Breeding like rabbits
==========

As usual, the zombie rabbits (zombits) are breeding... like rabbits! But instead of following the Fibonacci sequence like all good rabbits do, the zombit population changes according to this bizarre formula, where R(n) is the number of zombits at time n:

R(0) = 1
R(1) = 1
R(2) = 2
R(2n) = R(n) + R(n + 1) + n (for n > 1)
R(2n + 1) = R(n - 1) + R(n) + 1 (for n >= 1)

(At time 2, we realized the difficulty of a breeding program with only one zombit and so added an additional zombit.)

Being bored with the day-to-day duties of a henchman, a bunch of Professor Boolean's minions passed the time by playing a guessing game: when will the zombit population be equal to a certain amount? Then, some clever minion objected that this was too easy, and proposed a slightly different game: when is the last time that the zombit population will be equal to a certain amount? And thus, much fun was had, and much merry was made.

(Not in this story: Professor Boolean later downsizes his operation, and you can guess what happens to these minions.)

Write a function answer(str_S) which, given the base-10 string representation of an integer S, returns the largest n such that R(n) = S. Return the answer as a string in base-10 representation. If there is no such n, return "None". S will be a positive integer no greater than 10^25.

Languages
=========

To provide a Python solution, edit solution.py
To provide a Java solution, edit solution.java

Test cases
==========

Inputs:
    (string) str_S = "7"
Output:
    (string) "4"

Inputs:
   (string) str_S = "100"
Output:
   (string) "None"

Author: Neil VonHoltum
```
package com.google.challenges;
import java.util.HashMap;
import java.math.BigInteger;

public class Answer {

   private static HashMap<BigInteger, BigInteger> population;
   private static BigInteger one, two;

   public static String answer(String str_S) {

      one = BigInteger.ONE;
      two = BigInteger.valueOf(2);
      BigInteger s = new BigInteger(str_S), four = BigInteger.valueOf(4),
            zero = BigInteger.ZERO, time = s.divide(two), distance = s.divide(four);
      population = new HashMap<BigInteger, BigInteger>();
      population.put(zero, one);
      population.put(one, one);
      population.put(two, two);

      //Ensure time is odd.
      if(time.mod(two).equals(zero)){

         time = time.add(one);
      }

      /*
      Psuedo binary search odd times for s and return immediately if found,
      because odd timed populations are always lower than greater even timed populations.
      */
      do{

         BigInteger pop = zpopodd(time);

         if(pop.equals(s)){
```

```
        return time.toString();
    }

    /*
    Ensure distance is even, so that when it's added or subtracted
    from the currently odd time, another odd time results.
    */
    if(distance.mod(two).equals(one)){

        distance = distance.add(one);
    }

    if(pop.compareTo(s) < 0){

        time = time.add(distance);
    }

    else{

        time = time.subtract(distance);
    }

    /*
    Not real binary search, but still works for all the tests.
    Halve our new distance to be added or subtracted from time, thus
    roughly reaching the midpoint of the upper and lower bounds.
    */

    distance = distance.divide(two);

}while(distance.compareTo(one) > 0);

time = s.divide(two);
distance = s.divide(four);

//Ensure time is even.
if(time.mod(two).equals(one)){

    time = time.add(one);
}

//Check even times for s.
do{
```

```
        BigInteger pop = zpopeven(time);

        if(pop.equals(s)){

            return time.toString();
        }

        if(distance.mod(two).equals(one)){

            distance = distance.subtract(one);
        }

        if(pop.compareTo(s) < 0){

            time = time.add(distance);
        }

        else{

            time = time.subtract(distance);
        }

        distance = distance.divide(two);

    }while(distance.compareTo(one) > 0);

    return "None";
}

private static BigInteger zpopodd(BigInteger t){

    if(population.containsKey(t)){

        return population.get(t);
    }

    BigInteger poptoret, n = t.subtract(one).divide(two), nsubone = n.subtract(one);

    if(n.mod(two).equals(one)){

        poptoret = zpopeven(nsubone).add(zpopodd(n));
    }
```

```
    else{

        poptoret = zpopodd(nsubone).add(zpopeven(n));
    }

    /*
    Speed up experiment. When n > 2,
    r((n-1)*2) is poptoret + (n-1) since it requires the same value poptoret prior
    to adding one.
    */
    if(n.compareTo(two) > 0){

        population.put(nsubone.multiply(two), poptoret.add(nsubone));
    }

    poptoret = poptoret.add(one);
    population.put(t, poptoret);
    return poptoret;
}

private static BigInteger zpopeven(BigInteger t){

    if(population.containsKey(t)){

        return population.get(t);
    }

    BigInteger poptoret, n = t.divide(two), naddone = n.add(one);

    if(n.mod(two).equals(one)){

        poptoret = zpopodd(n).add(zpopeven(naddone));
    }

    else{

        poptoret = zpopeven(n).add(zpopodd(naddone));
    }

    /*
    speedup experiment.
    n = n+1 for odd time since they use the same poptoret value prior
```

```
        to adding n. No need for safety check here because n > 0, so n+1 > 1.
        */
        population.put(naddone.multiply(two).add(one), poptoret.add(one));
        poptoret = poptoret.add(n);
        population.put(t, poptoret);
        return poptoret;
    }
}
```