Lesson 1: Solving Problems with Algorithms

Creation Emerging from the Collapse of Wholeness to a Point

Wholeness of the Lesson

Much of the power of modern software arises from the fact that it makes use of highly efficient algorithmic solutions to a variety of problems. These solutions represent the concentrated intelligence of great thinkers of the past half-century. Yet the range of problems for which such solutions are known is actually only a tiny speck in the landscape of abstract problems. Likewise, the full range of Nature's computational power is too vast to be grasped by the intellect alone, yet harnessing even a little of that power is enough to produce powerful results.

Algorithms As Concentrated Intelligence

- An algorithm is a procedure or sequence of steps for computing outputs from given inputs
- Algorithms can be implemented in programming languages like Java and C
- We will study techniques for creating algorithms, measuring their efficiency, and refining their performance
- In this lesson, we show that, as remarkable as the many feasible algorithms that have been found are, and as deep as the intelligence is that is behind such solutions, the truth is that the vast majority of "problems" that could be solved remains – and will always remain – completely beyond the reach of any kind of solution that could in principle be implemented on a computer.

Examples of Problems

1. (*GCD Problem*) Given two positive integers m, n, is there a positive integer d that is a factor of both m and n and that is bigger than or equal to every integer d' that is also a factor of m and n?
2. (*Sorting Problem*) Given a list of integers, is there a rearrangement of these integers in which they occur in ascending order?
3. (*Subset Sum Problem*) Given a set S of positive integers and a nonnegative integer k, is there a subset T of S so that the sum of the integers in T equals k?
4. (*n x n Chess Problem*) Given an arbitrary position of a generalized chess-game on an n × n chessboard, can White (Black) win from that position?

5. (*Halting Problem*) Given a Java program R and an input integer n, when R is executed with input n, does R terminate normally? (No runtime exceptions, no infinite loops.)

Solving P Problems

1. (GCD Problem) The Euclidean Algorithm computes the gcd of two positive integers m <= n. It requires fewer than n steps to output a result. We say it *runs in O(n) time*.
2. (SORTING Problem) MergeSort is an algorithm that accepts as input an array of n integers and outputs an array consisting of the same array elements, but in sorted order. It requires fewer than n^2 steps to output a result. We say it runs in $O(n^2)$ time.

Both Problems 1 and 2 have algorithms that run in *polynomial time;* this means that the number of steps of processing required is a polynomial function of the input size.

If a problem can be solved using an algorithm that runs in polynomial time, the problem is said to be a *P Problem.*

P Problems are said to have *feasible solutions* because an algorithm that runs in polynomial time can output its results in an acceptable length of time, typically.

NP and EXP Problems

3. (SUBSETSUM Problem) Every known algorithm to solve SubsetSum (with input array containing n elements) requires approximately 2^n steps of processing ("exponential time") to output a result (in the worst case). However, it is still possible that someone will one day find an algorithm that solves SubsetSum in polynomial time.
4. (N X N CHESS Problem) Like SubsetSum, every known algorithmic solution requires exponential time in the worst case. However, it has also been shown that no algorithmic solution could *ever* be found that runs in polynomial time.

- Both SubsetSum and n x n Chess are problems that belong to **EXP**. This means that each has an exponential time algorithm that solves it. The possibility remains that SubsetSum may also belong to the smaller class P, but this question remains unsolved. Read about the class **EXP** at http://en.wikipedia.org/wiki/EXPTIME
- n x n Chess is a special type of problem in **EXP**, known as **EXP-complete**. For this lecture, it suffices to say that **EXP**-complete means that there is no way to devise an algorithm that solves it in polynomial time. See

  http://www.ms.mff.cuni.cz/~truno7am/slozitostHer/chessExptime.pdf

When the only known solutions to a problem are exponential, the problem is said to *have no known feasible solution.*

NP and EXP Problems (continued)

- The SubsetSum Problem is also an NP Problem: This means that there is an algorithm A and an integer k so that, given a solution T to a SubsetSum problem instance with input size n, A can verify that T is indeed a correct solution and does so in fewer than n^k steps (i.e. in polynomial time)
- Sample Verification:

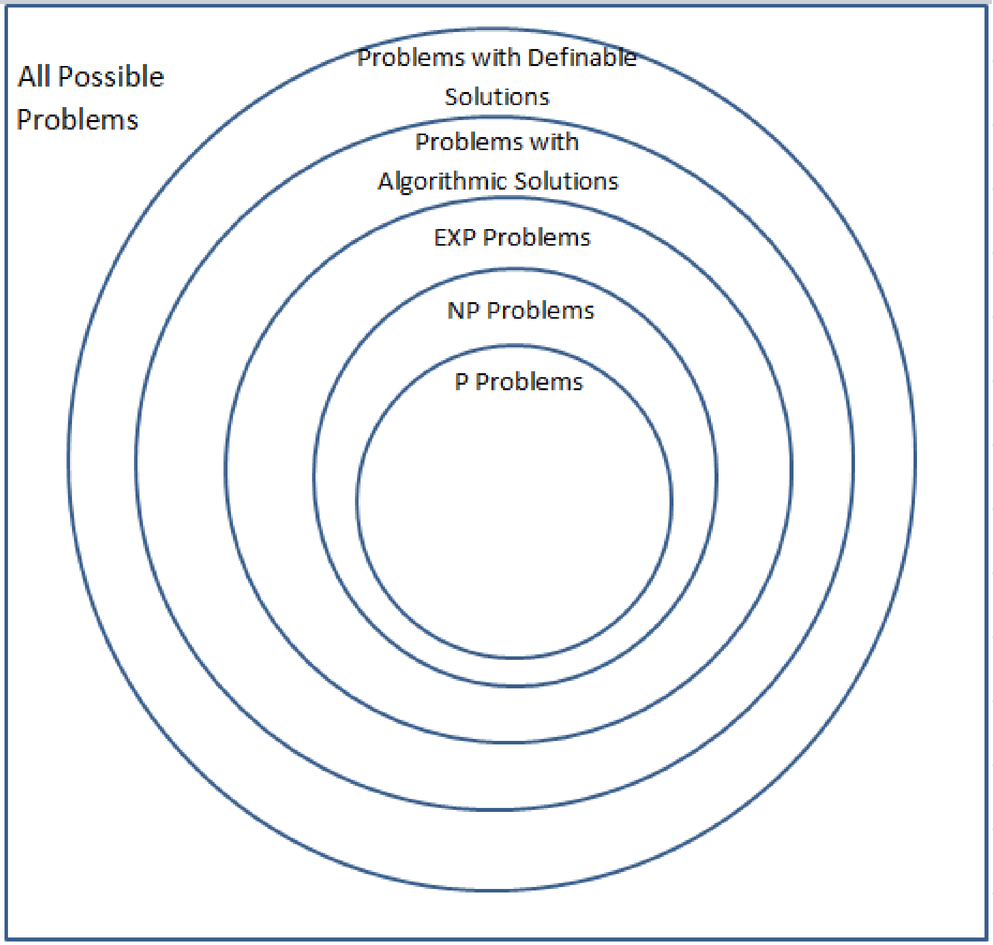
  Start with a SubsetSum problem instance $S = \{s_1, s_2, . . ., s_n\}$ and k, where $s_1, \ldots, s_n$ and k are all positive integers. Given also a solution T: A solution is a subset of S whose elements add up to k.

  Verification:

  ```
  verify all elements in T belong to S;
  sum = 0;
  for each j in T
      sum += j
  if(sum == k) return true;
  else, return false;
  ```

- n x n Chess is not known to belong to NP (but one day someone may prove that it does belong to NP). In general, every NP problem belongs to EXP, but whether NP = EXP is not known. See `http://en.wikipedia.org/wiki/EXPTIME`

Classes of Problems


All Possible Problems
Problems with Definable Solutions
Problems with Algorithmic Solutions
EXP Problems
NP Problems
P Problems

"Unsolvability" of the HALTING Problem

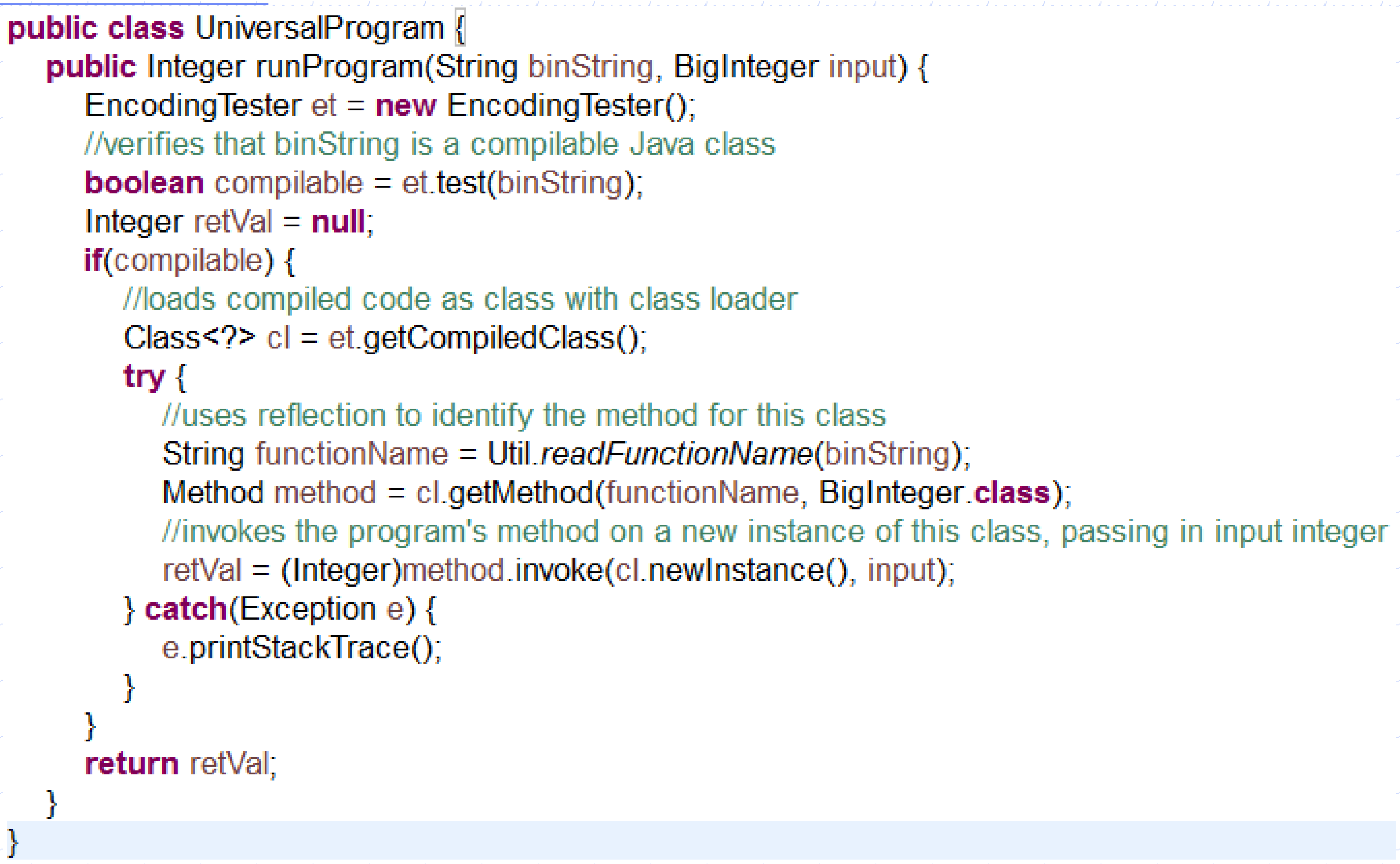
The Problem: Is there a Java program which accepts as input a normal Java program R and an integer n (which we represent as a BigInteger), and which outputs 1 (or "true") if R terminates normally when run on n, or else outputs 0 ("false") if R does not terminate normally.
[A "normal" Java program is one that has a public method that accepts a BigInteger argument and returns an Integer value]

Theorem: *The Halting Problem is unsolvable.*

Assumption: This is a theorem from Theory of Computation – to make sense of it, we must assume that our Java programs are running on an *idealilzed computer* that has no memory limits. The limitation on computation described here has nothing to do with the physical limitations of the computer hardware currently in use.

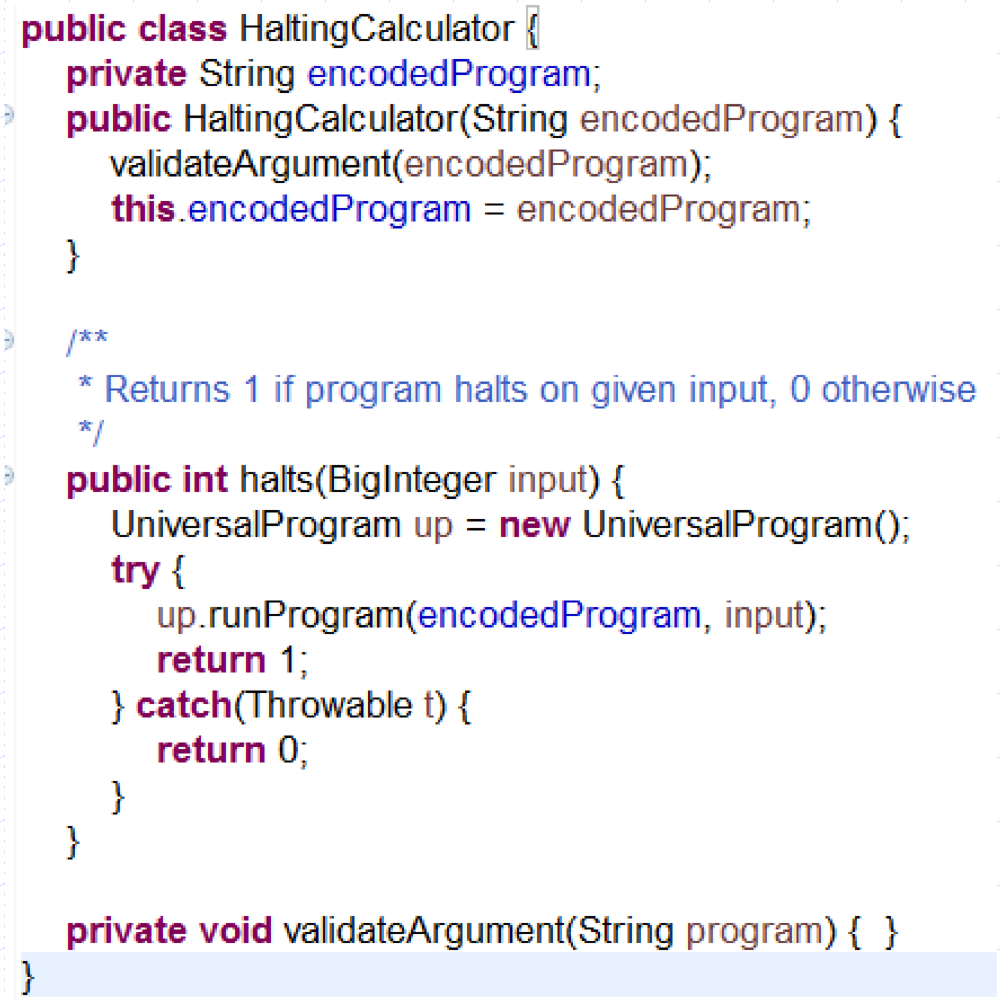
Let’s Try to Solve the Halting Problem

Let’s Try to Solve the Halting Problem



```java
public class UniversalProgram {
    public Integer runProgram(String binString, BigInteger input) {
        EncodingTester et = new EncodingTester();
        //verifies that binString is a compilable Java class
        boolean compilable = et.test(binString);
        Integer retVal = null;
        if(compilable) {
            //loads compiled code as class with class loader
            Class<?> cl = et.getCompiledClass();
            try {
                //uses reflection to identify the method for this class
                String functionName = Util.readFunctionName(binString);
                Method method = cl.getMethod(functionName, BigInteger.class);
                //invokes the program's method on a new instance of this class, passing in input integer
                retVal = (Integer)method.invoke(cl.newInstance(), input);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
        return retVal;
    }
}
```

Let’s Try to Solve the Halting Problem



```
public class HaltingCalculator {
    private String encodedProgram;
    public HaltingCalculator(String encodedProgram) {
        validateArgument(encodedProgram);
        this.encodedProgram = encodedProgram;
    }

    /**
     * Returns 1 if program halts on given input, 0 otherwise
     */
    public int halts(BigInteger input) {
        UniversalProgram up = new UniversalProgram();
        try {
            up.runProgram(encodedProgram, input);
            return 1;
        } catch(Throwable t) {
            return 0;
        }
    }

    private void validateArgument(String program) {  }
}
```

Notes

- A normal Java program is a Java class that is declared public, and has a public method that accepts a BigInteger argument and returns an Integer value.
- A universal Java program accepts any normal Java program R, together with a BigInteger n, as inputs, and runs the method of R on n, and returns the value that R's method returns. In our implementation, the method that does this is `runProgram`
- Encoding. Instead of passing an actual program R to `runProgram`, we pass in an *encoding* of R as a binary string `binString` (string of 0s and 1s) – we use extended ASCII codes to convert each character in a given program file to a binary string of length 8. Examples:

  'a' corresponds to 01100001

  'A' corresponds to 01000001

  Encoding (from program to binary) and decoding (from binary to the program as a string) is done in the `halting2` package by the `EncodeDecode` class.

Some ASCII Values

01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01000110	F
01000111	G
01001000	H
01001001	I
01001010	J
01001011	K
01001100	L
01001101	M
01001110	N
01001111	O
01010000	P
01010001	Q

01100001	a
01100010	b
01100011	c
01100100	d
01100101	e
01100110	f
01100111	g
01101000	h
01101001	i
01101010	j
01101011	k
01101100	l
01101101	m
01101110	n
01101111	o
01110000	p
01110001	q

Notes

- Determining whether a binary string encodes a program. A given binary string `binStr` can be tested to see if it encodes a Java program. This test is performed by the `test` method of `EncodingTester`. It checks the following:
  - That the length of `binStr` is a multiple of 8
  - `binStr` is converted to a sequence of characters using the ASCII table – the resulting string `program` then needs to be checked to see if it is really a Java class
  - The Java compiler API is used to run the compiler on `program`
  - If there are no compiler errors, then we conclude that `binStr` does encode a Java program
- Remaining steps in `runProgram`, in the Universal Program. After the input binary string is tested for compilability:
  - the encoded Java class is compiled into a Class object `cl`
  - the class name and method name are extracted
  - an instance `inst` of `cl` is created
  - reflection is used to run the method on `inst` using the `BigInteger` input
  - the result of this method call is then returned.

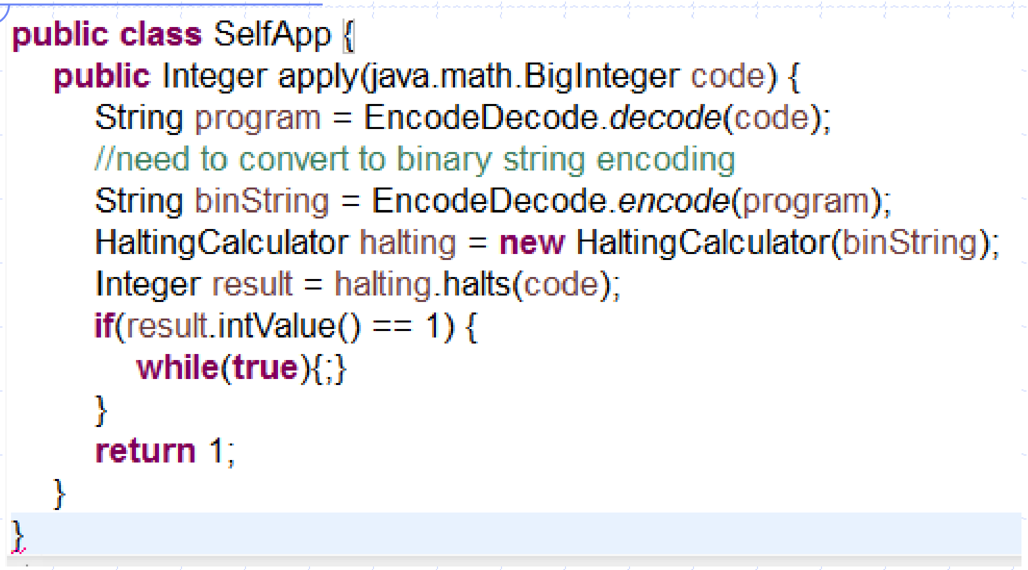
Notes

- HaltingCalculator. The Halting Problem is then "solved" by taking an encoding binString for a normal Java program P and a BigInteger n, asking the Universal Program to run P on input n, and then outputting 1 if runProgram terminates normally, 0 if not.
- Question Does the HaltingCalculator work correctly? Can you think of a normal program P that you could give to it that would cause it to fail?

Proving the Halting Problem is Unsolvable

- The `HaltingCalculator` does not work because it produces no output if the input program goes into an endless loop (but it should output 0).
- This failing, however, does not prove that the Halting Problem is unsolvable (why not?)
- We create another program `SelfApp` that helps us prove that *no version* of the `HaltingCalculator` could ever work.

SelfApp



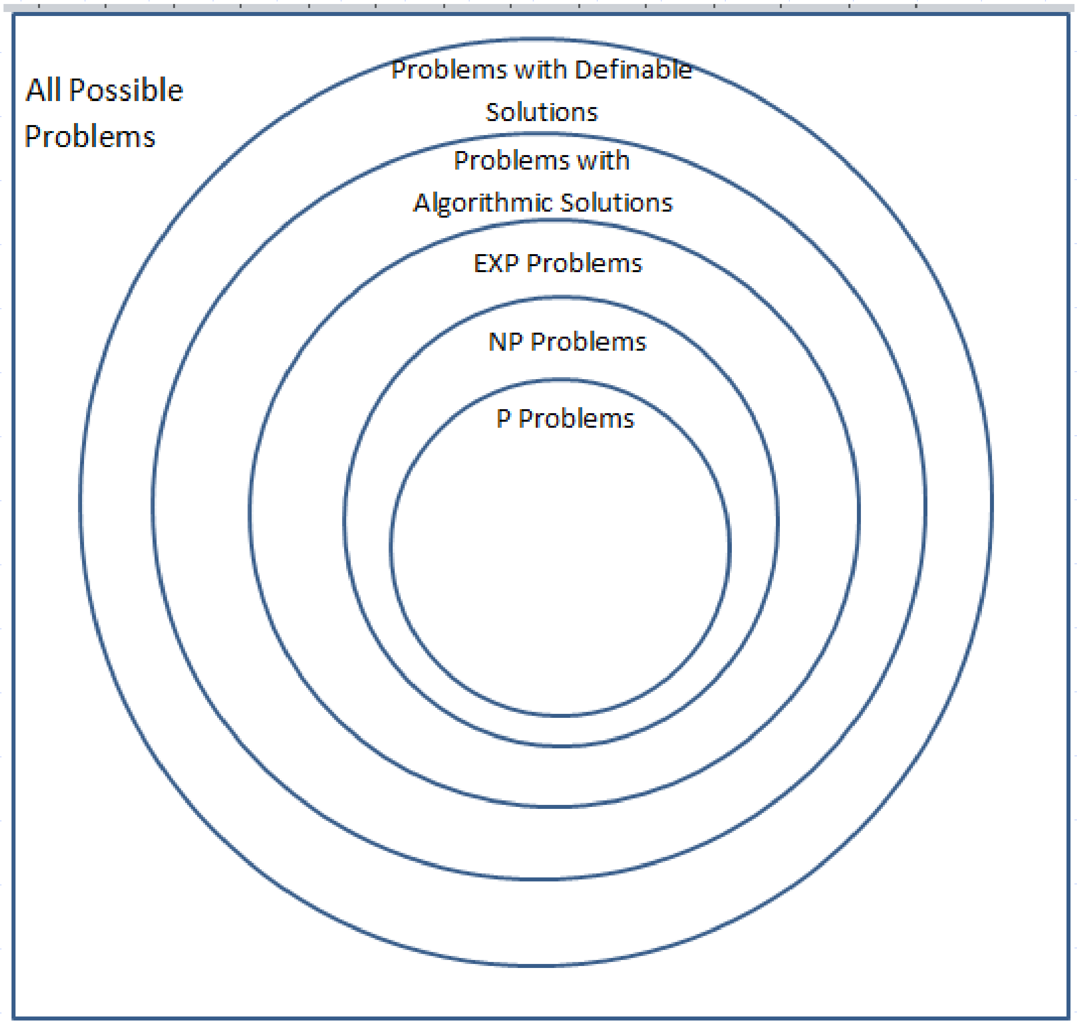
```java
public class SelfApp {
    public Integer apply(java.math.BigInteger code) {
        String program = EncodeDecode.decode(code);
        //need to convert to binary string encoding
        String binString = EncodeDecode.encode(program);
        HaltingCalculator halting = new HaltingCalculator(binString);
        Integer result = halting.halts(code);
        if(result.intValue() == 1) {
            while(true){;}
        }
        return 1;
    }
}
```

<u>Note</u>: Input is a BigInteger, but it is easy to convert between a BigInteger and a binary string, so these can be considered equivalent.

The Logic:

1. Let `binString` encode the same Java program as the input `BigInteger code`
2. On input `code`, the `apply` method:
   - *returns 1* when `halting.halts(binString, code)` returns 0
   - *goes into infinite loop* if `halting.halts(binString, code)` returns 1
3. Suppose `bstr` is the encoding of `SelfApp` (notice it is a normal Java program) and `bi` the corresponding `BigInteger`. Does `SelfApp.apply` halt on input `bi`?
   - If yes, then it returns 1, so that `halting.halts` returns 0 on input `bstr, bi`, which means `SelfApp.apply` does *not* halt on iniput `bi`.
   - If no, then `halting.halts` returns 1 on input `bstr, bi,` , which means `SelfApp.apply` *does* halt on
     input `bi`.
4. Contradiction! Shows *no version* of `HaltingCalculator` could ever work!

Classes of Problems


All Possible Problems
Problems with Definable Solutions
Problems with Algorithmic Solutions
EXP Problems
NP Problems
P Problems

The Full Range of "Problems"

- The Halting Problem has a *definable* solution because there is a definable function H (definable in the standard model of arithmetic) that answers all questions about whether a given Java program terminates on a given input.
- Likewise, the abstract notion of "problem" is related to the concept of a function. Speaking generally, every definable function corresponds to (and is a definable solution for) a "problem".
- More generally, *every* function from f: N -> N (whether or not definable) specifies a problem. But *most* such functions are *not* even definable – therefore, most "problems" cannot even be formulated precisely in the language of mathematics.

Main Point

It has turned out that the problems that are most needed to be solved for the purpose of developing modern-day software projects happen to lie in the very specialized class of P Problems – problems that have feasible solutions. For these problems, the creativity and intelligence of the industry has been concentrated to a point; the algorithms that have been developed for these are extremely fast and highly optimized. At the same time, this tiny point value reveals how vast is the range of problems that cannot be solved in a feasible way. These points illustrate that creative expression arises in the collapse of unboundedness to a point, and also that unboundedness itself is beyond the grasp of the intellect.

Connecting the Parts of Knowledge With The Wholeness of Knowledge

COLLAPSE OF INFINITY TO A POINT; EXPANSION OF POINT TO INFINITY

1. The functions that are used and analyzed in practice in the development and analysis of software are the *polynomial bounded functions.*
2. The polynomial bounded functions form only a tiny speck in the expansive class of all number-theoretic functions N → N.

3. *Transcendental Consciousness* is the fully expanded field of consciousness, beyond even the most general notion of “function”.
4. *Impulses Within The Transcendental Field*. As pure consciousness becomes conscious of itself, it undergoes transformational dynamics, as knower, known, and process of knowing interact. The process of knowing is the first sprout of the notion of “function.”
5. *Wholeness Moving Within Itself.* In Unity Consciousness, the transformational dynamics of consciousness, at the basis of all of creation, are appreciated as the lively impulses of one’ s own consciousness, one’s own being.