Introduction to Algorithms **Guide to HW 9**
CS 4820, Spring 2019

(1) *(10 points)* Let $\mathcal{L} \subseteq \{0,1\}^*$ be a language, and let $\overline{\mathcal{L}}$ be its complement, i.e. the set of all finite binary strings that don't belong to $\mathcal{L}$. Prove that $\mathcal{L}$ is decidable if and only if both $\mathcal{L}$ and $\overline{\mathcal{L}}$ are recognizable.

Solution. We start out by proving the easier direction. Suppose that $\mathcal{L}$ is decidable, and let M be an SJava program that decides it. Further let Q be the SJava program that outputs flips the output of M (i.e, Q outputs `true` when M outputs `false` and outputs `false` when M outputs `true`). See Figure 1 for a skeleton code of Q. Clearly by definition M decides $\mathcal{L}$ and Q decides $\overline{\mathcal{L}}$.

```
bool flip(string w) {
        bool a=main(w); #assume main is the first function of M
        return !a;
  }



#
# code for M goes here
#
```

Figure 1: Skeleton code for Q

Now, in the other direction suppose that both $\mathcal{L}$ and $\overline{\mathcal{L}}$ are recognizable. Let programs M and Q recognize $\mathcal{L}$ and $\overline{\mathcal{L}}$ respectively. We now construct a program P that on input w does the following:

```
for i=1,2,...{
        Use the Universal SJava program to simulate $M$ on $w$ for $i$ steps. If $M$ accepts $w$, then
    return true and halt. (Else do nothing)
        Use the Universal SJava program to simulate $Q$ on $w$ for $i$ steps. If $Q$ accepts $w$, then
    output false and halt. (Else do nothing)
}
```

See Figure 2 for a skeleton code implementing P. We claim that P decides $\mathcal{L}$. First suppose $w \in \mathcal{L}$. Since M recognizes $\mathcal{L}$, it follows that M halts on input w, and returns `true`. Let j denote the number of steps that M takes to accept w. We note that for any $i < j$, in the i'th iteration of the loop in P, we do not return any output. This follows since by assumption, M does not accept w in less than j steps. Further, Q never accepts w since $w \notin \overline{\mathcal{L}}$. Finally, in the j'th iteration of the loop we return `true` since M accepts w in j steps.

Next suppose $w \notin \mathcal{L}$. This implies that $w \in \overline{\mathcal{L}}$, and hence Q halts on input w, and returns `true`. The argument that P returns false in this case is similar to the argument used in the above paragraph.

```
bool runboth(string w) {
   codeM = # gigantic string constant containing the entire code for M
   codeQ = # gigantic string constant containing the entire code for Q

   execM = initialExecSate(codeM,w);
   execQ = initialExecState(codeQ,w);
   while (true) {
       execM = singleStep(codeM,w);
       execQ = singleStep(codeQ,w);
       done = testIfHalted(execM) || testIfHalted(execQ);
       if (done) {
         if (execM == "true") {
           return true;
         }
         if (execQ == "true") {
           return false;
         }
       }                                       // end if (done)
   }                                           // end while (true)
}

#
# code for interpreter goes here.
#
```

Figure 2: Skeleton code for P

(2) *(10 points)* As you probably know, buffer overflows are a common security flaw in computer programs that can be exploited by attackers to overwrite areas of memory that should have been protected from access. This exercise explores why it's so difficult to perform static analysis of programs to discover potential buffer overflows. In each of the sub-questions below, we use the term *buffer program* to refer to an SJAVA program whose base function contains a variable named "buffer" of string type. The *length of the buffer* refers to the number of characters in this string.

For each of the folloiwng decision problems, determine whether the problem is decidable, recognizable but not decidable, or not recognizable. Substantiate your answer with a proof. If one part of your solution uses material from an earlier part, it is fine to refer back to that earlier part of your solution rather than repeating the material.

(2a) *(5 points)* $\mathcal{L}_{\mathrm{A}}$ is the set of all ordered triples $\langle M, x, k\rangle$ such that M is a buffer program, and when M runs on input x the length of the buffer exceeds k at least once during its execution.

(2b) *(5 points)* $\mathcal{L}_{\mathrm{B}}$ is the set of all ordered pairs $\langle M, x\rangle$ such that M is a buffer program, and when M runs on input x the length of the buffer grows unboundedly large. (Meaning: M runs forever on input x, and for all k there is a time during its execution when the length of the buffer exceeds k.)

Solution for (2a): The problem $\mathcal{L}_{\mathrm{A}}$ is recognizable, but not decidable. A program that recognizes $\mathcal{L}_{\mathrm{A}}$ can be written by using a universal SJAVA program to simulate the execution of M on input x step by step, checking after each step of the execution whether the length of the buffer has exceeded k. Skeleton code for this algorithm is as follows; we omit the code for the universal SJAVA program and the code to implement a simple additional function `bufferLength` that takes an execution state and returns the length of the buffer.

```
boolean recognizeOverflow(string M, string x,  int  k) {
  execState = initialExecState(M, x);
  done = false;
  while  (!done) {
    if  (bufferLength(execState) > k) {
      return true;                              // buffer  overflowed
    }
    execState = singleStep(M, execState);
    done = testIfHalted(execState);
  }
  return  false ;                               // M halted without buffer overflow.
}

#
#  code for  bufferLength function goes here
#

#
# code for SJava interpreter  goes here
#
```

To prove that $\mathcal{L}_{\mathrm{A}}$ is not decidable, we show that $\mathcal{L}_{\mathrm{HALT}} \leq \mathcal{L}_{\mathrm{A}}$. This means we need to describe a reduction that takes an instance of the halting problem, $\langle M, x\rangle$, and uses a decision procedure for $\mathcal{L}_{\mathrm{A}}$ to decide whether M halts on input x. This is quite easy to do: given M and x, write a wrapper program M' that takes an input string y, runs M on input x, then when M halts (if that ever happens) it writes "1" to the buffer. Now ask the decision procedure for $\mathcal{L}_{\mathrm{A}}$ whether $\langle M', x, 0\rangle$ belongs to $\mathcal{L}_{\mathrm{A}}$. If the answer is yes, it means that when M' runs on input x, it writes a non-zero number of characters to the buffer. This can only happen if M halts on input x, hence $\langle M, x\rangle \in \mathcal{L}_{\mathrm{HALT}}$. Conversely, if $\langle M, x\rangle \in \mathcal{L}_{\mathrm{HALT}}$ then when M' runs on input x, it writes "1" to the buffer, which means the length of the buffer exceeds 0, hence $\langle M', x, 0\rangle$ belongs to $\mathcal{L}_{\mathrm{A}}$.

Below we present SJava code for a `haltChecker` program that implements the idea described in the preceding paragraph. We assume there is a hypothetical `decideOverflow` program that decides $\mathcal{L}_{\mathrm{A}}$. We also assume (without providing explicit code) that there are three string processing subroutines included in the `haltChecker`:

1. `substitute` takes three strings x, y, z and transforms x by replacing the first occurrence of substring y in x (if any) with string z.
2. `baseFuncName` takes an SJava program and returns the name of the base function.
3. `bigString` takes no inputs, and it outputs a string which is equal to the following piece of code.

   ```
   int  potentialBufferWriter( string  x)  {
     a = SUBST_BASE(x);
     buffer  =  ‘‘1’’;
   }

   SUBST_PROG
   ```

Now here is the skeleton code for `haltChecker`.

```
boolean haltChecker(string M, string  x)  {
  basefunc = baseFuncName(M);
  temp_string = substitute(bigString (),”SUBST_BASE”,baseFunc);
  Mprime = substitute(temp_string,”SUBST_PROG”,M);
  return  decideOverflow(Mprime,x,0);
```

```
}

#
# code for baseFuncName, substitute, and bigString() goes here
#

#
# code for decideOverflow goes here
#
```

Solution for (2b): Since the halting problem is recognizable but not decidable, we know from problem 1 on this assignment that $\overline{\mathcal{L}_{\text{HALT}}}$, the complement of the halting problem, is not recognizable. We will prove that $\overline{\mathcal{L}_{\text{HALT}}} \le \mathcal{L}_{\text{B}}$, thus establish that $\mathcal{L}_{\text{B}}$ is not recognizable. In other words, assuming existence of a program `recognizeUnbounded` that accepts $\langle M, x\rangle$ if and only if the buffer length grows unbounded when M runs on input x, we show how to write a program `loopChecker` that accepts $\langle M, x\rangle$ if and only if M runs forever on input x.

For we provide a plain-English description of the `loopChecker` decision procedure works. Given $\langle M, x\rangle$, we write a modified program M' that, on input x, simulates M running on x while appending an extra character to the buffer after every step in the simulated execution. Then we ask the `recognizeUnbounded` subroutine whether it accepts $\langle M', x\rangle$ and (if it ever terminates) we output its answer. This works because during the execution of M' on input x, the buffer grows unboundedly large if and only if the execution of M on input x runs forever, hence $\langle M', x\rangle$ belongs to $\mathcal{L}_{\text{B}}$ if and only if $\langle M, x\rangle$ belongs to $\overline{\mathcal{L}_{\text{HALT}}}$.

Now we provide skeleton code for `loopChecker` in SJava. As before, we assume (without providing explicit code) that there are certain string processing subroutines.

1. `substitute` is the same function we used in part (2a).
2. `univSJavaProg` is a function that takes no arguments and outputs a gigantic string constituting the universal SJava program.
3. `anotherBigString` is a function that takes no arguments and outputs a string which is equal to the following piece of code.

```
    string simulateWithAppend(string x) {
      program = "SUBST_PROG";
      execState = initialExecState(program, x);
      done = false;
      pos = 0;
      while (!done) {
          buffer[pos] = "1";
          pos = pos + 1;
          execState = singleStep(program, execState);
          done = testIfHalted(execState);
      }
    }

    SUBST_INTERP
```

```
boolean loopChecker(string M, string x) {
  usjp = univSJavaProg();
  temp_string = substitute(anotherBigString(),"SUBST_PROG",M);
  Mprime = substitute(temp_string,"SUBST_INTERP",usjp);
  return recognizeUnbounded(Mprime,x);
}

#
# code for substitute, univSJavaProg() and anotherBigString() goes here
#

#
# code for decideOverflow goes here
#
```

(3) *(10 points)* Recall that the execution state of an SJava program is defined to be a stack of function states. Further, recall that this is encoded as a string called `execState` by the universal SJava program `interpreter`. (See Section 5 of the Computability handout for more details.)

Define $\mathcal{L}_{\text{SPACE}}$ to be the set of all ordered triples $\langle M, x, C\rangle$ such that in the execution of `interpreter`(M,x), the string `execState` never exceeds length C. Either prove that there exists a SJava program that decides $\mathcal{L}_{\text{SPACE}}$, or prove that $\mathcal{L}_{\text{SPACE}}$ is undecidable.

Solution. We prove that $\mathcal{L}_{\text{SPACE}}$ is decidable. Let Σ denote the allowed set of values for the data type `char` in SJava, and let the cardinality of Σ be ℓ. Further let $T = (\ell+1)^C$. We will prove that the following simple program P decides $\mathcal{L}_{\text{SPACE}}$. On input $\langle M, x, C\rangle$, the program P does the following:

- Use the universal SJava program to simulate M on x for T steps, maintaining the execution state of M in the string `execState`. During this simulation, if at any execution step $t \le T$,
  - the length of string `execState` is more than C, return `false` and halt.
  - M returns an output and halts, then ignore this output and return `true`.
- If M does not return an output on x after T steps, return `true`.

See Figure 3 for a skeleton code for program P.

The correctness of P follows from the following claim: Assume that M does not halt on x in T steps. Further suppose that during the simulation of M on input x for T steps (by the universal SJava program), the length of the string `execState` does not exceed length C. Then the length of the string `execState` never exceeds C in the simulation of M on input x.

To prove the above claim, let's use the notation that $\texttt{execState}_k$ denotes the value of string `execState` after k steps. The observation is the following: if for all $0 \le k \le T$, the length of each the string $\texttt{execState}_k$, is at most C, then there are distinct steps $0 \le i < j \le T$ such that the string $\texttt{execState}_i$ = $\texttt{execState}_j$. This can be proved as follows: note that the number of strings of length at most C using alphabet Σ is bounded by $(\ell+1)^C = T$. Now, by pigeonhole principle, it follows that if there are $T+1$ strings of length at most C, then two strings must be identical. Now applying this on the $T+1$ strings $\texttt{execState}_k$, $k = 0, \ldots, T$.

Using the above observation, it follows that the program M on input x has the same execution state at simulation step i and simulation step j. Thus, M must be have entered an infinite loop on input x, and will keep looping through execution states $\texttt{execState}_i$, $\texttt{execState}_{i+1}$,. . ., $\texttt{execState}_{j-1}$. SInce each of these strings is of length at most C, it proves our claim that the string `execState` never exceeds C in the simulation of M on input x.

Given the above claim, correctness of P is now direct. If M indeed halts on x before T steps, P checks whether the string `execState` ever exceeds length C during the simulation. In this case, P outputs `false` if `execState` exceeds length C at some point in the execution; else it outputs `true`. If In the case, M does not halt on x after T steps, but at some point during the simulation, the string `execState` ever exceeds length C, then P returns `false`. Finally in the case that M does not halt on x after T steps but the string `execState` never exceeds length C during this simulation, from the above claim it follows that `execState` will never exceed length C at a later point in the simulation. Thus, in this case, P outputs `true`.

```
bool P(string M, string w, int C) {
  execState = initialExecState(M,w);

   ell  = # integer constant representing alphabet size

 // Now compute T=(ell+1)^C...
 base = ell + 1;
 T = 1;
 i = 0;
 while (i < C) {
   i = i + 1;
   T = base * T;
 }

 i = 0;
 while (i < T) {
   execState= singleStep(M,w);
   if (length(execState) > C) {
     return false;
               }
   done = testIfHalted(execState);
               if (done) {
     return true;
   }
   i = i + 1;
 }
 return true;
}


 #
 # code for interpreter goes here.
 #
```

Figure 3: Skeleton code for P