Do any **eight** of the **ten** problems below. If you attempt more than eight problems, please indicate which ones to grade (otherwise we will make a random choice). This allows you to attempt 104 points. The grading will be on a scale of 100. Thus, there are 4 points of extra credit built into the exam.

1. **(13 points):** Powers of Three.
   Describe a TM that accepts the set $\{a^n \mid n \text{ is a power of } 3\}$. You should sketch how the machine works, but you don't need to give a detailed description of its transition function.

   Solution: One can construct a machine that makes repeated sweeps across the tape. On the first sweep, it looks for the first blank and replaces it with a right endmarker (e.g. $\dashv$). If on any sweep, there is exactly one a, then the machine accepts (1 is a power of three). If on any sweep, there are no a's then the machine rejects. Otherwise, the machine replaces the first two of every three a symbols that it reads with blanks. In so doing, it keeps track of the number of a's mod three. If the number of a's on this pass is not divisible by three, the machine rejects. Otherwise, it has divided the number of a's by three. The machine continues making these passes until it accepts or rejects.

2. **(13 points):** Primes.
   Is the set
   $$B \quad = \quad \{M \mid M \text{ accepts iff the length of its input is a prime number}\}$$
   recursive, recursively enumerable, neither or both? Give a short justification for your answer.

   Solution: B is not r.e.; therefore, it is not recursive.
   Clearly, there is a TM that tests primality – Kozen described such a machine based on the sieve of Eratosthenes. Let M_{sieve} be such a machine. Let M_{Σ^*} be a machine that accepts Σ^*, for any non-empty alphabet Σ. Clearly $M_{\Sigma^*} \notin B$. Finally, we note that $L(M_{sieve}) \subset L(M_{\Sigma^*})$. Thus, the second form of Rice's theorem applies to show that B is not r.e.

3. **(13 points):** Leftist Turing Machines.
   Is the set
   $$B \quad = \quad \{M\#x \mid M \text{ moves its head to the left at most 10 times when run with input } x\}$$
   recursive, recursively enumerable, neither or both? Give a short justification for your answer.

   Solution: B is recursive, and therefore recursively enumerable.
   I'll show that we can simulate M with a DFA. From any configuration, M can move to the left by at most 10 tape-squares before halting, exhausting its limit of 10 moves to the left, or moving to the square to the right of the current one. My DFA will store in its finite state the state of M, the contents of the ten tape squares to the left of the current position, and the number of left moves that M has made. My DFA has $N = 11 * |Q| * |\Gamma|^1 0 + 1$ states to do this, where Q and Γ are the states and tape alphabet respectively of M. At each step, the DFA considers the current input symbol, and based on the next up-to 21 moves of M, determines its next state. This state includes a count of the total number of left-moves that M has made. The states for which this count is less than or equal to 10 are accepting states. If the DFA determines that M has made more that ten moves to the left, it moves to a permanently rejecting state. When the DFA has read $|x|$ input symbols, the rest are all blanks. The DFA must loop within N steps. At this point, if the DFA is in an accepting state, then M will never move make more than 10 moves to the left, and $M\#x \in B$. Otherwise, if the DFA is in a rejecting state; M has made more than 10 moves to the left; and $M\#x \notin B$.
   Note: My construction obscenely overestimates the number of steps needed to determining whether or not M will move to the left more than 10 times. But the point of this class is not to do precise combinatorics. We only need to determine whether or not the property is decidable; efficiency is not a concern.

4. **(13 points):** Outer Thirds.
Let A be a language. We define

$$OuterThirds(A) \quad = \quad \{w \mid \exists x, y, z.(|x| = |y| = |z|) \wedge (w = xz) \wedge xyz \in A\}$$

Show that there is a language A such that A is regular but $OuterThirds(A)$ is not regular. Give a proof with your answer.

Solution: Let $A = (ab)^*a$, and let $B = OuterThirds(A)$. If B were regular, then it would have a pumping lemma constant, k. Let $w = (ab)^{3k+1}a$, $x = z = (ab)^k a$, and $y = (ba)^k b$. Clearly $w \in A$, $w = xyz$, and $|x| = |y| = |z| = 2k + 1$. Thus, $xz = (ab)^k aa(ba)^k$ is in B. We force the demon to pump in the initial first k symbols of x. No matter what substring the demon chooses, we can pump xz such that there are two consecutive a's in the second half of the resulting string. That string cannot be in B because all strings in A have alternating a's and b's. This shows that the B does not satisfy the pumping lemma for regular languages. Thus, B is not regular which shows that $OuterThirds$ of a regular language is not guaranteed to be regular.

5. **(13 points):** Context-Free, or Not?
Consider the two languages below:

$$\begin{array}{rcl} B & = & \{w \in \{a, b\}^* \mid \exists i.(w = a^i b^{2i}) \vee (w = a^{2i} b^i)\} \\ \widetilde{B} & = & \sim B \end{array}$$

(a) **(7 points):** Is B context-free? If it is, give a CFG that generates it and a brief explanation of how your CFG works; otherwise, prove that B is not context-free.

Solution: B is context-free.
Here's a grammar:

$$\begin{array}{rcl} S & \rightarrow & X \mid Y \\ X & \rightarrow & \epsilon \mid a\,X\,bb \\ Y & \rightarrow & \epsilon \mid aa\,Y\,b \end{array}$$

where S is the start symbol. The non-terminal X generates all strings of the form $a^i b^{2i}$. Likewise, the non-terminal Y generates all strings of the form $a^{2i} b^i$.

(b) **(6 points):** Is $\widetilde{B}$ context-free? If it is, give a CFG that generates it and a brief explanation of how your CFG works; otherwise, prove that $\widetilde{B}$ is not context-free.

Solution: $\widetilde{B}$ is context-free.
Here's a grammar:

$$\begin{array}{rcl} S & \rightarrow & V \mid W \mid X\,B \mid Y \mid A\,Z \mid \\ W & \rightarrow & ab \mid a\,W\,b \mid a\,W\,bb \\ X & \rightarrow & \epsilon \mid a\,X\,bb \\ Y & \rightarrow & ab \mid a\,Y\,b \mid aa\,Y\,b \\ Z & \rightarrow & \epsilon \mid aa\,Z\,b \\ A & \rightarrow & a \mid A\,a \\ B & \rightarrow & b \mid B\,b \\ V & \rightarrow & C\,ba\,C \\ C & \rightarrow & \epsilon \mid a\,C \mid b\,C \end{array}$$

where S is the start symbol. All strings in $\widetilde{B}$ are of the form $a^i b^j$ with one of the four conditions holding:

1. $1 < i \leq j < 2i$
2. $2i < j$
3. $1 < j \leq i < 2j$
4. $2j < i$

Any string generated by W has the form $a^{m+n+1}b^{m+2n+1}$ where m is the number of $W \rightarrow a\,W\,b$ productions in the derivation, and n is the number of $W \rightarrow a\,W\,bb$ productions. Let $i = m + n + 1$ and $j = m + 2n + 1$. Because m and n are non-negative, we have $1 < i \leq j < 2i$, which shows that the string generated by W is in $\widetilde{B}$. Likewise, for any i and j with $1 < i \leq j < 2i$, we can derive $a^i b^j$ from W by using $j - i$ applications of $W \rightarrow a\,W\,bb$, i applications of $W \rightarrow a\,W\,b$, and one application of $W \rightarrow ab$.
Any string generated by $X\,B$ has the form $a^m b^{2m+n+1}$ where m is the number of $X \rightarrow a\,W\,bb$ productions in the derivation, and n is the number of $B \rightarrow B\,b$ productions. By arguments similar to those above, we can show that $X\,B$ generates exactly those strings of the form $a^i b^j$ with $2i < j$. Finally, arguments similar to those above show that Y generates exactly those strings corresponding to $1 < j \leq i < 2j$, and Z generates the strings for $2j < i$. The productions for V and C generate any string that is not of the form a^*b^*.

6. **(13 points):** No Exceptions.
Let x be an array in a Java program. If x has n elements, then the valid values for indices for x are $0 \ldots n-1$. A Java program throws an ArrayIndexOutOfBounds exception if the program attempts to access an array with an index value that is outside of this range.

For simplicity, we'll assume that the Java programs that we want to analyse have no command line arguments and read all of their input from stdin.

Let $A = \{src\#input \mid Except(src, input)\}$ where $Except(src, input)$ is true iff the Java program with source-code *src* throws an ArrayIndexOutOfBounds exception when run with input *input*.

Is the set A recursive, recursively-enumerable, neither, or both? Give a short justification for your answer.

Solution: A is r.e. but not recursive. Proof that A is not recursive: I'll reduce the halting problem to testing whether or not a Java program throws an ArrayIndexOutOfBounds exception. We can write a Java method with parameters M and x that simulates Turing machine M running on input x. Let's call this method

boolean turingSim(TuringMachine M, InputString x)

This method returns true if M accepts x, false if M rejects x, and runs forever if M loops on x. Furthermore, we can write turingSim so that it is guaranteed never to throw an ArrayIndexOutOfBounds exception – for example, we could write it without using any arrays. Now, we write a program, P, that given input $M\#x$ first calls turingSim with M and x. After turingSim returns, this program commits an array bounds violation. Thus, P throws an ArrayIndexOutOfBounds exception iff M halts when run with input x. This reduces the halting problem to testing for array bounds violations and shows the latter to be undecidable.

Proof that A is r.e. We can make a TM, E, that takes as input $J\#x$ where J is the source code for a Java program and x is an input string for this program. TM E simulates the execution of program J with input x. If J ever throws an ArrayIndexOutOfBounds exception, E accepts. If J terminates without throwing such an exception, E rejects. If J runs forever, so does E. By construction, $L(E) = A$. The set A is accepted by a TM and is therefore r.e.

7. **(13 points):** Context-Free Shuffling.
Recall the shuffle operation. If $x, y \in \Sigma^*$, we write $x \| y$ for the set of all strings that can be obtained by shuffling strings x and y together like a deck of cards. Formally,

$$\begin{aligned} \epsilon \| y &\stackrel{\text{def}}{=} \{y\} \\ x \| \epsilon &\stackrel{\text{def}}{=} \{x\} \\ xa \| yb &\stackrel{\text{def}}{=} ((x \| yb) \cdot a) \cup ((xa \| y) \cdot b) \end{aligned}$$

The shuffle of two languages, A and B, denoted $A \| B$, is the set of all strings obtained by shuffling a string from A with a string from B:

$$A \| B \stackrel{\text{def}}{=} \bigcup_{\substack{x \in A \\ y \in B}} x \| y$$

Are the context-free languages closed under the shuffle operation? In other words, if A and B are context-free, is $A \| B$ guaranteed to be context-free? Justify your answer.

Solution: The CFLs are not closed under shuffling.
Let $A_1 = \{a^n b^n\}$, $A_2 = \{c^n d^n\}$, and $A_{1\|2} = A_1 \| A_2$. If $A_{1\|2}$ were context-free, then it would have a pumping lemma constant. Let k be this constant. Let $s = a^k c^k b^k d^k$. If $A_{1\|2}$ were context-free, then we could find u, v, w, x, and y such that $x = uvxwy$, $|vxw| \leq k$, $|vw| > 1$, and $uv^i wx^i y \in A_{1\|2}$ for all $i \geq 0$. If vx contains an a, then neither can contain a b or $|vwx|$ would be greater than k. Then, pumping s would change the number of a's without changing the number of b's and produce a string that is not in $A_{1\|2}$. Similar argument hold if vx contains any of the other input symbols. Thus, vx must be empty, which violates the conditions of the pumping lemma. The language $A_{1\|2}$ does not satisfy the pumping lemma for CFLs; thus $A_{1\|2}$ is not context-free.

8. **(13 points):** Integer Constants in Java.
An integer constant in Java can be an octal constant, a decimal constant, or a hexadecimal constant. An octal constant starts with the character '0' and is followed by zero or more octal digits. A decimal constant starts with the any of the characters, '1', '2', '3', '4', '5', '6', '7', '8', or '9', and is followed by zero or more decimal digits. A hexadecimal constant starts with the string "0x" and is followed by one or more hexadecimal digits. Any of these forms may have the character 'l' (a lower case L) at the end to indicate that it is a long (i.e. 64 bit) constant. An octal digit is any element of the set

$$Oct = \{\text{'0'}, \text{'1'}, \text{'2'}, \text{'3'}, \text{'4'}, \text{'5'}, \text{'6'}, \text{'7'}\}$$

A decimal digit is any element of the set

$$Dec = Oct \cup \{\text{'8'}, \text{'9'}\}$$

A hexadecimal digit is any element of the set

$$Hex = Dec \cup \{\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \text{'e'}, \text{'f'}\}$$

Write a regular expression that matches Java integer constants as defined above (and matches no other strings). You may build up your regular expression from smaller expressions, and you may use Oct, Dec, and Hex within your expressions.

Solution: Let the regular expression $D_1 = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ denote the allowed starting digits for a decimal constant. My regular expression for Java integer constants is:

$$((0 \cdot Oct^*) + (D_1 \cdot Dec^*) + (0 \cdot x \cdot Hex \cdot Hex^*))(l + \epsilon)$$

9. **(13 points):** Parsing Poker.
The POKER programming language has four arithmetic operators: $\clubsuit$, $\diamondsuit$, $\heartsuit$, and $\diamondsuit$ each defined on the non-negative integers as shown below:

$$\begin{array}{lcll} x \clubsuit y & = & x^2 + y^2, & 12 \clubsuit 8 = 208 \\ x \diamondsuit y & = & \mathsf{gcd}(x, y), & 12 \diamondsuit 8 = 4 \\ x \heartsuit y & = & \mathsf{lcm}(x, y), & 12 \heartsuit 8 = 24 \\ x \spadesuit y & = & x^y, & 12 \spadesuit 8 = 429,981,696 \end{array}$$

where gcd and lcm denote the greatest-common-divisor and least-common-multiple respectively. The $\clubsuit$ and $\heartsuit$ operators can also be used as unary operators. The expression $\clubsuit x$ denotes the sum of the positive factors of x (other than x itself) – for example, $\clubsuit 12 = 1 + 2 + 3 + 4 + 6 = 16$, and $\clubsuit 137,438,691,328 = 137,438,691,328$. The expression $x \heartsuit$ denotes the largest prime factor of x – for example, $12 \heartsuit = 3$.

Here's a CFG for POKER:

$$\begin{array}{rcl} \mathit{expr} & \rightarrow & \alpha \mid \clubsuit\ \mathit{expr} \\ \alpha & \rightarrow & \beta \mid \alpha \clubsuit \beta \\ \beta & \rightarrow & \gamma \mid \beta \diamondsuit \gamma \\ \gamma & \rightarrow & \eta \mid \gamma \heartsuit \eta \\ \eta & \rightarrow & \theta \mid \eta \heartsuit \\ \theta & \rightarrow & \mu \mid \theta \spadesuit \mu \\ \mu & \rightarrow & \mathsf{CONST} \mid \mathsf{LPAREN}\ \mathit{expr}\ \mathsf{RPAREN} \end{array}$$

where the terminal symbols are $\clubsuit$, $\diamondsuit$, $\heartsuit$, $\spadesuit$, CONST (an integer constant, written in decimal notation), LPAREN (a left parenthesis), and RPAREN (a right parenthesis).

(a) **(7 points):** Is "$3 \clubsuit 52 \diamondsuit 42 \heartsuit\ \heartsuit 8$" generated by the CFG for POKER expressions? If so, draw the parse-tree for the expression and determine the value of the expression; otherwise, give a brief explanation (one or two sentences) of why it is not possible to generate this expression.

Solution: This string is generated by the grammar.

Figure 1 shows the parse tree. I've written each CONST terminal by giving the string for the integer

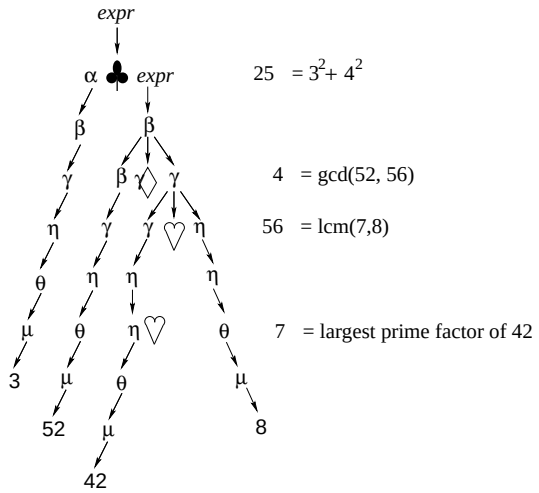

Figure 1: Parse tree for "$3 \clubsuit 52 \diamondsuit 42 \heartsuit\ \heartsuit 8$"

constant. At the same level as each production that involves one of the operators, I've written the value produced by that operator for this expression. The value for the expression is 25.

(b) **(6 points):** Is "$3\clubsuit\ \clubsuit\ 52\diamondsuit 42\heartsuit 8$" generated by the CFG for POKER expressions? If so, draw the parse-tree for the expression and determine the value of the expression; otherwise, give a brief explanation (one or two sentences) of why it is not possible to generate this expression.

Solution: This string is not generated by the grammar.

The "problem" is the two consecutive $\clubsuit$ operators. The only derivation that produces the prefix $3\clubsuit$ is

$$\begin{aligned} expr &\rightarrow \alpha \\ &\rightarrow \alpha \clubsuit \beta \\ &\rightarrow \beta \clubsuit \beta \\ &\rightarrow \gamma \clubsuit \beta \\ &\rightarrow \eta \clubsuit \beta \\ &\rightarrow \theta \clubsuit \beta \\ &\rightarrow \mu \clubsuit \beta \\ &\rightarrow \mathsf{CONST} \clubsuit \beta \end{aligned}$$

However, all derivations of strings from β have a CONST as their leftmost terminal, not a $\clubsuit$.

10. **(13 points):** Functions that Grow Really Fast.
Consider the following function:

f:

$$\begin{aligned} f(n,0) &= n! \\ f(n,m) &= f(f(f(f\ldots f(n,m-1)\ldots),m-1),m-1), \quad n \text{ nested applications of } f(\cdot,m-1). \end{aligned}$$

If you prefer, here's an equivalent definition of f:

```
integer f(integer n, integer m) {
  if(m == 0) return(n!);
  else {
    integer q = n;
    for(int i = 0; i < n; i++) q = f(q, m-1);
    return(q);
  }
}
```

For example,

$$\begin{aligned} f(3,0) &= 3! &&= 6 \\ f(3,1) &= f(f(f(3,0),0),0) &&= f(f(3!,0),0) \\ &= &&= f(f(6,0),0) \\ &= &&= f(6!,0) \\ &= &&= f(720,0) \\ &= &&= 720! \\ &= &&\approx 2.60121894 * 10^{1746} \end{aligned}$$

To calculate $f(3,2)$, we start with $720!$ – let $X = 720!$. We then calculate $X!!!\ldots!$, with $720!$ (roughly $2.6 * 10^{1746}$) factorial operations total. Let that number be Y. We now calculate $Y!!!\ldots!$ with Y factorial operations. If you think this number is big, note that it's only $f(3,2)$. The number for $f(3,3)$ is way, way bigger. In fact, if we tried to write it as $10^{10^{10^{\cdot^{\cdot^{\cdot}}}}}$, we would need more exponents in the stack than the number of atoms in the universe.

g: Let $g(n) = f(n,n)$. For example,

$$\begin{aligned} g(0) &= 1 \\ g(1) &= 1 \\ g(2) &= 2 \\ g(3) &= f(3,3), \quad \text{(really, } \textbf{REALLY} \text{ big, see comments above)} \end{aligned}$$

h: Let

$$HP_n \quad = \quad \{M\#x \in HP \mid |M\#x| \leq n\}$$

where $M\#x \in HP$ iff Turing machine M halts when run on input x. Define

$$\begin{array}{rcll} h(n) & = & \max_{M\#x \in HP_n} nsteps(M,x), & \text{if } HP_n \neq \varnothing \\ & = & -1, & \text{otherwise} \end{array}$$

where $nsteps(M,x)$ is the number of steps the M takes before halting when run with input x.

We say that function F_1 *dominates* function F_2 iff there exists some integer m such that for all $n \geq m$, $F_1(n) > F_2(n)$.

(a) **(7 points):** Does g dominate h?

Solution: g does not dominate h.

Function g is computable by a Turing machine. If g dominated h, we could solve the halting problem. Assume that g dominates h and let m be an integer such that for all $n \geq m$, $g(n) > h(n)$. Because g is monotonic, $g(\max(n,m)) > h(n)$ for all n. Given $M\#g$, we could simulate M for up $g(\max(|M\#x|,m))$ steps. If M halts within this number of steps, we accept; otherwise reject. This would be correct because no TM M that halts on input x takes more then $h(|M\#x|)$ steps to halt, and $g(\max(|M\#x|,m) > h(|M\#x|)$. We know that the halting problem is undecidable; therefore, it must not be the case that g dominates h.

(b) **(6 points):** Does h dominate g?

Solution: h does dominate g.

We can build a TM, M, that with input x, computes the sum of $|x|$ and the length of the description of M. Note that it is easy to embed the length of M into M. Using binary, we only need $\log(|M|)$ extra states to encode $|M|$. Let s denote this sum. M now calculates $g(s)$ and then repeatedly subtracts 1 from the result until it reaches 0. Then, M halts. M performs $g(|M\#x|)$ operations in this decrement, plus some more to calculate $g(s)$ in the first place. Thus, M halts on input x after more than $g(|M\#x|)$ steps. By definition, $h(|M\#x|)$ must be at least as large as the number of steps that M takes to do this calculation. Thus, $h(n) > g(n)$ for any $n > |M|$ for the machine M that we have just described. Note: the function f that I described here is a simple variation of Ackermann's function. As you can see, Ackermann's function grows very quickly.