Code Jam 2013 - Round 1A

Analysis: Bullseye

First, we need to know the area of the first black ring. It can be calculated by subtracting the area of a white circle with radius **r** from a black circle with radius **r**+1 cm. That is, the area is $(\mathbf{r}+1)^2\pi-\mathbf{r}^2\pi$ cm^2. Expanding we get $(2\mathbf{r}+1)\pi$ cm^2. Since 1mL of paint can cover area π cm^2, we need exactly 2**r**+1 mL of paint to draw the first black ring. For the second black ring, we do the same: subtracting the area of a white circle with radius **r**+2 cm from a black circle with radius **r**+3 cm, thus we need 2**r**+5 mL to draw. In general, we need exactly $(\mathbf{r}+2\mathbf{k}-1)^2-(\mathbf{r}+2\mathbf{k}-2)^2 = 2\mathbf{r}+4\mathbf{k}-3$ mL of paint to print the **k**-th black ring.

For small we know that the answer is less than **t** = 1000 anyway, so that we can try adding black rings one by one until the total amount of paint used including the next black ring exceeds **t**, then we stop adding and output the number of black rings we have included so far.

However, this does not work well with the large input as the answer can be much bigger! This can be verified by the fourth sample (we intended to be kind to contestants, but turns out many still failed the large due to integer overflow :-(). How can we improve the algorithm?

The key is the following: if you can paint at most **k** black rings using **t** mL of paint, for sure you can also use it to draw 1 black ring, 2 black rings, ... up to **k**-1 black rings. So instead of searching for the answers linearly, we can perform binary search on "Is it possible to use **t** mL of paint to draw **k** black rings?" to find the answer more efficiently. Now the remaining question is how much paint is used to draw **k** black rings.

Looking back about the total amount of paint to draw individual black rings, it is easily observed that they form an arithmetic progression: 2**r**+1, 2**r**+5, 2**r**+9, ... 2**r**+4**k**-3. Then the total amount is simply the sum of the progression, which is $(2\mathbf{r}+1+2\mathbf{r}+4\mathbf{k}-3)\times\mathbf{k}\div2 = (2\mathbf{r}+2\mathbf{k}-1)\mathbf{k}$ mL. This is sufficient to completely solve the problem.

Here is a complete solution in Python for reference:

```
num_cases = int(raw_input())
for casenum in range(1, num_cases+1):
  r, t = [int(z) for z in raw_input().split()]
  res, lo, hi = 0, 1, t
  while lo <= hi:
    mid = (lo + hi) / 2
    if mid * (2 * r + 2 * mid - 1) > t:
      hi = mid - 1
    else:
      lo, res = mid + 1, mid
  print "Case #%d: %d" % (casenum, res)
```

You might think that you need big integer libraries to solve the large input, in fact you don't. For example, double precision floating point numbers suffice to check the condition correctly. A more elegant way is to find the first range with **exponential growth** in length so that there exists a value in the range which fails to satisfy the condition. It takes logarithmic time to find the required range. Then we perform binary search on it and compute the answer. Here is a C++ snippet that describes the algorithm without using any big integer library.

```
// Check if we can draw k black rings.
bool Check(long long r, long long t, long long k) {
  return 2 * k * k + (2 * r - 1) * k <= t;
```

```
}

// Find the maximum number of black rings that can be drawn.
long long Solve(long long r, long long t) {
  long long left = 0, right = 1;
  // Find the range that the answer lies in.
  while(Check(r,t,right)) {
    left = right;
    right *= 2;
  }

  // Binary search on the range [left, right) for the answer.
  while(right - left > 1) {
    long long k = (left + right) / 2;
    if (Check(r, t , k))
      left = k;
    else
      right = k;
  }
  return left;
}
```