Problem 1

```
  nfa<-regex :: Regex -> NFA // from class
  dfa<-nfa   :: NFA -> DFA   // from class
  complement :: DFA -> DFA   // flips the final states
  intersect  :: DFA -> DFA -> DFA // from class, accepts iff both DFAs accept

def algo(R :: Regex, S :: Regex)
  dfa_R = dfa<-nfa (nfa<-regex R)
  dfa_S = dfa<-nfa (nfa<-regex S)
  dfa_ans = dfa_R intersect (complement dfa_S)
  return dfa_ans
```

For correctness, note that $L(dfa_{ans}) = \emptyset \Leftrightarrow L(R) \cap \overline{L(S)} = \emptyset \Leftrightarrow L(R) \subseteq L(S)$.
For termination, we note that (1) our function does not have loops / recursions and (2) all functions our function calls terminate.

Problem 2

(a)

```
Let RA be a recognizer for A.
We will contruct a recognizer R_CATM for (complement A_TM) as follows:

def R_CATM (M, x):
  def m1(z):
    if (M(x) == accept)
      then accept
      else reject
  def m2(z):
    reject
  return RA(m1, m2)


Proof (not required for full credit)

case (M,x) is in (complement A_TM):
  (M,x) not in A_TM
  M(x) does not accept
  L(m1) = emptyset
  L(m2) = emptyset
  (m1, m2) is in A
  RA(m1, m2) halts + accepts
  Yay!

case (M,x) is NOT in (complement A_TM):
  (M,x) is in A_TM
  M(x) accepts
  L(m1) = all strings
  L(m2) = emptyset
  (m1, m2) is NOT in A
  RA(m1, m2) does not accept
  Yay!
```

(b)
```
Let RCA be a recognizer for (complement A).
We will contruct a recognizer R_CATM for (complement A_TM) as follows:

def R_CATM (M, x):
  def m1(z):
    accept
  def m2(z):
    if (M(x) == accept)
      then accept
      else reject
  return RCA(m1, m2)


Proof (not required for full credit)

case (M,x) is in (complement A_TM):
  (M,x) not in A_TM
  M(x) does not accept
  L(m1) = all strings
  L(m2) = emptyset
  (m1, m2) is NOT in A
  RCA(m1, m2) halts + accepts
  Yay!

case (M,x) is NOT in (complement A_TM):
  (M,x) is in A_TM
  M(x) accepts
  L(m1) = all strings
  L(m2) = all strings
  (m1, m2) is in A
  RCA(m1, m2) does not accept
  Yay!
```

Problem 3

(a)
```
worker(M, s):
  run M(s) for |s|^2 steps
  if halted, reject
  if still running, accept

R(M):
  for s in lexicographical order
    fork worker(M, s);
    if (any existing worker accepts), accept;
```

(b)
```
Let RL be a recognizer for L.
We will now construct a recognizer RC_HTM for (complement H_TM):

def RC_HTM(M, x):
```

```
  def m1(z):
    run M(x) for |z|^2 steps
    if M(x) is still running, halt
    if M(x) halted, inf loop
  return RL(m1)

Proof (not required for full credit):

If (M,x) is in (complement H_TM):
  M(x) does not halt
  m1(z), for all z, halts after |z|^2 + 2 steps
  m1 in L ==> RL(m1) halts + accepts

If (M,x) is NOT in (complement H_TM):
  M(x) halts after k steps
  consider some z s.t. |z|^2 > k
  m1(z):
    sims M(x) for |z|^2 > k steps
    M(x) halts ==> inf loops
  thus m1 not in L ==> RL(m1) does not accept
```

Grading Rubric

- There's 5 separate sections: P1 (30), P2a (15), P2b (15), P3a (10), P3b (30).
- For each section:
  - Decide if solution is "basically correct" or "way off" (incorrect reduction; reducing in wrong direction; etc ...)
  - "Way off" solutions = 0 points
  - "Basically correct solutions" = start from full credit, deduct points as necessary for minor technical mistakes
  - When taking off points, provide a short (1-2 sentence) explaination for why points are being deducted.
- For P3: we allow students the following variations (instead of $|x|^2$ time steps):
  - $|x|^2 + 100$
  - $9999 * |x|^2 + 9999$
  - $|x|^2$ requirement for all $|x| > k$