CS 3100, Spring 2020,
Final Battery of Problems (solutions, if present, in sf font)
(These will be adapted for Canvas)

1 WHAT IS ON Midterm-3? What to Study?

Midterm-3 will cover Chapter 13, 14, selected items from Chapter 15, and my videos and slides available online on the class syllabus `bit.ly/3100S20Syllabus`. Here are some specific topics to study:

- Asg-6 solutions.
- How to design TMs and how to read a TM description and "mentally calculate" its language. (Think of all strings that seem to take it to a final state; summarize these strings via an RE.) This will be Question-1 worth a lot of points.
- RE and Recursive Languages. Chapter 14, and the proofs in 14.5.
- Diagonalization proof in 15.4.
- The PCP Chapter till about 15.3.
- 14.5.3 that discusses Linear Bounded Automata. Study this in connection with Figure 14.2 (that was discussed in detail in one lecture) and Figure 13.16.

2 WHAT IS ON THE FINALS?

- Basic material covering whole course
- Mapping Reductions (Chapter 15.5 and Assignment 7)
- Chapter 16 on NP-Completeness
  - What do these mean:
    - P, NP
    - NP-Hard, NP-Complete
    - Define NP-Complete in two ways:
    - Way-1:
      - In NP
      - All of NP mapping reduces to the new problem (the new problem is NP-Hard)
    - Way-2:
      - In NP
      - A specific NPC problem mapping reduces to the new problem (still achieves showing that the new problem is NP-Hard)
    - Why are these ways equivalent?
    - What is the real danger of "showing" a problem to be NPC by simply showing that it is NPH (and forgetting to show that it is in NP)? (Answer: See Theorem 16.7.1 and its proof. It tells you that NPH-problems could be undecidable.)
  - Mapping-reductions in NPC-land:
    - What is the basic nature of these mapping-reductions?
    - How does showing new problems to be NPC work (taking advantage of already known NPC problems)?
    - How was the first NPC problem shown to be so?

        · The definition of NP only says there exists an NDTM P-time decider. We don't know exactly how long it takes. Then how did the mapping reduction of the first NPC problem really work? (Discussions on Page 256, just before 16.5.3)

  – How likely is it that a problem is NPC and its complement is NP? How does this result help predict new results?
  – What is the complexity of checking for primality of a binary- (or decimal)-encoded number?
  – What is know about *factoring* a prime number and its complexity?
  – What is a conjunctive normal form (CNF) formula?
  – What is a disjunctive normal form (DNF) formula?
  – CNF-sat is NPC; DNF-sat is linear; CNF formulae can be converted to DNF; so why can't we "beat" the NPC complexity of CNF-sat by converting it into DNF first and checking there?
  – 3SAT is NP-Complete; does this mean that nobody should use SAT-checking tools?
  – Name four applications of SAT-checking tools in practice

- Chapter 17 on BDDs
  – What are BDDs? How do they relate to minimal DFA?
  – Why do BDDs play an important role in practice?
  – What is one thing that governs the size of a BDD? When can we hope to have compact BDDs? Is this always possible?
  –
- What is diagonalization? How did we show that there are more languages than RE languages? (Read Appendix C of the book.)

3 ASSORTED PROBLEMS

Questions below give you practice pertaining to Asg-6 and Asg-7.

1. Is $\emptyset$ recursive?

   Answer yes. There is an algorithmic membership test. Algorithm: always return false.

2. Is $\{1, 2, 3\}$ recursive?

   Answer yes. Implement a C function (or TM) that given 1 or 2 or 3 returns True else False.

3. Is $LanguageOf(0*1*)$ recursive?

   Answer yes. Implement a pattern matcher that matches for that regexp.

4. Is $LanguageOf(CFG\ G)$ recursive?

   Answer yes. Implement a parser for G.

5. Is $LanguageOf(PDA\ P)$ recursive?

   Answer yes. Run the pda on the given membership test string.

6. Is $LanguageOf(TM\ M)$ recursive?

   Answer no. The proof that A_{TM} being not recursive, from first principles. (We build the D machine and ask what happens to $D(D)$.)

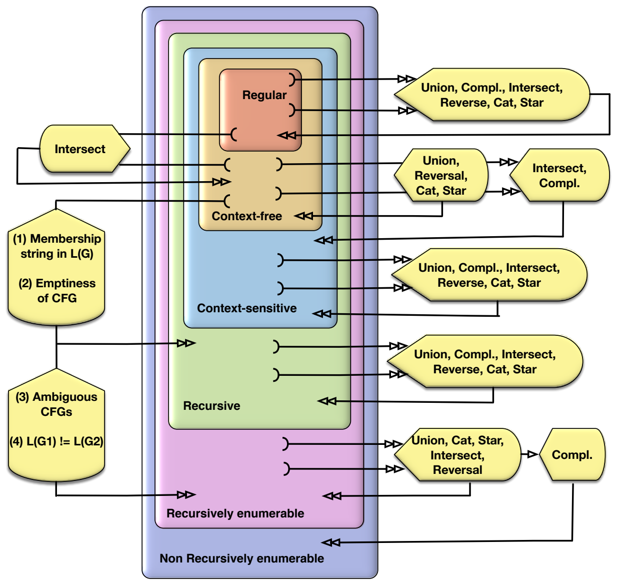
7. Is $LanguageOf(TM\ M)$ RE?

   Answer yes. Dovetail list strings from Σ^* and run M in tiny increments of fuel (ever-increasing) on those strings. Whenever M accepts a string, list that string. That enumerates the language of M.

8. Is the language called A_{TM} RE?

   Answer yes. Dovetail list pairs of strings from Σ^* such that the first pair is a Turing machine (or C program) M and the second is w. M in tiny increments of fuel (ever-increasing) on w. Keep doing this for various M and w. Whenever M accepts some string w, list that pair That enumerates the language of A_{TM}.

3.1 The language hierarchy


Regular
Union, Compl., Intersect, Reverse, Cat, Star
Intersect
Union, Reversal, Cat, Star
Intersect, Compl.
Context-free
(1) Membership string in L(G)
(2) Emptiness of CFG
Union, Compl., Intersect, Reverse, Cat, Star
Context-sensitive
Union, Compl., Intersect, Reverse, Cat, Star
(3) Ambiguous CFGs
(4) L(G1) != L(G2)
Recursive
Union, Cat, Star, Intersect, Reversal
Compl.
Recursively enumerable
Non Recursively enumerable

1. Given the language hierarchy (Figure 14.2, Section 14.4, reproduced on this page), answer the following:

   (a) Is $L_{emptycfg} = \{\langle G \rangle\ :\ L(G) = \emptyset\}$ recursive? Here, G is a CFG.

   Answer: Yes. Keep sweeping from the start symbol of G to see if G gets caught up in infinite recursion. Actually sweep from the terminals back to the S (start) symbol of G. Somehow if G manages to produce any string s at all, then $L(G) \neq \emptyset$. Else it is equal to $\emptyset$.

   (b) Is $L_{G1eqG2} = \{\langle G_1, G_2 \rangle\ :\ L(G_1) = L(G_2)\}$ RE, where G_1 and G_2 are CFGs?

   Answer: No. If you find a string w accepted by one grammar but not the other, then the grammars are not language-equivalent. But if you keep finding all w within the language of both grammars, you can't stop. Intuitively, grammar equality is not established with one string accepted by both. One has to "exhaust" all strings. The formal proof was discussed as a canvas response but not needed.

2. From the language hierarchy, we notice that regular languages are contained in the family of context-free languages, but not vice-versa. On the other hand, the set Σ^* is regular, and every CFL is a subset of this regular language. Explain!

   Answer: One talks about language inclusion, the other talks about string inclusion.

3. Prove that there exist non-RE languages. Argue how each RE language can be represented by a bit-vector, calling the bits

```
L0  b00 b01 b01 ...
L1  b10 b11 b12 ...
L2  b21 b22 ...
L3 ...
```

   Now consider the complemented diagonal. Write it out. Argue how it compares with each of the languages L0, L1, ...

   Detailed Answer: Attempt to show that all TMs can be "numbered" (listed out). Thus, all the RE languages can be numbered. That is what I mean by L0, L1, L2, ... above. But now, one can find ONE language not in ANY of these languages. This is discussed in Appendix-C of the book (last "chapter").

4. How to write a proof for a language L being RE.

   Answer: In general, state an approach to list L via an enumerator Turing machine M. Assume that the notion of "dovetail order" is well defined. Then, here is the pseudo-code for listing the language of TM M, which is the TM whose language L is:

```
For TM M whose language is an RE set, here is how we enumerate its language:

// Enumerate the language of a TM called M
//
TM_enumerator(M) {
<fuel,enumLen> = <0,0>;
//we advance in dovetail order i.e. 00,01,10,02,11,20,...
// i.e. all pairs that add up to N, then pairs that add up to N+1..
 forever do {
  generate next <fuel,enumLen> in dovetail order;
  for each x in Sigma* upto enumLen in numeric order
    { run M(x) with fuel;
      if M accepts x, print x;
    }
 }
}
// We would run M on every x in Sigma* for every amount of fuel
// If at all x in the language of M, we will discover it via this enumeration
```

5. How to write a proof for a language L being recursive.

   Answer: In general, state the algorithm, or state how one can enumerate L and its complement.

6. (a) Prove that the set of Turing machines T that halt when started with input string w, and that do not write outside of positions p_1 and p_2 on the tape ($p_1 \leq p_2$) is recursive.

   Answer:

   i. Notice that the TM is not allowed to write more than a finite number of cells.

   ii. Keep simulating this TM, observing the IDs attained by the TM.

iii. The number of IDs is finite. So when the IDs begin repeating and the TM has not yet halted, we can conclude that the TM will never halt.

iv. Following the aforesaid algorithm, we can conclude that this set of $\langle T, w, p_1, p_2 \rangle$ is recursive.

(b) Prove that given an initial chessboard (pieces placed as usual) and a final checkmate position P, the moves (a string m) necessary to attain such a checkmate are decidable. That is, this language is recursive:

$$\{\langle P, m \rangle \ : \ P \text{ is a checkmate achieved via } m\}$$

Answer: Chess moves can be algorithmically checked.

7. Are RE sets closed under complementation? Answer:

If an arbitrary set (or language) S is RE and also $\overline{S}$ is RE (assuming closure under complementation), then we can enumerate S and $\overline{S}$ and decide membership of any given string in S or $\overline{S}$. This makes S recursive.

Now answer this question. S is an arbitrary set. Is any arbitrary set recursive? Answer: No! A_M is not.

RE sets are closed under union, concatenation, intersection, reversal. Here is a taste for one proof (closure under intersection); write the others:

Answer:

- Define the following TM that takes M_1 and M_2:
  - intersectM1M2(M_1,M_2):
  - run M_1 and M_2 in dovetail order on inputs, also generated by the dovetail enumeration procedure
  - When this simulation notices M_1 and M_2 accepting some string, it emits that string
  - This procedure now lists all strings in M_1 and M_2.

To recap, if a set S and its complement $\overline{S}$ are RE, then both S and $\overline{S}$ are also recursive. Also, every recursive set is RE.

8. Show that the acceptance problem is undecidable (not recursive), but the language A_{TM} is RE. Also, the Halting problem is undecidable, but the language $Halt_{TM}$ is RE. Answer: The RE part is shown by defining an enumeration as in Question 7. The undecidable part is by the diagonal machine "D" or the corresponding diagonal machine "E" defined during Lecture 26 on Tuesday, Nov 27th.

9. Direct proof by contradiction of A_{TM} being undecidable, using $Halt_{TM}$ being undecidable as the "old existing undecidable problem." Answer: Review the "boxes within boxes" construction that involved the two boxes for A_{TM} and the "edit" feature.

10. The idea of mapping reductions

(a) What sort of functions are required in the mapping process, and why?

(b) Answer for computability and complexity.

Answer: For computability, we need the mapping reduction function to be a turing-computable function that is defined on all inputs (an algorithm). The `Translate` is an example of a mapping reduction function that prints M'.

11. What is a Boolean formula? (You may have called it *propositional formula* in CS 2100.)

12. What is a conjunctive normal form? Disjunctive normal form? What is meant by validity? Satisfiability? Write down some examples. Answer: Class notes of the last week.

13. What are BDDs? How to show that a formula is Sat/Valid using BDD? (Illustrate this.) Answer: BDD chapter.

14. If a formula is not valid, then it is satisfiable/falsifiable (choose one). If a formula is not satisfiable, then it its negation is valid/satisfiable/falsifiable (choose the correct and most descriptive answer). Answer: Valid is "tautology". Satisfiable is "not a contradiction."

15. Argue the **iff** part of the mapping reductions, for mapping

    (a) $3SAT$ (old problem) to *Clique* (new problem)

    (b) If the problem is SAT, there is a compatible "tour" across each island, pairwise. If not SAT, in one case, we should fail to find that "island to island bridge." Argue this through examples.

16. Work out the 3SAT to Clique mapping (the slides had examples).

4 One more take on the Reg_{TM} proof mapping reduction

Let us assume that we are working in C (or pseudo-C syntax). Let us introduce some types.

- `string s`: means `s` is of type string.
- `int i`: means `i` is of type int.
- `TM M`: means `M` is of type Turing Machine.
- C allows format strings such as `%s` and `%c`. I'll use `%s1` and `%s2` if I feed in multiple strings.
- `CProg C`: means `C` is of type `CProg` ("C program")
- We assume that anything of type `TM` is also a `C` program.
- Note that in C, when we say `int`, we mean "the set of integers."
- Thus, `int i` means `i` is a member of the set `int`.
- Likewise, `ATM <M, w>`. This means that `ATM` is a type, or set, and `<M,w>` are members of this set `ATM`.
- Thus when you see A_{TM}, imagine the type or set `ATM`. Its members are pairs `<M,w>`.
- You also know by now that the members of `ATM` are pairs of the form `<CProg, string>`. That is, `<M,w>` is nothing but something that has type `<CProg, string>`.

Let us introduce these C functions that are allegedly in some C library:

- `function SemiDeciderATM()`: This is a semi-decider for the set (or type) `ATM`. It is not a full decider but only a semidecider.
- `function FullDeciderATM()`: This is a full decider for the set (or type) `ATM`. **We have mathematically shown** that `function FullDeciderATM()` cannot exist.
- `function FullDeciderRegTM()`: This is a full decider for the set (or type) `REGTM`. This is the set of TMs whose languages are a regular set. **We are going to show that** if `function FullDeciderRegTM()` exists, then `function FullDeciderATM()` will end up existing. This will result in a contradiction.

Now define a translator from TMs to TMs (or CProg to CProg) called function `Translate`. This is our mapping reduction function! Function `Translate` must take `<M,w>` and produce another `CProg`.

```
CProg Translate(CProg M, string w) { // This is the mapping reduction function
 ... body of Translate ...
}
```

Our goal is to design `CProg Translate(...)` in such a way that it tries to trick `FullDeciderRegTM()` if it were to exist. For that, the design of `CProg Translate` must hide within it the power to define `FullDeciderATM()`.

We design things so that `Translate()` prints out `M'` by splicing-in `M` and `w`.

```
CProg Translate(CProg M, string w) { // This is the mapping reduction function
 string codebuf; // Declare a buffer to collect the code to be generated
 sprintf
   (codebuf, // Inject the code specified below into codebuf
    "
    M'(x) {
      if x is of the form 0^n 1^n then goto accept_M';

      Run %s1 on %s2 ; // First %s1 will splice-in M, second %s2 will splice-in w
```

```
        If this execution results in %s1 accepting %s2,
        then M' goes to accept_M';

        If %s1 rejects %s2, then M' goes to reject_M';

      }", M, w);
    return codebuf; // return the code of the M' machine
 }
```

4.1 How does Translate() help?

Notice that `Translate()` is our mapping reduction thingie. It takes an `M` and `w` and splices it into a print statement, where the **entire** print statement's body is the `M'` function!

Now if the output of `Translate()` – which is the `M'` machine – is presented to `FullDeciderRegTM()` – if it were to exist – then we have essentially created `FullDeciderATM()`. This is a contradiction.

Here is how `FullDeciderATM()` will work then (can be written in Pseudo-C).

```
Bool FullDeciderATM(CProg M, string w) { // All deciders output a boolean decision

 return FullDeciderRegTM( Translate(M,w) ) ;

}
```

4.2 Why does this work?

The only way the language of the `M'` machine will be deemed regular is if `M` accepts `w`. Else its language can't be regular. Thus this "secret" is revealed by judging that `M'` has a regular language.

5 A program specialization method for understanding mapping reductions

The idea of program specialization is well understood. Basically, if a program has a conditional statement `if(P) code1 else code2`, then if we know that `P` is true, we can replace the entire `if(P) code1 else code2` statement with `code1`. Likewise, if `P` is false, we can replace it with `code2`. We will employ program specialization to prove that the language L_{101} is not recursive.

$$L_{101} = \{\langle M\rangle \ : \ M \text{ is a TM whose language contains string 101}\}$$

Use any method starting from language A_{TM}. Write a clear proof, showing the mapping reduction that yields machine $M^{'}$ from M and w.[1] Argue that machine M accepts string w if and only if $M^{'}$'s language contains the string 101.

1. Describe $M^{'}$ clearly in pseudo-code form. It wires in a conditional test and then something to do with "101."

```
// This machine is emitted by mapping-reduction from <M,w>
// This machine has a state M'_accept and M'_reject
// This machine splices in M and w that are from the A_TM domain
M'(x) {

 Run M on w; // Language of M' is emptyset if M does not accept w,
             // which is what happens if there is an infinite loop here
 if (M accepts w) { // We find the "run M on w" halting in M_accept
```

[1]As a memory aid, during the last lecture, some of you were comfortable calling M by the name M_{36}, w by the name w_{63}, and $M^{'}$ by the name M_{75}.

```
   if (x==101)      // Lang(M') == {101} if M accepts w
   {  M' jumps to M'_accept;  }
   else
   { M' jumps to M'_reject; }
 }

 // We find the "run M on w" halting in M_reject
 M' jumps to M'_reject; // Lang(M')==emptyset if M does not accept w

}
```

There are many many other variations that also work. Here is one:

```
M'(x) {
 Run M on w; // Language of M' is emptyset if M does not accept w,
             // which is what happens if there is an infinite loop here
 if (M accepts w)
   {  M' jumps to M'_accept;  } // Language of M' is Sigma* if M accepts w
 Loop; // Language of M' is emptyset if M does not accept w
}
```

This is the desired mapping reduction because $L(M^{'})$ includes the string 101 if and only if $\langle M, w\rangle$ in A_{TM}.

2. Argue that M accepts w implies that the language of $M^{'}$ contains (or does not contain, as the case may be) string 101.

   We will argue that M accepts w implies $M^{'}$'s language contains 101, and M does not accept w implies $M^{'}$'s language does not contain 101.

   Let us go with the first pseudo-code. Now, if M accepts w, that condition allows us to specialize the code of `M'` to the following code of `M'` whose language does not contain $\{101\}$.

```
M'(x) {
   if (x==101)  {  M' jumps to M'_accept; }
   else         {  M' jumps to M'_reject;  }
}
```

   The language of $M^{'}$ is now $\{101\}$ which contains 101.

3. Argue that M does not accept w implies that the language of $M^{'}$ does not contain (or contains, as the case may be) string 101.

   If M does not accept w, that condition allows us to specialize the code of `M'` to the following code of `M'` whose language is $\{\}$.

```
M'(x) {
 Run M on w;              // Either loop here...
 M' jumps to M'_reject;  // ... or come out with M rejecting w
}
```

   The language of $M^{'}$ is now $\{\}$ which does not contain 101.

   Specializations applied to the second variation which is

```
M'(x) {
 Run M on w; // Language of M' is emptyset if M does not accept w,
             // which is what happens if there is an infinite loop here
 if (M accepts w)
   {  M' jumps to M'_accept;  } // Language of M' is Sigma* if M accepts w
 Loop; // Language of M' is emptyset if M does not accept w
}
```

These are the specializations:

- Case M accepts w: The specialization is:

  ```
  M'(x) {
   Run M on w; // This run WILL halt in accept
   M' jumps to M'_accept;
  }
  ```

  The language of $M^{'}$ is now Σ^* which contains 101.

- Case M does not accept w: The specialization is:

  ```
  M'(x) {
   Run M on w; // This may loop
   Loop;       // Or the loop may be here
  }
  ```

  The language of $M^{'}$ is now $\emptyset$ which does not contain 101.

5.1 Program specialization applied to Reg_{TM}

Let us see what `M'` below specializes to:

```
M'(x) {
  if x is of the form 0^n 1^n then goto accept_M';

  Run M on w ;

  If this execution results in M accepting w,
  then M' goes to accept_M';

  If M rejects w, then M' goes to reject_M';
 }
```

These are the specializations:

- Case M accepts w: The specialization is:

  ```
  M'(x) {
    if x is of the form 0^n 1^n then goto accept_M';
    Run M on w ;     // This will halt in accept
    go to accept_M';
  }
  ```

  The language of $M^{'}$ is now Σ^* because whether `x` is of the form 0^n1^n or not, `M'` will accept. This is a regular set.

- Case M does not accept w: The specialization is:

```
M'(x) {
  if x is of the form 0^n 1^n then goto accept_M';
  Run M on w ;  // This may loop
  Go to reject_M';
}
```

The language of $M^{'}$ is now strings of the form 0^n1^n which now constitutes a CFL that is not regular.

Thus by deciding regularity of a TM's language just by analyzing its code, we will have the capability to solve M's acceptance of w. This results in a contradiction.

5.2 How to view mapping reductions: Checklist

Some of you are missing a critical step or two in following mapping reduction proofs. I offer a checklist for you to go thru, taking $Regular_{TM}$ as an example. Stop and ask at the first step you do not follow. You may of course be told to go back and review things and try again.

1. Mapping reductions are established for various purposes. One purpose is to help show the undecidability of a new problem based on a previous undecidability result. You can check this box when we are done with this checklist. ☐

2. A_{TM} is a language that is recursively enumerable (RE) but not recursive (or decidable). ☐

3. Thus we will demonstrate a mapping reduction **from** A_{TM} to $Regular_{TM}$. *That will be the point at which you can utter a QED.* The existence of a mapping reduction from A_{TM} to $Regular_{TM}$ means that **if** you now have a decider for $Regular_{TM}$, then there will exist a decider for A_{TM}. *The point of a mapping reduction is to avoid saying this time and again!* Just exhibit a mapping reduction, put down your chalk, smile, and say QED. You can check this box when we are done with this checklist. ☐

4. A mapping reduction maps things in its domain to things in its codomain using a computable function. The mapping function is written as "f." It is also written $\leq_M$. Thus for our mapping reduction, we can write it as
$$A_{TM} \leq_M Regular_{TM}.$$
We use the less-than-or-equal-to symbol as it is perfectly appropriate. It means "less harder than or the same hardness." Thus, if we show that A_{TM} is less harder than or at the same hardness level as $Regular_{TM}$, we are done. It can't then be true that we can decide $Regular_{TM}$ but not A_{TM}. ☐

5. The mapping reduction function f takes an $\langle M, w\rangle$ pair from A_{TM}. Call it $\langle M_{36}, w_{63}\rangle$ for a specific example. It prints out a new function M'. Call it M_{75} as a specific example. ☐

6. The design of M_{75} is purposeful. We must ensure that its language is regular exactly when M_{36} accepts w_{63}. There may be 1000 ways to do it. Each way looks arbitrary, but fulfils the purpose. We basically have to create a "yes/no" situation – either regular or not. The "or not regular" part is achieved by including strings that **do not look regular**. This is why most popular constructions involve the pattern 0^n1^n. ☐

7. We claim that the language of M' below is regular exactly when M accepts w. To make things clear, I'm going to use the specific example in the description of `M_75` below. ☐

8. Now `M_75` is a TM, shown using pseudo-code. It is never intended to be run, but merely presented to a claimed decider of Reg_{TM} – say $DRegTM()$. ☐

9. If $DRegTM(M_{75})$ comes out with the verdict "M_{75} has a regular language," then $DRegTM$ has "divined that" M_{36} accepts w_{63}. Else it has divined that M_{36} does not accept w_{63}. Hence contradiction. ☐

10. Now do the Asg-5 problem and also the "TM's language has 101" problem. ☐

```
M_75(x) {
  if x is of the form 0^n 1^n then goto accept_M_75 ;
  Run M_36 on w_63 ;
  If this execution results in M_36 going to one of its
  accept states and getting stuck there
  (i.e., M_36 accepts w_36, then M_75 goes to accept_75
   (let's say M_75 has only one accept state called F_accept_75);
  If M_36 rejects w_63, then M_75 goes to reject_M_75
  (let us say its only reject state);
}
```