_n + 1$mg. By our inductive hypothesis, we have

$$c_{n+1} = ½c_n + 1\text{mg} = ½(2 - 1/2^n)\text{mg} + 1\text{mg} = (1 - 1/2^{n+1})\text{mg} + 1\text{mg} = (2 - 1/2^{n+1})\text{mg}$$

so $c_{n+1} = (2 - 1/2^{n+1})$mg, completing the induction. ■

Why we asked this question: We wanted to include one question on the final exam that was a representative of the discrete math portion of the course. I chose this problem because it has practical applications, shows off the utility of mathematical induction, and proves a result that is somewhat surprising.

Common mistakes: Most people did very well on this problem, though there were two common errors. First, some proofs defined $P(n)$ incorrectly with statements like

Let $P(n) = 2 - 1/2^n$

or used $P(n)$ as a number, as in the expression

Therefore, $P(n + 1) = ½(2 - 1/2^n) + 1$

Remember that $P(n)$ should be a logical assertion (something that's either true or false) rather than a numerical quantity.

Additionally, many solutions tried to prove the base case using logic like this:

To prove $P(0)$, note that $c_0 = 2 - 1/2^0 = 2 - 1 = 1$, which is true because $c_0 = 1$.

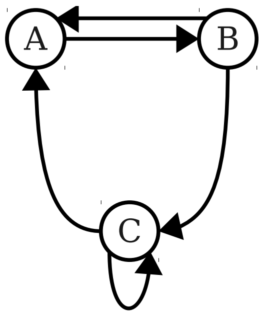
The problem with this statement is that it begins with the assumption that $c_0 = (2 - 1/2^0)$mg, which can't be assumed without first being proven A more proper way to do this would be to start with the known statement $c_0 = 1$mg and to then derive that $c_0 = (2 - 1/2^0)$mg.

Question 75th Percentile: 20 / 20

Question 50th Percentile: 20 / 20

Question 25th Percentile: 17 / 20

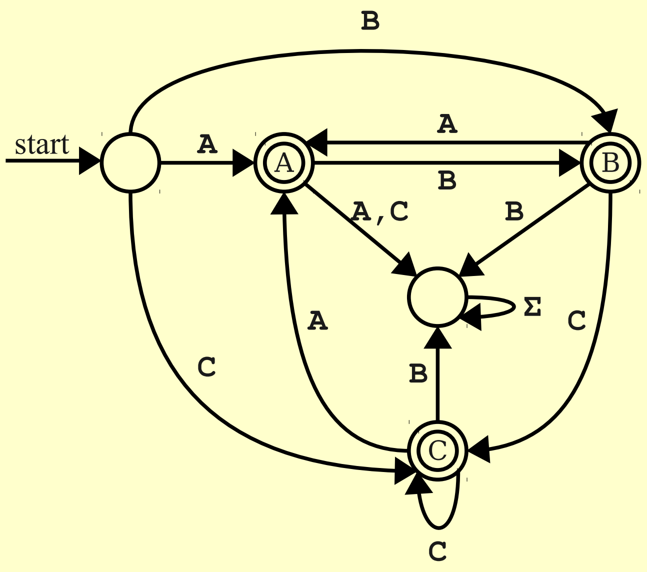
Problem Two: Regular Languages (45 Points Total)



(i) Path Automata (10 Points)

Let $L = \{ w \in \Sigma^* \mid w$ represents a path in $G \}$, where G is the graph given above. Design a DFA for L.

Here is one possible solution:



Intuitively, this automaton is formed by taking the graph, adding a start state to jump into the various nodes in the graph, and adding missing transitions into a dead state.

Why we asked this question: The language in this question is very different from the others and was chosen to exercise your understanding of graphs and the intuition that DFAs are essentially graphs with extra information in them. We also chose this problem to make sure you remembered that DFAs must have all transitions defined on all states.

Common mistakes: Most people got this problem right. The most common errors included accidentally leaving off transitions (remember that DFAs have to have a transition defined for all state/symbol pairs), forgetting to add a dead state, or including a dead state and forgetting the transition on Σ.

(ii) Regular Expressions **(10 Points)**

Let Σ = {**1**, **2**, **≤**} and let *L* = { *w* ∈ Σ* | *w* is a valid chain of inequalities relating the numbers **1** and **2** }. For example:

1≤2 ∈ *L*	**2≤≤** ∉ *L*
1≤1≤2≤2 ∈ *L*	**≤2** ∉ *L*
2≤2≤2 ∈ *L*	ε ∉ *L*
1≤1≤1≤1 ∈ *L*	**1** ∉ *L*
1≤1≤2 ∈ *L*	**12≤22** ∉ *L*

Note in particular that inequalities involving numbers like 12, 222, 121212, etc. whose digits are 1 and 2 aren't allowed (the inequality should only relate the numbers 1 and 2) and any individual number itself isn't allowed.

Write a regular expression for *L*.

One possible answer is

```
(1≤)*(1≤1 | 1≤2 | 2≤2)(≤2)*
```

The middle portion of this regular expression generates at least one inequality, with the starred expressions on the outside adding in other parts of the inequality as necessary.

Why we asked this question: We chose this language because strings in the language have a simple form that is made slightly more complex by the requirement that there be at least one inequality sign somewhere in the chain. Our solution works by putting it into the middle and growing around that, though it's also possible to split the regex into three cases, each of which handles one of the possibilities of what the shape of the inequality chain will be.

Common mistakes: Most people got this one right. Common mistakes included accidentally generating strings consisting of a single number.

(iii) Distinguishability **(5 Points)**

Let *L* be an arbitrary language. Prove that if *x* ∈ *L* and *y* ∉ *L*, then *x* and *y* are distinguishable relative to *L*.

Proof: Let *L* be an arbitrary language and let *x* and *y* be strings where *x* ∈ *L* and *y* ∉ *L*. Let *w* = ε. Then *xw* = *x* and *yw* = *y*, so *xw* ∈ *L* and *yw* ∉ *L*. Therefore, *x* and *y* are distinguishable relative to *L*. ■

Why we asked this question: This question is the first part in a three-part series for lower-bounding the number of states in an NFA, something we didn't talk about directly this quarter but which follows from the concepts we've already seen.

Common mistakes: Almost everyone got this problem right. Excellent job!

(iv) State Lower Bounds **(10 Points)**

In Problem Set 6, you proved that any DFA for the language

L = { w ∈ {`0`, `1`}* | w contains at least two `1`'s separated by exactly 5 characters }

must have at least 64 states. One possible proof of this result (which is given in our solution set) is to consider the set of all strings of `0`s and `1`s whose length is exactly six. There are 64 of these strings, and all of them are distinguishable relative to L.

It turns out you can prove a stronger claim: any DFA for this language must have at least 65 states. Using your result from part (iii), prove that any DFA for L must have at least 65 states. *(Hint: The above logic gives you a set of 64 strings distinguishable relative to L and you can use this fact without proof. Can you find a string distinguishable from all of them?)*

Proof: First, note that no string of six characters is in L, since in order for a string to belong to L, it must have at least seven characters (two ones and five intervening characters). So let S be any set with the 64 strings described above (none of which are in L) and any string in L. This set has cardinality 65. Moreover, we claim that all strings in it are distinguishable relative to L. To see this, consider any two strings in S. If those strings are both strings of length six, then they're distinguishable relative to L, as mentioned in the problem statement. Otherwise, one string must be as string of length six, which is not in L, and the other must be a string in L, so by our result from (iii) they are distinguishable relative to L. Since S is a set of 65 strings distinguishable relative to L, by the result from Problem Set Six, we know that any DFA for L must have at least 65 states. ■

Why we asked this question: One of the more common questions we received in office hours and on the staff list about Problem Set Six concerned the state lower bound question. Many of you were interested in whether the bounds that we found were tight bounds. This questions shows that that lower bound of 64 states actually *isn't* a tight bound, since we can find a set of 65 strings that are distinguishable relative to L. Additionally, we hoped that this question would be a way to assess whether you understood the connection between distinguishability and the number of states in a DFA.

Common mistakes: One of the most common mistakes we saw was starting with the set S of all strings of length six, but then choosing a 65th string that wasn't distinguishable from all of the strings in S. As an example, ε is not distinguishable from the string `000000` (it's a good exercise to see why.) Commonly, this error arose from incorrectly claiming that all strings of length six are in S (they aren't), or talking about distinguishability relative to S rather than distinguishability relative to L. Another common error was to find two strings, one in L and one not in L, and claiming that this would require there to be 65 states. Although any string in L is distinguishable from any string not in L, that doesn't necessarily mean that those strings are distinguishable from all the strings in S.

(v) State Lower Bounds, Part Two **(10 Points)**

In Problem Set Five, you designed an NFA for the language

$$L = \{ w \in \{\texttt{0}, \texttt{1}\}^* \mid w \text{ contains at least two } \texttt{1}\text{'s separated by exactly 5 characters} \}$$

Using your result from part (iv), which says that any DFA for L must have at least 65 states, prove that every NFA for L must have at least seven states. *(Hint: Think about the subset construction.)*

Proof: By contradiction; assume there is an NFA N for L with six or fewer states. By the subset construction, we know that we can convert N into a DFA D whose language is L and whose number of states is at most $2^6 = 64$. This is impossible, since by our result from (iv) we know that any DFA for L must have at least 65 states. We have reached a contradiction, so our assumption must have been wrong. Thus any NFA for L must have at least seven states. ■

Why we asked this question: This question, which we intended to be a bit of a challenge, was designed to see if you understood the equivalence of NFAs and DFAs at a deeper level. The construction that turns NFAs into DFAs can significantly increase the size of the NFA, but it can't do so arbitrarily. In this case, we can use the fact that since the blowup is "only" exponential, any NFA for L has to have at least $\lceil \log_2 65 \rceil = 7$ states in it.

Common mistakes: The most common mistakes on this problem involved trying to run the subset construction "backwards" and trying to argue that any DFA must have states that correspond to sets of states in some NFA. This is not always true, and arguments of this form don't fully establish that no small NFA exists for L.

Another common mistake was to try to argue about how the NFA would have to be structured. Many solutions claim that the NFA has to have a chain of states between two states that mark the ones in the string in order to remember where the `1`s in the string are. This is true, but proving it is challenging. To prove that a particular solution does not exist, it's not sufficient to describe one possible solution and claim that any arbitrary solution must use the same strategy as your solution. Instead, you would need to argue why any solution following a different route would have to fail.

As a challenge, it's possible to prove with a more nuanced argument that any NFA for this language must have at least *eight* states, not just seven. Try seeing if you can show this. As a hint, think about the length of the shortest path from the start state in the NFA to any accepting state of that NFA.

75th Percentile: 45 / 45

50th Percentile: 38 / 45

25th Percentile: 30 / 45

Problem Three: Context-Free Languages (20 Points)

Write a CFG for *NEAR*. Then, draw parse trees for **`11+1≈11`** and **`1+11≈11`**. We've included space to draw those parse trees at the bottom of the page.

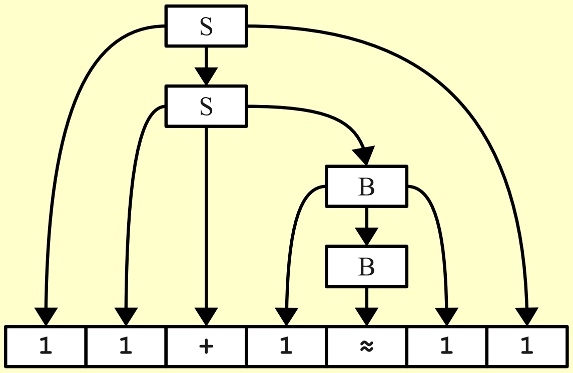
One possible grammar is the following:

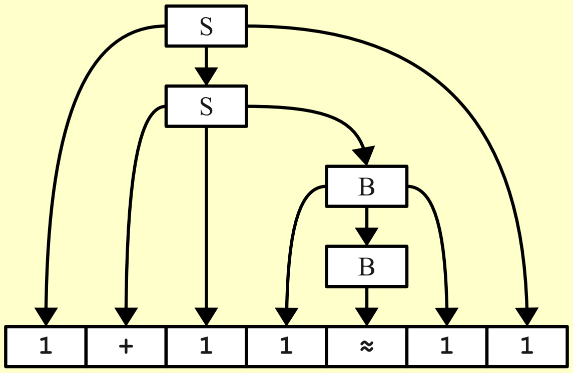
S → **`1S1`** | **`+1B`** | **`1+B`**

B → **`1B1`** | **`≈`**

This grammar is similar to the grammar for *ADD* given in the solution set to Problem Set Six, except when transitioning from the S nonterminal to the B nonterminal, we add an extra **`1`** either right before or right after the S.

Parse trees are shown here:





Why we asked this question: There are many different solutions to this problem. The one we gave above works by recognizing that there's a convenient time to insert the extra **`1`** when switching from the first number to the second. Other solutions work by adding extra nonterminals to keep track of whether an extra **`1`** should be inserted in the first group of **`1`**s or the second. We hoped that by giving you this flexibility, you would have a good chance of coming up with a working solution. Additionally, we wanted to ensure that you understood how to draw parse trees from a given grammar.

Common mistakes: Many grammars required that the number of **`1`**s on the right-hand side of the **`≈`** be at least one, which excludes strings like **`1+≈`** that are in the language. Several solutions allowed for multiple **`+`** signs on both sides of the **`≈`**, which isn't permitted by the definition of the language. We also saw several grammars that forgot to generate **`1`**s after the **`+`** sign and grammars that accidentally made the number of **`1`**s on the left-hand side twice the number of those on the left-hand side.

75th Percentile: 20 / 20

50th Percentile: 20 / 20

25th Percentile: 15 / 20

Problem Four: R, RE, and co-RE Languages (45 Points Total)

Consider the following language L:

$$L = \{ \langle M \rangle \mid M \text{ is a TM and every string in } \mathscr{L}(M) \text{ is a palindrome} \}$$

(i) Palindromes and RE Languages (25 Points)

Prove that $L \notin$ **RE**.

Proof: We will prove $\overline{A_{TM}} \le_M L$. Since $\overline{A_{TM}} \notin$ **RE**, this proves that $L \notin$ **RE**, as required.

To show $\overline{A_{TM}} \le_M L$, we give a mapping reduction from $\overline{A_{TM}}$ to L. For any TM/string pair $\langle M, w \rangle$, define $f(\langle M, w \rangle) = \langle \text{Amp}(M, w) \rangle$. As proven in lecture, this function is computable. Moreover, we claim that it is a mapping reduction from $\overline{A_{TM}}$ to L, and will prove this by showing that for all TMs M and strings w that $\langle M, w \rangle \in \overline{A_{TM}}$ iff $\langle \text{Amp}(M, w) \rangle \in L$.

By the definition of $\overline{A_{TM}}$, we know that $\langle M, w \rangle \in \overline{A_{TM}}$ iff M does not accept w. We claim that this happens iff every string in $\mathscr{L}(\text{Amp}(M, w))$ is a palindrome. We will prove both directions of implication:

($\Rightarrow$) If M does not accept w, then $\mathscr{L}(\text{Amp}(M, w)) = \emptyset$. Therefore, the statement "every string in $\mathscr{L}(M)$ is a palindrome" is vacuously true.

($\Leftarrow$) If M accepts w, then $\mathscr{L}(\text{Amp}(M, w)) = \Sigma^*$. If we choose the alphabet of the amplifier machine to have two or more characters in it (say, **a** and **b**), then not all strings in $\mathscr{L}(\text{Amp}(M, w))$ are palindromes because, for example, the string **ab** is in $\mathscr{L}(\text{Amp}(M, w))$.

Finally, note that every string in $\mathscr{L}(\text{Amp}(M, w))$ is a palindrome iff $\langle \text{Amp}(M, w) \rangle \in L$. Combining these statements together, we see that $\langle M, w \rangle \in \overline{A_{TM}}$ iff $\langle \text{Amp}(M, w) \rangle \in L$. Thus f is a mapping reduction from $\overline{A_{TM}}$ to L, so $\overline{A_{TM}} \le_M L$, as required. ■

Why we asked this question: This question was designed to test your ability to use reductions to prove that a language does not belong to some class (here, **RE**). We expected you to come up with an example of a non-**RE** language that you could reduce to L, then come up with the reduction and prove why it was correct. Our reduction reduces $\overline{A_{TM}}$ to L, though it's also possible to reduce L_e (the set of TMs that accept nothing) or other non-**RE** languages to L.

Common mistakes: This problem was tough. Many solutions tried to perform reductions that would not prove that $L \notin$ **RE** (for example, showing that $A_{TM} \le_M L$ or that $L \le_M L_D$) or reductions that would show that $L \notin$ **RE**, but which cannot be made to work. For example, some solutions tried to reduce EQ_{TM} or A_{ALL} to L, but those languages cannot reduce to L because $L \in$ co-**RE** while neither EQ_{TM} or A_{ALL} are co-**RE**. These reductions would invariably hit a snag and fail somewhere.

Of the reductions that would show that $L \notin$ **RE**, many ran into trouble due to logic errors or invalid assumptions. Some reductions from $\overline{A_{TM}}$ to L tried to map $\langle M, w \rangle$ to $\langle M \rangle$, which does not work correctly. Other reductions involving $\overline{A_{TM}}$ worked correctly, yet made incorrect claims such as "$\langle M, w \rangle \in \overline{A_{TM}}$ iff M rejects w" (make sure you see why this isn't true!)

Finally, some solutions interpreted L as the set of TMs that accepted *every* palindrome, rather than the set of TMs that only accept palindromes.

(ii) Palindromes and co-RE Languages **(20 Points)**

Prove that $L \in$ co-**RE**.

Proof: We will prove that $\overline{L} \in$ **RE**, from which it follows that $L \in$ co-**RE**. Note that the language $\overline{L}$ is the language $\{ \langle M \rangle \mid M$ is a TM and some string in $\mathscr{L}(M)$ is *not* a palindrome $\}$.

Consider the following NTM N:

```
N = "On input ⟨M⟩, where M is a TM:
        Nondeterministically guess a string w.
        If w is a palindrome, then N rejects ⟨M⟩.
        Run M on w.
        If M accepts w, then N accepts ⟨M⟩.
        If M rejects w, then N rejects ⟨M⟩."
```

We claim that $\mathscr{L}(N) = \overline{L}$, from which we get that $\overline{L} \in$ **RE**, as required. To see this, we will prove that there is a nondeterministic choice of w such that N accepts $\langle M \rangle$ iff $\langle M \rangle \in \overline{L}$. To see this, note that there is a nondeterministic choice of w such that N accepts $\langle M \rangle$ iff there is a non-palindrome string w such that M accepts w. This in turn happens iff there is a non-palindrome string w such that $w \in \mathscr{L}(M)$. Finally, by definition of $\overline{L}$, this happens iff $\langle M \rangle \in \overline{L}$. Combining these statements together, we get that there is a nondeterministic choice of w such that N accepts $\langle M \rangle$ iff $\langle M \rangle \in \overline{L}$, so $\mathscr{L}(N) = \overline{L}$, as required. ■

Why we asked this question: This question was designed to test your understanding of nondeterminism and the class co-**RE**. We hadn't asked you to prove that a language is co-**RE** before and intended this question to test whether you were comfortable establishing why a language was co-**RE**. We also hoped this question would assess how well you could design and prove results about NTMs.

Common mistakes: There are many ways to approach this problem, so many of the common mistakes we saw are associated with particular solution routes. Some solutions used reductions to prove that L is co-**RE**. One common mistake was attempting a reduction such as proving $L_D \leq_M L$ that wouldn't prove that L is co-**RE**. Another common mistake was to attempt a reduction from L to $\overline{A_{TM}}$ using a reduction of the form $f(\langle M \rangle) = \langle M, w \rangle$, where w is some string that isn't a palindrome. This reduction doesn't work because $\langle M \rangle$ might not be in L even if it M doesn't accept a particular non-palindrome; to show that $\langle M \rangle$ is in L, you need to show that M never accepts any non-palindromes.

The other common reduction mistake we saw was to try to use a "nondeterministic reduction" to $\overline{A_{TM}}$ by nondeterministically guessing a string w to use. Defining a function as "nondeterministically choose a string" isn't legal, since it doesn't describe the behavior of the function and is not necessarily computable. The other common mistake we saw on this problem was trying to build a nondeterministic co-recognizer for L that guessed a non-palindrome string that M would accept, then rejecting if M accepted the string. Remember that nondeterminism can be used to find an *accepting* path if one exists, and doesn't necessarily find a *rejecting* path if one exists.

75th Percentile: 26 / 45

50th Percentile: 16 / 45

25th Percentile: 9 / 45

Problem Five: P and NP Languages (20 Points)

The above proof, unfortunately, is incorrect. What is wrong with this proof? Be specific. (The function *f* described in the lemma can indeed be computed in polynomial time, so that isn't the error in the proof.)

The error in the proof is that proving *2COLOR* $\leq_P$ *3COLOR* doesn't prove that *2COLOR* is **NP**-hard. As mentioned in lecture, *all* **NP** problems polynomial-time reduce to all **NP**-complete problems, so we already know that *2COLOR* $\leq_P$ *3COLOR*.

Why we asked this question: There are a few critical properties of **P** and **NP** that we want you to remember going forward, and one of them is the directionality implied by **NP**-completeness. To show that something is **NP**-complete, you need to reduce a known **NP**-complete language to it, not the other way around. One of the most common mistakes people make with reductions is doing the reduction in the wrong direction, so we put this question here in the hopes that this knowledge follows you after you leave CS103.

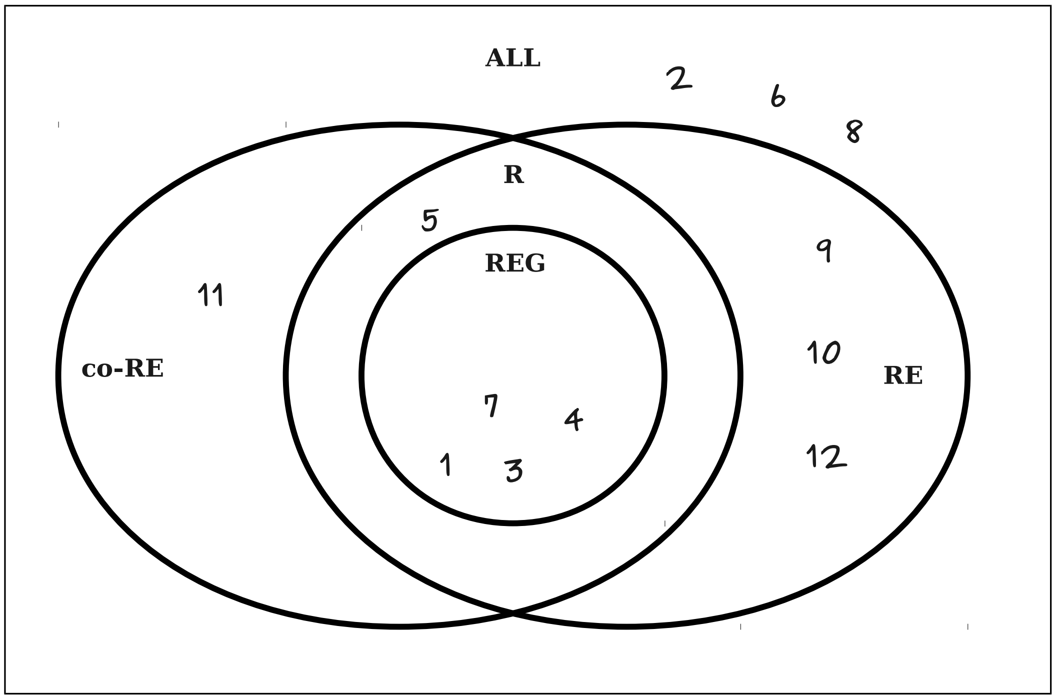
Common mistakes: Almost everyone got this problem right. The most common errors we saw were in solutions that listed two or more errors in the proof or which made incorrect claims about reducibility while justifying exactly why the invalid step was erroneous.

75th Percentile: 20 / 20

50th Percentile: 20 / 20

25th Percentile: 20 / 20

Problem Six: The Big Picture (30 Points Total)



3. { $\mathtt{a}^n$ | $n \in \mathbb{N}$ }

This is regular. It's given by the regular expression **`a*`**.

4. { $\mathtt{a}^n$ | $n \in \mathbb{N}$ and is a multiple of 137. }

This is regular. It's given by the regular expression $\mathtt{(a^{137})*}$.

5. { $\mathtt{a}^n$ | $n \in \mathbb{N}$ and is not prime. }

This is decidable (we can try dividing n by all numbers in the range 2 to $n - 1$ to see if any of them are factors of the number). It's not regular, however, because the complement of the language is the language { $\mathtt{a}^n$ | $n \in \mathbb{N}$ and is prime }, which we know isn't regular.

6. { $\langle M \rangle$ | M is a Turing machine and $\mathscr{L}(M)$ is finite. }

This is neither **RE** nor co-**RE**. Its complement is the language of all TMs with infinite languages, which was proven in one of the discussion handouts to be neither **RE** nor co-**RE**. Its complement, therefore, is also neither **RE** nor co-**RE**.

7. { $\langle M \rangle$ | M is a Turing machine and $\mathscr{L}(M) = L_D$. }

This is (surprisingly!) a regular language. No TM has language L_D, so this set is empty and the empty set is a regular language.

8. { ⟨*M*⟩ | *M* is a Turing machine and $\mathscr{L}(M) = A_{TM}$. }

This is neither **RE** nor co-**RE**. Intuitively, you can't check whether a TM has language A_{TM} because if you feed the TM a string that is not in A_{TM}, the TM might go into an infinite loop. You'd have to be certain that the TM didn't accept in this case, but doing that would require you to check whether it would loop. Similarly, suppose you want to check whether TM has a language that isn't A_{TM}. Then there's some string in A_{TM} that the TM doesn't accept, but to check this you might have to see if the TM loops infinitely on that particular string, which you can't actually do.

9. { ⟨*M, n*⟩ | *M* is a TM, $n \in \mathbb{N}$, and *M* ***accepts*** all strings of length at most *n.* }

This is in **RE**. If a TM accepts all strings of length at most *n*, you can run the TM on all of those strings, watch it accept all of them, and then accept. However, because this requires you to actually watch the TM accept all of the strings, this language can't be co-**RE** because A_{TM} isn't co-**RE**. You can formalize this by using the amplifier machine in a reduction from A_{TM}.

10. { ⟨*M, n*⟩ | *M* is a TM, $n \in \mathbb{N}$, and *M* ***rejects*** all strings of length at most *n.* }

This is also in **RE**. Flipping the accept and reject states of the TM gives a great reduction from this language to language #9 (proving that this language is **RE**) and from language #9 to this language (proving that this language isn't co-**RE**.)

11. { ⟨*M, n*⟩ | *M* is a TM, $n \in \mathbb{N}$, and *M* ***loops*** on all strings of length at most *n.* }

This language is co-**RE**. If a TM doesn't loop on some string of length at most *n*, you can nondeterministically guess that string, run the TM on the string, and watch it halt to prove that the TM doesn't belong to the language. It's not **RE**, though, since no TM could confirm whether a TM loops on some arbitrary string.

12. { ⟨M_1, M_2, M_3, w⟩ | M_1, M_2, and M_3 are TMs, w is a string, and at least two of M_1, M_2, and M_3 accept *w*. }

This language is **RE** but not co-**RE**. If at least two of the three TMs accept, then you can nondeterministically choose those TMs, run those TMs on *w*, and then accept after they accept. This language isn't co-**RE**, however, because you can reduce A_{TM} to it by setting all three TMs to be the input TM M.

Why we asked this question: This question was designed to test a variety of different topics. Some of the languages were designed to see if you understood closure properties of different types of languages. Others were designed to test your intuition behind the **R**, **RE**, and co-**RE** languages. A few were designed to test your understanding of nondeterminism. Language #7 was specifically chosen to see if you understood that the empty language was regular, while Language #8 was designed as a challenge to see if you could see why it would be hard to check if a TM was a universal Turing machine.

Interestingly, this question is the most tightly correlated with overall exam scores.

75th Percentile: 21 / 30

50th Percentile: 15 / 30

25th Percentile: 12 / 30