Project Euler #207: Integer Partition | HackerRank challenges

For some positive integers *k*, there exists an integer partition of the form $4^t = 2^t + k$, where 4^t, 2^t and *k* are all positive integers and *t* is a real number.

The first two such partitions are $4^1 = 2^1 + 2$ and $4^{1.58496...} = 2^{1.58496...} + 6$.

Partitions where *t* is also an integer are called *perfect*.
For any m > 1 let *P*(*m*) be the proportion of such partitions that are perfect with $k \le m$.
Thus *P*(6) = 1/2.

In the following table are listed some values of

P(5) = 1/1
P(10) = 1/2
P(15) = 2/3
P(20) = 1/2
P(25) = 1/2
P(30) = 2/5
...
P(180) = 1/4
P(185) = 3/13

Find the smallest *m* for which *P*(*m*) < a/b.

Input Format

First line of each test file contains a single integer *q* that is the number of queries per test file. *q* lines follow, with two integers *a* and *b* separated by a single space on each.

Constraints

- $1 \le q \le 3 \cdot 10^5$
- $1 \le a < b \le 10^{18}$

Output Format

Print exactly *q* lines with an answer for the corresponding query on each.

Sample Input 0

```
2
2 3
9 20
```

Sample Output 0

```
6
30
```

Explanation 0

P(2) = *P*(3) = *P*(4) = *P*(5) = 1/1 > 2/3, but *P*(6) = 1/2 < 2/3, therefore, an answer for the first query is 6.
P(30) = 2/5, which is the first value less than 9/20 among all *P*(m) where 1 < m ≤ 30.

Solution

Algebraic analysis

$4^t = 2^t + k \Rightarrow 4^t - 2^t - k = 0 \Rightarrow (2^t)^2 - 2^t - k = 0 \overset{2^t=x}{\Longrightarrow} x^2 - x - k = 0 \Rightarrow x = \dfrac{1 \pm \sqrt{1+4k}}{2} \overset{x=2^t>0}{\Longrightarrow}$

$\Rightarrow \boxed{x = \dfrac{1+\sqrt{1+4k}}{2}}\ (1)$

For $2^t \equiv x$ to be an integer, the above square root must be an odd integer. So, if we define

$\boxed{sqroot = \sqrt{1+4k}}\ (2)$, the integer partitions occur for $sqroot = 1, 3, 5, \dots$. For the shake of clarity, any integer (at the problem is represented with m) at the position of k can result to a partition which is not necessarily an integer partition (2^t is not necessarily an integer), but k numbers exclusively result to integer partitions.

Every time 2^t is incremented by one (meaning a next integer partition occurs), the sum of partitions is incremented by one, too. So, a partition counter, k_counter, progresses the same as 2^t. But, as stated at the description of the problem, 1st partition occurs for $t = 1$, thus 1st count for the k_counter happens when $2^t = 2^1 = 2$, so a minus 1 is needed to calibrate k_counter. Consequently:

$$\boxed{k\_counter = 2^t - 1} \overset{(1),(2)}{\Longrightarrow} \boxed{k\_counter = \frac{1+sqroot}{2} - 1}\ (3)$$

Calculating t:

$$2^t = x = \frac{1+\sqrt{1+4k}}{2} \overset{(2)}{\Rightarrow} \boxed{t = \log_2(1+sqroot)}\ (4)$$

Now, for a perfect partition to occur, t has to be an integer. Therefore, 1 + sqroot (this sum is always even) has to be a power of 2 (or a point which the order of magnitude of the binary system changes). So, likewise, every time t is incremented be one, the sum of perfect partitions, perf_counter, is incremented by one, too. Therefore:

$$\boxed{perf\_counter = \log_2(1+sqroot) - 1}\ (5)$$

To sum up, the variables of the problem are:

- $\boldsymbol{sqroot} = \sqrt{1+4k} \overset{(3)}{=} 2(k\_counter\ + 1) - 1 \overset{(4)}{=} 2^{perf\_counter+1} - 1$
- $\boldsymbol{k} = \frac{sqroot^2-1}{4} \overset{(3)}{=} k\_counter(k\_counter + 1)$
- $\boldsymbol{t} = \log_2(1+sqroot)$
- $\boldsymbol{k\_counter} = \frac{sqroot+1}{2} - 1$
- $\boldsymbol{perf\_counter}\ = \log_2(1+sqroot) - 1 = t - 1$
- $\boldsymbol{P(k)} = \frac{perf\_counter}{k\_counter}$
- the in query input, which is stated as $(\boldsymbol{double})\boldsymbol{a}/\boldsymbol{b}$

The following table demonstrates the progress of the variables for the first 20 perfect partitions.

perf_counter	k_counter	P	P//N	sqroot	prev P	prev P	k
1	1	1/1	1	3			2
2	3	2/3	0.666666667	7	1/2	0.500000000	12
3	7	3/7	0.428571429	15	2/6	0.333333333	56
4	15	4/15	0.266666667	31	3/14	0.214285714	240
5	31	5/31	0.161290323	63	4/30	0.133333333	992
6	63	6/63	0.095238095	127	5/62	0.080645161	4032
7	127	7/127	0.055118110	255	6/126	0.047619048	16256
8	255	8/255	0.031372549	511	7/254	0.027559055	65280
9	511	9/511	0.017612524	1023	8/510	0.015686275	261632
10	1023	10/1023	0.009775171	2047	9/1022	0.008806262	1047552
11	2047	11/2047	0.005373718	4095	10/2048	0.004887586	4192256
12	4095	12/4095	0.002930403	8191	11/4094	0.002686859	16773120
13	8191	13/8191	0.001587108	16383	12/8190	0.001465201	67100672
14	16383	14/16383	0.000854544	32767	13/16382	0.000793554	268419072
15	32767	15/32767	0.000457778	65535	14/32766	0.000427272	1073709056
16	65535	16/65535	0.000244144	131071	15/65534	0.000228889	4294901760
17	131071	17/131071	0.000129701	262143	16/131070	0.000122072	17179738112
18	262143	18/262143	0.000068665	524287	17/262142	0.000064850	68719214592
19	524287	19/524287	0.000036240	1048575	18/524286	0.000034332	2.74877E+11
20	1048575	20/1048575	0.000019074	2097151	19/1048574	0.000018120	1.09951E+12

Code strategy

There are 3 programs presented. The 1st is a super simple but super slow loop, the 2nd is a significant faster and interesting Linked List implementation and the 3rd is a fast and slightly more complex solution, which passes the 2sec timeout mark at all tests.

Having a quick peek at the above table, it is clear that a fast solution should loop through perf_counter, which evolves in a logarithmic manner. This will be the 3rd implementation, but let's keep it simple at first.

1st implementation

The program loops through sqroot, incrementing by two at each step, which is the same as looping through the consequent partitions (or k_counter counts), and checks if there is a perfect partition (whether t is an integer). Finally, at every step program checks if the fraction perf_counter/k_counter is less than the in query portion.

You can refer to "Project_Euler_207-Integer_Partition-super_simple.cpp" at the src directory.

2nd implementation

The basic notion is to save any found partitions, so as to search through them first when working with a new query, without recalculating them, resulting to a faster code. If the solution isn't found, the program will continue iterating through the square roots, starting from the last one listed, while listing new partitions. This can be implemented with other stl data structures , too, like queue or vector.

See "Project_Euler_207-Integer_Partition-linkedList.cpp" at the src directory.

3rd implementation

As mentioned above, the fastest solution is to loop through perfect partitions, namely, through perf_counter. Notice that the fractions that correspond to perfect partitions are sorted descending, thus every such fraction is the representative of the set that extends to the previous "perfect" fraction. So, while iterating through perf_counter, the answer will lie at the first set whom the representative fraction is lesser than the query. In other words, instead of iterating through every single partition, the search body is divided in a logarithmic manner, and the program iterates through the fractions that represent those regions. To be fully correct, the real representative is not the "perfect fraction", but the immediately previous fraction (prev P at the table):

(double)(perf_counter-1) / (perf_counter_to_k_counter(perf_counter)-1)

because the -1 numerator is more significant from the -1 denominator, in that proportion. So, in reality the program iterates through the immediately previous fractions.

Every such set is defined by fractions with constant numerator (perf_counter - 1) and descending denominator (k_counter). Thus, to find the fraction that is smaller than the query, we can iterate in the set. But, there is a faster method. While the numerator is fixed we can find the almost equivalent to the query fraction:

$$denominator = \frac{perf\_counter - 1}{query}$$

or denominator ≡ k_counter = ceil((double)(perf_counter-1) / ((double)a/b))

This leeds to 3 options:

1. if $\frac{perf\_counter-1}{denominator} < query$ → print the k that corresonds to this k_counter
2. if $\frac{perf\_counter-1}{denominator} = query$ → print the k that corresonds to the next k_counter
3. if denominator lies on perf partition → print the k that corresonds to this k_counter + 2

At last, notice that the printed variable k is the one that scales up the fastest (k increases with the square of k_counter) and, because of the constraints of the problem, it even overflows the 64bit integer type. Therefore, an 128bit integer must be printed, a type which is not supported inherently by C++, thus an implicit method is implemented as follows:

The equation is:

$$\boxed{k = k\_counter(k\_counter + 1)}$$

As an example, let k_counter be 12121 and the divider 100.

1. Just before overflow occurs (perf_counter > 32) k_counter is partitioned in two segments, leading and trailing, which are the quotient ($k\_counter$ / divider) and the remainder($k\_counter$ % devider), respectively, as we divide with a suitable constant (divider) – green 121 and red 21, respectively.
2. Next, we multiply each with (k_counter + 1)
3. Then, we partition the same way the trailing – red 62 in the box is the trailing of trailing*k_counter – and this will be the trailing of the output.
4. To form the leading of the output, we add leading*k_counter (1466762) with the leading of trailing*k_counter (2545).

5. Final output is the printf of leading (1469307) + printf of trailing (62).

$$
\begin{array}{r}
12122 \\
\times\, 12121 \\
\hline
12122 \\
+\,24244 \\
\hline
2545\boxed{62} \\
12122 \\
24244 \\
+\,12122 \\
\hline
1466762
\end{array}
$$

$$
\begin{array}{rr}
 & 2545 \\
 & +\,1466762 \\
\cline{2-2}
\text{return} \longrightarrow & 1469307\boxed{62}
\end{array}
$$

See “Project_Euler_207-Integer_Partition.cpp” at the src directory.

Thanasis Mattas, 2019
atmattas@physics.auth.gr