CSE 480 – Final Exam Name: Jeremy Duong

This exam is open book, and open note. You are welcome to use the internet to learn more about these topics generally, but please don't search and find the answer to the exact question here. I trust you to use proper judgment and understand what this means. If you do use the internet, please cite your sources.

You are welcome to use additional paper if you feel you need to, but I have tried to be generous in the space allowed to accommodate what I anticipate are reasonable answers.

Have fun!

1. a) In your own words, briefly define each of the following classes. (I am looking for a sentence or less for each).

- EXPSPACE

  The class of languages where, in order to decide whether a string belong to them, it takes a Turing machine **exponentially as much space as the length of the string itself** in order to do so.

- EXPTIME

  The class of languages where, in order to decide whether a string belong to them, it takes a Turing machine **exponentially as many steps of operation as the length of the string itself** in order to do so.

- L

  The class of languages that are **decidable by a deterministic Turing machine in Logarithmic Space.**

- P

  The class of languages that are **decidable by a deterministic Turing machine in Polynomial Time.**

- PSPACE

  The class of languages that are **decidable by a deterministic Turing machine in Polynomial Space.**

- NL

  The class of languages that are **decidable by a non-deterministic Turing machine in Logarithmic Space**

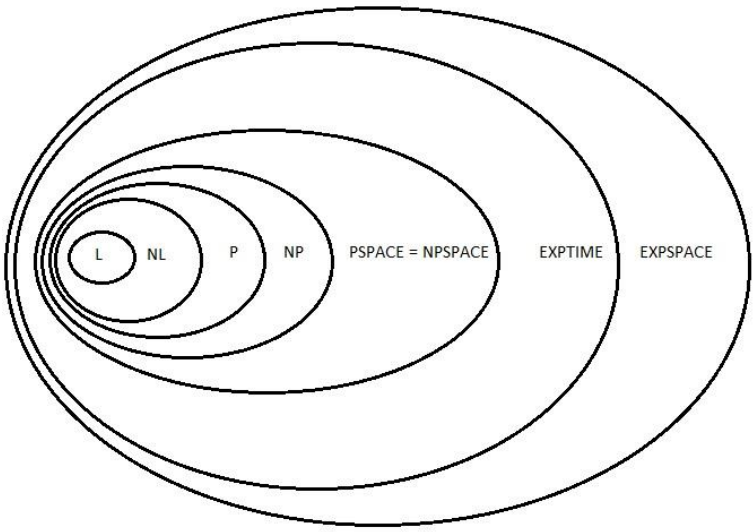
- NP

  The class of languages that are **decidable by a non-deterministic Turing machine in Polynomial Time.**

- NPSPACE

  The class of languages that are **decidable by a non-deterministic Turing machine in Polynomial Space**

b) Draw a Venn Diagram showing the known relationships among the above classes.



c) List any containment relationships that have been shown to be proper (i.e., where we have shown that two classes cannot be equal).

P ⊊ EXPTIME	(Corollary 9.13)
NL ⊊ PSPACE	(Corollary 9.6)
PSPACE ⊊ EXSPACE	(Corollary 9.7)

2. Describe the P vs NP issue. For example: What does this issue refer to? Why is it so consequential? What do most current researchers think about the issue?

The P vs NP issue revolves around the question: Are the P-class and NP-class the same class of languages? For the longest time people in the field of computer science have been trying their best to either prove that they are the same, or that they are NOT the same. However, no one has been able to prove one way or the other.

The reason why this issue is so consequential and people care about it so much is because if the answer to the question is "yes," and P and NP are the same class of languages, then a lot of problems we think can only be solved non-deterministically in polynomial time (not practical) can now be solved deterministically in polynomial time, which means "practically fast." These problems happen to include those such as decryption, breaching security, and a host of other problems that hold the internet together, which up until now have been, of necessity, designed to be hard to solve. Thus, if someone found a way to prove that P and NP are the same, all of these important problems would be solved really fast, causing the whole internet to fall apart.

Even though no one has been able to prove it, current researchers believe that P and NP must be different since no one has been able to prove otherwise either.

3. Define "mutual friends" in a graph to be a group of three nodes where all three are connected. *FRIENDS* = {<*G*> | *G* is a graph than contains a group of mutual friends}. Show that *FRIENDS* $\in \boldsymbol{P}$.

To show that FRIENDS is in P, we design a deterministic Turing machine M to decide FRIENDS in polynomial time. There is no mention of whether G should be a directed or undirected graph so we will assume G is undirected. M works as follow:

M: "On input <G>:

1. For each edge of form (a, b), with a and b being 2 nodes in G:
    a. Look through all other edges:
        i. If an edge of form (x, c) is found where x = a or x = b, look through the other edges again to find an edge of form (b, c) if x = a, or of form (a, c) if x = b. If all the desired edges are found, then **ACCEPT**.
2. If all edges have been looked at by step 1 and the machine hasn't accepted, then **REJECT**."

Notice that step 1 attempts to look through all the edges, so the time complexity of step 1 is O(n). At each edge that step 1 looks at, it looks at all other edges again in step a, so step a is also O(n), and the combination of step 1 and step a is $O(n^2)$. Then, while step a is running, every time the machine finds an edge that contains a node that the current edge also contains, it attempts to loop through all other edges again the third time in step i to find another edge with

both of the other 2 nodes. Step i attempts to look through all the edges as well, so it is also O(n), and the combination of step 1, step a, and step i is $O(n^3)$.

We see that Turing machine M might not be the most efficient, it is indeed deterministic and operates in polynomial time $O(n^3)$. As such, we see that FRIENDS is in P.

4. Consider the problem of scheduling final exams on campus. Assume you have:

- ***F* – The set of all final exams.**
- ***S* – A list of the final exams for each individual student (i.e., this is a list of sets, where each set is a subset of F, namely the set of finals that a particular student is taking).**
- ***h* – The number of final exam spots available on campus that can be scheduled.**

***FINALS* = {<*F, S, h*> | Where all finals *F* can be scheduled in *h* slots such that no student is taking two exams at the time time}. Show that *FINALS* $\in \boldsymbol{NP}$.**

To show that FINALS is in NP, we build a **verifier V** for FINALS that takes in input <<F, S, h>, c>. The certificate **c** that we use in this case is the appropriate schedule for the situation. **c** is the same as F, which is the set of all final exams, except each final exam in **c** is coupled with a number that expresses which timeslot that exam is scheduled in h.
So if F looks like this:

F = <EXAM1, EXAM2, EXAM3,…>

…then **c** would look something like this:

c = <<EXAM1, h_1>, <EXAM2, h_2>, <EXAM3, h_3>,…>, with h_i <= h.

For simplicity, we will design V to have 2 tapes: one for the input, the other is a work tape. Because of theorem 7.8, we know that if V runs in time t(n), its single tape version will run in time $O(t^2(n))$. As such, if V runs in polynomial time, its single tape version will also run in polynomial time. Verifier V will work as follow:

V = "On input <<F, S, h>, c>:

1. If input is not of the expected form, or if **c** contains an exam that is not in F, or if **c** contains $h_i > h$, then **REJECT**.
2. For each student $S_i = \{E_1,\ldots, E_k\}$ in S, where E_1 through E_k are the exams that student needs to take:
    a. Look up each E_i in **c** and record its corresponding $\mathbf{h_i}$ on the 2nd tape.
    b. Once all $E_i \in S$ have been looked up in c, traverse through tape 2 to see if there is any repeated value. If there is a repeated value, **REJECT**. If there isn't a repeated value, erase tape 2 to prepare for the next student.
3. If all the students have been looked at by step 2 and the machine hasn't rejected, then **ACCEPT**."

We can see that step 1 takes at most $O(n^2)$ time, since it needs to traverse through the whole input to see if it's the right form, then go back and forth between c, F, and h to see if c contains any final exam or h value that is invalid.

Step 2 looks through all the students so it's O(n). Step a looks through each exam the student needs to take, then look up the corresponding for each h value in c, so this step might take $O(n^2)$ time. Step b looks through all the h values corresponding to each exam the student is taking, and then look for the repeated value at each h value, so step b also take $O(n^2)$ time. The combination of step a and b is $O(n^2) + O(n^2) = O(n^2)$. The combination of step 2, a and b is $n * O(n^2) = O(n^3)$.

Step 3 doesn't traverse through anything so it's O(1).

So the total time complexity of verifier V is $O(n^2) + O(n^3) + O(1) =$ **$O(n^3)$**

Because we can construct a **polynomial time verifier V** for FINALS, by definition 7.19 (pg. 294), we know that **FINALS must be in NP**.

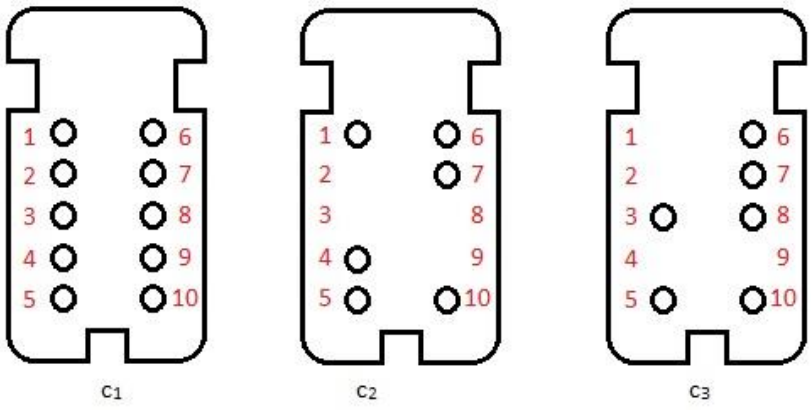
5. Please see Problem 7.28 on Page 325 in the textbook. Show that this language, *PUZZLE*, is NP-Complete.

Ok, this one is going to be long… 😊

To prove that PUZZLE is NP-complete, by definition 7.34 (pg.304), we need to prove 2 things:

1. PUZZLE is in NP, and
2. Every A in NP is polynomial time reducible to PUZZLE

Before proving PUZZLE is in NP, I think it's helpful to talk about how each card can be represented as a string of input. From the problem, we see that each card is unique because of the way the holes are configured. Since there are 2 columns of holes, we define the maximum number of holes each card can have to be number n, where n is an even number. Bellow is an example of cards where n = 10 because the maximum number of punched holes each card can have is 10.



Each hole position can be a punched hole or an un-punched hole (or no hole). We can represent each hole position with a bit where 0 means that the hole is punched and 1 means that the hole is not punched (no hole). As such, we can represent the 3 cards above as follow:

c_1 = <0000000000> (it has a hole in every position)
c_2 = <0110000110> (there are 4 positions where there is no hole: 2, 3, 8, and 9)
c_3 = <1101000010> (there are 4 positions where there is no hole: 1, 2, 4, and 9)

Generally, a card will be presented as followed:

$c_i = <b_1 \ldots b_n>$ for $1 <= i <= k$ (k is the number of cards)

Notice that when we flip a card, we simple switch the bit at position *i with position (i + n/2).* For example, when we flip one of the above cards, we switch position 3 with position (3 + 10/2) = 8, which are the 2 positions right next to each other. This might become important later on.

1. <u>PUZZLE is in NP:</u>

Now that we figured out how each card is represented, let's prove that PUZZLE is in NP. In this case, I think the simpler way to prove this is to build a non-deterministic Turing machine M that decides cards. Machine M will work as follow:

M = "On input $<c_1, \ldots, c_k>$:

1. Determine the number of hole positions n on each card by counting the bits in each card.

2. Look through each card individually and **non-deterministically choose** one of the 2 ways to flip the card (one branch leaves the card as it is and the other branch switches bit *i* with bit (*i + n/2*) for $1 <= i <= n/2$).

3. Once step 2 is done, each branch of the machine traverses through all the hole positions from 1 to n. For each hole position, traverse through all the cards again to see if there is any 1 at that specific hole position. If no "1" is found, **REJECT**.

4. At any branch, if step 3 finishes without rejecting then **ACCEPT.** Obviously, if all branches reject then reject."

We see that step 1 is O(n) because it needs to loop through the number of possible hole positions on each card by counting the bits in a card. Step 2 is potentially $O(n^2)$ since the machine needs to go through every card, then for each card the branch that decides to flip the card must go through all the holes. Step 3 traverses through all hole positions and all

the cards for each hole, so it's time complexity is also $O(n^2)$. Step 4 doesn't count since it's just an accept if step 3 doesn't reject. So, the time complexity of M is:

$$O(n) + O(n^2) + O(n^2) = \mathbf{O(n^2)}$$

Since M is a **Non-deterministic TM** that decides puzzle in **polynomial time $O(n^2)$**, we know that **PUZZLE is in NP**, according to Theorem 7.20 on page 294.

2. Every language A in NP is reducible to PUZZLE:

PROOF IDEA: We prove that **3SAT** can be reduced to PUZZLE in polynomial time. Then, because 3SAT is NP-complete, PUZZLE is also NP-complete (Theorem 7.36 on page 304).

PROOF: To show that 3SAT can be reduced to PUZZLE, we create a polynomial time algorithm that converts the Boolean formula Φ that 3SAT has to deal with into a special deck of cards that can be decided by M.
Before we do the reduction, we must notice first the similarity between PUZZLE and 3SAT. The goal of PUZZLE is to decide whether all the hole positions can be covered by all the cards. To cover 1 single hole position, **we only need 1 single card** on which that specific hole position is unpunched (bit value of 1). This works the exact same way as the OR statement that SAT needs to deal with: to make a clause evaluate to true, **we only need 1 variable to be true**. To make sure that the bottom of the box is covered in PUZZLE, we need to make sure that **all the hole positions are covered**. Again, this works exactly like the AND statement in 3SAT: to make the whole expression Φ evaluate to true, **all the clauses must be true.**

From this observation, we see that the reduction can work as follow:

- Each **clause** is equivalent to **a hole position on the left column** of a card. To make things simple, we don't care about the right column. To make the right column insignificant, we create a special card whose holes are all punched on the left, but none is punched on the right. Whether the expression is satisfiable depends on the ability of the deck (excluding the special card) to cover the left column.

- Each **variable** is equivalent to a **card**. Flipping a card means switching the truth value of a variable. If a variable is present in a clause, the hole position corresponding to that clause on left column of the card is unpunched (bit value 1), and the other hole position on the right column is punched (bit value 0). If its complement is present in the clause, then we have "0 – 1" instead of "1 – 0." This means that most of the time, each row of hole positions will contain 2 positions that are always opposite to each other. There are only 2 exceptions:
    - First, we notice that **if a variable and its complement are both present** in a clause, then both positions on a row corresponding to that clause are

unpunched (bit value 1) since ($\overline{X}$ v X v Y) always evaluate to true so that clause is guaranteed to be true regardless of what X and Y are.

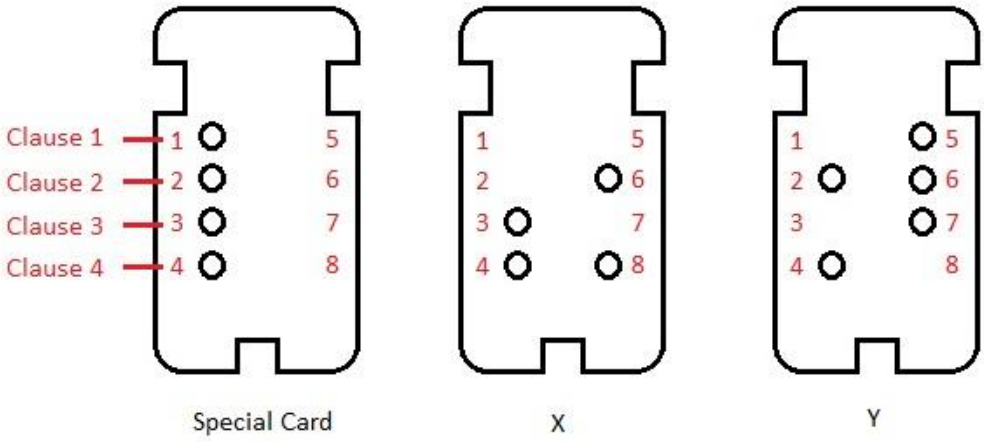
- Second, if a **variable is not present in a clause**, then its value is inconsequential to the value of the clause, therefore, both hole positions will be punched (bit value 0).

Let's look at one example of how this reduction is done. Let Φ be the 3-conjunctive-normal-form Boolean expression:

$$\Phi = (X \vee \overline{X} \vee Y) \wedge (X \vee X \vee X) \wedge (\overline{X} \vee Y \vee Y) \wedge (\overline{Y} \vee \overline{Y} \vee \overline{Y})$$

Since there are only 2 variables X and Y, there will be only 2 cards other than the special card. There are 4 clauses, so there will be 4 rows of holes on each card.

The deck of cards that will be created from this expression are shown below:



Notice that for variable X, since both X and $\overline{X}$ are present in the first clause, both holes on the first row are unpunched (bit value 1). In clause 2, X is present but not its complement, the left hole is unpunched (bit value 1), and the right hole is punched (bit value 0). In clause 3, $\overline{X}$ is present but not X, so its left hole is punched (0), and its right hole is unpunched (1). In clause 4, X is completely not present, so both holes are punched (0). The same process is used to determine which holes to be punched on the Y card.

With this set up, we see that the deck of cards created satisfies PUZZLE only if all the cards other than the Special Card can cover at least 1 column. If they are unable to cover at least 1 column, then even with the present of the special card, they still can't cover the bottom of the box. To make things simple, let's just look at the left column. Clause 1 (hole position 1) will always be covered since both position 1 and position 5 on X card are unpunched. Clause 2 (hole position 2) can only be covered if X = 1, which means that the X card cannot be flipped the other way. This means that on clause 3, Y has to be 1 (not flipped) to cover the left hole (hole position 3). However, with this set up, hole number 4 clearly cannot be

covered. As such, we see that the collection of these 3 cards are NOT in PUZZLE, and thus the expression Φ is NOT in 3SAT.

(Note: we use the left column as the column of concern to make it easy to visualize. However, if the left column cannot be covered by the all the cards other than the special card, that means that neither one of the 2 columns can be covered by them)

Now, let's generalize the reduction algorithm and analyze its time complexity:

"On Boolean formula Φ:

1. Go through Φ and count the number of clauses k. We know that 2k = n (number of hole positions. So this helps us know the number of hole positions each card has.
2. Create a Special Card $c_s = <0^k1^k>$.
3. Go through Φ again. For every new variable encountered, create a new card:
   $c = <b_1b_2…b_{2k}>$
4. For each card created, go through each clause *i* (1 <= *i* <= k):
   a. If the variable is not found in clause *i*, punch 2 holes on row *i* of the card. This means punching holes on position *i* and (*i + n/2) or i + k.*
   b. If it (but not its complement) is found in a clause, leave the left hole unpunched and the right hole punched on row *i* of the card. If only its complement is found, do the opposite.
   c. If both it and its complement is found, leave both holes unpunched on row *i* of the card.*"*

Once the algorithm is done, we have a deck of cards ready to be fed into machine M that decides PUZZLE, which result answers our 3SAT question.

Step 1 of the algorithm runs in time O(n). Step 2 also runs in O(n) since it needs to change n bits of the Special Card. Step 3 can potentially run in time $O(n^2)$ since for each variable it encounters, the head of the machine needs to go back and check to see if it has encountered it. Step 4 goes through all the cards created, and then, for each card, it needs to go through all the clauses to do the appropriate hole-punching for each card. So the time complexity of Step 4 is $O(n^2)$. The combination of all 4 steps is:

$O(n) + O(n) + O(n^2) + O(n^2) =$ **$O(n^2)$**

We see that the algorithm to reduce 3SAT to PUZZLE runs in polynomial time. In other words, **3SAT is polynomial time reducible to PUZZLE**.

Since 3SAT is NP-complete, we know that PUZZLE is also NP-complete (Theorem 7.36), thus, ended our proof.

6. Define *BALANCED-BRACES* to be the language of strings over the alphabet *{ () }* (in other words, opening and closing parentheses) where the parentheses are properly balanced/nested. In other words, the strings "( )" or "(( ) (( ))) ( )" or "(( ) ( ))" are in the language, but ") (" or "(( )" or "( )) ( )" are not. Show that *BALANCED-BRACES* $\in L$.

To prove that BALANCED-BRACES are in *L,* we will build Turing machine M that decides it using Log space. We will make it so that machine M has 2 tapes as described by Sipser on page 349: One tape is a read-only input tape while the other is the work tape. Only the cells scanned on the work tape contribute to the space complexity of machine M.

M = "On input *w*:

1. Look through each character of the input and count the number of "(" and, separately, the number of ")" in binary. The space on the work tape is used to record these 2 binary counters. Once this is done, check the 2 counters. If they are not the same, then REJECT. If they are the same, continue to step 2.
2. Now the machine looks again through the input and make sure they are in the right order. To do this, the machine will need to **keep track of 2 values on the work tape**: the position of the last "(" found and the position of the last ")" found. We will call these 2 values **A and B** for convenience. At the beginning, the initial values of these 2 numbers are A = "$" and B = "$".
    a. The machine moves the head to the start of the input tape and look for "(".
        i. If A = "$", it makes sure that the first value in the input is a "(". If it is, then update A to "0" (binary value of leftmost input cell), then move on to step b. If it is not then REJECT.
        ii. If A is a binary number, it increments the head until it matches that position, then continue to go right, looking for the next "(". If the next "(" is found, it updates A to the current position, then continue to step b. If the head reaches the end of the input without finding "(", then ACCEPT.
    b. Starting from the position that A holds, the machine tries to go right.
        i. If the B = "$", the machine simply continues to move right to find the first ")". If it is found, update B to the position of that ")" character. If it is not found, REJECT.
        ii. If the position of B is a binary number, increment the head until its position reaches B, then continue to move right, looking for a ")". If it is not found, REJECT. If it is found, repeat step a."

Note that this algorithm does not leave out any open parenthesis "(", but only look for closing parenthesis ")" on the right of the last "(" found to match it. As such, if any closing parenthesis is out of order, it will not be counted, which causes the last search for a closing parenthesis ")" to fail and the machine to REJECT.

Now, let's analyze the space complexity of M. Step 1 needs to store 2 binary counters. Since a unary number of length n takes up log(n) space when stored in binary, clear the 2 counters take up 2 * O(logn) = **O(logn)** space. In step 2, the machine needs to store 2 values A and B that specify a position within the input. Again, since these numbers are expressed in binary, they clearly take up **O(logn)** space with n being the length of the input. The combine space complexity of step 1 and step 2 is O(log n) + O(log n) = **O(log n).**
Thus, this proves that **BALANCE-BRACES is in *L*.**

7. **Karl has developed a non-deterministic algorithm that can run in $O(n^3)$ space. His algorithm is tight and clean and doesn't show any clear ways for improvement. His brother, Kevin, claims to have a deterministic solution for the same problem that runs in *PSPACE*.**

What do you think about the possibility of Kevin's claim?

Because Karl's non-deterministic algorithm runs in $O(n^3)$ space, we know that the problem he is solving is in **NPSPACE** (non-deterministic polynomial space). Specifically, it runs in $NSPACE(n^3)$. By Savitch's theorem, we know that **NPSPACE = PSPACE**. For Karl's problem in specific, we see that by applying Savitch's theorem, we get:

$NSPACE(n^3) = SPACE((n^3)^2) = SPACE(n^6)$, which is still in **PSPACE** (pg. 334).

This means that **any problem in NPSPACE is also in PSPACE and vice versa**. Therefore, it is completely reasonable for Kevin to be able to claim that he has an algorithm that solve Karl's problem in **PSPACE**.

8. Recall that a coloring of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color, and that *3COLOR* = {<*G*> | *G* is colorable with 3 colors}.

Assume you had an oracle that could decide the *3COLOR* problem. What impact would this oracle have on solving the *FINALS* problem discussed above? Explain how you might make use of the oracle.

An Oracle allows a machine to decide if a string is a member of a language in one single computation step or constant time O(1). As such, having a 3COLOR oracle allows any Turing Machine to decide 3COLOR in constant time.

Recall that 3COLOR is an NP-complete problem (we proved this in problem 7.29), which means that all problems in the class NP can be reduced to it in polynomial time. Since we just proved that FINALS is in NP, we know it can be reduced to 3COLOR in polynomial time. So, if we have an Oracle that decides 3COLOR, we can solve FINALS by reducing it to 3COLOR and then query the oracle to decide it, which would happen only in polynomial time. In short, we can solve FINALS **deterministically in polynomial time.** So, all of a sudden, FINALS is as easy as any problem in P.

I do have one other observation, though I might be wrong about this, that an oracle only tell you Yes or No. So if we have an oracle for 3COLOR, using the process above, we can gain an advantage in that we can decide FINALS a lot quicker. However, the oracle certainly doesn't tell you how a task is accomplished. With our FINALS problem, our Turing Machine needs to decide if an input <S, F, h> is schedulable (Yes or No), but we might also care about what that specific schedule that is the solution to the problem is. That sometimes is more important than just knowing whether it can be done. Since an oracle only tells you Yes or No without telling you how, if you actually care about the specific solution to the problem, meaning the actual schedule that works, then I don't think an oracle will be able to help much.

9. Consider the language *CATS* that contains the binary representation of .PNG files that contain a picture of a cat. Emily has developed a probabilistic algorithm that determines whether a given picture contains a cat, but the algorithm is not great. When the picture contains a cat, it only accurately predicts it with 60% accuracy. When the picture does not contain a cat, it accurately rejects it 58% of the time.

A) Is *CATS* $\in$ $\boldsymbol{BPP}$? Explain.

From definition 10.4, we see that "BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of 1/3" and "any constant error would yield and equivalent definition as long as it is strictly between 0 and 1/2" (pg. 397).

Because Emily's algorithm is 60% accurate when the picture contains a cat, and 58% of the times when the picture doesn't contain a cat, we can say that the overall Emily's algorithm is accurate more than half the times. In other words, her algorithm is more than 50% accurate. This also means that the error rate is "strictly between 0 and 1/2." As such, by definition, we see that CATS $\in$ BPP.

B) Is *CATS* $\in$ $\boldsymbol{RP}$? Explain.

From definition 10.10, we know that "RP is the class of languages decided by probabilistic Turing machines where inputs in the language are accepted with a probability of at least 1/2, and inputs not in the language are rejected with a probability of 1." (pg. 403) This is called a one-sided error, which means that one of the 2 predictions are always correct.

We see that Emily's algorithm decides CATS accurately 60% of the times when the input is a cat (more than 1/2). However, it rejects a picture correctly only 58% of the time. This means that there exists an uncertainty that it might incorrectly reject an input it's supposed to accept. So, Emily's algorithm does not have the "one-sided error" which is characteristic of an algorithm that decides a language in RP. At this point, it is tempting to say that CATS is not in RP.
However, I doubt this answer a little bit because just because Emily's algorithm doesn't have the one-sided error doesn't mean that an algorithm that does have this property that also decides CATS doesn't exist. As such, **my answer is that it might be, I just don't have enough information to tell.**

C) Explain how you could reduce the error of Emily's algorithm so that it is substantially more accurate.

Because the algorithm Emily came up with has an error rate that is less than ½, we see that as we run this algorithm multiple times, we can get this error rate to decrease exponentially. The first time, the chance that the machine makes an error is ½. The probability that the machine guesses it wrong a second time is (½)^2 = 1/4 . As such, the more we run, the smaller the error exponentially gets.

10. Which theorem from the second half of the semester did you find the most interesting? Why?

After working on problem number 5, I think my favorite theorem is **theorem 7.37**, which says that SAT is NP-complete, and the corollary that follows, **corollary 7.42**, which says 3SAT is NP-complete. What I like about these 2 problems is that they can be easily mapped to a lot of problems that we try to prove the NP-completeness of. This is because a lot of problems are solved just by examining some Boolean expression. If such Boolean expression evaluates to true then accept, if not then reject. SAT and 3SAT deal with exactly this, that's why they are so useful in proving NP-completeness.